Лекция 6. Инструменты Git

Лекция 6. Инструменты Git

К этому времени вы уже изучили большинство повседневных команд и способы организации рабочего процесса, необходимые для того, чтобы поддерживать Git-репозиторий для управления версиями 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. Вы совсем не обязательно будете использовать их каждый день, но, возможно, в какой-то момент они вам понадобятся.

Выбор ревизии

Git позволяет вам 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.

Сокращенный 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.

Например, предположим, что вы хотите посмотреть содержимое какого-то конкретного коммита. Вы выполняете команду git logи 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..... Если вы будете использовать git show, чтобы посмотреть содержимое этого коммита следующие команды эквивалентны (предполагая, что сокращенные 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 на Git, ядре Linux только начинает появляться необходимость использовать 12 символов из 40 возможных для сохранения уникальности.

Небольшое замечание о SHA-1

Многие люди интересуются, что произойдет, если они в какой-то момент, по некоторой случайности, получат два объекта в репозитории, которые будут иметь два одинаковых значения SHA-1 хеша. Что тогда?

Если вы вдруг закоммитите объект, SHA-1 хеш которого такой же, как у некоторого предыдущего объекта в вашем репозитории, Git обнаружит предыдущий объект в вашей базе

данных Git, и посчитает, что он был уже записан. Если вы в какой-то момент попытаетесь получить этот объект опять, вы всегда будете получать данные первого объекта.

Однако, вы должны осознавать то, как смехотворно маловероятен этот сценарий. Длина SHA-1 составляет 20 байт или 160 бит. Количество случайно хешированных объектов, необходимое для того, чтобы получить 50% вероятность одиночного совпадения составляет порядка 2^{80} (формула для определения вероятности совпадения: p = (n(n-1)/2) * (1/2^160))).

2^{80} это 1.2×10^{24} или один миллион миллиарда миллиардов. Это в 1200 раз больше количества песчинок на земле.

Вот пример для того, чтобы вы поняли, что необходимо, чтобы получить SHA-1 коллизию. Если бы все 6.5 миллиардов людей на Земле программировали, и каждую секунду каждый из них производил количество кода, эквивалентное всей истории ядра Linux (1 миллион Git объектов) и отправлял его в один огромный Git-репозиторий, то потребовалось бы 5 лет для того, чтобы заполнить репозиторий достаточно для того, чтобы получить 50% вероятность единичной SHA-1 коллизии. Более вероятно, что каждый член вашей команды программистов будет атакован и убит волками в несвязанных друг с другом случаях в одну и ту же ночь.

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, которая ожидает коммит или значение SHA-1. Например, если вы хотите посмотреть последний коммит в ветке, следующие команды эквивалентны, предполагая, что ветка topic1ссылается 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, которая называется rev-parse. Вы можете заглянуть в Главу 9 для получения большей информации о служебных утилитах; 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-parseк 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 эту информацию в эту временную историю. И вы можете использовать и эти данные, чтобы задать прошлый коммит. Если вы хотите посмотреть какое значение HEAD имел пять шагов назад для своего репозитория, вы можете использовать ссылку вида @{n}, 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 masterвчера, 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 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~3будет

```
$ 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 6-1.

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

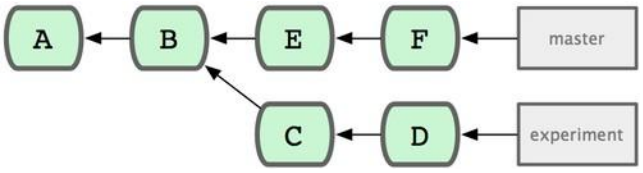

Рисунок 6.1: Пример истории для выбора набора коммитов.

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 собираете 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 refAили 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 6-1. Если вы хотите увидеть, что находится в masterили 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, какой коммит или коммиты вы хотите изучить.

Интерактивное индексирование

Вместе с Git поставляется пара сценариев (script), облегчающих выполнение некоторых задач в командной строке. Сейчас мы посмотрим на несколько интерактивных команд, которые помогут вам легко смастерить свои коммиты так, чтобы включить в них только определённые части файлов. Эти инструменты сильно помогают в случае, когда вы поменяли кучу файлов, а потом решили, что хотите, чтобы эти изменения были в нескольких сфокусированных коммитах, а не в одном большом путанном коммите. Так вы сможете убедиться, что ваши коммиты это логически разделённые наборы изменений, которые будет легко просматривать другим разработчикам работающими с вами.

Затем идёт раздел Commands (команды). Тут можно сделать многие вещи, включая добавление файлов в индекс, удаление файлов из индекса, индексирование файлов частями, добавление неотслеживаемых файлов и просмотр дельт (diff) проиндексированных изменений.

Добавление и удаление файлов из индекса

Если набрать 2 или u в приглашении What now>, сценарий спросит какие файлы вы хотите добавить в индекс:

