

Магия **Git**

Ben Lynn

Магия Git
Ben Lynn

Содержание

1. Предисловие	1
1.1. Благодарности	1
1.2. Лицензия	2
2. Введение	3
2.1. Работа - это игра	3
2.2. Контроль версий	3
2.3. Распределенный контроль	4
2.3.1. Глупые предрассудки	5
2.4. Конфликты при слиянии	5
3. Базовые операции	7
3.1. Сохранение состояния	7
3.1.1. Добавление, удаление, переименование	7
3.2. Расширенная отмена/Восстановление	8
3.2.1. Возвраты	9
3.3. Создание списка изменений	9
3.4. Скачивание файлов	9
3.5. На острие ножа	10
3.6. Публичный доступ	10
3.7. Что я наделал?	11
3.8. Упражнение	11
4. Все о клонировании	13
4.1. Синхронизация компьютеров	13
4.2. Классический контроль исходного кода	13
4.3. Создание форка проекта	14
4.4. Окончательные бэкапы	15
4.5. Многозадачность со скоростью света	15
4.6. Другие системы контроля версий	15
5. Чудеса ветвления	17
5.1. Кнопка босса	17
5.2. Грязная работа	18
5.3. Быстрые исправления	19
5.4. Бесперебойный рабочий процесс	19
5.5. Собрать все в кучу	20
5.6. Управление Ветками	21
5.7. Временные Ветки	21
5.8. Работайте как вам нравится	22
6. Уроки истории	23
6.1. Оставаясь корректным	23
6.2. ... И кое-что еще	23
6.3. Локальные изменения сохраняются	24
6.4. Переписывая историю	24
6.5. Создавая Историю	25
6.6. Когда же все пошло не так?	26
6.7. Из-за кого все пошло наперекосяк?	27

6.8. Личный опыт.....	28
7. Групповая работа в Git.....	30
7.1. Кто я?.....	30
7.2. Git через SSH, HTTP	30
7.3. Git через что угодно	31
7.4. Патчи: Общее применения	31
7.5. К сожалению, мы переехали	33
7.6. Удаленные Ветки	33
7.7. Несколько Удаленных Веток	34
7.8. Мои Настройки	35
8. Гроссмейстерство Git.....	36
8.1. Релизы исходников	36
8.2. Сохранение изменений	36
8.3. Слишком большой коммит	36
8.3.1. Этапные изменения	37
8.4. Не теряй HEAD	37
8.5. Охота за HEAD'ами	38
8.6. Git как основа	39
8.7. Опасные трюки	40
8.8. Улучшаем свой публичный образ	41
9. Раскрываем тайны.....	43
9.1. Невидимость	43
9.2. Целостность.....	43
9.3. Интеллект	44
9.4. Индексация	44
9.5. Голые репозитории	44
9.6. Происхождение Git.....	45
9.7. База данных объектов	45
9.7.1. Blobs.....	45
9.7.2. Деревья	46
9.7.3. Коммиты.....	47
9.7.4. Неотличимо от магии.....	48
10. Недостатки Git	50
10.1. Недостатки SHA1	50
10.2. Microsoft Windows.....	50
10.3. Несвязанные файлы.....	50
10.4. Кто и что редактировал ?	50
10.5. История файлов.....	51
10.6. Начальное Клонирование	51
10.7. Изменчивые Проекты.....	51
10.8. Глобальный счетчик	52
10.9. Пустые подкаталоги	53
10.10. Первоначальный коммит	53
11. Приложение А: Перевод этого руководства.....	54

Глава 1. Предисловие

Git (<http://git.or.cz/>) это Швейцарский армейский нож контроля версий. Надежный универсальный многоцелевой инструмент контроля версий, чья чрезвычайная гибкость делает его сложным в изучении даже для многих профессионалов.

Как говорил Артур Кларк - любая достаточно развитая технология неотличима от магии. Это отличный способ подхода Git: новички могут игнорировать принципы его внутренней работы и рассматривать Git как гизмо, который может поражать друзей и приводить в бешенство врагов своими чудесными способностями.

Вместо того, чтобы вдаваться в подробности, мы предоставляем грубую инструкцию для конкретных действий. После частого использования вы постепенно поймете как работает каждый трюк и как адаптировать рецепты для ваших нужд.

Переводчики

- Китайский (упрощенный) (http://docs.google.com/View?id=dfwthj68_675gz3bw8kj): JunJie, Meng и JiangWei. Хостинг - Google Docs.
- Испанский ([/~blynn/gitmagic/intl/es/](http://blynn/gitmagic/intl/es/)): Rodrigo Toledo.

Другие Издания

- HTML одной страницей ([book.html](#)): один HTML без CSS.
- PDF файл ([book.pdf](#)): для печати.
- Пакет Debian (<http://packages.debian.org/search?searchon=names&keywords=gitmagic>): получите локальную копию этого сайта. Пакет Ubuntu (Jaunty Jackalope) (<http://packages.ubuntu.com/jaunty/gitmagic>). Также доступен когда этот сервер недоступен для сопровождающих (<http://csdcmf.stanford.edu/status/>).

1.1. Благодарности

Благодарю Dustin Sallings, Alberto Bertogli, James Cameron, Douglas Livingstone, Michael Budde, Richard Albury, Tarmigan, Derek Mahar и Frode Aannevik за внесенные предложения и улучшения. Спасибо Daniel Baumann за создание и сопровождение пакета для Debian. Также благодарен JunJie, Meng и JiangWei за перевод на китайский, и Rodrigo Toledo за перевод на испанский. [Если я забыл про вас, пожалуйста, сообщите мне, поскольку я часто забываю обновлять этот раздел.]

Мои благодарности остальным за вашу поддержку и похвалу. Если бы это была реальная физическая книга, я смог бы разместить Ваши щедрые отзывы о ней на обложке, чтобы прорекламировать ее! Seriously, я высоко ценю каждое ваше сообщение. Чтение их всегда поднимает мне настроение.

Бесплатные хостинги Git

- <http://repo.or.cz/> хостинг свободных проектов. Первый сайт Git-хостинга. Основан и поддерживается одним из первых разработчиков Git.
- <http://gitorious.org/> другой сайт Git-хостинга, направленный на проекты с открытым кодом.
- <http://github.com/> хостинг проектов с открытым кодом, и закрытых проектов на платной основе.

Большое спасибо каждому из этих участков для размещения данного руководства.

1.2. Лицензия

Это руководство выпущено под GNU General Public License version 3 (<http://www.gnu.org/licenses/gpl-3.0.html>). Естественно, что источник находится в репозитории Git, и можно получить, набрав:

```
$ git clone git://repo.or.cz/gitmagic.git # Создает каталог "gitmagic".
```

или с одного из зеркал:

```
$ git clone git://github.com/blynn/gitmagic.git  
$ git clone git://gitorious.org/gitmagic/mainline.git
```

Глава 2. Введение

Я буду использовать аналогии, чтобы объяснить, что такое контроль версий. Если вам нужно более точное объяснение, обратитесь к статье википедии (http://en.wikipedia.org/wiki/Revision_control).

2.1. Работа - это игра

Я играл в компьютерные игры почти всю свою жизнь. А вот использовать системы контроля версий я начал уже будучи взрослым. Подозреваю, что я не один такой, и сравнение этих двух занятий может помочь объяснению и пониманию их концепций.

Представьте, что редактирование кода, документа или чего-либо еще — игра. Далеко продвинувшись, вы захотите сохраниться. Чтобы сделать это, вы нажмете на кнопку "Сохранить" в вашем любимом редакторе.

Но это перезапишет старую версию. Это как в древних играх, где был только один слот для сохранения: вы можете сохраниться, но вы никогда не сможете вернуться к прежнему состоянию. Что досадно, так как одно из прежних сохранений может указывать на одно из очень интересных мест в игре, и, может быть, однажды вы захотите вернуться к нему. Или, что еще хуже, вы сейчас находитесь в безвыигрышном состоянии и вынуждены начинать заново.

2.2. Контроль версий

Во время редактирования вы можете "Сохранить как ..." в другой файл, или скопировать файл куда-нибудь перед сохранением, чтобы уберечь более старые версии. Может быть и заархивировав их для сохранения места на диске. Это самый примитивный вид контроля версий, к тому же требующий интенсивной ручной работы. Компьютерные игры прошли этот этап давным-давно, в большинстве из них есть множество слотов для сохранения с автоматическими временными метками.

Давайте усложним условия. Пусть у вас есть несколько файлов, используемых вместе, например, исходный код проекта или файлы для вебсайта. Теперь, чтобы сохранить старую версию, вы должны скопировать весь каталог. Поддерживать множество таких версий вручную неудобно, и быстро становится дорогим удовольствием.

В некоторых играх сохранение — это и есть каталог с кучей файлов внутри. Эти игры скрывают детали от игрока и предоставляют удобный интерфейс для управления различными версиями этого каталога.

В системах контроля версий всё точно так же. У них у всех есть приятный интерфейс для управления каталогом, полным-полно всяких данных. Вы можете сохранять состояние каталога так часто, как пожелаете, и вы можете восстановить любую из предыдущих сохраненных версий. Но, в отличие от компьютерных игр, они умнее используют дисковое пространство. Обычно только несколько файлов меняется от версии к версии, не более. Сохранение различий, а не всей копии каталога, не так сильно расходует свободное место.

2.3. Распределенный контроль

А теперь представьте очень сложную компьютерную игру. Её настолько сложно пройти, что множество опытных игроков по всему миру решили объединиться и использовать общие сохранения, чтобы попытаться выиграть. Прохождения на скорость — живой пример. Игроки, специализирующиеся на разных уровнях игры, объединяются, чтобы в итоге получить потрясающий результат.

Как бы вы организовали такую систему, чтобы игроки смогли легко забирать сохранения других? А загружать свои?

Раньше каждый проект использовал централизованную систему контроля версий. Какой-нибудь сервер хранил все сохраненные игры. И никто больше. На машине каждого игрока хранилась только очень маленькая часть. Когда игрок хотел пройти немного дальше, он выкачивал самое последнее сохранение с сервера, играл немного, сохранялся и закачивал уже свое сохранение обратно на сервер, чтобы кто-нибудь другой смог его использовать.

А что, если игрок по какой-то причине захотел использовать более старую сохраненную игру? Возможно, текущая версия сохраненной игры безвыигрышна, потому что кто-то из игроков забыл взять какой-либо игровой предмет на каком-то предыдущем уровне, и они хотят найти последнюю сохраненную игру, которая все еще выигрышна. Или, может быть, они хотят сравнить две более старые сохраненные игры, чтобы установить вклад каждого игрока.

Может быть много причин вернуться к более старой версии, но выход один. Они должны запросить центральный сервер о той старой сохраненной игре. Чем больше сохраненных игр они захотят, тем больше им понадобится связываться с сервером.

Новое поколение систем контроля версий, к которым относится **Git**, известны как распределенные системы, их можно понимать как обобщение централизованных систем. Когда игроки загружаются с главного сервера, они получают каждую сохраненную игру, а не только последнюю. Это как если они зеркалируют центральный сервер.

Эти первоначальные операции клонирования могут быть интенсивными, особенно если присутствует длинная история разработки, но это сполна окупается при длительной работе.

Одна немедленная выгода состоит в том, что если по какой-то причине потребуется более старая версия, дополнительное обращение к серверу не понадобится.

