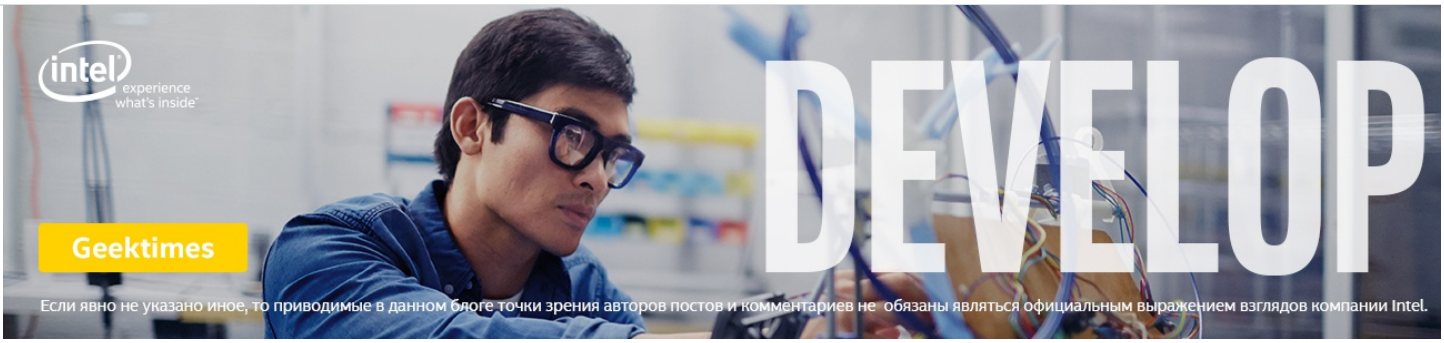


Intel 142,52
Компания

vikky13 20 декабря 2017 в 08:58

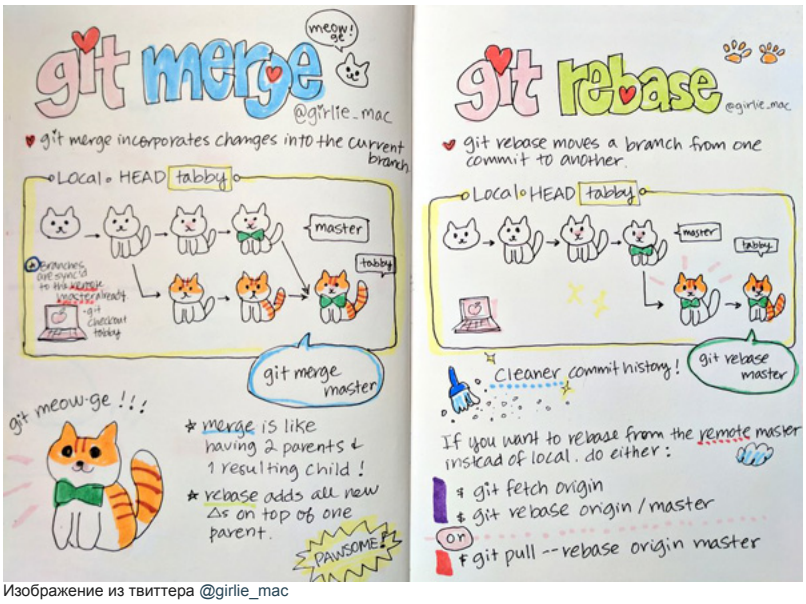
Git снизу вверх

https://jwiegley.github.io/git-from-the-bottom-up/

Системы управления версиями, Программирование, Git, Блог компании Intel

Перевод

У этого перевода не совсем обычная история. Системы контроля версий далеки от моих профессиональных интересов. Для рабочих проектов они мне требовались нечасто, причем, разные, так что, каждый раз, когда возникала такая необходимость, я заново вспоминала, как в них делается та или иная операция. А для личных проектов мне хватало возможностей Dropbox, хранящей историю версий файлов.



Изображение из твиттера @girlie_mac

Но вот однажды я на три незабываемых дня попала в роддом — это иногда случается с женщинами. Из развлечений у меня были новорожденн дочь и телефон с большим экраном. Дочь поначалу развлекала плохо (дома она быстро исправилась), а на телефоне помимо книг и фильмов обнаружился текст «Git from the bottom up», который оказался более чем годным… С тех пор прошло почти 3 года, ~~подросшей дочке уже пора самой начинать использовать Git~~ Git стал мейнстримом, если не сказать стандартом в современной разработке, а я с удивлением обнаружила, что перевода на русский этого чуда, полезного не только начинающим, но и продвинутым пользователям Git, до сих пор нет. Исправляю эту ситуацию.

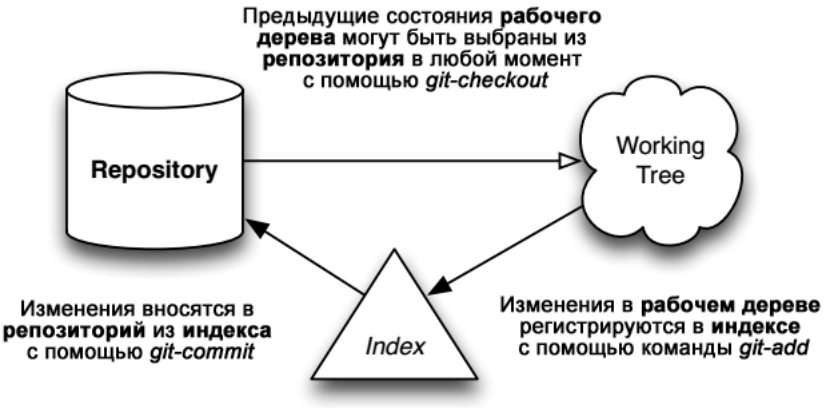
Добро пожаловать в мир Git. Хотя при взгляде снаружи масса возможностей Git и кажется сбивающей с толку, при взгляде снизу вверх Git прекрасен и прост. Я надеюсь, что данный документ поможет вам разобраться с этой могучей системой контроля версий.

И начнем мы со списка терминов, встречающихся в тексте и необходимых для его понимания.

