ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ЛЕКЦИЯ № 5

ПРЕПОДАВАТЕЛЬ: ХУСТОЧКА А.В.

ВЕТВЛЕНИЕ В GIT

- Почти каждая система контроля версий 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 системах контроля версий 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 Git'а это его "killer feature" и она безусловно выделяет Git среди других систем. Что же в ней такого особенного? Способ ветвления в Git'е чрезвычайно легковесен, что делает операции ветвления практически мгновенными и переключение туда-сюда между ветками обычно так же быстрым. В отличие от многих других систем контроля версий, 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.

ЧТО ТАКОЕ ВЕТКА?

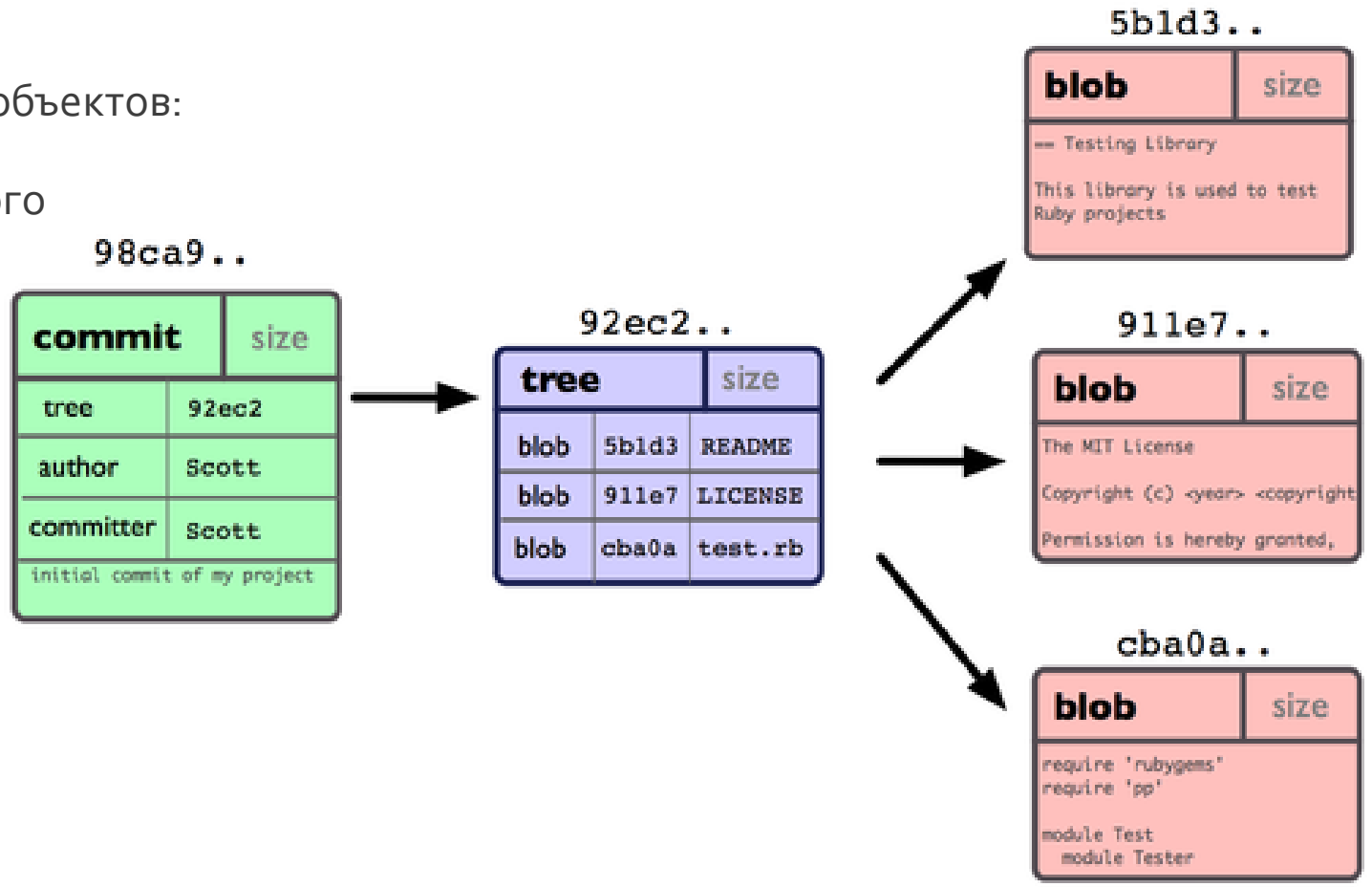
- Чтобы на самом деле разобраться в том, как Git работает с ветками, мы должны сделать шаг назад и рассмотреть, как Git хранит свои данные. Как вы, наверное, помните, Git хранит данные не как последовательность изменений или дельт, а как последовательность снимков состояния (snapshot).
- Когда вы создаёте коммит в Git'е, Git записывает в базу объект-коммит, который содержит указатель на снимок состояния, записанный ранее в индекс, метаданные автора и комментария и ноль и более указателей на коммиты, являющиеся 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 хотите добавить их все в индекс и сделать коммит. При добавлении файлов в индекс для каждого из них вычислится контрольная сумма (SHA-1 хеш, о котором мы упоминали в главе 1), затем эти версии файлов будут сохранены в Git-репозиторий (Git обращается к ним как к двоичным данным), а их контрольные суммы добавятся 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 эти объекты-деревья в Git-репозиторий. Затем Git создаёт объект для коммита, в котором есть метаданные и указатель на объект-дерево для корня проекта. Таким образом, Git сможет воссоздать текущее состояние, когда будет нужно.

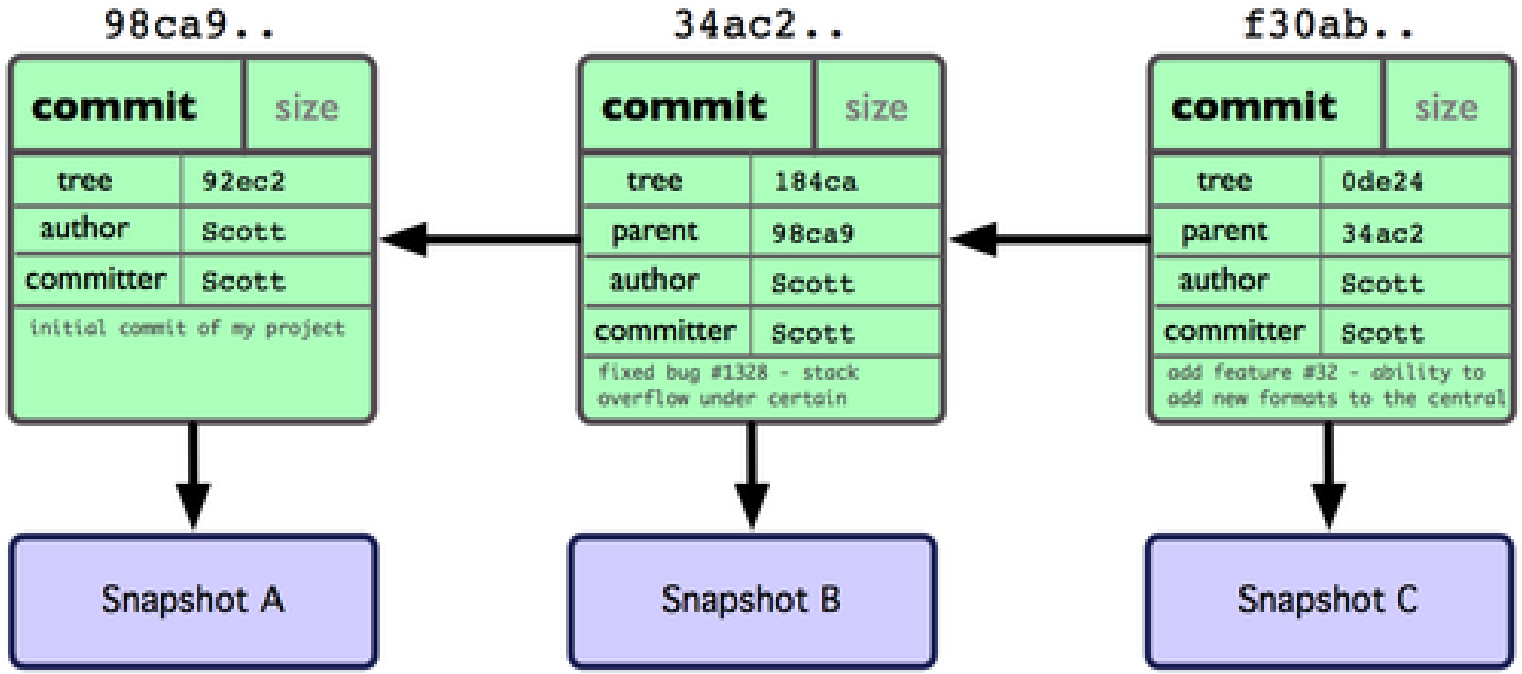
Ваш Git-репозиторий теперь содержит пять объектов:

- по одному блобу для содержимого каждого из трёх файлов,
- одно дерево, в котором перечислено содержимое каталога и определено соответствие имён файлов и блобов,
- один коммит с указателем на тот самый объект-дерево для корня и со всеми метаданными коммита.



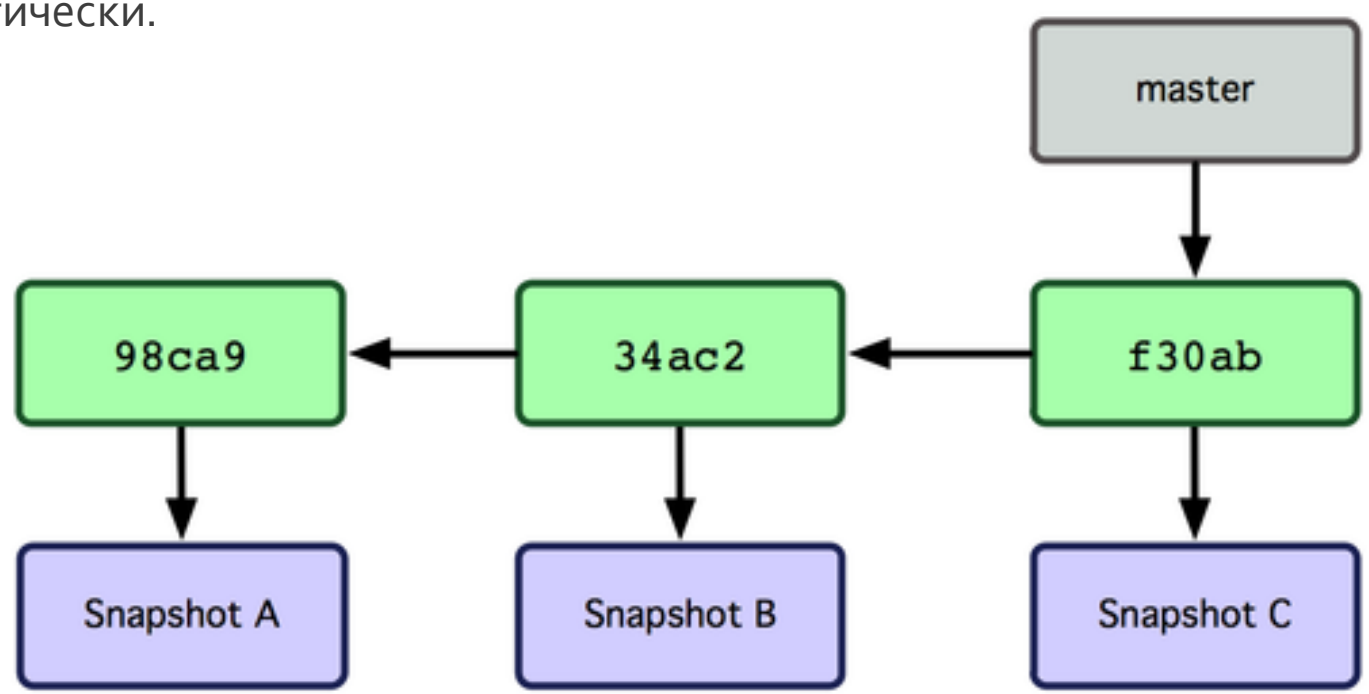
Данные репозитория с единственным коммитом

- Если вы сделаете некоторые изменения и создадите новый коммит, то следующий коммит сохранит указатель на коммит, который шёл непосредственно перед ним. После следующих двух коммитов история может выглядеть, как на рисунке.



Данные объектов Git’a для нескольких коммитов.

- Ветка в Git’е — это просто легковесный подвижный указатель на один из этих коммитов. Ветка по умолчанию в Git’е называется master. Когда вы создаёте коммиты на начальном этапе, вам дана ветка master, указывающая на последний сделанный коммит. При каждом новом коммите она сдвигается вперёд автоматически.

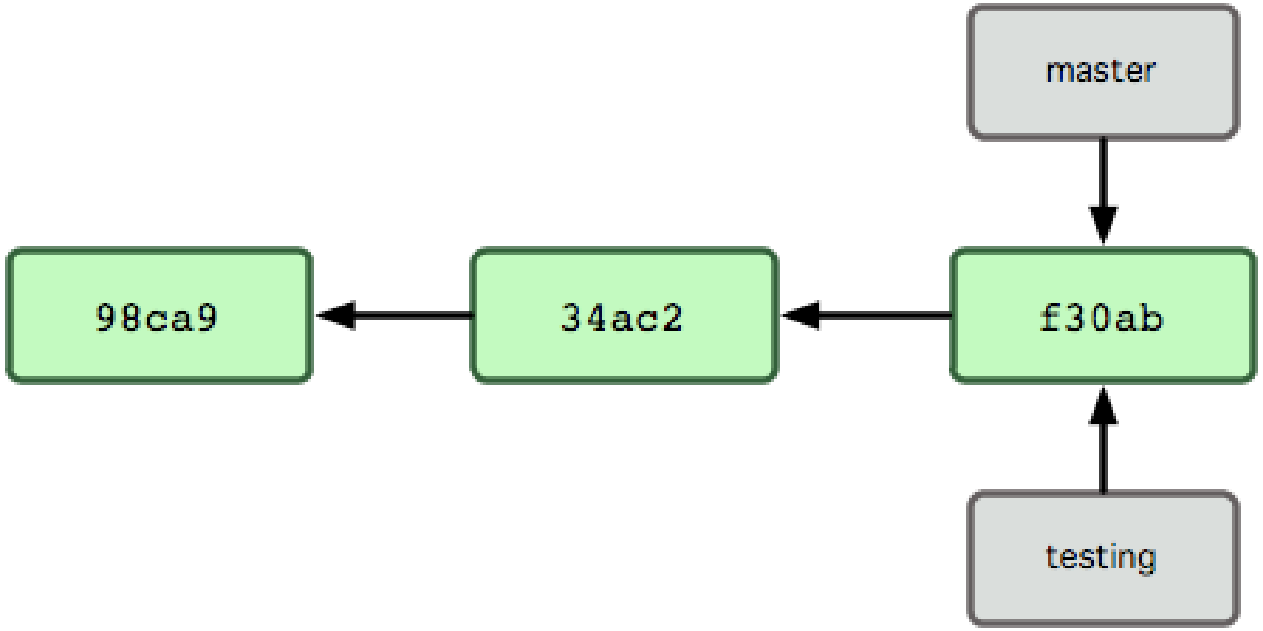


Ветка указывает на историю коммитов.

- Что произойдёт, если вы создадите новую ветку? Итак, этим вы создадите новый указатель, который можно будет перемещать. Скажем, создадим новую ветку под названием testing. Это делается командой git branch:

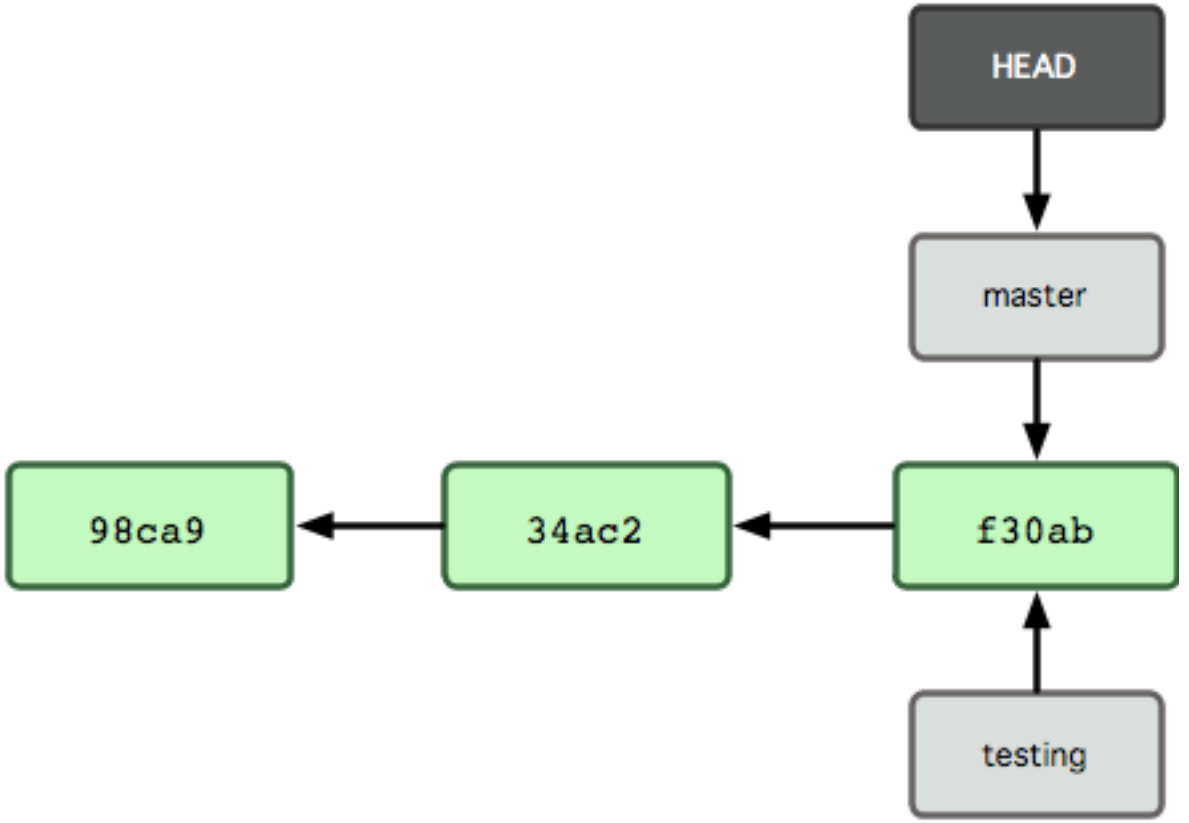
```
$ git branch testing
```

- Эта команда создаст новый указатель на тот самый коммит, на котором вы сейчас находитесь.



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 системах контроля версий, таких как Subversion или CVS, к которым вы, возможно, привыкли. В Git'е это указатель на локальную ветку, на которой вы находитесь. В данный момент вы всё ещё на ветке master. Команда git branch только создала новую ветку, она не переключила вас на неё.

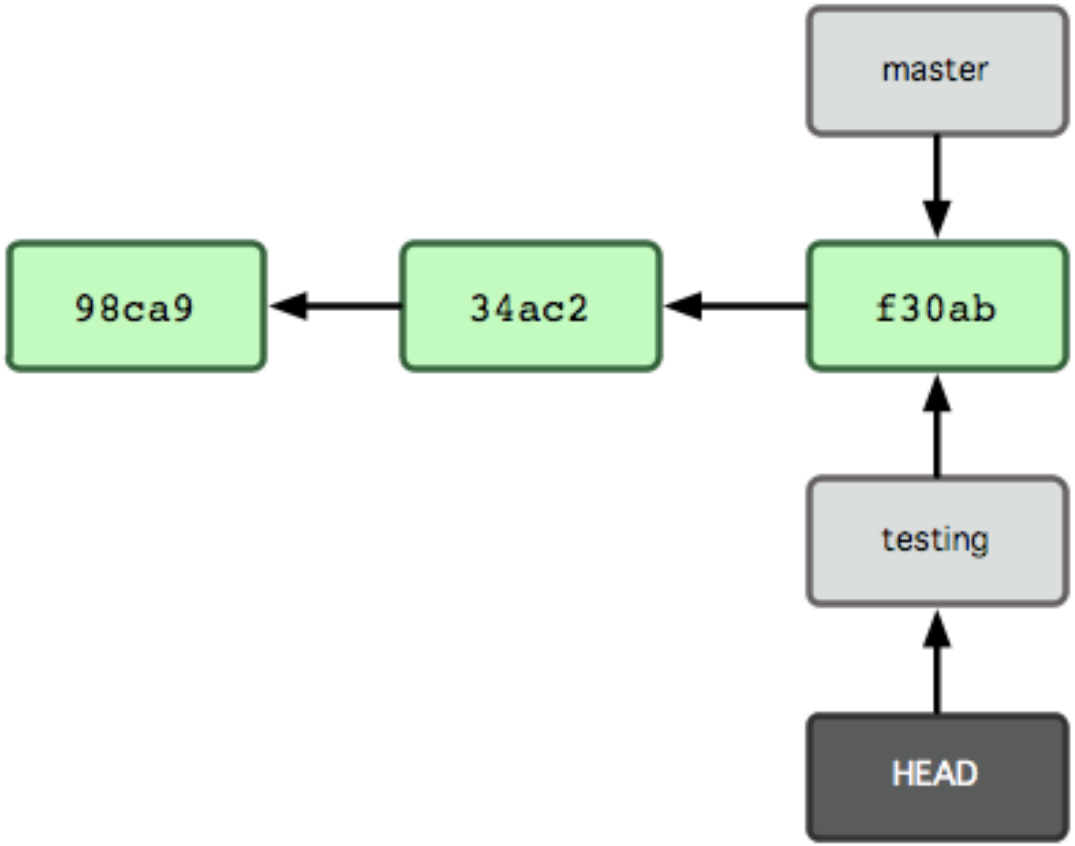


Файл HEAD указывает на текущую ветку.

- Чтобы перейти на существующую ветку, вам надо выполнить команду git checkout. Давайте перейдём на новую ветку testing:

```
$ git checkout testing
```

- Это действие передвинет HEAD так, чтобы тот указывал на ветку testing.