2.3.1. Глупые предрассудки

Широко распространенное заблуждение состоит в том, что распределенные системы непригодны для проектов, требующих официального централизованного репозитория. Ничто не может быть более далеким от истины. Получение фотоснимка не приводит к тому, что мы крадем чью-то душу. Проще говоря, клонирование главного репозитория не уменьшает его важность.

В первом приближении можно сказать, что все, что делает централизованная система контроля версий, хорошо сконструированная распределенная система может сделать лучше. Сетевые ресурсы просто дороже локальных. Хотя дальше мы увидим, что в распределенном подходе есть свои недостатки, менее вероятно провести ложные аналогии руководствуясь этим приближенным правилом.

Небольшому проекту может понадобиться лишь часть функций, предлагаемых такой системой. Но будете ли вы использовать римские цифры в расчетах с небольшими числами? Более того, ваш проект может вырасти за пределы ваших первоначальных ожиданий. Использовать Git с самого начала — это как держать наготове швейцарский армейский нож, даже если вы только открываете им бутылки. Однажды вам безумно понадобится отвертка и вы будете рады, что под рукой у вас нечто большее, чем простая открывалка.

2.4. Конфликты при слиянии

Для этой темы аналогия с компьютерной игрой становится слишком натянутой. Вместо этого, давайте вернемся к редактированию документа.

Итак, допустим, что Алиса вставила строчку в начале файла, а Боб — в конец. Оба они закачивают свои изменения. Большинство систем автоматически сделает разумный вывод: принять и объединить их изменения так, чтобы обе правки — и Алисы, и Боба — были применены.

Теперь предположим, что и Алиса, и Боб независимо друг от друга сделали изменения в одной и той же строке. Тогда становится невозможным разрешить конфликт без человеческого вмешательства. Тот, кто вторым из них закачает на сервер изменения, будет информирован о конфликте слияния, и должен либо выбрать, чье изменение перекроет другое, либо заново отредактировать строку целиком.

Могут случаться и более сложные ситуации. Системы контроля версий разрешают простые ситуации сами, и оставляют сложные для человека. Обычно такое их поведение поддается настройке.

Глава 3. Базовые операции

Прежде чем погружаться в дебри многочисленных команд **Git**, попробуйте воспользоваться приведёнными ниже простыми примерами, чтобы немного освоиться. Несмотря на свою простоту, каждый из них является полезным.

В самом деле, первые месяцы использования **Git** я не выходил за рамки материала, изложенного в этой главе.

3.1. Сохранение состояния

Выполняете опасную операцию? Прежде чем сделать это, создайте снимок всех файлов в текущей директории с помощью команд:

```
$ git init
$ git add .
$ git commit -m "Мой первый бекап"
```

Теперь, если ваши новые правки всё испортили, запустите:

```
$ git reset --hard
```

чтобы вернуться к исходному состоянию. Чтобы вновь сохраниться, выполните:

```
$ git commit -a -m "Другой бекап"
```

3.1.1. Добавление, удаление, переименование

Приведенный выше пример будет отслеживать файлы, которые вы добавили, когда впервые запустили **git add**. Если вы хотите добавить новые файлы или поддиректории, вам придётся сказать **Git**:

```
$ git add НОВЫЕ_ФАЙЛЫ...
```

Аналогично, если вы хотите, чтобы **Git** забыл о некоторых файлах, например, потому что вы удалили их:

```
$ git rm СТАРЫЕ_ФАЙЛЫ...
```

Переименование файла — это то же, что и удаление старого имени и добавления нового. Для этого есть **git mv**, который имеет тот же синтаксис, что и команда **mv**. Например:

```
$ git mv OLDFILE NEWFILE
```

3.2. Расширенная отмена/Восстановление

Иногда вы просто хотите вернуться к определенной точке и забыть все изменения, потому что все они были неправильными. В таком случае:

```
$ git log
```

покажет вам список последних изменений (коммитов, прим. пер.) и их SHA1 хеши. Далее введите:

```
$ git reset --hard SHA1_HASH
```

для восстановления состояния до указанного коммита и удаления всех последующих коммитов безвозвратно.

Возможно, в другой раз вы захотите быстро вернуться к старому состоянию. В этом случае наберите:

```
$ git checkout SHA1_HASH
```

Это перенесет вас назад во времени, до тех пор пока вы не сделаете новые коммиты. Как и в фантастических фильмах о путешествиях во времени, если вы редактируете и коммитите код, вы будете находиться в альтернативной реальности, потому что ваши действия отличаются от тех, что вы делали до этого.

Эта альтернативная реальность называется «ветвь» (**branch**, прим. пер.), и мы поговорим об этом больше чуть позже. А сейчас просто запомните:

```
$ git checkout master
```

вернёт вас обратно в настоящее. Кроме того, чтобы не получать предупреждений от **Git**, всегда делайте коммит или сброс ваших изменений до запуска **checkout**.

Ещё раз воспользуемся терминологией компьютерных игр:

- **git reset --hard**: загружает ранее сохраненную игру и удаляет все версии, сохраненные после только что загруженной.
- **git checkout**: загружает старую игру, но если вы продолжаете играть, состояние игры будет отличаться от более новых сохранений, которые вы сделали в первый раз. Любая

игра, которую вы теперь сохраняете, попадает в отдельную ветвь, представляющую альтернативную реальность, в которую вы вошли. Как мы договорились.

Вы можете выбрать для восстановления только определенные файлы и поддиректории путём перечисления их имён после команды:

```
$ git checkout SHA1_HASH some.file another.file
```

Будьте внимательны: такая форма **checkout** может незаметно перезаписать файлы. Чтобы избежать неприятных неожиданностей, выполняйте коммит до запуска **checkout**, особенно если вы только осваиваетесь с **Git**. Вообще, если вы не уверены в какой-либо операции, будь то команда **Git** или нет, сперва выполните **git commit -a**.

Не любите копировать и вставлять хеши? Используйте:

```
$ git checkout :/"Мой первый б"
```

для перехода на коммит, который начинается с приведенной строки.

Можно также запросить 5-ое с конца сохраненное состояние:

```
$ git checkout master~5
```

3.2.1. Возвраты

В зале суда в протокол могут вноситься изменения прямо во время слушания. Подобным образом и вы можете выбирать коммиты для возврата.

```
$ git commit -a  
$ git revert SHA1_HASH
```

отменит коммит с выбранным хешем. Запущенный **git log** показывает, что изменение записано в качестве нового коммита.

3.3. Создание списка изменений

Некоторым проектам требуется список изменений (<http://en.wikipedia.org/wiki/Changelog>) (changelog, прим. пер.). Создать такой список вы можете, просто направив вывод **git log** в файл:

```
$ git log > ChangeLog
```

3.4. Скачивание файлов

Получить копию проекта под управлением Git можно, набрав:

```
$ git clone git://server/path/to/files
```

Например, чтобы получить все файлы, я создавал такой сайт:

```
$ git clone git://git.or.cz/gitmagic.git
```

Мы поговорим больше о команде **clone** позже.

3.5. На острие ножа

Если вы уже загрузили копию проекта с помощью **git clone**, вы можете обновить его до последней версии, используя:

```
$ git pull
```

3.6. Публичный доступ

Предположим, вы написали скрипт, которым хотите поделиться с другими. Можно просто позволить всем загружать его с вашего компьютера, но, если они будут делать это в то время, как вы дорабатываете его или добавляете экспериментальную функциональность, у них могут возникнуть проблемы. Конечно, это одна из причин существования цикла разработки. Разработчики могут долго работать над проектом, но открывать доступ к коду следует только после того, как код приведен в приличный вид.

Чтобы сделать это с помощью Git, выполните в директории, где лежит ваш скрипт:

```
$ git init
$ git add .
$ git commit -m "Первый релиз"
```

Затем скажите вашим пользователям запустить:

```
$ git clone ваш.компьютер:/path/to/script
```

для того чтобы загрузить ваш скрипт. Это подразумевает что у вас есть доступ по ssh. Если нет, запустите **git daemon** и скажите остальным использовать для запуска:

```
$ git clone git://ваш.компьютер/path/to/script
```

Теперь, всякий раз когда ваш скрипт готов к релизу, выполняйте:

```
$ git commit -a -m "Следующий релиз"
```

и ваши пользователи смогут обновить свои версии, перейдя в директорию, содержащую ваш скрипт, и, набрав:

```
$ git pull
```

ваши пользователи никогда не попадут на версии скрипта, доступ к которым вы скрываете. Безусловно, этот трюк работает для всего, а не только в случаях со скриптами.

3.7. Что я наделал?

Выясните, какие изменения вы сделали со времени последнего коммита:

```
$ git diff
```

Или со вчерашнего дня:

```
$ git diff "@{yesterday}"
```

Или между определенной версией и версией, сделанной 2 коммита назад:

```
$ git diff SHA1_HASH "master~2"
```

В каждом случае на выходе будет патч, который может быть применён с помощью **git apply**.

Попробуйте также:

```
$ git whatchanged --since="2 weeks ago"
```

Часто я смотрю историю при помощи **qgit** (<http://sourceforge.net/projects/qgit>) вместо стандартного способа, из-за приятного интерфейса, или с помощью **tig** (<http://jonas.nitro.dk/tig>) с текстовым интерфейсом, который хорошо работает при медленном соединении. В качестве альтернативы, можно запустить веб-сервер, введя **git instaweb**, и запустить любой веб-браузер.

3.8. Упражнение

Пусть A, B, C, D четыре успешных коммита, где B — то же, что и A, но с несколькими удаленными файлами. Мы хотим вернуть файлы в D, но не в B. Как это можно сделать?

Существует как минимум три решения. Предположим, что мы находимся на D.

1. Разница между A и B — удаление файлов. Мы можем создать патч, отражающий эти изменения, и применить его:

```
$ git diff B A | git apply
```

2. Поскольку в коммите A мы сохранили файлы, мы можем получить их обратно:

```
$ git checkout A FILES...
```

3. Мы можем рассматривать переход от A до B в качестве изменений, которые мы хотим отменить:

```
$ git revert B
```

Какой способ лучший? Тот, который вам больше нравится. С Git легко сделать все, что вы хотите, причём часто существует много разных способов сделать одно и то-же.

Глава 4. Все о клонировании

В старых системах контроля версий `checkout` - это стандартная операция для получения файлов. Вы получаете файлы в нужном сохраненном состоянии.

В `Git` и других распределенных системах контроля версий, клонирование - это обычно дело. Для получения файлов вы создаете клон всего репозитория. Другими словами, вы создаете зеркало центрального сервера. При этом все что можно делать в основном репозитории, можно делать и в локальном.

4.1. Синхронизация компьютеров

По этой причине я начал использовать `Git`. Я вполне приемлю синхронизацию архивами или использование `rsync` для бэкапа или процедуры стандартной синхронизации. Но я работаю то на ноутбуке, то на стационарном компьютере, которые никак между собой не взаимодействуют.

Инициализируйте `Git` репозиторий и делайте коммит файлов на одном компьютере. А потом выполните следующие операции на другом:

```
$ git clone other.computer:/path/to/files
```

для создания второй копии файлов и `Git` репозитория. С этого момента выполняйте,

```
$ git commit -a  
$ git pull other.computer:/path/to/files HEAD
```

это синхронизирует состояние ваших файлов с состоянием файлов другого компьютера. Если вы внесли изменения, которые будут конфликтовать с таким же файлом, `Git` даст Вам знать об этом, и вам придется сделать коммит еще раз, уже после устранения конфликтующих изменений.