```
   What now> 2
            staged                unstaged path
                                     +0/-1 TODO
  1:    unchanged
                                  +1/-1 index.html
Update>>
```

Чтобы проиндексировать файлы TODO и index.html, нужно набрать их номера:

```
Update>> 1,2
staged              unstaged path
* 1:                  unchanged        +0/-1 TODO
* 2:                  unchanged        +1/-1 index.html
3: unchanged         +5/-1 lib/simplegit.rb Update>>
```

Символ * рядом с каждым файлом означает, что файл выбран для индексирования. Если вы сейчас ничего не будете вводить, а нажмёте Enter в приглашении Update>>, то Git возьмёт всё, что уже выбрано, и добавит в индекс:

```
    Update>>
updated 2 paths

               2: update      3: revert      4: add untracked
*** Commands ***
    1: status
                staged      unstaged path
       1:        +0/-1       nothing TODO
     2:        +1/-1      nothing index.html
```

Как видите, теперь файлы TODO и index.html проиндексированы (staged), а файл simple- git.rb всё ещё нет. Если в этот момент вы хотите удалить файл TODO из индекса, используйте опцию 3или r(revert):

```
*** Commands ***
  1: status      2: update      3: revert      4: add untracked
  5: patch
                staged     unstaged path
       1:        +0/-1      nothing TODO
     2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
                Revert>> 1
                staged     unstaged path
     * 1:        +0/-1      nothing TODO
```

Взглянув на статус снова, вы увидите, что файл TODO удалён из индекса:

```
                              *** Commands ***
```

```
  1: status     2: update      3: revert      4: add untracked
  5: patch       6: diff        7: quit           8: help
                staged     unstaged path
       1:    unchanged        +0/-1 TODO
     2:        +1/-1      nothing index.html
```

Чтобы посмотреть дельту для проиндексированных изменений, используйте команду 6 или d(diff). Она покажет вам список проиндексированных файлов, и вы можете выбрать те, для которых хотите посмотреть дельту. Это почти то же, что указать git diff --cached в командной строке:

```
*** Commands ***
                2: update      3: revert     4: add untracked
  1: status
  5: patch
                       staged     unstaged path
  1:         +1/-1      nothing index.html
             Review diff>> 1

diff --git a/index.html b/index.html
    index 4d07108..4335f49 100644

                      --- a/index.html

                      +++ b/index.html

               @@ -16,7 +16,7 @@ Date Finder

                     <p id="out">...</p>

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

С помощью этих базовых команд, вы можете использовать интерактивный режим для git add, чтобы немного проще работать со своим индексом.

Индексирование по частям

Для Git также возможно индексировать определённые части файлов, а не всё сразу. Например, если вы сделали несколько изменений в файле simplegit.rb и хотите проиндексировать одно из них, а другое — нет, то сделать такое в Git очень легко. В строке приглашения интерактивного режима наберите 5 или p (patch). Git спросит, какие файлы вы хотите индексировать частями; затем для каждой части изменений в выбранных файлах, один за другим будут показываться куски дельт файла и вас будут спрашивать, хотите ли вы занести их в индекс:

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
          index dd5ecc4..57399e0 100644

                                --- a/lib/simplegit.rb
```

```
                                +++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
                end

                                def log(treeish = 'master')
                    -    command("git log -n 25 #{treeish}")
+    command("git log -n 30 #{treeish}")
                  end

