

# 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. 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`. Если вы знакомы с другими системами контроля версий, такими как Subversion, то заметите, что команда называется `clone`, а не `checkout`. Это важное отличие — 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 (вы можете потерять часть серверных перехватчиков (server-side hooks) и т.п., но все данные, помещённые под версионный контроль, будут сохранены, подробнее см. в главе 4). 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/schacon/grit.git
```

Эта команда создаёт каталог с именем `grit`, 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 `grit`, вы увидите в нём проектные файлы, пригодные для работы и использования. Если вы хотите клонировать репозиторий в каталог, отличный от `grit`, можно это указать в следующем параметре командной строки:

```
$ git clone git://github.com/schacon/grit.git mygrit
```

Эта команда делает всё то же самое, что и предыдущая, только результирующий каталог будет назван `mygrit`.

В Git'e 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

## 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; в этой главе это не важно. В следующей главе будет подробно рассказано про ветки и ссылки. 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:

```
$ vim 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)
.
```

## 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 ваш каталог 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).

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
#
```

## 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]
*~
```

Вот ещё один пример файла `.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
```

## 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`.

```
$ 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.

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.

## 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`:

```
$ git commit
```



## Удаление файлов

Для того чтобы удалить файл из Git'a, 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)

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'a.

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 `~`.

# Пример

## 1) Создать репозиторий на GitHub:

1) <https://github.com/vovaroman/laba1>

## 2) Клонирование репозитория на свой ПК

1) `$ git clone https://github.com/vovaroman/laba1`

```
Last login: Wed Oct 17 02:30:44 on ttys001
MacBook-Pro-Vova:~ vr$ mkdir sorceres666
MacBook-Pro-Vova:~ vr$ cd sorceres666/
MacBook-Pro-Vova:sorceres666 vr$ git clone https://github.com/vovaroman/laba1
Cloning into 'laba1'...
remote: Enumerating objects: 64, done.
remote: Counting objects: 100% (64/64), done.
remote: Compressing objects: 100% (43/43), done.
remote: Total 64 (delta 16), reused 57 (delta 9), pack-reused 0
Unpacking objects: 100% (64/64), done.
MacBook-Pro-Vova:sorceres666 vr$
```

## 3) Создание программы в Visual Studio

## 4) Git add \*

## 5) Git commit -m "Init commit"

## 6) Git push

## История работы:

Commits on Oct 11, 2018		
deleted test file vovaroman committed 7 days ago	395ca58	<>
test vovaroman committed 7 days ago	036caa5	<>
test commit VMM01 committed 7 days ago	81b0e6e	<>
added file with git commands vovaroman committed 7 days ago	61e6a9c	<>
added gitignore vovaroman committed 7 days ago	962e93c	<>
converted type of d from float to int VMM01 committed 7 days ago	39dd2fa	<>
Commits on Oct 4, 2018		
Initial commit vovaroman committed 13 days ago	163dbc4	<>
added readme vovaroman committed 13 days ago	8c7bde2	<>

## 7) Работа с ветками git branch testing

## 8) Переход на новую ветку git checkout testing

## 9) git merge testing

**Overview** Yours Active Stale All branches

Search branches...

**Default branch**

master Updated 7 days ago by vovaroman **Default** Change default branch

**Your branches**

newBranch Updated 7 days ago by vovaroman 2 | 4 New pull request

testbranch Updated 7 days ago by VMM01 2 | 2 New pull request

**Active branches**

newBranch Updated 7 days ago by vovaroman 2 | 4 New pull request

testbranch Updated 7 days ago by VMM01 2 | 2 New pull request

## 10) Файл с историей изучения гита

```
MacBook-Pro-Vova:lab1 vr$ cat commands.txt
git log -p -2
git log
git commit -m "message"
git pull
git push
git rm
git add
git commit --amend // update commit
git remote -v
git remote add *link*
git fetch
git remote show origin
git tag -l 'v1.4.2.*'
git tag -a v1.4 -m 'my version 1.4'
git tag
git show v1.4
git push origin v1.5
git push origin --tags
git branch testing
git checkout testing //перейти на ветку
git checkout master
git checkout -b iss53 // создать и перейти
git merge iss53
git branch -d hotfix // delete branch
```

```
MacBook-Pro-Vova:lab1 vr$
```