Лекция 3. Ветвление в Git

Ветвление в Git

Почти каждая СУВ имеет в какой-то форме поддержку ветвления. Ветвление означает, что вы отклоняетесь от основной линии разработки и продолжаете работу, не вмешиваясь в основную линию. Во многих СУВ это в некотором роде дорогостоящий процесс, зачастую требующий от вас создания новой копии каталога с исходным кодом, что может занять продолжительное время для больших проектов.

Некоторые говорят, что модель ветвления в Git это его “killer feature“ и она безусловно выделяет Git в СУВ-сообществе. Что же в ней такого особенного? Способ ветвления в Git чрезвычайно легковесен, что делает операции ветвления практически мгновенными и переключение туда-сюда между ветками обычно так же быстрым. В отличие от многих других СУВ, Git поощряет процесс работы, при котором ветвление и слияние осуществляется часто, даже по несколько раз в день. Понимание и владение этой функциональностью даёт вам уникальный мощный инструмент и может буквально изменить то, как вы ведёте разработку.

3.1 Что такое ветка?

Чтобы на самом деле разобраться в том, как Git работает с ветками, мы должны сделать шаг назад и рассмотреть, как Git хранит свои данные. Как вы, наверное, помните из Главы 1, Git хранит данные не как последовательность изменений или дельт, а как последовательность снимков состояния (snapshot).

Когда вы фиксируете изменения в Git, Git сохраняет фиксируемый объект, который содержит указатель на снимок содержимого индекса, метаданные автора и комментария и ноль или больше указателей на коммиты, которые были прямыми предками этого коммита: ноль предков для первого коммита, один — для обычного коммита и несколько — для коммита, полученного в результате слияния двух или более веток.

Для наглядности давайте предположим, что у вас есть каталог, содержащий три файла, и вы их все индексируете и делаете коммит. При подготовке файлов для каждого из них вычисляется контрольная сумма (SHA-1 хеш мы упоминали в Главе 1), затем эти версии файлов

сохраняются в Git-репозиторий (Git обращается к ним как к двоичным данным), а их контрольные суммы добавляются в индекс:

```
$ git add README test.rb LICENSE

$ git commit -m 'initial commit of my project'
```

Когда вы создаёте коммит, выполняя git commit, Git вычисляет контрольную сумму каждого подкаталога (в нашем случае только корневого каталога) и сохраняет объекты для этого дерева в Git-репозиторий. Затем Git создаёт объект для коммита, который имеет метаданные и указатель на корень проектного дерева. Таким образом, Git может воссоздать текущее состояние, когда нужно.

Ваш Git-репозиторий теперь содержит пять объектов: по одному массиву двоичных данных для содержимого каждого из трёх файлов, одно дерево, которое перечисляет содержимое каталога и определяет соответствие имён файлов и массивов двоичных данных, и один коммит с указателем на корень этого дерева и все метаданные коммита. Схематично данные в вашем Git-репозитории выглядят, как показано на Рисунке 3-1.

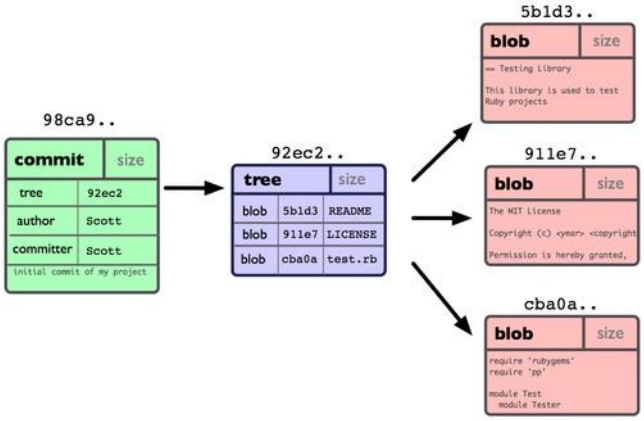

Рисунок 3.1: Данные репозитория с единственным коммитом.

Если вы сделаете некоторые изменения и зафиксируете их, следующий коммит сохранит указатель на коммит, который шёл непосредственно перед ним. После еще двух коммитов ваша история может выглядеть, как показано на Рисунке 3-2.

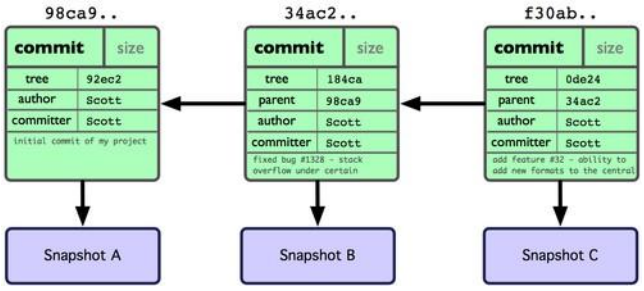

Рисунок 3.2: *Данные объектов Git в случае нескольких коммитов.*

Ветка в Git — это просто легковесный подвижный указатель на один из этих коммитов. Имя ветки по умолчанию в Git — master. Когда вы вначале создаёте коммиты, вам даётся ветка master, указывающая на последний сделанный коммит. При каждом новом коммите указатель сдвигается вперёд автоматически.

Что происходит, когда вы создаёте новую ветку? Итак, этим вы создаёте новый указатель, который вы можете перемещать. Скажем, вы создаёте новую ветку под названием testing. Это делается командой git branch:

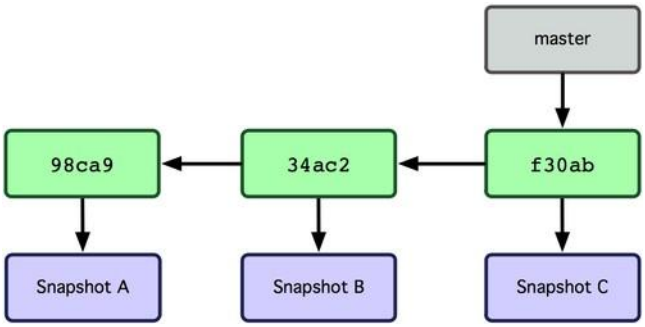

Рисунок 3.3: Ветка указывает на историю коммитов.

```
$ git branch testing
```

Эта команда создает новый указатель на тот самый коммит, на котором вы сейчас находитесь (см. Рисунок 3-4).

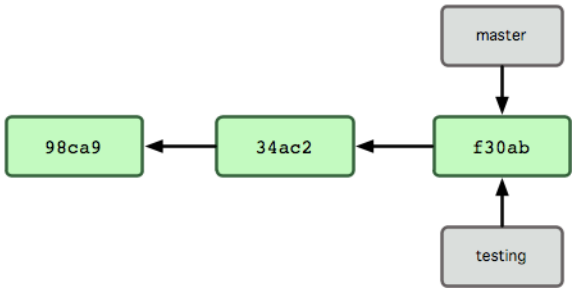

Рисунок 3.4: Несколько веток, указывающих на историю коммитов.

Откуда Git узнает, на какой ветке вы находитесь в данный момент? Он хранит специальный указатель, который называется HEAD (верхушка). Учтите, что это сильно отличается от концепции HEAD в других СУВ, таких как Subversion или CVS, к которым вы, возможно, привыкли. В

Git это указатель на локальную ветку, на которой вы находитесь. В данном случае вы всё ещё на ветке master. Команда git branch только создала новую ветку, она не переключила вас на неё (см. Рисунок 3-5).

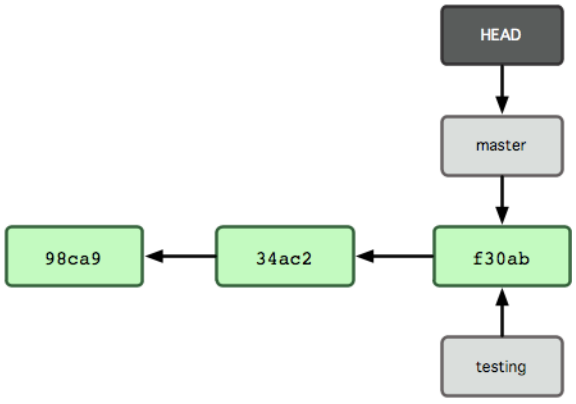

Рисунок 3.5: Файл HEAD указывает на текущую ветку.

Чтобы перейти на существующую ветку, вам надо выполнить команду git checkout.

Давайте перейдем на новую ветку testing:

```
$ git checkout testing
```

3-6).

Это действие перемещает HEAD так, чтобы тот указывал на ветку testing (см. Рисунок

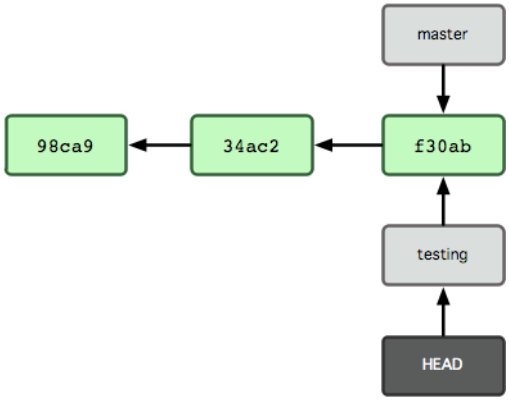

Рисунок 3.6: HEAD указывает на другую ветку, когда вы их переключаете.

