

Лекция 3. Git. Основные сведения

1.1. Общие сведения

Git (произносится «гит») — распределённая система управления версиями. Проект был создан Линусом Торвальдсом для управления разработкой ядра Linux, первая версия выпущена 7 апреля 2005 года. На сегодняшний день его поддерживает Джунио Хамано.

Среди проектов, использующих Git — ядро Linux, Swift, Android, Drupal, Cairo, GNU Core Utilities, Mesa, Wine, Chromium, Compiz Fusion, FlightGear, jQuery, PHP, NASM, MediaWiki, DokuWiki, Qt, ряд дистрибутивов Linux.

Программа является свободной и выпущена под лицензией GNU GPL 2 (открытое свободно распространяемое ПО, в которое вы можете вносить любые модификации).

В Git клиенты не просто скачивают снимок всех файлов (состояние файлов на определённый момент времени): они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, умрёт, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом (backup — резервная копия) всех данных.

1.2. История создания

Разработка ядра Linux велась на проприетарной системе BitKeeper, которую автор, — Ларри Маквой, сам разработчик Linux, — предоставил проекту по бесплатной лицензии. Но из-за конфликта между ним и разработчиками Маквой обвинил их в нарушении соглашения и отозвал лицензию, и Линус Торвальдс взялся за новую систему: ни одна из открытых систем не позволяла тысячам программистов кооперировать свои усилия.

Преследуемые цели:

- изменение подхода CVS;
- добавление надёжности в систему;
- увеличение скорости;
- простая архитектура;
- хорошая поддержка нелинейной разработки (тысячи параллельных веток);
- полная децентрализация;
- высокая скорость работы и разумное использование дискового пространства

Начальная разработка велась меньше, чем неделю: 3 апреля 2005 года разработка началась, и уже 7 апреля код Git управлялся неготовой системой. 16 июня Linux был переведён на Git, а 25 июля Торвальдс отказался от обязанностей ведущего разработчика.

Торвальдс так саркастически отозвался о выбранном им названии git (что на английском сленге означает «мерзавец»):

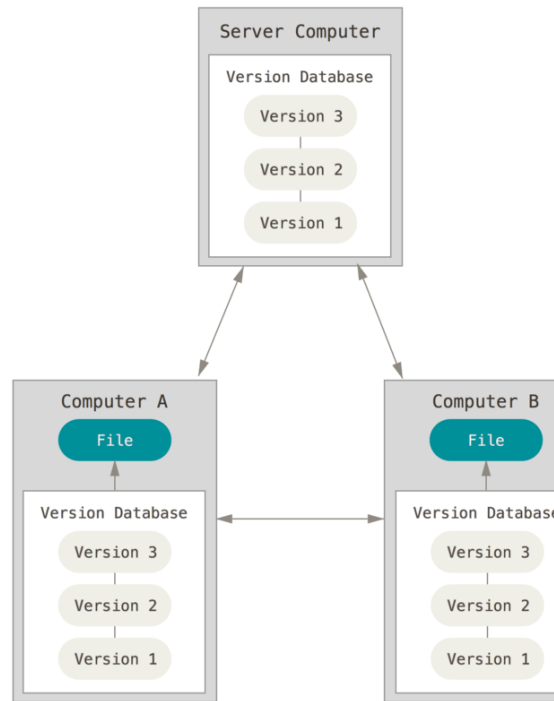
I'm an egotistical bastard, so I name Я эгоист, и поэтому называю все свои проек-

all my projects after myself. First ты в честь себя. Сначала Linux, теперь git.
Linux, now git.