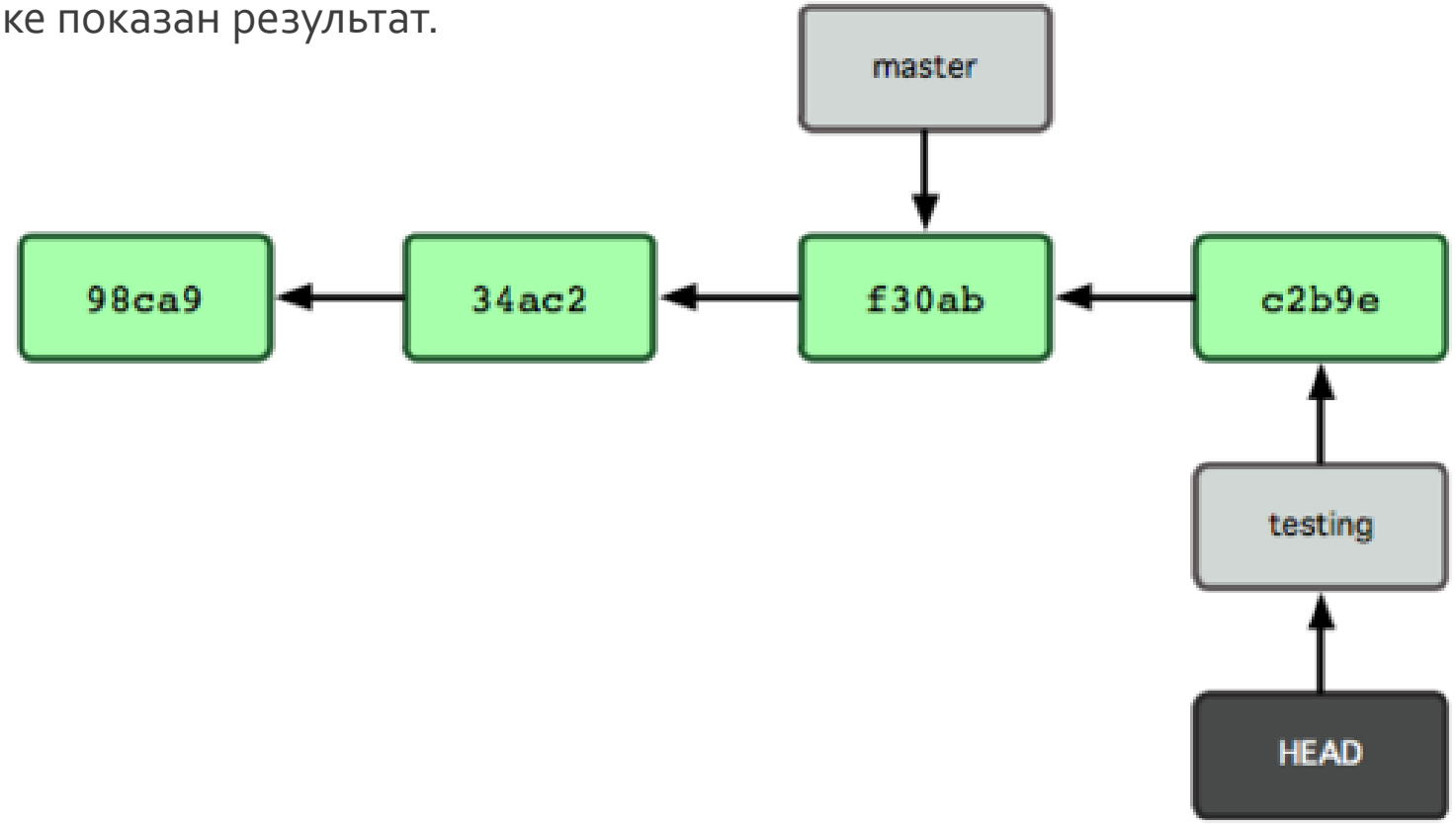


HEAD указывает на другую ветку после переключения веток.

- В чём же важность этого? Давайте сделаем ещё один коммит:

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

- На рисунке показан результат.

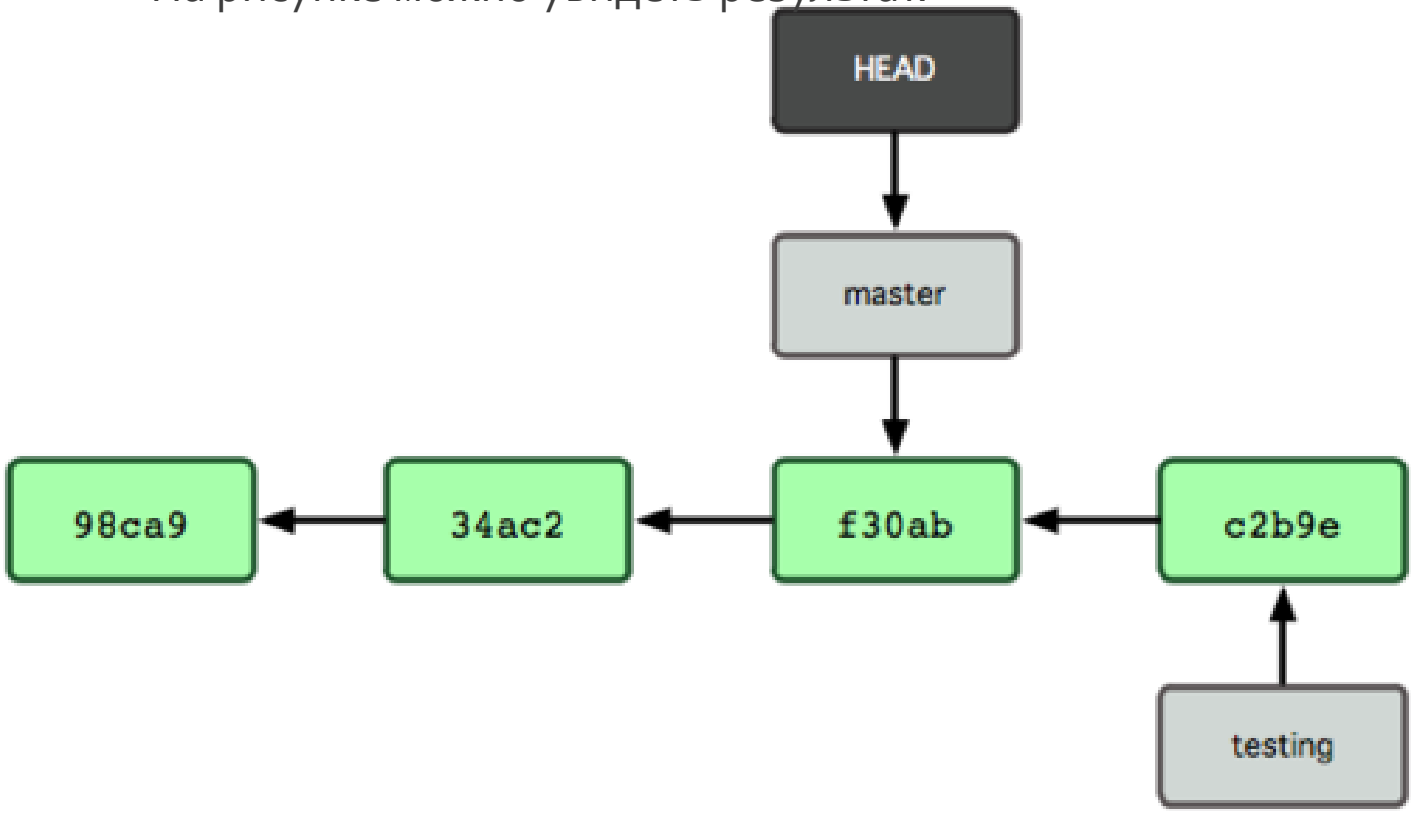


Ветка, на которую указывает HEAD, движется вперёд с каждым коммитом.

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

```
$ git checkout master
```

- На рисунке можно увидеть результат.



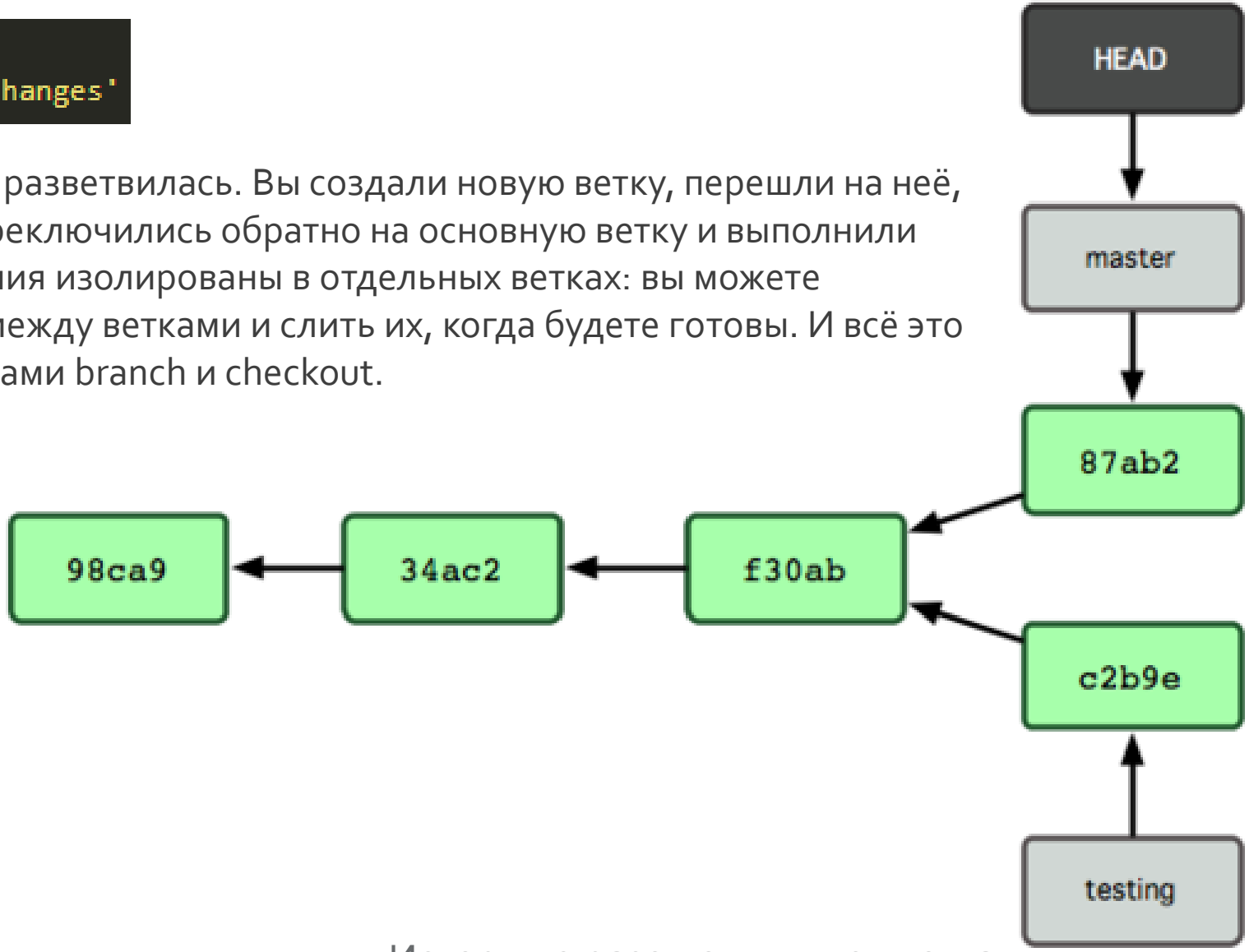
HEAD перемещается на другую ветку при checkout'е.

- 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, по сути, откатывает изменения, которые вы временно делали на ветке testing, так что дальше вы можете двигаться в другом направлении.

- Давайте снова внесём немного изменений и сделаем коммит:

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

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



История с разошедшимися ветками.

- Из-за того, что ветка в Git’е 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 быстрое и простое, как запись 41 байта в файл (40 символов + символ новой строки).
- Это разительно отличается от того, как в большинстве систем контроля версий 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’е это всегда происходит моментально.
- 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.

ОСНОВЫ ВЕТВЛЕНИЯ И СЛИЯНИЯ

Давайте рассмотрим ветвление и слияние на простом примере с таким процессом работы, который вы могли бы использовать в настоящей разработке. Мы выполним следующие шаги:

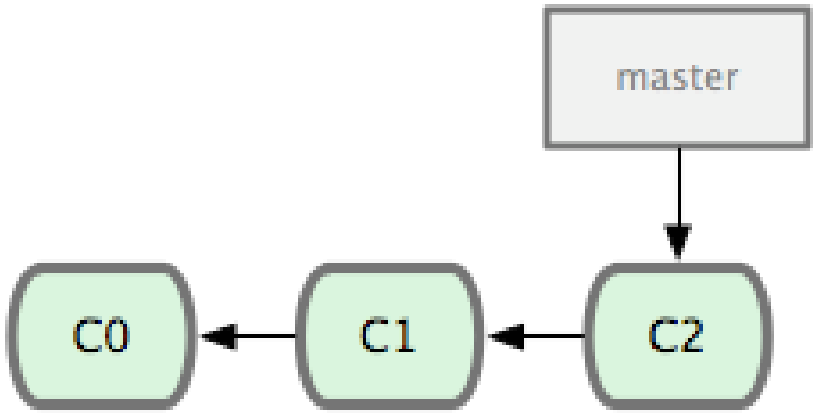
- Поработаем над веб-сайтом.
- Создадим ветку для работы над новой задачей.
- Выполним некоторую работу на этой ветке.

На этом этапе вам поступит звонок о том, что сейчас критична другая проблема, и её надо срочно решить. Мы сделаем следующее:

- Вернёмся на ветку для версии в производстве.
- Создадим ветку для исправления ошибки.
- После тестирования ветки с исправлением сольём её обратно и отправим в продакшн.
- Вернёмся к своей исходной задаче и продолжим работать над ней.

ОСНОВЫ ВЕТВЛЕНИЯ

- Для начала представим, что вы работаете над своим проектом и уже имеете пару коммитов.



- 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. Так как проблема №53 является обособленной задачей, над которой вы собираетесь работать, мы создадим новую ветку и будем работать на ней.

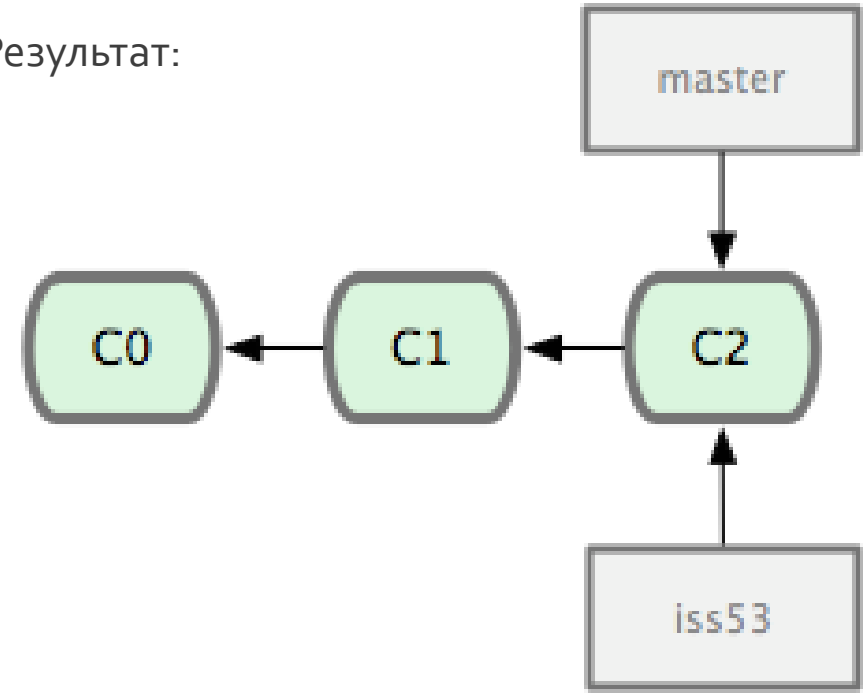
- Чтобы создать ветку и сразу же перейти на неё, вы можете выполнить команду git checkout с ключом -b:

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

- Это сокращение для:

```
$ git branch iss53
$ git checkout iss53
```

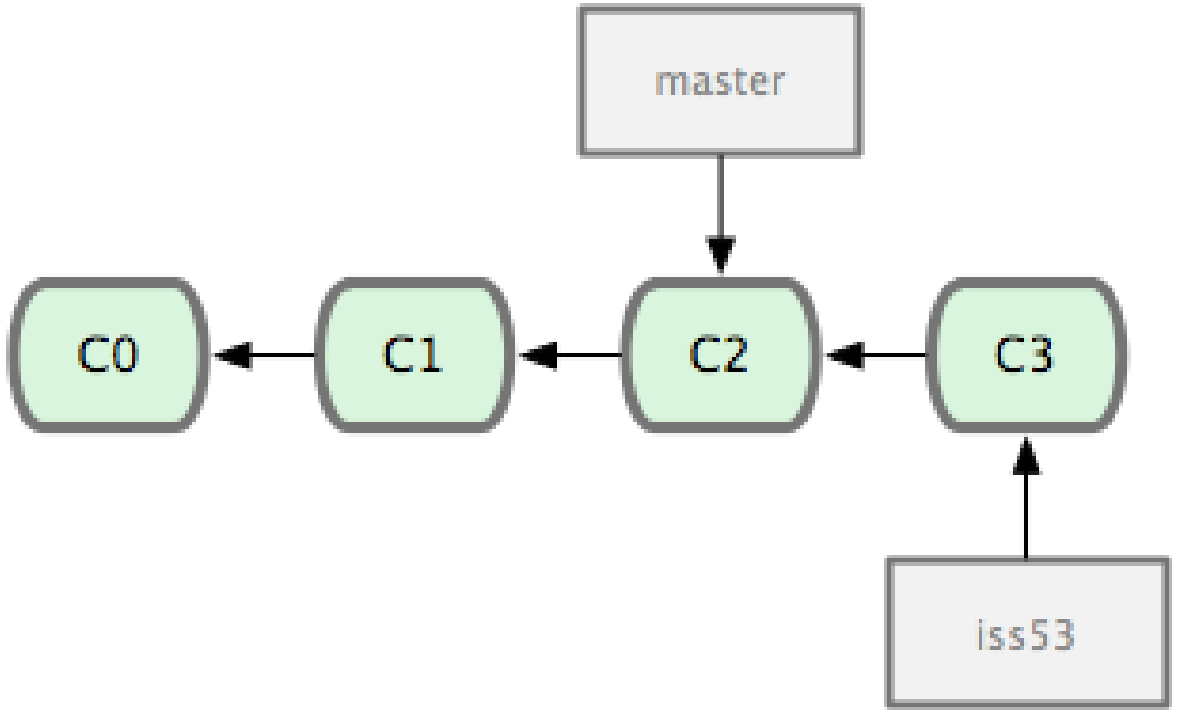
- Результат:



Создание новой ветки / указателя.

- Во время работы над своим 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:

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```



Ветка iss53 передвинулась вперёд во время работы.

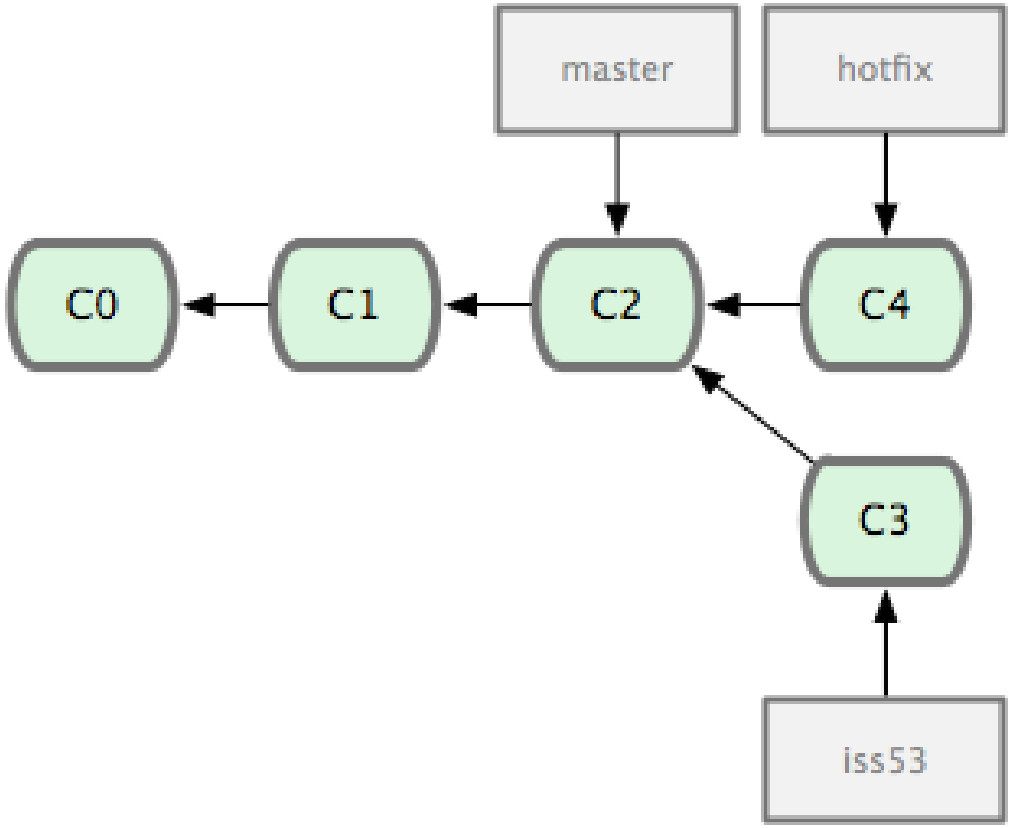
- Теперь вы получаете звонок о том, что есть проблема с веб-сайтом, которую необходимо немедленно устранить. С Git'ом вам нет нужды делать исправления для неё поверх тех изменений, которые вы уже сделали в iss53, и нет необходимости прикладывать много усилий для отмены этих изменений перед тем, как вы 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 все изменения были добавлены в коммит, и теперь вы 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 исправлении срочной проблемы. 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:

```
$ 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(-)
```



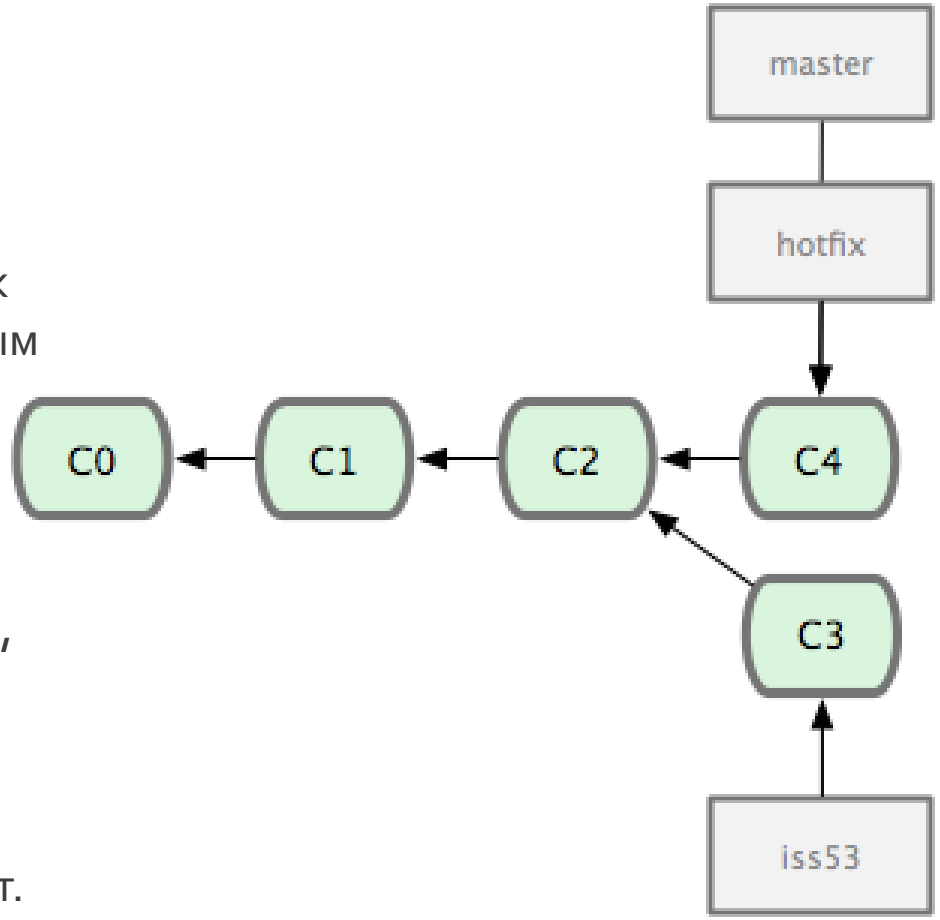
Ветка для решения срочной проблемы базируется на ветке master.

- Вы можете запустить тесты, убедиться, что решение работает, и слить (merge) изменения назад в ветку master, чтобы включить их 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 “перемотка” (fast forward).

- Ваши изменения теперь в снимке состояния коммита, на который указывает ветка master, и вы можете включить изменения в продукт.