- **Рабочее дерево (Working tree)** — Любая директория в вашей файловой системе, связанная с *репозиторием* (что можно видеть по наличи в ней поддиректории «.git»). Включает в себя все файлы и поддиректории.
- **Коммит (Commit)**. В роли существительного: «моментальный снимок» *рабочего дерева* в какой-то момент времени. В роли глагола: коммитить (закоммитить) — добавлять *коммит* в *репозиторий*.
- **Репозиторий (Repository)** — это набор *коммитов*, т.е. просто архив прошлых состояний *рабочего дерева* проекта на вашей или чьей-то машине.
- **Ветка (Branch)** — просто имя для *коммита*, также называемое *ссылкой* (reference). Определяет происхождение — «родословную» *коммит* и таким образом, является типичным представлением «ветки разработки»
- **Checkout** — операция переключения между *ветками* или восстановления файлов *рабочего дерева*
- **Метка (Tag)** — также имя для *коммита*, отличающееся от *ветки* тем, что оно всегда постоянно указывает на один и тот же *коммит*, а также может иметь свое текстовое описание
- **Мастер (Master)**. Условно «главная» или «основная» *ветка репозитория*, но по сути ничем не отличающаяся от прочих *веток*
- **Индекс (Index)**. В отличие от других подобных инструментов, Git не передает изменения из *рабочего дерева* в *репозиторий* напрямую. Вместо этого изменения сначала регистрируются в *индексе*, или «области подготовки» (staging area). Это можно рассматривать как способ «подтверждения» ваших изменений перед совершением *коммита*, который запишет в *репозиторий* все одобренные изменения.
- **HEAD** — заголовок. Используется *репозиторием* для определения того, что выбрано с помощью *checkout*
  - Если субъект *checkout* — ветка, то *HEAD* будет ссылаться на нее, показывая, что имя ветки должно быть обновлено во время следующе *коммита*
  - Если субъект *checkout* — коммит, то *HEAD* будет ссылаться только на него. В этом случае *HEAD* называется обособленным (detached)

Взаимодействие с Git обычно выглядит так:

После создания репозитория работа происходит в рабочем дереве. Как только достигается значительная веха — устранение бага; конец рабоче дня; момент, когда, наконец, все начинает компилироваться — вы добавляете свои изменения в индекс. Как только все, что вы собираетесь коммитить, оказывается в индексе, вы записываете его содержимое в репозиторий. На диаграмме ниже — типичный цикл жизни проекта:



Теперь давайте посмотрим, как каждая из показанных на картинке сущностей работает в git

Репозиторий: отслеживание содержимого директории

Как следует из определений выше, то, что делает Git — элементарно: он содержит моментальные снимки содержимого директории. Большую часть его внутреннего дизайна можно понять в терминах этой основной задачи.

Дизайн репозитория Git во многих отношениях воспроизводит файловую систему UNIX. Файловая система начинается с корневой директории, которая обычно состоит из других директорий, множество которых имеет узлы-листья, т.е. файлы, содержащие данные. Метаданные файлов

хранятся как в директории (имена), так и в i-узлах, которые ссылются на содержимое этих файлов (размер, тип, разрешения доступа и тп), Каждый i-узел имеет уникальный номер, идентифицирующий содержимое соответствующего файла. Хотя в директории может существовать множество объектов, указывающих на конкретный i-узел (т.е. хард-линки), именно i-узел «владеет» контентом, хранящимся в вашей файловой системе.

Внутренняя архитектура Git имеет поразительно схожую структуру с одним небольшим отличием.

Все начинается с того, что Git представляет содержимое ваших файлов в виде так называемых «фрагментов» («blobs»), которые являются узлами-листьями в структуре, очень похожей на директорию и называемой деревом. Так же как i-узел однозначно идентифицируется присвоенным ему системой номером, фрагмент в Git маркируется путем вычисления SHA-1 хэша от его размера и содержания. Для всех мыслимых применений это всего лишь произвольный номер, как и у i-узла, за исключением двух дополнительных свойств: во-первых, он контролирует неизменность содержимого фрагмента, а во вторых гарантирует, что одно и то же содержимое будет всегда представлено одним и тем же фрагментом независимо от того, где оно будет встречаться — в разных коммитах, репозиториях, или даже в разных частях Интернета. Если несколько деревьев ссылаются на тот же фрагмент, то это похоже на хард-линки: фрагмент не исчезнет из вашего репозитория до тех пор пока на него существует хотя бы одна ссылка.

Разница между файлом в файловой системе и фрагментом в Git состоит в том, что сам фрагмент не хранит метаданных о его содержимом. Вся эта информация хранится в дереве, к которому принадлежит фрагмент. Одно дерево может считать это содержимое файлом «foo», созданным августе 2004, в то время как другое дерево может знать то же содержимое под именем файла «bar», созданным на пять лет позже. В нормальной файловой системе такие два файла с совпадающим содержимым, но различающимися метаданными, будут всегда представлены как два независимых файла.

Чем вызвано такое отличие? В основном, тем, что файловая система спроектирована для поддержки изменяющихся файлов, а Git — нет. Имен из-за неизменности данных в репозитории, Git потребовался новый дизайн. И, как оказалось, этот дизайн дал возможность более компактного хранения — ведь все объекты с идентичным содержимым будут общими независимо от их местонахождения.

Знакомимся с фрагментом

Теперь, когда общая картина нарисована, посмотрим на практические примеры. Создадим тестовый репозиторий и покажем, как Git работает в нем с самого низа. (*прим. переводчика — все примеры этого текста проверены на git version 2.13.0.windows.1*)

```
$ mkdir sample; cd sample
$ echo 'Hello, world!' > greeting
```

Здесь я создал новую директорию *sample*, содержащую файл с прозаично предсказуемым содержимым. Пока я даже не создал репозитория, но уже могу начать использовать некоторые команды Git, чтобы понять, что он собирается делать. Для начала я хочу узнать, под каким хэшем Git будет хранить мое приветствие

```
$ git hash-object greeting
af5626b4a114abcb82d63db7c8082c3c4756e51b
```

При запуске этой команды на вашей системе вы получите тот же хэш-идентификатор (*прим. переводчика: далее в тексте — хэш id*). Несмотря на то, что мы создаем два различных репозитория (возможно, в различных частях света), наши фрагменты с приветствиями будут иметь одинаковый хэш id.

Я даже смогу получать коммиты из вашего репозитория в мой, и Git поймет, что мы отслеживаем одно и то же содержимое, и, соответственно, будет хранить только одну его копию.
Следующий шаг — это инициализация нового репозитория и коммит в него:

```
$ git init
$ git add greeting
$ git commit -m "Added my greeting"
```

На этой стадии наш фрагмент должен быть в системе и использовать, как мы и ожидали, определенный выше хэш id. Для удобства Git требует только начальные цифры хэша, однозначно определяющие фрагмент в репозитории. Обычно 6 или 7 цифр для этого достаточно.

```
$ git cat-file -t af5626b
blob
$ git cat-file blob af5626b
Hello, world!
```

Вот оно! Я даже не посмотрел, в каком коммите или в каком дереве это находится, но основываясь исключительно на содержимом, я смог предположить, что оно там есть, и не ошибся. Данное содержимое будет иметь тот же самый идентификатор независимо от времени жизни репозитория или положения файла в нем. То есть, данные гарантированно сохранены навсегда.

Таким образом, фрагмент — это фундаментальная единица данных в Git. На самом деле вся эта система — просто управление фрагментами.

Фрагменты хранятся в деревьях

Уникальное содержимое ваших файлов хранится во фрагментах, но сами эти фрагменты совершенно безлики. У них нет ни имени, ни структуры фрагменты — они и есть фрагменты. Поэтому для представления структуры и имен ваших файлов Git прикрепляет фрагменты в качестве узло — «листьев» к дереву.

Сейчас я не могу узнать, к какому дереву (деревьям) принадлежит фрагмент, так как он может иметь очень много владельцев. Но я знаю, что он точно должен быть где-то в дереве, которым владеет только что созданный мной коммит.

```
$ git ls-tree HEAD
100644 blob af5626b4a114abcb82d63db7c8082c3c4756e51b greeting
```

Т.е этот первый коммит, добавивший мой файл с приветствием в репозиторий, содержит одно дерево с единственным листом: фрагментом с приветствием.