1.3. Основные понятия

1.3.1. Хранение изменений в Git

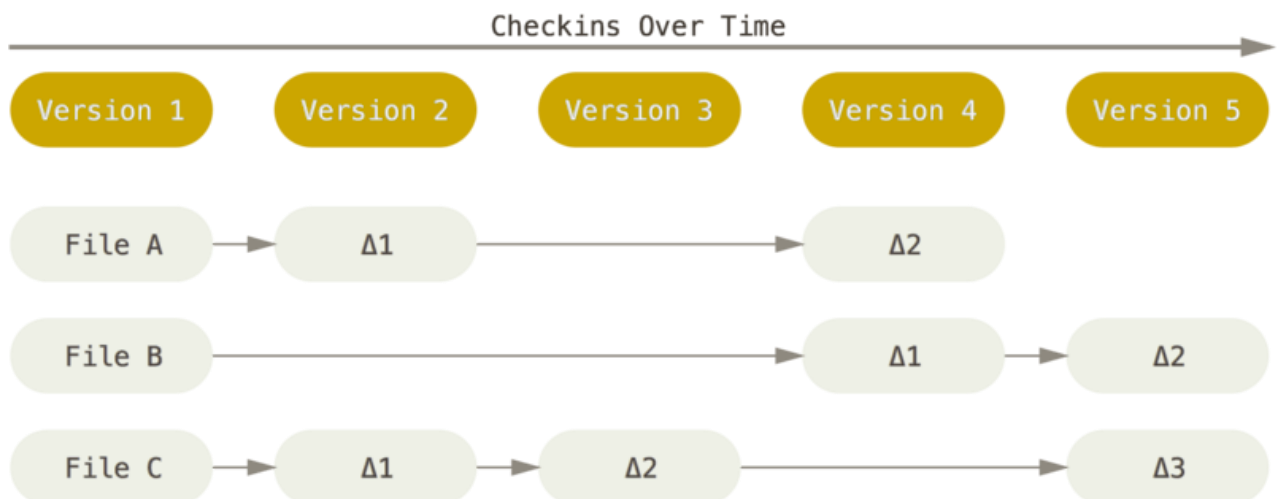
Git является, как говорилось раньше, распределённой системой контроля версий.



Распределённая СКВ

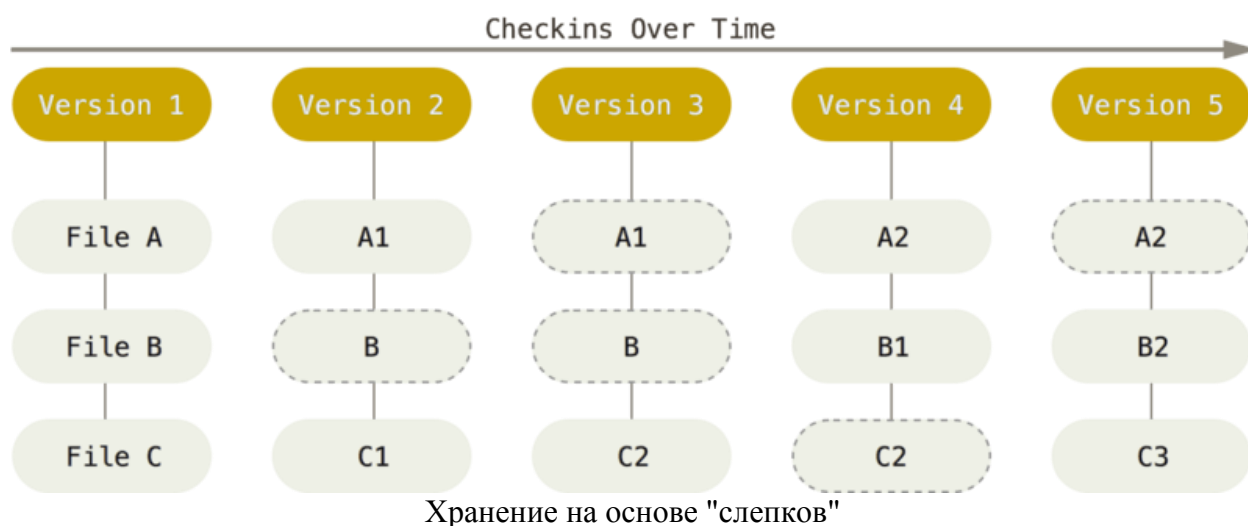
В Git используется своя собственная модель разработки, основанная на «снимках» системы.

