ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ЛЕКЦИЯ № 4

ПРЕПОДАВАТЕЛЬ: ХУСТОЧКА А.В.

СИСТЕМА КОНТРОЛЯ ВЕРСИЙ

- Контроль версий, также известный как управление исходным кодом, — это практика отслеживания изменений программного кода и управления ими. Системы контроля версий — это программные инструменты, помогающие командам разработчиков управлять изменениями в исходном коде с течением времени. В свете усложнения сред разработки они помогают командам разработчиков работать быстрее и эффективнее.
- Разработчики программного обеспечения, работающие в командах, постоянно пишут новый исходный код и изменяют существующий. Код проекта, приложения или программного компонента обычно организован в виде структуры папок или «дерева файлов». Один разработчик в команде может работать над новой возможностью, а другой в это же время изменять код для исправления несвязанной ошибки, т. е. каждый разработчик может вносить свои изменения в несколько частей дерева файлов.

СИСТЕМА КОНТРОЛЯ ВЕРСИЙ

- Контроль версий помогает командам решать подобные проблемы путем отслеживания каждого изменения, внесенного каждым участником, и предотвращать возникновение конфликтов при параллельной работе. Изменения, внесенные в одну часть программного обеспечения, могут быть не совместимы с изменениями, внесенными другим разработчиком, работавшим параллельно. Такая проблема должна быть обнаружена и решена согласно регламенту, не создавая препятствий для работы остальной части команды. Кроме того, во время разработки программного обеспечения любое изменение может само по себе привести к появлению новых ошибок, и новому ПО нельзя доверять до тех пор, пока оно не пройдет тестирование. Вот почему процессы тестирования и разработки идут рука об руку, пока новая версия не будет готова.
- Хорошее программное обеспечение для управления версиями поддерживает предпочтительный рабочий процесс разработчика, не навязывая определенный способ работы. В идеале оно также работает на любой платформе и не принуждает разработчика использовать определенную операционную систему или цепочку инструментов. Хорошие системы управления версиями обеспечивают плавный и непрерывный процесс внесения изменений в код и не прибегают к громоздкому и неудобному механизму блокировки файлов, который дает зеленый свет одному разработчику, но при этом блокирует работу других.

GIT

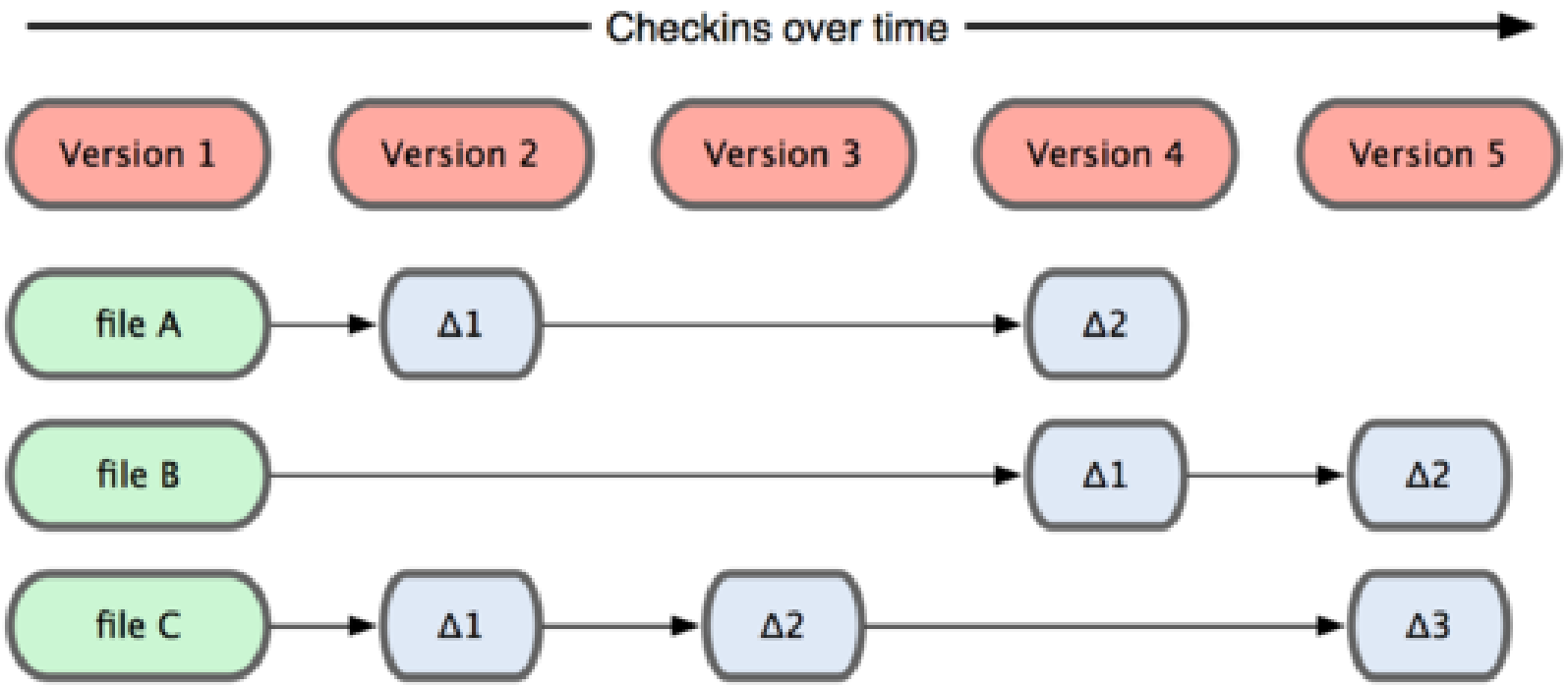
- Git — абсолютный лидер по популярности среди современных систем управления версиями. Это развитый проект с активной поддержкой и открытым исходным кодом. Система Git была изначально разработана в 2005 году Линусом Торвальдсом — создателем ядра операционной системы Linux. Git применяется для управления версиями в рамках колоссального количества проектов по разработке ПО, как коммерческих, так и с открытым исходным кодом. Система используется множеством профессиональных разработчиков программного обеспечения. Она превосходно работает под управлением различных операционных систем и может применяться со множеством интегрированных сред разработки (IDE).

GIT

- Git — система управления версиями с распределенной архитектурой. В отличие от некогда популярных систем вроде CVS и Subversion (SVN), где полная история версий проекта доступна лишь в одном месте, в Git каждая рабочая копия кода сама по себе является репозиторием. Это позволяет всем разработчикам хранить историю изменений в полном объеме.
- Разработка в Git ориентирована на обеспечение высокой производительности, безопасности и гибкости распределенной системы.

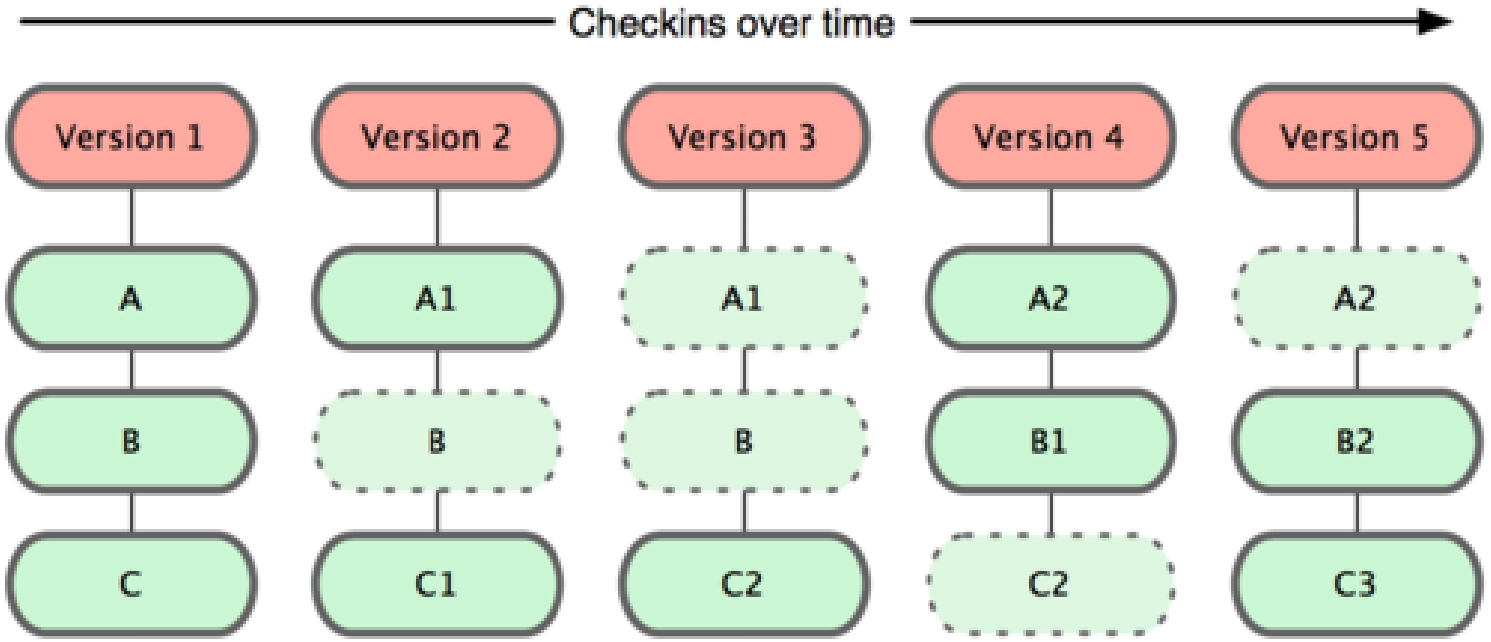
GIT

- Главное отличие Git'а от любых других СКВ (например, Subversion и ей подобных) — это то, как Git смотрит на свои данные. В принципе, большинство других систем хранит информацию как список изменений (патчей) для файлов. Эти системы (CVS, Subversion, Perforce, Bazaar и другие) относятся к хранимым данным как к набору файлов и изменений, сделанных для каждого из этих файлов во времени.



