**Актуальность**

В последние годы популярность git демонстрирует взрывной рост. Эта система контроля версий используется различными проектами с открытым исходным кодом.

Новичков часто пугает большое количество замысловатых команд и сложных аргументов. Но для начала все они и не нужны. Можно начать с изучения наиболее часто используемых команд, и после этого постепенно расширять свои знания.

**Основы**

Git — это набор консольных утилит, которые отслеживают и фиксируют изменения в файлах (чаще всего речь идет об исходном коде программ). С его помощью вы можете откатиться на более старую версию вашего проекта, сравнивать, анализировать, сливать изменения и многое другое. Этот процесс называется контролем версий. Существуют различные системы для контроля версий. Вы, возможно, о них слышали: SVN, Mercurial, Perforce, CVS, Bitkeeper и другие.

Git является распределенным, то есть не зависит от одного центрального сервера, на котором хранятся файлы. Вместо этого он работает полностью локально, сохраняя данные в папках на жестком диске, которые называются репозиторием. Тем не менее, вы можете хранить копию репозитория онлайн, это сильно облегчает работу над одним проектом для нескольких людей. Для этого используются сайты вроде github и bitbucket.

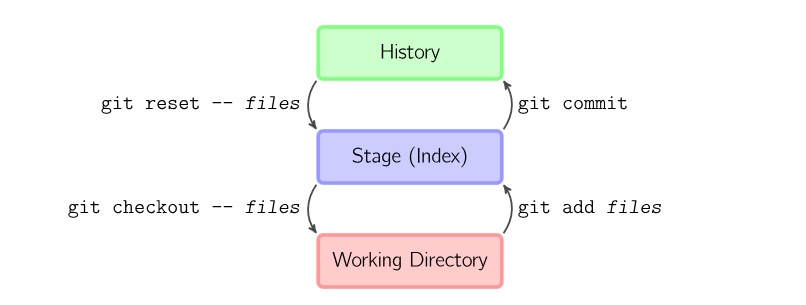
Работать с git можно непосредственно из консоли cmd, git-консоли bash, через "Декстопное" приложение.

Разберем работу с консолью.

**Основные команды.**

Следующие четыре команды предназначены для копирования файлов между рабочей директорией, сценой, также известной как «индекс», и историей, представленной в форме коммитов.

* git add *файлы* копирует *файлы* в их текущем состоянии на сцену.
* git commit сохраняет снимок сцены в виде коммита.
* git reset -- *файлы* восстанавливает файлы на сцене, а именно копирует *файлы* из последнего коммита на сцену. Используйте эту команду для отмены изменений, внесённых командой git add *файлы*. Вы также можете выполнитьgit reset, чтобы восстановить все файлы на сцене.
* git checkout -- *файлы* копирует *файлы* со сцены в рабочую директорию. Эту команду удобно использовать, чтобы сбросить нежелательные изменения в рабочей директории.



**Настройка**

Если вы никогда ранее не использовали git, для начала вам необходимо осуществить установку. Выполните следующие команды, чтобы git узнал ваше имя и электронную почту. Если git уже установлен, можете переходить к разделу окончания строк.

|  |  |
| --- | --- |
| 1  2 | git config --global user.name "My Name"  git config --global user.email myEmail@example.com |

Теперь каждое действие будет отмечено именем и почтой. Таким образом, пользователи всегда будут в курсе, кто отвечает за какие изменения — это вносит порядок.

**Создание нового репозитория**

Как мы отметили ранее, git хранит свои файлы и историю прямо в папке проекта. Чтобы создать новый репозиторий, нам нужно открыть терминал, зайти в папку нашего проекта и выполнить команду init. Это включит приложение в этой конкретной папке и создаст скрытую директорию .git, где будет храниться история репозитория и настройки.  
Создайте на рабочем столе папку под названием git\_exercise. Для этого в окне терминала введите:

|  |  |
| --- | --- |
| 1  2  3 | $ mkdir Desktop/git\_exercise/  $ cd Desktop/git\_exercise/  $ git init |

Командная строка должна вернуть что-то вроде:

|  |  |
| --- | --- |
| 1 | Initialized empty Git repository in /home/user/Desktop/git\_exercise/.git/ |

Это значит, что наш репозиторий был успешно создан, но пока что пуст. Теперь создайте текстовый файл под названием hello.txt и сохраните его в директории git\_exercise.

# Определение состояния

status — это еще одна важнейшая команда, которая показывает информацию о текущем состоянии репозитория: актуальна ли информация на нём, нет ли чего-то нового, что поменялось, и так далее. Запуск git status на нашем свежесозданном репозитории должен выдать:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | $ git status  On branch master  Initial commit  Untracked files:  (use "git add ..." to include in what will be committed)  hello.txt |

Сообщение говорит о том, что файл hello.txt неотслеживаемый. Это значит, что файл новый и система еще не знает, нужно ли следить за изменениями в файле или его можно просто игнорировать. Для того, чтобы начать отслеживать новый файл, нужно его специальным образом объявить.