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

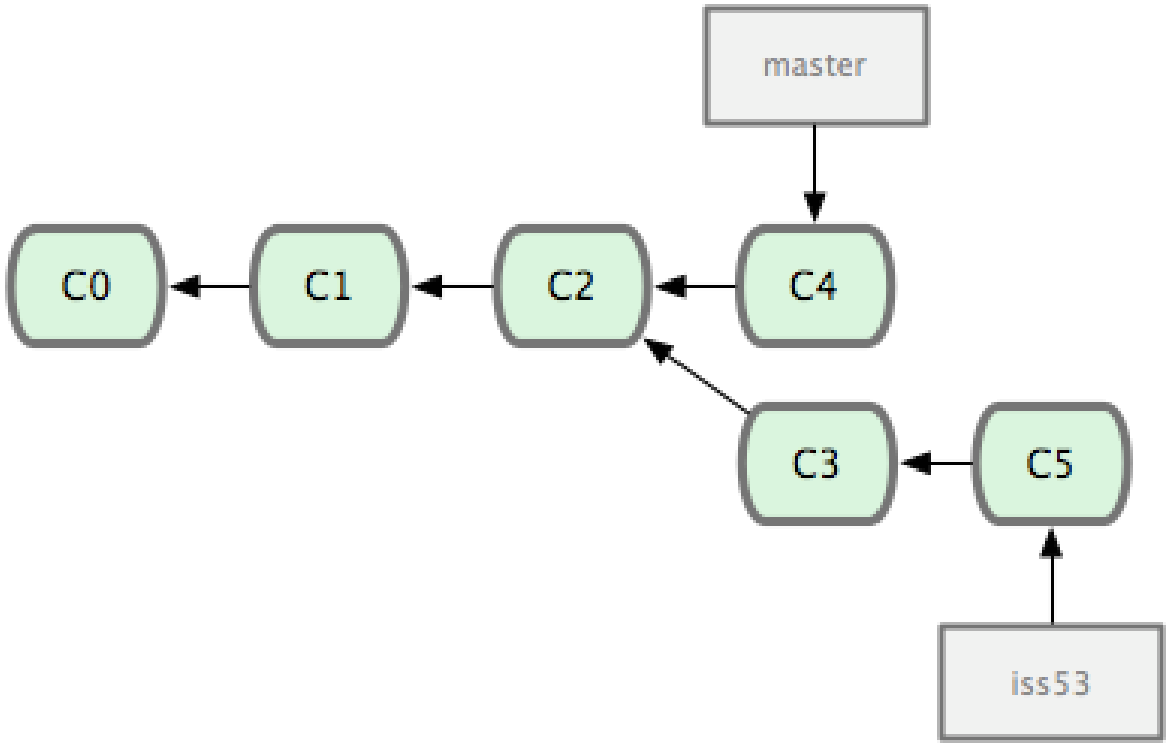
- После того как очень важная проблема решена, вы готовы вернуться обратно к тому, над чем вы работали перед тем, как вас прервали. Однако, сначала удалите ветку hotfix, так как она больше не нужна — ветка master уже указывает на то же место. Вы можете удалить ветку с помощью опции -d к git branch:

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

- Теперь вы можете вернуться обратно к рабочей ветке для проблемы №53 и продолжить работать над ней:

```
$ 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(-)
```

- Стоит напомнить, что работа, сделанная на ветке hotfix, не включена в файлы на ветке iss53. Если вам это необходимо, вы можете слить ветку 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.



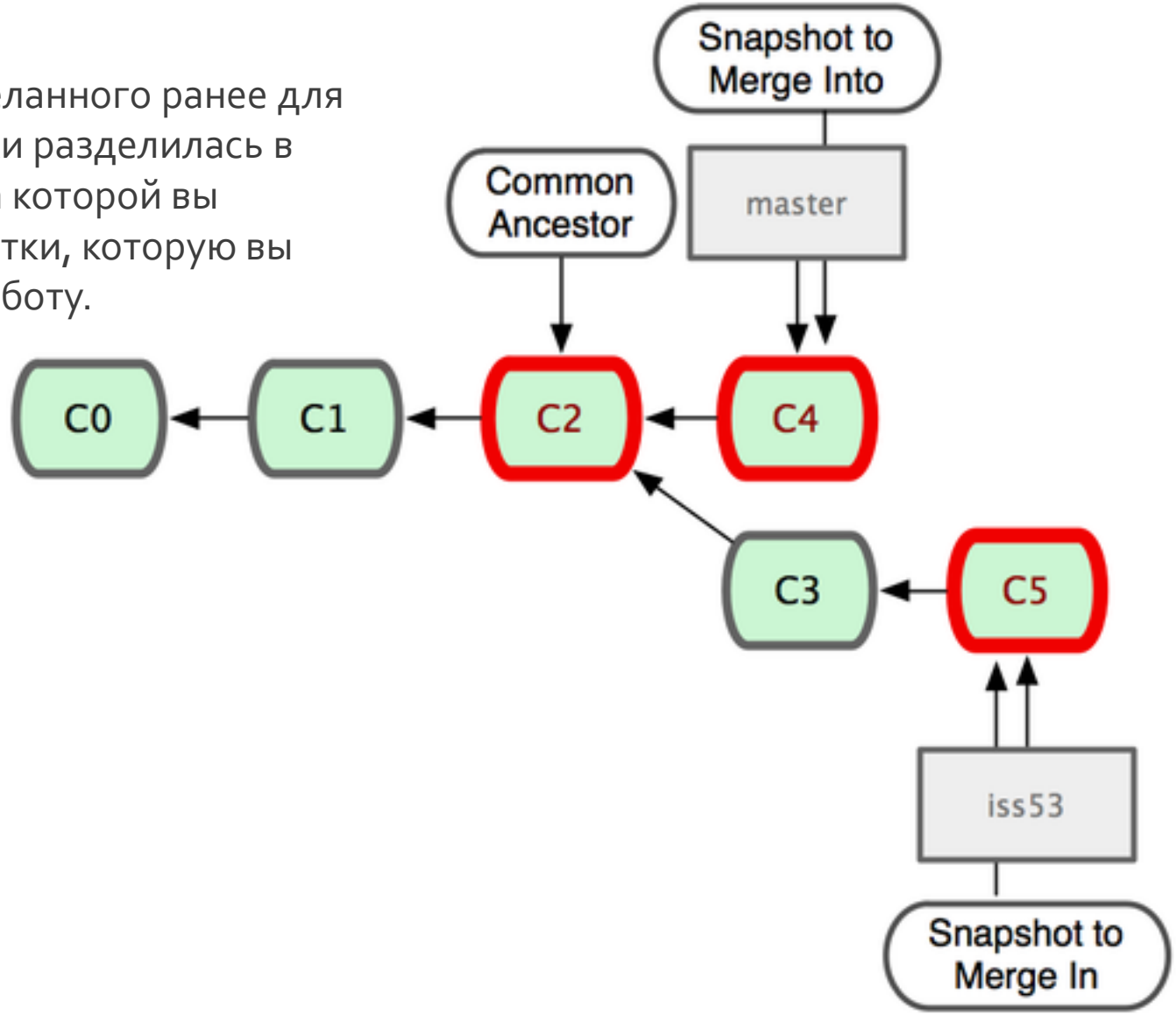
Ветка iss53 может двигаться вперёд независимо.

ОСНОВЫ СЛИЯНИЯ

- Допустим, вы разобрались с проблемой №53 и готовы объединить эту ветку и свой master. Чтобы сделать это, мы сольём ветку iss53 в ветку master точно так же, как мы делали это ранее с веткой hotfix. Всё, что вам нужно сделать, — перейти на ту ветку, в которую вы хотите слить свои изменения, и выполнить команду git merge:

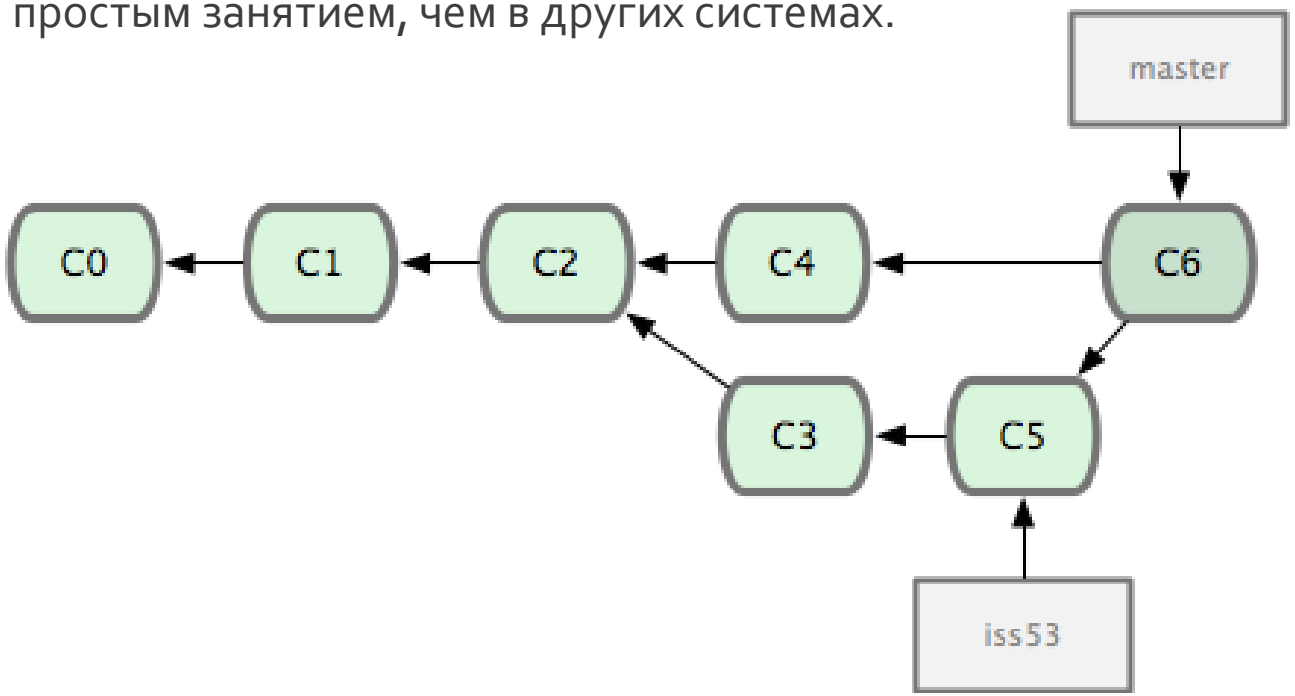
```
$ git checkout master
$ git merge iss53
Merge made by recursive.
 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

- Это слияние немного отличается от слияния, сделанного ранее для ветки hotfix. В данном случае 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 трёхходовое слияние, используя при этом те два снимка состояния репозитория, на которые указывают вершины веток, и общий для этих двух веток снимок-прародитель. На рисунке выделены три снимка состояния, которые Git будет использовать для слияния в данном случае.



Git автоматически определяет наилучшего общего предка для слияния веток.

- Вместо того чтобы просто передвинуть указатель ветки вперёд, Git создаёт новый снимок состояния, который является результатом трёхходового слияния, и автоматически создаёт новый коммит, который указывает на этот новый снимок состояния. Такой коммит называют коммит-слияние, так как он является особенным из-за того, что имеет больше одного предка.
- Стоит отметить, что Git сам 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'е гораздо более простым занятием, чем в других системах.



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

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

Git автоматически создаёт новый коммит, содержащий результаты слияния.

ОСНОВЫ КОНФЛИКТОВ ПРИ СЛИЯНИИ

- Иногда процесс слияния не идёт гладко. Если вы изменили одну и ту же часть файла по-разному в двух ветках, которые собираетесь слить, 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 так:

```
$ 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), выполните команду git status:

```
[master*]$ git status
index.html: needs merge
# On branch master
# Changes not staged for commit:
#   (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>
>>>>>>> iss53:index.html
```

- 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 разобрались с каждой из таких секций в каждом из конфликтных файлов, выполните git add для каждого конфликтного файла. Индексирование будет означать для Git'а, что все конфликты в файле теперь разрешены. Если вы хотите использовать графические инструменты для разрешения конфликтов, можете выполнить команду git mergetool, 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 в примере, так как команда была выполнена на Mac'е). Вы можете увидеть все поддерживаемые инструменты, указанные выше после "merge tool candidates". Укажите название предпочтительного для вас инструмента. В рамках следующих лекций 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 разрешён.
- 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, которые вам пригодятся, когда вы начнёте использовать ветки постоянно.
- Команда git branch делает несколько больше, чем просто создаёт и удаляет ветки. Если вы выполните её без аргументов, то получите простой список имеющихся у вас веток:

```
$ git branch
  iss53
* master
  testing
```

- Обратите внимание на символ *, стоящий перед веткой master: он указывает на 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
```

- Ещё одна полезная возможность для выяснения состояния веток состоит в том, чтобы оставить в этом списке только те ветки, которые вы слили (или не слили) в ветку, на которой сейчас находитесь. Для этих целей в Git'е 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
```

- Из-за того что мы ранее слили iss53, мы видим её в этом списке. Те ветки из этого списка, перед которыми нет символа *, можно смело удалять командой git branch -d; вы уже включили наработки из этих веток в другую ветку, так что вы ничего не потеряете.
- Чтобы увидеть все ветки, содержащие наработки, которые вы пока ещё не слили в текущую ветку, 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 её 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.

ПРИЁМЫ РАБОТЫ С ВЕТКАМИ

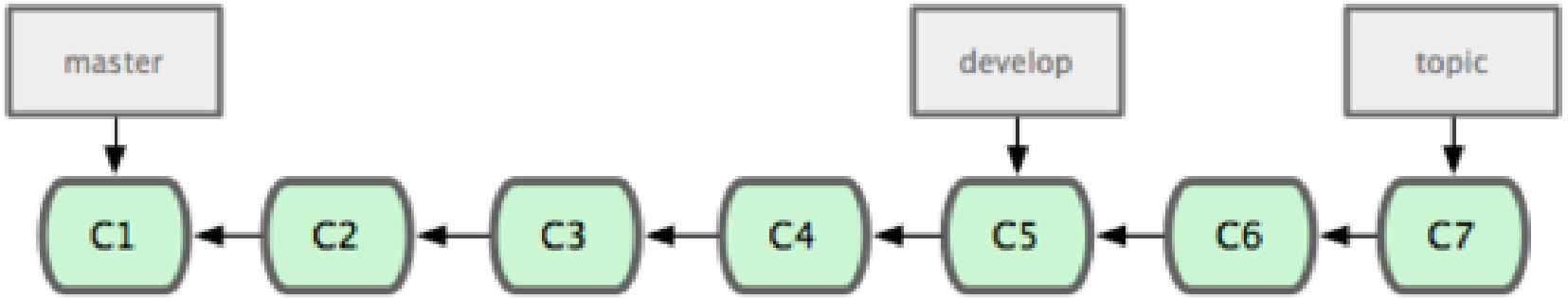
- Теперь, когда вы познакомились с основами ветвления и слияния, что вам делать с ветками дальше? В этом разделе мы рассмотрим некоторые стандартные приёмы работы, которые становятся возможными благодаря лёгкости осуществления ветвления. И вы сможете выбрать, включить ли вам какие-то из них в свой цикл разработки.

ДОЛГОЖИВУЩИЕ ВЕТКИ

- Так как Git использует простое трёхходовое 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 их одну в другую.

- 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.

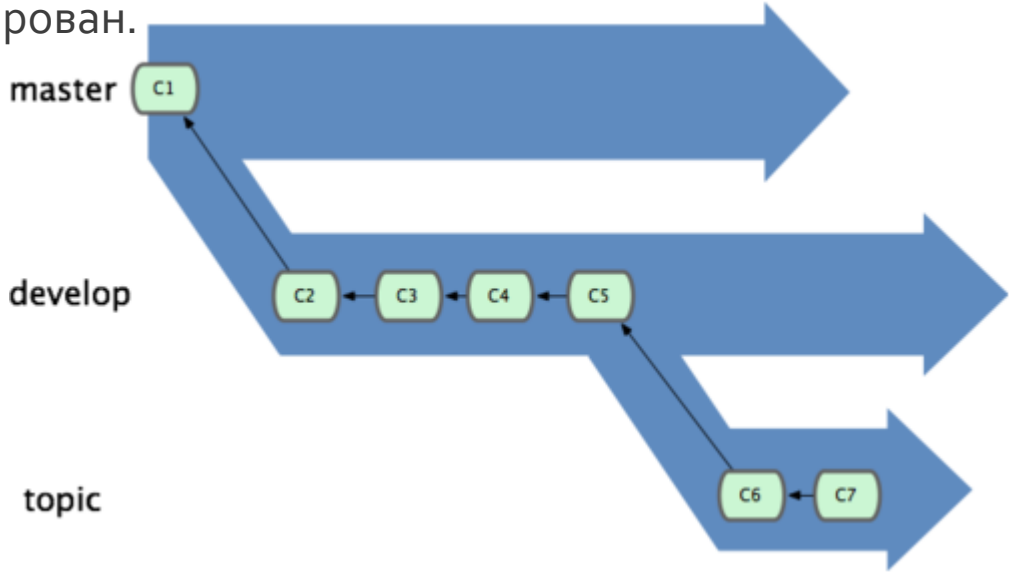
- В действительности же, мы говорим об указателях, передвигающихся вверх по линии коммитов, которые вы делаете. Стабильные ветки далеко внизу линии вашей истории коммитов, наиболее свежие ветки находятся ближе к верхушке этой линии.



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.

- 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, иметь долгоживущие ветки не обязательно, но зачастую это полезно, особенно когда вы имеете дело с очень большими и сложными проектами.

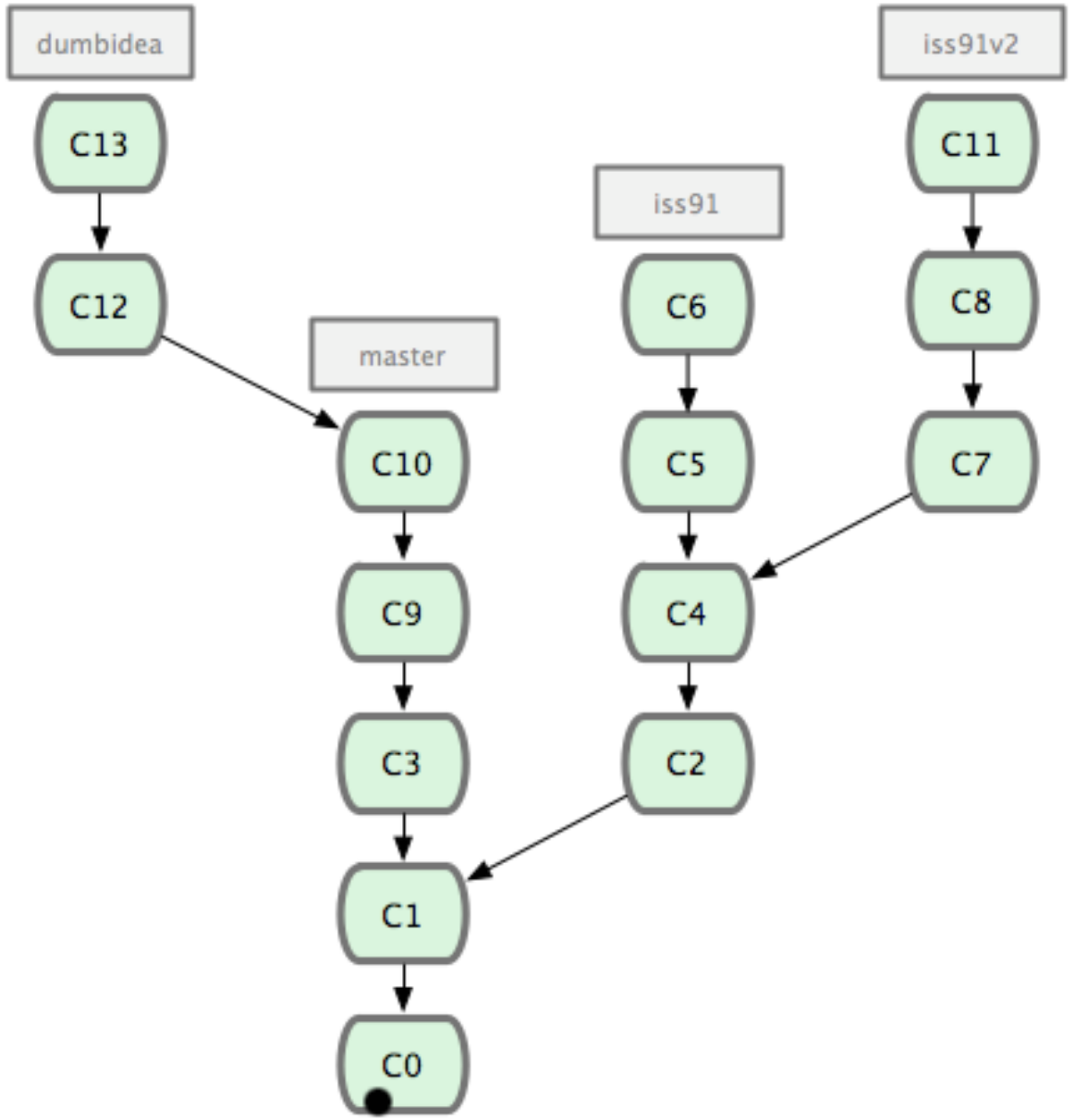


ТЕМАТИЧЕСКИЕ ВЕТКИ

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

- Мы видели подобное в последнем разделе, где вы создавали ветки iss53 и hotfix. Вы сделали всего несколько коммитов на этих ветках и удалили их сразу же после слияния с основной веткой. Такая техника позволяет быстро и полноценно переключать контекст. Ибо когда все изменения разбиты по веткам и определённым темам, намного проще понять, что было сделано, во время проверки и просмотра кода. Вы можете сохранить там изменения на несколько минут, дней или месяцев, а затем, когда они готовы, слить их в основную ветку, независимо от порядка, в котором их создавали или работали над ними.

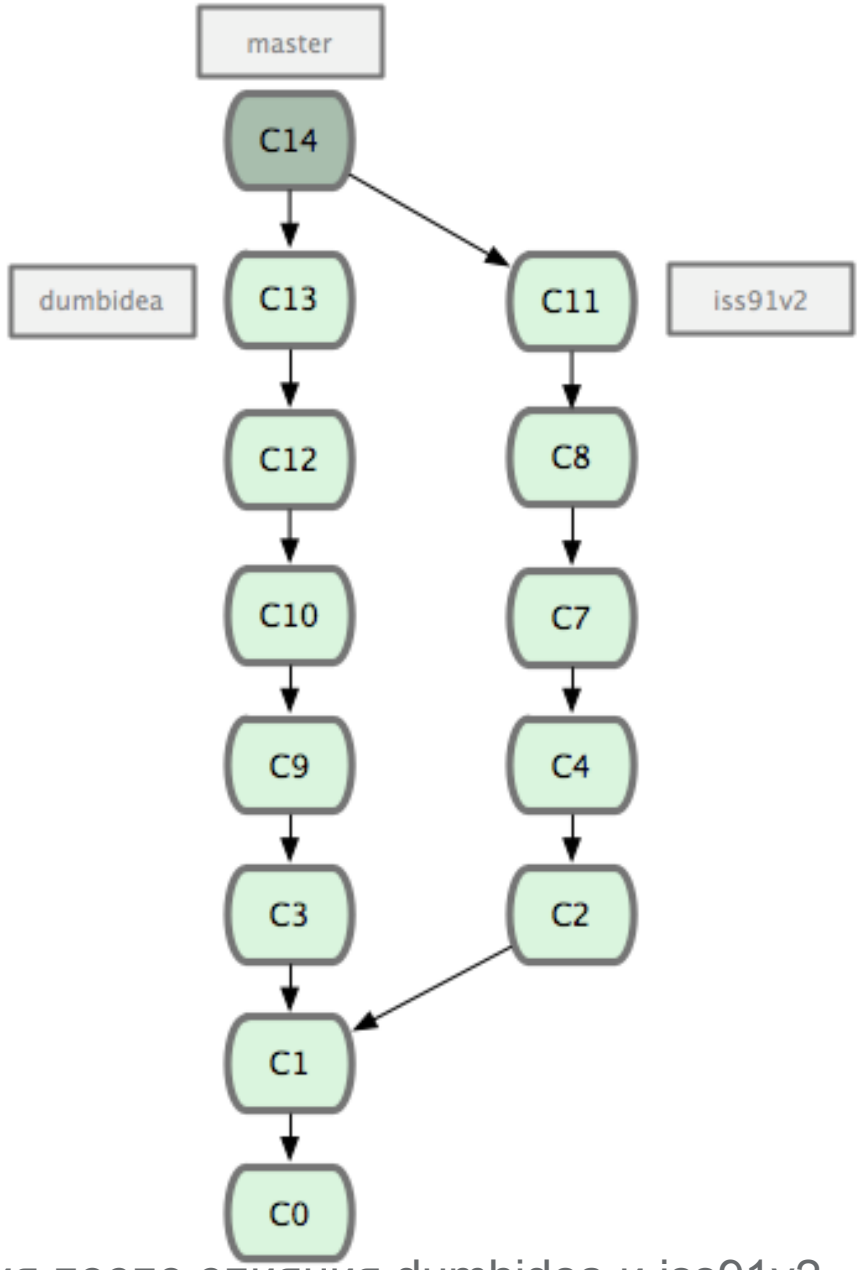
- Рассмотрим пример, когда при выполнении некоторой работы в ветке master, делается новая ветка для решения некой проблемы (iss91), выполняется немного работы на ней, от неё ответвляется ещё одна ветка для другого пути решения той же задачи (iss91v2), потом осуществляется переход назад на основную ветку (master), и некоторое время работа ведётся 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.