Хотя с помощью команды *ls-tree HEAD* я могу посмотреть на дерево, содержащее мой фрагмент, но я еще не видел базовый объект дерева, на который ссылается данный коммит.
Вот еще несколько команд, чтобы подчеркнуть это различие и исследовать мое дерево:

```
$ git rev-parse HEAD
588483b99a46342501d99e3f10630cfc1219ea32 # отличается на вашей системе
$ git cat-file -t HEAD
commit
$ git cat-file commit HEAD
tree 0563f77d884e4f79ce95117e2d686d7d6e282887
author John Wiegley <johnw@newartisans.com> 1209512110 -0400
committer John Wiegley <johnw@newartisans.com> 1209512110 -0400
Added my greeting
```

Первая команда декодирует заголовок HEAD в коммит, на который он ссылается, вторая проверяет его тип, а третья показывает хэш id дерева, которым владеет коммит, а также другую информацию, хранимую в коммите. Хэш id коммита уникален для моего репозитория, так как он включает мое имя и дату совершения коммита, но хэш id дерева должен быть одинаковым для моего и вашего примеров, так как он содержит один и тот фрагмент под тем же именем.
Убедимся, что это и правда тот же объект:

```
$ git ls-tree 0563f77
100644 blob af5626b4a114abcb82d63db7c8082c3c4756e51b greeting
```

То есть, мой репозиторий содержит единственный коммит, который ссылается на дерево, содержащее фрагмент с тем, что я хочу записать.

Вот еще одна команда, которую я могу использовать для подтверждения этого:

```
$ find .git/objects -type f | sort
.git/objects/05/63f77d884e4f79ce95117e2d686d7d6e282887
```

```
.git/objects/58/8483b99a46342501d99e3f10630cfc1219ea32
.git/objects/af/5626b4a114abcb82d63db7c8082c3c4756e51b
```

Отсюда видно, что весь мой репозиторий содержит 3 объекта, хэш id которых мы уже видели в предыдущих примерах. Давайте просто из любопытства бросим последний взгляд на типы этих объектов:

```
$ git cat-file -t 588483b99a46342501d99e3f10630cfc1219ea32
commit
$ git cat-file -t 0563f77d884e4f79ce95117e2d686d7d6e282887
tree
$ git cat-file -t af5626b4a114abcb82d63db7c8082c3c4756e51b
blob
```

Я также мог использовать команду *show* для просмотра краткого содержания каждого из этих объектов, но оставлю это упражнение читателям.

Как образуются деревья?

Каждый коммит содержит единственное дерево. Но как образуются деревья? Мы знаем, что фрагменты создаются при помощи «нарезки» содержимого вашего файла, и что деревья владеют фрагментами, но мы еще не видели, как образуются эти деревья и как деревья связываютс со своими родительскими коммитами.

Давайте снова начнем с нового репозитория, но на этот раз сделаем все вручную.

```
$ rm -fr greeting .git
$ echo 'Hello, world!' > greeting
$ git init
$ git add greeting
```

Все начинается с добавления файла в индекс.Пока можно считать, что индекс — это то, что вы используете для первоначального создания фрагментов из файлов. Когда я добавил файл *greeting*, в моем репозитории произошли изменения. Хотя это еще и не коммит, есть способ на ни посмотреть:

```
$ git log  # не сработает -коммиты отсутствуют!
fatal: bad default revision 'HEAD'
$ git ls-files --stage # покажет фрагменты, на которые ссылается индекс
100644 af5626b4a114abcb82d63db7c8082c3c4756e51b 0 greeting
```

Что это? Коммитов еще нет, а объект уже есть. У него тот же хэш id, с которого я начал все это предприятие, так что я знаю, что он представляе содержимое файла *greeting*. Я мог бы использовать команду *git cat-file -t* для этого хэш id, и увидел бы, что это — фрагмент — тот же самый, который я получил в первый раз, когда создал этот репозиторий (кто бы сомневался).

На этот фрагмент еще не ссылается ни дерево ни коммиты. Пока ссылка на него есть только в файле *.git/index*, содержащем ссылки на фрагменты и деревья, которые собственно и образуют текущий индекс. А теперь давайте создадим в репозитории дерево, на котором и будут висеть фрагменты:

```
$ git write-tree # записать содержимое индекса в дерево
0563f77d884e4f79ce95117e2d686d7d6e282887
```

Знакомое число, не правда ли? Дерево, содержащее одни и те же фрагменты (и под-деревья) будет всегда иметь тот же самый хэш. Хотя у мен до сих пор нет объекта коммита, но зато в этом репозитории есть объект дерево, содержащее фрагменты. Цель низкоуровневой команды *write-tree* — взять содержимое индекса и поместить его в новое дерево для дальнейшего создания коммита.

Новый объект коммита можно создать вручную используя данное дерево напрямую. Именно это и делает команда *commit-tree* — берет хэш id дерева и создает для него объект коммита. Если бы я хотел, чтобы у коммита был родительский объект, я должен был бы указать его явно с

использованием ключа -p.

```
$ echo "Initial commit" | git commit-tree 0563f77
5f1bc85745dcccce6121494fdd37658cb4ad441f
```

Заметьте, что полученный хэш id отличается от того, который получится на вашей системе потому, что коммит ссылается на мое имя и время ег создания, а эти детали будут всегда отличны от ваших.

Но на этом работа не заканчивается — я еще не зарегистрировал коммит в качестве нового заголовка текущей ветки:

```
$ echo 5f1bc85745dcccce6121494fdd37658cb4ad441f > .git/refs/heads/master
```

Эта команда сообщает Git, что ветка «master» теперь должна ссылаться на данный коммит.
Другой, более безопасный способ достичь той же цели — это использовать команду *update-ref*:

```
$ git update-ref refs/heads/master 5f1bc857
```

После создания ветки *master*, мы должны связать с ней наше дерево. Это обычно происходит когда вы переключаете ветку:

```
$ git symbolic-ref HEAD refs/heads/master
```

Эта команда создает символическую ссылку *HEAD* на ветку *master*. Это очень важно, так как все дальнейшие коммиты из рабочего дерева тепе будут автоматически обновлять значение *refs/heads/master*.

Трудно поверить, что это так просто, но теперь я могу использовать команду *log* для просмотра моего свежесозданного коммита.

```
$ git log
commit 5f1bc85745dcccce6121494fdd37658cb4ad441f
Author: John Wiegley <johnw@newartisans.com>
Date: Mon Apr 14 11:14:58 2008 -0400
Initial commit
```

Замечу, что если бы я не сделал так, чтобы *refs/heads/master* указывал на новый коммит, тот можно было считать недоступным — так как на нег никто не ссылается, и он не является родительским объектом другого доступного коммита. В этом случае объект коммита когда-нибудь будет удален из репозитория вместе со своим деревом и всеми фрагментами (это происходит автоматически при помощи команды «gc», которая ред используется пользователями Git вручную). При связывании коммита с именем в *refs/heads*, как мы сделали выше, он становится доступным, ч гарантирует его дальнейшее сохранение Git.

