# Git

## VCS

Для сложных проектов с множеством файлов следует использовать систему контроля версий (Version Control System, VCS). Она позволяет возвращать в предыдущее состояние как отдельные файлы, так и целый проект, сравнивать сделанные изменения, смотреть, кто и когда последним редактировал файл, являющийся источником проблемы, и многое другое. Наличие VCS в общем случае означает, что при сбое или потере файлов вы можете легко вернуться в рабочее состояние.

Многие разработчики в качестве метода контроля версий выбирают простое копирование файлов в другую папку с говорящими именами, например, датой создания. Этот подход крайне популярен в силу своей простоты, но при этом он ненадёжен. Можно легко забыть, в какой папке вы работаете, и случайно сделать запись не в тот файл или скопировать вовсе не те файлы, которые вы хотели.

Люди бились над этой проблемой и стали создавать различные локальные системы контроля версий. Затем возникла потребность совместной работы над одним проектом группой разработчиков. Системы стали усложняться. Появились централизованные системы контроля версий (Centralized Version Control System, CVCS). Они обладали единым сервером, содержащим все версии файлов, и набором клиентов, выгружающих файлы с сервера. На протяжении многих лет это было стандартом управления версиями. Таким является Subversion.

Такая схема имеет много преимуществ перед локальными системами контроля версий. Каждый разработчик, работающий над проектом, может узнать, чем занимаются его коллеги. Администраторы могут контролировать права допуска прочих сотрудников. Однако у этой схемы есть и серьезные недостатки. Центральный сервер может дать сбой и тогда любые взаимодействия невозможны. Вы не сможете сохранять вносимые изменения. При повреждении жесткого диска центральной базы данных и отсутствии нужных резервных копий теряется вся информация — вся история разработки проекта за исключением единичных снимков состояния, которые могут остаться на локальных компьютерах пользователей. Впрочем, та же самая проблема характерна и для локальных систем контроля версий — храня историю разработки проекта в одном месте, вы рискуете потерять все.

На смену им пришли распределённые системы контроля версий (Distributed Version Control System, DVCS). В DVCS (к примеру, Git, Mercurial, Bazaar или Darcs) клиенты не просто выгружают последние снимки файлов, они создают полную зеркальную копию репозитория. В случае выхода из строя одного из серверов его работоспособность можно восстановить, скопировав один из клиентских репозиториев. Каждая такая выгрузка сопровождается полным резервным копированием всех данных.

## Основы Git

Главным отличием Git от любой другой системы контроля версий является восприятие данных. Большинство систем хранит информацию в виде списка изменений, связанных с файлами. Эти системы рассматривают хранимые данные как набор файлов и изменений, которые вносились в эти файлы в течение их жизни.

Git не воспринимает и не хранит файлы подобным образом. Для неё данные представляют собой набор снимков состояния миниатюрной файловой системы. Каждый раз, когда вы создаете новую версию или сохраняете состояние проекта в Git, по сути, делается снимок всех файлов в конкретный момент времени и сохраняется ссылка на этот снимок. Для повышения продуктивности вместо файлов, которые не претерпели изменений, сохраняется всего лишь ссылка на их ранее сохраненные версии. Git воспринимает данные скорее как поток снимков состояния(stream of snapshots).

Для осуществления практически всех операций системе Git требуются только локальные файлы и ресурсы — в общем случае информация с других компьютеров сети не нужна. Например, для просмотра истории проекта системе Git нет нужды обращаться к серверу, получать там историю и выводить ее на экран — система просто читает все непосредственно из локальной базы данных. То есть вы видите историю проекта практически сразу же. Если вы хотите посмотреть, чем текущая версия файла отличается от версии месячной давности, Git ищет старый файл и вычисляет внесенные в него правки, вместо того чтобы просить об этой операции удаленный сервер или считывать с этого сервера старую версию файла для локального сравнения.

В системе Git для всех данных перед сохранением вычисляется контрольная сумма, по которой они впоследствии ищутся. То есть сохранить содержимое файла или папки таким образом, чтобы система Git об этом не узнала, невозможно. Эта функциональность встроена в Git на самом низком уровне и является неотъемлемым принципом ее работы. Невозможно потерять информацию или повредить файл скрытно от Git.