История коммитов с несколькими тематическими ветками.

- 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.

- 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.



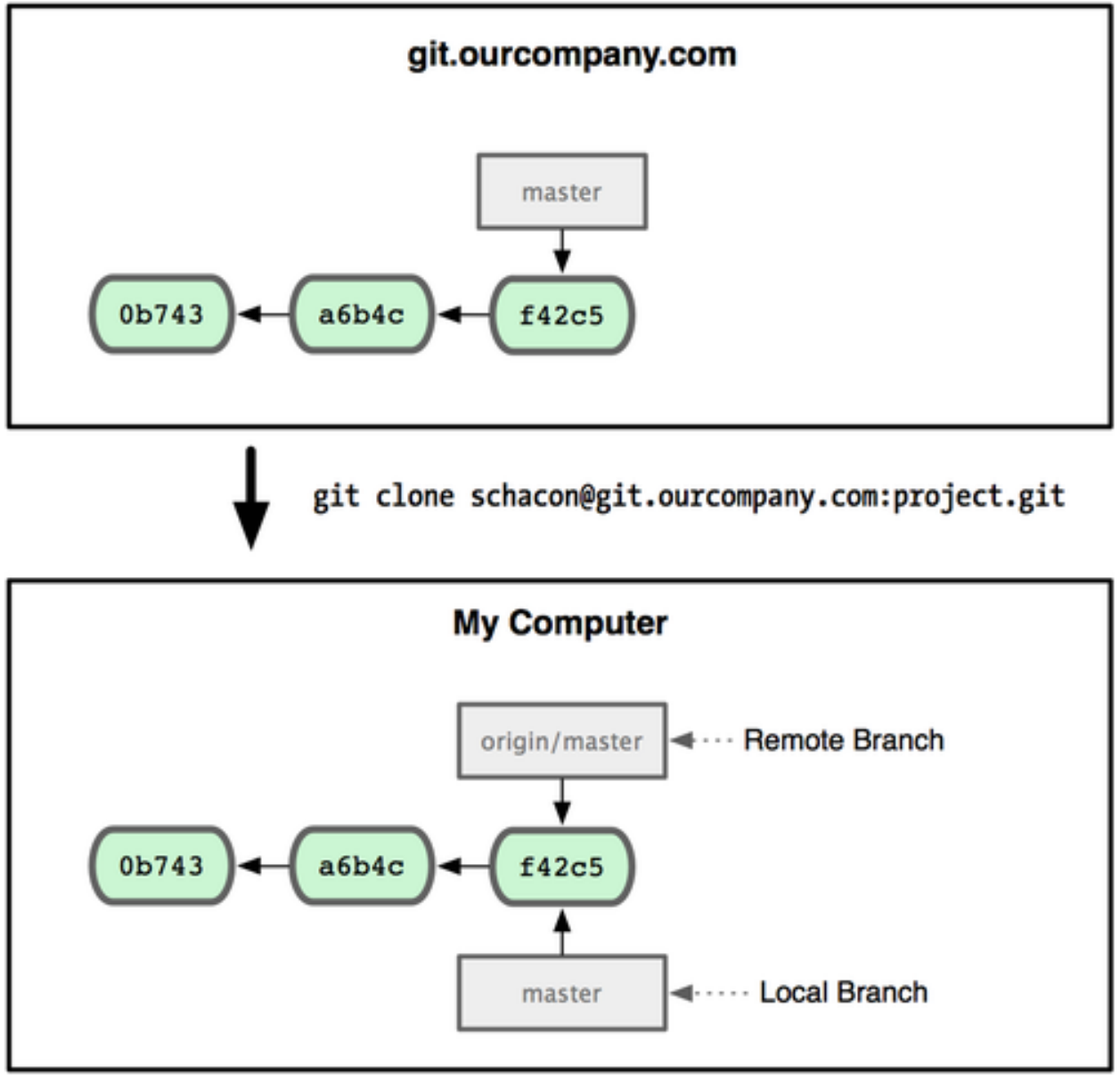
Ваша история после слияния dumbidea и iss91v2.

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.

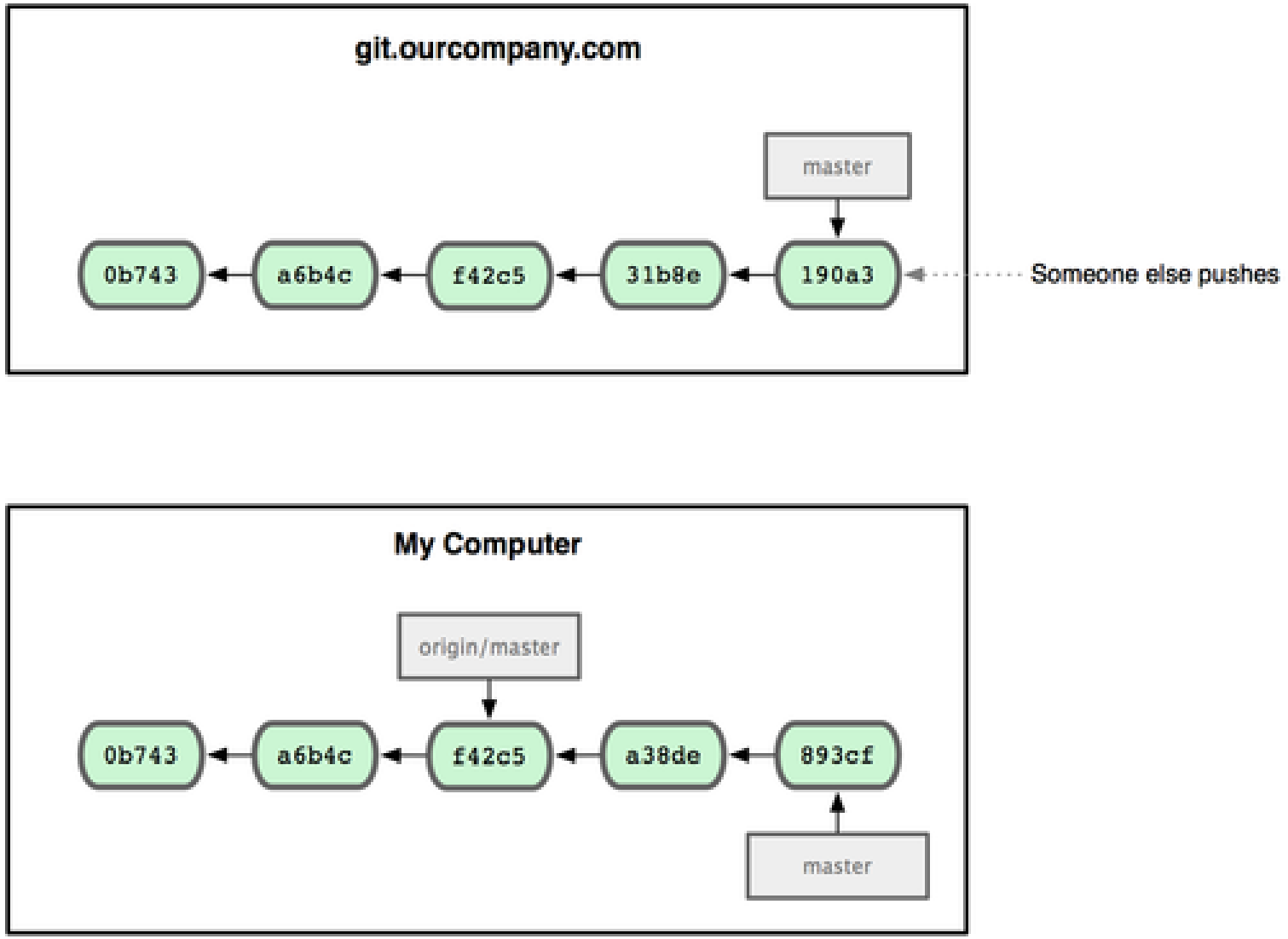
- Они выглядят как (имя удал. репоз.)/(ветка). Например, если вы хотите посмотреть, как выглядела ветка master на сервере origin во время последнего соединения с ним, проверьте ветку origin/master. Если вы с партнёром работали над одной проблемой, и он выложил ветку iss53, у вас может быть своя локальная 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 в сети есть свой Git-сервер на git.ourcompany.com. Если вы с него что-то склонируете (clone), Git автоматически назовёт его origin, заберёт оттуда все данные, создаст указатель на то, на что там указывает ветка 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 работать.



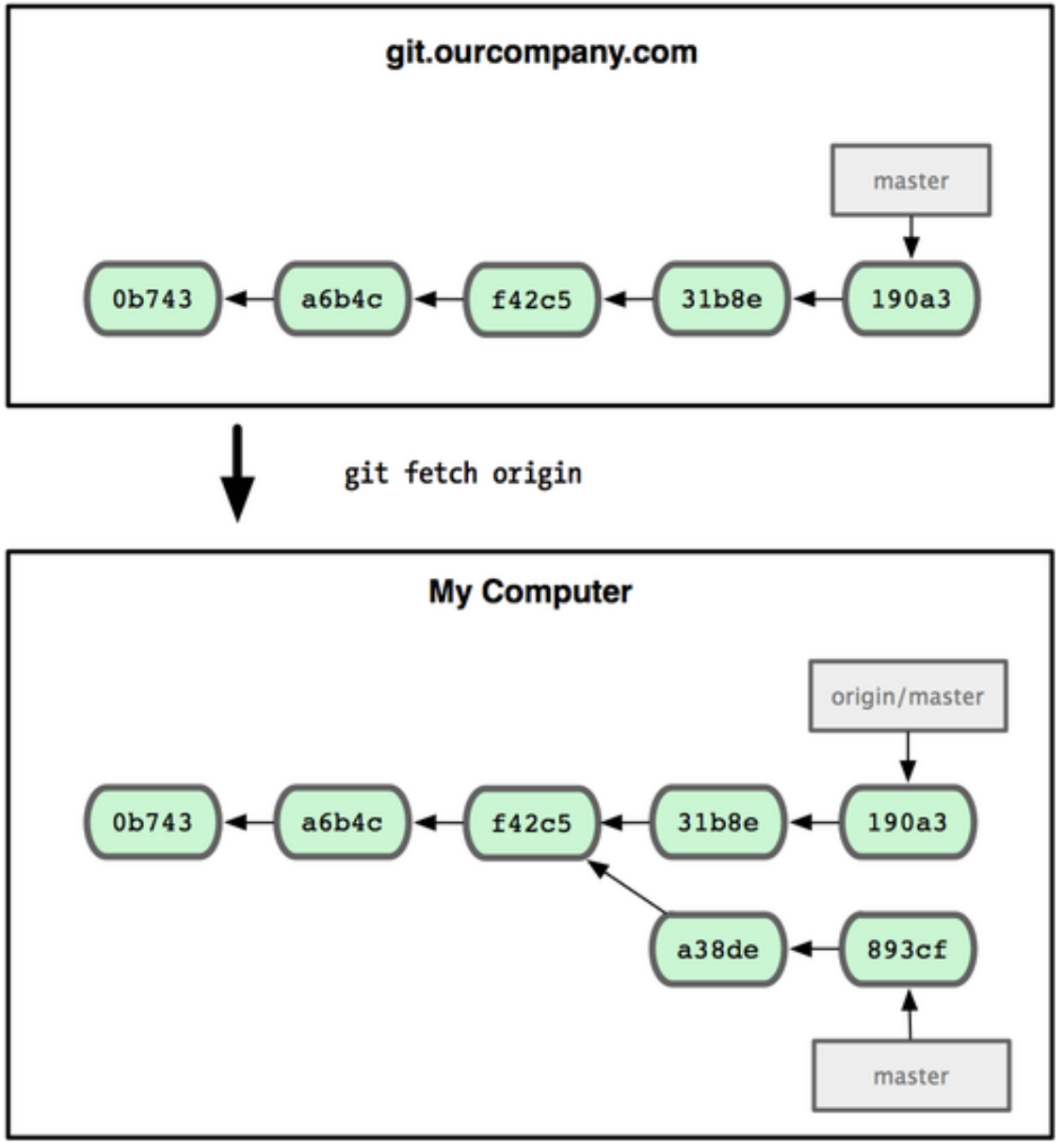
Клонирование Git-проекта даёт вам собственную ветку master и origin/master, указывающий на ветку master в origin.

- 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. Ещё, до тех пор, пока вы не свяжетесь с сервером origin, ваш указатель origin/master не будет сдвигаться.



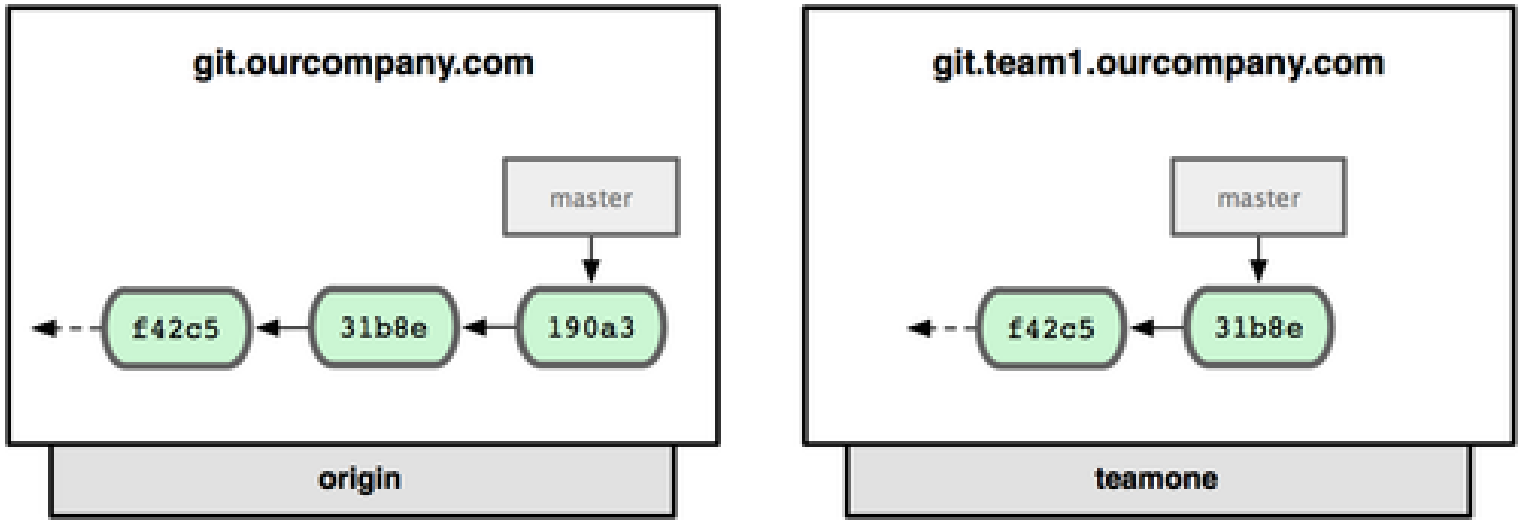
При выполнении локальной работы и отправке кем-то изменений на удалённый сервер каждая история продолжается по-разному.

- 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.

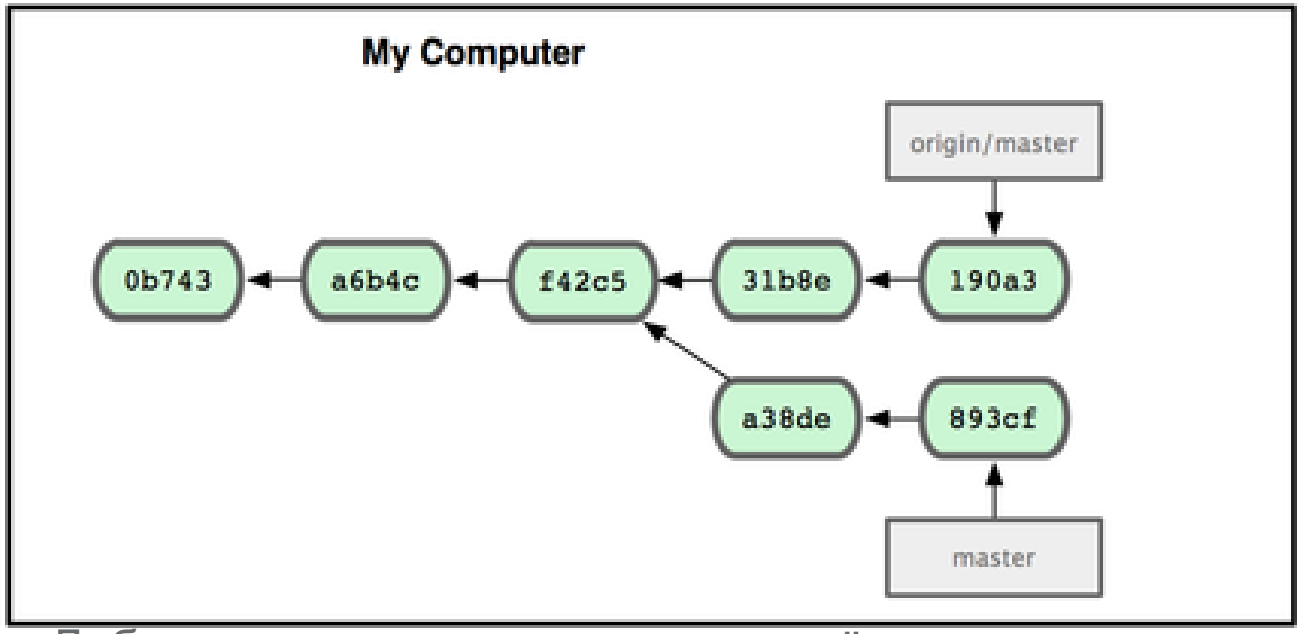


Команда git fetch обновляет ваши удалённые ссылки.

- 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 ранее. Дайте этому удалённому серверу имя teamone, которое будет сокращением для полного URL.

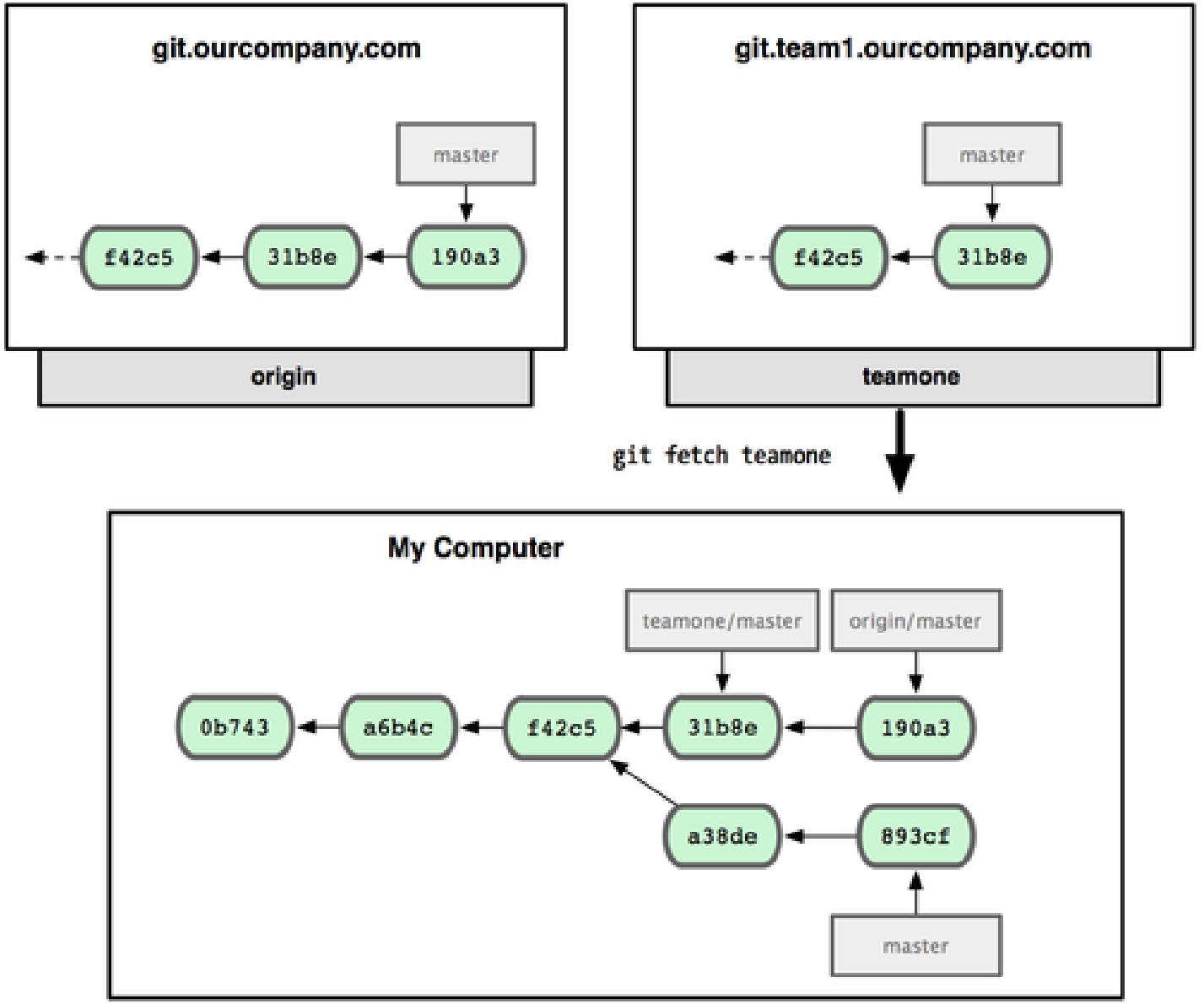



```
git remote add teamone git://git.team1.ourcompany.com
```



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.