Вся прелесть коммитов

Некоторые системы контроля версий делают ветки магическими сущностями, часто выделяя их из основной линии разработки, в то время как другие обсуждают концепцию ветвлений так, будто она сильно отличается от коммитов. Но в Git ветки не являются отдельной сущностью — зде есть исключительно фрагменты, деревья и коммиты (ну и тэги, но они являются просто ссылками на коммиты, так что их можно игнорировать) Так как у коммита может иметься один и более родителей, и эти комммиты в свою очередь могут принадлежать своим родителям, мы можем рассматривать единичный коммит как ветвление — ведь он знает всю свою «родословную».

Вы можете в любой момент посмотреть все коммиты верхнего уровня используя команду branch

```
$ git branch -v
* master 5f1bc85 Initial commit
```

Повторяйте за мной: Ветка — это просто именованная ссылка на коммит.

Ветки и тэги идентичны за единственным исключением — тэги могут иметь свои собственные описания — подобно коммитам, на которые они ссылаются. Ветки -это просто имена, а тэги — это описания, можно сказать, этикетки.

По правде говоря, у нас нет реальной необходимости использовать эти «сущности- псевдонимы». Например, при желании, я мог бы ссылаться любой объект репозитория, используя исключительно хэш-id его коммитов.

Вот, например, команда, перезагружающая HEAD моего рабочего дерева на заданный коммит;

```
$ git reset --hard 5f1bc85
```

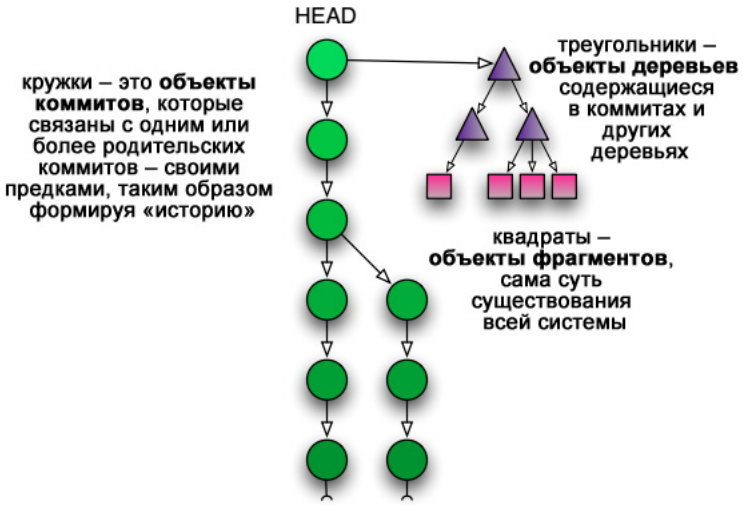
Ключ *--hard* вызывает удаление всех текущих изменений моего рабочего дерева независимо от того, были ли они зарегистрированы для будуще помещения в репозиторий (мы еще поговорим об этой команде ниже).

Более безопасный способ перейти к конкретному коммиту — это использовать команду *checkout*:

```
$ git checkout 5f1bc85
```

Разница с предыдущей командой состоит в том, что файлы, измененные в моем рабочем дереве, сохранятся. Если я добавлю к *checkout* ключ - то команда подействует так же, как и *reset -hard*, за исключением того, что checkout меняет только рабочее дерево, а *reset* -hard кроме этого ещ изменяет HEAD текущей ветви, чтобы он указывал на заданную версию дерева.

Еще одно достоинство системы, основанной на коммитах, это возможность перефразировать термины даже самых сложных систем контроля версий на простом языке. Например, если у коммита несколько родителей, то это — объединенный (merged) коммит. Или, если у коммита несколько потомков, то он представляет собой предка ветки и тп. Но для Git между этими сущностями нет никакой разницы для него мир представляет собой просто набор коммитов, каждый из которых содержит дерево, ссылающееся на другие деревья и фрагменты, хранящие ва данные. Все, что сложнее этого — просто система условных обозначений.
Вот иллюстрация того, как все это работает:



Коммит под любым другим именем...

Понимание коммитов — ключ к постижению Git. Вы поймете, что достигли просветления, когда в вашем уме будут содержаться исключительно топологии коммитов, а не мешанина веток, тэгов, локальных и удаленных репозиториев и тп. Надеюсь, что это понимание не потребует от вас отрубать себе руки (как это сделал второй последователь учения Дзен), хотя я бы оценил, если к этому моменту у вас зародилось такое желани

Если коммиты — ключ, то их названия — дверь к мастерству. Существует огромное множество способов называть коммиты, группы коммитов и даже некоторые объекты, содержащиеся в коммитах, которые поддерживаются большинством команд Git. Вот сводка основных из них:

- **Имя ветки кода (branchname)** — Как было сказано выше, имя любой ветки — просто псевдоним самого недавнего коммита в эту ветку. Эт равносильно использованию слова HEAD при выборе данной ветки.

- **Имя тэга (tagname)** — Также, как и имя ветки — это имя коммита. Единственная разница — имя тэга никогда не меняется, в то время как и ветки изменяется каждый раз при поступлении туда нового коммита.
- **HEAD** — Алиас названия текущего выбранного коммита. Если вы выбираете определенный коммит — вместо имени ветки, то HEAD ссылается исключительно на него, а не на имя ветки. Это — специальный случай, называемый “использование отделенной головы” (я увере что тут должна быть какая-нибудь шутка).
- **c82a22c39cbc32…** — К коммиту можно всегда обратиться по его полному, 40-символьному хэш-id Обычно это происходит во время копирования и вставки, т.к. обычно для этого есть другие, более удобные способы.
- **c82a22c** — Вам необходимо использовать только ту часть хэш-id, которая однозначно идентифицирует коммит в репозитории. Обычно для этого достаточно 6-7 цифр.
- **name^** — Для ссылки на родителя любого коммита используется символ ^. В случае, когда у коммита более одного родителя (коммит слияния), берется первый из них. Если вам требуется n-ый родитель, то обратиться к нему можно как name^n
- **name^^** — Родитель родителя данного коммита. Эту последовательность можно продолжить…
- **name~10** — …. но не нужно. Чтобы обратиться к n-предку данного коммита используется ~n (что эквивалентно n символам ^ подряд)
- **name:path** — Для обращения к определенному файлу внутри дерева коммита, укажите имя файла после двоеточия. Это бывает полезно д команды show или для сравнения двух версий файла между коммитами:

  ```
  $ git diff HEAD^1:Makefile HEAD^2:Makefile
  ```