4.2. Классический контроль исходного кода

Инициализируйте `Git`-репозиторий для ваших файлов:

```
$ git init  
$ git add .  
$ git commit -m "Initial commit"
```

На центральном сервере инициализируйте пустой Git-репозиторий с присвоением какого-нибудь имени,

и запустите Git-демон, если необходимо:

```
$ GIT_DIR=proj.git git init
$ git daemon --detach # возможно уже запущен
```

Публичные репозитории, такие как `repo.or.cz` (<http://repo.or.cz>), имеют собственные схемы по организации изначально пустых Git-репозитариев, которые обычно предполагают регистрацию и заполнение формы.

Чтобы сохранить ваши изменения в центральный репозиторий, запустите:

```
$ git push git://central.server/path/to/proj.git HEAD
```

Для клонирования, как уже было описано выше, необходимо:

```
$ git clone git://central.server/path/to/proj.git
```

После внесения изменений, код записывается на главный сервер с помощью:

```
$ git commit -a
$ git push
```

Если в ходе работы на сервере уже происходили изменения, необходимо обновить локальную копию репозитория перед сохранением собственных изменений. Для синхронизации:

```
$ git commit -a
$ git pull
```

4.3. Создание форка проекта

Не нравится путь развития проекта? Думаете можете сделать лучше? Тогда на Вашем сервере:

```
$ git clone git://main.server/path/to/files
```

Теперь расскажите всем, что новый проект находится на вашем сервере.

В любое время вы можете объединить изменения с изначальным проектом, используя:

```
$ git pull
```

4.4. Окончательные бэкапы

Хотите создать множество географически разнесенных, разных, защищенных, резервных архивов? Если в вашем проекте много разработчиков - делать ничего не надо! Каждый клон - это и есть бэкап, причем отражающий не только текущее состояние, но и всю историю изменений проекта. Благодаря криптографическому хэшированию, при нарушении какого-либо из клонов, этот клон будет помечен, и это будет видно при любой попытке взаимодействия с ним.

Если ваш проект не такой популярный, размещайте клоны на как можно большем количестве серверов.

Особо беспокоящимся рекомендуется всегда записывать самый последний 20-байтный SHA1 хэш HEAD в безопасном месте. В безопасном, но не тайном. Например, в газете, это будет эффективным, потому как сложно изменить каждую копию газеты.

4.5. Многозадачность со скоростью света

Скажем, вы хотите выполнять несколько задач параллельно. Тогда сохраните свой проект и запустите:

```
$ git clone . /some/new/directory
```

Git использует жесткие ссылки и обмен файлами настолько безопасно, насколько это возможно для создания такого клона, он будет готов мгновенно, после чего можно будет работать с разными функциями одновременно. Например, можно редактировать один клон, компилируя в это время другой.

В любое время можно сделать коммит и вытянуть изменения из другого клона.

```
$ git pull /the/other/clone HEAD
```

4.6. Другие системы контроля версий

Вы работаете над проектом, который использует другую систему контроля версий, и вам не хватает Git? Тогда инициализируйте Git-репозиторий в свою рабочую папку:

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

затем клонируйте его:

```
$ git clone . /some/new/directory
```

Теперь перейдите в новую директорию и работайте в ней, используя для контроля версий **Git**. Когда вам понадобится синхронизировать изменения с другими, перейдите в изначальную директорию и произведите синхронизацию с помощью другой системы контроля версий, затем наберите:

```
$ git add .
$ git commit -m "Синхронизироваться с другими"
```

Теперь перейдите в новую директорию и запустите:

```
$ git commit -a -m "Описание моих изменений"
$ git pull
```

Процедура обмена изменениями с другими зависит от используемой системы контроля версий. Новая директория содержит файлы с вашими изменениями. Для загрузки файлов в центральный репозиторий требуется запуск любых необходимых команд другой системы контроля версий.

Команда **git svn** автоматизирует этот процесс для репозитория Subversion и может быть использована для экспорта **Git** проекта в Subversion репозиторий (<http://google-opensource.blogspot.com/2008/05/export-git-project-to-google-code.html>).

Глава 5. Чудеса ветвления

Возможности мгновенного разветвления и слияния - самые уникальные особенности **Git**.

Задача: какие-то причины требуют переключения процессов. В новой версии внезапно возникает серьезная ошибка. Срок завершения работы над определенным свойством близится к концу. Разработчик, помощь которого очень нужна Вам в работе над ключевым разделом, собирается в отпуск. Итак, Вам нужно срочно бросить все, над чем Вы трудитесь в настоящий момент, и переключиться на совершенно другие дела.

Переключение внимания с одного на другое может серьезно снизить эффективность работы, и чем сложнее переход между процессами, тем больше будет потеря. При централизованном управлении версиями необходимо скачивать вновь разработанную рабочую копию с центрального сервера. Распределенная система предоставляет лучшие возможности, так как позволяет клонировать нужную версию в локальное рабочее место.

Однако клонирование все же предполагает копирование всей рабочей директории, а, значит, всей истории изменений до настоящего момента. Даже при том, что **Git** позволяет сэкономить средства за счет возможности совместного использования файлов и жестких ссылок, все файлы проекта придется полностью воссоздать в новой рабочей директории.

Решение: у **Git** есть более удобный инструмент для этих целей, который, в отличие от клонирования, экономит и время, и дисковое пространство - это **git branch**.

С этой волшебной командой файлы в вашей директории мгновенно изменяются с одной версии на другую. Это изменение позволяет сделать намного больше, чем просто вернуться назад или продвинуться вперед в истории. Ваши файлы могут измениться с последней версии на экспериментальную, с экспериментальной - на опытную, с опытной - на версию вашего друга и так далее.

5.1. Кнопка босса

Наверняка, вы играли в одну из тех игр, где при нажатии определенной клавиши ("кнопка босса"), игра быстро сворачивается и на экране отображается рабочая таблица или что-нибудь другое? То есть, если в офис зашел начальник, а вы играете в игру, вы должны уметь быстро ее скрыть.

В какой-нибудь директории:

```
$ echo "Я хитрее моего босса" > myfile.txt
$ git init
$ git add .
```

```
$ git commit -m "Начальный коммит"
```

Мы создали Git-репозиторий который содержит один текстовый файл с определенным сообщением. Теперь выполните:

```
$ git checkout -b boss # вероятно, это последнее изменение
$ echo "Мой босс меня умнее" > myfile.txt
$ git commit -a -m "Другой коммит"
```

Похоже на то, как будто мы перезаписали файл и сделали коммит. Но это иллюзия. Наберите:

```
$ git checkout master # переключиться на оригинальную версию файла
```

Вуаля! Текстовый файл восстановлен. И если босс будет рядом, запустите

```
$ git checkout boss # перейти на специальную версию для босса
```

Вы можете переключаться между двумя версиями этого файла так часто, как вам хочется и делать коммиты в каждый из них независимо.

5.2. Грязная работа

Допустим, вы работаете над созданием какой-либо функции, и по каким-то причинам необходимо вернуться назад к старой версии и временно загрузить старый код, чтобы посмотреть как что-либо работало, тогда введите:

```
$ git commit -a
$ git checkout SHA1_HASH
```

Теперь вы можете добавить везде, где необходимо, временный черновой код. Можно даже сделать коммит изменений. Когда закончите, выполните:

```
$ git checkout master
```

чтобы вернуться к исходной работе. Заметьте, что любые изменения, не внесенные в коммит, будут перенесены.

А что, если вы все-таки хотели сохранить временные изменения? Пожалуйста:

```
$ git checkout -b dirty
```

а затем сделайте коммит перед тем, как переключетесь на ветвь `master`. Всякий раз когда вы захотите вернуться к черновым изменениям, просто выполните:

```
$ git checkout dirty
```

Мы говорили об этой команде ранее, в другой главе, когда обсуждали загрузку старых состояний. Теперь у нас перед глазами полная картина: файлы меняются на нужное состояние, но мы должны оставить ветвь **master**. Любые коммиты, сделанные с этого момента, направят файлы по другому пути, который может быть назван позже.

Другими словами, после переключения с более старого состояния, **Git** автоматически направляет вас в новую еще не названную ветвь, которой можно дать имя и сохранить с помощью **git checkout -b**.

5.3. Быстрые исправления

Ваша работа в самом разгаре, когда вдруг выясняется, что нужно все бросить и исправить только что обнаруженную ошибку:

```
$ git commit -a
$ git checkout -b fixes SHA1_HASH
```

Затем, после того, как вы исправили ошибку:

```
$ git commit -a -m "Ошибка исправлена"
$ git push # в центральный репозиторий
$ git checkout master
```

и вернитесь к работе над вашими исходными задачами.

5.4. Бесперебойный рабочий процесс

В некоторых проектах необходимо проверять код до того, как выложить его. Чтобы облегчить задачу другим разработчикам, которые будут проверять ваш код, при внесении значительных изменений, разбивайте измененный проект на 2 или более частей и выкладывайте по одной для проверки.

А что, если вторую часть нельзя записать до того, как первая часть проверена и принята? Во многих системах управления версиями отправить на рассмотрение вторую часть можно только после утверждения первой части.

На самом деле это не совсем правда, но необходимость в таких системах редактировать часть II перед тем, как отправить часть I, связана с трудностями и неудобствами. В **Git**, ветвление и слияние гораздо безболезненней (говоря техническим языком - это можно

сделать быстрее и на локальном уровне). Поэтому, после того, как вы сделали коммит первой части, и направили его для рассмотрения:

```
$ git checkout -b part2
```

Далее, можно изменять вторую часть, не ожидая принятия первой части. После того, как первая часть утверждена и выложена,

```
$ git checkout master
$ git merge part2
$ git branch -d part2 # эта ветка больше не нужна
```

и вторая часть правок готова к проверке.

Однако не торопитесь! Что, если не все так просто? Что, если в первой части вы сделали ошибку, которую необходимо было исправить до отправки. Запросто! Во-первых, вернитесь в master-ветвь с помощью:

```
$ git checkout master
```

Исправьте изменения в первой части и отправьте на проверку. Если же они не будут приняты, можно просто повторить этот шаг. Вы, вероятно, также захотите произвести слияние исправлений части I с частью II, тогда выполните:

```
$ git checkout part2
$ git merge master
```

И теперь - тоже самое. После того, как первая часть утверждена и выложена:

```
$ git checkout master
$ git merge part2
$ git branch -d part2
```

и снова, вторая часть готова к проверке.

Вы можете легко использовать этот трюк для любого количества частей.

5.5. Собрать все в кучу

Предположим, вам нравится работать над всеми аспектами проекта в одной и той же ветке. Вы хотите закрыть свой рабочий процесс от других, чтобы все видели ваши коммиты только после того, как они будут хорошо оформлены. Создайте несколько веток:

```
$ git checkout -b sanitized
```



```
$ git checkout -b medley
```

Далее, работайте над чем угодно: исправляйте ошибки, добавляйте новые функции, добавляйте временный код и т.д., при этом постоянно выполняя коммиты. Затем:

```
$ git checkout sanitized  
$ git cherry-pick SHA1_HASH
```

применит данный коммит для ветви "sanitized". С правильно выбранными элементами вы сможете собрать ветвь, которая будет содержать только проверенный код и соответствующие коммиты, сгруппированные вместе.