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 serverfix до refs/heads/serverfix:refs/heads/serverfix, что означает “возьми мою локальную ветку serverfix и обнови из неё удалённую ветку serverfix”. Вы также можете выполнить git push origin serverfix:serverfix — произойдёт то же самое — здесь говорится “возьми мой serverfix и сделай его удалённым serverfix”. Можно использовать этот формат для отправки локальной ветки в удалённую ветку с другим именем. 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 origin/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 свою текущую рабочую ветку, выполните git merge origin/serverfix. Если вам нужна своя собственная ветка serverfix, над которой вы сможете работать, то вы 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
```

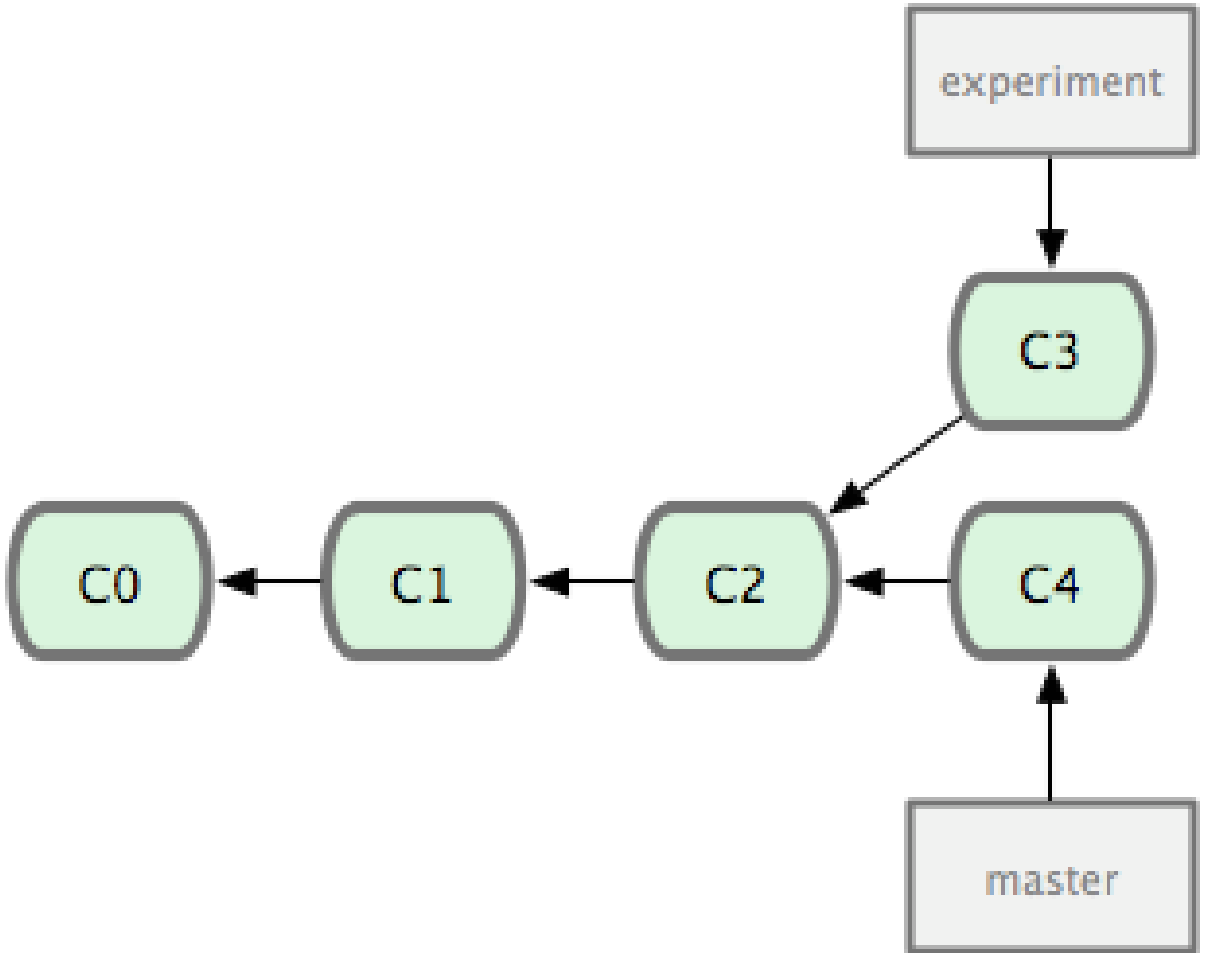
- Хлоп. Нет больше ветки на вашем сервере. Можно запомнить эту команду вернувшись к синтаксису git push [удал. сервер] [лок. ветка]:[удал. ветка], который мы рассматривали немного раньше. Опуская часть [лок. ветка], вы, по сути, говорите “возьми ничто в моём репозитории и сделай так, чтобы в [удал. ветка] было то же самое”.

ПЕРЕМЕЩЕНИЕ

- В Git’е 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.

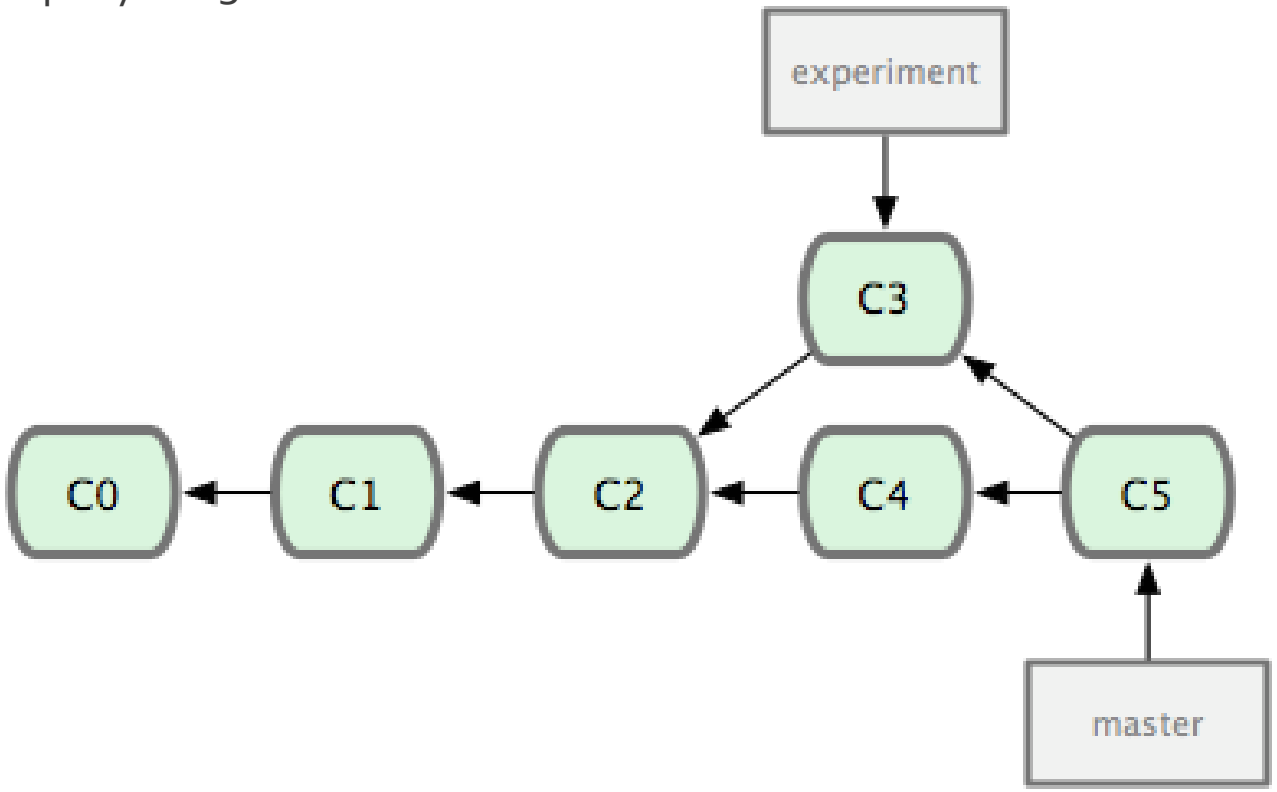
ОСНОВЫ ПЕРЕМЕЩЕНИЯ

- Если мы вернёмся назад к одному из ранних примеров из раздела про слияние, увидим, что мы разделили свою работу на два направления и сделали коммиты на двух разных ветках.



Впервые разделенная история коммитов.

- Наиболее простое решение для объединения веток, как мы уже выяснили, команда merge. Эта команда выполняет трёхходовое 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.

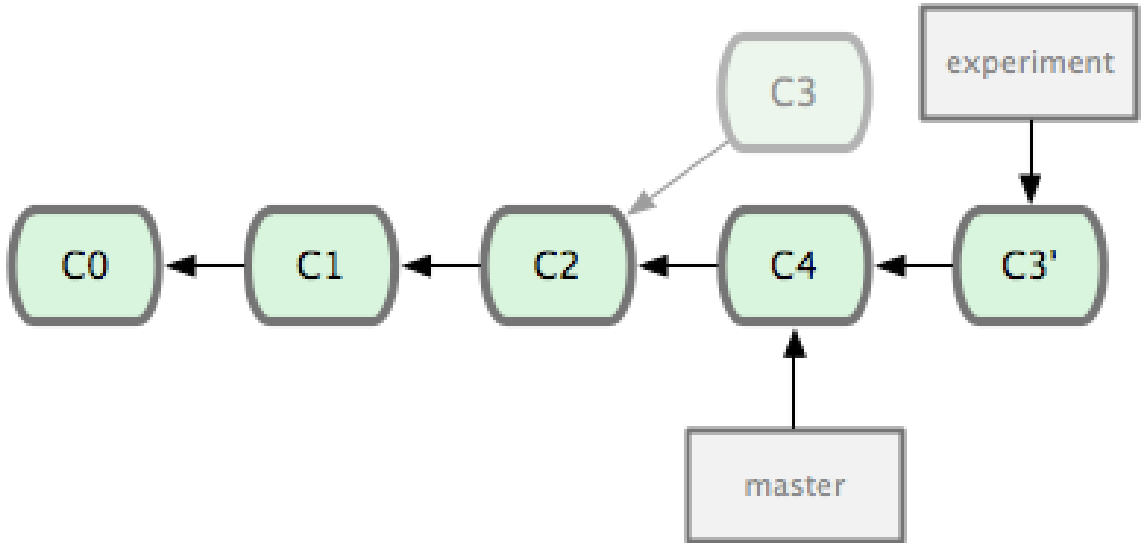


Слияние ветки для объединения разделившейся истории разработки.

- Однако, есть и другой путь: вы можете взять изменения, представленные в C3, и применить их поверх C4. В Git'е 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.
- Для этого примера надо выполнить следующее:

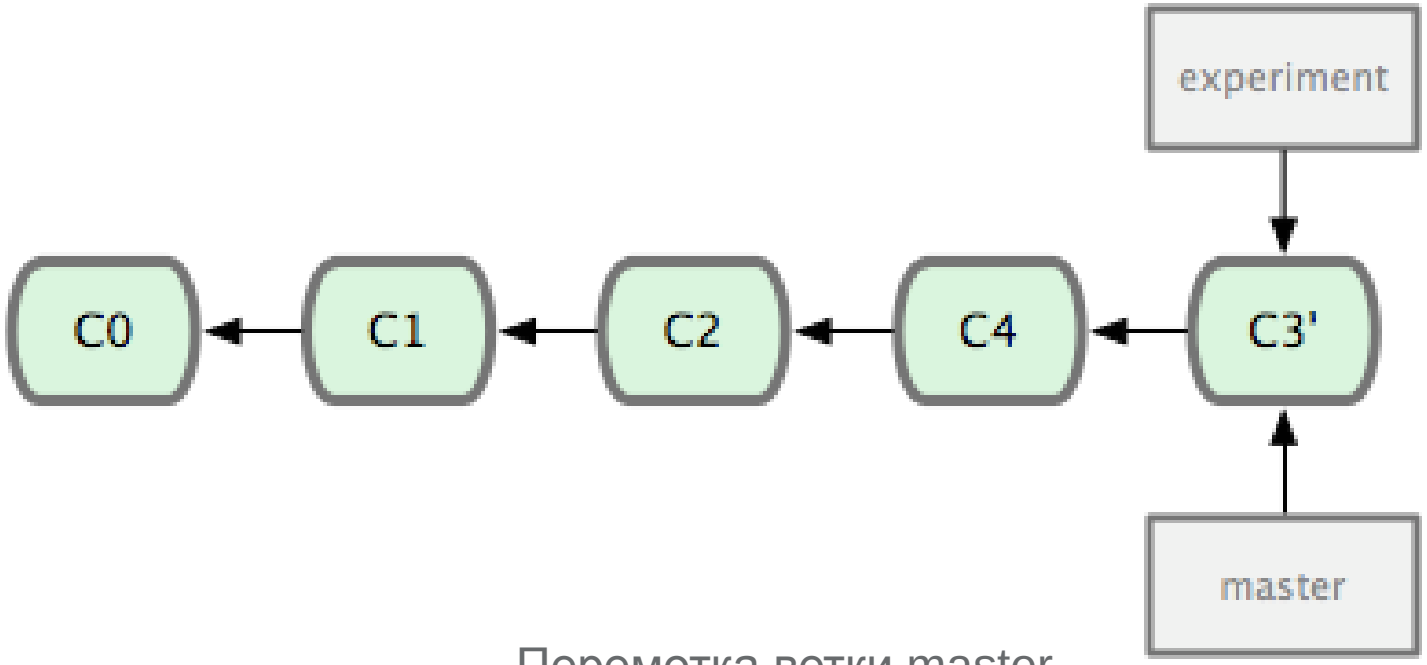
```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

- Перемещение работает следующим образом: находится общий предок для двух веток (на которой вы находитесь сейчас и на которую вы выполняете перемещение); для каждого из коммитов в текущей ветке берётся его дельта и сохраняется во временный файл; текущая ветка устанавливается на тот же коммит, что и ветка, на которую выполняется перемещение; и, наконец, одно за другим применяются все изменения. Рисунок иллюстрирует этот процесс.



Перемещение изменений, сделанных в C3, на C4.

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



Перемотка ветки master.

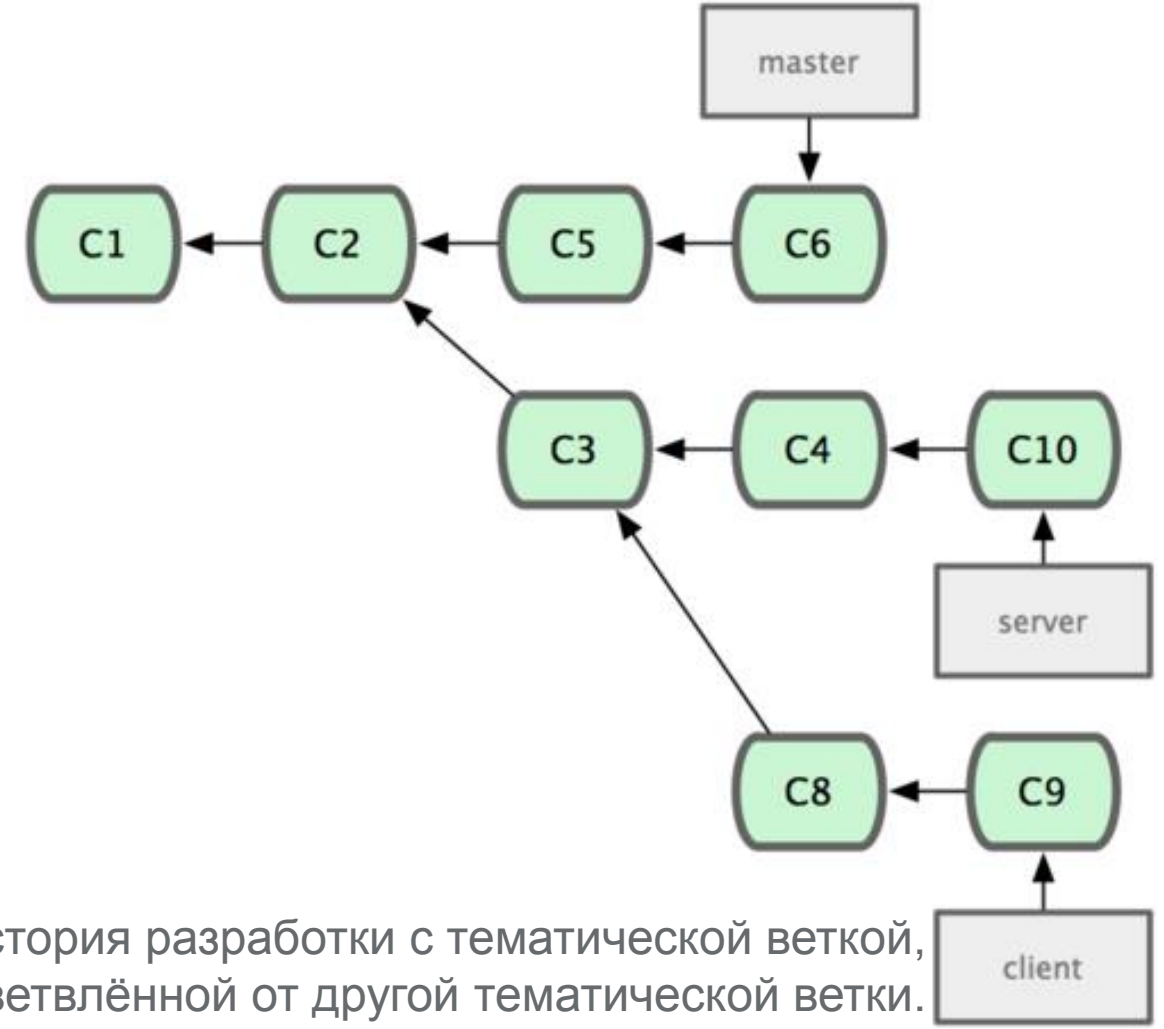
- Теперь снимок состояния, на который указывает C3′, точно такой же, как тот, на который указывал C5 в примере со слиянием. Нет никакой разницы в конечном результате объединения, но перемещение выполняется для того, чтобы история была более аккуратной. Если вы посмотрите лог для перемещённой ветки, то 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 удалённых 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 какой-нибудь ветке, а затем, когда будете готовы внести свои изменения в основной проект, выполните перемещение вашей работы на origin/master. Таким образом, владельцу проекта не придётся 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 перемещённым 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

- Можно также сделать так, чтобы при перемещении воспроизведение коммитов начиналось не от той ветки, на которую делается перемещение. Возьмём, например, историю разработки: 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.

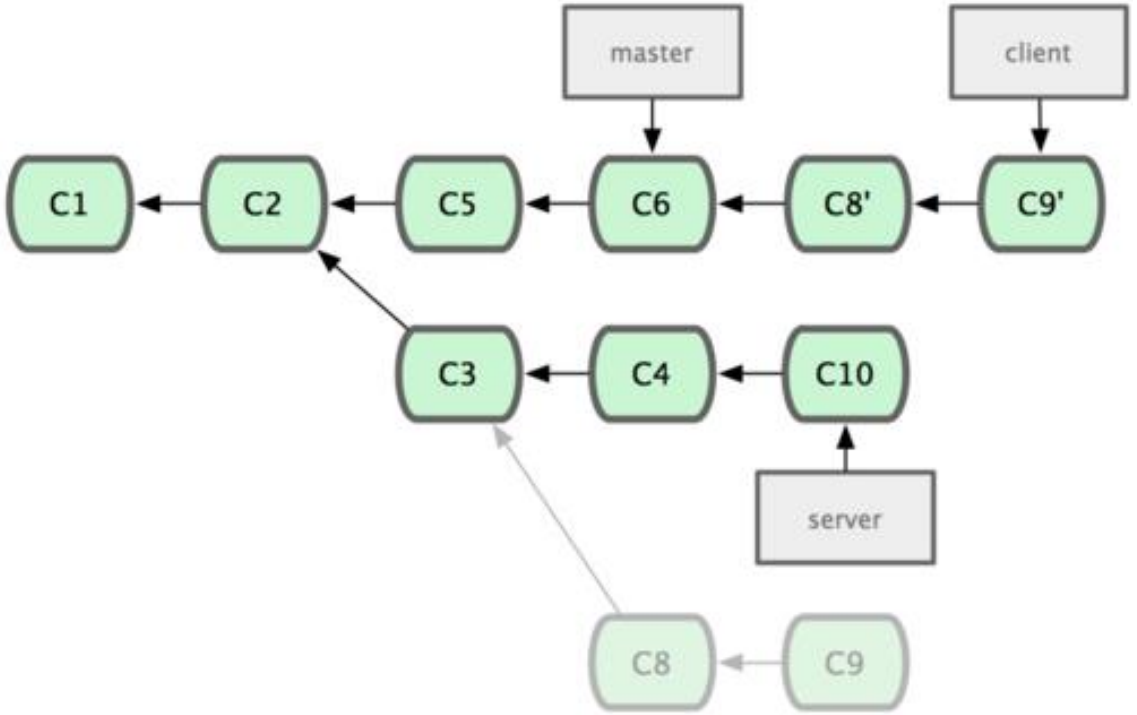


История разработки с тематической веткой, ответвлённой от другой тематической ветки.

- Предположим, вы решили, что хотите внести свои 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 в server (C8 и C9), и применить их на ветке master при помощи опции --onto команды git rebase:

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

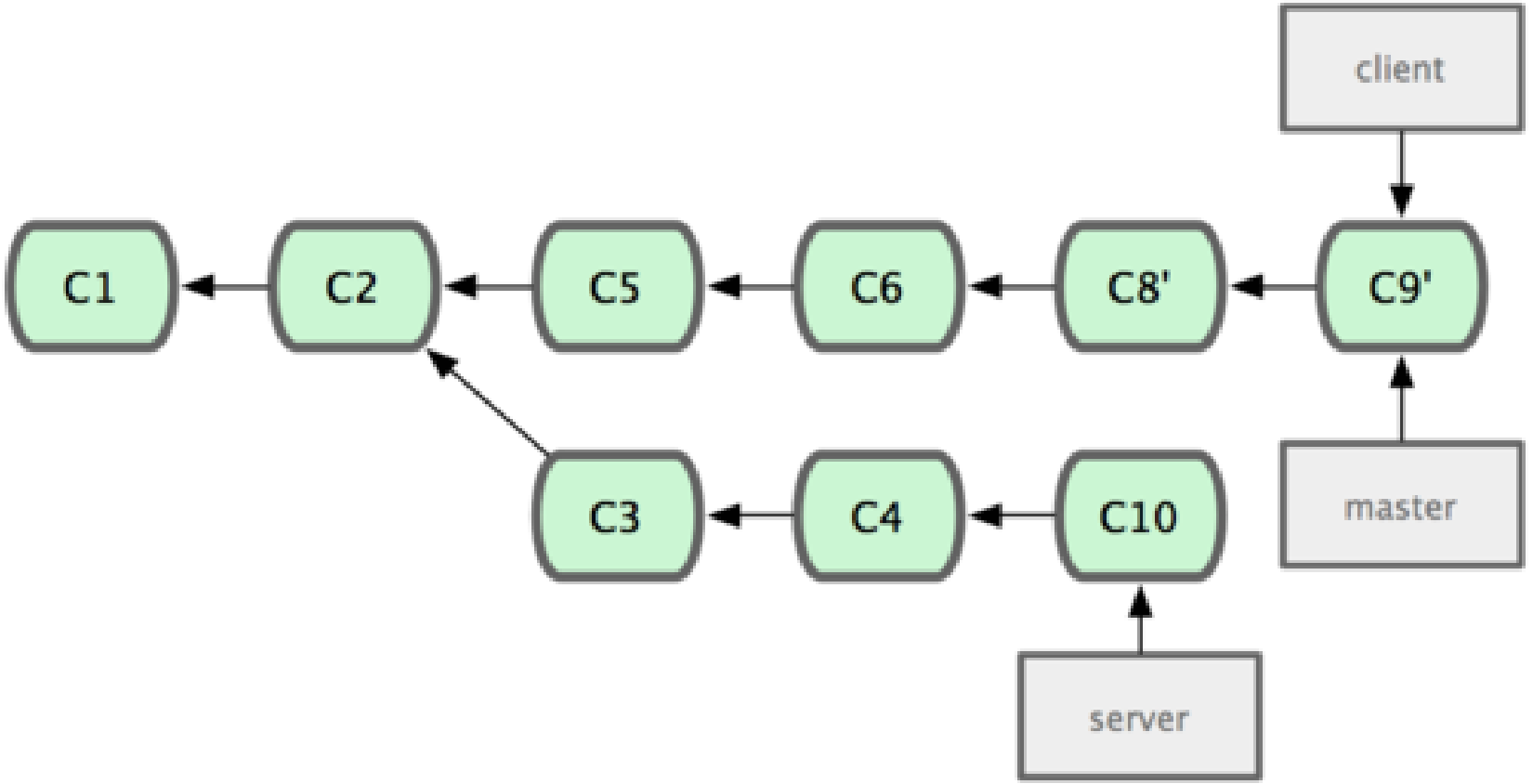
- По сути, это указание “переключиться на ветку client, взять изменения от общего предка веток client и server и повторить их на master”. Это немного сложно; но результат, показанный на рисунке, довольно классный.