                                      def blame(path)
                     Stage this hunk [y,n,a,d,/,j,J,g,e,?]?
```

На этой стадии у вас много вариантов действий. Набрав ?вы получите список того, что вы можете сделать:

Stage this hunk [y,n,a,d,/,j,J,g,e,?]? ?

y - stage this hunk (добавить этот кусок в индекс)

n - do not stage this hunk (не добавлять этот кусок в индекс)

a - stage this and all the remaining hunks in the file (добавить этот и все оставшиеся куски в этом файле в инд d - do not stage this hunk nor any of the remaining hunks in the file (не добавлять в индекс ни этот, ни послед g - select a hunk to go to (выбрать кусок и перейти к нему)

/ - search for a hunk matching the given regex (поиск куска по регулярному выражению)

j - leave this hunk undecided, see next undecided hunk (отложить решение для этого куска, перейти к следующем J - leave this hunk undecided, see next hunk (отложить решение для этого куска, перейти к следующему куску)

k - leave this hunk undecided, see previous undecided hunk (отложить решение для этого куска, перейти к преды K - leave this hunk undecided, see previous hunk (отложить решение для этого куска, перейти к предыдущему кус s - split the current hunk into smaller hunks (разбить текущий кусок на меньшие части)

e - manually edit the current hunk (отредактировать текущий кусок вручную)

? - print help (вывести справку)

Как правило, вы будете использовать у или n для индексирования каждого куска, но индексирование всех кусков сразу в некоторых файлах или откладывание решения на потом также может оказаться полезным. Если вы добавите в индекс одну часть файла, а другую часть

— нет, вывод статуса будет выглядеть так:

```
  What now> 1
                           unstaged path
         staged
                              +0/-1 TODO
 1:    unchanged
```

Статус файла simplegit.rb выглядит любопытно. Он показывает, что часть строк в индексе, а часть — не в индексе. Мы частично проиндексировали этот файл. Теперь вы можете выйти из интерактивного сценария и выполнить git commit, чтобы создать коммит из этих частично проиндексированных файлов.

В заключение скажем, что нет необходимости входить в интерактивный режим git add, чтобы выполнять индексирование частями — вы можете запустить тот же сценарий набрав git add -рили git add --patchв командной строке.

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
#   (use "git reset HEAD <file>..." to unstage)
#
       modified:   index.html
#
# Changed but not updated:
#   (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
#    (use "git add <file>..." to update what will be committed)
#
       modified:       index.html
#
```

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
#    (use "git reset HEAD <file>..." to unstage)
#
#        modified:    index.html
#
# Changed but not updated:
```

```
#    (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
```

так:

Если хотите, сделайте псевдоним и добавьте в свой git команду stash-unapply. Например,

```
$ 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 gitstashbranch, 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
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   index.html
#
# Changed but not updated:
#   (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, вам может захотеться по какой-либо причине исправить свою историю коммитов. Одна из чудесных особенностей Git заключается в том, что он даёт возможность принять решение в самый последний момент. Вы можете решить какие файлы пойдут в какие коммиты перед тем как сделать коммит используя индекс, вы можете решить, что над чем-то ещё не стоило начинать работать и использовать команду stash. А также вы можете переписать уже сделанные коммиты так, как-будто они были сделаны как-то по- другому. В частности это может быть изменение порядка следования коммитов, изменение сообщений или изменение 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 выполнять подобные полезные задачи и как сделать так, чтобы история коммитов выглядела так как вам хочется перед тем, как вы её опубликуете.

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 --amendменяет 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 нет специального инструмента для редактирования истории, но вы можете использовать rebaseдля 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 далекие 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~2ˆили

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 эта команда для перемещения, то есть все коммиты в диапазоне HEAD~3..HEADбудут 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 как-нибудь так:

```
                        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.
```

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 типа следующего:

```
$ 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
```

Эта команда применит оставшиеся два коммита автоматически, и тогда всё. Если вы измените pick на edit для большего количества строк, то вы повторите эти шаги для каждого коммита, где вы напишите 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 на такой:

```
pick f7f3f6d changed my name a bit

pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

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

Когда вы сохраните и выйдите 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.
```

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 выйдите 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. Например, 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
```

Теперь, когда сценарий выбросит вас в командную строку, отмените этот коммит с помощью reset, возьмите изменения, которые были сброшены и создайте из них несколько коммитов. Когда вы сохраните и выйдите 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 этот коммит в смешанном режиме с помощью git reset

HEADˆ — это эффективно отменит этот коммит и оставит изменённые файлы непроиндексированными.

Теперь вы можете добавлять файлы в индекс и делать коммиты, пока не получите 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
```

так:

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

```
$ 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 не успели уже отправить в общий репозиторий.

Крайнее средство: filter-branch

Есть ещё один вариант переписывания истории, который можно использовать если надо переписать большое количество коммитов в автоматизируемой форме — например, везде поменять свой e-mail адрес или удалить файл из каждого коммита — это команда filter-branch.

Она может переписать огромные периоды вашей истории, так что, возможно, вообще не стоит использовать её, если только ваш проект не успел ещё стать публичным и другие люди не успели ещё проделать работу на основе коммитов, которые вы собрались переписать. Однако, она может быть весьма полезной. Мы посмотрим на некоторые типичные варианты использования команды так, чтобы вы получили представление о тех вещах, на которые она способна.

Удаление файла изо всех коммитов. Такое случается довольно часто. Кто-нибудь случайно добавляет в коммит огромный бинарный файл необдуманно выполнив git add ., и вы хотите удалить его отовсюду. Или, может быть, вы нечаянно добавили в коммит файл содержащий пароль, а теперь хотите сделать код этого проекта открытым. filter-branch — это тот инструмент, который вы наверняка захотите использовать, чтобы прочесать всю историю. Чтобы удалить файл с именем passwords.txt изо всей истории, используйте опцию --tree-filter для filter-branch:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
  Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)

                        Ref 'refs/heads/master' was rewritten
```

Опция --tree-filter выполняет указанную команду после выгрузки каждой версии проекта и затем заново делает коммит из результата. В нашем случае, мы удалили файл с именем passwords.txt из каждого снимка состояния независимо от того существовал ли он там или нет. Если вы хотите удалить все случайно добавленные резервные копии сделанные вашим текстовым редактором, выполните что-то типа git filter-branch --tree-filter 'rm -f *~' HEAD.

Вы увидите как Git переписывает деревья и коммиты, а в конце переставляет указатель ветки. Как правило, хороший вариант — делать это в тестовой ветке, а затем жёстко сбрасывать ветку master с помощью reset --hard, когда вы поймёте, что результат это то, чего вы действительно добивались. Чтобы запустить filter-branch для всех веток, можно передать команде параметр --all.

Сделать подкаталог новым корнем Предположим, вы импортировали репозиторий из другой системы управления версиями и в нём есть бессмысленные каталоги (trunk, tags, и др.). Если вы хотите сделать trunkновым корнем проекта, команда filter-branchможет помочь вам сделать и это:

```
   $ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
         Ref 'refs/heads/master' was rewritten
```

Теперь всюду корневой каталог проекта будет в подкаталоге trunk. Git также автоматически удалит все коммиты, которые не затрагивают данный подкаталог.

Глобальное именение e-mail адреса Ещё один типичный случай это, когда вы забыли выполнить git config, чтобы задать своё имя и e-mail адрес перед тем как начать работать. Или, возможно, вы хотите открыть код своего проекта с работы и поменять все свои рабочие e- mail'ы на свой личный адрес. В любом случае, с помощью filter-branchвы с таким же успехом можете поменять адреса почты в нескольких коммитах за один раз. Вам надо быть аккуратным, чтобы не поменять и чужие адреса, поэтому используйте --commit-filter:

```
                    $ git filter-branch --commit-filter '

        if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
                            then

                GIT_AUTHOR_NAME="Scott Chacon";
              GIT_AUTHOR_EMAIL="schacon@example.com";
                    git commit-tree "$@";else

             git commit-tree "$@";
               fi' HEAD
```

Эта команда проходит по всем коммитам и переписывает их так, чтобы там был указан новый адрес. Так как коммиты содержат значения SHA-1 своих родителей, эта команда поменяет все SHA в вашей истории, а не только те, в которых есть указанный e-mail адрес.