5.6. Управление Ветками

Для просмотра списка всех веток наберите:

```
$ git branch
```

Здесь всегда будет ветка с названием "master", с нее вы начинаете работать по умолчанию. Кому-то нравится оставлять ветку "master" нетронутой и создавать новые ветки со своими изменениями.

Опции **-d** и **-m** позволяют удалять и перемещать (переименовывать) ветки.

См. **git help branch**.

Ветка "master" - это полезная традиция. Все считают очевидным то, что ваш репозиторий должен содержать ветку с таким именем, и эта ветка содержит официальную версию проекта. Вы можете переименовать или удалить ветку "master", однако лучше соблюсти всеобщую традицию.

5.7. Временные Ветки

Через какое-то время вы можете обнаружить, что создаете множество временных веток для одной и той краткосрочной цели: нужно сохранить текущее состояние, чтобы была возможность вернуться назад и исправить грубую ошибку или сделать что-то еще.

Это похоже на то, как вы переключаете телевизионные каналы, чтобы посмотреть что показывают по другим. Но здесь, вместо того, чтобы нажать на пару кнопок на пульте, нужно будет создать, сделать отладку, а затем удалить временные ветки и коммиты. К

счастью, в **Git** есть удобные ярлыки, имитирующие работу дистанционного пульта управления.

```
$ git stash
```

Это сохраняет текущее состояние в во временном месте (*копилке*) и восстанавливает предыдущее состояние. Ваша директория становится точно такой, как и была до того, как вы начали редактирование, и вы можете исправить ошибки, загрузить удаленные изменения и прочее. Когда вы хотите вернуться назад в состояние копилки, наберите:

```
$ git stash apply # Возможно, понадобится устранить какие-либо конфликты.
```

Можно создавать несколько копилек, приумножать их и использовать по-разному. Смотрите **git help stash**. Как вы могли догадаться, этот чудесный прием возможен благодаря способности **Git** поддерживать создание закрытых "тайных" веток.

5.8. Работайте как вам нравится

Такие приложения как Mozilla Firefox (<http://www.mozilla.com/>) позволяют открывать несколько вкладок, и несколько окон. Переключение вкладок позволяет обозревать разное содержание в одном и том же окне. Ветки в **Git** подобны вкладкам для вашей рабочей директории. Если продолжить аналогию, клонирование в **Git** подобно открытию нового окна. Эти две возможности делают работу в **Git** удобной и приятной для пользователя.

На более высоком уровне, многие оконные менеджеры **Linux** поддерживают несколько рабочих столов.

Ветки в **Git** подобны переключению на другой рабочий стол, а клонирование подобно подключению дополнительно монитора для получения дополнительного рабочего стола.

Другой пример - это утилита **screen** (<http://www.gnu.org/software/screen/>). Эта программа позволяет создавать, закрывать и переключаться между множеством терминальных сессий в одном и том же терминале. Вместо открытия нового терминала (операция клонирования), можно использовать этот, запустив **screen** (создать ветвь). На самом деле у **screen** еще много возможностей, но это уже тема для другой главы.

Наличие возможностей клонирования, ветвления и быстрого локального слияния позволяет вам выбрать комбинировать их так, как удобно и нужно вам. **Git** позволяет работать именно так, как вам хочется.

Глава 6. Уроки истории

Вследствие распределенной природы Git, историю изменений можно легко редактировать. Однако, если вы вмешиваетесь в прошлое, будьте осторожны: изменяйте только ту часть истории, которой владеете вы и только вы. И также как государства бесконечно выясняют, кто же именно совершил и какие бесчинства, так и у вас будут проблемы с примирением после взаимодействия деревьев истории.

Конечно, если вы также контролируете и все остальные деревья, то нет никаких проблем поскольку вы можете переписать их.

Некоторые разработчики убеждены, что история должна быть неизменна со всеми огрехами и прочим. Другие считают, что деревья должны быть презентабельными, до того как они выпустят их в публичный доступ. Git учитывает оба мнения. Также как клонирование, ветвление и слияние, переписывание истории - это просто еще одна возможность, которую дает вам Git. Разумное ее использование зависит только от вас.

6.1. Оставаясь корректным

Только что сделали коммит и уже хотите изменить запись в журнале? Запустите:

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

чтобы изменить последнее сообщение коммита. Осознали, что забыли добавить файл? Запустите **git add**, чтобы это сделать и выполните вышеуказанную команду.

Захотелось добавить еще немного изменений в последний коммит? Так сделайте их и запустите:

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

6.2. ... И кое-что еще

Давайте представим себе, что предыдущая проблема на самом деле в десять раз хуже. После длительной работы вы сделали ряд фиксаций, но вы не очень-то довольны тем, как они организованы и кое-какие записи в журнале (commit messages) надо бы слегка переформулировать. В этом случае запустите:

```
$ git rebase -i HEAD~10
```

и записи в журнале от последних 10-ти фиксаций появятся в вашем любимом редакторе (задается переменной окружения `$EDITOR`). Вот кусок примера:

```
pick 5c6eb73 Добавил ссылку repo.or.cz
pick a311a64 Сменил порядок в "Работай как хочешь"
pick 100834f Добавил цель для push в Makefile
```

После чего:

- Убираем коммиты, удаляя строки.
- Меняем порядок коммитов, меняя порядок строк.
- Заменяем "pick" на "edit", если требуется внести изменения в коммиты.
- Заменяем "pick" на "squash" для слияния коммита с предыдущим.

Если вы отметили коммит для исправлений, запустите:

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

Если нет, запустите:

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

В общем, делайте коммиты как можно раньше и как можно чаще - всегда потом сможете за собой подчистить при помощи `rebase`.

6.3. Локальные изменения сохраняются

Предположим, вы работаете над активным проектом. За какое-то время были проведены несколько коммитов, после чего синхронизируетесь с официальным деревом через слияние. Цикл повторяется несколько раз до тех пор, пока вы не готовы отправить (`push`) изменения в центральное дерево.

Однако теперь история изменений в локальном клоне `Git` представляет собой кашу из ваших и официальных изменений. Вам бы хотелось видеть все изменения непрерывной секцией даже после слияния с официальными.

Это работа для команды **`git rebase`** в виде, описанном выше. Зачастую, имеет смысл использовать флаг **`--onto`**, чтобы не использовать интерактивный режим.

Также стоит взглянуть на вывод команды **`git help rebase`** для получения подробных примеров использования этой замечательной команды. Вы можете объединять фиксации, вы можете даже перестраивать ветки.

6.4. Переписывая историю

Время от времени вам может понадобиться эквивалент "замазывания" людей на официальных фотографиях, только для систем контроля версий, как бы стирая их из истории в духе сталинизма. Например, предположим, что мы уже собираемся выпустить релиз проекта, но он содержит файл, который не должен стать достоянием общественности по каким-то причинам. Может я сохранил номер своей кредитки в текстовый файл и случайно добавил этот файл к файлам проекта? Просто стереть файл бесполезно - из-за контроля версий всегда можно вытащить такую из старых версий, где этот документ еще есть. Нам надо удалить файл из всех версий когда-либо существовавших. Для этого:

```
$ git filter-branch --tree-filter 'rm top/secret/file' HEAD
```

Стоит посмотреть вывод команды **git help filter-branch**, где обсуждается этот пример и даже предлагается более быстрый метод решения. Короче говоря, **filter-branch** позволяет изменять существенные части истории при помощи одной-единственной команды.

В результате директория `.git/refs/original` будет описывать состояние дел до выполнения операции. Убедитесь, что команда **filter-branch** проделала все, что вы хотели, и, если хотите опять использовать команду, удалите эту директорию.

И, наконец, замените клоны вашего проекта обновленной версией, если собираетесь в дальнейшем с ними иметь дело.

6.5. Создавая Историю

Хотите перевести проект под управление Git? Если сейчас он находится под управлением какой-либо из хорошо известных систем контроля версий, то весьма велики шансы на то, что кто-нибудь уже позаботился и написал необходимые скрипты для экспорта всей истории проекта в Git.

А если все-таки нет? Тогда смотрите в сторону команды **git fast-import**, которая считывает текстовый ввод в специальном формате для создания истории Git "с нуля".

Обычно скрипт, использующий эту команду - это наспех состряпанное нечто, предназначенное для однократного использования, чтоб "перетащить" проект за один раз.

В качестве примера, вставьте следующий листинг в какой-нибудь файл, типа `/tmp/history`:

```
commit refs/heads/master
committer Alice <alice@example.com> Thu, 01 Jan 1970 00:00:00 +0000
data <<EOT
Стартовый коммит.
```

```

EOT

M 100644 inline hello.c
data <<EOT
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
EOT

commit refs/heads/master
committer Bob <bob@example.com> Tue, 14 Mar 2000 01:59:26 -0800
data <<EOT
Заменен printf() на write()
EOT

M 100644 inline hello.c
data <<EOT
#include <unistd.h>

int main() {
    write(1, "Hello, world!\n", 14);
    return 0;
}
EOT

```

Затем создайте репозиторий **Git** из этого временного файла при помощи команд:

```
$ mkdir project; cd project; git init $ git fast-import < /tmp/history
```

Вы можете извлечь (**checkout**) последнюю версию проекта при помощи команды:

```
$ git checkout master .
```

Команда **git fast-export** преобразует любой репозиторий **Git** в формат, понимаемый командой **git fast-import**. То есть, ее выход может быть использован как образец для тех кто пишет скрипты-преобразователи и для того, чтобы переносить репозитории в формате, хоть как-то пригодном для чтения человеком. Естественно, при помощи этих команд можно пересылать репозитории текстовых файлов через каналы передачи текста.

6.6. Когда же все пошло не так?

Вы только что обнаружили, что кое-какой функционал вашей программы не работает, но вы совершенно отчетливо помните, что он работал всего несколько месяцев назад. Ну вот, блин! Откуда же взялась ошибка? Вы же это проверяли сразу как разработали.

В любом случае, уже поздно сожалеть. Однако, если вам хватило ума фиксировать свои изменения часто, то **Git** сможет сильно помочь точно определить проблему.

```
$ git bisect start
$ git bisect bad SHA1_OF_BAD_VERSION
$ git bisect good SHA1_OF_GOOD_VERSION
```

Git извлечет состояние ровно посередине. Проверьте работает ли то, что сломалось, и если все еще нет, то введите:

```
$ git bisect bad
```

Если же работает, то замените "bad" на "good" в предыдущей команде. **Git** снова переместит вас в состояние посередине между "хорошей" и "плохой" ревизиями, при этом сузив круг подозреваемых ревизий вдвое.

После нескольких итераций, этот бинарный поиск приведет вас к тому коммиту, на котором все и сломалось. После окончания выяснения, вернуть исходное состояние можно командой:

```
$ git bisect reset
```

Вместо того, чтоб вручную тестировать каждое изменение - автоматизируйте этот процесс при помощи команды:

```
$ git bisect run COMMAND
```

Git использует возвращаемое значение заданной команды, обычно "одноразового" скрипта, чтоб определить "хорошее" это было изменение или "плохое". Скрипт должен вернуть 0, если "хорошее", 125 если изменение надо пропустить и любое число от 1 до 127 если "плохое". Отрицательное возвращаемое значение прерывает бинарный поиск.