Перемещение тематической ветки, ответвлённой от другой тематической ветки.

- Теперь вы можете выполнить перемотку (fast-forward) для ветки master:

```
$ git checkout master
$ git merge client
```

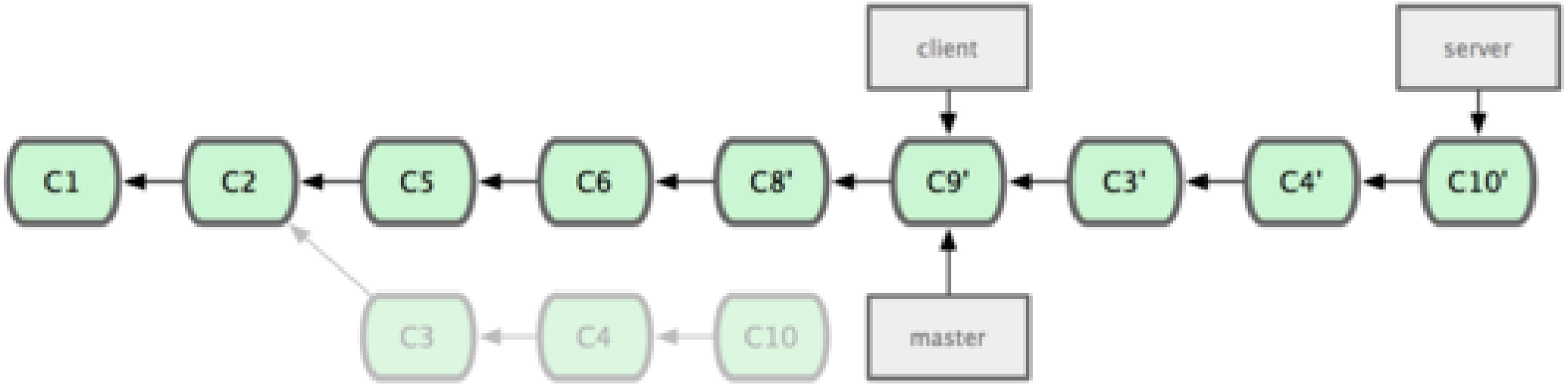


Перемотка ветки master для добавления изменений из ветки client.

- Представим, что вы решили включить работу и из ветки server тоже. 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.



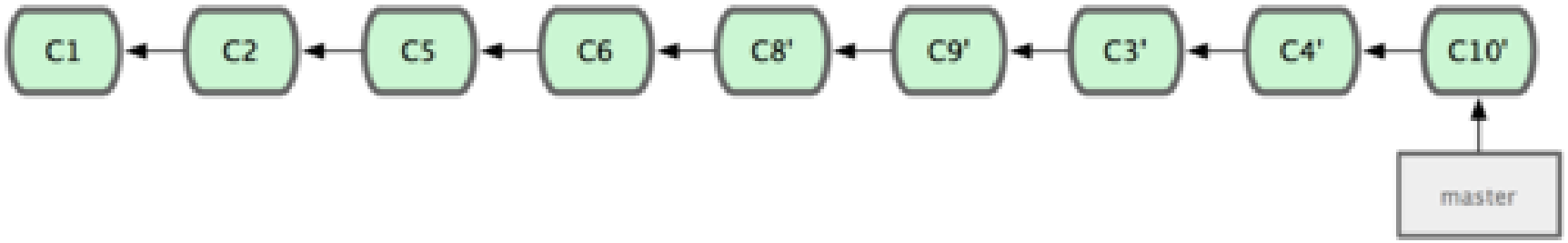
Перемещение ветки server на вершину ветки master.

- Затем вы можете выполнить перемотку 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:

```
$ git branch -d client
$ git branch -d server
```

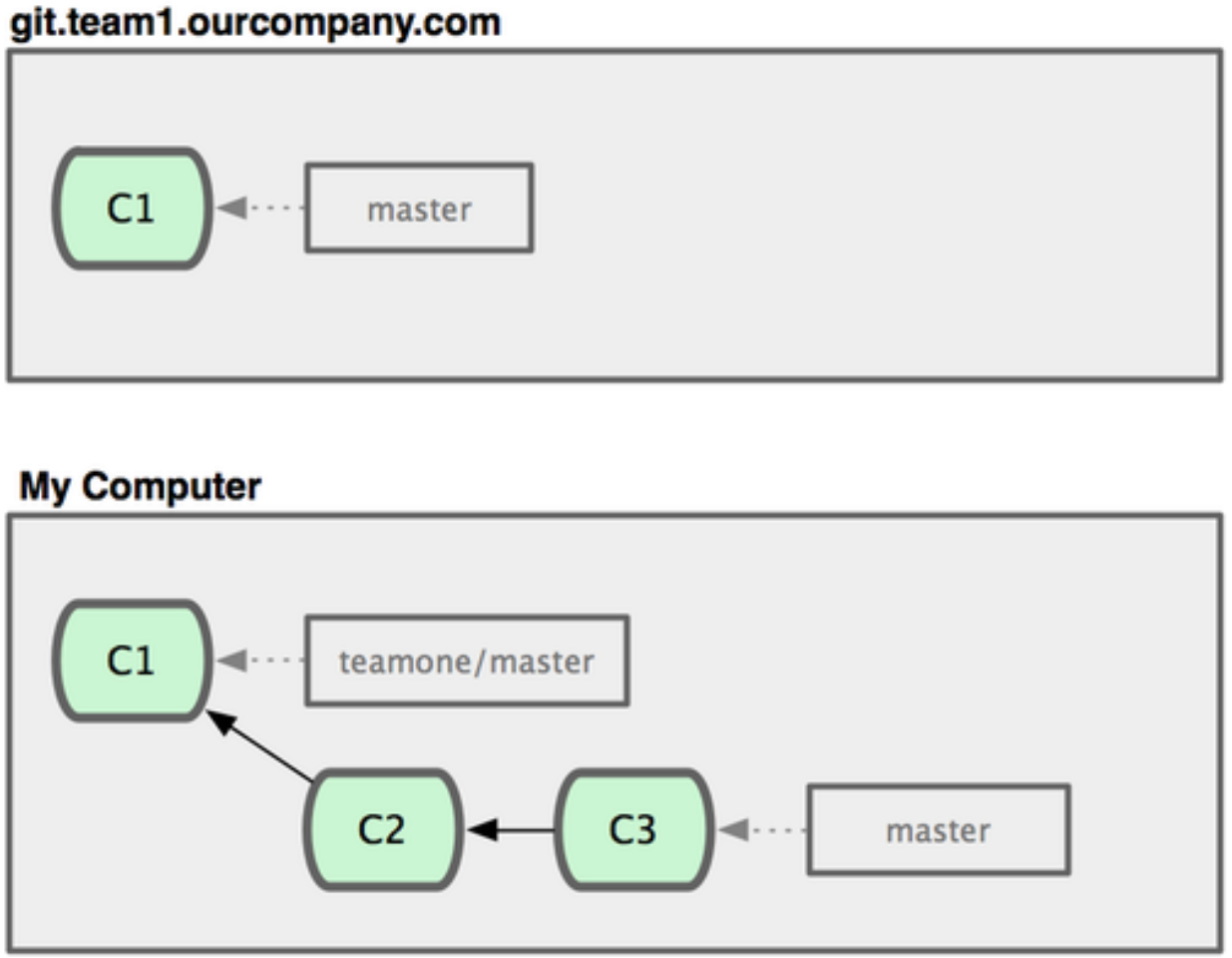


Финальная история коммитов.

ВОЗМОЖНЫЕ РИСКИ ПЕРЕМЕЩЕНИЯ

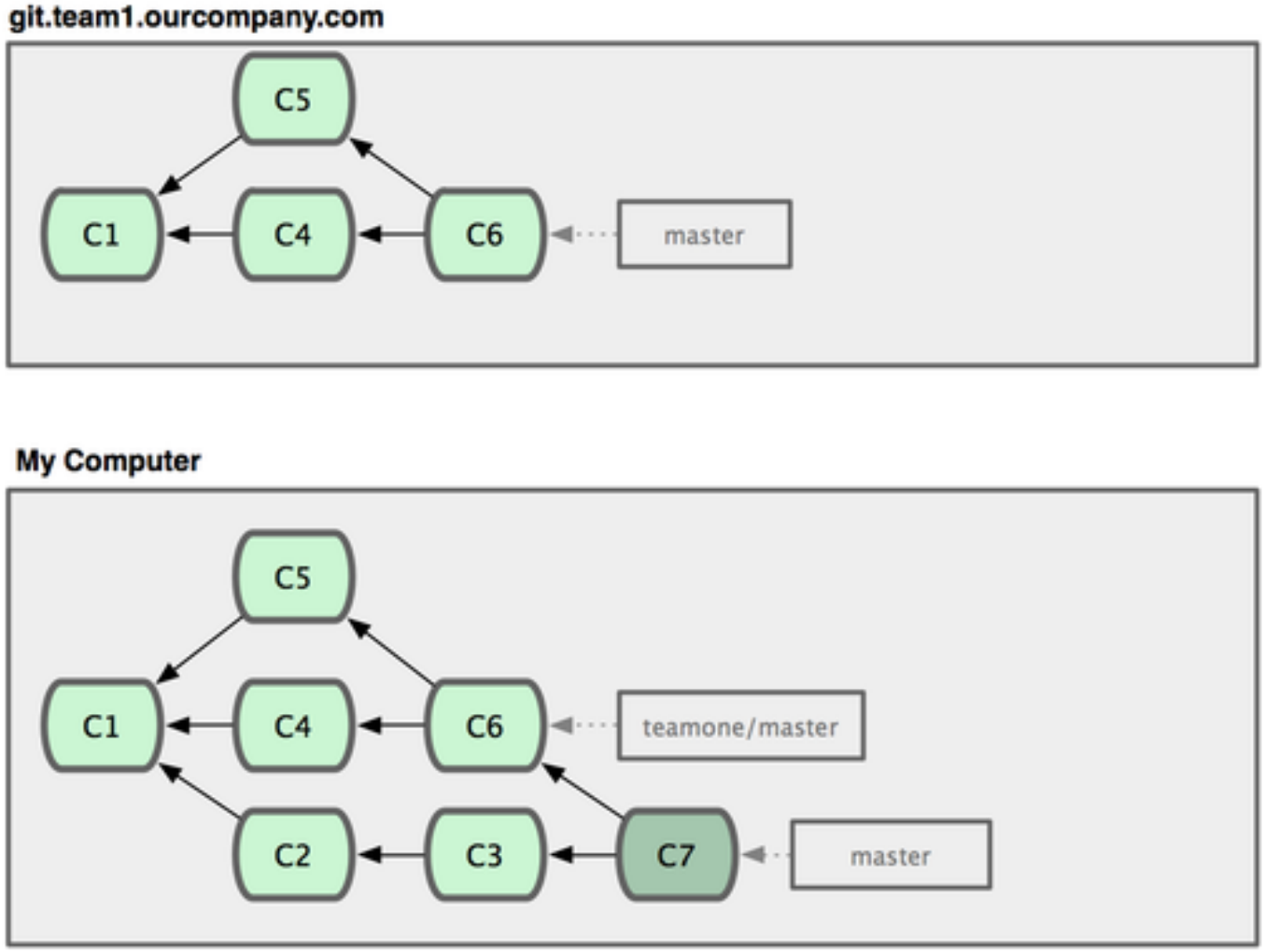
- Всё бы хорошо, но кое-что омрачает всю прелесть использования перемещения. Это выражается одной строчкой:
- **Не перемещайте коммиты, которые вы уже отправили в публичный репозиторий.**
- Если вы будете следовать этому указанию, всё будет хорошо. Если нет — люди возненавидят вас, вас будут презирать ваши друзья и семья.
- Когда вы что-то перемещаете, вы отменяете существующие коммиты и создаёте новые, которые похожи на старые, но являются другими. Если вы выкладываете (push) свои коммиты куда-нибудь, и другие забирают (pull) 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. В итоге вы получите путаницу, когда в очередной раз попытаетесь включить их работу в свою.

- Давайте рассмотрим пример того, как перемещение публично доступных наработок может вызвать проблемы. Представьте себе, что вы склонировали себе репозиторий с центрального сервера и поработали в нём. И ваша история коммитов выглядит как на рисунке.



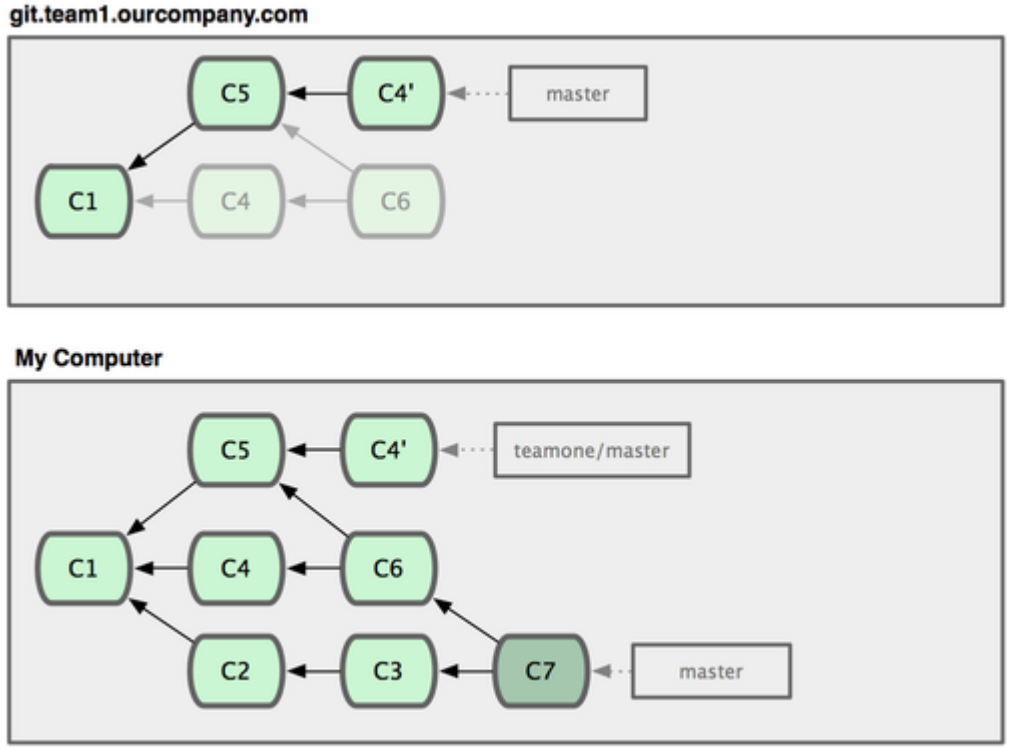
Клонирование репозитория и выполнение в нём какой-то работы.

- 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.



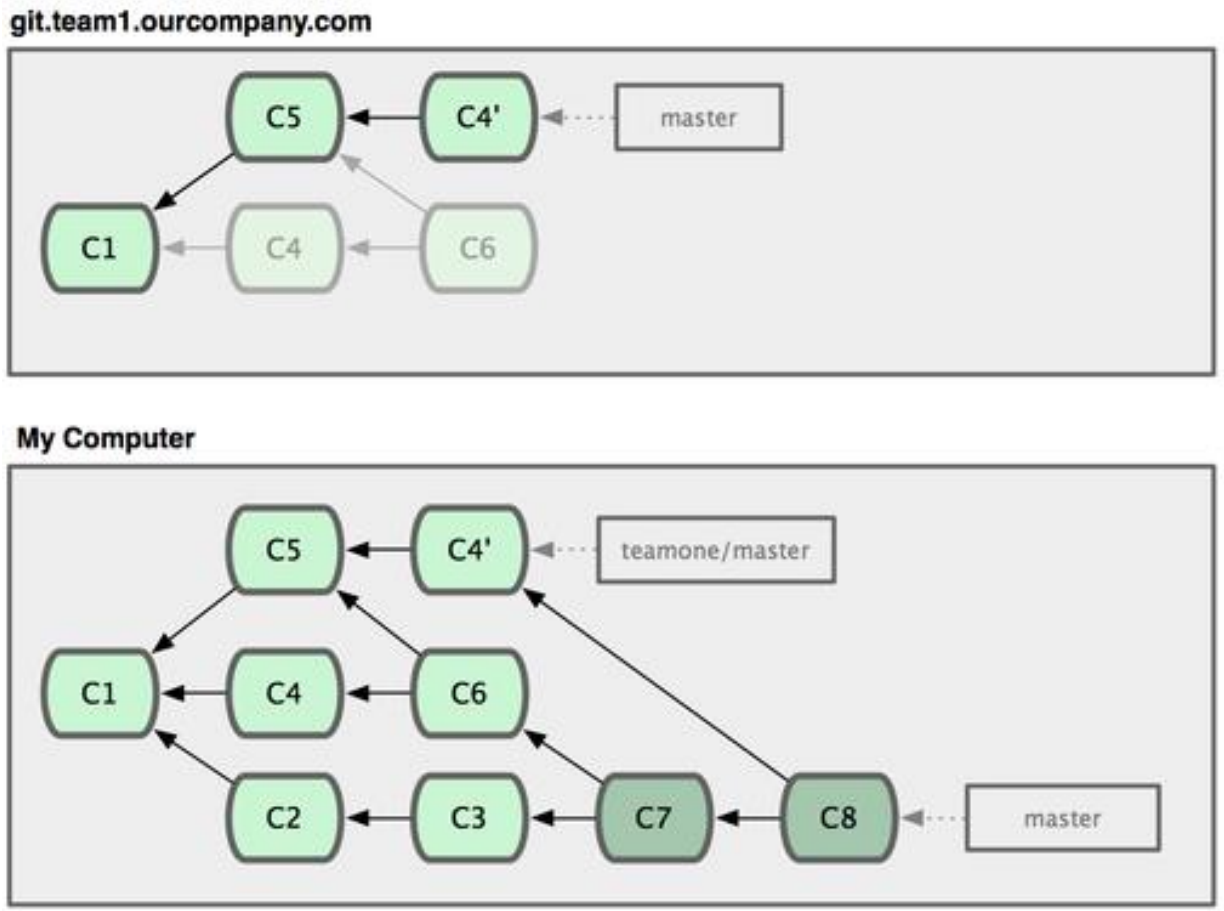
Извлечение коммитов и слияние их со своей работой.

- 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.



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'а они выглядят как новые коммиты, тогда как на самом деле вы уже располагаете наработками из C4 в своей истории.



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

- 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.

- 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.

ИНСТРУМЕНТЫ GIT

- К этому времени вы уже изучили большинство повседневных команд и способы организации рабочего процесса, необходимые для того, чтобы поддерживать Git-репозиторий для контроля вашего исходного кода. Вы выполнили основные задания, связанные с добавлением файлов под версионный контроль и записью сделанных изменений, и вы вооружились мощью подготовительной области (staging area), легковесного ветвления и слияния.

- Сейчас вы познакомитесь с множеством весьма сильных возможностей Git'а. Вы совсем не обязательно будете использовать их каждый день, но, возможно, в какой-то момент они вам понадобятся.

ВЫБОР РЕВИЗИИ

- Git позволяет указывать конкретные коммиты или их последовательности несколькими способами. Они не всегда очевидны, но иногда их полезно знать.
- **Одиночные ревизии**
- Вы можете просто сослаться на коммит по его SHA-1 хешу, но также существуют более понятные для человека способы ссылаться на коммиты. В этом разделе кратко описаны различные способы обратиться к одному определённому коммиту.
- **Сокращённый SHA**
- Git достаточно умён для того, чтобы понять какой коммит вы имеете в виду по первым нескольким символам (частичному хешу), конечно, если их не меньше четырёх и они однозначны, то есть если хеш только одного объекта в вашем репозитории начинается с этих символов.