Механизм, которым пользуется Git для вычисления контрольных сумм, называется хешем SHA-1. Это строка из 40 символов, включающая в себя числа в шестнадцатеричной системе (0–9 и a–f) и вычисляемая на основе содержимого файла или структуры папки в Git. Хеш SHA-1 выглядит примерно так: 24b9da6552252987aa493b52f8696cd6d3b00373. Вы будете постоянно наталкиваться на эти хеш-значения, так как Git использует их повсеместно. По сути, Git сохраняет данные в базе не по именам файлов, а по хешу их содержимого.

Практически все операции в Git приводят к добавлению данных в базу. Систему сложно заставить выполнить неотменяемое действие или каким-то образом стереть данные. Как и при работе с любой VCS, незафиксированные данные можно потерять или повредить; но как только снимок состояния зафиксирован, потерять его становится крайне сложно, особенно если вы регулярно копируете базу в другой репозиторий. Это делает работу с Git крайне комфортной, так как можно экспериментировать, не опасаясь серьезно навредить проекту.

Файлы в Git могут находиться в трёх основных состояниях: зафиксированном, модифицированном и индексированном.

Зафиксированное(committed) состояние означает, что данные надежно сохранены в локальной базе. Модифицированное(modified) состояние означает, что изменения уже внесены в файл, но пока не зафиксированы в базе данных. Индексированное (staged) состояние означает, что вы пометили текущую версию модифицированного файла как предназначенную для следующей фиксации.

В результате Git-проект разбивается на три основные области: папка Git, рабочая папка и область индексирования.

Папка Git — это место, где Git хранит метаданные и объектную базу данных проекта. Это наиболее важная часть Git, которая копируется при дублировании репозитория (хранилища) с другого компьютера.

Рабочая папка — это место, куда выполняется выгрузка одной из версий проекта. Эти файлы извлекаются из сжатой базы данных в папке Git и помещаются на жесткий диск вашего компьютера, готовые к использованию или редактированию.

Область индексирования — это файл, обычно находящийся в папке Git и хранящий информацию о том, что именно войдет в следующую операцию фиксации. Иногда ее еще называют промежуточной областью.

Базовый рабочий процесс в Git выглядит так:

1. Вы редактируете файлы в рабочей папке.
2. Вы индексируете файлы, добавляя их снимки в область индексирования.
3. Вы выполняете фиксацию, беря файлы из области индексирования и сохраняя снимки в папке Git.

При наличии конкретной версии файла в папке Git файл считается зафиксированным. Если после внесения изменений в файл он был перемещен в область индексирования, он называется проиндексированным. А отредактированный после выгрузки, но не проиндексированный файл называется модифицированным.

## Командная строка

Использовать Git можно с помощью командной строки или графических интерфейсов. Для полного понимания Git рекомендуется изучить возможности командной строки, а потом по желанию можете использовать графический интерфейс.

## Установка Git

Перед тем как приступить к работе с Git, эту систему следует установить на компьютер. Для Windows есть несколько способов установки.

Один из них - это установка **GitHub для Windows**. Пакет установки включает в себя версию как с командной строкой, так и с GUI. Он хорошо работает с оболочкой Powershell и обеспечивает надежное кэширование учетных данных и работоспособные настройки CRLF. Загрузить эту версию можно с сайта <http://windows.github.com>.

После установки на Рабочем столе появятся два ярлыка - Git Shell (для работы с командами Git в Powershell) и GitHub для работы с GitHub в графическом режиме.

После установки, нужно настроить ее окружение. Эта процедура проводится только один раз; при обновлениях все настройки сохраняются. Вы можете изменить их в любой момент, снова воспользовавшись указанными командами.

Второй вариант - пакет **Git** с сайта <https://git-scm.com/>. Включает несколько утилит командной строки - **git-bash.exe** и **git-cmd.exe**. Желательно использовать первый вариант.

Для других операционных систем существуют аналоги.

После установки в контекстном меню Рабочего стола и Проводника появятся пункты **Git GUI Here** и **Git Bash Here**.

## Настройка

Для начала введём первую команду для получения версии Git.

git --version

Система Git поставляется с инструментом **git config**, позволяющим получать и устанавливать переменные конфигурации, которые задают все аспекты внешнего вида и работы Git. Эти переменные хранятся в разных местах.