# Подготовка файлов

В git есть концепция области подготовленных файлов. Можно представить ее как холст, на который наносят изменения, которые нужны в коммите. Сперва он пустой, но затем мы добавляем на него файлы (или части файлов, или даже одиночные строчки) командой add и, наконец, коммитим все нужное в репозиторий (создаем слепок нужного нам состояния) командой commit.  
В нашем случае у нас только один файл, так что добавим его:

|  |  |
| --- | --- |
| 1 | $ git add hello.txt |

Если нам нужно добавить все, что находится в директории, мы можем использовать

|  |  |
| --- | --- |
| 1 | $ git add -A |

Проверим статус снова, на этот раз мы должны получить другой ответ:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | $ git status  On branch master  Initial commit  Changes to be committed:  (use "git rm --cached ..." to unstage)  new file: hello.txt |

Файл готов к коммиту. Сообщение о состоянии также говорит нам о том, какие изменения относительно файла были проведены в области подготовки — в данном случае это новый файл, но файлы могут быть модифицированы или удалены.

# Коммит(фиксация изменений)

Коммит представляет собой состояние репозитория в определенный момент времени. Это похоже на снапшот, к которому мы можем вернуться и увидеть состояние объектов на определенный момент времени.  
Чтобы зафиксировать изменения, нам нужно хотя бы одно изменение в области подготовки (мы только что создали его при помощи git add), после которого мы может коммитить:

|  |  |
| --- | --- |
| 1 | $ git commit -m "Initial commit." |

Эта команда создаст новый коммит со всеми изменениями из области подготовки (добавление файла hello.txt). Ключ -m и сообщение «Initial commit.» — это созданное пользователем описание всех изменений, включенных в коммит. Считается хорошей практикой делать коммиты часто и всегда писать содержательные комментарии.

# Удаленные репозитории

Сейчас наш коммит является локальным — существует только в директории .git на нашей файловой системе. Несмотря на то, что сам по себе локальный репозиторий полезен, в большинстве случаев мы хотим поделиться нашей работой или доставить код на сервер, где он будет выполняться.

Чтобы загрузить что-нибудь в удаленный репозиторий, сначала нужно к нему подключиться. Cоздать свой репозиторий можно в GitHub, BitBucket или любом другом сервисе. Регистрация и установка может занять время, но все подобные сервисы предоставляют хорошую документацию.  
Чтобы связать наш локальный репозиторий с репозиторием на GitHub, нужно выполнить следующую команду в терминале. Обратите внимание, что нужно обязательно изменить URL репозитория на свой.

|  |  |
| --- | --- |
| 1  2 | # This is only an example. Replace the URL with your own repository address.  $ git **remote** add origin https://github.com/tutorialzine/awesome-project.git |

Проект может иметь несколько удаленных репозиториев одновременно. Чтобы их различать, мы дадим им разные имена. Обычно главный репозиторий называется origin.

# Отправка изменений на сервер

Сейчас самое время переслать наш локальный коммит на сервер. Этот процесс происходит каждый раз, когда мы хотим обновить данные в удаленном репозитории.  
Команда, предназначенная для этого — push. Она принимает два параметра: имя удаленного репозитория (мы назвали наш origin) и ветку, в которую необходимо внести изменения (master — это ветка по умолчанию для всех репозиториев).

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | $ git **push** origin master  Counting objects: 3, done.  Writing objects: 100% (3/3), 212 bytes | 0 bytes/s, done.  Total 3 (delta 0), reused 0 (delta 0)  To https://github.com/tutorialzine/awesome-project.git  \* [new branch] master -> master |

В зависимости от сервиса, который вы используете, вам может потребоваться аутентифицироваться, чтобы изменения отправились. Если все сделано правильно, то когда вы посмотрите в удаленный репозиторий при помощи браузера, вы увидете файл hello.txt

## Клонирование репозитория

Сейчас другие пользователи GitHub могут просматривать ваш репозиторий. Они могут скачать из него данные и получить полностью работоспособную копию вашего проекта при помощи команды clone.

|  |  |
| --- | --- |
| 1 | $ git **clone** https://github.com/tutorialzine/awesome-project.git |

Новый локальный репозиторий создается автоматически с GitHub в качестве удаленного репозитория.

