# Начало работы

**Управление версиями**

Что такое «управление версиями» и почему оно должно нас интересовать? Система управления версиями, или контроля версий, записывает историю изменения файла или набора файлов, чтобы в будущем была возможность вернуться к конкретной версии. В книге процесс управления версиями будет рассматриваться на фрагментах кода, в реальности же эти операции можно проделывать с файлами практически всех типов.

**Локальные системы контроля версий**

Многие люди в качестве метода контроля версий выбирают простое копирование файлов в другую папку (в лучшем случае такие папки помечаются метками времени). Этот подход крайне популярен в силу своей простоты, но при этом он потрясающе ненадежен. Легко забыть, в какой папке вы работаете, и случайно сделать запись не в тот файл или скопировать вовсе не те файлы, которые вы хотели.

**Централизованные системы контроля версий**

Следующая большая проблема — необходимость сотрудничать с разработчиками других систем. Для ее решения были разработаны централизованные системы контроля версий (Centralized Version Control System, CVCS).

Такая схема имеет много преимуществ, особенно перед локальными системами контроля версий. Например, каждый человек, работающий над проектом, до определенной степени знает, чем занимаются его коллеги. Администраторы могут детально контролировать права допуска прочих сотрудников; администрировать CVCS намного проще, чем управлять локальными базами данных на каждом клиенте.

Однако есть у этой схемы и серьезные недостатки. Самым очевидным является единая точка отказа, представленная центральным сервером. Отключение этого сервера на час означает, что в течение часа любые взаимодействия невозможны.

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.

**Распределенные системы контроля версий**

Именно здесь на первый план выходят распределенные 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. Соответственно 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.

**Краткая история Git**

Ядро Linux представляет собой крайне масштабный проект ПО с открытым исходным кодом. В истории поддержки ядра Linux изменения программ долгое время передавались в виде исправлений (patches) и архивированных файлов. В 2002 году для проекта Linux стали использовать собственную систему DVCS, которая называлась BitKeeper.

В 2005 году отношения между сообществом, разрабатывавшим ядро Linux, и коммерческой фирмой, создавшей BitKeeper, были разорваны и бесплатное использование этой системы контроля версий стало невозможным, что побудило сообщество разработчиков Linux (и в частности создателя этой операционной системы Линуса Торвальдса) начать работу над собственным инструментом, взяв за основу некоторые идеи BitKeeper. Вот цели, которые ставились для новой системы:

* быстродействие;
* простое проектное решение;
* мощная поддержка нелинейной разработки (тысячи параллельных ветвей);
* полностью распределенная система;
* возможность эффективной (в плане быстродействия и объема данных) работы с большими проектами, такими как ядро Linux.

С момента своего появления в 2005 году система Git развивалась и совершенствовалась в попытках добиться простоты использования при сохранении изначальных характеристик. Она работает необыкновенно быстро, крайне эффективна для больших проектов и обладает потрясающей ветвящейся системой нелинейной разработки.

**Основы Git**

Система Git 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).

Git больше напоминает не простую систему контроля версий, а миниатюрную файловую систему с удивительно мощным инструментарием.

Для осуществления практически всех операций системе Git требуются только локальные файлы и ресурсы — в общем случае информация с других компьютеров сети не нужна. Когда вся история проекта хранится на локальном диске, кажется, что большинство операций выполняется почти мгновенно.

**Целостность Git**

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.

**Три состояния**

Файлы в Git могут находиться в трех 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.



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 — это файл, располагающийся в вашей Git-директории, в нём содержится информация о том, какие изменения попадут в следующий коммит. Эту область ещё называют “индекс”, однако называть её stage-область также общепринято.

Базовый подход в работе с Git выглядит так:

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

Если определённая версия файла есть в Git-директории, эта версия закоммичена. Если файл изменён и добавлен в индекс, значит, он будет добавлен в следующий коммит. И если файл был изменён с момента последнего распаковывания из репозитория, но не был добавлен в индекс, он считается изменённым.

**Установка Git**

Перед тем как приступить к работе с Git, эту систему следует установить на компьютер. Если она у вас уже установлена, имеет смысл обновить ее до последней версии.

**Установка в Windows**

Установка Git в операционной системе Windows также осуществляется разными способами. Официальный вариант системы доступен на сайте Git. Достаточно зайтина страницу http://git-scm.com/download/win, и загрузка начнется автоматически. Обратите внимание, что этот проект называется Git для Windows (или msysGit) и отличается от Git; дополнительная информация находится на сайте http://msysgit.github.io/.

Другой простой способ — 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. Подробно мы рассмотрим эти вещи чуть позже, пока же достаточно сказать, что они вам потребуются. Загрузить эту версию можно с сайта <http://windows.github.com>.

Настройки каждого следующего уровня переопределяют настройки предыдущего, то есть конфигурация из .git/config перекроет конфигурацию из /etc/gitconfig.

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 MSys, расположение которого вы сами выбираете при установке Git.

**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 "John Doe"

git config --global user.email johndoe@example.com

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.

**Проверка настроек**

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

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

git config user.name

**Как получить помощь?**

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

# Основы Git

**Создание репозитория в Git**

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. 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 \*.c

$ git add LICENSE

$ git commit -m 'первоначальная версия проекта'

**Клонирование существующего репозитория**

Получение копии существующего репозитория, например проекта, в котором вы хотите принять участие, выполняется командой git clone. 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

Клонирование репозитория осуществляется командой git clone [url].

$ git clone <https://github.com/libgit2/libgit2>

или

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

**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.

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



Коммит в git репозитории хранит снимок всех файлов в директории. Почти как огромная копия, только лучше

Git пытается быть лёгким и быстрым насколько это только возможно, так что он не просто слепо копирует всю директорию каждый раз, а ужимает (когда это возможно) коммит в набор изменений или «дельту» между текущей версией и предыдущей.