В системах семейства Windows Git ищет файл **.gitconfig** в папке **$HOME** (в большинстве случаев она вложена в папку C:\Users\$USER). Кроме того, ищется файл **/etc/gitconfig**, но уже относительно корневого каталога, расположение которого вы сами выбираете при установке Git.

При установке Git первым делом следует указать имя пользователя и адрес электронной почты. Это важно, так как данную информацию Git будет включать в каждую фиксируемую вами версию, и она обязательно включается во все создаваемые вами коммиты (зафиксированные данные):

git config --global user.name "John Rambo"

git config --global user.email johnrambo@action.com

Передача параметра **--global** позволяет сделать эти настройки всего один раз, так как в этом случае Git будет использовать данную информацию для всех ваших действий в системе. Если для конкретного проекта требуется указать другое имя или адрес электронной почты, войдите в папку с проектом и выполните эту команду без параметра **--global**.

git config user.name "Test User"

git config user.email "test@test.com"

Соответственно, вы можете переключаться между разными пользователями, входя под нужным именем.

Проверить выбранные настройки позволяет команда **git config --list**, выводящая список всех обнаруженных в текущий момент параметров.

Некоторые ключи могут появляться более одного раза, так как Git считывает один и тот же ключ из разных файлов. В этом случае система использует последнее значение для каждого обнаруженного ею уникального ключа.

Можно проверить значение конкретного ключа, воспользовавшись командой **git config [ключ]**:

git config user.name

Существует три способа доступа к странице со справочной информацией по любой Git-команде:

git help [команда]

git [команда] --help

man git-[команда]

К примеру, справка по команде **config** открывается так:

git help config

## Работа с Git

Будем учиться настраивать и инициализировать репозитории, начинать и прекращать слежение за файлами, индексировать и фиксировать изменения. Заодно узнаем, как заставить Git игнорировать отдельные файлы и их группы, как быстро и просто отменить ошибочные изменения, как посмотреть историю проекта и изменения, вносившиеся в каждую версию, как отправить файл в удаленный репозиторий и извлечь его оттуда.

Есть два подхода к созданию Git-проекта. Можно взять существующий проект или папку и импортировать в Git. А можно клонировать уже существующий репозиторий с другого сервера.

### Инициализация репозитория в существующей папке

Чтобы начать слежение за существующим проектом, перейдите в папку этого проекта и введите команду:

git init

В результате в существующей папке появится еще одна папка с именем **.git** и всеми нужными вам файлами репозитория — это будет основа вашего Git-репозитория. По умолчанию вы можете не увидеть эту папку с файлами. Поэтому вручную введите полный путь к папке в Проводнике, чтобы открыть её (или включите показ скрытых папок и файлов в настройках).

Создайте в папке проекта какой-нибудь текстовый файл для опытов.

Пока контроль ваших файлов отсутствует. Чтобы начать управление версиями существующих файлов (в противовес пустому каталогу), укажите файлы, за которыми должна следить система, и выполните первую фиксацию изменений. Для этого потребуется несколько команд **git add**, добавляющих файлы, за которыми вы хотите следить, а затем команда **git commit**:

git add \*.txt

git add LICENSE

git commit -m 'первоначальная версия проекта'

У нас появился первый репозиторий Git с добавленными туда файлами и первой зафиксированной версией.

Можно указывать конкретный файл вместо \*.txt.

### Клонирование существующего репозитория

Получение копии существующего репозитория, например проекта, в котором вы хотите принять участие, выполняется командой **git clone**. Вы получаете полную копию практически всех данных с сервера. Команда **git clone** по умолчанию забирает все версии всех файлов за всю историю проекта. Это означает, что при повреждении серверного диска практически любой клон на любом из клиентов может использоваться для возвращения сервера в состояние, в котором он пребывал до момента клонирования (при этом может быть утрачена часть хуков со стороны сервера, но все данные, относящиеся к версиям, будут на месте).

Клонирование репозитория осуществляется командой **git clone [url]**. К примеру, вот как клонируется подключаемая Git-библиотека libgit2:

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

Команда создает папку с именем libgit2, инициализирует в ней папку .git, считывает из репозитория все данные и выгружает рабочую копию последней версии.