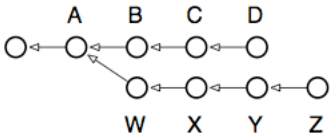
- **name^{tree}** — Вы можете обратиться не к самому коммиту, а к содержащему его дереву.
- **name1..name2** — Это и последующие наименования относятся к диапазону коммитов и очень полезны в командах типа log для просмотра изменений, сделанных в выбранный промежуток времени.
  В данном случае команда адресует все предшествующие коммиты начиная с name2 вплоть до (но не включительно!) name1. Если одно из этих имен будет опущено, то вместо него используется HEAD
- **name1...name2** — Троеточие в диапазоне — совсем не то, что две точки. Для команд типа log оно обозначает все коммиты, на которые ссылаются или name1 или name2, но не оба сразу. Результат — это список уникальных коммитов в обеих ветках.
- **--since=«2 weeks ago»** — Адресует все коммиты, начиная с заданной даты
- **--until=«1 week ago»** — Адресует все коммиты, вплоть до заданной даты
- **--grep=pattern** — Адресует все коммиты, чье сообщение подходит под заданный шаблон регулярного выражения
- **--committer=pattern** — Адресует все коммиты, внесенные в репозиторий человеком, информация о котором подходит под заданный шабло
- **--author=pattern** — Адресует все коммиты, информация об авторе которых подходит под заданный шаблон. Автор коммита — это создател изменений, которые коммит представляет. Для локальной разработки это — тот же самый человек, который вносит коммит. Но когда патчи посылаются по почте, автор и человек, реально совершающий коммит обычно отличаются.
- **--no-merges** — Адресует все коммиты с единственным родителем, т.е. игнорирует коммиты слияния

Большинство этих опций могут комбинироваться. Вот пример, показывающий, как получить информацию обо всех изменениях текущей ветки (отпочковавшейся от master), сделанных мной в последний месяц, и содержащих текст «foo».

```
$ git log --grep='foo' --author='johnw' --since="1 month ago" master..
```

Ветвления и сила rebase

Одна из самых действенных команд для работы с коммитами — это команда с непритязательным названием *rebase*. Работает она так: каждая существующая ветка в Git имеет один или более «базовых коммитов» — тех, от которых она и произошла. Давайте для примера посмотрим на следующий типичный сценарий (на рисунке ниже). Здесь стрелки показывают назад во времени так как каждый коммит ссылается на своего родителя (родителей), но не потомков. Следовательно, D и Z являются заголовками своих веток.



В этом можно убедиться при помощи команды

```
$ git branch
  Z
* D
```

И в деталях:

```
$ git show-branch
! [Z] Z
 * [D] D
--
 * [D] D
 * [D^] C
 * [D~2] B
+  [Z]Z
+  [Z^]Y
+  [Z~2] X
+  [Z~3] W
+* [D~3] A
```

К такому виду обозначений требуется привыкнуть, но, в сущности, это просто описание диаграммы выше. (*прим. переводчика: обратите внимание на пробелы в записи, они имеют значение — делят вывод на колонки*)

И вот что оно нам говорит:

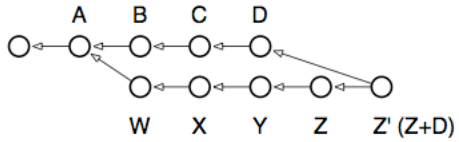
Наша текущая ветка впервые разделилась на коммите A (также известном как коммит
D~3, и даже Z~4 если вам так больше нравится. Для тех, кто пропустил таблицу выше, напомню, что синтакс *commit^* используется для обозначения родителя коммита, а *commit~3* — его предок третьего уровня, т.е. прадедушка.

- Если читать снизу вверх, первая колонка (со знаками +) показывает отпочковавшуюся ветку Z с четырьмя коммитами — W, X, Y и Z.
- Второй столбец (со звездочками) показывает коммиты, сделанные в текущей ветке (и символ * всегда ее обозначает), а именно три коммита — B, C и D.
- И, наконец, верхняя часть вывода, отделенная от низа разделительной линией, показывает список имеющихся веток, то, в какой колонке находятся их коммиты и каким символом они помечены.

Теперь нам требуется привести рабочую ветку Z в соответствие с основной D. Т.е. включить работу, сделанную в B, C и D в Z. В других системах контроля версий такие вещи делаются исключительно при помощи слияния веток (*branch merge*). В Git возможность слияния тоже есть. Она реализована командой *merge* и применяется когда Z — опубликованная ветвь, и мы не хотим менять ее историю коммитов. Вот необходимые для этого команды:

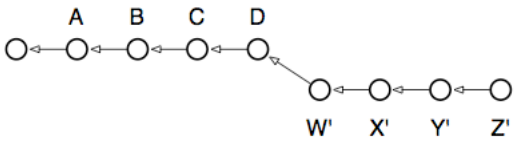
```
$ git checkout Z # переключаемся в ветку Z
$ git merge D # объединяем B, C и D в Z
```

Теперь репозиторий будет выглядеть так:



Если теперь мы выберем ветку Z, она будет содержать все, что там было ранее, объединенное с содержимым D *(тут переводчик тяжело вздыхает: реальная операция слияния потребовала бы разрешения конфликтов между состояниями D и Z)*.

Хотя новый Z теперь и содержит изменения из D, но он также включает новый коммит Z', представляющий собой слияние Z и D. Он не добавля ничего нового, но представляет работу по соединению Z и D. В каком-то смысле это — «мета-коммит» так как его содержимое относится исключительно к измерениям репозитория, а не к новой работе, сделанной в рабочем дереве.
Но в Git есть метод трансплантации Z непосредственно в D, фактически передвижение Z во времени — используя могущественную команду rebase. Вот граф, который мы хотим получить в результате:



То есть, наша локальная ветка разработки Z должна базироваться на последнем состоянии основной ветки D. Именно поэтому команда и называется *rebase* — она меняет базовый коммит ветки, из которой она вызывается. Если вы вызовете ее несколько раз, то сможете внести неограниченное количество изменений, оставаясь при этом в соответствии с основной веткой разработки, но не добавляя излишних коммитов слияния к вашей ветке разработки.

Вот команды, необходимые для выполнения вышеописанной операции с помощью rebase — сравните их с использованием merge выше:

```
$ git checkout Z # переключаемся в ветку Z
$ git rebase D  #изменяем базовый коммит  Z, чтобы он указывал на D
```

Хотя, в некоторых ситуациях *merge* предпочтительнее *rebase*.
У rebase имеется недостаток — даже если перебазированное рабочее дерево компилируется, это не гарантирует, что будут компилироваться промежуточные коммиты, так как они никогда не компилировались в своем перебазированном состоянии. Если историческая валидность кода для вас важна, выбирайте merge.

Заметьте, что при каждом перебазировании вы потенциально меняете каждый коммит в ветке. Раньше, когда W базировался на A, он содержал только изменения необходимые для преобразования A в W, а после перебазирования он будет содержать изменения, необходимые для преобразования D в W'. Даже преобразования из W в X изменятся, так как A+W+X теперь выглядит как D+W'+X' и так далее. Заметьте, что до с пор я подчеркивал, что ветка Z — локальная. Если бы это была ветка, чьи изменения видны публично, и кто-нибудь породил свои локальные ветки от Z, то теперь их ветки будут указывать на старую Z, а не на новую Z'.

Обычно можно использовать следующее простое правило — используйте *rebase* для локальных веток, не имеющих потомков, а для всех остальных случаев — *merge*. Также merge будет полезен тогда, когда вы захотите вернуть ваши локальные изменения в основную ветку разработки.