GIT

- Git не хранит свои данные в таком виде. Вместо этого Git считает хранимые данные набором слепков небольшой файловой системы. Каждый раз, когда вы фиксируете текущую версию проекта, Git, по сути, сохраняет слепок того, как выглядят все файлы проекта на текущий момент. Ради эффективности, если файл не менялся, Git не сохраняет файл снова, а делает ссылку на ранее сохранённый файл. То, как Git подходит к хранению данных.



GIT

- Это важное отличие Git’а от практически всех других систем контроля версий. Из-за него Git вынужден пересмотреть практически все аспекты контроля версий, которые другие системы переняли от своих предшественниц. Git больше похож на небольшую файловую систему с невероятно мощными инструментами, работающими поверх неё, чем на просто систему контроля версий. Позже, коснувшись работы с ветвями в Git’е, мы узнаем, какие преимущества даёт такое понимание данных.

ПОЧТИ ВСЕ ОПЕРАЦИИ — ЛОКАЛЬНЫЕ

- Для совершения большинства операций в Git'е необходимы только локальные файлы и ресурсы, т.е. обычно информация с других компьютеров в сети не нужна. Если вы пользовались централизованными системами, где практически на каждую операцию накладывается сетевая задержка, вы, возможно, подумаете, что боги наделили Git неземной силой. Поскольку вся история проекта хранится локально у вас на диске, большинство операций кажутся практически мгновенными.
- К примеру, чтобы показать историю проекта, Git'у не нужно скачивать её с сервера, он просто читает её прямо из вашего локального репозитория. Поэтому историю вы увидите практически мгновенно. Если вам нужно просмотреть изменения между текущей версией файла и версией, сделанной месяц назад, Git может взять файл месячной давности и вычислить разницу на месте, вместо того чтобы запрашивать разницу у CKB-сервера или качать с него старую версию файла и делать локальное сравнение.
- Кроме того, работа локально означает, что мало чего нельзя сделать без доступа к Сети или VPN. Если вы в самолёте или в поезде и хотите немного поработать, можно спокойно делать коммиты, а затем отправить их, как только станет доступна сеть. Если вы пришли домой, а VPN-клиент не работает, всё равно можно продолжать работать. Во многих других системах это не возможно или же крайне неудобно. Например, используя Perforce, вы мало что можете сделать без соединения с сервером. Работая с Subversion и CVS, вы можете редактировать файлы, но сохранить изменения в вашу базу данных нельзя (потому что она отключена от репозитория). Вроде ничего серьёзного, но потом вы удивитесь, насколько это меняет дело.

ЦЕЛОСТНОСТЬ ДАННЫХ

- Перед сохранением любого файла Git вычисляет контрольную сумму, и она становится индексом этого файла. Поэтому невозможно изменить содержимое файла или каталога так, чтобы Git не узнал об этом. Эта функциональность встроена в сам фундамент Git'а и является важной составляющей его философии. Если информация потеряется при передаче или повредится на диске, Git всегда это выявит.
- Механизм, используемый Git'ом для вычисления контрольных сумм, называется SHA-1 хешем. Это строка из 40 шестнадцатеричных символов (0-9 и a-f), вычисляемая в Git'е на основе содержимого файла или структуры каталога. SHA-1 хеш выглядит примерно так:
  24b9da6552252987aa493b52f8696cd6d3b00373
- Работая с Git'ом, вы будете встречать эти хеши повсюду, поскольку он их очень широко использует. Фактически, в своей базе данных Git сохраняет всё не по именам файлов, а по хешам их содержимого

ЧАЩЕ ВСЕГО ДАННЫЕ В GIT ТОЛЬКО ДОБАВЛЯЮТСЯ

- Практически все действия, которые вы совершаете в Git’е, только добавляют данные в базу. Очень сложно заставить систему удалить данные или сделать что-то неотменяемое. Можно, как и в любой другой системе контроля версий, потерять данные, которые вы ещё не сохранили, но как только они зафиксированы, их очень сложно потерять, особенно если вы регулярно отправляете изменения в другой репозиторий.
- Поэтому пользоваться Git’ом — удовольствие, потому что можно экспериментировать, не боясь что-то серьёзно поломать.

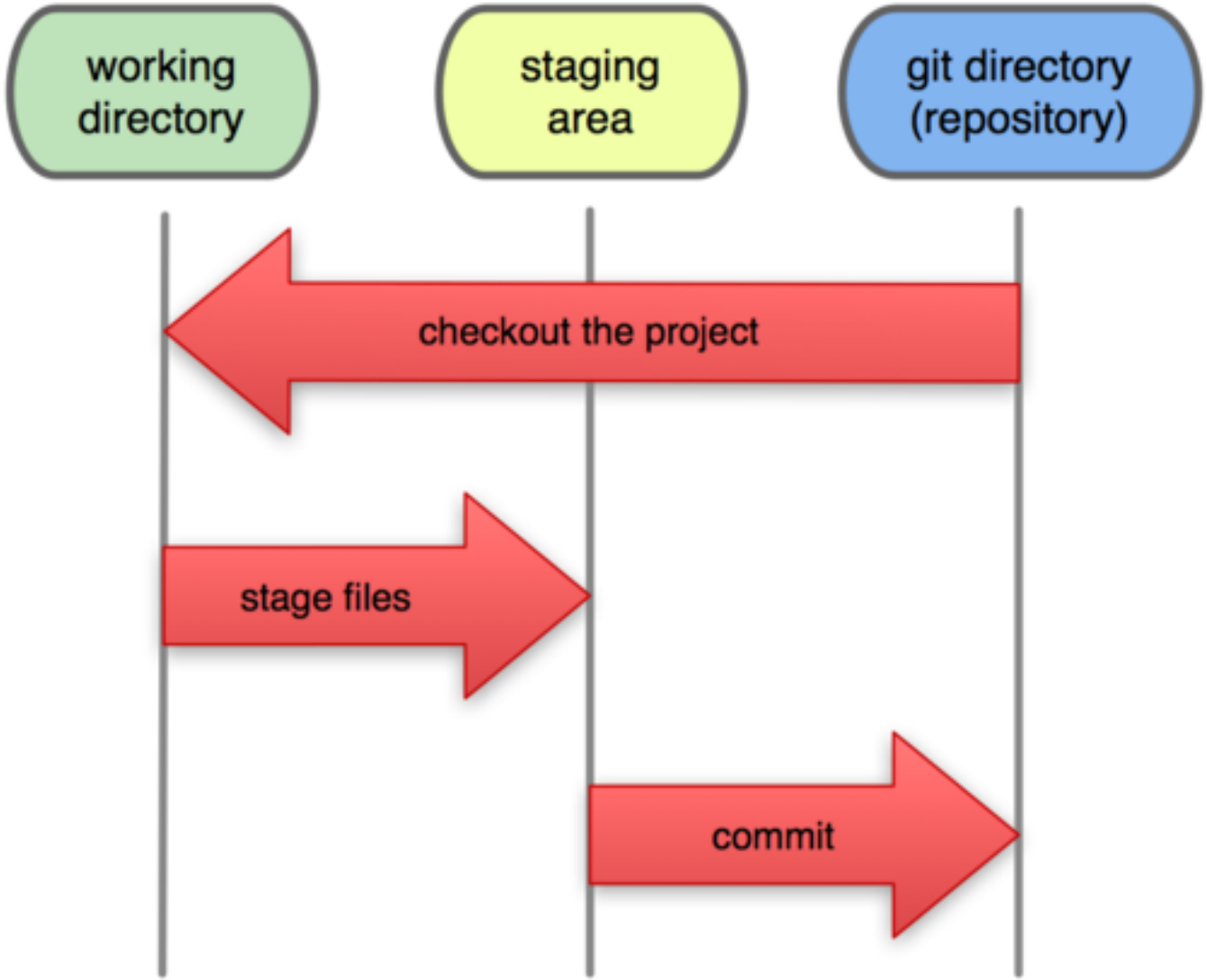
ТРИ СОСТОЯНИЯ

Теперь внимание. Это самое важное, что нужно помнить про Git, если вы хотите, чтобы дальше изучение шло гладко. В Git'е файлы могут находиться в одном из трёх состояний: зафиксированном, изменённом и подготовленном. "Зафиксированный" значит, что файл уже сохранён в вашей локальной базе. К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы. Подготовленные файлы — это изменённые файлы, отмеченные для включения в следующий коммит.

Таким образом, в проектах, использующих Git, есть три части:

- каталог Git'а (Git directory),
- рабочий каталог (working directory),
- область подготовленных файлов (staging area).