Также Git хранит всю историю о том, когда какой коммит был сделан. Вот почему большинство коммитов имеют предков

**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 нем нет отслеживаемых измененных файлов. Git также не обнаружил неотслеживаемых файлов, в противном случае они бы были перечислены здесь. Наконец, команда сообщает вам на какой ветке вы находитесь и сообщает вам, что она не расходится с веткой на сервере.

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

Статус ``Untracked files``, 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.

Отслеживание новых файлов

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.

Индексация изменённых файлов

Если вы измените отслеживаемый файл ``CONTRIBUTING.md`` и после этого снова выполните команду git 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)

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

modified: CONTRIBUTING.md

Файл ``CONTRIBUTING.md`` 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.

Выполним git add, чтобы проиндексировать ``CONTRIBUTING.md``, а затем снова выполним git status:

$ git add CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

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

new file: README

modified: CONTRIBUTING.md

Теперь оба файла проиндексированы и войдут в следующий коммит.

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:

\*.[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 (~).

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.

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 показывает вам непосредственно добавленные и удалённые строки — собственно заплатку (patch).

git diff 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 (--staged и --cached синонимы). Эта команда сравнивает ваши индексированные изменения с последним коммитом

Коммит изменений

Все проиндексированные изменения последней командой git add, войдут в коммит. Простейший способ зафиксировать изменения — это набрать git commit:

$ git commit

Эта команда откроет выбранный вами текстовый редактор.

Есть и другой способ — вы можете набрать свой комментарий к коммиту в командной строке вместе с командой commit указав его после параметра -m.

git commit -m "Story 182: Fix benchmarks for speed"

Игнорирование индексации

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 commit -a -m 'added new benchmarks'

Удаление файлов

Для того чтобы удалить файл из Git, 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”.

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 жёстком диске, и убрать его из-под бдительного ока Git. 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

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

git rm log/\\*.log

Перемещение файлов

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.md README

git status

On branch master

Changes to be committed:

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

renamed: README.md -> README

Однако, это эквивалентно выполнению следующих команд:

mv README.md README

git rm README.md

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

Одним из основных и наиболее мощных инструментов для этого является команда git log.

Одним из самых полезных аргументов является -p, который показывает разницу, внесенную в каждый коммит. Так же вы можете использовать аргумент -2, который позволяет установить лимит на вывод количества коммитов.

Операции отмены

Отмена может потребоваться, если вы сделали коммит слишком рано, например, забыв добавить какие-то файлы или комментарий к коммиту. Если вы хотите переделать коммит, можно запустить commit с параметром --amend (дополнить):

$ git commit --amend

Например, если вы фиксируете изменения, и понимаете, что забыли проиндексировать изменения в файле, который хотели включить в коммит, можно сделать примерно так:

$ git commit -m 'initial commit'

$ git add forgotten\_file

$ git commit --amend

В итоге получится единый коммит — второй коммит заменит результаты первого.

Отмена подготовки файла

git reset

отменяет последнюю команду git add

Основы Git - Работа с удалёнными репозиториями

Работа с удалёнными репозиториями

Просмотр удалённых репозиториев

Для того, чтобы просмотреть список настроенных удалённых репозиториев, вы можете запустить команду git remote. Она выведет названия доступных удалённых репозиториев. Если вы клонировали репозиторий, то увидите как минимум ``origin`` — имя по умолчанию для исходного репозитория:

Вы можете также указать ключ -v, чтобы просмотреть адреса для чтения и записи, привязанные к репозиторию:

git remote -v

Добавление удалённых репозиториев

Для того, чтобы добавить удалённый репозиторий и присвоить ему имя (shortname), просто выполните команду git remote add [shortname] [url]:

git remote

origin

git remote add pb https://github.com/paulboone/ticgit

git remote -v

origin https://github.com/schacon/ticgit (fetch)

origin https://github.com/schacon/ticgit (push)

pb https://github.com/paulboone/ticgit (fetch)

pb https://github.com/paulboone/ticgit (push)

Теперь вместо указания полного пути вы можете использовать pb. Например, если вы хотите получить изменения, которые есть у Пола, но нету у вас, вы можете выполнить команду git fetch pb.

Получение изменений из удалённого репозитория - Fetch и Pull

git fetch [remote-name]

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.

Отправка изменений в удаленный репозиторий (Push)

Когда вы хотите поделиться своими наработками, вам необходимо отправить (push) их в главный репозиторий. Команда для этого действия простая: git push [remote-name] [branch-name].

Просмотр удаленного репозитория

Если хотите получить побольше информации об одном из удалённых репозиториев, вы можете использовать команду git remote show [remote-name].

Удаление и переименование удалённых репозиториев

Для переименования ссылок в новых версиях Git’а можно вылолнить git remote rename, это изменит сокращённое имя, используемое для удалённого репозитория. Например, если вы хотите переименовать pb в paul, вы можете это сделать при помощи git remote rename:

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, теперь 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 СКВ, Git имеет возможность помечать (tag) определённые моменты в истории как важные.

Просмотр меток

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

git tag

оздание 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'

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

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

Выставление меток позже

Также возможно помечать уже пройденные коммиты.

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

$ git tag -a v1.2 9fceb02

Можете проверить, что коммит теперь отмечен:

git tag

git show v1.2

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

**Создание ветки**

git branch [название ветки]

переключение на ветку

git checkout [название ветки]

**Слияние веток**

Сливаются ветки текущая с веткой упомянутой в merge

git merge [название ветки]