Интерактивный rebase

При использовании *rebase* выше, он автоматически переписывал все коммиты от W до Z, для перебазирования ветки Z в коммит D (в заголовочный коммит ветви D). Однако, вы можете полностью контролировать происходящее вручную. Если задать ключ -i для rebase, вы сможете выбрать то, что будет сделано для каждого коммита в локальной ветви Z.

Выбор осуществляется из следующих опций.

- **pick** (выбрать) — Поведение по умолчанию, используемое для каждого коммита в ветке, если вы не используете интерактивный режим. Он означает, что данный коммит должен быть применен к своему (теперь уже переписанному) родительскому коммиту. При возникновении конфликтов rebase предложит вам их разрешить.
- **squash** (сплющить)— Содержимое коммита будет «вложено» в содержимое предыдущего коммита. Это может быть сделано любое количество раз. Если взять вышеиспользованную для примера ветку и сплющить все ее коммиты (конечно же, кроме первого, который должен быть просто выбран), то новая ветка Z будет иметь единственный коммит поверх D. Это может быть полезным, если ваши изменени размазаны по множественным коммитам, но вы хотите, чтобы в истории они показывались как единый коммит.

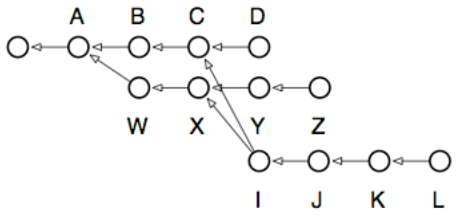
- **edit** (редактировать) — если вы пометите коммит таким образом, то процесс rebase остановится, а вы попадете в оболочку, где сможете совершить дополнительные необходимые изменения, после чего команда rebase --continue продолжит исполнение так, будто внесенные изменения были там изначально.
- **drop** (выбросить) — если вы удалите коммит из интерактивного rebase, то он просто пропадет — так, будто никогда и не был сделан. Заметьте, что в этом случае возможны конфликты слияния, в случае если более поздние коммиты зависели от данных изменений.

Всю мощь команды rebase трудно оценить с первого взгляда, но она дает вам практически неограниченный контроль над видом каждой ветки кода. Вы можете использовать rebase для:

- объединения нескольких коммитов в один,
- переупорядочивания коммитов,
- удаления нежелательных изменений, о которых вы начали сожалеть,
- перемещения вашей ветки на любой существующий коммит в репозитории,
- изменения единичного коммита спустя значительное время после его создания.

В общем, я бы рекомендовал вам почитать мануал по *rebase*, там содержится несколько хороших примеров использования всей мощи этой команды. Здесь же приведу заключительный пример, демонстрирующий супер-возможности *rebase*.

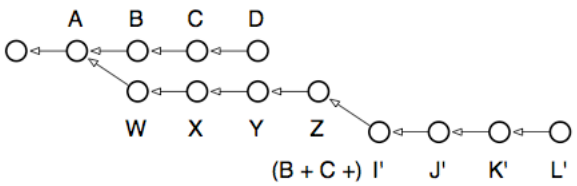
Представьте себе следующий сценарий и подумайте, что бы вы делали, если в один прекрасный день вы бы захотели, чтобы вспомогательная ветка I стала новым заголовком Z:



То есть, у нас есть основная линия разработки — D, которая три коммита назад была разветвлена для начала гипотетической разработки Z. В какой-то момент в середине этого процесса, когда еще C и X были заголовками своих соответствующих веток, мы решили начать еще одну исследовательскую ветку, которая в итоге произвела L. После чего мы обнаружили, что код в L — хороший, но не настолько, чтобы влить его назад в основную линию разработки D, так что мы решили переместить изменения в ветку Z, чтобы все изменения были там. Но в процессе мы осознаем, что необходимо быстро подправить J, так как там в копирайте стоит неверный год. Вот команды, необходимые для распутывания это узла:

```
$ git checkout L
$ git rebase -i Z
```

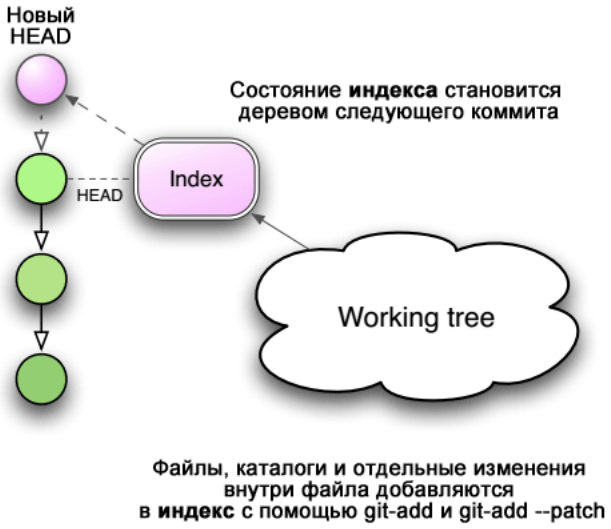
После разрешения возможных конфликтов *(тут переводчик снова тяжело вздыхает)*, репозиторий будет выглядеть так:



Индекс: Встречайте посредника

Между вашими файлами, хранящимися в файловой системе и фрагментами Git, хранящимися в репозитории, стоит в некоторой мере загадочн сущность — Git индекс. Понять его непросто отчасти потому, что ему не повезло с названием. Это индекс в том смысле, что он ссылается на набор новых деревьев и фрагментов, которые вы создали с помощью команды add. Эти новые объекты, возможно, скоро будут объединены в новое дерево с целью коммита в ваш репозиторий. Но до тех пор на них ссылается только индекс. Это значит, что если вы разрегистрируете

изменения в индексе с помощью reset, то ваши фрагменты будут, фактически потеряны, и впоследствии удалены. Индекс — это просто область подготовки для вашего следующего коммита, и для его существования есть уважительная причина: он поддерживает модель разработки, котора может показаться чужеродной пользователям CVS и Subversion, но более чем знакомой пользователям Darcs — возможность построения коммита в несколько стадий.



Но, при этом, есть способ почти полностью игнорировать индекс — путем передачи ключа -a команде commit. Давайте посмотрим, как, наприме работает другая система контроля версий — Subversion.

В ней имеется команда *svn status*, выдающая список действий, которые будт применены к вашему репозиторию при следующем вызове *svn commit*. Этот «список следующих действий» и есть в некотором роде неформальный индекс, определяемый путем сравнения состояния вашего рабочего дерева с состоянием HEAD. Если какой-либо файл был изменен, при следующем коммите эти изменения будут сохранены. Если напротив имени файла в этом списке стоит знак вопроса, то он будет игнорирован, а новый файл, добавленный с помощью *svn add* будет помещен в репозиторий.

Именно это и происходит при использовании вами в Git команды *commit -a*: новые неизвестные файлы игнорируются, а новые, добавленные в команде add, попадут в репозиторий, так же, как и любые изменения существующих файлов.