Local Operations



- Каталог Git’a — это место, где Git хранит метаданные и базу данных объектов вашего проекта. Это наиболее важная часть Git’a, и именно она копируется, когда вы клонируете репозиторий с другого компьютера.
- Рабочий каталог — это извлечённая из базы копия определённой версии проекта. Эти файлы достаются из сжатой базы данных в каталоге Git’a и помещаются на диск для того, чтобы вы их просматривали и редактировали.
- Область подготовленных файлов — это обычный файл, обычно хранящийся в каталоге Git’a, который содержит информацию о том, что должно войти в следующий коммит. Иногда его называют индексом (index), но в последнее время становится стандартом называть его областью подготовленных файлов (staging area).
- Стандартный рабочий процесс с использованием Git’a выглядит примерно так:
  - Вы вносите изменения в файлы в своём рабочем каталоге.
  - Подготавливаете файлы, добавляя их слепки в область подготовленных файлов.
  - Делаете коммит, который берёт подготовленные файлы из индекса и помещает их в каталог Git’a на постоянное хранение.
- Если рабочая версия файла совпадает с версией в каталоге Git’a, файл считается зафиксированным. Если файл изменён, но добавлен в область подготовленных данных, он подготовлен. Если же файл изменился после выгрузки из БД, но не был подготовлен, то он считается изменённым. Поподробнее об этих трёх состояниях и как можно либо воспользоваться этим, либо пропустить стадию подготовки мы рассмотрим на следующей лекции.

УСТАНОВКА GIT

- Linux (через командную строку):

1. apt-get install git
2. yum install git-core

- Mac:

1. Скачать графический установщик с сайта: http://code.google.com/p/git-osx-installer
2. Через командную строку: port install git-core +svn +doc +bash_completion +gitweb

- Windows:

1. Скачать графический установщик: https://git-scm.com/downloads

ПЕРВОНАЧАЛЬНАЯ НАСТРОЙКА

- Теперь, когда Git установлен в вашей системе, хотелось бы сделать кое-какие вещи, чтобы настроить среду для работы с Git'ом под себя. Это нужно сделать только один раз — при обновлении версии Git'а настройки сохранятся. Но вы можете поменять их в любой момент, выполнив те же команды снова.
- В состав Git'а входит утилита git config, которая позволяет просматривать и устанавливать параметры, контролирующие все аспекты работы Git'а и его внешний вид. Эти параметры могут быть сохранены в трёх местах:
- Файл /etc/gitconfig содержит значения, общие для всех пользователей системы и для всех их репозиториев. Если при запуске git config указать параметр --system, то параметры будут читаться и сохраняться именно в этот файл.
- Файл ~/.gitconfig хранит настройки конкретного пользователя. Этот файл используется при указании параметра --global.
- Конфигурационный файл в каталоге Git'а (.git/config) в том репозитории, где вы находитесь в данный момент. Эти параметры действуют только для данного конкретного репозитория. Настройки на каждом следующем уровне подменяют настройки из предыдущих уровней, то есть значения в .git/config перекрывают соответствующие значения в /etc/gitconfig.
- В системах семейства Windows Git ищет файл .gitconfig в каталоге $HOME (C:\Documents and Settings\$USER или C:\Users\$USER для большинства пользователей). Кроме того Git ищет файл /etc/gitconfig, но уже относительно корневого каталога MSys, который находится там, куда вы решили установить Git, когда запускали инсталлятор.

ИМЯ ПОЛЬЗОВАТЕЛЯ

- Первое, что вам следует сделать после установки Git’а, — указать ваше имя и адрес электронной почты. Это важно, потому что каждый коммит в Git’е содержит эту информацию, и она включена в коммиты, передаваемые вами, и не может быть далее изменена:

```
$ git config --global user.name “kotik"
```

```
$ git config --global user.email kotik@example.com
```

- Повторюсь, что, если указана опция --global, то эти настройки достаточно сделать только один раз, поскольку в этом случае Git будет использовать эти данные для всего, что вы делаете в этой системе. Если для каких-то отдельных проектов вы хотите указать другое имя или электронную почту, можно выполнить эту же команду без параметра --global в каталоге с нужным проектом.

ВЫБОР РЕДАКТОРА

- Вы указали своё имя, и теперь можно выбрать текстовый редактор, который будет использоваться, если будет нужно набрать сообщение в Git'е. По умолчанию Git использует стандартный редактор вашей системы, обычно это Vi или Vim. Если вы хотите использовать другой текстовый редактор, например, Emacs, можно сделать следующее:

```
$ git config --global core.editor emacs
```

- Git умеет делать слияния при помощи kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge и opendiff, но вы можете настроить и другую утилиту.

ПРОВЕРКА НАСТРОЕК

- Если вы хотите проверить используемые настройки, можете использовать команду git config --list, чтобы показать все, которые Git найдёт:

```
$ git config –list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

- Некоторые ключи (названия) настроек могут появиться несколько раз, потому что Git читает один и тот же ключ из разных файлов (например из /etc/gitconfig и ~/.gitconfig). В этом случае Git использует последнее значение для каждого ключа.
- Также вы можете проверить значение конкретного ключа, выполнив git config {ключ}:

```
$ git config user.name

Scott Chacon
```

ОСНОВЫ GIT. СОЗДАНИЕ GIT-РЕПОЗИТОРИЯ

- Для создания Git-репозитория существуют два основных подхода. Первый подход — импорт в Git уже существующего проекта или каталога. Второй — клонирование уже существующего репозитория с сервера.

СОЗДАНИЕ РЕПОЗИТОРИЯ В СУЩЕСТВУЮЩЕМ КАТАЛОГЕ

- Если вы собираетесь начать использовать Git для существующего проекта, то вам необходимо перейти в проектный каталог и в командной строке ввести

```
$ git init
```

- Эта команда создаёт в текущем каталоге новый подкаталог с именем .git содержащий все необходимые файлы репозитория — основу Git-репозитория. На этом этапе ваш проект ещё не находится под версионным контролем. (Более подробно рассмотрим файлы содержащихся в только что созданном вами каталоге .git позже, в рамках следующих лекций)
- Если вы хотите добавить под версионный контроль существующие файлы (в отличие от пустого каталога), вам стоит проиндексировать эти файлы и осуществить первую фиксацию изменений. Осуществить это вы можете с помощью нескольких команд git add указывающих индексируемые файлы, а затем commit:

```
$ git add *.c
$ git add README
$ git commit -m 'initial project version'
```

- Мы разберём, что делают эти команды чуть позже. На данном этапе, у вас есть Git-репозиторий с добавленными файлами и начальным коммитом.

КЛОНИРОВАНИЕ СУЩЕСТВУЮЩЕГО РЕПОЗИТОРИЯ

- Если вы желаете получить копию существующего репозитория Git, например, проекта, в котором вы хотите поучаствовать, то вам нужна команда git clone. Git получает копию практически всех данных, что есть на сервере. Каждая версия каждого файла из истории проекта забирается (pulled) с сервера, когда вы выполняете git clone. Фактически, если серверный диск выйдет из строя, вы можете использовать любой из клонов на любом из клиентов, для того чтобы вернуть сервер в то состояние, в котором он находился в момент клонирования.
- Клонирование репозитория осуществляется командой git clone [url]. Например, если вы хотите клонировать библиотеку Ruby Git, известную как Grit, вы можете сделать это следующим образом:

```
$ git clone git://github.com/libgit2/rugged.git
```

- Эта команда создаёт каталог с именем rugged, инициализирует в нём каталог .git, скачивает все данные для этого репозитория и создаёт (checks out) рабочую копию последней версии. Если вы зайдёте в новый каталог rugged, вы увидите в нём проектные файлы, пригодные для работы и использования. Если вы хотите клонировать репозиторий в каталог, отличный от rugged, можно это указать в следующем параметре командной строки:

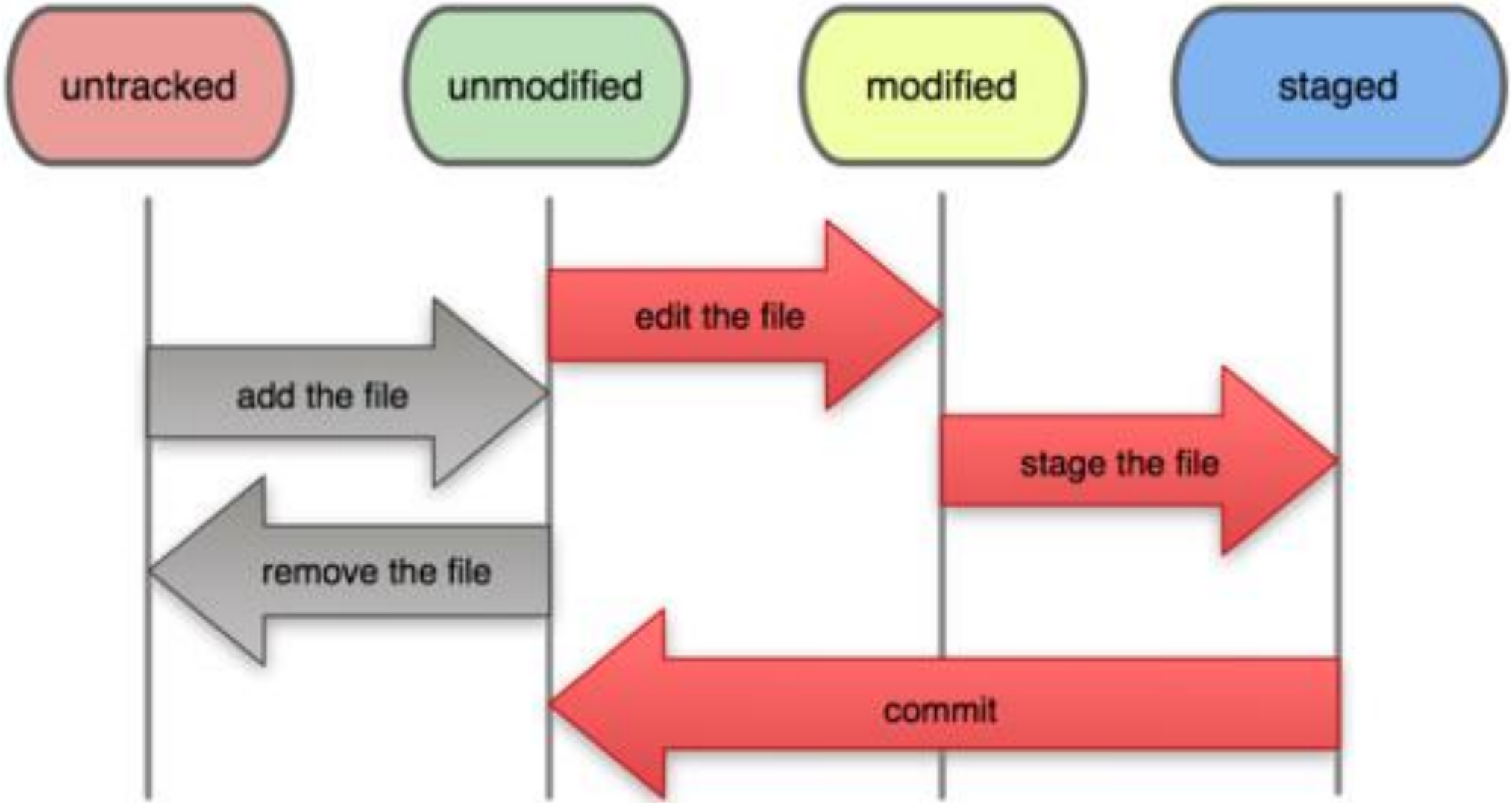
```
$ git clone git://github.com/libgit2/rugged.git myrugget
```

- Эта команда делает всё то же самое, что и предыдущая, только результирующий каталог будет назван myrugget.
- В Git’е реализовано несколько транспортных протоколов, которые вы можете использовать. В предыдущем примере использовался протокол git://, вы также можете встретить http(s):// или user@server:/path.git, использующий протокол передачи SSH..

ЗАПИСЬ ИЗМЕНЕНИЙ В РЕПОЗИТОРИЙ

- Итак, у вас имеется настоящий Git-репозиторий и рабочая копия файлов для некоторого проекта. Вам нужно делать некоторые изменения и фиксировать “снимки” состояния (snapshots) этих изменений в вашем репозитории каждый раз, когда проект достигает состояния, которое вам хотелось бы сохранить.
- Запомните, каждый файл в вашем рабочем каталоге может находиться в одном из двух состояний: под версионным контролем (отслеживаемые) и нет (неотслеживаемые). Отслеживаемые файлы — это те файлы, которые были в последнем слепке состояния проекта (snapshot); они могут быть неизменёнными, изменёнными или подготовленными к коммиту (staged). Неотслеживаемые файлы — это всё остальное, любые файлы в вашем рабочем каталоге, которые не входили в ваш последний слепок состояния и не подготовлены к коммиту. Когда вы впервые клонируете репозиторий, все файлы будут отслеживаемыми и неизменёнными, потому что вы только взяли их из хранилища (checked them out) и ничего пока не редактировали.
- Как только вы отредактируете файлы, Git будет рассматривать их как изменённые, т.к. вы изменили их с момента последнего коммита. Вы индексируете (stage) эти изменения и затем фиксируете все индексированные изменения, а затем цикл повторяется. Этот жизненный цикл изображён на рисунке.

File Status Lifecycle



ОПРЕДЕЛЕНИЕ СОСТОЯНИЯ ФАЙЛОВ

- Основной инструмент, используемый для определения, какие файлы в каком состоянии находятся — это команда git status. Если вы выполните эту команду сразу после клонирования, вы увидите что-то вроде этого:

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

- Это означает, что у вас чистый рабочий каталог, другими словами — в нём нет отслеживаемых изменённых файлов. Git также не обнаружил неотслеживаемых файлов, в противном случае они бы были перечислены здесь. И наконец, команда сообщает вам на какой ветке (branch) вы сейчас находитесь. Пока что это всегда ветка master — это ветка по умолчанию. Позже подробно поработаем с ветками и ссылками.

- Предположим, вы добавили в свой проект новый файл, простой файл README. Если этого файла раньше не было, и вы выполните git status, вы увидите свой неотслеживаемый файл вот так:

```
$ touch README
$ git status
# On branch master
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
# README nothing added to commit but untracked files present (use "git add" to track)
```

- Понять, что новый файл README неотслеживаемый можно по тому, что он находится в секции “Untracked files” в выводе команды status. Статус “неотслеживаемый файл”, по сути, означает, что Git видит файл, отсутствующий в предыдущем снимке состояния (коммите); Git не станет добавлять его в ваши коммиты, пока вы его явно об этом не попросите. Это предохранит вас от случайного добавления в репозиторий сгенерированных бинарных файлов или каких-либо других, которые вы и не думали добавлять. Мы хотели добавить README, так давайте сделаем это.

ОТСЛЕЖИВАНИЕ НОВЫХ ФАЙЛОВ

- Для того чтобы начать отслеживать (добавить под версионный контроль) новый файл, используется команда git add. Чтобы начать отслеживание файла README, вы можете выполнить следующее:

```
$ git add README
```

- Если вы снова выполните команду status, то увидите, что файл README теперь отслеживаемый и индексированный:

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

- Вы можете видеть, что файл проиндексирован по тому, что он находится в секции “Changes to be committed”. Если вы выполните коммит в этот момент, то версия файла, существовавшая на момент выполнения вами команды git add, будет добавлена в историю снимков состояния. Как вы помните, когда вы ранее выполнили git init, вы затем выполнили git add (файлы) — это было сделано для того, чтобы добавить файлы в вашем каталоге под версионный контроль. Команда git add принимает параметром путь к файлу или каталогу, если это каталог, команда рекурсивно добавляет (индексирует) все файлы в данном каталоге.

ИНДЕКСАЦИЯ ИЗМЕНЁННЫХ ФАЙЛОВ

- Давайте модифицируем файл, уже находящийся под версионным контролем. Если вы измените отслеживаемый файл benchmarks.rb и после этого снова выполните команду status, то результат будет примерно следующим:

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

- Файл benchmarks.rb находится в секции “Changes not staged for commit” — это означает, что отслеживаемый файл был изменён в рабочем каталоге, но пока не проиндексирован. Чтобы проиндексировать его, необходимо выполнить команду git add (это многофункциональная команда, она используется для добавления под версионный контроль новых файлов, для индексации изменений, а также для других целей, например для указания файлов с исправленным конфликтом слияния).

ИНДЕКСАЦИЯ ИЗМЕНЁННЫХ ФАЙЛОВ

- Выполним git add, чтобы проиндексировать benchmarks.rb, а затем снова выполним git status:

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:   benchmarks.rb
```

- Теперь оба файла проиндексированы и войдут в следующий коммит. В этот момент вы, предположим, вспомнили одно небольшое изменение, которое вы хотите сделать в benchmarks.rb до фиксации.

ИНДЕКСАЦИЯ ИЗМЕНЁННЫХ ФАЙЛОВ

- Вы открываете файл, вносите и сохраняете необходимые изменения и вроде бы готовы к коммиту. Но давайте-ка ещё раз выполним git status:

```
$ vim benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:   benchmarks.rb
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   benchmarks.rb
```

- Теперь benchmarks.rb отображается как проиндексированный и непроиндексированный одновременно. Как такое возможно? Такая ситуация наглядно демонстрирует, что Git индексирует файл в точности в том состоянии, в котором он находился, когда вы выполнили команду git add. Если вы выполните коммит сейчас, то файл benchmarks.rb попадёт в коммит в том состоянии, в котором он находился, когда вы последний раз выполняли команду git add, а не в том, в котором он находится в вашем рабочем каталоге в момент выполнения git commit.