Обычно в других СКВ (CVS, Subversion, Perforce, Bazaar и т.д.) используется подход на основе диффов (diffs - информация в виде набора файлов и изменений, сделанных в каждом файле, по времени).



Хранение на основе диффов

Подход Git'a к хранению данных больше похож на набор снимков миниатюрной файловой системы. Каждый раз, когда вы делаете коммит, то есть сохраняете состояние своего проекта в Git'е, система запоминает, как выглядит каждый файл в этот момент, и сохраняет ссылку на этот снимок. Для увеличения эффективности, если файлы не были изменены, Git не запоминает эти файлы вновь, а только создаёт ссылку на предыдущую версию идентичного файла, который уже сохранён. Git представляет свои данные как, скажем, поток снимков.



1.3.2. Почти все операции выполняются локально

Для работы большинства операций в Git достаточно локальных файлов и ресурсов — в основном, системе не нужна никакая информация с других компьютеров в вашей сети.

Для примера, чтобы посмотреть историю проекта, Git'у не нужно соединяться с сервером для её получения и отображения — система просто считывает данные напрямую из локальной базы данных. Это означает, что вы увидите историю проекта практически моментально.

Это также означает, что есть лишь небольшое количество действий, которые вы не сможете выполнить, если вы находитесь оффлайн или не имеете доступа к VPN в данный момент. Если вы в самолёте или в поезде и хотите немного поработать, вы сможете создавать коммиты без каких-либо проблем: когда будет возможность подключиться к сети, все изменения можно будет синхронизировать. Если вы ушли домой и не можете подключиться через VPN, вы всё равно сможете работать.

1.3.3. Целостность Git

В Git'е для всего вычисляется хеш-сумма, и только потом происходит сохранение. В дальнейшем обращение к сохранённым объектам происходит по этой хеш-сумме. Это

значит, что невозможно изменить содержимое файла или директории так, чтобы Git не узнал об этом. Вы не потеряете информацию во время её передачи и не получите повреждённый файл без ведома Git.

Git для вычисления хэш – сумм использует SHA-1 хеш (строка длиной в 40 шестнадцатеричных символов, вычисляемая на основе содержимого файла или структуры каталога). SHA-1 хеш выглядит примерно так:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Коммит (commit) – сохранение состояния проекта в репозиторий.

У коммитов также есть своя SHA-1, по которой, в сокращённом или полном виде можно обращаться к коммиту.

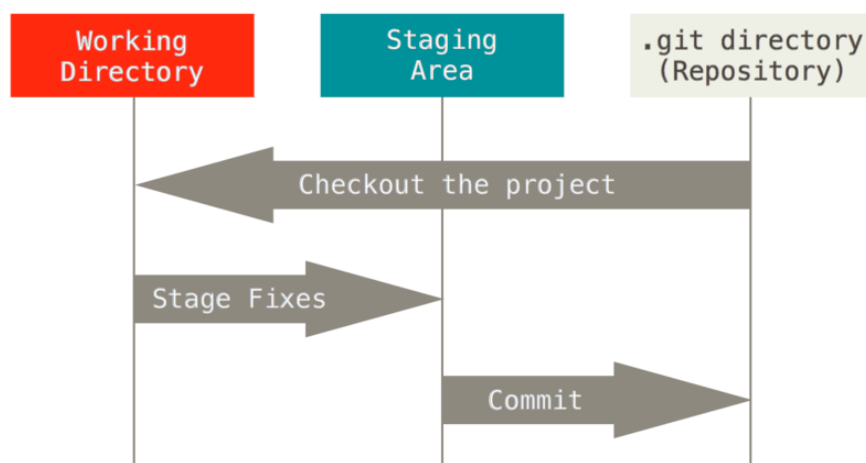
Git только добавляет данные

Когда вы производите какие-либо действия в Git, практически все из них только добавляют новые данные в базу Git. Очень сложно заставить систему удалить данные либо сделать что-то, что нельзя впоследствии отменить, особенно если вы отправили свои изменения в удалённый репозиторий.