В чём важность этого действия? Давайте сделаем ещё один коммит:

```
$ vim test.rb

$ git commit -a -m 'made a change'
```

На Рисунке 3-7 показан результат.

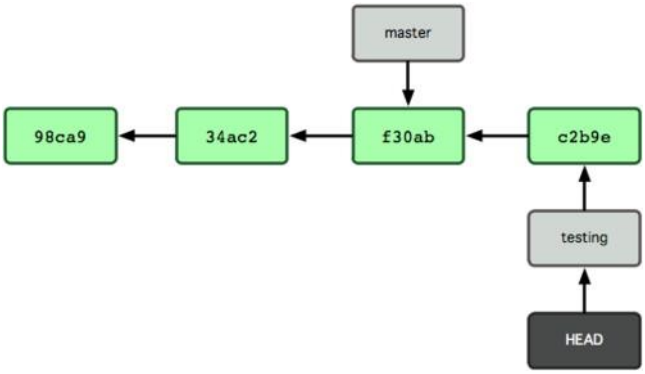

Рисунок 3.7: Ветка, на которую указывает HEAD, движется вперёд с каждым коммитом.

Это интересно, потому что теперь ваша ветка testing передвинулась вперёд, но ваша ветка master всё ещё указывает на коммит, на котором вы были, когда выполняли git check-out, чтобы переключить ветки. Давайте перейдём обратно на ветку master:

```
$ git checkout master
```

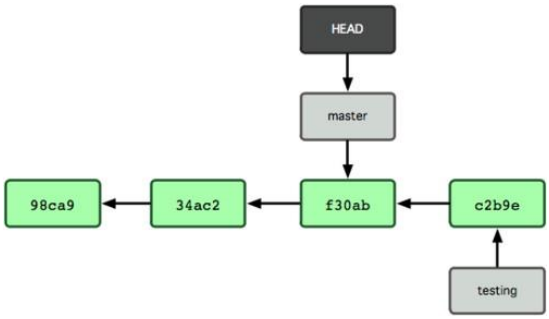

Рисунок 3.8: HEAD перемещается на другую ветку при checkout’e.

На Рисунке 3-8 можно увидеть результат.

Эта команда выполнила два действия. Она передвинула указатель HEAD назад на ветку master и вернула файлы в вашем рабочем каталоге назад, в соответствие со снимком состояния, на который указывает master. Это также означает, что изменения, которые вы делаете, начиная с этого момента, будут ответвляться от старой версии проекта. Это полностью откатывает изменения, которые вы временно делали на ветке testing. Таким образом, дальше вы можете двигаться в другом направлении.

Давайте снова сделаем немного изменений и зафиксируем их:

```
$ vim test.rb

$ git commit -a -m 'made other changes'
```

Теперь история вашего проекта разветвилась (см. Рисунок 3-9). Вы создали новую ветку, перешли на неё, поработали на ней немного, переключились обратно на основную ветку и выполнили другую работу. Оба эти изменения изолированы на отдельных ветках: вы можете переключаться туда и обратно между ветками и слить их, когда будете готовы. И вы сделали всё это простыми командами branch и checkout.

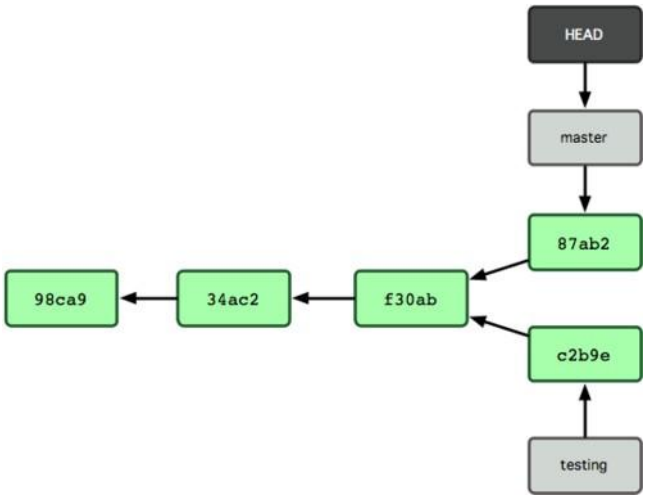

Рисунок 3.9: История с разошедшимися ветками.

Из-за того, что ветка в Git на самом деле является простым файлом, который содержит 40 символов контрольной суммы SHA-1 коммита, на который он указывает, создание и удаление веток практически беззатратно. Создание новой ветки настолько же

быстро и просто, как запись 41 байта в файл (40 символов + символ перехода на новую строку).

Это разительно отличается от того, как в большинстве СУВ делается ветвление. Там это приводит к копированию всех файлов проекта в другой каталог. Это может занять несколько секунд или даже минут, в зависимости от размера проекта, тогда как в Git этот процесс всегда моментален. Также благодаря тому, что мы запоминаем предков для каждого коммита, поиск нужной базовой версии для слияния уже автоматически выполнен за нас, и в общем случае слияние делается легко. Эти особенности помогают поощрять разработчиков к частому созданию и использованию веток.

Давайте поймём, почему вам стоит так делать.

Основы ветвления и слияния

Давайте рассмотрим простой пример ветвления и слияния с таким процессом работы, который вы могли бы использовать в настоящей разработке. Вы будете делать следующее:

1. Работать над веб-сайтом.
2. Создадите ветку для новой задачи, над которой вы работаете.
3. Выполните некоторую работу на этой ветке.

На этом этапе вы получите звонок о том, что сейчас критична другая проблема, и её надо срочно решить. Вы сделаете следующее:

1. Вернётесь на производственную ветку.
2. Создадите ветку для исправления ошибки.
3. После тестирования ветки с исправлением сольёте её обратно и отправите в продакшн.
4. Вернётесь к своей исходной задаче и продолжите работать над ней.

Основы ветвления

Для начала представим, что вы работаете над своим проектом и уже имеете пару коммитов (см. Рисунок 3-10).

Вы решили, что вы будете работать над проблемой №53 из системы отслеживания ошибок, используемой вашей компанией. Разумеется, Git не привязан к какой-то определенной системе отслеживания ошибок. Просто из-за того, что проблема №53 является основной задачей, над которой вы хотите работать, вы создадите новую ветку для работы в

ней. Чтобы создать ветку и сразу же перейти на неё, вы можете выполнить команду git checkout с ключом -b:

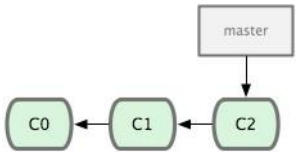

Рисунок 3.10: Короткая и простая история коммитов.

```
$ git checkout -b iss53

Switched to a new branch "iss53"
```

Это сокращение для:

```
$ git branch iss53

$ git checkout iss53
```

Рисунок 3-11 показывает результат.

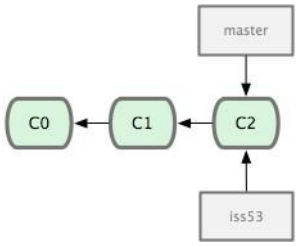

Рисунок 3.11: Создание новой ветки / указателя.

Во время работы над вашим веб-сайтом вы делаете несколько коммитов. Эти действия сдвигают ветку iss53 вперёд, потому что вы на неё перешли (то есть ваш HEAD указывает на неё; см. Рисунок 3-12):

```
$ vim index.html

$ git commit -a -m 'added a new footer [issue 53]'
```

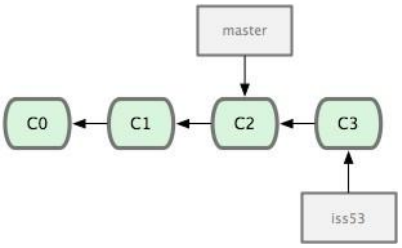

Рисунок 3.12: Ветка iss53 передвинулась вперёд во время работы.

Теперь вы получаете звонок о том, что есть проблема с веб-сайтом, которую необходимо немедленно устранить. С Git вам нет нужды создавать заплатку вместе с теми изменениями, которые вы уже сделали для iss53. А также не надо прикладывать много усилий, чтобы отменить эти изменения перед тем, как вы сможете начать работать над решением срочной проблемы. Всё, что вам нужно сделать, это перейти на ветку master.

Однако, прежде чем сделать это, учтите, что если в вашем рабочем каталоге или индексе имеются незафиксированные изменения, которые конфликтуют с веткой, на которую вы переходите, Git не позволит переключить ветки. Лучше всего при переключении веток иметь чистое рабочее состояние. Существует несколько способов добиться этого (а именно, прятанье (stash) работы

и правка (amend) коммита), которые мы рассмотрим позже. А на данный момент представим, что вы зафиксировали все изменения и можете переключиться обратно на ветку master:

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

Теперь рабочий каталог проекта находится точно в таком же состоянии, что и в момент начала работы над проблемой №53, так что вы можете сконцентрироваться на срочном изменении. Очень важно запомнить: Git возвращает ваш рабочий каталог к снимку состояния того коммита, на который указывает ветка, на которую вы переходите. Он добавляет, удаляет и изменяет файлы автоматически, чтобы гарантировать, что состояние вашей рабочей копии идентично последнему коммиту на ветке.