Это действие почти идентично тому, что делает *Subversion*, за единственным исключением: в случае *Subversion* «список следующих действий» всегда определяется по текущему рабочему дереву, а в Git этот лист представляет собой содержимое индекса, то есть, то, что станет следующ состоянием HEAD. Такая предварительная подготовка изменений дает вам дополнительный уровень контроля — возможность повлиять на то, что произойдет при последующем выполнении *commit*.

Если это еще не совсем понятно, то давайте рассмотрим следующий пример: пусть у вас имеется исходный файл foo.c, в котором вы сделали два набора независимых изменений. И теперь вы хотите разделить эти изменения на 2 отдельных коммита, каждый со своим описанием. Вот ка это делается в *Subversion*:

```
$ svn diff foo.c > foo.patch
$ vi foo.patch
<редактируем foo.patch, сохраняя изменения, которые я хочу позже включить в коммит>
$ patch -p1 -R < foo.patch # убираем второй набор изменений
$ svn commit -m "Сообщение первого коммита"
$ patch -p1 < foo.patch # применяем оставшиеся изменения
$ svn commit -m "Сообщение второго коммита"
```

Правда весело? А теперь повторите эти операции многократно для сложного динамического набора изменений.

А теперь версия той же операции для Git, использующая индекс:

```
$ git add --patch foo.c
<интерактивно выбираем изменения для первого коммита>
$ git commit -m "Сообщение первого коммита"
```

```
$ git add foo.c # добавляем оставшиеся изменения
$ git commit -m "Сообщение второго коммита"
```

Reset или не reset?

Одна из самых сложных команд для освоения в Git это *reset* — обычно она смущает больше, чем другие команды. Что вполне понятно, учитыва возможность *reset* изменять как ваше рабочее дерево, так и текущий HEAD. Так что будет полезно привести здесь краткий обзор работы этой команды.

Можно сказать, что *reset* — это редактор ссылок, индекса и рабочего дерева. Столько всего — немудрено и запутаться. Давайте посмотрим на разницу между этими тремя режимами и на то, как они вписываются в модель коммитов Git.

Делаем комбинированный reset

Если вы используете опцию --mixed (или вообще не используете опций, так как mixed — берется по умолчанию), *reset* вернет ваш индекс, а такж HEAD к состоянию выбранного коммита. Основное отличие этой опции от -soft состоит в том, что -soft меняет исключительно значение HEAD, н затрагивая индекс.

```
$ git add foo.c # добавляем изменения в индекс в виде нового фрагмента
$ git reset HEAD # убираем все изменения, внесенные в индекс
$ git add foo.c # мы ошиблись, возвращаем все назад
```

Делаем soft reset

Использование reset опции -soft эквивалентно простому переводу HEAD на другой коммит. Изменений в рабочем дереве при этом не происходи Это означает эквивалентность двух нижеследующих команд:

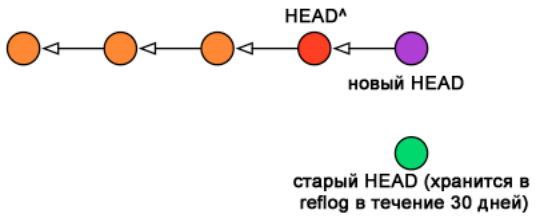
```
$ git reset --soft HEAD^ # перевести HEAD на ее родителя, т.е. фактически игнорировать последний коммит
$ git update-ref HEAD HEAD^ # делает то же самое вручную
```

В обоих случаях ваше рабочее дерево теперь соответствует предыдущему состоянию HEAD, так что если запустить *git status*, вы увидите больш изменений. И это — не реальное изменения в ваших файлах, а просто отражение того, что они теперь сравниваются с предыдущими версиями Это дает вам шанс создать новый коммит вместо старого.

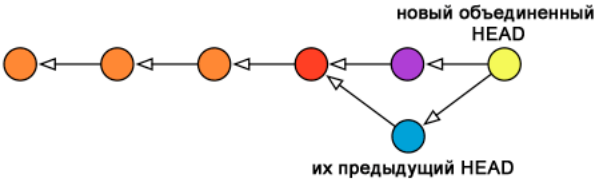
Тут надо упомянуть, что если вы хотите изменить самый недавний выбранный коммит, то для этого существует команда *commit -amend*, добавляющая ваши новейшие изменения к последнему коммиту так, как будто вы внесли их туда изначально.

Но, пожалуйста, заметьте: если у вашего проекта есть внешние пользователи и они сделали что-то поверх вашего предыдущего HEAD, того, который вы отбросили с помощью *reset*, то это приведет к автоматическому слиянию (*merge*) после того, как пользователи возьмут новую верси проекта.

Вот как будет выглядеть ваше дерево после soft reset и нового коммита:



А вот как будет выглядеть HEAD ваших пользователей после того как они возьмут новую версию — цвета на рисунке показывают соответствие различных коммитов:



Делаем hard reset

Опция --hard у команды *reset* — штука потенциально опасная, так как если вы сделаете hard reset для текущего HEAD, то это приведет к стиранию всех изменений в вашем рабочем дереве так, что ваши текущие файлы станут соответствовать содержимому HEAD.

Существует еще одна команда — *checkout*, которая действует так же, как и reset --hard, но только в случае, если индекс пуст. В противном случа она приводит ваше рабочее дерево в соответствие с индексом.

Теперь, если вы сделаете hard reset до какого-нибудь раннего коммита, то это будет эквивалентно использованию сначала *reset --soft*, а потом *reset -hard* для перезагрузки вашего рабочекго дерева. То есть, следующие команды равнозначны:

```
$ git reset --hard HEAD~3 #  Возвращаемся назад во времени, отбрасывая изменения

$ git reset --soft HEAD~3 # Устанавливаем HEAD на ранний коммит
$ git reset --hard #  Стираем изменения в рабочем дереве
```

Как можно видеть, hard reset может быть очень разрушительным. К счастью, существует более безопасный способ достижения того же самого эффекта — с использованием *git stash* (см. следующий раздел)

```
$ git stash
$ git checkout -b new-branch HEAD~3 # заголовок вернулся назад во времени!
new HEAD
HEAD
```

У данного подхода есть два значительных преимущества в случае, когда вы не уверены в том, что действительно хотите радикально изменить текущую ветку:

1. Он сохраняет вашу работу в хранилище (stash), откуда вы можете забрать ее назад в любой момент. Заметьте, что это хранилище не привязано к конкретной ветке, так что вы можете сохранить состояние вашего рабочего дерева в одной ветке, а позже наложить отличия на другую ветку.
2. Stash возвращает ваше дерево в прошлое состояние, но в новой ветке. Так то если вы решите сделать коммит с вашими изменениями по сравнению с прошлым состоянием, вы не измените вашу оригинальную ветку.

Если же вы вносите изменения в новую ветку new-branch и затем решаете, что она должна стать вашей новой master, то необходимо выполнить следующие команды:

```
$ git branch -D master # прощай, старый master (пока еще сохранится в reflog)
$ git branch -m new-branch master # new-branch становится master
```

Мораль этого раздела такова: хотя вы и можете делать значительные оперативные вмешательства в вашу текущую ветку с использованием *res --soft* и *reset --hard* (который также изменяет рабочее дерево), я не вижу причин, по которым это следует делать. Git делает работу с ветками столь простой и дешевой, что почти всегда стоит сделать ваши деструктивные изменения в новой ветке, а уже затем переместить ее, чтобы она заняла место вашей старой master. В этом состоит настоящая мудрость Джедаев!