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 глючный, 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}")
                                                                    end
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 14)
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 16)
         (Magnus Chacon 2008-04-13                                  def blame(path)
79eaf55d (Scott Chacon  2008-04-06 10:15:08 -0700 17)
         (Magnus Chacon 2008-04-13                                 command("git blame #{path}")
```

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 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 GIT- PackUpload.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)
                                                - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 142)
f344f58d GITServerHandler.m (Scott 2009-01-04 143)                {
70befddd GITServerHandler.m (Scott 2009-03-22
ad11ac80  GITPackUpload.m   (Scott  2009-03-24 144)        NSString *parentSha;
ad11ac80  GITPackUpload.m   (Scott  2009-03-24             GITCommit *commit = [g
ad11ac80  GITPackUpload.m   (Scott  2009-03-24 145)
ad11ac80  GITPackUpload.m   (Scott  2009-03-24 146)
ad11ac80  GITPackUpload.m   (Scott  2009-03-24             //NSLog(@"GATHER COMMI
ad11ac80  GITPackUpload.m   (Scott  2009-03-24 147)
```

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, не смотря 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 bi- sect 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 gitbisectgood[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, вы говорите Git об этом набрав git bisect goodи 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 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 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 их команде bisect start, указав 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.

Вот пример. Предположим, вы разрабатываете веб-сайт и создаёте Atom-ленты. И вместо того, чтобы писать собственный код генерирующий Atom, вы решили использовать библиотеку. Вы, вероятно, должны либо подключить нужный код с помощью разделяемой библиотеки, такой как устанавливаемый модуль CPAN или пакет RubyGem, либо скопировать исходный код в дерево собственного проекта. Проблема с подключением библиотеки в том, что библиотеку сложно хоть как-то модифицировать под свои нужды, и зачастую её сложнее распространять. Ведь вы вынуждены удостовериться в том, что эта библиотека доступна на каждом клиенте.

Проблема с включением кода в ваш собственный проект в том, что любые изменения, вносимые вами, могут конфликтовать с изменениями, которые появятся в основном проекте, и эти изменения будет сложно слить.

Git решает эту задачу используя подмодули (submodule). Подмодули позволяют содержать репозиторий Git как подкаталог другого репозитория Git. Это даёт возможность клонировать ещё один репозиторий внутрь проекта и держать 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 rackнаходится 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 statuscразу после добавления подмодуля, то увидите две вещи:

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

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 gitstatusприсутствует элемент rack. Если вы выполните git diffдля 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 rackявляется 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, который по существу означает, что в качестве записи в каталоге сохраняется коммит, а не подкаталог или файл.

Вы можете расценивать каталог rackкак отдельный проект и обновлять ваш основной проект время от времени с указателем на самый последний коммит в данном подпроекте. Все команды Git работают независимо в двух каталогах:

```
                                    $ 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
```

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 sub- module 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.
```

Теперь ваш подкаталог rackточно 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

# Changed but not updated:

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

```
#
       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
```

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 — 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 updateиз 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 updatев следующий раз, она все 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 все 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

#    (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 sub- module 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 еще 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/rackобратно 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 в том, что вы имеете два проекта, и один из проектов отображается в подкаталог другого и наоборот. Если вы зададите в качестве стратегии слияния метод sub- tree, то Git будет достаточно умным, чтобы понять, что один из проектов

является поддеревом другого и выполнит слияние в соответствии с этим. И это довольно удивительно.

Сначала добавьте приложение Rack в свой проект. Добавьте проект Rack как внешнюю ссылку в свой собственный проект, и затем поместите его в собственную 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.
```

Теперь у вас есть корень проекта Rack в ветке rack_branchи ваш проект в ветке mas- ter. Если вы переключитесь на одну ветку, а затем на другую, то увидете, что содержимое их корневых каталогов различно:

```
$ ls
AUTHORS      KNOWN-ISSUES   Rakefile      contrib         lib
$ git checkout master

Switched to branch "master"

$ ls
```

Допустим, вы хотите поместить проект Rack в подкаталог своего проекта в ветке mas- ter. Вы можете сделать это в Git’е командой git read-tree. Вы узнаете больше про команду read-treeи 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 ветвь rackв подкаталог rackвашего 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 вашу 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 sub- tree:

```
$ 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-treec веткой, с которой вы хотите сравнить:

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

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

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

Итоги

Вы познакомились с рядом продвинутых инструментов, которые позволяют вам манипулировать вашими коммитами и индексом более совершенно. Когда вы замечаете проблему, то сможете легко выяснить, каким коммитом она внесена, когда и кем. Если вы хотите использовать подпроекты в вашем проекте — вы узнали несколько путей, как приспособится к этим нуждам. С этого момента, вы должны быть в состоянии делать

большинство вещей в Git, которые вам будут необходимы повседневно в командной строке, и будете чувствовать себя при этом комфортно.