## Запрос изменений с сервера

Если вы сделали изменения в вашем репозитории, другие пользователи могут скачать изменения при помощи команды pull.

|  |  |
| --- | --- |
| 1  2  3  4 | $ git **pull** origin master  From https://github.com/tutorialzine/awesome-project  \* branch master -> FETCH\_HEAD  Already up-to-date. |

Так как новых коммитов с тех пор, как мы склонировали себе проект, не было, никаких изменений доступных для скачивания нет.

# Ветвление

Во время разработки новой функциональности считается хорошей практикой работать с копией оригинального проекта, которую называют веткой. Ветви имеют свою собственную историю и изолированные друг от друга изменения до тех пор, пока вы не решаете слить изменения вместе. Это происходит по набору причин:

* Уже рабочая, стабильная версия кода сохраняется.
* Различные новые функции могут разрабатываться параллельно разными программистами.
* Разработчики могут работать с собственными ветками без риска, что кодовая база поменяется из-за чужих изменений.
* В случае сомнений, различные реализации одной и той же идеи могут быть разработаны в разных ветках и затем сравниваться.

## Создание новой ветки

Основная ветка в каждом репозитории называется master. Чтобы создать еще одну ветку, используем команду branch <name>

|  |  |
| --- | --- |
| 1 | $ git branch amazing\_new\_feature |

Это создаст новую ветку, пока что точную копию ветки master.

## Переключение между ветками

Сейчас, если мы запустим branch, мы увидим две доступные опции:

|  |  |
| --- | --- |
| 1  2  3 | $ git **branch**  amazing\_new\_feature  \* master |

master — это активная ветка, она помечена звездочкой. Но мы хотим работать с нашей “новой потрясающей фичей”, так что нам понадобится переключиться на другую ветку. Для этого воспользуемся командой checkout, она принимает один параметр — имя ветки, на которую необходимо переключиться.

|  |  |
| --- | --- |
| 1 | $ git **checkout** amazing\_new\_feature |

## Слияние веток

Наша “ новая фича” будет еще одним текстовым файлом под названием feature.txt. Мы создадим его, добавим и закоммитим:

|  |  |
| --- | --- |
| 1  2 | $ git **add** feature.txt  $ git **commit** -m "New feature complete.” |

Изменения завершены, теперь мы можем переключиться обратно на ветку master.

|  |  |
| --- | --- |
| 1 | $ git checkout master |

Теперь, если мы откроем наш проект в файловом менеджере, мы не увидим файла feature.txt, потому что мы переключились обратно на ветку master, в которой такого файла не существует. Чтобы он появился, нужно воспользоваться merge для объединения веток (применения изменений из ветки amazing\_new\_feature к основной версии проекта).

|  |  |
| --- | --- |
| 1 | $ git **merge** amazing\_new\_feature |

Теперь ветка master актуальна. Ветка amazing\_new\_feature больше не нужна, и ее можно удалить.

|  |  |
| --- | --- |
| 1 | $ git branch -d awesome\_new\_feature |

## Отслеживание изменений, сделанных в коммитах

У каждого коммита есть свой уникальный идентификатор в виде строки цифр и букв. Чтобы просмотреть список всех коммитов и их идентификаторов, можно использовать команду log:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | $ git **log**  commit ba25c0ff30e1b2f0259157b42b9f8f5d174d80d7  Author: Tutorialzine  Date: Mon May 30 17:15:28 2016 +0300  New feature complete  commit b10cc1238e355c02a044ef9f9860811ff605c9b4  Author: Tutorialzine  Date: Mon May 30 16:30:04 2016 +0300  Added content to hello.txt  commit 09bd8cc171d7084e78e4d118a2346b7487dca059  Author: Tutorialzine  Date: Sat May 28 17:52:14 2016 +0300  Initial commit |

Как вы можете заметить, идентификаторы довольно длинные, но для работы с ними не обязательно копировать их целиком — первых нескольких символов будет вполне достаточно. Чтобы посмотреть, что нового появилось в коммите, мы можем воспользоваться командой show [commit]

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | $ git **show** b10cc123  commit b10cc1238e355c02a044ef9f9860811ff605c9b4  Author: Tutorialzine  Date: Mon May 30 16:30:04 2016 +0300  Added content to hello.txt  diff --git a/hello.txt b/hello.txt  index e69de29..b546a21 100644  --- a/hello.txt  +++ b/hello.txt  @@ -0,0 +1 @@  +Nice weather today, isn't it? |

Чтобы увидеть разницу между двумя коммитами, используется команда diff (с указанием промежутка между коммитами):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | $ git **diff** 09bd8cc..ba25c0ff  diff --git a/feature.txt b/feature.txt  new file mode 100644  index 0000000..e69de29  diff --git a/hello.txt b/hello.txt  index e69de29..b546a21 100644  --- a/hello.txt  +++ b/hello.txt  @@ -0,0 +1 @@  +Nice weather today, isn't it? |

Мы сравнили первый коммит с последним, чтобы увидеть все изменения, которые были когда-либо сделаны. Обычно проще использовать git difftool, так как эта команда запускает графический клиент, в котором наглядно сопоставляет все изменения.