1.3.4. Жизненный цикл файла в git (локальный)

Git имеет три основных состояния, в которых могут находиться ваши файлы: зафиксированном (committed), изменённом (modified) и подготовленном (staged). “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.

У Git существует три основные области: Git-директория (Git directory), рабочая директория (working directory) и область подготовленных файлов (staging area).



Жизненный цикл локального репозитория

Метаданные – различные служебные данные для корректной работы приложения/утилиты и т.д.

Git-директория — это то место, где Git хранит метаданные и базу объектов вашего проекта. Это самая важная часть Git, и это та часть, которая копируется при клонировании репозитория с другого компьютера.

Рабочая директория является снимком версии проекта. Файлы распаковываются из сжатой базы данных в Git-директории и располагаются на диске, для того чтобы их можно было изменять и использовать.

Область подготовленных файлов (staging area) — это файл, располагающийся в вашей Git-директории, в нём содержится информация о том, какие изменения попадут в следующий коммит. Эту область ещё называют “индекс”, или stage-область.

Базовый подход в работе с Git выглядит так:

1. Вы изменяете файлы в вашей рабочей директории.
2. Вы добавляете файлы в индекс, добавляя тем самым их снимки в область подготовленных файлов.
3. Когда вы делаете коммит, используются файлы из индекса как есть, и этот снимок сохраняется в вашу Git директорию.

1.4. Первоначальная настройка репозитория

Когда вы установили Git, необходимо настроить репозиторий, чтобы можно было различить именно Вас при фиксировании изменений.

Ваши настройки могут храниться в файлах:

- `/etc/gitconfig` - содержит значения, общие для всех пользователей системы и для всех их репозиториев;
- Файл `~/.gitconfig` или `~/.config/git/config` хранит настройки конкретного пользователя. Этот файл используется при указании параметра `-global`;
- Файл `config` в каталоге Git'a (т.е. `.git/config`) в том репозитории, который вы используете, хранит локальные настройки.

Имя пользователя и почта

Указание вашего имени и адреса электронной почты важно, потому что каждый коммит в Git'e содержит эту информацию, и она включена в коммиты, передаваемые вами, и не может быть далее изменена:

```
$ git config --global user.name "Your Name"
```

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

Проверка настроек

Для проверки используемой конфигурации, можно использовать команду **git config --list**, чтобы показать все настройки, которые найдёт Git:

```
$ git config --list
user.name=Your Name
user.email=your_name@example.com
...
```

Также вы можете проверить значение конкретного ключа, выполнив `git config <key>`:

```
$ git config user.name
Your Name
```

1.5. Полезные команды

Если вам нужна помощь при использовании Git, есть три способа открыть страницу руководства по любой команде Git:

```
$ git help <глагол>
$ git <глагол> --help
$ man git-<глагол>
```

Например, так можно открыть руководство по команде `config`

```
$ git help config
```

1.6. Жизненный цикл коммитов в Git (работа с удалённым репозиторием)

Жизненный цикл коммитов выглядит следующим образом: сначала вы создаёте или клонируете к себе существующий Git — репозиторий, затем вносите в него изменения и фиксируете их в локальной копии. Затем отправляете изменения на удалённый сервер.



1.6.1. Создание / клонирование репозитория

Для создания нового проекта в текущей директории, необходимо выполнить:

```
$git init
```

Эта команда создаёт в текущей директории новую скрытую поддиректорию с именем `.git`, содержащую все необходимые файлы репозитория — основу Git-репозитория. В дальнейшем в ней будут содержаться сведения обо всех изменениях.