В папке libgit2 вы найдете все файлы проекта, подготовленные к работе. Репозиторий можно клонировать и в другое место, достаточно указать имя нужной папки в следующем параметре командной строки:

git clone https://github.com/libgit2/libgit2 mylibgit

Эта команда делает то же самое, что и предыдущая, но все файлы оказываются в папке mylibgit.

Для работы Git применяются разные транспортные протоколы. В предыдущем примере использовался протокол https://, но можно также встретить вариант **git://** или **user@server:path/to/repo.git**. В последнем случае применяется протокол SSH.

### Запись изменений в репозиторий

Итак, у вас есть настоящий Git-репозиторий и некая выгрузка, то есть рабочие копии файлов нашего проекта. Теперь в файлы можно вносить изменения и фиксировать их, как только проект достигнет состояния, которое вы хотели бы сохранить. Помните, что каждый файл в рабочей папке может пребывать в одном из двух состояний: отслеживаемом и неотслеживаемом. Первый случай — это файлы, входящие в последний снимок системы; они могут быть неизмененными, измененными и подготовленными к фиксации. Второй случай — это все остальные файлы рабочей папки, не вошедшие в последний снимок системы и не проиндексированные для последующей фиксации. После первого клонирования репозитория все файлы оказываются отслеживаемыми и неизмененными, потому что вы просто выгрузили их и пока не отредактировали.

Отредактированный файл Git рассматривает как измененный, ведь его состояние отличается от последнего зафиксированного. Вы индексируете эти измененные файлы, фиксируете все проиндексированные изменения, после чего цикл повторяется.

### Проверка состояния файлов

Основным инструментом определения состояния файлов является команда **git status**. Непосредственно после клонирования она дает примерно такой результат:

$ git status

On branch master

nothing to commit, working tree clean

Это означает, что в рабочей папке отсутствуют отслеживаемые и измененные файлы. Кроме того, система Git не обнаружила неотслеживаемых файлов, в противном случае они были бы перечислены в выводимых командой данных. Наконец, команда сообщает имя ветки, на которой вы в данный момент находитесь, и информирует о совпадении состояний этой ветки и аналогичной ветки на сервере. Пока мы будем работать только с веткой master, предлагаемой по умолчанию.

Если отредактировать какой-нибудь файл и снова запустить команду, то ответ будет уже другим.

Предположим, вы добавили в проект простой файл README. Если до этого момента он не существовал, команда **git status** покажет данный неотслеживаемый файл примерно так:

$ echo 'My Project' > 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 является неотслеживаемым, можно по заголовку **Untracked files** в выводимых командой **status** данных. Указанное состояние, по сути, означает, что Git видит файл, отсутствующий в предыдущем снимке состояния (в коммите); без явного указания с вашей стороны Git не будет добавлять этот файл в число коммитов. Подобный подход гарантирует, что в репозитории не сможет случайно появиться сгенерированный бинарный файл или другие файлы, которые вы не собирались туда добавлять. Однако файл README туда добавить нужно, поэтому давайте сделаем его отслеживаемым.

Чтобы начать слежение за новым файлом, воспользуйтесь командой **git add**. К примеру, для файла README она будет выглядеть так:

git add README

Теперь команда **status** покажет, что этот файл является отслеживаемым и проиндексированным:

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

Все проиндексированные файлы перечисляются под заголовком **Changes to be committed**. Если в этот момент произвести фиксацию, версия файла, существовавшая на момент выполнения команды **git add**, попадет в историю снимков состояния. Надеюсь, вы помните, что за командой **git init** следовала команда **git add (имена файлов)**, что заставило систему начать слежение за файлами в папке. Команда **git add** работает с маршрутом доступа к файлу или к папке; если указан путь к папке, команда рекурсивно добавляет все находящиеся в ней файлы.