Итак, вам надо срочно исправить ошибку. Давайте создадим для этого ветку, на которой вы будете работать (см. Рисунок 3-13):

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

                                    $ vim index.html

    $ git commit -a -m 'fixed the broken email address'
[hotfix]: created 3a0874c: "fixed the broken email address"

                    1 files changed, 0 insertions(+), 1 deletions(-)
```

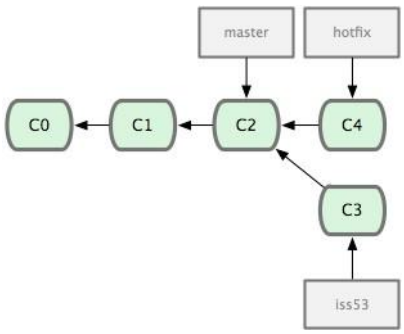

Рисунок 3.13: Ветка для решения срочной проблемы базируется на ветке master.

Вы можете запустить тесты, убедиться, что решение работает, и слить (merge) изменения назад в ветку master, чтобы включить его в продукт. Это делается с помощью команды git merge:

```
$ git checkout master

$ git merge hotfix
Updating f42c576..3a0874c
Fast forward
 README |    1 -
 1 files changed, 0 insertions(+), 1 deletions(-)
```

Наверное, вы заметили фразу «Fast forward» в этом слиянии. Так как ветка, которую мы слили, указывала на коммит, являющийся прямым родителем коммита, на котором мы сейчас находимся, Git просто сдвинул её указатель вперёд. Иными словами, когда вы пытаетесь слить один коммит с другим таким, которого можно достигнуть, проследовав по истории первого коммита, Git поступает проще, перемещая указатель вперёд, так как нет расходящихся изменений, которые нужно было бы сливать воедино. Это называется «fast forward» (перемотка).

Ваши изменения теперь в снимке состояния коммита, на который указывает ветка master, и вы можете включить изменения в продукт (см. Рисунок 3-14).

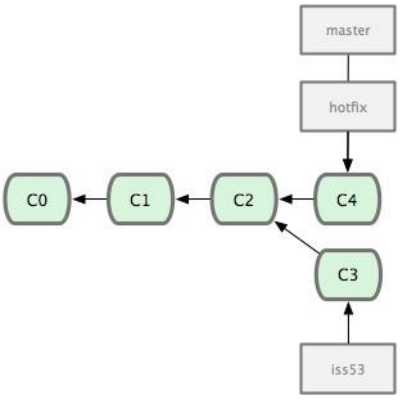

Рисунок 3.14: После слияния ветка master указывает туда же, куда и ветка hotfix.

После того, как очень важная проблема решена, вы готовы вернуться обратно к работе, которую делали, прежде чем были прерваны. Однако, сначала удалите ветку hotfix, так как она больше не нужна — ветка master уже указывает на то же место. Вы можете удалить ветку с помощью опции -d к git branch:

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

Теперь вы можете вернуться обратно к рабочей ветке для проблемы №53 и продолжить работать над ней (см. Рисунок 3-15):

```
   $ git checkout iss53
Switched to branch "iss53"

                                  $ vim index.html

   $ git commit -a -m 'finished the new footer [issue 53]'
[iss53]: created ad82d7a: "finished the new footer [issue 53]"

                    1 files changed, 1 insertions(+), 0 deletions(-)
```

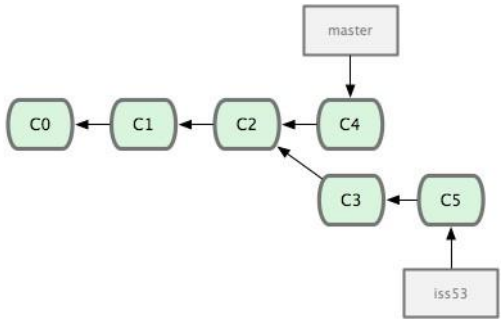

Рисунок 3.15: Ветка iss53 может двигаться вперёд независимо.

Стоит напомнить, что работа, сделанная на ветке hotfix, не включена в файлы на ветке iss53. Если вам это необходимо, вы можете выполнить слияние ветки master в ветку iss53 посредством команды git merge master. Или же вы можете подождать с интеграцией изменений до тех пор, пока не решите включить изменения на iss53 в продуктовую ветку master.

Основы слияния

Представьте себе, что вы разобрались с проблемой №53 и готовы объединить эту ветку и свой master. Чтобы сделать это, вы выполните слияние вашей ветки iss53 в ветку master точно так же, как делали ранее с веткой hotfix. Всё, что вы должны сделать — перейти на ту ветку, в которую вы хотите внести свои изменения и выполнить команду git merge:

```
                                $ git checkout master

                                  $ git merge iss53

Merge made by recursive.
     README |    1 +

                    1 files changed, 1 insertions(+), 0 deletions(-)
```

Сейчас слияние выглядит немного не так, как для ветки hotfix, которое вы делали ранее. В данном случае ваша история разработки разделилась в некоторой точке. Так как коммит на той ветке, на которой вы находитесь, не является прямым предком для ветки, которую вы сливаете, Git-у придётся проделать кое-какую работу. В этом случае Git делает простое трехходовое слияние, используя при этом два снимка состояния репозитория, на которые указывают вершины веток, и общий снимок-прародитель для этих двух веток. На

рисунке 3-16 выделены три снимка, которые Git будет использовать для слияния в этом случае.

Вместо того, чтобы просто передвинуть указатель ветки вперёд, Git создаёт новый снимок состояния, который является результатом трехходового слияния, и автоматически создает новый коммит, который указывает на этот новый снимок состояния (смотри Рисунок 3-17). Такой коммит называют коммит-слияние, так как он является особенным из-за того, что имеет больше одного предка.

Стоит отметить, что Git определяет наилучшего общего предка для слияния веток; в CVS или Subversion (версии ранее 1.5) этого не происходит. Разработчик должен сам указать основу для слияния. Это делает слияние в Git гораздо более простым занятием, чем в других системах.

Теперь, когда вы осуществили слияние ваших наработок, ветка iss53 вам больше не нужна. Можете удалить ее и затем вручную закрыть карточку (ticket) в вашей системе:

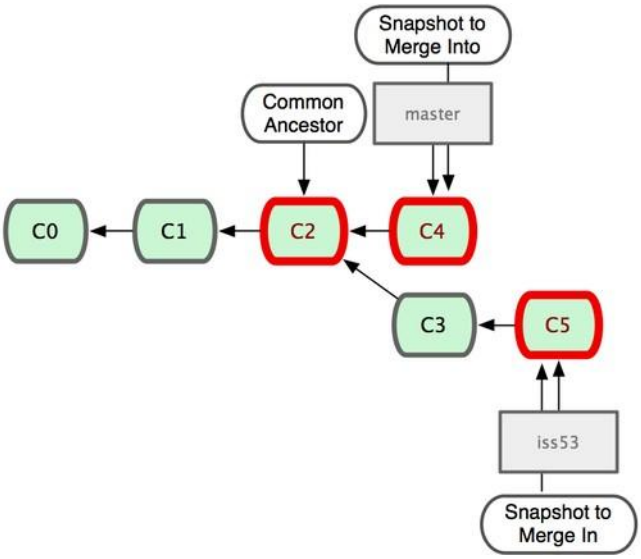

Рисунок 3.16: Git автоматически определяет наилучшего общего предка для слияния веток.

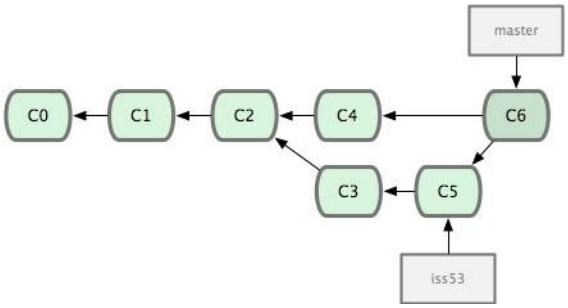

Рисунок 3.17: Git автоматически создает новый коммит, содержащий результаты слияния.

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

Основы конфликтов при слиянии

Иногда процесс слияния не идет гладко. Если вы изменили одну и ту же часть файла по-разному в двух ветках, которые собираетесь объединить, Git не сможет сделать это чисто. Если ваше решение проблемы №53 изменяет ту же часть файла, что и hotfix, вы получите конфликт слияния, и выглядеть он будет примерно следующим образом:

```
$ git merge iss53

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

Automatic merge failed; fix conflicts and then commit the result.
```

Git не создал новый коммит для слияния. Он приостановил этот процесс до тех пор, пока вы не разрешите конфликт. Если вы хотите посмотреть, какие файлы не прошли слияние (на любом этапе после возникновения конфликта), можете выполнить команду git status:

```
 [master*]$ git status
index.html: needs merge

# On branch master

# Changed but not updated:

#   (use "git add <file>..." to update what will be committed)

#   (use "git checkout -- <file>..." to discard changes in working directory)

#

# unmerged:   index.html
```

Всё, что имеет отношение к конфликту слияния и что не было разрешено, отмечено как unmerged. Git добавляет стандартные маркеры к файлам, которые имеют конфликт, так что вы можете открыть их вручную и разрешить эти конфликты. Ваш файл содержит секцию, которая выглядит примерно так:

```
<<<<<<< HEAD:index.html

<div id="footer">contact : email.support@github.com</div>

=======

<div id="footer">

please contact us at support@github.com

</div>
```

В верхней части блока (всё что выше =======) это версия из HEAD (вашей ветки master, так как именно на неё вы перешли перед выполнением команды merge), всё, что находится в нижней части — версия в iss53. Чтобы разрешить конфликт, вы должны либо выбрать одну из этих частей, либо как-то объединить содержимое по своему усмотрению.

Например, вы можете разрешить этот конфликт заменой всего блока, показанного выше, следующим блоком:

```
<div id="footer">

please contact us at email.support@github.com

</div>
```

Это решение содержит понемногу из каждой части, и я полностью удалил строки <<<<<<<,

======= и >>>>>>>. После того, как вы разрешили каждую из таких секций с каждым из конфликтных файлов, выполните git add для каждого конфликтного файла. Индексирование будет означать для Git, что все конфликты в файле теперь разрешены. Если вы хотите использовать графические инструменты для разрешения конфликтов, можете выполнить команду git merge- tool, которая запустит соответствующий графический инструмент и покажет конфликтные ситуации:

```
$ git mergetool

merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge vimdiff
Merging the files: index.html
```

```
Normal merge conflict for 'index.html':

{local}: modified

{remote}: modified

Hit return to start merge resolution tool (opendiff):
```

Если вы хотите использовать другой инструмент для слияния, нежели выбираемый по умолчанию (Git выбрал opendiff для меня, так как я выполнил команду на Mac). Вы можете увидеть все поддерживаемые инструменты, указанные выше после «merge tool candidates». Укажите название предпочтительного для вас инструмента. В Главе 7 мы обсудим, как изменить это значение по умолчанию для вашего окружения.

После того, как вы выйдете из инструмента для выполнения слияния, Git спросит вас, было ли оно успешным. Если вы отвечаете, что да — файл индексируется (добавляется в область для коммита), чтобы дать вам понять, что конфликт разрешен.

Можете выполнить git status ещё раз, чтобы убедиться, что все конфликты были разрешены:

```
$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

# modified:   index.html
```

Если вы довольны тем, что получили, и удостоверились, что всё, имевшее конфликты, было проиндексировано, можете выполнить git commit для завершения слияния. По умолчанию сообщение коммита будет выглядеть примерно так:

```
Merge branch 'iss53'

    Conflicts:

  index.html

#

# It looks like you may be committing a MERGE.

# If this is not correct, please remove the file

# .git/MERGE_HEAD

# and try again.
```

Вы можете дополнить это сообщение информацией о том, как вы разрешили конфликт, если считаете, что это может быть полезно для других в будущем. Например, можете указать почему вы сделали то, что сделали, если это не очевидно, конечно.

Управление ветками

Теперь, когда вы уже попробовали создавать, объединять и удалять ветки, пора познакомиться с некоторыми инструментами для управления ветками. Когда вы начнете постоянно использовать ветки, эти инструменты очень вам пригодятся.

Команда git branch делает несколько больше, чем просто создает и удаляет ветки.

Если вы выполните ее без аргументов, то получите простой список ваших текущих веток:

```
$ git branch
    iss53

* master
  testing
```

Обратите внимание на символ *, стоящий перед веткой master: он указывает на ту ветку, на которой вы находитесь в настоящий момент. Это означает, что если вы сейчас

выполните коммит, ветка master переместится вперёд в соответствии с вашими последними изменениями. Чтобы посмотреть последний коммит на каждой из веток, выполните команду git branch

-v:

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

Другая полезная возможность для выяснения состояния ваших веток состоит в том, чтобы оставить в этом списке только те ветки, для которых вы выполнили (или не выполнили) слияние с веткой, на которой сейчас находитесь. Для этих целей в Git, начиная с версии 1.5.6, есть опции --merged и --no-merged. Чтобы посмотреть те ветки, которые вы уже слили с текущей, можете выполнить команду git branch --merged:

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

Так как вы уже выполняли слияние для ветки iss53 ранее, вы видите ее в своем списке. Неплохой идеей было бы удалить командой git branch -d те ветки из этого списка, перед которыми нет символа *; вы уже объединили наработки из этих веток с другой веткой, так что вы ничего не теряете.

Чтобы посмотреть все ветки, содержащие наработки, которые вы еще не объединили с текущей веткой, выполните команду git branch --no-merged:

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

Вы увидите оставшуюся ветку. Так как она содержит ещё не слитые наработки, попытка удалить ее командой git branch -d не увенчается успехом:

```
$ git branch -d testing
error: The branch 'testing' is not an ancestor of your current HEAD.
 If you are sure you want to delete it, run 'git branch -D testing'.
```

Если вы действительно хотите удалить ветку и потерять наработки, вы можете сделать это при помощи опции -D, как указано в подсказке.

Приемы работы с ветками

Теперь, когда вы познакомились с основами ветвления и слияния, что вам делать с ветками дальше? В этом разделе мы рассмотрим некоторые стандартные приёмы работы,

которые становятся возможными, благодаря лёгкому осуществлению ветвления. И вы сможете выбрать, включить ли вам какие-то из них в свой цикл разработки.

Долгоживущие ветки

Так как Git использует простое трехходовое слияние, периодически сливать одну ветку с другой на протяжении большого промежутка времени достаточно просто. Это значит, вы можете иметь несколько веток, которые всегда открыты и которые вы используете для разных стадий вашего цикла разработки; вы можете регулярно сливать одну из них с другой.

Многие разработчики Git'а придерживаются такого подхода, при котором ветка master содержит исключительно стабильный код — единственный выпускаемый код. Для разработки и тестирования используется параллельная ветка, называемая develop или next, она может не быть стабильной постоянно, но в стабильные моменты её можно слить в master. Эта ветка используется для объединения завершённых задач из тематических веток (временных веток наподобие iss53), чтобы удостовериться, что эти изменения проходят все тесты и не вызывают ошибок.

В действительности же, мы говорим об указателях, передвигающихся вверх по линии коммитов, которые вы делаете. Стабильные ветки далеко внизу линии вашей истории коммитов, наиболее свежие ветки находятся ближе к верхушке этой линии (смотри Рисунок 3-18).

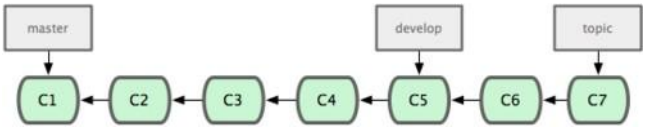

Рисунок 3.18: Более стабильные ветки, как правило, находятся дальше в истории коммитов.

В общем, об этом проще думать как о силосных башнях, где набор коммитов переходит в более стабильную башню только тогда, когда он полностью протестирован (смотри Рисунок 3-19).

Вы можете применять эту идею для нескольких разных уровней стабильности. Некоторые большие проекты также имеют ветку proposed или pu (proposed updates — предлагаемые изменения), которые включают в себя ветки, не готовые для перехода в ветку next или master. Идея такова, что ваши ветки находятся на разных уровнях стабильности; когда они достигают более высокого уровня стабильности, они сливаются с веткой, стоящей на более

высоком уровне. Опять-таки, не обязательно иметь долгоживущие ветки, но часто это очень полезно, особенно когда вы имеете дело с очень большими и сложными проектами.

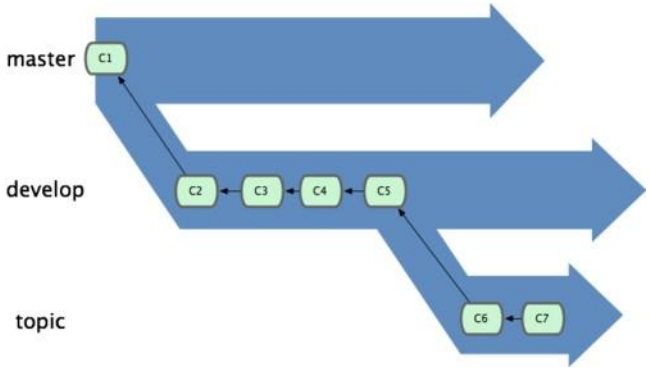

Рисунок 3.19: Может быть полезным думать о ветках как о силосных башнях.

Тематические ветки

Тематические ветки, однако, полезны в проектах любого размера. Тематическая ветка — недолговечная ветка, которую вы создаете и используете для некоторой отдельной возможности или вспомогательной работы. Это то, чего вы, вероятно, никогда не делали с системами управления версиями раньше, так как создание и слияние веток обычно слишком затратно. Но в Git принято создавать ветки, работать над ними, объединять и удалять их по несколько раз в день.

Вы видели подобное в последнем разделе, где вы создавали ветки iss53 и hotfix. Вы сделали несколько коммитов на этих ветках и удалили их сразу после объединения с вашей основной веткой. Такая техника позволяет вам быстро и полноценно переключать контекст. Когда все изменения в данной ветке относятся к определённой теме, достаточно просто отслеживать, что происходило во время работы с кодом. Вы можете сохранить там изменения на несколько минут, дней или месяцев, а затем, когда они готовы, слить их с основной веткой, независимо от порядка, в котором их создавали или работали над ними.

Рассмотрим пример, когда выполняется некоторая работа (в ветке master), делается ответвление для решения проблемы (iss91), выполняется немного работы на ней, делается ответвление второй ветки для другого пути решения той же задачи (iss91v2), осуществляется переход назад на вашу основную ветку (master) и выполнение работы на ней, затем делается ответвление от неё для выполнения чего-то, в чём вы не уверены, что это хорошая идея (ветка dumbidea).

Ваша история коммитов будет выглядеть примерно так как на Рисунке 3-20.

Теперь представим, вы решили, что вам больше нравится второе решение для вашей задачи (iss91v2); и вы показываете ветку dumbidea вашим коллегам и оказывается, что она

просто гениальна. Так что вы можете выбросить оригинальную ветку iss91 (теряя при этом коммиты C5 и C6) и слить две другие. Тогда ваша история будет выглядеть как на Рисунке 3-21.

Важно запомнить, что когда вы выполняете все эти действия, ветки являются полностью локальными. Когда вы выполняете ветвление и слияние, всё происходит только в вашем репозитории — связь с сервером не осуществляется.

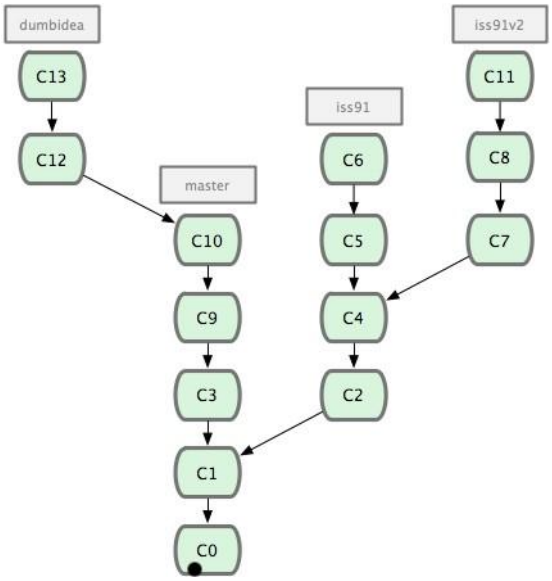

Рисунок 3.20: История коммитов с несколькими тематическими ветками.

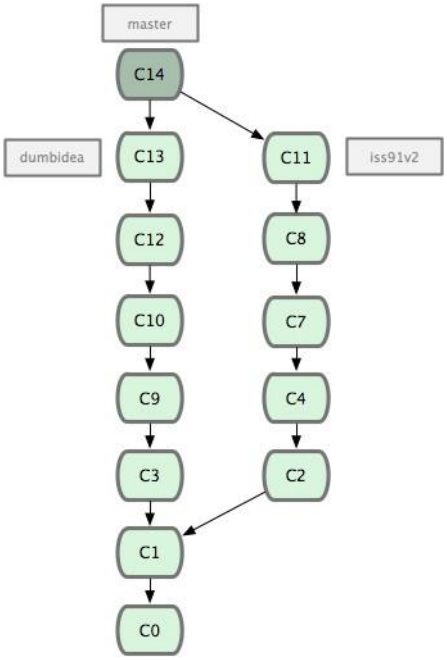

Рисунок 3.21: Ваша история после слияния dumbidea и iss91v2.

Удалённые ветки

Удалённые ветки — это ссылки на состояние веток в ваших удалённых репозиториях. Это локальные ветки, которые нельзя перемещать; они двигаются автоматически всякий раз,

когда вы осуществляете связь по сети. Удалённые ветки действуют как закладки для напоминания о том, где ветки в удалённых репозиториях находились во время последнего подключения к ним.

Они выглядят как (имя удал. репоз.)/(ветка). Например, если вы хотите посмотреть, как выглядела ветка master на сервере origin во время последнего соединения с ним, проверьте ветку origin/master. Если вы с партнёром работали над одной проблемой, и он выложил ветку iss53, у вас может быть своя локальная ветка iss53; но та ветка на сервере будет указывать на коммит в origin/iss53.

Всё это, возможно, сбивает с толку, поэтому давайте рассмотрим пример. Скажем, у вас есть Git-сервер в сети на git.ourcompany.com. Если вы склонируете (clone) с него, Git автоматически назовёт его для вас origin, заберёт с него все данные, создаст указатель на его ветку master и назовёт его локально origin/master (но вы не можете его двигать). Git также сделает вам вашу собственную ветку master, которая будет начинаться там же, где и ветка master в origin, так что вам будет с чем начать работать (смотри Рис. 3-22).

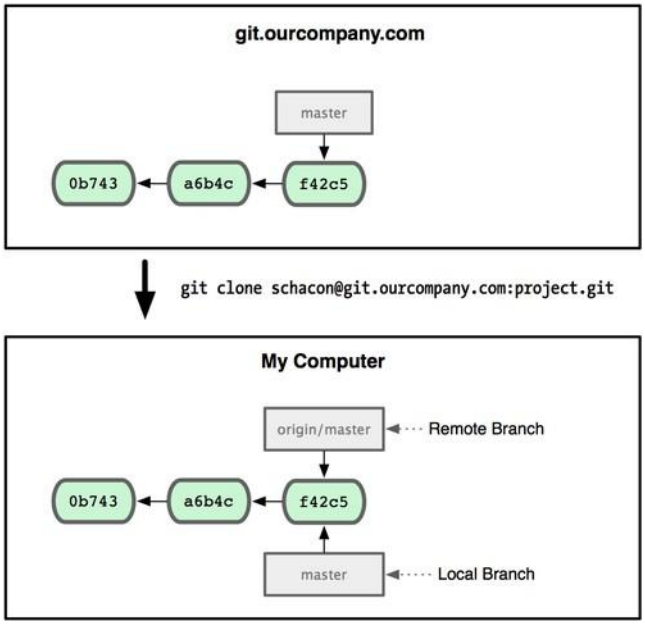

Рисунок 3.22: Клонирование Git-проекта даёт вам собственную ветку master и origin/master, указывающий на ветку master в origin.

Если вы сделаете что-то в своей локальной ветке master, а тем временем кто-то ещё отправит (push) изменения на git.ourcompany.com и обновит там ветку master, то ваши истории продолжатся по-разному. К тому же, до тех пор, пока вы не свяжетесь с сервером origin, ваш указатель origin/master не будет сдвигаться (смотри Рисунок 3-23).

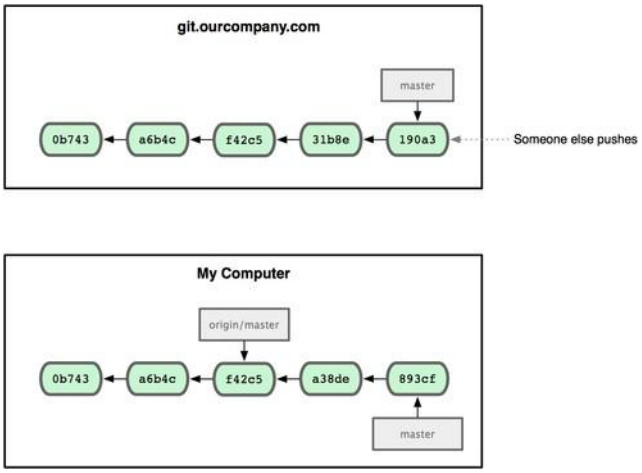

Рисунок 3.23: При выполнении локальной работы и отправке кем-то изменений на удалённый сервер каждая история продолжается по-разному.

Для синхронизации вашей работы выполняется команда git fetch origin. Эта команда ищет, какому серверу соответствует origin (в нашем случае это git.ourcompany.com); извлекает оттуда все данные, которых у вас ещё нет, и обновляет ваше локальное хранилище данных; сдвигает указатель origin/master на новую позицию (смотри Рисунок 3-24).

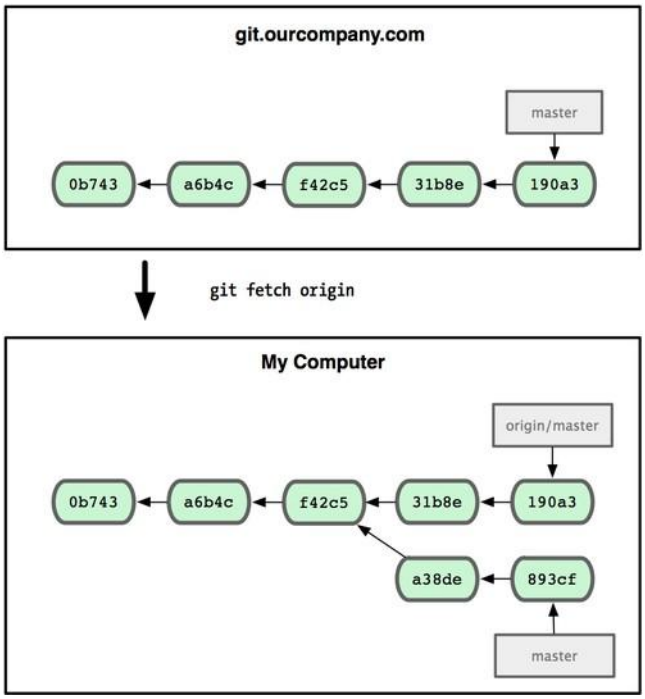

Рисунок 3.24: Команда git fetch обновляет ваши удалённые ссылки.

Чтобы продемонстрировать то, как будут выглядеть удалённые ветки в ситуации с несколькими удалёнными серверами, предположим, что у вас есть ещё один внутренний Git-сервер, который используется для разработки только одной из ваших команд разработчиков. Этот сервер находится на git.team1.ourcompany.com. Вы можете добавить его в качестве новой удалённой ссылки на проект, над которым вы сейчас работаете с помощью команды

git remote add так же, как было описано в Главе 2. Дайте этому удалённому серверу имя teamone, которое будет сокращением для полного URL (смотри Рисунок 3-25).

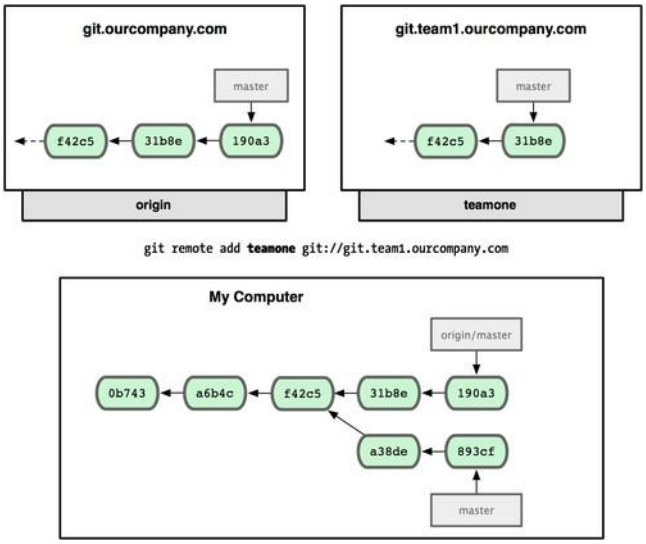

Рисунок 3.25: Добавление дополнительного удалённого сервера.

Теперь можете выполнить git fetch teamone, чтобы извлечь всё, что есть на сервере и нет у вас. Так как в данный момент на этом сервере есть только часть данных, которые есть на сервере origin, Git не получает никаких данных, но выставляет удалённую ветку с именем teamone/master, которая указывает на тот же коммит, что и ветка master на сервере teamone (смотри Рисунок 3-26).

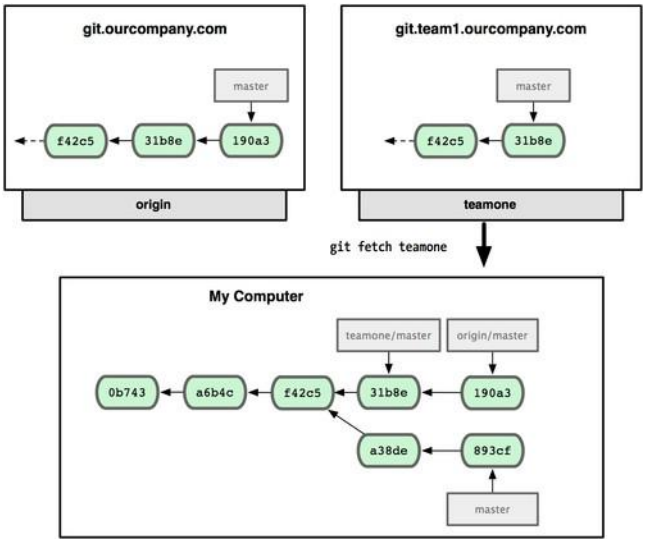

Рисунок 3.26: У вас появилась локальная ссылка на ветку master на teamone-е.

Отправка изменений

Когда вы хотите поделиться веткой с окружающими, вам необходимо отправить (push) её

на удалённый сервер, на котором у вас есть права на запись. Ваши локальные ветки автоматически не синхронизируются с удалёнными серверами — вам нужно явно отправить

те ветки, которыми вы хотите поделиться. Таким образом, вы можете использовать свои личные ветки для работы, которую вы не хотите показывать, и отправлять только те тематические ветки, над которыми

вы хотите работать с кем-то совместно.

Если у вас есть ветка serverfix, над которой вы хотите работать с кем-то ещё, вы можете отправить её точно так же, как вы отправляли вашу первую ветку. Выполните git push (удал. сервер) (ветка):