## Возвращение файла к предыдущему состоянию

Гит позволяет вернуть выбранный файл к состоянию на момент определенного коммита. Это делается уже знакомой нам командой checkout, которую мы ранее использовали для переключения между ветками. Но она также может быть использована для переключения между коммитами (это довольно распространенная ситуация для Гита — использование одной команды для различных, на первый взгляд, слабо связанных задач).  
В следующем примере мы возьмем файл hello.txt и откатим все изменения, совершенные над ним к первому коммиту. Чтобы сделать это, мы подставим в команду идентификатор нужного коммита, а также путь до файла:

|  |  |
| --- | --- |
| 1 | $ git checkout 09bd8cc1 hello.txt |

## Исправление коммита

Если вы опечатались в комментарии или забыли добавить файл и заметили это сразу после того, как закоммитили изменения, вы легко можете это поправить при помощи commit —amend. Эта команда добавит все из последнего коммита в область подготовленных файлов и попытается сделать новый коммит. Это дает вам возможность поправить комментарий или добавить недостающие файлы в область подготовленных файлов.  
Для более сложных исправлений, например, не в последнем коммите или если вы успели отправить изменения на сервер, нужно использовать revert. Эта команда создаст коммит, отменяющий изменения, совершенные в коммите с заданным идентификатором.  
Самый последний коммит может быть доступен по алиасу HEAD:

|  |  |
| --- | --- |
| 1 | $ git revert HEAD |

Для остальных будем использовать идентификаторы:

|  |  |
| --- | --- |
| 1 | $ git revert b10cc123 |

При отмене старых коммитов нужно быть готовым к тому, что возникнут конфликты. Такое случается, если файл был изменен еще одним, более новым коммитом. И теперь git не может найти строчки, состояние которых нужно откатить, так как они больше не существуют.

## Разрешение конфликтов при слиянии

Помимо сценария, описанного в предыдущем пункте, конфликты регулярно возникают при слиянии ветвей или при отправке чужого кода. Иногда конфликты исправляются автоматически, но обычно с этим приходится разбираться вручную — решать, какой код остается, а какой нужно удалить.  
Давайте посмотрим на примеры, где мы попытаемся слить две ветки под названием john\_branch и tim\_branch. И Тим, и Джон правят один и тот же файл: функцию, которая отображает элементы массива.  
Джон использует цикл:

|  |  |
| --- | --- |
| 1  2  3  4 | // Use a for loop to console.log contents.  for(var i=0; i<arr.length; i++) {  console.log(arr[i]);  } |

Тим предпочитает forEach:

|  |  |
| --- | --- |
| 1  2  3  4 | // Use forEach to console.log contents.  arr.forEach(function(item) {  console.log(item);  }); |

Они оба коммитят свой код в соответствующую ветку. Теперь, если они попытаются слить две ветки, они получат сообщение об ошибке:

|  |  |
| --- | --- |
| 1  2  3  4 | $ git merge tim\_branch  Auto-merging print\_array.js  CONFLICT (content): Merge conflict in print\_array.js  Automatic merge failed; fix conflicts and then commit the result. |

Система не смогла разрешить конфликт автоматически, значит, это придется сделать разработчикам. Приложение отметило строки, содержащие конфликт:

|  |  |
| --- | --- |
| 1 | <<<<<<< HEAD // Use a for loop to console.log contents. for(var i=0; i<arr.length; i++) { console.log(arr[i]); } ======= // Use forEach to console.log contents. arr.forEach(function(item) { console.log(item); }); >>>>>>> Tim's commit. |

Над разделителем ======= мы видим последний (HEAD) коммит, а под ним — конфликтующий. Таким образом, мы можем увидеть, чем они отличаются и решать, какая версия лучше. Или вовсе написать новую. В этой ситуации мы так и поступим, перепишем все, удалив разделители, и дадим git понять, что закончили.

|  |  |
| --- | --- |
| 1  2  3 | // Not using for loop or forEach.  // Use Array.toString() to console.log contents.  console.log(arr.toString()); |

Когда все готово, нужно закоммитить изменения, чтобы закончить процесс:

|  |  |
| --- | --- |
| 1  2 | $ git add -A  $ git commit -m "Array printing conflict resolved." |

Как вы можете заметить, процесс довольно утомительный и может быть очень сложным в больших проектах. Многие разработчики предпочитают использовать для разрешения конфликтов клиенты с графическим интерфейсом. (Для запуска нужно набрать git mergetool).