На самом деле возможностей еще больше! На странице помощи объясняется как визуализировать бинарный поиск, проанализировать или даже воспроизвести журнал поиска, исключить изменения в которых точно все в порядке для ускорения поиска.

6.7. Из-за кого все пошло наперекосяк?

Как и многие другие системы контроля версий, Git поддерживает команду **blame**:

```
$ git blame FILE
```

которая показывает кто и когда изменил каждую строку выбранного файла. В отличие же от многих других систем контроля версий эта операция происходит оффлайн, то есть данные берутся с локального диска.

6.8. Личный опыт

В централизованных системах контроля версий, изменения истории - достаточно сложная операция и для ее проведения необходимо привлечение администраторов. Процедуры клонирования, ветвления и слияния невозможно осуществить без сетевого соединения. Также обстоят дела и с такими базовыми операциями как просмотр журнала изменений или фиксация изменений. В некоторых системах сетевое соединение требуется даже для просмотра собственных изменений или открытия файла для редактирования.

Централизованные системы исключают возможность работать оффлайн и требуют более дорогой сетевой инфраструктуры, особенно с увеличением количества разработчиков. Еще более важно, из-за того все операции происходят медленнее, пользователи избегают пользоваться "продвинутыми" командами до тех пор пока не "припечет как следует". В особо "запущенных" случаях это касается и большинства базовых команд тоже. Продуктивность страдает из-за остановок в работе, когда пользователи вынуждены запускать "долгоиграющие" команды.

Я испытал этот феномен на себе. Git был моей первой системой контроля версий. Я быстро привык нему и стал относиться к его возможностям как к должному. Я предположил, что и другие системы похожи на него: выбор системы контроля версий не должен отличаться от выбора текстового редактора или браузера.

Когда, немного позже, я был вынужден использовать централизованную систему контроля версий, я был шокирован. При использовании Git, мое, зачастую капризное, интернет-соединение не имело большого значения, но теперь разработка стала невыносимой когда от него потребовалась надежность как у жесткого диска. Плюс к этому, я обнаружил, что стал избегать использовать некоторые команды из-за получающихся в результате их выполнения задержек, а без них оказалось невозможным следовать моему стилю разработки.

Когда мне было нужно запустить "медленную" команду, нарушение хода цепочки моих мыслей оказывало несоизмеримый ущерб разработке. Ожидая окончания связи с сервером, я должен был заниматься чем-то другим, чтоб скоротать время, например, чтением почты

или написанием документации. Через некоторое время я возвращался к первоначальной задаче, но приходилось тратить уйму времени, чтоб вспомнить на чем же я остановился и что хотел делать дальше. Все-таки люди не очень приспособлены для многозадачности.

Есть еще один интересный эффект, так называемая, трагедия общин: предвидя будущую загрузку сети, некоторые люди начинают использовать более "широкие" каналы чем им реально требуются для текущих операций в попытке предотвратить будущие задержки. Суммарная активность увеличивает загрузку сети, поощряя людей задействовать еще более высокоскоростные каналы в следующий раз, чтоб избежать еще больших задержек.

Глава 7. Групповая работа в Git

Сначала я использовал Git для личного проекта, в котором был единственным разработчиком. Среди команд, связанных с распределенными свойствами Git, мне требовались только **pull** и **clone**, чтобы хранить один и тот же проект в разных местах.

Позднее я захотел опубликовать свой код при помощи Git и включить изменения помощников. Мне пришлось научиться управлять проектами, которые разрабатываются множеством людей со всего мира. К счастью, это преимущество Git и, возможно, смысл ее существования.

7.1. Кто я?

Каждый коммит содержит имя автора и адрес электронной почты, которые отображаются по команде **git log**. По умолчанию Git использует системные настройки для заполнения этих полей. Чтобы установить их явно, введите:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Чтобы установить эти параметры только для текущего репозитория, опустите флаг **--global**.

7.2. Git через SSH, HTTP

Предположим, у вас есть SSH доступ к веб-серверу, но Git не установлен. Git может связываться через HTTP, хотя это и менее эффективно, чем его собственный протокол.

Скачайте, скомпилируйте, установите Git в вашем аккаунте; создайте репозиторий в директории, доступной через web:

```
$ GIT_DIR=proj.git git init
```

В директории "proj.git" запустите:

```
$ git --bare update-server-info
$ cp hooks/post-update.sample hooks/post-update
```

Для старых версий Git команда копирования выдаст ошибку, и вы должны будете запустить:

```
$ chmod a+x hooks/post-update
```

Теперь вы можете публиковать свои последние правки через **SSH** из любой копии:

```
$ git push web.server:/path/to/proj.git master
```

и кто угодно сможет взять ваш проект через **HTTP**:

```
$ git clone http://web.server/proj.git
```

7.3. Git через что угодно

Хотите синхронизировать репозитории без серверов или даже без сетевого подключения? Необходимо импровизировать в чрезвычайной ситуации? Мы рассмотрели, как **git fast-export** и **git fast-import** могут преобразовать репозитории в один файл и обратно. Мы можем обмениваться такими файлами между репозиториями **git** с помощью любых носителей, но более эффективным инструментом является **git bundle**.

Отправитель создает пакет (*bundle*):

```
$ git bundle create somefile HEAD
```

Затем перенесите **bundle**, **somefile**, такими средствами, как: **email**, флешка, дискета, **xxd** печать и последующее распознавание символов, чтение битов по телефону, дымовые сигналы и т.д. Получатель восстанавливает коммиты из пакета, вводя:

```
$ git pull somefile
```

Получатель может сделать это даже с пустым хранилищем. Несмотря на свой размер, **somefile** содержит весь исходный **Git** репозиторий.

В больших проектах для уменьшения объема пакет содержит только изменения, которых нет в других репозиториях:

```
$ git bundle create somefile HEAD ^COMMON_SHA1
```

Если это делается часто, то можно легко забыть, какой коммит был отправлен последним. Справка предлагает для решения этой проблемы использовать метки. А именно, после передачи пакета введите:

```
$ git tag -f lastbundle HEAD
```

и создавайте пакеты обновления с помощью:

```
$ git bundle create newbundle HEAD ^lastbundle
```

7.4. Патчи: Общее применения

Патчи представляют собой текст изменений, который может быть легко понят как человеком, так и компьютером. Это делает его очень привлекательным форматом обмена. Можно послать электронное письмо с патчем для разработчиков, независимо от того, какую систему контроля версий они используют. Пока ваши корреспонденты могут читать электронную почту, они могут увидеть ваши изменения. Аналогичным образом с Вашей стороны все, что Вам требуется, - это адрес электронной почты: нет необходимости в установке онлайн хранилища Git.

Напомним, в первой главе:

```
$ git diff COMMIT
```

выводит патч, который может быть вставлен в сообщение электронной почты для обсуждения. В Git репозитории, введите:

```
$ git apply < FILE
```

чтобы применить патч.

В более формальной обстановке, когда имя автора и подписи должны быть зарегистрированы, генерируйте соответствующие патчи с определенной точки, набрав:

```
$ git format-patch START_COMMIT
```

Полученные файлы могут быть отправлены с помощью **git-send-email** или вручную. Вы также можете указать диапазон коммитов:

```
$ git format-patch START_COMMIT..END_COMMIT
```

На принимающей стороне сохраните email в файл и введите:

```
$ git am < FILE
```

Это применит входящие исправления, а также создаст коммит, включающий такую информацию, как автор.

С web-клиентом электронной почты вам, возможно, потребуется нажать кнопку, чтобы посмотреть электронную почту в своем первоначальном виде до сохранения исправлений в файл.

Есть небольшие различия для Mbox-подобных клиентов электронной почты, но если вы используете один из них, то вы, вероятно, тот человек, который может легко настроить его

без чтения руководства!

7.5. К сожалению, мы переехали

Из предыдущих глав, мы видели, что после клонирования репозитория, набор **git push** или **git pull** автоматически проталкивает или стягивает с оригинального URL. Каким образом Git это делает? Секрет кроется в параметрах конфигурации инициализированных при создании клона. Давайте выполним команду:

```
$ git config --list
```

Опция `remote.origin.url` контролирует исходный URL; "origin" - имя исходного репозитория. Как и имя ветки "master" - это соглашение, мы можем изменить или удалить этот ник, но как правило, нет причин для этого.

Если адрес оригинального репозитория изменился, можно обновить его с помощью команды:

```
$ git config remote.origin.url NEW_URL
```

Опция `branch.master.merge` задает удаленную ветвь по умолчанию для **git pull**. В ходе первоначального клонирования, он установлен на текущую ветвь исходного репозитория, так что даже если HEAD исходного репозитория впоследствии переходит на другую ветку, **pull** будет продолжать выполняться для исходной ветви.

Этот параметр применим только к хранилищу, которое мы впервые клонировали, в его настройках записана `branch.master.remote`. Если мы выполняем **pull** из других хранилищ, то мы должны указать необходимому нам ветку:

```
$ git pull ANOTHER_URL master
```

Это объясняет, почему некоторых из наших более ранних примеров **push** и **pull** не имели аргументов.

7.6. Удаленные Ветки

Когда вы клонируете репозиторий, вы также клонируете все его ветки. Вы можете не заметить это, потому что Git скрывает их: вы должны указать это явно. Это предотвращает пересечение веток в удаленном хранилище с вашими, а также делает Git проще для начинающих.

Список удаленных веток можно посмотреть командой:

```
$ git branch -r
```

Вы должны увидеть что-то вроде:

```
origin/HEAD
origin/master
origin/experimental
```

Они представляют собой ветки и HEAD в удаленном хранилище, и могут быть использованы в обычных командах Git. Например, предположим, что вы сделали много коммитов, и хотели бы сравнить с последней загруженной версией. Вы можете найти через журналы для соответствующего SHA1 хэш, но это гораздо легче набрать:

```
$ git diff origin/HEAD
```

Также можно увидеть лог ветки "experimental" набрав:

```
$ git log origin/experimental
```

7.7. Несколько Удаленных Веток

Предположим, что два других разработчиков работает над нашим проектом, и мы хотим следить за обоими. Мы можем наблюдать более чем за одним хранилище одновременно введя:

```
$ git remote add other ANOTHER_URL
$ git pull other some_branch
```

Сейчас мы объединились в ветвь второго хранилища, и мы получили легкий доступ для всех ветвей во всех репозиториях:

```
$ git diff origin/experimental^ other/some_branch~5
```

Но что, если мы просто хотим сравнить их изменения, не затрагивающие нашу собственную работу? Иными словами, мы хотим, чтобы изучить их ветки без изменения нашей рабочей папки. В этом случае вместо pull наберите:

```
$ git fetch # Fetch from origin, the default.
$ git fetch other # Fetch from the second programmer.
```

Это выбирает их историю, и ничего больше, так что, хотя рабочий каталог остается нетронутыми, мы можем обратиться в любую ветвь в любом хранилище командами Git.

Кстати, за кулисами, **pull** просто **fetch** за которым следует **git merge**; а последний добавляется в рабочую директорию. Обычно мы используем **pull**, потому что мы хотим объединить после выполнения **fetch**; эта ситуация представляет собой заметное исключение.

См. **git help remote** о том, как удалить удаленные хранилища, игнорировать определенные ветки и многое другое.

7.8. Мои Настройки

Для моих проектов я люблю использовать готовые Git репозитории, в которые я могу сделать **pull**. Некоторые Git хостинги позволяют создавать собственные ветки проекта нажатием одной кнопки.