## Индексация изменённых файлов

Внесём изменения в файл, находящийся под наблюдением. Если отредактировать ранее отслеживаемый файл и воспользоваться командой **git status**, результат будет примерно таким:

$ 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)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: cat.txt

Файл **cat.txt** появляется под заголовком **Changed but not staged for commit** — это означает, что отслеживаемый файл из рабочей папки был изменен, но пока не проиндексирован. Индексирование выполняется уже знакомой вам командой **git add**. Это многоцелевая команда, позволяющая начать слежение за новыми файлами, произвести индексирование файлов, а также пометить файлы с конфликтом слияния как разрешенные. Возможно, целесообразнее воспринимать эту команду как «добавление содержимого к следующему коммиту», а не как «добавление файла к проекту». Итак, воспользуемся командой **git add** для индексирования файла **cat.txt**, а затем запустим команду **git status**:

$ git add cat.txt

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

modified: cat.txt

Теперь оба файла проиндексированы и войдут в следующий коммит. Но предположим, что вы вспомнили про небольшое изменение, которое следует внести в файл **cat.txt** до его фиксации. Вы снова открываете файл, редактируете его, после чего все готово к фиксации. Но сначала еще раз воспользуемся командой **git status**:

Теперь утверждается, что файл одновременно является и проиндексированным, и непроиндексированным. Как такое может быть? Оказывается, Git индексирует файл в том состоянии, в котором он пребывал на момент выполнения команды **git add**. Если сейчас зафиксировать изменения, в коммит войдет версия файла, появившаяся после последнего запуска команды git add, а не версия, находившаяся в рабочей папке при запуске команды **git commit**. Редактирование файла после выполнения команды **git add** требует повторного запуска этой команды для индексирования самой последней версии файла.

Команда **git status** дает многословный результат. Существует флаг, позволяющий получить сведения в более компактной форме - **git status -s** или **git status --short**.

Рядом с именами новых неотслеживаемых файлов будет стоять знак ??, новые файлы, добавленные в область предварительной подготовки, помечаются буквой A, а модифицированные файлы — буквой M и т. п. Пометки выводятся в два столбца: в первом указывается, индексирован ли файл, а во втором — вносились ли в него изменения.

## Игнорирование файлов

Бывает так, что некоторый класс файлов вы не хотите ни автоматически добавлять в репозиторий, ни видеть в списке неотслеживаемых. В эту категорию, как правило, попадают автоматически генерируемые файлы, например журналы регистрации или файлы, генерируемые системой сборки. В подобных случаях создается файл **.gitignore** со списком соответствующих паттернов. Вот пример такого файла:

$ cat .gitignore

\*.[oa]

\*~

Первая строка заставляет Git игнорировать любые файлы, заканчивающиеся на «.o» или «.a», — объектные и архивные файлы, которые могут появляться в процессе сборки кода. Вторая строка предписывает игнорировать файлы, заканчивающиеся на тильду (~). Этим значком многие текстовые редакторы, в том числе Emacs, обозначают временные файлы. В этот список можно включить также папки log, tmp или pid, автоматически генерируемую документацию и т. п. Работу всегда имеет смысл начинать с настройки файла **.gitignore**, так как это защищает от случайного добавления в репозиторий файлов, которые там не нужны.

Правила для паттернов, которые можно вставлять в файл **.gitignore**:

* Игнорируются пустые строки и строки, начинающиеся с символа #
* Работают стандартные глобальные паттерны
* Паттерны можно заканчивать слешем (/), указывая папку
* Можно инвертировать паттерн, начав его с восклицательного знака (!)

Глобальные паттерны напоминают упрощенные регулярные выражения, используемые интерпретаторами команд. Звездочка (\*) замещает произвольное число символов, запись [abc] соответствует любому символу из указанных в скобках (в данном случае это символ a, b или c), знак вопроса (?) замещает собой один символ, а разделенные дефисом символы в квадратных скобках ([0-9]) совпадают с любым символом из указанного диапазона (в данном случае от 0 до 9). Кроме того, две звездочки применяются для обозначения вложенных папок; запись a/\*\*/z может означать a/z, a/b/z, a/b/c/z и т. д.

Вот еще один пример файла **.gitignore**:

# комментарий – это игнорируется

\*.a # пропускать файлы, заканчивающиеся на .a

!lib.a # но отслеживать файлы lib.a, несмотря на пропуск файлов на .a

/TODO # игнорировать только корневой файл TODO, а не файлы вида subdir/TODO

build/ # игнорировать все файлы в папке build/

doc/\*.txt # игнорировать doc/notes.txt, но не doc/server/arch.txt