- Предположим, что вы хотите посмотреть содержимое какого-то конкретного коммита. Вы выполняете команду git log и находите этот коммит (например, тот, в котором вы добавили какую-то функциональность):

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff
```

- В нашем случае выберем коммит 1c002dd….. При использовании git show для просмотра содержимого этого коммита следующие команды эквивалентны (предполагая, что сокращённые версии однозначны):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

- Git может показать короткие уникальные сокращения ваших SHA-1 хешей. Если вы передадите опцию --abbrev-commit команде git log, то её вывод будет использовать сокращённые значения, сохраняя их уникальными; по умолчанию будут использоваться семь символов, но при необходимости длина будет увеличена для сохранения однозначности хешей:

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

- В общем случае восемь-десять символов более чем достаточно для уникальности внутри проекта. В одном из самых больших проектов, использующих Git, ядре Linux только начинает появляться необходимость использовать 12 символов из 40 возможных для сохранения уникальности.

ССЫЛКИ НА ВЕТКИ

- Для самого прямого метода указать коммит необходимо, чтобы этот коммит имел ветку, ссылающуюся на него. Тогда вы можете использовать имя ветки в любой команде Git'а, которая ожидает коммит или значение SHA-1. Например, если вы хотите посмотреть последний коммит в ветке, следующие команды эквивалентны, предполагая, что ветка topic1 ссылается на ca82a6d:

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

- Чтобы посмотреть, на какой именно SHA указывает ветка, или понять для какого-то из приведённых примеров, к каким SHA он сводится, можно использовать служебную (plumbing) утилиту Git'а, которая называется rev-parse. В основном rev-parse нужна для выполнения низкоуровневых операций и не предназначена для использования в повседневной работе. Однако, она может пригодиться, если вам необходимо разобраться, что происходит на самом деле. Сейчас вы можете попробовать применить rev-parse к своей ветке.

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

REFLOG-СОКРАЩЕНИЯ

- Одна из вещей, которую Git делает в фоновом режиме, пока вы работаете, это запоминание ссылочного лога — лога того, где находились HEAD и ветки в течение последних нескольких месяцев.
- Ссылочный лог можно просмотреть с помощью git reflog:

```
$ git reflog
734713b... HEAD@{0}: commit: fixed refs handling, added gc auto, updated
d921970... HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
1c002dd... HEAD@{2}: commit: added some blame and merge stuff
1c36188... HEAD@{3}: rebase -i (squash): updating HEAD
95df984... HEAD@{4}: commit: # This is a combination of two commits.
1c36188... HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5... HEAD@{6}: rebase -i (pick): updating HEAD
```

- Каждый раз, когда верхушка ветки обновляется по какой-либо причине, Git сохраняет информацию об этом в эту временную историю. И вы можете использовать и эти данные для задания старых коммитов. Если вы хотите посмотреть, какое значение было у HEAD в вашем репозитории пять шагов назад, используйте ссылку вида @{n} так же, как показано в выводе команды reflog:

```
$ git show HEAD@{5}
```

- Также вы можете использовать эту команду, чтобы увидеть, где ветка была некоторое время назад. Например, чтобы увидеть, где была ветка master вчера, наберите

```
$ git show master@{yesterday}
```

- Эта команда покажет, где верхушка ветки находилась вчера. Такой подход работает только для данных, которые всё ещё находятся в ссылочном логе. Так что вы не сможете использовать его для коммитов с давностью в несколько месяцев.
- Чтобы просмотреть информацию ссылочного лога в таком же формате, как вывод git log, можно выполнить git log -g:

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

- Важно отметить, что информация в ссылочном логе строго локальная — это лог того, чем вы занимались со своим репозиторием. Ссылки не будут теми же самыми в чьей-то чужой копии репозитория; и после того как вы только что склонировали репозиторий, ссылочный лог будет пустым, так как вы ещё ничего не делали со своим репозиторием. Команда git show HEAD@{2.months.ago} сработает, только если вы склонировали свой проект как минимум два месяца назад. Если вы склонировали его пять минут назад, то вы ничего не получите.

ССЫЛКИ НА ПРЕДКОВ

- Ещё один основной способ указать коммит — указать коммит через его предков. Если поставить ^ в конце ссылки, для Git'а это будет означать родителя этого коммита. Допустим, история вашего проекта выглядит следующим образом:

```
$ git log --pretty=format:'%h %s' --graph
* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd added some blame and merge stuff
|/
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list
```

- В этом случае вы можете посмотреть предыдущий коммит, указав HEAD^, что означает “родитель HEAD”:

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

- Вы также можете указать некоторое число после ^. Например, d921970^2 означает “второй родитель коммита d921970”. Такой синтаксис полезен только для коммитов-слияний, которые имеют больше, чем одного родителя. Первый родитель — это ветка, на которой вы находились во время слияния, а второй — коммит на ветке, которая была слита:

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff

$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date:   Wed Dec 10 22:22:03 2008 +0000

    Some rdoc changes
```

- Другое основное обозначение для указания на предков — это ~. Это тоже ссылка на первого родителя, поэтому HEAD~ и HEAD^ эквивалентны. Различия становятся очевидными, только когда вы указываете число. HEAD~2 означает первого родителя первого родителя HEAD или прародителя — это переход по первым родителям указанное количество раз. Например, для показанной выше истории, HEAD~3 будет

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

- То же самое можно записать как HEAD^^^, что опять же означает первого родителя первого родителя первого родителя:

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

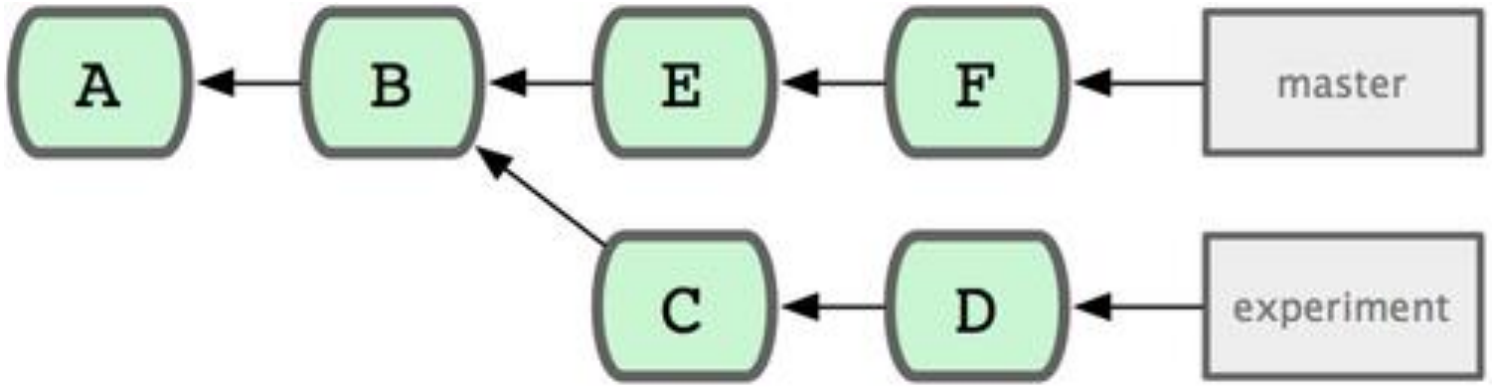
- Кроме того, можно комбинировать эти обозначения. Например, можно получить второго родителя для предыдущей ссылки (мы предполагаем, что это коммит-слияние) написав HEAD~3^2, ну и так далее.

ДИАПАЗОН КОММИТОВ

- Теперь, когда вы умеете задавать отдельные коммиты, разберёмся, как указать диапазон коммитов. Это особенно полезно при управлении ветками — если у вас много веток, вы можете использовать обозначения диапазонов, чтобы ответить на вопросы типа “Какие в этой ветке есть коммиты, которые не были слиты в основную ветку?”

ДВЕ ТОЧКИ

- Наиболее распространённый способ задать диапазон коммитов — это запись с двумя точками. По существу, таким образом вы просите Git взять набор коммитов, достижимых из одного коммита, но не достижимых из другого. Например, пускай ваша история коммитов выглядит так, как показано на рисунке



Пример истории для выбора набора коммитов.

- Допустим, вы хотите посмотреть, что в вашей ветке experiment ещё не было слито в ветку master. Можно попросить Git показать вам лог только таких коммитов с помощью master..experiment — эта запись означает “все коммиты, достижимые из experiment, которые недостижимы из master”. Для краткости и большей понятности в примерах мы будем использовать буквы для обозначения коммитов на диаграмме вместо настоящего вывода лога в том порядке, в каком они будут отображены:

```
$ git log master..experiment
D
C
```

- С другой стороны, если вы хотите получить обратное — все коммиты в master, которых нет в experiment, можно переставить имена веток. Запись experiment..master покажет всё, что есть в master, но недостижимо из experiment:

```
$ git log experiment..master
F
E
```

- Такое полезно, если вы хотите, чтобы ветка experiment была обновлённой, и хотите посмотреть, что вы собираетесь в неё слить. Ещё один частый случай использования этого синтаксиса — посмотреть, что вы собираетесь отправить на удалённый сервер:

```
$ git log origin/master..HEAD
```

- Эта команда покажет вам все коммиты в текущей ветке, которых нет в ветке master на сервере origin. Если бы вы выполнили git push, при условии, что текущая ветка отслеживает origin/master, то коммиты, которые перечислены в выводе git log origin/master..HEAD это те коммиты, которые были бы отправлены на сервер. Кроме того, можно опустить одну из сторон в такой записи — Git подставит туда HEAD. Например, вы можете получить такой же результат, как и в предыдущем примере, набрав git log origin/master.. — Git подставит HEAD сам, если одна из сторон отсутствует.

МНОЖЕСТВО ВЕРШИН

- Запись с двумя точками полезна как сокращение, но, возможно, вы захотите указать больше двух веток, чтобы указать нужную ревизию. Например, чтобы посмотреть, какие коммиты находятся в одной из нескольких веток, но не в текущей. Git позволяет сделать это с помощью использования либо символа ^, либо --not перед любыми ссылками, коммиты, достижимые из которых, вы не хотите видеть. Таким образом, следующие три команды эквивалентны:

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

- Это удобно, потому что с помощью такого синтаксиса можно указать более двух ссылок в своём запросе, чего вы не сможете сделать с помощью двух точек. Например, если вы хотите увидеть все коммиты достижимые из refA или refB, но не из refC, можно набрать одну из таких команд:

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

- Всё это делает систему выбора ревизий очень мощной, что должно помочь вам определять, что содержится в ваших ветках.

ТРИ ТОЧКИ

- Последняя основная запись для выбора диапазона коммитов — это запись с тремя точками, которая означает те коммиты, которые достижимы по одной из двух ссылок, но не по обеим одновременно. Вернёмся к примеру истории коммитов на рисунке. Если вы хотите увидеть, что находится в master или experiment, но не в обоих сразу, выполните

```
$ git log master...experiment
F
E
D
C
```

- Повторимся, что это даст вам стандартный log-вывод, но покажет только информацию об этих четырёх коммитах, упорядоченных по дате коммита, как и обычно.
- В этом случае вместе с командой log обычно используют параметр --left-right, который показывает, на какой стороне диапазона находится каждый коммит. Это помогает сделать данные полезнее:

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

- С помощью этих инструментов вы можете намного легче объяснить Git'у, какой коммит или коммиты вы хотите изучить.

ПРЯТАНЬЕ

- Часто возникает такая ситуация, что пока вы работаете над частью своего проекта, всё находится в беспорядочном состоянии, а вам нужно переключить ветки, чтобы немного поработать над чем-то другим. Проблема в том, что вы не хотите делать коммит с наполовину доделанной работой только для того, чтобы позже можно было вернуться в это же состояние. Ответ на эту проблему — команда git stash.

- Прятанье поглощает грязное состояние рабочего каталога, то есть изменённые отслеживаемые файлы и изменения в индексе, и сохраняет их в стек незавершённых изменений, которые вы потом в любое время можете снова применить.

ПРЯТАНЬЕ СВОИХ ТРУДОВ

- Чтобы продемонстрировать, как это работает, предположим, что вы идёте к своему проекту и начинаете работать над парой файлов и, возможно, добавляете в индекс одно из изменений. Если вы выполните git status, вы увидите грязное состояние проекта:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   lib/simplegit.rb
```

- Теперь вы хотите поменять ветку, но не хотите делать коммит с тем, над чем вы ещё работаете; тогда вы прячете эти изменения. Чтобы создать новую “заначку”, выполните git stash:

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

- Ваш рабочий каталог чист:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

- В данный момент вы легко можете переключить ветки и поработать где-то ещё; ваши изменения сохранены в стеке. Чтобы посмотреть, что у вас есть припрятанного, используйте git stash list:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
```

- В нашем случае две “заначки” были сделаны ранее, так что у вас теперь три разных припрятанных работы. Вы можете снова применить ту, которую только что спрятали, с помощью команды, показанной в справке в выводе первоначальной команды stash: git stash apply. Если вы хотите применить одну из старых заначек, можете сделать это, указав её имя так: git stash apply stash@{2}. Если не указывать ничего, Git будет подразумевать, что вы хотите применить последнюю спрятанную работу:

```
$ git stash apply
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   index.html
#       modified:   lib/simplegit.rb
```

- Как видите, Git восстановил изменения в файлах, которые вы отменили, когда использовали команду stash. В нашем случае у вас был чистый рабочий каталог, когда вы восстанавливали спрятанные изменения, и к тому же вы делали это на той же ветке, на которой находились во время прятанья. Но наличие чистого рабочего каталога и применение на той же ветке не обязательны для git stash apply. Вы можете спрятать изменения на одной ветке, переключиться позже на другую ветку и попытаться восстановить изменения. У вас в рабочем каталоге также могут быть изменённые и недокоммиченные файлы во время применения спрятанного — Git выдаст вам конфликты слияния, если что-то уже не может быть применено чисто.

- Изменения в файлах были восстановлены, но файлы в индексе — нет. Чтобы добиться такого, необходимо выполнить команду git stash apply с опцией --index, тогда команда попытается применить изменения в индексе. Если бы вы выполнили команду так, а не как раньше, то получили бы исходное состояние:

```
$ git stash apply --index
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   lib/simplegit.rb
```

- Всё, что делает опция apply — это пытается применить спрятанную работу — то, что вы спрятали, всё ещё будет находиться в стеке. Чтобы удалить спрятанное, выполните git stash drop с именем “заначки”, которую нужно удалить:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

- Также можно выполнить git stash pop, чтобы применить спрятанные изменения и сразу же удалить их из стека.

ОТКАТ ПРИМЕНЕНИЯ СПРЯТАННЫХ ИЗМЕНЕНИЙ

- При некоторых сценариях использования, может понадобиться применить спрятанные изменения, поработать, а потом отменить изменения, внесённые командой stash apply. Git не предоставляет команды stash unapply, но можно добиться того же эффекта получив сначала патч для спрятанных изменений, а потом применив его в перевёрнутом виде:

```
$ git stash show -p stash@{0} | git apply -R
```

- Снова, если вы не указываете параметр для stash, Git подразумевает то, что было спрятано последним:

```
$ git stash show -p | git apply -R
```

- Если хотите, сделайте псевдоним и добавьте в свой Git команду stash-unapply. Например, так:

```
$ git config --global alias.stash-unapply '!git stash show -p | git apply -R'
$ git stash
$ #... work work work
$ git stash-unapply
```

СОЗДАНИЕ ВЕТКИ ИЗ СПРЯТАННЫХ ИЗМЕНЕНИЙ

- Если вы спрятали какие-то наработки и оставили их на время, а в это время продолжили работать на той же ветке, то у вас могут возникнуть трудности с восстановлением спрятанной работы. Если apply попытается изменить файл, который вы редактировали после прятанья, то возникнет конфликт слияния, который надо будет разрешить. Если нужен более простой способ снова потестировать спрятанную работу, можно выполнить команду git stash branch, которая создаст вам новую ветку с началом из того коммита, на котором вы находились во время прятанья, восстановит в ней вашу работу и затем удалит спрятанное, если оно применилось успешно:

```
$ git stash branch testchanges
Switched to a new branch "testchanges"
# On branch testchanges
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

- Это сокращение удобно для того, чтобы легко восстановить свою работу, а затем поработать над ней в новой ветке.

ПЕРЕЗАПИСЬ ИСТОРИИ

- Неоднократно во время работы с Git'ом, вам может захотеться по какой-либо причине исправить свою историю коммитов. Одна из чудесных особенностей Git'а заключается в том, что он даёт возможность принять решение в самый последний момент. Используя индекс, вы можете решить, какие файлы пойдут в какие коммиты, непосредственно перед тем, как сделать коммит. Вы можете воспользоваться командой stash, решив, что над чем-то ещё не стоило начинать работать. А также вы можете переписать уже сделанные коммиты так, будто они были сделаны как-то по-другому. В частности, это может быть изменение порядка следования коммитов, редактирование сообщений или модифицирование файлов в коммите, уплотнение и разделение коммитов, а также полное удаление некоторых коммитов — но только до того, как вы поделитесь наработками с другими.

- В этом разделе вы узнаете, как делать подобные полезные вещи, чтобы перед публикацией приводить историю коммитов в желаемый вид.

ИЗМЕНЕНИЕ ПОСЛЕДНЕГО КОММИТА

- Изменение последнего коммита — это, вероятно, наиболее типичный случай переписывания истории, который вы будете делать. Как правило, вам от вашего последнего коммита понадобятся две основные вещи: изменить сообщение коммита или изменить только что записанный снимок состояния, добавив, изменив или удалив из него файлы.
- Если вы всего лишь хотите изменить сообщение последнего коммита — это очень просто:

```
$ git commit --amend
```

- Выполнив это, вы попадёте в свой текстовый редактор, в котором будет находиться сообщение последнего коммита, готовое к тому, чтобы его отредактировали. Когда вы сохраните текст и закроете редактор, Git создаст новый коммит с вашим сообщением и сделает его новым последним коммитом.
- Если вы сделали коммит и затем хотите изменить снимок состояния в коммите, добавив или изменив файлы, допустим, потому что вы забыли добавить только что созданный файл, когда делали коммит, то процесс выглядит в основном так же. Вы добавляете в индекс изменения, которые хотите, редактируя файл и выполняя для него git add или выполняя git rm для отслеживаемого файла, и затем git commit --amend возьмёт текущий индекс и сделает его снимком состояния нового коммита.
- Будьте осторожны, используя этот приём, потому что git commit --amend меняет SHA-1 коммита. Тут как с маленьким перемещением (rebase) — не правьте последний коммит, если вы его уже куда-то отправили.

ИЗМЕНЕНИЕ СООБЩЕНИЙ НЕСКОЛЬКИХ КОММИТОВ

- Чтобы изменить коммит, находящийся глубоко в истории, вам придётся перейти к использованию более сложных инструментов. В Git'е нет специального инструмента для редактирования истории, но вы можете использовать rebase для перемещения ряда коммитов на то же самое место, где они были изначально, а не куда-то в другое место. Используя инструмент для интерактивного перемещения, вы можете останавливаться на каждом коммите, который хотите изменить, и редактировать сообщение, добавлять файлы или делать что-то ещё. Интерактивное перемещение можно запустить, добавив опцию -i к git rebase. Необходимо указать, насколько далёкие в истории коммиты вы хотите переписать, сообщив команде, на какой коммит выполняется перемещение.
- Например, если вы хотите изменить сообщения последних трёх коммитов или сообщения для только некоторых коммитов в этой группе, вам надо передать в git rebase -i в качестве аргумента родителя последнего коммита, который вы хотите изменить, то есть HEAD~2^ или HEAD~3. Наверное, проще запомнить ~3, потому что вы пытаетесь отредактировать три последних коммита, но имейте в виду, что на самом деле вы обозначили четвёртый сверху коммит — родительский коммит для того, который хотите отредактировать:

```
$ git rebase -i HEAD~3
```

- Снова напомним, что это команда для перемещения, то есть все коммиты в диапазоне HEAD~3..HEAD будут переписаны вне зависимости от того, меняли ли вы в них сообщение или нет. Не трогайте те коммиты, которые вы уже отправили на центральный сервер — сделав так, вы запутаете других разработчиков, дав им разные версии одних и тех же изменений.
- Запуск этой команды выдаст вам в текстовом редакторе список коммитов, который будет выглядеть следующим образом:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
#  p, pick = use commit
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
```

- Важно отметить, что эти коммиты выведены в обратном порядке по сравнению с тем, как вы их обычно видите, используя команду log. Запустив log, вы получите что-то вроде этого:

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

- Обратите внимание на обратный порядок. Интерактивное перемещение выдаёт сценарий, который будет выполнен. Он начнётся с коммита, который вы указали в командной строке (HEAD~3), и воспроизведёт изменения, сделанные каждым из этих коммитов, сверху вниз. Наверху указан самый старый коммит, а не самый новый, потому что он будет воспроизведён первым.
- Вам надо отредактировать сценарий так, чтобы он останавливался на коммитах, которые вы хотите отредактировать. Чтобы сделать это, замените слово pick на слово edit для каждого коммита, на котором сценарий должен остановиться. Например, чтобы изменить сообщение только для третьего коммита, отредактируйте файл так, чтобы он выглядел следующим образом:

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

- Когда вы сохраните и выйдете из редактора, Git откатит вас назад к последнему коммиту в списке и выкинет вас в командную строку, выдав следующее сообщение:

```
$ git rebase -i HEAD~3
Stopped at 7482e0d... updated the gemspec to hopefully work better
You can amend the commit now, with

       git commit --amend

Once you're satisfied with your changes, run

       git rebase --continue
```

- В этой инструкции в точности сказано, что надо сделать. Наберите

```
$ git commit --amend
```

- Измените сообщение коммита и выйдите из редактора. Теперь выполните

```
$ git rebase --continue
```

- Данная команда применит оставшиеся два коммита автоматически и закончит на этом. Если вы измените pick на edit для большего количества строк, то вы повторите эти шаги для каждого коммита, где вы напишете edit. Каждый раз Git будет останавливаться, давая вам исправить коммит, а потом, когда вы закончите, будет продолжать.

ПЕРЕУПОРЯДОЧЕНИЕ КОММИТОВ

- Интерактивное перемещение можно также использовать для изменения порядка следования и для полного удаления коммитов. Если вы хотите удалить коммит “added cat-file” и поменять порядок, в котором идут два других коммита, измените сценарий для rebase с такого

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

- на такой:

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

- Когда вы сохраните и выйдете из редактора, Git откатит вашу ветку к родительскому для этих трёх коммиту, применит 310154e, затем f7f3f6d, а потом остановится. Вы фактически поменяли порядок следования коммитов и полностью удалили коммит “added cat-file”.

УПЛОТНЕНИЕ КОММИТОВ

- С помощью интерактивного перемещения также возможно взять несколько коммитов и сплющить их в один коммит. Сценарий выдаёт полезное сообщение с инструкциями для перемещения:

```
# Commands:
#  p, pick = use commit
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
```

- Если вместо “pick” или “edit” указать “squash”, Git применит изменения и из этого коммита, и из предыдущего, а затем даст вам объединить сообщения для коммитов. Итак, чтобы сделать один коммит из трёх наших коммитов, надо сделать так, чтобы сценарий выглядел следующим образом:

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

- Когда вы сохраните и выйдете из редактора, Git применит все три изменения, а затем опять выдаст вам редактор для того, чтобы объединить сообщения трёх коммитов:

```
# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit

# This is the 2nd commit message:

updated README formatting and added blame

# This is the 3rd commit message:

added cat-file
```

- Когда вы это сохраните, у вас будет один коммит, который вносит изменения такие же, как три бывших коммита.

РАЗБИЕНИЕ КОММИТА

- Разбиение коммита — это отмена коммита, а затем индексирование изменений частями и добавление коммитов столько раз, сколько коммитов вы хотите получить. Предположим, что вы хотите разбить средний из наших трёх коммитов. Вместо “updated README formatting and added blame” вы хотите получить два отдельных коммита: “updated README formatting” в качестве первого и “added blame” в качестве второго. Вы можете сделать это в сценарии rebase -i, поставив “edit” в инструкции для коммита, который хотите разбить:

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

- Когда вы сохраните и выйдете из редактора, Git откатится к родителю первого коммита в списке, применит первый коммит (f7f3f6d), применит второй (310154e) и выбросит вас в консоль. Здесь вы можете сбросить последний коммит в смешанном режиме с помощью git reset HEAD^ — это в сущности отменит этот коммит и оставит изменённые файлы непроиндексированными. Теперь вы можете взять сброшенные изменения и создать из них несколько коммитов. Просто добавляйте файлы в индекс и делайте коммиты, пока не сделаете несколько штук. Затем, когда закончите, выполните git rebase --continue:

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'added blame'
$ git rebase --continue
```

- Когда Git применит последний коммит (a5f4a0d) в сценарии, история будет выглядеть так:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

- Повторимся ещё раз, что эта операция меняет SHA всех коммитов в списке, так что убедитесь, что ни один из коммитов в этом списке вы ещё не успели отправить в общий репозиторий.

ОТЛАДКА С ПОМОЩЬЮ GIT

- Git также предоставляет несколько инструментов, призванных помочь вам в отладке ваших проектов. Так как Git сконструирован так, чтобы работать с практически любыми типами проектов, эти инструменты довольно общие, но зачастую они могут помочь отловить ошибку или её виновника, если что-то пошло не так.

АННОТАЦИЯ ФАЙЛА

- Если вы отловили ошибку в коде и хотите узнать, когда и по какой причине она была внесена, то аннотация файла — лучший инструмент для этого случая. Он покажет вам, какие коммиты модифицировали каждую строку файла в последний раз. Так что, если вы видите, что какой-то метод в коде содержит ошибку, то можно сделать аннотацию нужного файла с помощью git blame, чтобы посмотреть, когда и кем каждая строка метода была в последний раз отредактирована. В этом примере используется опция -L, чтобы ограничить вывод строками с 12ой по 22ую:

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 12)  def show(tree = 'master')
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 13)   command("git show #{tree}")
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 14)  end
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 16)  def log(tree = 'master')
79eaf55d (Scott Chacon  2008-04-06 10:15:08 -0700 17)   command("git log #{tree}")
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 18)  end
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20)  def blame(path)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)   command("git blame #{path}")
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22)  end
```

- Заметьте, что первое поле — это частичная SHA-1 коммита, в котором последний раз менялась строка. Следующие два поля — это значения, полученные из этого коммита — имя автора и дата создания коммита. Так что вы легко можете понять, кто и когда менял данную строку. Затем идут номера строк и содержимое файла. Также обратите внимание на строки с ^4832fe2, это те строки, которые находятся здесь со времён первого коммита для этого файла. Это коммит, в котором этот файл был впервые добавлен в проект, и с тех пор те строки не менялись. Это всё несколько сбивает с толку, потому что только что вы увидели по крайней мере три разных способа изменить SHA коммита с помощью ^, но тут вот такое значение.