Так же проект может уже существовать, и располагаться на удалённом сервере. В этом случае получить его копию можно, используя команду **git clone <url>**.

При выполнении `git clone` с сервера забирается (pulled) каждая версия каждого файла из истории проекта. 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.

Пример:

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

В Git'е реализовано несколько транспортных протоколов, которые вы можете использовать. В предыдущем примере использовался протокол `https://`, вы также можете встретить `git://` или `user@server:path/to/repo.git`, использующий протокол передачи SSH.

Протокол `git` отличается своей простотой и лёгкостью развёртки.

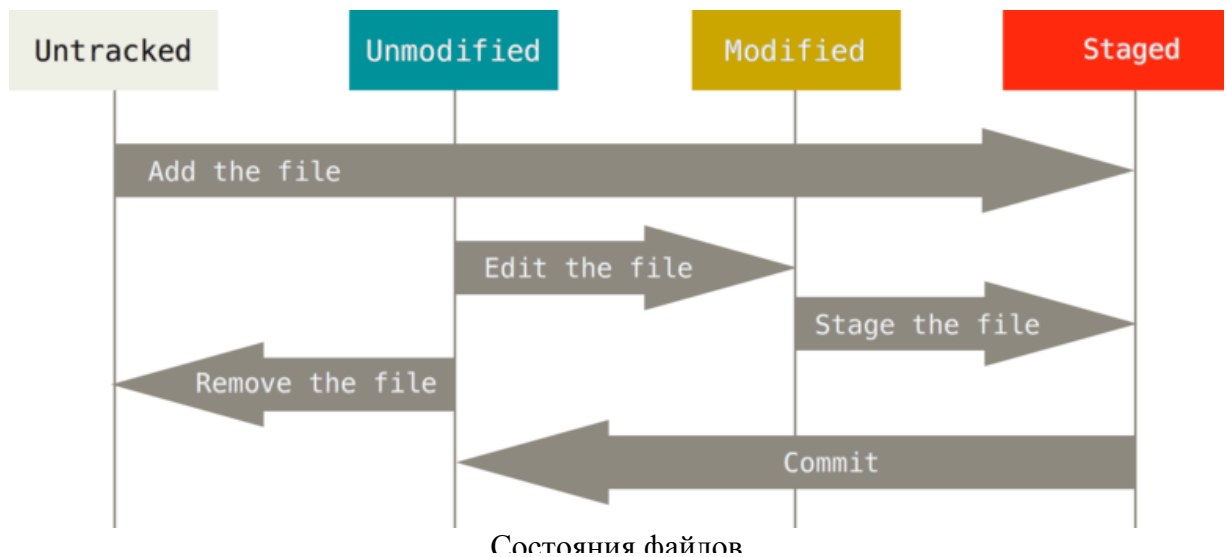
Передача по `ssh` (secure shell) позволяет производить удалённое управление ОС и передавать внутри себя практически любой другой сетевой протокол. Так же по протоколу SSH все данные подлежат шифрованию. Как следствие, для их получения необходима аутентификация.

1.6.2. Изменение файлов

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.

Важно отличие от Subversion состоит в том, что вы можете пользоваться обычными операциями удаления, переименования и перемещения файлов вместо `svn move`, `svn delete` и т.д. Можно использовать команды `git mv`, `git rm`, но это не обязательно.



Определить состояние файлов на любом из шагов можно с помощью команды **git status**. При первоначальном создании или клонировании репозитория она выдаст:

```
$ git status
On branch master
nothing to commit, working directory clean
```

У этой команды есть более короткий формат вывода в с ключом `-s` (`--short`):

```
$ git status -s

M README
MM Rakefile
A lib/git.rb
M lib/simplegit.rb
?? LICENSE.txt
```

Короткий вывод состоит из двух столбцов буквенных обозначений: первый показывает статус файла, а второй – был ли он изменён с момента добавления в отслеживаемые.