А что если вы случайно выполните *reset -hard*, потеряв не только ваши текущие изменения, но и коммиты из вашей master? Если вы не использовали stash для создания моментальных снимков состояния (см. следующий раздел), то ваше рабочее дерево утеряно безвозвратно.

Единственное, что можно сделать, это восстановить вашу ветвь к ее предыдущему состоянию с использованием reflog (также описан в

следующем разделе):

```
$ git reset --hard HEAD@{1} # восстановить из  reflog перед изменением
```

Техника безопасности работы в Git требует использования *stash* перед *reset -hard*. Это впоследствии убережет вас от появления седых волос. Если вы использовали *stash*, то его можно использовать для восстановления вашего рабочего дерева следующим образом:

```
$ git stash # просто потому, что это хорошо бы сделать
$ git reset --hard HEAD~3 # возвращаемся назад во времени
$ git reset --hard HEAD@{1} # ААААА! это была ошибка, давайте вернем все назад
$ git stash apply # возвращаем прежнее состояние нашего рабочего дерева.
```

Последние звенья в цепи: stash и reflog

До сих пор мы описали два отрезка пути, которыми фрагменты попадают в Git: сначала они создаются в индексе, без родительского дерева и владеющего ими коммита, а потом они коммитятся в репозиторий, где и существуют в качестве листьев, висящих на дереве, содержащемся в данном коммите.

Но, оказывается, существуют еще два места пребывания фрагментов в вашем репозитории. Первый — Git reflog, мета-репозиторий, записывающий в форме коммитов все изменения в вашем репозитории. Это означает, что когда вы создаете дерево из своего индекса и сохраняете его в коммите (все это делает команда *commit*), вы также, незаметно для себя добавляете этот коммит в reflog, посмотреть который можно при помощи следующей команды:

```
$ git reflog
5f1bc85... HEAD@{0}: commit (initial): Initial commit
```

Прелесть reflog состоит в его сохранении данных независимо от других изменений в вашем репозитории. Это означает, что я могу отвязать какс либо коммит от моего репозитория (используя *reset*), но ссылка на него останется в reflog еще на 30 дней, защищая его от автоматической «уборки мусора». Что дает мне месяц на возможное восстановление коммита, если вдруг возникнет такая необходимость.

Второе место существования фрагментов, хотя и косвенное — это само рабочее дерево. И вот что я имею в виду. Если вы, например, изменили файл foo.c, но еще не добавили эти изменения в индекс, Git мог еще не создать фрагментов, но эти изменения уже существуют, ведь существуе новое содержимое файла. Только оно находится в файловой системе, а не в репозитории Git. Несмотря на то, что реальных фрагментов там не у файла даже имеется свой SHA1 хэш id, посмотреть который можно так:

```
$git hash-object foo.c
<покажет  hash id>
```

В чем польза этого для вас? Предположим, что в конце тяжелого рабочего дня вы остановитесь на полностью раскуроченном рабочем дереве. Тогда хорошей привычкой будет отложить ваши изменения в stash: (*прим. перев.: один из интересных вариантов перевода слова stash на русский — «заныкать»*)

```
$ git stash
```

Эта команда возьмет все содержимое вашей директории, включая как рабочее дерево, так и содержимое индекса, и создаст для них: фрагмент в репозитории, дерево, содержащее эти фрагменты, а также пару stash коммитов — для рабочего дерева и индекса. Будет записано и время вашего обращения к stash.

На следующий день вы просто вытащите ваши изменения из stash с помощью *stash apply*, но при этом вас останется reflog всех отложенных изменений.

Вот, что вы сделаете, придя на работу следующим утром (здесь WIP — сокращение от «Work in Progress» — «Работаем…»:

```
$ git stash list
stash@{0}: WIP on master: 5f1bc85... Initial commit

$ git reflog show stash # тот же вывод плюс хэш id для stash коммита
2add13e... stash@{0}: WIP on master: 5f1bc85... Initial commit
$ git stash apply
```

Так как ваше сохраненное в stash рабочее дерево хранилось в коммите, вы можете работать с ним точно так же, как и с любой веткой кода в любое время. Это значит, что вы можете посмотреть лог, увидеть то, что вы сохранили и выбрать любое из ваших предыдущих рабочих деревьe начиная с момента их сохранения:

```
$ git stash list
stash@{0}: WIP on master: 73ab4c1... Initial commit
...
stash@{32}: WIP on master: 5f1bc85... Initial commit

$ git log stash@{32} # когда я это сделал?
$ git show stash@{32} # покажите мне, над чем я работал
$ git checkout -b temp stash@{32} # посмотрим на это старое рабочее дерево!
```

Последняя команда особенно могущественная: смотрите, я сейчас получил в свое распоряжение не включенное ни в какие коммиты рабочее дерево давностью более месяца! Я никогда не добавлял эти файлы в индекс, я просто использовал простой прием: вызов stash перед тем, как разлогиниться каждый день вечером (конечно, только если я вносил какие-то изменения в мое рабочее дерево), и соответствующий вызов *stas apply* при каждом новом логине.
Если вы захотите почистить свой stash — скажем, хранить там только действия за последние 30 дней, не используйте stash clear, вместо него пользуйтесь командой reflog expire

```
$ git stash clear # Только не это! Вы потеряете всю эту историю
$ git reflog expire --expire=30.days refs/stash
<покажет сохраненные узлы stash >
```

То есть stash позволяет вам использовать незаметный контроль версий в отношении самого рабочего процесса — различных состояний вашего рабочего дерева изо дня в день. При желании вы можете даже использовать stash на регулярной основе, при помощи следующего скрипта (Uni OS):

```
$ cat <<EOF > /usr/local/bin/git-snapshot
#!/bin/sh
git stash && git stash apply
EOF
$ chmod +x $_
$ git snapshot
```

И я не вижу никаких причин для того, чтобы не запускать этот скрипт автоматически каждый час, а каждую неделю — команду reflog expire

Выводы

За долгие годы я использовал множество систем контроля версий и схем бекапа. Все они включали средства получения предыдущего содержания файла. Многие из них могли показать, как файл менялся со временем. Многие позволяли вернуться назад во времени и начать альтернативную линию развития, а впоследствии соединить эту новую линию с текущей. Гораздо меньшее количество из них допускало тонкий контроль над этим процессом, позволяя вам решить, какие именно части вы хотите сделать публичными. Git может сделать все эти вещи, прич относительно просто — как только вы поймете его основы.

Это — не единственная система с подобным набором возможностей, и нельзя сказать, что она всегда предоставляет лучший интерфейс к ним. Но то, что делает Git, является отличным фундаментом для работы. Я могу представить множество новых систем, которые будут разработаны будущем для эффективного использования гибкости, предоставляемой Git.

Большинство других известных мне систем оставили ощущение достижения своего концептуального плато — то есть, того, что все их