```
$ git push origin serverfix
Counting objects: 20, done.

   Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
    Total 15 (delta 5), reused 0 (delta 0)

                      To git@github.com:schacon/simplegit.git

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

Это в некотором роде сокращение. Git автоматически разворачивает имя ветки server-fix до refs/heads/serverfix:refs/heads/serverfix, что означает «возьми мою локальную ветку serverfix и обнови из неё удалённую ветку serverfix». Мы подробно обсудим часть с refs/heads/ в Главе 9, но обычно можно её опустить. Вы можете сделать также git push origin serverfix:serverfix, что означает то же самое — здесь говорится «возьми мой serverfix и сделай его удалённым serverfix». Можно использовать этот формат для отправки локальной ветки в удалённую ветку, которая называется по-другому. Если вы не хотите, чтобы ветка называлась serverfix на удалённом сервере, то вместо предыдущей команды выполните git push origin serverfix:awesomebranch. Так ваша локальная ветка serverfix отправится в ветку awesomebranch удалённого проекта.

В следующий раз, когда один из ваших соавторов будет получать обновления с сервера, он получит ссылку на то, на что указывает serverfix на сервере, как удалённую ветку ori- gin/serverfix:

```
                                $ git fetch origin

                          remote: Counting objects: 20, done.

remote: Compressing objects: 100% (14/14), done.
 remote: Total 15 (delta 5), reused 0 (delta 0)
     Unpacking objects: 100% (15/15), done.

                          From git@github.com:schacon/simplegit

                 * [new branch]      serverfix     -> origin/serverfix
```

Важно отметить, что когда при получении данных у вас появляются новые удалённые ветки, вы не получаете автоматически для них локальных редактируемых копий. Другими

словами, в нашем случае вы не получите новую ветку serverfix — только указатель origin/serverfix, который вы не можете менять.

Чтобы слить эти наработки в вашу текущую рабочую ветку, можете выполнить git merge origin/serverfix. Если вы хотите иметь собственную ветку serverfix, над которой вы сможете работать, можете создать её на основе удалённой ветки:

```
$ git checkout -b serverfix origin/serverfix

Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"
```

Это даст вам локальную ветку, на которой можно работать. Она будет начинаться там, где и origin/serverfix.

Отслеживание веток

Получение локальной ветки с помощью git checkout из удалённой ветки автоматически создаёт то, что называется *отслеживаемой веткой*. Отслеживаемые ветки — это локальные ветки, которые напрямую связаны с удалённой веткой. Если, находясь на отслеживаемой ветке, вы наберёте git push, Git уже будет знать, на какой сервер и в какую ветку отправлять изменения. Аналогично выполнение git pull на одной из таких веток сначала получает все удалённые ссылки, а затем автоматически делает слияние с соответствующей удалённой веткой.

При клонировании репозитория, как правило, автоматически создаётся ветка master, которая отслеживает origin/master, поэтому git push и git pull работают для этой ветки «из коробки» и не требуют дополнительных аргументов. Однако, вы можете настроить отслеживание и других веток удалённого репозитория. Простой пример, как это сделать, вы увидели только что — git checkout -b [ветка] [удал. сервер]/[ветка]. Если вы используете Git версии 1.6.2 или более позднюю, можете также воспользоваться сокращением --track:

```
$ git checkout --track origin/serverfix

Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"
```

Чтобы настроить локальную ветку с именем, отличным от имени удалённой ветки, вы можете легко использовать первую версию с другим именем локальной ветки:

```
$ git checkout -b sf origin/serverfix

Branch sf set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "sf"
```

Теперь ваша локальная ветка sf будет автоматически отправлять (push) и получать (pull) изменения из origin/serverfix.

Удаление веток на удалённом сервере

Скажем, вы и ваши соавторы закончили с нововведением и слили его в ветку master на удалённом сервере (или в какую-то другую ветку, где хранится стабильный код). Вы можете удалить ветку на удалённом сервере, используя несколько бестолковый синтаксис git push [удал. сервер] :[ветка]. Чтобы удалить ветку serverfix на сервере, выполните следующее:

```
$ git push origin :serverfix

To git@github.com:schacon/simplegit.git

- [deleted]         serverfix
```

Хлоп. Нет больше ветки на вашем сервере. Вам может захотеться сделать закладку на текущей странице, так как эта команда вам понадобится, а синтаксис вы, скорее всего, забудете. Можно запомнить эту команду вернувшись к синтаксису git push [удал. сервер] [лок. ветка]:[удал. ветка], который мы рассматривали немного раньше. Опуская часть [лок. ветка], вы по сути говорите «возьми ничто в моём репозитории и сделай так, чтобы в [удал. ветка] было то же самое».

Перемещение

В Git есть два способа включить изменения из одной ветки в другую: merge (слияние) и rebase (перемещение). В этом разделе вы узнаете, что такое перемещение, как его осуществлять, почему это удивительный инструмент и в каких случаях вам не следует его использовать.