После того как я выгрузил дерево, я запускаю Git команды для навигации и изучения изменения, которые в идеале хорошо организованны и описаны. Я объединяю мои собственные неопубликованные изменения, и, возможно, делаю дальнейшие изменения. После изменения, я выгружаю изменения в официальный репозиторий.

Хотя я редко получаю взносы, я считаю, этот подход хорошо масштабируется. Смотрите этот пост в блоге Линуса Торвальдса (<http://torvalds-family.blogspot.com/2009/06/happiness-is-warm-scm.html>).

Пребывание в мире Git немного более удобно, чем патч-файлы, как это экономит мне шаги конвертации их в коммиты Git. Кроме того, Git автоматически обрабатывает информацию, такую как запись с именем автора и адресом электронной почты, а также время и дату, и просит автора описывать свои собственные изменения.

Глава 8. Гроссмейстерство Git

Эта претенциозно названная глава является собранием приемов работы с Git, которые я не смог отнести к другим главам.

8.1. Релизы исходников

В моих проектах Git управляет только теми файлами, которые я собираюсь архивировать и пускать в релиз. Чтобы создать тарбол с исходниками, я выполняю:

```
$ git archive --format=tar --prefix=proj-1.2.3/ HEAD
```

8.2. Сохранение изменений

Вручную сообщать Git о том, что вы добавили, удалили или переименовали файлы, может стать непростой задачей в некоторых проектах. Вместо этого вы можете выполнить команды:

```
$ git add .  
$ git add -u
```

Git просмотрит файлы в текущем каталоге и обработает изменения сам. Вместо второй команды **add**, выполните **git commit -a**, если вы хотите также сделать коммит с изменениями. В **git help ignore** можно посмотреть, как указать, какие файлы должны игнорироваться.

Вы можете выполнить все это в один прием:

```
$ git ls-files -d -m -o -z | xargs -0 git update-index --add --remove
```

Опции **-z** и **-0** предотвращают побочные эффекты от файловых имен, содержащих специальные символы. Поскольку эта команда добавляет игнорируемые файлы, вы можете использовать опции **-x** или **-X**.

8.3. Слишком большой коммит

Вы пренебрегали коммитами слишком долго? Яростно писали код и вспомнили о контроле исходников только сейчас? Внесли ряд несвязанных изменений, потому что это ваш стиль?

Никаких проблем. Выполните:

```
$ git add -p
```

Для каждого внесенного изменения Git покажет измененный участок кода и спросит, должно ли это изменение пройти в следующем коммите. Отвечаем "y" или "n". Если вы хотите сделать что-то другое, например отложить выбор, введите "?" чтобы получить дополнительную информацию.

Как только все будет готово, выполните:

```
$ git commit
```

для коммита именно тех изменений, которые вы выбрали (*staged* изменения). Убедитесь, что вы не указали опцию **-a**, в противном случае Git добавит в коммит все изменения.

Что делать, если вы изменили множество файлов во многих местах? Просмотр каждого отдельного изменения - удручающая задача. В этом случае используйте **git add -i**, чей интерфейс менее прост, но более гибок. При помощи нескольких нажатий кнопок можно добавить на этап или убрать с этапа несколько файлов одновременно, либо просмотреть и выделить изменения в отдельных файлах. Как вариант, запустите **git commit --interactive**, который автоматически сделает коммит после того, как вы закончите.

8.3.1. Этапные изменения

До сих пор мы избегали такой известной части Git, как *index*, но теперь мы должны разобраться с ней, чтобы пояснить вышесказанное. Индекс представляет собой временную область. Git редко перемещает данные непосредственно между вашим проектом и его историей. Вместо этого, Git сначала записывает данные в индекс, а уж затем копирует данные из индекса по месту назначения.

Например, **commit -a** это на самом деле двухэтапный процесс. Сначала снимок текущего состояния отслеживаемых файлов помещается в индекс. Затем снимок, находящийся в индексе, записывается в историю. Коммит без опции **-a** выполняет только второй этап, и имеет смысл только после выполнения команд, которые изменяют индекс, например **git add**.

Обычно мы можем не обращать внимания на индекс и считать, что взаимодействуем с историей напрямую. Но в такого рода сложных случаях нам нужен усиленный контроль над тем, что записывается в историю, и мы вынуждены работать с индексом. Мы помещаем снимок только части наших изменений в индекс, а потом записываем этот аккуратно сформированный снимок.

8.4. Не теряй HEAD

Тег **HEAD** - как курсор. В нормальном состоянии он указывает на последний коммит, продвигаясь вместе с каждым новым коммитом. Есть команды **Git**, которые позволяют перемещать этот тег. Например:

```
$ git reset HEAD~3
```

переместит **HEAD** на три коммита назад. Теперь все команды **Git** будут работать так, как будто вы не делали последних трех коммитов, хотя файлы останутся в текущем состоянии. В справке описаны некоторые методы использования этого эффекта.

Но как вернуться назад в будущее? Ведь предыдущие коммиты о нем ничего не знают.

Если у вас есть **SHA1** оригинального **HEAD**, то:

```
$ git reset SHA1
```

Но предположим, что вы никогда его не записывали. Тут тоже беспокоиться не стоит. Для команд такого рода **Git** сохраняет оригинальный **HEAD** как тег под названием **ORIG_HEAD**, и вы можете вернуться безопасно и без проблем:

```
$ git reset ORIG_HEAD
```

8.5. Охота за HEAD'ами

Предположим **ORIG_HEAD** недостаточно. Предположим, что вы только что осознали, что допустили громадную ошибку, и вам нужно вернуться в очень старый коммит давно забытой ветки.

По умолчанию **Git** хранит коммиты по крайней мере в течении двух недель, даже если вы сказали ему уничтожить ветку с ними. Проблема в нахождении подходящего хеша.

Вы можете просмотреть хеши в `.git/objects` и методом проб и ошибок найти нужный. Но есть путь значительно легче.

Git записывает все хеши коммитов в `.git/logs`. В папке `refs` содержится история активности на всех ветках, а файл **HEAD** содержит каждое значение хеша, которое когда-либо принимал **HEAD**. Второе можно использовать чтобы найти хеши коммитов на случайно обрубленных ветках.

Команда **reflog** предоставляет удобный интерфейс работы с этими логами. Используйте:

```
$ git reflog
```

Вместа копипейста хешей из **reflog**, попробуйте:

```
$ git checkout "@{10 minutes ago}"
```

Или сделайте чекаут пятого из последних посещенных коммитов с помощью:

```
$ git checkout "{5}"
```

Смотрите секцию "Specifying Revisions" **git help rev-parse**, если вам нужна дополнительная информация. Вам может потребоваться установить более долгий период сохранения удаляемых коммитов.

Например, выполнение:

```
$ git config gc.pruneexpire "30 days"
```

означает, что в удаленные коммиты будут окончательно потеряны только после того, как пройдут 30 дней с момента удаления, и будет запущена **git gc**.

Также вам может потребоваться отключить автоматическое выполнение **git gc**:

```
$ git config gc.auto 0
```

После этого коммиты будут удаляться только когда вы будете запускать **git gc** самостоятельно.

8.6. Git как основа

Дизайн Git'a, разработанный в UNIX стиле, позволяет использовать Git как низкоуровневый компонент других программ: GUI, веб-интерфейсов, альтернативных командных строк, инструментов управления патчами, импортирования, преобразования, и т.д. На самом деле, многие команды Git - сами по себе скрипты, стоящие на плечах гигантов. Немного поигравшись с Git, вы можете заставить его удовлетворять многие ваши потребности.

Простейший трюк - использование алиасов Git для выполнения часто

используемых команд:

```
$ git config --global alias.co checkout
$ git config --global --get-regexp alias # отображает текущие алиасы
```

```
alias.co checkout
$ git co foo # то-же, что 'git checkout foo'
```

Также можно выводить текущую ветку в командную строку или название окна терминала.

Запуск

```
$ git symbolic-ref HEAD
```

выводит название текущей ветки. Для практического использования вы скорее всего захотите убрать "refs/heads/" и сообщения об ошибках:

```
$ git symbolic-ref HEAD 2> /dev/null | cut -b 12-
```

Папка `contrib` это целая сокровищница инструментов, построенных на Git. Со временем некоторые из них могут становиться официальными командами. Под **Debian** и **Ubuntu** эта папка находится в `/usr/share/doc/git-core/contrib`.

`workdir/git-new-workdir` - один из популярных и часто используемых инструментов. С помощью хитрых симлинков этот скрипт создает новую рабочую папку, которая будет иметь общую историю с оригинальным репозиторием:

```
$ git-new-workdir an/existing/repo new/directory
```

Можно думать о новой папке и о файлах в ней, как о клоне, за исключением того, что так как история общая, два дерева автоматически синхронизируются. Нет необходимости в **merge**, **push** и **pull**.

8.7. Опасные трюки

Случайно уничтожить данные очень сложно с сегодняшними версиями Git. Но если вы знаете, что делаете, вы можете обойти защиту для общих команд.

Checkout: Незакоммиченные изменения не дают сделать чекаут. Чтобы все-таки сделать чекаут нужного коммита, уничтожив свои изменения, используется флаг **-f**:

```
$ git checkout -f COMMIT
```

С другой стороны, если вы укажете отдельный путь для чекаута, то проверки на безопасность произведены не будут, и по указанным путям все будет переписываться без каких-либо сообщений. Будьте осторожны, если вы используете чекаут таким образом.

Reset: Ресет также нельзя провести, если есть незакоммиченные изменения. Чтобы обойти это, запустите:

```
$ git reset --hard [COMMIT]
```

Branch: Удаление ветки не проходит, если приводит к потере изменений. Для принудительного удаления используйте:

```
$ git branch -D BRANCH # вместо -d
```

Аналогично, попытка перезаписи ветки путем перемещения не пройдет, если какие-то данные будут потеряны. Чтобы принудительно переместить ветку, введите:

```
$ git branch -M [SOURCE] TARGET #вместо -m
```

В отличие от чекаута и ресета, эти две команды задерживают удаление данных. Изменения остаются в папке `.git` и могут быть восстановлены с помощью нужного хеша из `.git/logs`(смотрите параграф "Охота за HEAD'ами" выше).

По умолчанию они будут храниться по крайней мере две недели.

Clean: Некоторые команды не будут выполняться, если они могут повредить неотслеживаемые файлы. Если вы уверены, что все неотслеживаемые файлы и папки вам не нужны, то безжалостно удаляйте их командой:

```
$ git clean -f -d
```

В следующий раз эта надоедливая команда выполнится!

8.8. Улучшаем свой публичный образ

История многих моих проектов полна глупых ошибок. Самое ужасное это кучи недостающих файлов, которые появляются, когда забываешь выполнить **git add**. К счастью, я пока не терял важных файлов из-за того, что пропускал их, потому что я редко удаляю оригинальные рабочие папки. Обычно я замечаю ошибку несколько коммитов спустя, так что единственный вред это отсутствующая история и осознание своей вины.

Также я регулярно совершаю(и коммичу) меньшее зло - завершающие пробелы. Несмотря на безвредность, я не хотел бы, чтобы это появлялось в публичных записях.