Таблица кодов представлена ниже:

Буквенное обозначение	Расшифровка	Описание
M	Modified	Файл был изменён
A	Added	Файл был добавлен

?	Untracked	Файл пока не находится в состоянии отслеживания
---	-----------	---

1.6.3. Запись изменений в репозиторий

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 индексируемый (находится в staging area):

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

    new file:   README
```

Для того, чтобы записать файлы в локальный репозиторий, применяется команда **git commit**. Её можно использовать с опцией `-m`, чтобы сразу написать комментарий к данному коммиту. Пример:

```
git commit -m "Добавил файл readme.md"
[master 463dc4f] Добавил файл readme.md
1 files changed, 15 insertions(+)
create mode 100644 README
```

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.

Каждый раз, когда вы делаете коммит, вы сохраняете снимок состояния вашего проекта, который позже вы можете восстановить или с которым можно сравнить текущее состояние.

Ещё одной полезной опцией оказывается коммит без индексации каждого файла по отдельности. 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 commit -a -m 'added new benchmarks'
```

1.6.4. Просмотр изменений

Вы можете использовать команду `git diff` для просмотра того, что изменилось в файлах с момента коммита.

Допустим, вы снова изменили и проиндексировали файл `README`, а затем изменили файл `CONTRIBUTING.md` без индексирования.

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

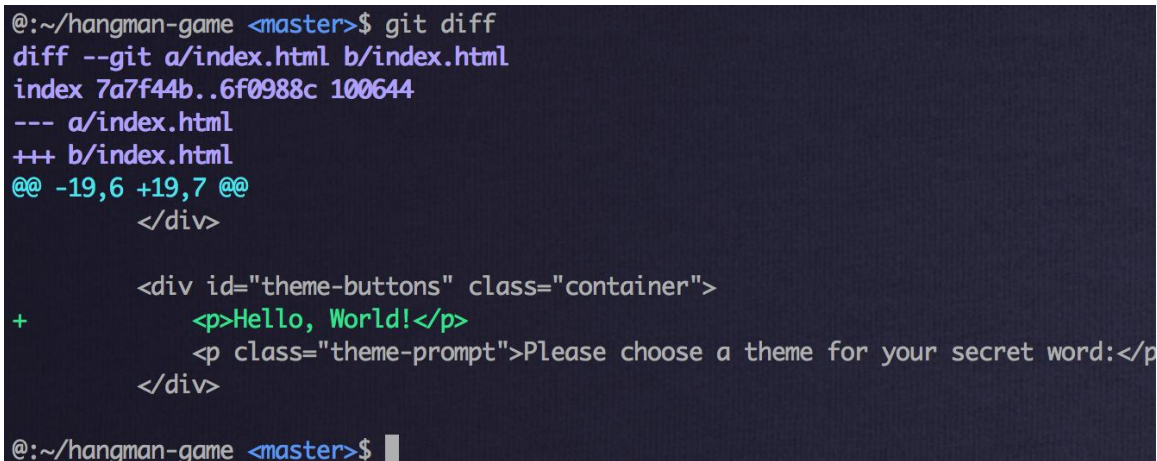
```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
```

...

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 --staged`. Эта команда сравнивает ваши индексированные изменения с последним коммитом.

Другой пример: вы можете использовать `git diff` для просмотра как индексированных изменений в этом файле, так и тех, что пока не проиндексированы.



```
@:~/hangman-game <master>$ git diff
diff --git a/index.html b/index.html
index 7a7f44b..6f0988c 100644
--- a/index.html
+++ b/index.html
@@ -19,6 +19,7 @@
</div>

+   <div id="theme-buttons" class="container">
+       <p>Hello, World!</p>
+       <p class="theme-prompt">Please choose a theme for your secret word:</p>
+   </div>

@:~/hangman-game <master>$
```

Просмотр изменений

Git Diff во внешних инструментах

Существует еще один способ просматривать эти изменения, если вы предпочитаете графический просмотр или внешнюю программу просмотра различий, вместо консоли. Выполните команду `git difftool` вместо `git diff`, таким образом вы сможете просмотреть изменения

в файле с помощью таких программ как Araxis, emerge, vimdiff и других. Выполните `git difftool --tool-help` чтобы увидеть какие из них уже установлены в вашей системе.