3.5.1 Основы перемещения

Если вы вернетесь назад к примеру из раздела Слияние (смотри Рисунок 3-27), вы увидите, что вы разделили вашу работу на два направления и выполняли коммиты на двух разных ветках.

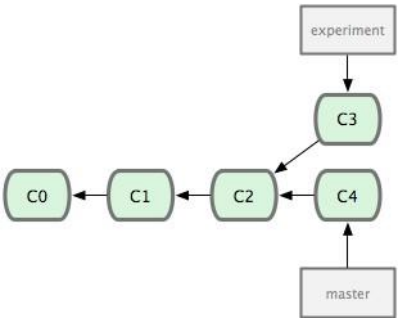

Рисунок 3.27: Впервые разделенная история коммитов.

Наиболее простое решение для объединения веток, как мы уже выяснили, команда merge. Эта команда выполняет трехходовое слияние между двумя последними снимками состояний из веток (C3 и C4) и последним общим предком этих двух веток (C2), создавая новый снимок состояния (и коммит), как показано на Рисунке 3-28.

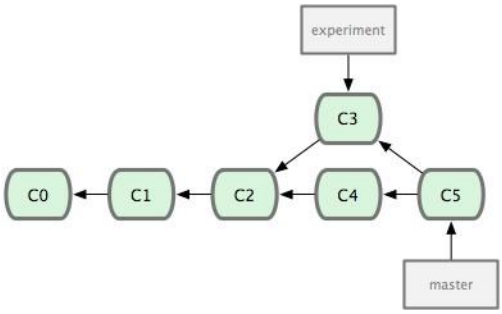

Рисунок 3.28: Слияние ветки для объединения разделившейся истории разработки.

Однако, есть и другой путь: вы можете взять изменения, представленные в C3, и применить их сверху C4. В Git это называется *перемещение* (rebasing). При помощи команды rebase вы можете взять все изменения, которые попали в коммиты на одной из веток, и повторить их на другой.

В этом примере вы выполните следующее:

```
$ git checkout experiment

$ git rebase master

First, rewinding head to replay your work on top of it...
Applying: added staged command
```

Это работает следующим образом: находится общий предок для двух веток (на которой вы находитесь сейчас и на которую вы выполняете перемещение); берётся разница, представленная в каждом из коммитов на текущей ветке, и сохраняется во временные файлы; текущая ветка устанавливается на такой же коммит, что и ветка, на которую вы выполняете перемещение; и, наконец, последовательно применяются все изменения. Рисунок 3-29 иллюстрирует этот процесс.

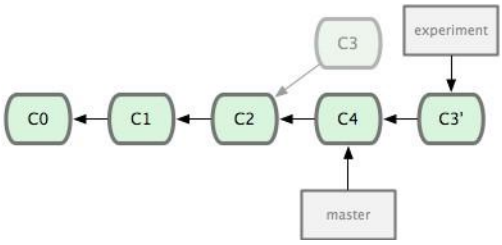

Рисунок 3.29: Перемещение изменений, сделанных в C3, на C4.

На этом этапе можно переключиться на ветку master и выполнить слияние-перемотку (fast-forward merge) (смотри Рисунок 3-30).

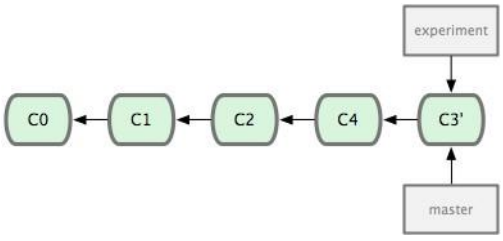

Рисунок 3.30: Перемотка ветки master.

Теперь снимок состояния, на который указывает C3, точно такой же, что тот, на который указывал C5 в примере со слиянием. Нет никакой разницы в конечном результате объединения, но перемещение выполняется для того, чтобы история была более аккуратной. Если вы посмотрите лог (log) перемещенной ветки, то увидите, что он выглядит как линейная история работы: выходит, что вся работа выполнялась последовательно, когда в действительности она выполнялась параллельно.

Часто вы будете делать это, чтобы удостовериться, что ваши коммиты правильно применяются для удаленных веток — возможно для проекта, владельцем которого вы не являетесь, но в который вы хотите внести свой вклад. В этом случае вы будете выполнять работу в ветке, а затем, когда будете готовы внести свои изменения в основной проект, выполните перемещение вашей работы на origin/master. Таким образом, владельцу проекта не придется делать никаких действий по объединению — просто перемотка (fast-forward) или чистое применение патчей.

Заметьте, что снимок состояния, на который указывает последний коммит, который у вас получился, является ли этот коммит последним перемещенным коммитом (для случая выполнения перемещения) или итоговым коммитом слияния (для случая выполнения слияния), есть один и тот же снимок — разной будет только история. Перемещение применяет изменения из одной линии разработки в другую в том порядке, в котором они были представлены, тогда как слияние объединяет вместе конечные точки двух веток.

Более интересные перемещения

Вы также можете выполнять перемещение не только для перемещения ветки. Возьмём, например, историю разработки как на Рисунке 3-31. Вы создали тематическую ветку (server), чтобы добавить в проект некоторый функционал для серверной части, и сделали коммит. Затем вы выполнили ответвление, чтобы сделать изменения для клиентской части, и несколько раз выполнили коммиты. Наконец, вы вернулись на ветку server и сделали ещё несколько коммитов.

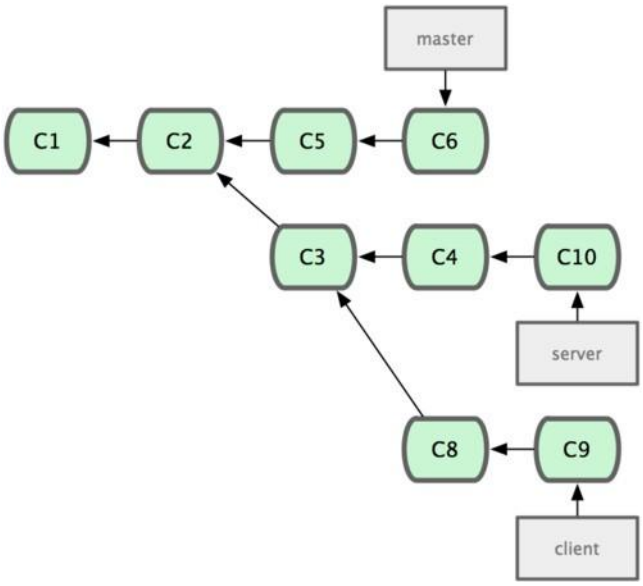

Рисунок 3.31: История разработки с тематической веткой, ответвленной от другой тематической ветки.

Предположим, вы решили, что хотите внести ваши изменения для клиентской части в основную линию разработки для релиза, но при этом хотите оставить в стороне изменения для серверной части, пока они не будут полностью протестированы. Вы можете взять изменения из ветки client, которых нет на server (C8 и C9), и применить их на ветке master при помощи опции --onto команды git rebase:

```
$ git rebase --onto master server client
```

По сути, это указание «переключиться на ветку client, взять изменения от общего предка веток client и server и повторить их на master». Это немного сложно; но результат, показанный на Рисунке 3-32, достаточно классный.

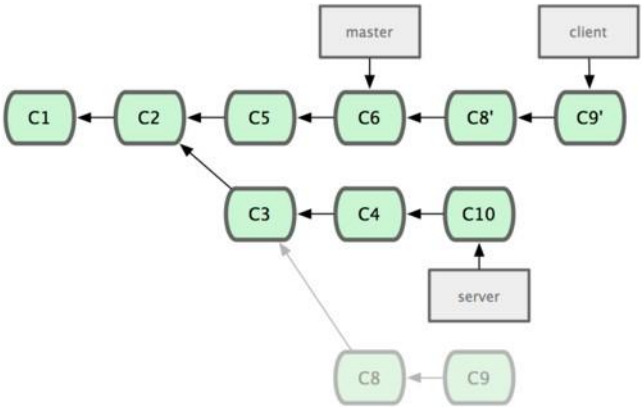

Рисунок 3.32: Перемещение тематической ветки, ответвленной от другой тематической ветки.

Теперь вы можете выполнить перемотку (fast-forward) для вашей ветки master (смотри Рисунок 3-33):

```
$ git checkout master

$ git merge client
```

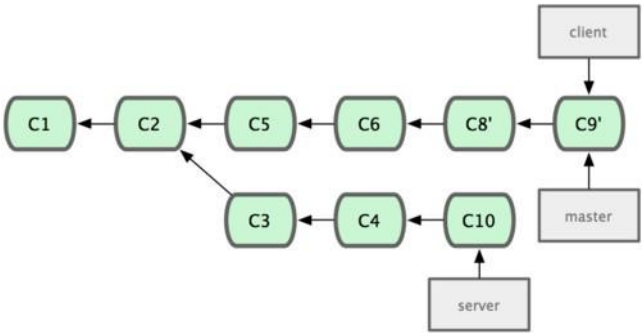

Рисунок 3.33: Перемотка ветки master, чтобы включить изменения из ветки client.