ИНДЕКСАЦИЯ ИЗМЕНЁННЫХ ФАЙЛОВ

- Если вы изменили файл после выполнения git add, вам придётся снова выполнить git add, чтобы проиндексировать последнюю версию файла:

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:   benchmarks.rb
```

ИГНОРИРОВАНИЕ ФАЙЛОВ

- Зачастую, у вас имеется группа файлов, которые вы не только не хотите автоматически добавлять в репозиторий, но и видеть в списках неотслеживаемых. К таким файлам обычно относятся автоматически генерируемые файлы (различные логи, результаты сборки программ и т.п.). В таком случае, вы можете создать файл .gitignore с перечислением шаблонов соответствующих таким файлам. Вот пример файла .gitignore:

```
$ cat .gitignore
*.[oa]
*~
```

- Первая строка предписывает Git'у игнорировать любые файлы заканчивающиеся на .o или .a — объектные и архивные файлы, которые могут появиться во время сборки кода. Вторая строка предписывает игнорировать все файлы заканчивающиеся на тильду (~), которая используется во многих текстовых редакторах, например Emacs, для обозначения временных файлов. Вы можете также включить каталоги log, tmp или pid; автоматически создаваемую документацию; и т.д. и т.п. Хорошая практика заключается в настройке файла .gitignore до того, как начать серьёзно работать, это защитит вас от случайного добавления в репозиторий файлов, которых вы там видеть не хотите.

ИГНОРИРОВАНИЕ ФАЙЛОВ

К шаблонам в файле .gitignore применяются следующие правила:

- Пустые строки, а также строки, начинающиеся с #, игнорируются.
- Можно использовать стандартные glob шаблоны.
  - Glob-шаблоны представляют собой упрощённые регулярные выражения используемые командными интерпретаторами. Символ * соответствует 0 или более символам; последовательность [abc] — любому символу из указанных в скобках (в данном примере a, b или c); знак вопроса (?) соответствует одному символу; [0-9] соответствует любому символу из интервала (в данном случае от 0 до 9).
- Можно заканчивать шаблон символом слэша (/) для указания каталога.
- Можно инвертировать шаблон, использовав восклицательный знак (!) в качестве первого символа.

- Пример .gitignore:

```
# комментарий – эта строка игнорируется
# не обрабатывать файлы, имя которых заканчивается на .a
*.a
# НО отслеживать файл lib.a, несмотря на то, что мы игнорируем все .a файлы с помощью предыдущего правила
!lib.a
# игнорировать только файл TODO находящийся в корневом каталоге, не относится к файлам вида subdir/TODO
/TODO
# игнорировать все файлы в каталоге build/
build/
# игнорировать doc/notes.txt, но не doc/server/arch.txt
doc/*.txt
# игнорировать все .txt файлы в каталоге doc/
doc/**/*.txt
```

- Ссылка на популярные .gitignore: https://github.com/github/gitignore

ПРОСМОТР ИНДЕКСИРОВАННЫХ И НЕИНДЕКСИРОВАННЫХ ИЗМЕНЕНИЙ

- Если результат работы команды git status недостаточно информативен для вас — вам хочется знать, что конкретно поменялось, а не только какие файлы были изменены — вы можете использовать команду git diff. Позже мы рассмотрим команду git diff подробнее; вы, скорее всего, будете использовать эту команду для получения ответов на два вопроса: что вы изменили, но ещё не проиндексировали, и что вы проиндексировали и собираетесь фиксировать. Если git status отвечает на эти вопросы слишком обобщённо, то git diff показывает вам непосредственно добавленные и удалённые строки — собственно заплатку (patch).
- Допустим, вы снова изменили и проиндексировали файл README, а затем изменили файл benchmarks.rb без индексирования. Если вы выполните команду status, вы опять увидите что-то вроде:

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

- Чтобы увидеть, что же вы изменили, но пока не проиндексировали, наберите git diff без аргументов:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
           @commit.parents[0].parents[0].parents[0]
         end

+        run_code(x, 'commits 1') do
+          git.commits.size
+        end
+
         run_code(x, 'commits 2') do
           log = git.commits('master', 15)
           log.size
```

- Эта команда сравнивает содержимое вашего рабочего каталога с содержимым индекса. Результат показывает ещё не проиндексированные изменения.

- Если вы хотите посмотреть, что вы проиндексировали и что войдёт в следующий коммит, вы можете выполнить git diff --cached. (В Git'е версии 1.6.1 и выше, вы также можете использовать git diff --staged, которая легче запоминается.) Эта команда сравнивает ваши индексированные изменения с последним коммитом:

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git repository
```

- Важно отметить, что git diff сама по себе не показывает все изменения сделанные с последнего коммита — только те, что ещё не проиндексированы. Такое поведение может сбивать с толку, так как если вы проиндексируете все свои изменения, то git diff ничего не вернёт.
- Другой пример: вы проиндексировали файл benchmarks.rb и затем изменили его, вы можете использовать git diff для просмотра как индексированных изменений в этом файле, так и тех, что пока не проиндексированы:

```
$ git add benchmarks.rb
$ echo '# test line' >> benchmarks.rb
$ git status
# On branch master
#
# Changes to be committed:
#
#       modified:   benchmarks.rb
#
# Changes not staged for commit:
#
#       modified:   benchmarks.rb
```

- Теперь вы можете используя git diff посмотреть непроиндексированные изменения:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index e445e28..86b2f7c 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -127,3 +127,4 @@ end
 main()

 ##pp Grit::GitRuby.cache_client.stats
+# test line
```

- а также уже проиндексированные, используя git diff --cached:

```
$ git diff --cached
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
           @commit.parents[0].parents[0].parents[0]
         end

+        run_code(x, 'commits 1') do
+          git.commits.size
+        end
+
         run_code(x, 'commits 2') do
           log = git.commits('master', 15)
           log.size
```

ФИКСАЦИЯ ИЗМЕНЕНИЙ

- Теперь, когда ваш индекс настроен так, как вам и хотелось, вы можете зафиксировать свои изменения. Запомните, всё, что до сих пор не проиндексировано — любые файлы, созданные или изменённые вами, и для которых вы не выполнили git add после момента редактирования — не войдут в этот коммит. Они останутся изменёнными файлами на вашем диске. В нашем случае, когда вы в последний раз выполняли git status, вы видели что всё проиндексировано, и вот, вы готовы к коммиту. Простейший способ зафиксировать изменения — это набрать git commit.
- Эта команда откроет выбранный вами текстовый редактор. (Редактор устанавливается системной переменной $EDITOR — обычно это vim или emacs, хотя вы можете установить ваш любимый с помощью команды git config --global core.editor, как было показано в предыдущей лекции).
- В редакторе будет отображён следующий текст (это пример окна Vim'а):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:   benchmarks.rb
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```

- Вы можете видеть, что комментарий по умолчанию для коммита содержит закомментированный результат работы (“выхлоп”) команды git status и ещё одну пустую строку сверху. Вы можете удалить эти комментарии и набрать своё сообщение или же оставить их для напоминания о том, что вы фиксируете. (Для ещё более подробного напоминания, что же именно вы поменяли, можете передать аргумент -v в команду git commit. Это приведёт к тому, что в комментарий будет также помещена дельта/diff изменений, таким образом вы сможете точно увидеть всё, что сделано.) Когда вы выходите из редактора, Git создаёт для вас коммит с этим сообщением (удаляя комментарии и вывод diff’а).
- Есть и другой способ — вы можете набрать свой комментарий к коммиту в командной строке вместе с командой commit, указав его после параметра -m, как в следующем примере:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master]: created 463dc4f: "Fix benchmarks for speed"
 2 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 README
```

- Итак, вы создали свой первый коммит! Вы можете видеть, что коммит вывел вам немного информации о себе: на какую ветку вы выполнили коммит (master), какая контрольная сумма SHA-1 у этого коммита (463dc4f), сколько файлов было изменено, а также статистику по добавленным/удалённым строкам в этом коммите.
- Запомните, что коммит сохраняет снимок состояния вашего индекса. Всё, что вы не проиндексировали, так и торчит в рабочем каталоге как изменённое; вы можете сделать ещё один коммит, чтобы добавить эти изменения в репозиторий. Каждый раз, когда вы делаете коммит, вы сохраняете снимок состояния вашего проекта, который позже вы можете восстановить или с которым можно сравнить текущее состояние.

ИГНОРИРОВАНИЕ ИНДЕКСАЦИИ

- Несмотря на то, что индекс может быть удивительно полезным для создания коммитов именно такими, как вам и хотелось, он временами несколько сложнее, чем вам нужно в процессе работы. Если у вас есть желание пропустить этап индексирования, Git предоставляет простой способ. Добавление параметра -a в команду git commit заставляет Git автоматически индексировать каждый уже отслеживаемый на момент коммита файл, позволяя вам обойтись без git add:

```
$ git status
# On branch master
#
# Changes not staged for commit:
#
#       modified:   benchmarks.rb
#
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 files changed, 5 insertions(+), 0 deletions(-)
```

- Обратите внимание на то, что в данном случае перед коммитом вам не нужно выполнять git add для файла benchmarks.rb.

УДАЛЕНИЕ ФАЙЛОВ

- Для того чтобы удалить файл из Git'а, вам необходимо удалить его из отслеживаемых файлов (точнее, удалить его из вашего индекса) а затем выполнить коммит. Это позволяет сделать команда git rm, которая также удаляет файл из вашего рабочего каталога, так что вы в следующий раз не увидите его как "неотслеживаемый".
- Если вы просто удалите файл из своего рабочего каталога, он будет показан в секции "Changes not staged for commit" ("Изменённые но не обновлённые" — читай не проиндексированные) вывода команды git status:

```
$ rm grit.gemspec
$ git status
# On branch master
#
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#
#       deleted:    grit.gemspec
```

- Затем, если вы выполните команду git rm, удаление файла попадёт в индекс:

```
$ git rm grit.gemspec
rm 'grit.gemspec'
$ git status
# On branch master
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    grit.gemspec
#
```

- После следующего коммита файл исчезнет и больше не будет отслеживаться. Если вы изменили файл и уже проиндексировали его, вы должны использовать принудительное удаление с помощью параметра -f. Это сделано для повышения безопасности, чтобы предотвратить ошибочное удаление данных, которые ещё не были записаны в снимок состояния и которые нельзя восстановить из Git'а.

- Другая полезная штука, которую вы можете захотеть сделать — это удалить файл из индекса, оставив его при этом в рабочем каталоге. Другими словами, вы можете захотеть оставить файл на винчестере, и убрать его из-под бдительного ока Git'а. Это особенно полезно, если вы забыли добавить что-то в файл .gitignore и по ошибке проиндексировали, например, большой файл с логами, или кучу промежуточных файлов компиляции. Чтобы сделать это, используйте опцию --cached:

```
$ git rm --cached readme.txt
```

- В команду git rm можно передавать файлы, каталоги или glob-шаблоны. Это означает, что вы можете вытворять что-то вроде:

```
$ git rm log/\*.log
```

- Обратите внимание на обратный слэш (\) перед *. Он необходим из-за того, что Git использует свой собственный обработчик имён файлов вдобавок к обработчику вашего командного интерпретатора. Эта команда удаляет все файлы, которые имеют расширение .log в каталоге log/. Или же вы можете сделать вот так:

```
$ git rm \*~
```

- Эта команда удаляет все файлы, чьи имена заканчиваются на ~.

ПЕРЕМЕЩЕНИЕ ФАЙЛОВ

- В отличие от многих других систем версионного контроля, Git не отслеживает перемещение файлов явно. Когда вы переименовываете файл в Git'е, в нём не сохраняется никаких метаданных, говорящих о том, что файл был переименован. Однако, Git довольно умён в плане обнаружения перемещений постфактум — мы рассмотрим обнаружение перемещения файлов чуть позже.
- Таким образом, наличие в Git'е команды mv выглядит несколько странным. Если вам хочется переименовать файл в Git'е, вы можете сделать что-то вроде:

```
$ git mv file_from file_to
```

- и это отлично сработает. На самом деле, если вы выполните что-то вроде этого и посмотрите на статус, вы увидите, что Git считает, что произошло переименование файла:

```
$ git mv README.txt README
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    README.txt -> README
```

- Однако, это эквивалентно выполнению следующих команд:

```
$ mv README.txt README
$ git rm README.txt
$ git add README
```

- Git неявно определяет, что произошло переименование, поэтому неважно, переименуете вы файл так или используя команду mv. Единственное отличие состоит лишь в том, что mv — это одна команда вместо трёх — это функция для удобства. Важнее другое — вы можете использовать любой удобный способ, чтобы переименовать файл, и затем воспользоваться add/rm перед коммитом.

ПРОСМОТР ИСТОРИИ КОММИТОВ

- После того как вы создадите несколько коммитов, или же вы склонируете репозиторий с уже существующей историей коммитов, вы, вероятно, захотите оглянуться назад и узнать, что же происходило с этим репозиторием. Наиболее простой и в то же время мощный инструмент для этого — команда git log.
- Данные примеры используют очень простой проект, названный simplegit. Чтобы получить этот проект, выполните:

```
$ git clone git://github.com/schacon/simplegit-progit.git
```

- В результате выполнения git log в данном проекте, вы должны получить что-то вроде этого:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

- По умолчанию, без аргументов, git log выводит список коммитов созданных в данном репозитории в обратном хронологическом порядке. То есть самые последние коммиты показываются первыми. Как вы можете видеть, эта команда отображает каждый коммит вместе с его контрольной суммой SHA-1, именем и электронной почтой автора, датой создания и комментарием.
- Существует превеликое множество параметров команды git log и их комбинаций, для того чтобы показать вам именно то, что вы ищете. Здесь мы покажем вам несколько наиболее часто применяемых.
- Один из наиболее полезных параметров — это -p, который показывает дельту (разницу/diff), привнесенную каждым коммитом. Вы также можете использовать -2, что ограничит вывод до 2-х последних записей:

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,5 +5,5 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
     end

 end
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
\ No newline at end of file
```

- Этот параметр показывает ту же самую информацию плюс внесённые изменения, отображаемые непосредственно после каждого коммита. Это очень удобно для инспекций кода или для того, чтобы быстро посмотреть, что происходило в результате последовательности коммитов, добавленных коллегой.
- В некоторых ситуациях гораздо удобней просматривать внесённые изменения на уровне слов, а не на уровне строк. Чтобы получить дельту по словам вместо обычной дельты по строкам, нужно дописать после команды git log -p опцию --word-diff. Дельты на уровне слов практически бесполезны при работе над программным кодом, но они буду очень кстати при работе над длинным текстом, таким как книга или диссертация. Рассмотрим пример:

```
$ git log -U1 --word-diff
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -7,3 +7,3 @@ spec = Gem::Specification.new do |s|
    s.name      =   "simplegit"
    s.version   =   [-"0.1.0"-]{+"0.1.1"+}
    s.author    =   "Scott Chacon"
```

- Как видите, в этом выводе нет ни добавленных ни удалённых строк, как для обычного diff'а. Вместо этого изменения показаны внутри строки. Добавленное слово заключено в {+ +}, а удалённое в [- -]. Также может быть полезно сократить обычные три строки контекста в выводе команды diff до одной строки, так как контекстом в данном случае являются слова, а не строки. Сделать это можно с помощью опции -U1 как было показано в примере выше.

- С командой git log вы также можете использовать группы суммирующих параметров. Например, если вы хотите получить некоторую краткую статистику по каждому коммиту, вы можете использовать параметр --stat:

- Как видно из лога, параметр --stat выводит под каждым коммитом список изменённых файлов, количество изменённых файлов, а также количество добавленных и удалённых строк в этих файлах. Он также выводит сводную информацию в конце.

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

 Rakefile |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

 lib/simplegit.rb |    5 -----
 1 files changed, 0 insertions(+), 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit

 README           |    6 ++++++
 Rakefile         |   23 +++++++++++++++++++++++
 lib/simplegit.rb |   25 +++++++++++++++++++++++++
 3 files changed, 54 insertions(+), 0 deletions(-)
```

- Другой действительно полезный параметр — это --pretty. Он позволяет изменить формат вывода лога. Для вас доступны несколько предустановленных вариантов. Параметр oneline выводит каждый коммит в одну строку, что удобно если вы просматриваете большое количество коммитов. В дополнение к этому, параметры short, full, и fuller, практически не меняя формат вывода, позволяют выводить меньше или больше деталей соответственно:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test code
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

- Наиболее интересный параметр — это format, который позволяет вам полностью создать собственный формат вывода лога. Это особенно полезно, когда вы создаёте отчёты для автоматического разбора (парсинга) — поскольку вы явно задаёте формат и уверены в том, что он не будет изменяться при обновлениях Git'а:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 11 months ago : changed the version number
085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code
a11bef0 - Scott Chacon, 11 months ago : first commit
```

- Таблица содержит список наиболее полезных параметров формата.

Параметр	Описание выводимых данных
%H	Хеш коммита
%h	Сокращённый хеш коммита
%T	Хеш дерева
%t	Сокращённый хеш дерева
%P	Хеши родительских коммитов
%p	Сокращённые хеши родительских коммитов
%an	Имя автора
%ae	Электронная почта автора
%ad	Дата автора (формат соответствует параметру `--date=`)
%ar	Дата автора, относительная (пр. "2 мес. назад")
%cn	Имя коммитера
%ce	Электронная почта коммитера
%cd	Дата коммитера
%cr	Дата коммитера, относительная
%s	Комментарий

- Вас может заинтересовать, в чём же разница между автором и коммитером. Автор — это человек, изначально сделавший работу, тогда как коммитер — это человек, который последним применил эту работу. Так что если вы послали патч (заплатку) в проект и один из основных разработчиков применил этот патч, вы оба не будете забыты — вы как автор, а разработчик как коммитер.

- Параметры oneline и format также полезны с другим параметром команды log — --graph. Этот параметр добавляет миленький ASCII-граф, показывающий историю ветвлений и слияний. Один из таких можно увидеть для нашей копии репозитория проекта Grit:

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
*  11d191e Merge branch 'defunkt' into local
```

- Мы рассмотрели только самые простые параметры форматирования вывода для git log — их гораздо больше. Таблица ниже содержит как уже рассмотренные нами параметры, так и другие полезные параметры вместе с описанием того, как они влияют на вывод команды log.