И наконец, я беспокоюсь о неразрешенных конфликтах, хотя пока они не приносили вреда. Обычно я замечаю их во время билда, но в некоторых случаях могу проглядеть. Если бы я

только поставил защиту от дурака, используя хук, который бы предупреждал меня об этих проблемах...

```
$ cd .git/hooks
$ cp pre-commit.sample pre-commit # В старых версиях Git: chmod +x pre-commit
```

Теперь Git отменяет коммит, если обнаружены ненужные пробелы или неразрешенные конфликты. Для этого руководства я в конце концов добавил следующее в начало **pre-commit** хука, чтобы защититься от своей рассеянности:

```
if git ls-files -o | grep \.txt$; then echo FAIL! Неотслеживаемые .txt файлы. exit 1 fi
```

Несколько операций Git поддерживают хуки, смотрите **git help hooks**. Вы можете добавить хуки, которые будут сообщать о грамматических ошибках в комментариях к коммиту, добавлять новые файлы, делать отступы перед параграфами, добавлять записи на веб-страничку, проигрывать звуки, в общем, делать что угодно...

Мы встречались с **post-update** хуком раньше, когда обсуждали Git через <http>. Этот хук обновлял несколько файлов, которые необходимы для ненативных видов коммуникации с Git.

Глава 9. Раскрываем тайны

Мы заглянем под капот и объясним, как Git творит свои чудеса. Я опущу некоторые детали. Для более подробного изучения описаний обратитесь к Руководству пользователя (<http://www.kernel.org/pub/software/scm/git/docs/user-manual.html>).

9.1. Невидимость

Как Git может быть таким ненавязчивым? Помимо коммитов время от времени и слияния, вы можете работать так, как будто вы не знали, что контроль версий существует. То есть, пока вам это не нужно, вы его не замечаете, и вот, когда он понадобился вы рады, что Git наблюдал за вами все это время.

Другие системы контроля версий, не позволят вам забыть о них. Права доступа к файлам может быть только для чтения, пока вы явно не укажете серверу, какие файлы вы хотите изменить. Центральный сервер может отслеживать кто извлекал какой код и когда. Когда сеть падает вы можете от этого пострадать. Разработчики постоянно борются с виртуальной волокитой и бюрократизмом.

Секрет заключается в каталоге `.git` в вашей рабочей директории. Git хранит историю вашего проекта здесь. Из-за имени, начинающегося с точки каталог не отображается в выводе команды `ls`. Кроме команд **git push** и **git pull** все остальные операции контроля версий работают в этом каталоге.

Вы имеете полный контроль над судьбой ваших файлов потому что Git не важно, что вы делаете с ними. Git можно легко восстановить сохраненное в `.git` состояние в любое время.

9.2. Целостность

Большинство людей ассоциируют с криптографией поддержание информации в секрете, но другой не менее важной задачей является поддержание информации в сохранности. Правильное использование криптографических хеш-функций может предотвратить случайные или злонамеренные повреждения данных.

SHA1 хэш может рассматриваться как уникальный 160-битный идентификационный номер для каждой строки байт, с которой вы сталкиваетесь в вашей жизни. На самом деле гораздо больше: каждая строка байтов, которую любой человек когда-нибудь использует в течение многих жизней.

Так как **SHA1** хэш сам является последовательностью байтов, мы можем получить хэш строки байтов, содержащей другие хэши. Это простое наблюдение на удивление полезно: смотрите *hash chains* (цепи хешей). Позднее мы увидим, как **Git** использует их для эффективного обеспечения целостности данных.

Короче говоря, **Git** хранит ваши данные в подкаталоге `".git/objects"`, где вместо обычных файлов, вы увидите только **ID**. С помощью идентификаторов как имен файлов, а также нескольких лок-файлов и трюков с временем создания файлов, **Git** преобразует любую скромную файловую систему в эффективную и надежную базу данных.

9.3. Интеллект

Каким образом **Git** знает, что вы переименовали файл, даже если вы никогда не упоминается тот факт явно? Конечно, вы можете запустить **git mv**, но это точно так же, как **git rm** и после **git add**.

Git эвристически находит переименованные файлы и копирует их в последующие версии. В самом деле, он может обнаружить, что куски кода были перемещены или скопированы между файлами! Хотя она не может охватить все случаи, это достойная работа, и эта функция всегда улучшается. Если это не работает, попробуйте включить опцию обнаружения копирования и рассмотреть вопрос апгрейда.

9.4. Индексация

Для каждого отслеживаемого файла, **Git** записывает информацию, такую как размер, время создания и время последнего изменения в файл, известный как "индекс". Чтобы определить, что файл был изменен, **Git** сравнивает его текущее состояние с тем, что сохранено в индексе. Если они совпадают, то **Git** может пропустить перечитывание это файла.

Поскольку чтение этой информации значительно быстрее, чем чтение всего файла, то если вы редактировали только несколько файлов, **Git** может обновить свой индекс почти мгновенно.

9.5. Голые репозитории

Вам, возможно, было интересно, какой формат используется в этих онлайн **Git** репозиториях. Они такие-же хранилища **Git**, как ваш каталог `.git`, кроме того что обычно называются `proj.git`, и они не имеют рабочую директорию связанную с ними.

Большинство команд **Git** рассчитывают что индекс **Git** находится в каталоге `.git`, и не смогут работать на этих голых хранилищах. Исправить это можно, установив переменную окружения `GIT_DIR` в значение, равное пути к репозиторию, или запустить **Git** в этом каталоге с опцией `--bare`.

9.6. Происхождение **Git**

Этот <http://lkml.org/lkml/2005/4/6/121> [пост] в **Linux Kernel Mailing List** описывает цепь событий, которые привели к появлению **Git**. Весь этот трейд - увлекательные археологические раскопки для историков **Git**.

9.7. База данных объектов

Вот как писать **Git**-подобной операционной системы с нуля в течение нескольких часов.

9.7.1. Blobs

Первый волшебный трюк. Выберите имя файла, любое имя файла. В пустой директории:

```
$ echo sweet > YOUR_FILENAME
$ git init
$ git add .
$ find .git/objects -type f
```

Вы увидите `.git/objects/aa/823728ea7d592acc69b36875a482cdf3fd5c8d`.

Откуда я знаю это, не зная имени файла? Это потому, что **SHA1** хэш строки:

```
"blob" SP "6" NUL "sweet" LF
```

является `aa823728ea7d592acc69b36875a482cdf3fd5c8d`, где **SP** это пробел, **NUL** является нулевым байтом и **LF** переводом строки. Вы можете проверить это, напечатав:

```
$ echo "blob 6"$'\001'"sweet" | tr '\001' '\000' | shasum
```

Кстати, это написано с учетом особенностей оболочки **Bash**, другие оболочки возможно способны обработать **NUL** в командной строке, что исключает необходимость использовать костыль с **tr**.

Git является *контент-адресуемым*: файлы хранятся в независимости от их имени, а по хэшу содержимого, которое мы называем **BLOB объект**. Мы можем думать о хеше как о уникальном идентификаторе для содержимого файла, так что в некотором смысле мы обращаемся к файлам по их содержимому. Начальный "blob 6" является лишь заголовком, состоящий из типа объекта и его длины в байтах; она упрощает внутренний учет.

Таким образом, я могу легко предсказать, что вы увидите. Имя файла не имеет никакого отношения: только данные внутри используется для построения BLOB объекта.

Вам может быть интересно, что происходит с идентичными файлами. Попробуйте добавить копии с любыми именами файлов вообще. Содержание `.git/objects` останется тем-же независимо от того, сколько копий вы добавите. Git только хранит данные один раз.

Кстати, файлы в директории `.git/objects` сжимаются с Zlib поэтому вы не сможете просмотреть их непосредственно. Пропустите их через фильтр <http://www.zlib.net/zpipe.c> [zpipe-D], или введите:

```
$ git cat-file -p aa823728ea7d592acc69b36875a482cdf3fd5c8d
```

который просто выведет данный объект.

9.7.2. Деревья

Но где же имена файлов? Они должны храниться где-то на определенном этапе. Git получает информацию об имени во время коммита:

```
$ git commit # Type some message.
$ find .git/objects -type f
```

Теперь вы должны увидеть 3 объекта. На этот раз я не могу сказать вам, какие 2 новые файлы, так как это частично зависит от выбранного имени файла. Допустим вы назвали его "rose". Если это не так, то вы можете переписать историю, чтобы она выглядела как будто вы это сделали:

```
$ git filter-branch --tree-filter 'mv YOUR_FILENAME rose'
$ find .git/objects -type f
```

Теперь вы должны увидеть файл

`.git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9`, потому что это SHA1 хэш его содержимого:

```
"tree" SP "32" NUL "100644 rose" NUL 0xaa823728ea7d592acc69b36875a482cdf3fd5c8d
```

Проверьте - этот файл действительно содержит указанную выше строку - наберите:

```
$ echo 05b217bb859794d08bb9e4f7f04cbda4b207fbe9 | git cat-file --batch
```

С `zpipe` легко проверить хеш:

```
$ zpipe -d < .git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9 | shasum
```

Проверка хеша сложнее чем через CAT-файла, поскольку его вывод содержит больше, чем сырой несжатый объектный файл.

Этот файл является объектом *tree*: список цепочек, состоящих из типа файла, имени файла, и хэша. В нашем примере это тип файла "100644", это означает что "rose", является обычным файлом и хэш BLOB объект, в котором находится содержимое "rose". Другие возможные типы файлов - исполняемые файлы, символические ссылки или каталоги. В последнем случае, хэш указывает на дереве объектов.

Если вы запускали `filter-branch`, у вас будут старые объекты которые вам больше не нужны. Хотя они будут автоматически выброшены сразу после истечения льготного периода, мы удалим их сейчас, чтобы наш игрушечный пример было легче исследовать:

```
$ rm -r .git/refs/original
$ git reflog expire --expire=now --all
$ git prune
```

Для реальных проектов, обычно вы должна избегать использовать такие команды, как эта, так как вы разрушаете резервные копии. Если вы хотите чистое хранилище, то обычно лучше сделать новый клон. Кроме того, будьте внимательны при непосредственном манипулировании `.git`: Что делать, если другая команда `Git` будет запущена в то же время, или внезапного произойдет отключение питания? В общем случае, ссылки должны быть удалены с помощью `git update-ref -d`, хотя обычно удалить `refs/original` вручную безопасно.

9.7.3. Коммиты

Мы объяснили 2 из 3 объектов. Третий объект - *коммит*. Его содержимое зависит от сообщения коммита, а также от даты и времени его создания. Для демонстрации того, что мы здесь имеем, мы должны настроить `Git` немного:

```
$ git commit --amend -m Shakespeare # Change the commit message.
$ git filter-branch --env-filter 'export
    GIT_AUTHOR_DATE="Fri 13 Feb 2009 15:31:30 -0800"
    GIT_AUTHOR_NAME="Alice"
    GIT_AUTHOR_EMAIL="alice@example.com"
    GIT_COMMITTER_DATE="Fri, 13 Feb 2009 15:31:30 -0800"
    GIT_COMMITTER_NAME="Bob"
    GIT_COMMITTER_EMAIL="bob@example.com"' # Rig timestamps and authors.
```

```
$ find .git/objects -type f
```

Теперь вы должны увидеть `.git/objects/49/993fe130c4b3bf24857a15d7969c396b7bc187` который является **SHA1** хэшем его содержание:

```
"commit 158" NUL
"tree 05b217bb859794d08bb9e4f7f04cbda4b207fbe9" LF
"author Alice <alice@example.com> 1234567890 -0800" LF
"committer Bob <bob@example.com> 1234567890 -0800" LF
LF
"Shakespeare" LF
```

Как и раньше, вы можете запустить `zpipe` или `cat-file`, чтобы увидеть это самостоятельно.