Представим, что вы также решили включить ветку server в основную ветку. Вы можете выполнить перемещение ветки server на ветку master без предварительного переключения на эту ветку при помощи команды git rebase [осн. ветка] [тем. ветка] — которая устанавливает тематическую ветку (в данном случае server) как текущую и применяет её изменения на основной ветке (master):

```
$ git rebase master server
```

Эта команда применит изменения из вашей работы над веткой server на вершину ветки master, как показано на Рисунке 3-34.

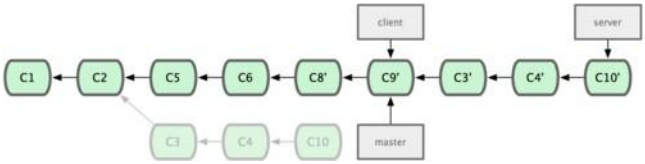

Рисунок 3.34: Перемещение вашей ветки server на вершину ветки master.

Затем вы можете выполнить перемотку (fast-forward) основной ветки (master):

```
$ git checkout master

$ git merge server
```

Вы можете удалить ветки client и server, так как вся работа из них включена в основную линию разработки и они вам больше не нужны. При этом полная история вашего рабочего процесса выглядит как на Рисунке 3-35:

```
$ git branch -d client

$ git branch -d server
```

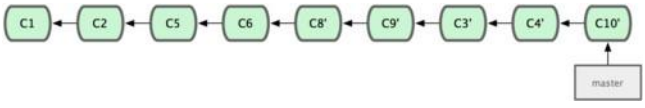

Рисунок 3.35: Финальная история коммитов.

Возможные риски перемещения

Все бы хорошо, но кое-что омрачает всю прелесть использования перемещения. Это выражается одной строчкой: **Не перемещайте коммиты, которые вы выложили в публичный репозиторий.**

Если вы будете следовать этому указанию, все будет хорошо. Если нет — люди возненавидят вас, вас будут презирать ваши друзья и семья.

Когда вы что-то перемещаете, вы отменяете существующие коммиты и создаете новые, которые являются похожими на старые, но в чем-то другими. Если вы выкладываете ваши коммиты куда-нибудь, и другие забирают их себе и в дальнейшем основывают на них свою работу, а затем вы переделываете эти коммиты командой git rebase и выкладываете их снова, ваши коллеги будут вынуждены заново выполнять слияние для своих наработок. Все запутается, когда вы в очередной раз попытаетесь включить их работу в свою.

Давайте рассмотрим пример того, как выполненное вами перемещение наработок, представленных для общего доступа, может вызвать проблемы. Представьте себе, что вы клонировали себе репозиторий с центрального сервера и поработали в нем. Ваша история коммитов выглядит как на Рисунке 3-36.

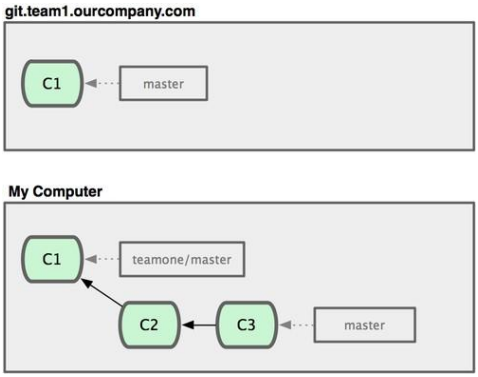

Рисунок 3.36: Клонирование репозитория и выполнение в нём какой-то работы.

Теперь кто-то ещё выполняет работу, причём работа включает в себя и слияние, и отправляет свои изменения на центральный сервер. Вы извлекаете их и сливаете новую удалённую ветку со своей работой. Тогда ваша история выглядит как на Рисунке 3-37.

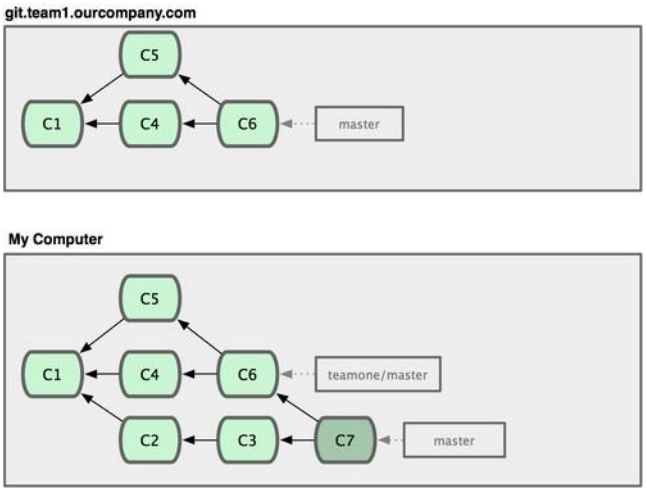

Рисунок 3.37: Извлечение коммитов и слияние их со своей работой.

Далее, человек, выложивший коммит, содержащий слияние, решает вернуться и вместо слияния (merge) переместить (rebase) свою работу; он выполняет git push --force, чтобы переписать историю на сервере. Затем вы извлекаете изменения с этого сервера, включая и новые коммиты.

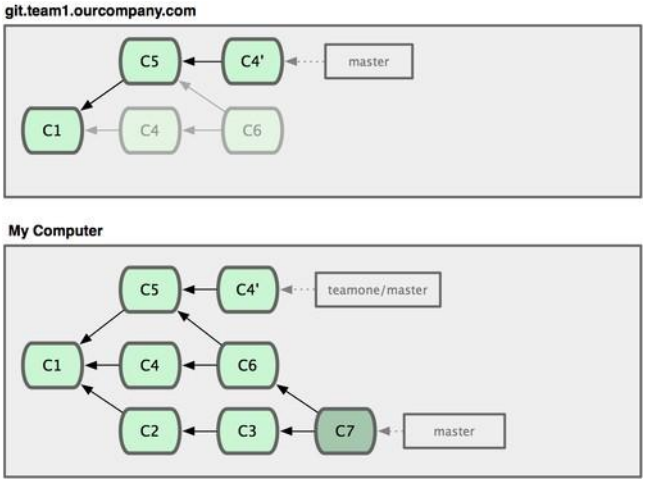

Рисунок 3.38: Кто-то выложил перемещённые коммиты, отменяя коммиты, на которых вы основывали свою работу.

На этом этапе вы вынуждены объединить эту работу со своей снова, даже если вы уже сделали это ранее. Перемещение изменяет у этих коммитов SHA-1 хеши, так что для Git они выглядят как новые коммиты, тогда как на самом деле вы уже располагаете наработками C4 в вашей истории (смотри Рисунок 3-39).

Вы вынуждены объединить эту работу со своей на каком-либо этапе, чтобы иметь возможность продолжать работать с другими разработчиками в будущем. После того, как вы сделаете это, ваша история коммитов будет содержать оба коммита — C4 и C4’, которые имеют разные SHA-1 хеши , но представляют собой одинаковые изменения и имеют одинаковые сообщения. Если вы выполните команду git log, когда ваша история выглядит таким образом, вы увидите два коммита, которые имеют одинакового автора и одни и те же

сообщения. Это сбивает с толку. Более того, если вы отправите такую историю обратно на сервер, вы добавите все эти перемещенные коммиты в репозиторий центрального сервера, что может ещё больше запутать людей.

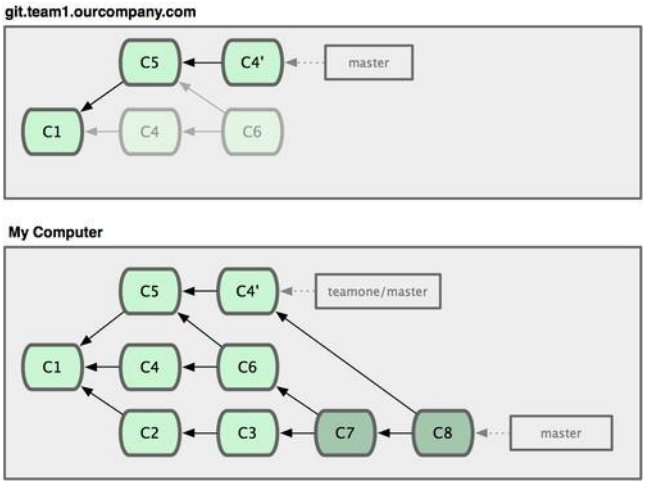

Рисунок 3.39: Вы снова выполняете слияние для той же самой работы в новый коммит слияния.

Если вы рассматриваете перемещение как возможность наведения порядка и работы с коммитами до того, как выложили их, и если вы перемещаете только коммиты, которые никогда не находились в публичном доступе — всё нормально. Если вы перемещаете коммиты, которые уже были представлены для общего доступа, и люди, возможно, основывали свою работу на этих коммитах, тогда вы можете получить наказание за разные неприятные проблемы.

Итоги

Мы рассмотрели основы ветвления и слияния в Git. Вы должны чувствовать себя уверенно при создании и переходе на новые ветки, переключении между ветками и слиянии локальных веток. Вы также должны иметь возможность делиться своими ветками, выкладывая их на общий сервер, работать с другими людьми над общими ветками и перемещать свои ветки до того, как они были представлены для общего доступа.