```
Параметр            Описание
-p                  Для каждого коммита показывать дельту внесённых им изменений.
--word-diff         Показывать изменения на уровне слов.
--stat          Для каждого коммита дополнительно выводить статистику по изменённым файлам.
--shortstat         Показывать только строку changed/insertions/deletions от вывода с опцией `--stat`.
--name-only         Показывать список изменённых файлов после информации о коммите.
--name-status   Выводить список изменённых файлов вместе с информацией о добавлении/изменении/удалении.
--abbrev-commit Выводить только первые несколько символов контрольной суммы SHA-1 вместо всех 40.
--relative-date Выводить дату в относительном формате (например, "2 weeks ago") вместо полной даты.
--graph         Показывать ASCII-граф истории ветвлений и слияний рядом с выводом лога.
--pretty            Отображать коммиты в альтернативном формате. Возможные параметры: `oneline`, `short`, `full`, `fuller` и `format` (где в
ы можете указать свой собственный формат).
```

ОГРАНИЧЕНИЕ ВЫВОДА КОМАНДЫ LOG

- Кроме опций для форматирования вывода, git log имеет ряд полезных ограничительных параметров, то есть параметров, которые дают возможность отобразить часть коммитов. Вы уже видели один из таких параметров — параметр -2, который отображает только два последних коммита. На самом деле, вы можете задать -<n>, где n это количество отображаемых коммитов. На практике вам вряд ли придётся часто этим пользоваться потому, что по умолчанию Git через канал (pipe) отправляет весь вывод на pager, так что вы всегда будете видеть только одну страницу.
- А вот параметры, ограничивающие по времени, такие как --since и --until, весьма полезны. Например, следующая команда выдаёт список коммитов, сделанных за последние две недели:

```
$ git log --since=2.weeks
```

- Такая команда может работать с множеством форматов — вы можете указать точную дату (“2008-01-15”) или относительную дату, такую как “2 years 1 day 3 minutes ago”.

- Вы также можете отфильтровать список коммитов по какому-либо критерию поиска. Опция --author позволяет фильтровать по автору, опция --grep позволяет искать по ключевым словам в сообщении. (Заметим, что, если вы укажете и опцию author, и опцию grep, то будут найдены все коммиты, которые удовлетворяют первому ИЛИ второму критерию. Чтобы найти коммиты, которые удовлетворяют первому И второму критерию, следует добавить опцию --all-match.)
- Последняя действительно полезная опция-фильтр для git log — это путь. Указав имя каталога или файла, вы ограничите вывод log теми коммитами, которые вносят изменения в указанные файлы. Эта опция всегда указывается последней и обычно предваряется двумя минусами (--), чтобы отделить пути от остальных опций.

- В таблице для справки приведён список часто употребляемых опций.

Опция	Описание
-(n)	Показать последние n коммитов
--since, --after	Ограничить коммиты теми, которые сделаны после указанной даты.
--until, --before	Ограничить коммиты теми, которые сделаны до указанной даты.
--author	Показать только те коммиты, автор которых соответствует указанной строке.
--committer	Показать только те коммиты, коммитер которых соответствует указанной строке.

ОТМЕНА ИЗМЕНЕНИЙ

- На любой стадии может возникнуть необходимость что-либо отменить. Здесь мы рассмотрим несколько основных инструментов для отмены произведённых изменений. Будьте осторожны, ибо не всегда можно отменить сами отмены. Это одно из немногих мест в Git’е, где вы можете потерять свою работу если сделаете что-то неправильно.

ИЗМЕНЕНИЕ ПОСЛЕДНЕГО КОММИТА

- Одна из типичных отмен происходит тогда, когда вы делаете коммит слишком рано, забыв добавить какие-то файлы, или напутали с комментарием к коммиту. Если вам хотелось бы сделать этот коммит ещё раз, вы можете выполнить commit с опцией --amend:

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

- Эта команда берёт индекс и использует его для коммита. Если после последнего коммита не было никаких изменений (например, вы запустили приведённую команду сразу после предыдущего коммита), то состояние проекта будет абсолютно таким же и всё, что вы измените, это комментарий к коммиту.
- Появится всё тот же редактор для комментариев к коммитам, но уже с введённым комментарием к последнему коммиту. Вы можете отредактировать это сообщение так же, как обычно, и оно перепишет предыдущее.
- Для примера, если после совершения коммита вы осознали, что забыли проиндексировать изменения в файле, которые хотели добавить в этот коммит, вы можете сделать что-то подобное:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

- Все три команды вместе дают один коммит — второй коммит заменяет результат первого.

ОТМЕНА ИНДЕКСАЦИИ ФАЙЛА

- В следующих двух разделах мы продемонстрируем, как переделать изменения в индексе и в рабочем каталоге. Приятно то, что команда, используемая для определения состояния этих двух вещей, дополнительно напоминает о том, как отменить изменения в них. Приведём пример. Допустим, вы внесли изменения в два файла и хотите записать их как два отдельных коммита, но случайно набрали git add . и проиндексировали оба файла. Как теперь отменить индексацию одного из двух файлов? Команда git status напомнит вам об этом:

```
$ git add .
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#       modified:   benchmarks.rb
```

- Сразу после надписи “Changes to be committed”, написано использовать git reset HEAD <файл>… для исключения из индекса.

- Так что давайте последуем совету и отменим индексацию файла benchmarks.rb:

```
$ git reset HEAD benchmarks.rb
benchmarks.rb: locally modified
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   benchmarks.rb
```

- Эта команда немного странновата, но она работает. Файл benchmarks.rb изменён, но снова не в индексе.

ОТМЕНА ИЗМЕНЕНИЙ ФАЙЛА

- Что, если вы поняли, что не хотите оставлять изменения, внесённые в файл benchmarks.rb? Как быстро отменить изменения, вернуть то состояние, в котором он находился во время последнего коммита (или первоначального клонирования, или какого-то другого действия, после которого файл попал в рабочий каталог)? К счастью, git status говорит, как добиться и этого. В выводе для последнего примера, неиндексированная область выглядит следующим образом:

```
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   benchmarks.rb
```

- Здесь довольно ясно сказано, как отменить сделанные изменения (по крайней мере новые версии Git'а, начиная с 1.6.1, делают это; если у вас версия старее, мы настоятельно рекомендуем обновиться, чтобы получать такие подсказки и сделать свою работу удобней). Давайте сделаем то, что написано:

```
$ git checkout -- benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
```

- Как вы видите, изменения были отменены. Вы должны понимать, что это опасная команда: все сделанные вами изменения в этом файле пропали — вы просто скопировали поверх него другой файл. Никогда не используйте эту команду, если вы не полностью уверены, что этот файл вам не нужен. Если вам нужно просто сделать, чтобы он не мешался, мы рассмотрим прятание (stash) и ветвление; эти способы обычно более предпочтительны.
- Помните, что всё, что является частью коммита в Git'е, почти всегда может быть восстановлено. Даже коммиты, которые находятся на ветках, которые были удалены, и коммиты переписанные с помощью --amend могут быть восстановлены. Несмотря на это, всё, что никогда не попадало в коммит, вы скорее всего уже не увидите снова.

РАБОТА С УДАЛЁННЫМИ РЕПОЗИТОРИЯМИ

- Чтобы иметь возможность совместной работы над каким-либо Git-проектом, необходимо знать, как управлять удалёнными репозиториями. Удалённые репозитории — это модификации проекта, которые хранятся в интернете или ещё где-то в сети. Их может быть несколько, каждый из которых, как правило, доступен для вас либо только на чтение, либо на чтение и запись. Совместная работа включает в себя управление удалёнными репозиториями и помещение (push) и получение (pull) данных в и из них тогда, когда нужно обменяться результатами работы. Управление удалёнными репозиториями включает умение добавлять удалённые репозитории, удалять те из них, которые больше не действуют, умение управлять различными удалёнными ветками и определять их как отслеживаемые (tracked) или нет и прочее.

ОТОБРАЖЕНИЕ УДАЛЁННЫХ РЕПОЗИТОРИЕВ

- Чтобы просмотреть, какие удалённые серверы у вас уже настроены, следует выполнить команду git remote. Она перечисляет список имён-сокращений для всех уже указанных удалённых дескрипторов. Если вы склонировали ваш репозиторий, у вас должен отобразиться, по крайней мере, origin — это имя по умолчанию, которое Git присваивает серверу, с которого вы склонировали:

```
$ git clone git://github.com/schacon/ticgit.git
Initialized empty Git repository in /private/tmp/ticgit/.git/
remote: Counting objects: 595, done.
remote: Compressing objects: 100% (269/269), done.
remote: Total 595 (delta 255), reused 589 (delta 253)
Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.
Resolving deltas: 100% (255/255), done.
$ cd ticgit
$ git remote
origin
```

- Чтобы посмотреть, какому URL соответствует сокращённое имя в Git, можно указать команде опцию -v:

```
$ git remote -v
origin  git://github.com/schacon/ticgit.git (fetch)
origin  git://github.com/schacon/ticgit.git (push)
```

- Если у вас больше одного удалённого репозитория, команда покажет их все. Например, репозиторий Grit выглядит следующим образом.

```
$ cd grit
$ git remote -v
bakkdoor  git://github.com/bakkdoor/grit.git
cho45     git://github.com/cho45/grit.git
defunkt   git://github.com/defunkt/grit.git
koke      git://github.com/koke/grit.git
origin    git@github.com:mojombo/grit.git
```

- Это означает, что мы легко можем получить изменения от любого из этих пользователей. Но, заметьте, что origin — это единственный удалённый сервер прописанный как SSH-ссылка, поэтому он единственный, в который можно помещать свои изменения (мы рассмотрим подробнее этот момент в рамках следуюзих лекций).

ДОБАВЛЕНИЕ УДАЛЁННЫХ РЕПОЗИТОРИЕВ

- В предыдущих разделах мы упомянули и немного продемонстрировали добавление удалённых репозиториев, сейчас мы рассмотрим это более детально. Чтобы добавить новый удалённый Git-репозиторий под именем-сокращением, к которому будет проще обращаться, выполните git remote add [сокращение] [url]:

```
$ git remote
origin
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
origin  git://github.com/schacon/ticgit.git
pb      git://github.com/paulboone/ticgit.git
```

- Теперь вы можете использовать в командной строке имя pb вместо полного URL. Например, если вы хотите извлечь (fetch) всю информацию, которая есть в репозитории pb, но нет в вашем, вы можете выполнить git fetch pb:

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

- Ветка master pb теперь доступна локально как pb/master. Вы можете слить (merge) её в одну из своих веток или перейти на эту ветку, если хотите её проверить.

FETCH И PULL

- Как вы только что узнали, для получения данных из удалённых проектов, следует выполнить:

```
$ git fetch [имя удал. сервера]
```

- Данная команда связывается с указанным удалённым проектом и забирает все те данные проекта, которых у вас ещё нет. После того как вы выполнили команду, у вас должны появиться ссылки на все ветки из этого удалённого проекта. Теперь эти ветки в любой момент могут быть просмотрены или слиты.
- Когда вы клонируете репозиторий, команда clone автоматически добавляет этот удалённый репозиторий под именем origin. Таким образом, git fetch origin извлекает все наработки, отправленные (push) на этот сервер после того, как вы склонировали его (или получили изменения с помощью fetch). Важно отметить, что команда fetch забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками и не модифицирует то, над чем вы работаете в данный момент. Вам необходимо вручную слить эти данные с вашими, когда вы будете готовы

- Если у вас есть ветка, настроенная на отслеживание удалённой ветки, то вы можете использовать команду git pull.
- Она автоматически извлекает и затем сливает данные из удалённой ветки в вашу текущую ветку. Этот способ может для вас оказаться более простым или более удобным.
- К тому же по умолчанию команда git clone автоматически настраивает вашу локальную ветку master на отслеживание удалённой ветки master на сервере, с которого вы клонировали (подразумевается, что на удалённом сервере есть ветка master).
- Выполнение git pull, как правило, извлекает (fetch) данные с сервера, с которого вы изначально склонировали, и автоматически пытается слить (merge) их с кодом, над которым вы в данный момент работаете.

PUSH

- Когда вы хотите поделиться своими наработками, вам необходимо отправить (push) их в главный репозиторий. Команда для этого действия простая: git push [удал. сервер] [ветка]. Чтобы отправить вашу ветку master на сервер origin (повторимся, что клонирование, как правило, настраивает оба этих имени автоматически), вы можете выполнить следующую команду для отправки наработок на сервер:

```
$ git push origin master
```

- Эта команда срабатывает только в случае, если вы клонировали с сервера, на котором у вас есть права на запись, и если никто другой с тех пор не выполнял команду push. Если вы и кто-то ещё одновременно клонируете, затем он выполняет команду push, а затем команду push выполняете вы, то ваш push точно будет отклонён. Вам придётся сначала вытянуть (pull) их изменения и объединить с вашими. Только после этого вам будет позволено выполнить push.

ИНСПЕКЦИЯ УДАЛЁННОГО РЕПОЗИТОРИЯ

- Если хотите получить побольше информации об одном из удалённых репозиториев, вы можете использовать команду git remote show [удал. сервер]. Если вы выполните эту команду с некоторым именем, например, origin, вы получите что-то подобное:

```
$ git remote show origin
* remote origin
  URL: git://github.com/schacon/ticgit.git
  Remote branch merged with 'git pull' while on branch master
    master
  Tracked remote branches
    master
    ticgit
```

- Она выдаёт URL удалённого репозитория, а также информацию об отслеживаемых ветках. Эта команда любезно сообщает вам, что если вы, находясь на ветке master, выполните git pull, ветка master с удалённого сервера будет автоматически влита в вашу сразу после получения всех необходимых данных. Она также выдаёт список всех полученных ею ссылок.

УДАЛЕНИЕ И ПЕРЕИМЕНОВАНИЕ УДАЛЁННЫХ РЕПОЗИТОРИЕВ

- Для переименования ссылок в новых версиях Git'а можно выполнить git remote rename, это изменит сокращённое имя, используемое для удалённого репозитория. Например, если вы хотите переименовать pb в paul, вы можете сделать это следующим образом:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

- Стоит упомянуть, что это также меняет для вас имена удалённых веток. То, к чему вы обращались как pb/master, стало paul/master.
- Если по какой-то причине вы хотите удалить ссылку (вы сменили сервер или больше не используете определённое зеркало, или, возможно, контрибьютор перестал быть активным), вы можете использовать git remote rm:

```
$ git remote rm paul
$ git remote
origin
```

РАБОТА С МЕТКАМИ

- Как и большинство систем контроля версий, Git имеет возможность помечать (tag) определённые моменты в истории как важные. Как правило, этот функционал используется для отметки моментов выпуска версий (v1.0, и т.п.).

ПРОСМОТР МЕТОК

- Просмотр имеющихся меток (tag) в Git'е делается просто. Достаточно набрать git tag:

```
$ git tag
v0.1
v1.3
```

- Данная команда перечисляет метки в алфавитном порядке; порядок их появления не имеет значения.
- Для меток вы также можете осуществлять поиск по шаблону. Например, репозиторий Git'а содержит более 240 меток. Если вас интересует просмотр только выпусков 1.4.2, вы можете выполнить следующее:

```
$ git tag -l 'v1.4.2.*'
v1.4.2.1
v1.4.2.2
v1.4.2.3
v1.4.2.4
```

СОЗДАНИЕ МЕТОК

- Git использует два основных типа меток: легковесные и аннотированные. Легковесная метка — это что-то весьма похожее на ветку, которая не меняется — это просто указатель на определённый коммит. А вот аннотированные метки хранятся в базе данных Git'а как полноценные объекты. Они имеют контрольную сумму, содержат имя поставившего метку, e-mail и дату, имеют комментарий и могут быть подписаны и проверены с помощью GNU Privacy Guard (GPG). Обычно рекомендуется создавать аннотированные метки, чтобы иметь всю перечисленную информацию; но если вы хотите сделать временную метку или по какой-то причине не хотите сохранять остальную информацию, то для этого годятся и легковесные метки.

АННОТИРОВАННЫЕ МЕТКИ

- Создание аннотированной метки в Git'е выполняется легко. Самый простой способ это указать -a при выполнении команды tag:

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

- Опция -m задаёт меточное сообщение, которое будет храниться вместе с меткой. Если не указать сообщение для аннотированной метки, Git запустит редактор, чтоб вы смогли его ввести.

- Вы можете посмотреть данные метки вместе с коммитом, который был помечен, с помощью команды git show:

```
$ git show v1.4
tag v1.4
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 14:45:11 2009 -0800

my version 1.4
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

- Она показывает информацию о выставившем метку, дату отметки коммита и аннотирующее сообщение перед информацией о коммите.

ПОДПИСАННЫЕ МЕТКИ

- Вы также можете подписывать свои метки с помощью GPG, конечно, если у вас есть ключ. Всё что нужно сделать, это использовать -s вместо -a:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gee-mail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

- Если вы выполните git show на этой метке, то увидите прикреплённую к ней GPG-подпись:
- Чуть позже вы узнаете, как верифицировать метки с подписью.

```
$ git show v1.5
tag v1.5
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:22:20 2009 -0800

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.8 (Darwin)

iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN
Ki0An2JeAVUCAiJ7Ox6ZEtK+NvZAj82/
=WryJ
-----END PGP SIGNATURE-----
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

ЛЕГКОВЕСНЫЕ МЕТКИ

- Легковесная метка — это ещё один способ отметки коммитов. В сущности, это контрольная сумма коммита, сохранённая в файл — больше никакой информации не хранится. Для создания легковесной метки не передавайте опций -a, -s и -m:

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

- На этот раз при выполнении git show на этой метке вы не увидите дополнительной информации. Команда просто покажет помеченный коммит:

```
$ git show v1.4-lw
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

ВЕРИФИКАЦИЯ МЕТОК

- Для верификации подписанной метки, используйте git tag -v [имя метки]. Эта команда использует GPG для верификации подписи. Вам нужен открытый ключ автора подписи, чтобы команда работала правильно:

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg:                 aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7  4A7D C0C6 D9A4 F311 9B9A
```

- Если у вас нет открытого ключа автора подписи, вы вместо этого получите что-то подобное:

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

ВЫСТАВЛЕНИЕ МЕТОК ПОЗЖЕ

- Также возможно помечать уже пройденные коммиты. Предположим, что история коммитов выглядит следующим образом:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

- Теперь предположим, что вы забыли отметить версию проекта v1.2, которая была там, где находится коммит “updated rakefile”. Вы можете добавить метку и позже. Для отметки коммита укажите его контрольную сумму (или её часть) в конце команды:

```
$ git tag -a v1.2 -m 'version 1.2' 9fceb02
```

- Можете проверить, что коммит теперь отмечен:

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    updated rakefile
```

ОБМЕН МЕТКАМИ

- По умолчанию, команда git push не отправляет метки на удалённые серверы. Необходимо явно отправить (push) метки на общий сервер после того, как вы их создали. Это делается так же, как и выкладывание в совместное пользование удалённых веток — нужно выполнить git push origin [имя метки].

```
$ git push origin v1.5
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag]         v1.5 -> v1.5
```

- Если у вас есть много меток, которые хотелось бы отправить все за один раз, можно использовать опцию --tags для команды git push. В таком случае все ваши метки отправятся на удалённый сервер (если только их уже там нет).

```
$ git push origin --tags
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
 * [new tag]         v0.1 -> v0.1
 * [new tag]         v1.2 -> v1.2
 * [new tag]         v1.4 -> v1.4
 * [new tag]         v1.4-lw -> v1.4-lw
 * [new tag]         v1.5 -> v1.5
```

- Теперь, если кто-то склонирует (clone) или выполнит git pull из вашего репозитория, то он получит вдобавок к остальному и ваши метки.