- Ещё одна крутая вещь в Git'е — это то, что он не отслеживает переименования файлов в явном виде. Он записывает снимки состояний, а затем пытается выяснить, что было переименовано неявно уже после того, как это случилось. Одна из интересных функций, возможная благодаря этому, заключается в том, что вы можете попросить дополнительно выявить все виды перемещений кода. Если вы передадите -C в git blame, Git проанализирует аннотируемый файл и попытается выявить, откуда фрагменты кода в нём появились изначально, если они были скопированы откуда-то. Недавно я занимался разбиением файла GITServerHandler.m на несколько файлов, один из которых был GITPackUpload.m. Вызвав blame с опцией -C для GITPackUpload.m, я могу понять откуда части кода здесь появились:

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144)         //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 146)         NSString *parentSha;
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 147)         GITCommit *commit = [g
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 149)         //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151)         if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152)                 [refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

- Это действительно удобно. Стандартно вам бы выдали в качестве начального коммита тот коммит, в котором вы скопировали код, так как это первый коммит, в котором вы поменяли эти строки в данном файле. А сейчас Git выдал вам изначальный коммит, в котором эти строки были написаны, несмотря на то, что это было в другом файле.

БИНАРНЫЙ ПОИСК

- Аннотирование файла помогает, когда вы знаете, где у вас ошибка, и есть с чего начинать. Если вы не знаете, что у вас сломалось, и с тех пор, когда код работал, были сделаны десятки или сотни коммитов, вы наверняка обратитесь за помощью к git bisect. Команда bisect выполняет бинарный поиск по истории коммитов, и призвана помочь как можно быстрее определить, в каком коммите была внесена ошибка.

- Положим, вы только что отправили новую версию вашего кода в производство, и теперь вы периодически получаете отчёты о какой-то ошибке, которая не проявлялась, пока вы работали над кодом, и вы не представляете, почему код ведёт себя так. Вы возвращаетесь к своему коду, и у вас получается воспроизвести ошибку, но вы не понимаете, что не так.

- Вы можете использовать bisect, чтобы выяснить это. Сначала выполните git bisect start, чтобы запустить процесс, а затем git bisect bad, чтобы сказать системе, что текущий коммит, на котором вы сейчас находитесь, сломан. Затем, необходимо сказать bisect, когда было последнее известное хорошее состояние с помощью git bisect good [хороший_коммит]:

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

- Git выяснил, что между коммитом, который вы указали как последний хороший коммит (v1.0), и текущей плохой версией было сделано примерно 12 коммитов, и он выгрузил вам версию из середины. В этот момент вы можете провести свои тесты и посмотреть, проявляется ли проблема в этом коммите. Если да, то она была внесена где-то раньше этого среднего коммита; если нет, то проблема появилась где-то после коммита в середине. Положим, что оказывается, что проблема здесь не проявилась, и вы сообщаете об этом Git'у, набрав git bisect good, и продолжаете свой путь:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

- Теперь вы на другом коммите, посередине между тем, который только что был протестирован и вашим плохим коммитом. Вы снова проводите тесты и выясняете, что текущий коммит сломан. Так что вы говорите об этом Git'у с помощью git bisect bad:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

- Этот коммит хороший, и теперь у Git'а есть вся необходимая информация, чтобы определить, где проблема была внесена впервые. Он выдаёт вам SHA-1 первого плохого коммита и некоторую информацию о нём, а также какие файлы были изменены в этом коммите, так что вы сможете понять, что случилось, что могло внести эту ошибку:

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date:   Tue Jan 27 14:48:32 2009 -0800

    secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M  config
```

- Если вы закончили, необходимо выполнить git bisect reset, чтобы сбросить HEAD туда, где он был до начала бинарного поиска, иначе вы окажетесь в странном состоянии:

```
$ git bisect reset
```

- Это мощный инструмент, который поможет вам за считанные минуты проверить сотни коммитов в поисках появившейся ошибки. На самом деле, если у вас есть сценарий (script), который возвращает на выходе 0, если проект хороший и не 0, если проект плохой, то вы можете полностью автоматизировать git bisect. Для начала ему снова надо задать область бинарного поиска, задав известные хороший и плохой коммиты. Если хотите, можете сделать это, указав команде bisect start известный плохой коммит первым, а хороший вторым:

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

- Сделав так, вы получите, что test-error.sh будет автоматически запускаться на каждом выгруженном коммите, пока Git не найдёт первый сломанный коммит. Вы также можете запускать что-нибудь типа make или make tests или что-то там ещё, что запускает ваши автоматические тесты.

ПОДМОДУЛИ

- Зачастую случается так, что во время работы над некоторым проектом появляется необходимость использовать внутри него ещё какой-то проект. Возможно, библиотеку, разрабатываемую сторонними разработчиками или разрабатываемую вами обособленно и используемую в нескольких родительских проектах. Типичная проблема, возникающая при использовании подобного сценария, это, как сделать так, чтобы иметь возможность рассматривать эти два проекта как отдельные, всё же имея возможность использовать один проект внутри другого.
- Git решает эту задачу, используя подмодули (submodule). Подмодули позволяют содержать один Git-репозиторий как подкаталог другого Git-репозитория. Это даёт возможность клонировать ещё один репозиторий внутрь проекта, храня коммиты для этого репозитория отдельно.

НАЧАЛО ИСПОЛЬЗОВАНИЯ ПОДМОДУЛЕЙ

- Предположим, вы хотите добавить библиотеку Rack (интерфейс шлюза веб-сервера Ruby) в свой проект, возможно, внося свои собственные изменения в него, но продолжая сливать их с изменениями основного проекта. Первое, что вам требуется сделать, это клонировать внешний репозиторий в подкаталог. Добавление внешних проектов в качестве подмодулей делается командой git submodule add:

```
$ git submodule add git://github.com/chneukirchen/rack.git rack
Initialized empty Git repository in /opt/subtest/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 422 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
```

- Теперь у вас внутри проекта в подкаталоге с именем rack находится проект Rack. Вы можете переходить в этот подкаталог, вносить изменения, добавить ваш собственный доступный для записи внешний репозиторий для отправки в него своих изменений, извлекать и сливать из исходного репозитория и многое другое. Если вы выполните git status сразу после добавления подмодуля, то увидите две вещи:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   .gitmodules
#       new file:   rack
```

- Вначале вы заметите файл .gitmodules. Это конфигурационный файл, который содержит соответствие между URL проекта и локальным подкаталогом, в который был загружен подмодуль:

```
$ cat .gitmodules
[submodule "rack"]
      path = rack
      url = git://github.com/chneukirchen/rack.git
```

- Если у вас несколько подмодулей, то в этом файле будет несколько записей. Важно обратить внимание на то, что этот файл находится под версионным контролем вместе с другими вашими файлами, так же как и файл .gitignore. Он отправляется при выполнении push и загружается при выполнении pull вместе с остальными файлами проекта. Так другие люди, которые клонируют этот проект, узнают, откуда взять проекты-подмодули.

- В следующем листинге вывода git status присутствует элемент rack. Если вы выполните git diff для него, то увидите кое-что интересное:

```
$ git diff --cached rack
diff --git a/rack b/rack
new file mode 160000
index 0000000..08d709f
--- /dev/null
+++ b/rack
@@ -0,0 +1 @@
+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

- Хотя rack является подкаталогом в вашем рабочем каталоге, Git видит его как подмодуль и не отслеживает его содержимое, если вы не находитесь в нём. Вместо этого, Git записывает его как один конкретный коммит из этого репозитория. Если вы производите изменения в этом подкаталоге и делаете коммит, основной проект замечает, что HEAD в подмодуле был изменён, и регистрирует тот хеш коммита, над которым вы в данный момент завершили работу в подмодуле. Таким образом, если кто-то склонирует этот проект, он сможет воссоздать окружение в точности.

- Это важная особенность подмодулей – вы запоминаете их как определенный коммит (состояние), в котором они находятся. Вы не можете записать подмодуль под ссылкой master или какой-либо другой символьной ссылкой.
- Если вы создадите коммит, то увидите что-то вроде этого:

```
$ git commit -m 'first commit with submodule rack'
[master 0550271] first commit with submodule rack
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 rack
```

- Обратите внимание на режим 160000 для элемента rack. Это специальный режим в Git'е, который по существу означает, что в качестве записи в каталоге сохраняется коммит, а не подкаталог или файл.

- Вы можете обращаться с каталогом rack как с отдельным проектом и время от времени обновлять свой “надпроект” с помощью указателя на самый последний коммит в этом подпроекте. Все команды Git’а в этих двух каталогах работают независимо друг от друга:

```
$ git log -1
commit 0550271328a0038865aad6331e620cd7238601bb
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Apr 9 09:03:56 2009 -0700

    first commit with submodule rack

$ cd rack/
$ git log -1
commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
Author: Christian Neukirchen <chneukirchen@gmail.com>
Date:   Wed Mar 25 14:49:04 2009 +0100

    Document version change
```

КЛОНИРОВАНИЕ ПРОЕКТА С ПОДМОДУЛЯМИ

- Сейчас мы склонируем проект, содержащий подмодуль. После получения такого проекта в вашей копии будут каталоги, содержащие подмодули, но пока что без единого файла в них:

```
$ git clone git://github.com/schacon/myproject.git
Initialized empty Git repository in /opt/myproject/.git/
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.
$ cd myproject
$ ls -l
total 8
-rw-r--r--  1 schacon  admin   3 Apr  9 09:11 README
drwxr-xr-x  2 schacon  admin  68 Apr  9 09:11 rack
$ ls rack/
```

- Каталог rack присутствует, но он пустой. Необходимо выполнить две команды: git submodule init для инициализации вашего локального файла конфигурации и git submodule update для получения всех данных из подмодуля и перехода к соответствующему коммиту, указанному в вашем основном проекте:

```
$ git submodule init
Submodule 'rack' (git://github.com/chneukirchen/rack.git) registered for path 'rack'
$ git submodule update
Initialized empty Git repository in /opt/myproject/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 173 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
Submodule path 'rack': checked out '08d709f78b8c5b0fbeb7821e37fa53e69afcf433'
```

- Теперь ваш подкаталог rack точно в том состоянии, в котором он был, когда вы раньше делали коммит. Если другой разработчик внесёт изменения в код rack и затем сделает коммит, а вы потом обновите эту ссылку и сольёте её, то вы получите что-то странное:

```
$ git merge origin/master
Updating 0550271..85a3eee
Fast forward
 rack |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
[master*]$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   rack
```

- Вы слили то, что по существу является изменением указателя на подмодуль. Но при этом обновления кода в каталоге подмодуля не произошло, так что всё выглядит так, как будто вы имеете грязное состояние в своём рабочем каталоге:

```
$ git diff
diff --git a/rack b/rack
index 6c5e70b..08d709f 160000
--- a/rack
+++ b/rack
@@ -1 +1 @@
-Subproject commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

- Это всё из-за того, что ваш указатель на подмодуль не соответствует тому, что на самом деле находится в каталоге подмодуля. Чтобы исправить это, необходимо снова выполнить git submodule update:

```
$ git submodule update
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 2 (delta 0)
Unpacking objects: 100% (3/3), done.
From git@github.com:schacon/rack
   08d709f..6c5e70b  master     -> origin/master
Submodule path 'rack': checked out '6c5e70b984a60b3cecd395edd5b48a7575bf58e0'
```

- Вы вынуждены делать так каждый раз, когда вы получаете изменения подмодуля в главном проекте. Это странно, но это работает.

- Распространённая проблема возникает, когда разработчик делает изменения в своей локальной копии подмодуля, но не отправляет их на общий сервер. Затем он создаёт коммит содержащий указатель на это непубличное состояние и отправляет его в основной проект. Когда другие разработчики пытаются выполнить git submodule update, система работы с подмодулями не может найти указанный коммит, потому что он существует только в системе первого разработчика. Если такое случится, вы увидите ошибку вроде этой:

```
$ git submodule update
fatal: reference isn't a tree: 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Unable to checkout '6c5e70b984a60b3cecd395edd5ba7575bf58e0' in submodule path 'rack'
```

- Вам надо посмотреть, кто последним менял подмодуль:

```
$ git log -1 rack
commit 85a3eee996800fcfa91e2119372dd4172bf76678
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Apr 9 09:19:14 2009 -0700

    added a submodule reference I will never make public. hahahahaha!
```

- А затем отправить этому человеку письмо со своими возмущениями.

СУПЕРПРОЕКТЫ

- Иногда разработчики хотят объединить подкаталоги крупного проекта в нечто связанное в зависимости от того, в какой они команде. Это типично для людей, перешедших с CVS или Subversion, где они определяли модуль или набор подкаталогов, и они хотят сохранить данный тип рабочего процесса.
- Хороший способ сделать такое в Git'е — это сделать каждый из подкаталогов отдельным Git-репозиторием, и создать Git-репозиторий для суперпроекта, который будет содержать несколько подмодулей. Преимущество такого подхода в том, что вы можете более гибко определять отношения между проектами при помощи меток и ветвей в суперпроектах.

ПРОБЛЕМЫ С ПОДМОДУЛЯМИ

- Однако, использование подмодулей не обходится без загвоздок. Во-первых, вы должны быть относительно осторожны, работая в каталоге подмодуля. Когда вы выполняете команду git submodule update, она возвращает определённую версию проекта, но не внутри ветви. Это называется состоянием с отделённым HEAD (detached HEAD) — это означает, что файл HEAD указывает на конкретный коммит, а не на символическую ссылку. Проблема в том, что вы, скорее всего, не хотите работать в окружении с отделённым HEAD, потому что так легко потерять изменения. Если вы сделаете первоначальный submodule update, сделаете коммит в каталоге подмодуля, не создавая ветки для работы в ней, и затем вновь выполните git submodule update из основного проекта, без создания коммита в суперпроекте, Git затрёт ваши изменения без предупреждения. Технически вы не потеряете проделанную работу, но у вас не будет ветки, указывающей на неё, так что будет несколько сложновато её восстановить.

- Для предотвращения этой проблемы создавайте ветвь, когда работаете в каталоге подмодуля с использованием команды git checkout -b work или какой-нибудь аналогичной. Когда вы сделаете обновление подмодуля командой submodule update в следующий раз, она всё же откатит вашу работу, но, по крайней мере, у вас будет указатель для возврата назад.
- Переключение веток с подмодулями в них также может быть мудрёным. Если вы создадите новую ветку, добавите туда подмодуль и затем переключитесь обратно, туда где не было этого подмодуля, вы всё ещё будете иметь каталог подмодуля в виде неотслеживаемого каталога:

```
$ git checkout -b rack
Switched to a new branch "rack"
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/myproj/rack/.git/
...
Receiving objects: 100% (3184/3184), 677.42 KiB | 34 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
$ git commit -am 'added rack submodule'
[rack cc49a69] added rack submodule
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 rack
$ git checkout master
Switched to branch "master"
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       rack/
```

- Вы будете вынуждены либо переместить каталог подмодуля в другое место, либо удалить его. В случае удаления вам потребуется клонировать его снова при переключении обратно, и тогда вы можете потерять локальные изменения или ветки, которые не были отправлены в основной репозиторий.
- Последняя проблема, которая возникает у многих, и о которой стоит предостеречь, возникает при переходе от подкаталогов к подмодулям. Если вы держали некоторые файлы под версионным контролем в своём проекте, а сейчас хотите перенести их в подмодуль, вам надо быть осторожным, иначе Git разозлится на вас. Допустим, вы держите файлы rack в подкаталоге проекта, и вы хотите вынести его в подмодуль. Если вы просто удалите подкаталог и затем выполните submodule add, Git наорёт на вас:

```
$ rm -Rf rack/
$ git submodule add git@github.com:schacon/rack.git rack
'rack' already exists in the index
```

- Вначале вам следует убрать каталог rack из индекса (убрать из-под версионного контроля). Потом можете добавить подмодуль:

```
$ git rm -r rack
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/testsub/rack/.git/
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 88 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
```

- Теперь, предположим, вы сделали это в ветке. Если вы попытаетесь переключиться обратно на ту ветку, где эти файлы всё ещё в актуальном дереве, а не в подмодуле, то вы получите такую ошибку:

```
$ git checkout master
error: Untracked working tree file 'rack/AUTHORS' would be overwritten by merge.
```

- Вам следует переместить каталог подмодуля rack, перед тем, как вы сможете переключиться на ветку, которая не содержит его:

```
$ mv rack /tmp/
$ git checkout master
Switched to branch "master"
$ ls
README  rack
```

- Затем, когда вы переключитесь обратно, вы получите пустой каталог rack. Вы сможете либо выполнить git submodule update для повторного клонирования, или вернуть содержимое вашего каталога /tmp/rack обратно в пустой каталог.

СЛИЯНИЕ ПОДДЕРЕВЬЕВ

- Теперь, когда вы увидели сложности системы подмодулей, давайте посмотрим на альтернативный путь решения той же проблемы. Когда Git выполняет слияние, он смотрит на то, что требуется слить воедино, и потом выбирает подходящую стратегию слияния. Если вы сливаете две ветви, Git использует *рекурсивную* (recursive) стратегию. Если вы объединяете более двух ветвей, Git выбирает стратегию *осьминога* (octopus). Эти стратегии выбираются за вас автоматически потому, что рекурсивная стратегия может обрабатывать сложные трёхсторонние ситуации слияния — например, более чем один общий предок — но она может сливать только две ветви. Слияние методом осьминога может справиться с множеством веток, но является более осторожным, чтобы предотвратить сложные конфликты, так что этот метод является стратегией по умолчанию при слиянии более двух веток.
- Однако, существуют другие стратегии, которые вы также можете выбрать. Одна из них — слияние *поддеревьев* (subtree), и вы можете использовать его для решения задачи с подпроектами. Сейчас вы увидите, как выполнить то же встраивание rack, как и в предыдущем разделе, но с использованием стратегии слияния поддеревьев.

- Идея слияния поддеревьев состоит в том, что у вас есть два проекта, и один из проектов отображается в подкаталог другого и наоборот. Если вы зададите в качестве стратегии слияния метод subtree, то Git будет достаточно умён, чтобы понять, что один из проектов является поддеревом другого и выполнит слияние в соответствии с этим. И это довольно удивительно.
- Сначала добавим приложение Rack в проект. Добавим проект Rack как внешнюю ссылку в свой проект, и затем поместим его в отдельную ветку:

```
$ git remote add rack_remote git@github.com:schacon/rack.git
$ git fetch rack_remote
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From git@github.com:schacon/rack
 * [new branch]      build      -> rack_remote/build
 * [new branch]      master     -> rack_remote/master
 * [new branch]      rack-0.4   -> rack_remote/rack-0.4
 * [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"
```

- Теперь у нас есть корень проекта Rack в ветке rack_branch и наш проект в ветке master. Если вы переключитесь на одну ветку, а затем на другую, то увидите, что содержимое их корневых каталогов различно:

```
$ ls
AUTHORS         KNOWN-ISSUES   Rakefile      contrib            lib
COPYING         README         bin           example            test
$ git checkout master
Switched to branch "master"
$ ls
README
```

- Допустим, вы хотите поместить проект Rack в подкаталог своего проекта в ветке master. Вы можете сделать это в Git'е командой git read-tree. Вы узнаете больше про команду read-tree и её друзей в главе 9, а пока достаточно знать, что она считывает корень дерева одной ветки в индекс и рабочий каталог. Вам достаточно переключиться обратно на ветку master и вытянуть ветку rack в подкаталог rack основного проекта из ветки master:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

- После того как вы сделаете коммит, все файлы проекта Rack будут находиться в этом подкаталоге — будто вы скопировали их туда из архива. Интересно то, что вы можете довольно легко слить изменения из одной ветки в другую. Так что если проект Rack изменится, вы сможете вытянуть изменения из основного проекта, переключившись в его ветку и выполнив git pull:

```
$ git checkout rack_branch
$ git pull
```

- Затем вы можете слить эти изменения обратно в свою главную ветку. Можно использовать git merge -s subtree — это сработает правильно, но тогда Git, кроме того, объединит вместе истории, чего вы, вероятно, не хотите. Чтобы получить изменения и заполнить сообщение коммита, используйте опции --squash и --no-commit вместе с опцией стратегии -s subtree:

```
$ git checkout master
$ git merge --squash -s subtree --no-commit rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

- Все изменения из проекта Rack слиты и готовы для локальной фиксации. Вы также можете сделать наоборот — внести изменения в подкаталог rack вашей ветки master, и затем слить их в ветку rack_branch, чтобы позже представить их мейнтейнерам или отправить их в основной репозиторий проекта с помощью git push.
- Для получения разности между тем, что у вас есть в подкаталоге rack и кодом в вашей ветке rack_branch, чтобы увидеть нужно ли вам объединять их, вы не можете использовать нормальную команду diff. Вместо этого вы должны выполнить git diff-tree с веткой, с которой вы хотите сравнить:

```
$ git diff-tree -p rack_branch
```

- Или, для сравнения того, что в вашем подкаталоге rack с тем, что было в ветке master на сервере во время последнего обновления, можно выполнить:

```
$ git diff-tree -p rack_remote/master
```

ДОПОЛНИТЕЛЬНЫЕ ЛЕКЦИИ ДЛЯ САМОСТОЯТЕЛЬНОГО ИЗУЧЕНИЯ

- Ранее рассмотренный материал был позаимствован у разработчика https://github.com/zzet
- Также у него есть ещё лекция про GIT-сервер:

http://zzet.org/git/learning/undev/coursify/2014/03/28/lection-4-git-course-undev.html

- И лекция про внутреннее устройство GIT'а:

http://zzet.org/git/learning/undev/coursify/2014/03/28/lection-5-git-course-undev.html

- Данный материал предлагается изучить самостоятельно.

ПОЛЕЗНЫЕ КОМАНДЫ

- Обновить локальную ветку до состояния «основной ветки» (как правило, develop или master):

```
git checkout develop
git pull
git checkout local_branch
git merge develop
git push origin local_branch
```

- Изменить сообщение последнего коммита (если не сделан git push):

```
git commit --amend –m“new message”
```

- Изменить сообщение N-го коммита (если не сделан git push):

git rebase –i HEAD~n (где n – количество коммитов)
pick замените на reword
в редакторе измените текст сообщения

- Изменить сообщение последнего коммита (если сделан git push):

git commit --amend –m“new message”
git push –force

- Посмотреть изменения добавленного, но не закоммиченного файла:

git diff --staged /path/to/file

- Метод дихотомии, для определения «плохого коммита», который что-то поломал:

git bisect start
git bisect bad #hash_commit
git bisect good #hash_commit
…
git bisect reset

НЕБОЛЬШИЕ СОВЕТЫ

- Используйте **git add –p**, а не **git add .** (Чтобы Ваши коммиты были максимально атомарны)
- Пишите понятые сообщения к коммитам
- Не храните бинарные файлы в **git**. Более корректно хранить их в артефактори (например, https://jfrog.com/artifactory/) Или в любом другом хранилище: расшаренная папка, **google disk** и прочее.
- Прежде, чем работать с **git**'ом, установите внутри команды правила:
  - О том, какой должен быть формат у коммита (**git** позволяет устанавливать формат коммитов, и не пропускает те коммиты, которые не подходят под формат)
  - О том, как отмечать релизы: выделять для них ветку или отмечать их **tag**'ами
  - О том, можно ли коммитить в основную ветку или же она предназначена только для слияний.

ПРИМЕР ВНУТРЕННИХ ПРАВИЛ ДЛЯ РАБОТЫ С GIT'ОМ

1. Создать ветку с номером задачи (например: `CLD-2222`) из ветки `develop`. Если необходимо создать ветку не из ветки `develop`, обсудить это с тимлидом.
2. Реализовать необходимый для задачи функционал.
3. Перед слиянием Вашей ветки в ветку `develop`:
    1. Обновить свою ветку до акктуального состояния ветки `develop` и разрешить конфликты, в случае их появления
    2. Проверить реализованный функционал средствами ручного тестирования.
    3. Запустить автоматический тесты.
4. Создать запрос на слияние Вашей ветки в ветку `develop`.
5. Указать в запросе на слияние проверяющего.
6. Исправить замечания проверяющего, если они есть.
7. Если в процессе исправления замечаний ветка `develop` изменила своё состояние (какой-то другой разработчик добавил туда свои наработки), то необходимо повторно обновить свою ветку до акктуального состояния `develop` и разрешить конфликты, в случае их появления