Это первый коммит, и поэтому нет родительского коммита, но последующие коммиты всегда будет содержать хотя бы одну строку идентифицирующую родительский коммит.

9.7.4. Неотлично от магии

Там мало сказано. Мы только что открыли секрет мощи **Git**. Это кажется слишком простым: похоже, что вы могли бы смешать вместе несколько скриптов оболочки и добавить немного кода на **C**, сделанного в считанные часы. По сути, это точное описание ранних версий **Git**. Тем не менее, помимо гениальных трюков упаковки, чтобы сэкономить место, и трюков индексации, чтобы сэкономить время, мы теперь знаем, как ловко **Git** преобразует файловую систему в базу данных, идеально подходящую для контроля версий.

Например, если какой-то файл объекта базы данных повредила ошибка диска, то его хэш больше не совпадает, предупреждая о проблеме. При хешировании хэшей других объектов, мы сохраняем целостность на всех уровнях. Коммит являются атомными, то есть, никогда нельзя закоммитить лишь часть изменений: мы можем только вычислить хэш коммита и сохранить его в базу данных после того как мы сохраним все соответствующие деревья, blobs и родительские коммиты. Объектная база данных застрахована от неожиданных прерываний работы с ней таких как перебои в подаче электроэнергии.

Мы наносим поражение даже самым хитрым противникам. Пусть кто-то попытается тайно изменить содержимое файла в древней версии проекта. Чтобы сохранить объектную базу данных согласованной, они также должны изменить хэш соответствующего объекта **BLOB**, поскольку это теперь другая последовательность байтов. Это означает, что нужно поменять хэш всех деревьев, содержащих ссылки на объект этого файла, что в свою очередь изменит хэши коммитов всех объектов с участием таких деревьев, в дополнение к хэсам всех потомков этих коммитов. Это означает, хэш официальной головной ревизии будет отличаться от хэша в этом плохом хранилище. По следам несовпадения хэшей мы можем локализовать изуродованный файл, а также коммит, где он впервые был поврежден.

Короче говоря, пока 20 байт представляющие последний коммит в безопасности, невозможно изменить репозиторий **Git**.

Как насчет знаменитых черт **Git**? Создание ветки? Слияние? Теги? Более подробно. Текущая **HEAD** хранится в файле `.git/HEAD`, который содержит хэш объекта фиксации. Хэш обновляется во время коммита, а также при выполнении многих других команд. Ветки почти одинаковы: они представляют собой файлы в `.git/refs/heads`. Теги тоже: они живут в `.git/refs/tags`, но они обновляются различными наборами команд.

Глава 10. Недостатки Git

Есть некоторые проблемы Git, которые я спрятал под сукно. Некоторые из них можно легко решить с помощью сценариев и хуков, некоторые требуют реорганизации или пересмотра проекта, и из нескольких оставшихся неприятности, одну придется терпеть. Или еще лучше, взяться за нее и решить!

10.1. Недостатки SHA1

Со временем, криптографы обнаружат более слабые SHA1. Уже нахождение хэш столкновений является допустимым для хорошо финансируемой организации. За годы, возможно, даже типичный ПК будет иметь достаточную вычислительную мощность, чтобы потихоньку скоррумпировать Git репозиторий.

Надеюсь Git мигрирует к лучшей хэш-функции до того, как дальнейшие исследования разрушат SHA1.

10.2. Microsoft Windows

Git на Microsoft Windows может быть накладным:

- Cygwin (<http://cygwin.com/>), Linux-среда для Windows, содержит порт Git на Windows (<http://cygwin.com/packages/git/>).

-Git на MSys (<http://code.google.com/p/msysgit/>) вариант, требующий минимальной рантайм поддержки, хотя для нескольких команд, нужна доработка.

10.3. Несвязанные файлы

Если ваш проект очень большой и содержит много не связанных файлов, которые постоянно изменяются, Git может оказаться в невыгодном положении больше, чем другие системы, поскольку отдельные файлы не отслеживаются. Git отслеживает изменения всего проекта, такой подход, как правило, полезен.

Решение - разбить проект на части, каждая из которых состоит из связанных файлов. Используйте **git submodule** если вы все еще хотите держать все в одном репозитории.

10.4. Кто и что редактировал ?

Некоторые системы контроля версий заставляют вас отметить файл до редактирования. Хотя это особенно раздражает, когда речь идет о работе с центральным сервером, этот способ имеет два преимущества:

1. Diff'ы быстры, потому что только отмеченные файлы должны быть изучены.
2. Можно обнаружить, кто еще работает с этим файлом, спросив центральный сервер, кто отметил его для редактирования.

При наличии соответствующих сценариев, вы можете добиться того же с Git. Это требует сотрудничества со стороны других программистов, которые должны выполнять сценарии при редактировании файла.

10.5. История файлов

Поскольку Git записывает изменения всего проекта, то чтобы восстановить историю одного файла требуется больше работы, чем в системах управления версиями, которые позволяют отслеживать отдельные файлы.

Потери, как правило, небольшие, а также стоит иметь в виду, что другие операции являются невероятно эффективными. Например, `git checkout` быстрее, чем `cp -a`, и вся дельта проекта сжимаются лучше, чем коллекций на основе дельт файлов.

10.6. Начальное Клонирование

Создание клона репозитория дороже обычного чекаута в другие системы управления версиями, особенно когда существует большая история.

Первоначальная цена окупается в долгосрочной перспективе, так как большинство операций, затем будут происходить быстро и в автономном режиме. Однако в некоторых ситуациях может быть предпочтительным создание мелких клонов с опцией `--depth`. Это намного быстрее, но в результате снижается функциональность клона.

10.7. Изменчивые Проекты

Git была написана, чтобы быть быстрым по отношению к большим изменениям. Люди делают небольшие исправления от версии к версии. Однострочные исправление здесь, новые функции там, исправленные комментарии, и так далее. Но если ваши файлы, радикально

отличаются в следующей ревизии, то на каждой фиксации, ваша история обязательно увеличивается на размер всего вашего проекта.

Не существует системы управления версиями, которая может решить эту проблему, но пользователи Git пострададут больше, поскольку клонируется вся история.

Причины, почему эти изменения бывают настолько велики, должны быть изучены. Возможно, формат файла должен быть изменен. Малые изменения должны быть причиной малых изменений в самих файлах.

Или, возможно, то, что вам было нужно, это базы данных или система резервного копирования или архивы, а не система контроля версий. Например, контроль версий может быть плохо приспособлен для управления фотографиями периодически получаемых с веб-камеры.

Если файлы действительно должны постоянно изменяться, и они действительно должны быть версионироваться, возможность использовать Git централизованным образом. Можно создать мелкий клон, с небольшой или без истории проекта. Конечно, многие инструменты Git будут недоступны, и исправления должны быть представлены в виде патчей. Вероятно, это штраф, потому как неясно, почему кто-то хочет вести историю дико неустойчиво файлов.

Другим примером является проект, зависящий от прошивки, которая принимает форму огромного бинарного файла. История этой микропрограммы неинтересна для пользователей, и обновления сжимаются плохо, так версии микропрограммы могут излишне увеличить размера хранилища.

В этом случае исходный код должен храниться в хранилище Git, а бинарные файлы - отдельно. Чтобы сделать жизнь проще, можно было бы распространять скрипт, который используется Git чтобы клонировать код и Rsync или мелкий клон Git для прошивки.

10.8. Глобальный счетчик

Некоторые централизованные системы контроля версий, используют положительные числа, которые возрастают, когда происходит новый коммит. Git идентифицирует изменения их хэшем, который лучше во многих обстоятельствах.

Но некоторым людям нравится эти целые вокруг. К счастью, легко написать сценарии, чтобы при каждом обновлении центральный репозиторий Git увеличивал целое число, возможно, в тегах, и связывает его с хэшем последним коммитом.

Каждый клон может поддерживать такой счетчик, но это, вероятно, будет бесполезным, поскольку только центральный репозиторий имеет значение для всех.

10.9. Пустые подкаталоги

Пустые подкаталоги не могут быть отслежены. Создайте пустой файл, чтобы обойти эту проблему.

В этом виноват не дизайн Git, а его текущая реализация. Если повезет и пользователи Git уделят больше внимания этой функции, возможно она будет реализована.

10.10. Первоначальный коммит

Обычный ученый в области информатики считает от 0, а не от 1. К сожалению, Git с его коммитами не присоединиться к этой конвенции. Многие команды плохо работают до первого коммита. Кроме того, некоторые частные случаи должны быть обработаны специально, такие, как **rebase** веток с различными начальными коммитами.

Git'у желательно определение нулевого совершенства: как только будет построено хранилище, **HEAD** будет установлен в строку, состоящую из 20 нулевых байтов. Эта специальный коммит представляет собой пустое дерево, без родителей, предшествовавшие всему в Git репозитории.

Затем запустить **git log**, например, будет показывать пользователю, что коммиты еще не были сделаны вместо выхода с фатальной ошибкой. Аналогично для других инструментов.

Каждая первоначальная фиксация - неявный потомок этого нулевого коммита. Например, несвязанный с какой-либо веткой **rebase** в целом будет привита на эту цель. В настоящее время применяются все, кроме первоначального коммита, в результате чего получается конфликт слияния. Один из способов заключается в использовании `git checkout` после `git commit` -с первоначального коммита, тогда **rebase** пройдет нормально.

К сожалению, в худшем случае, если несколько ветвей с различными начальными коммитами сливаются, то **rebase** результата требует значительного ручного вмешательства.

Глава 11. Приложение А: Перевод этого руководства

Клонируйте исходные тексты, затем создайте соответствующую директорию языковой тег IETF (<http://www.iana.org/assignments/language-subtag-registry>): см статья W3C по интернационализации (<http://www.w3.org/International/articles/language-tags/Overview.en.php>). К примеру, английский язык "en", японский "ja", традиционный китайский "zh-Hant". Скопируйте в директорию файлы `txt` из "en" поддиректории и переведите их.

К примеру, для перевода руководства на Klingon (http://en.wikipedia.org/wiki/Klingon_language), вы можете набрать:

```
$ git clone git://repo.or.cz/gitmagic.git
$ cd gitmagic
$ mkdir tlh # "tlh" - IETF языковой код для Klingon.
$ cd tlh
$ cp ../en/intro.txt .
$ edit intro.txt # Переведите файл.
```

и так с каждым файлом. Вы можете просмотреть всю вашу работу:

```
$ make LANG=tlh
$ firefox book.html
```

Почаще делайте коммиты, потом дайте знать когда ваш перевод готов На [GitHub.com](https://github.com) есть веб-интерфейс, который позволяет упростить всю работу: сделать форк "gitmagic" проекта, внести ваши изменения, потом попросить меня о слиянии. Я бы хотел чтобы переводы придерживались такой схемы и мои скрипты могли легко создать HTML и PDF версии. Также эта схема позволяет удобно хранить все переводы в официальном репозитории. Но пожалуйста, поступайте как вам удобнее: к примеру, китайский переводчик использовал Google Docs. Я рад тому что ваша работа открывает доступ большому числу людей к моей работе.