1.6.5. Отправка изменений на удалённый сервер

Для того, чтобы другие участники проекта увидели ваши изменения, их необходимо отправить на удалённый сервер. Это делается командой **git push**. По умолчанию синхронизируются изменения текущей ветки с веткой в удалённом репозитории.

Пример отправки изменений на удалённый сервер bitbucket.org:

```
$ git push
```

```
Password for 'https://Ksenia989@bitbucket.org':
```

```
Counting objects: 20, done.
```

```
Delta compression using up to 4 threads.
```

```
Compressing objects: 100% (20/20), done.
```

```
Writing objects: 100% (20/20), 534.20 KiB / 17.23 MiB/s, done.
```

```
Total 20 (delta 5), reused 0 (delta 0)
```

```
To https://bitbucket.org/Ksenia989/labs.git
```

```
c920410..64ac818 master -> master
```

1.7. Просмотр истории

Одним из основных и наиболее мощных инструментов для просмотра истории коммитов является команда **git log**.

Пример вывода:

```
$ 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 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` перечисляет коммиты, сделанные в репозитории в обратном к хронологическому порядку – последние коммиты находятся вверху. Из примера можно увидеть, что данная команда перечисляет коммиты с их SHA-1 контрольными суммами, именем и электронной почтой автора, датой создания и сообщением коммита.

Одним из самых полезных аргументов является `-p`, который показывает разницу, внесенную в каждый коммит. Так же вы можете использовать аргумент `-n`, который позволяет установить лимит на вывод количества коммитов (в количестве `n`).

Опция **`--pretty=format`** отображает лог в удобном для чтения виде. С параметрами этой опции вы можете ознакомиться в соответствующем разделе справки, а здесь приведём пример:

```
$ 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
* / e1193f8 support for heads with slashes in them
//
* d6016bc require time for xmlschema
, где %h — сокращённый хэш коммита, а %s — сообщение коммита.
```

1.8. Метки

Как и большинство СКВ, Git имеет возможность пометать (тегировать, tag) определённые моменты в истории как важные. Как правило, эта функциональность используется для отметки моментов выпуска версий (v1.0, и так далее).

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.

Создание меток

Git использует два основных типа меток: легковесные и аннотированные.

Легковесная метка — это 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. 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.

По умолчанию, команда `git push` не отправляет метки на удалённые серверы.

Аннотированные метки

Для создания аннотированной метки укажите `-a` при выполнении команды `tag` (сообщение добавляется с ключом `-m`):

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

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

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date: Sat May 3 20:19:12 2014 -0700
```

my version 1.4

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

changed the version number

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
```

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

changed the version number

Выставление меток позже

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

```
$ git log --pretty=oneline
```

```
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'  
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing  
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
```

Для отметки коммита укажите его контрольную сумму (или её часть) в конце команды:

```
$ git tag -a v1.2 9fceb02
```

1.9. Игнорирование файлов

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

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

Первая строка предписывает Git 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 для обозначения временных файлов.

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.

К шаблонам в файле `.gitignore` применяются следующие правила:

- Пустые строки, а также строки, начинающиеся с `#`, игнорируются (это комментарии);
- Можно использовать стандартные glob шаблоны (см. ниже);
- Можно начать шаблон символом слэша (`/`) чтобы избежать рекурсии;
- Можно заканчивать шаблон символом слэша (`/`) для указания каталога;
- Можно инвертировать шаблон, используя восклицательный знак (`!`) в качестве первого символа.

Glob-шаблоны представляют собой упрощённые регулярные выражения, используемые командными интерпретаторами:

- * соответствует 0 или более символам;
- последовательность [abc] — любому символу из указанных в скобках;
- знак вопроса (?) соответствует одному символу;
- квадратные скобки, в которые заключены символы, разделённые дефисом ([0-9]), соответствуют любому символу из интервала.

Вы также можете использовать две звёздочки, чтобы указать на вложенные директории: `a/**/z` соответствует `a/z`, `a/b/z`, `a/b/c/z`, и так далее.

Вот ещё один пример файла `.gitignore`:

```
# no .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# ignore all files in the build/ directory
build/

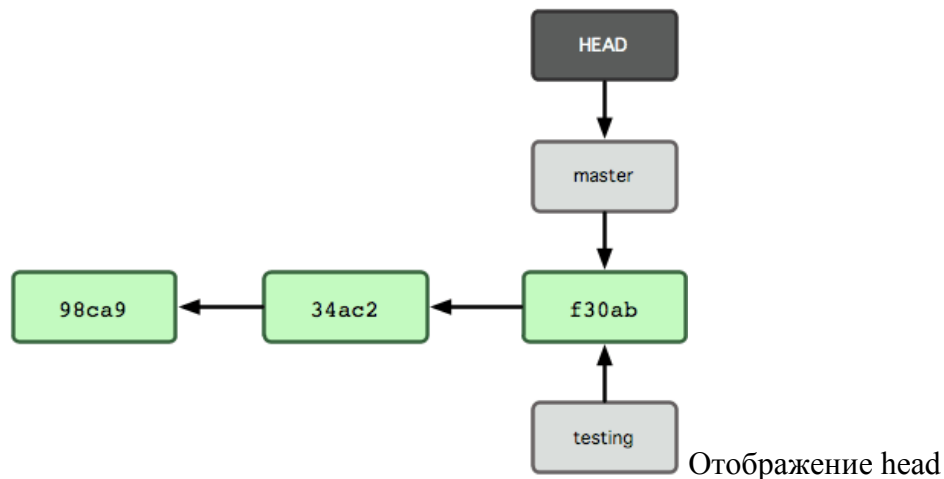
# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt
```

GitHub поддерживает довольно полный список примеров `.gitignore` файлов для множества проектов и языков (Java, c++, c, c# и т.д., различные IDE <https://github.com/github/gitignore> это может стать отправной точкой для `.gitignore` в вашем проекте.

1.10. Ссылка на последний коммит (HEAD)

Файл HEAD — это символическая ссылка на текущую ветку. Символическая ссылка отличается от обычной тем, что она содержит не сам хеш SHA-1, а указатель на другую ссылку — на последний коммит.

При выполнении `git commit` Git создаёт коммит, указывая его родителем объект, SHA-1 которого содержится в файле, на который ссылается HEAD.



Знать о HEAD полезно для таких операций, как, например, возвращение к предыдущему коммиту или откат к нему.

Например, переместиться на один коммит назад (с удалением своих изменений) можно командой:

```
git reset --hard HEAD^
```

1.11. Скрытие изменений

Команда `git stash save` позволяет спрятать все подготовленные изменения текущей ветки локального репозитория и вернуть её состояние в соответствие с последним коммитом.

В качестве параметра можно передать сообщение-описание для спрятанных изменений.

Команда `git stash list` выводит список всех сохраненных изменений.

Команда `git stash apply` применяет последние сохраненные изменения к текущей ветке, а `git stash pop` применяет и затем удаляет их.

Интересные источники для введения в Git:

- <https://learngitbranching.js.org/> - интерактивное обучение Git. Есть русский язык: темы варьируются от базовых к более сложным и интересным операциям (занимает не много времени);
- <http://rogerdudler.github.io/git-guide/> - краткая “шпаргалка” по основным командам git (с переводом на русский).

Использованные источники:

- Pro Git book
- <https://ru.wikipedia.org/wiki/Git>

- https://proselyte.net/tutorials/git/git_life_cycle/