**Команды командной строки Unix систем**

Перед началом работы необходимо сообщить git, кто вы и как вас представлять другим участникам распределенной системы контроля версий. Пока вы не подпишетесь, система не даст вам регистрировать снимки проекта, коммиты.

Все ваши изменения должны быть подписаны вашим именем и электронным адресом, чтобы другие участники проекта знали, чьи это правки и как с вами связаться. Сделать это нужно один раз, как правило, сразу после установки git, если вы переустановите систему, процедуру потребуется повторить.

Чтобы задать ваши имя и электронный адрес, следует воспользоваться командой **git config**:

**$ git config --global user.name "Evgeny Kirillin"**

**$ git config --global user.email** [**jenek96@gmail.com**](mailto:jenek96@gmail.com)

Убедиться в том, что настройки успешно установлены, можно, запросив их список при помощи команды **git config --list**

**$ git config --list**

**user.name=Evgeny Kirillin**

**user.email=jenek96@gmail.com**

Определить местоположение конфигурационного файла можно, если добавить к команде параметр **--show-origin**:

**$ git config --list --show-origin**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

В UNIX-подобных операционных системах в качестве разделителя директорий/каталогов используется прямой слеш — **/**.

В Unix для переключения между дисками нужно указать слеш вправо и имя диска.

Вывод команд в UNIX-подобных операционных системах, можно перенаправлять при помощи при помощи знака больше **>**:

**dir > text.txt**

запишет в файл text.txt содержимое команды dir (в данном случае, запишет туда список файлов директории).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Чтобы пометить текущую папку как git-проект, выполняем команду git init:

**$ git init**

Команда создает в текущей папке подпапку .git, именно в ней будет содержаться вся необходимая информация и сохраняться текущее состояние git-проекта.

**Генерация открытого и закрытого ключей SSH для работы с удаленными git-репозиториями**

Клиент заводит пару ключей: открытый и закрытый.

Закрытый ключ помещается в домашнем каталоге локального компьютера ~/.ssh/id\_rsa.

Открытый (public) ключ id\_rsa.pub размещается на сервере в конфигурационном файле ~/.ssh/authorized\_keys в домашнем каталоге пользователя, под учетной записью которого осуществляется вход на сервер.

Для генерации ключей служит команда **ssh-keygen**, которую следует выполнить на клиентской машине:

**$ ssh-keygen -t rsa**

**Generating public/private rsa key pair.**

**Enter file in which to save the key (/home/igor/.ssh/id\_rsa):**

**Created directory '/home/igor/.ssh'.**

**Enter passphrase (empty for no passphrase):**

**Enter same passphrase again:**

**Your identification has been saved in /home/igor/.ssh/id\_rsa.**

**Your public key has been saved in /home/igor/.ssh/id\_rsa.pub.**

**The key fingerprint is: SHA256:1eVTqU42ZVtns6DPMN2S1y3EDdK4K+ClI8XYApzevEU igor@cruiser**

**The key's randomart image is:**

**+---[RSA 2048]----+**

**| . . .+ooo|**

**| + E ..=+\*=|**

**| . + = . +oX.O|**

**| . = \*..+.O.\*o|**

**| \*S+ O.+. |**

**| o + . .+ |**

**| . . . |**

**| |**

**| |**

**+----[SHA256]-----+**

Во время выполнения команда задаст несколько вопросов, в частности, она попросит указать путь, куда будут сохранены ключи (по умолчанию папка **.ssh** домашнего каталога пользователя), далее будет предложено ввести пароль для закрытого ключа. Настоятельно рекомендуется его указать, т. к. в случае, если ключ будет похищен злоумышленниками, им не смогут воспользоваться без пароля.

**$ ls -la ~/.ssh**

В результате выполнения команда создаст в домашнем каталоге скрытый подкаталог **.ssh**, в котором размещается закрытый (**id\_rsa**) и открытый (**id\_rsa.pub**) ключи. Закрытый ключ никогда не должен попадать в чужие руки, передаваться через незащищенные сетевые каналы. В идеале он вообще не должен покидать компьютер, на котором был создан. Открытый ключ может свободно распространяться, вы можете его регистрировать на всех хостах, к которым хотите получить доступ, включая площадки хост-провайдеров и удаленные репозитории, вроде GitHub. После того, как ключи будут сгенерированы, открытый ключ **id\_rsa.pub** переправляется на сервер и дописывается в конец файла **authorized\_keys**:

**$ cd .ssh**

**$ cat authorized\_keys**

**$ echo "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCk1C/2ZgrQRKlOezXSuQX9NFr1y2ctMSWHkJwrMPKXtqLO2IZuCN+pyPA2X5GDywDml/yS50QRpJhuFJdQRHeSpOEbi1vmYEDh9PWty4gtGddrBfXfwysmjCmlpgfGaUCT1zc6U1yA/V8D79ufkfmmtGU6ozVwDJJ7obU6ruXNAqm08KRds/jXgnwr94x3QX79z+l94GiQ/M/z1O6+zeZqmLMhw76vqqniNjH3H+Z6cqnMurm78HwoLskbPFfl0D0dna1HkKWuHO69GT2qYt7In67ohET8/s+aBTSfEFckLz338qJnb7Xd sn+pSc4lbwC751fLGK1EESyryo1KOUIN igorsimdyanov@MacBook-Pro-Igor.local" >>**

**authorized\_keys**

В случае GitHub следует перейти в настройки (**Settings**) в раздел SSH and GPG keys и добавить новый SSH-ключ.

Порядок работы с SSH в Windows более подробно описан в методичеких материалах Урока 12. Впрочем верхний способ вполне себе работает в windows, если скачать командную строку unix - Git Bash. (короче см. Урок 12)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Добавление файлов в индекс и комментирование**

Чтобы добавить файл под контроль git, необходимо проиндексировать его при помощи команды git add:

**$ git add .**

Здесь точка означает все новые файлы в папке, вместо этого можно указать избранные файлы или

каталоги:

**$ git add hello.txt**

Такая команда добавит в индекс только файл hello.txt

После того, как мы сообщили Git, какие файлы проекта мы хотим отслеживать, их можно зафиксировать. Для этого создается снимок состояния, который называется коммитом. Сделать это можно при помощи команды **git commit**:

**$ git commit -m 'Первый коммит'**

В параметре **-m** мы задаем комментарий к коммиту.

Так же можно использовать просто команду:

**$ git commit**

для создания комментария. Тогда откроется редактор по умолчанию в котором можно будет ввести комментарий.

Для того, что бы сменить текстовый редактор по умолчанию, можно воспользоваться командой:

**$ git config core.editor notepad** (например notepad).

**Клонирование**

Если проект под управлением Git уже существует в каком-то сетевом репозитории, его можно загрузить при помощи команды **git clone**. Такой процесс называется клонированием.

В консоли выполняем команду **git clone**, передавая ей в качестве аргумента ссылку на клонируемый репозиторий:

**$ git clone https://github.com/web-standards-ru/dictionary.git**

Если необходимо переименовать папку, новое название можно указать сразу после адреса репозитория в команде **git clone**:

**$ git clone https://github.com/web-standards-ru/dictionary.git dict**

В результате создается копия репозитория в папке **dict**.

**GITK и GIT GUI**

Графический клиент **gitk** позволяет просматривать историю. Запустить его можно командой **gitk** в git-проекте:

**$ gitk**

В окне **gitk** можно видеть историю коммитов: время их создания, автора, а также изменения, которые были добавлены в этих коммитах.

Клиент **git-gui** предназначен для формирования коммитов:

**$ echo 'Hello, Git!' > hello.txt**

После запуска клиента командой **git gui** можно видеть измененные файлы, понимать, что в них поменялось, добавлять комментарии.

Нажимаем кнопку **Stage Changed**, чтобы проиндексировать файл. Это аналогично вызову команды **git add**. Далее для создания коммита можем нажать кнопку **commit**.

**Добавление файлов в git проект**

Еще раз остановимся на том, как файлы добавляются в git. Пусть есть проект, в нем файлы могут быть в нескольких состояниях:

**● отслеживаемом,**

**● не отслеживаемом.**

**В отслеживаемом** состоянии файл и его содержимое зарегистрированы в git, **в неотслеживаемом** - файл новый и git о нем пока ничего не знает.

В любом случае git реагирует на оба типа файлов и помечает их как доступные для добавления. Такие файлы могут быть переведены **в индексированное состояние** при помощи команды **git add**​. В будущий коммит войдут только те файлы, которые находятся в индексе.

Можно добавлять или убирать файлы из индекса. После выполнения команды ​ git commit содержимое индекса превращается в снимок системы контроля версий, а содержимое индекса обнуляется.

Для того, чтобы выяснить статус проекта, мы можем воспользоваться командой **git status**:

**$ git status**

**On branch master**

**nothing to commit, working tree clean**

Так команда сообщает нам, что изменений в проекте нет.

Обычно вывод команды ​ **git status** ​ довольно объемный, можно добиться более компактного вывода, если добавить ей параметр ​ **-s:**

**$ git status -s**

**M​ hello.txt**

**??​ readme.txt**

Новые, не отслеживаемые файлы помечены ​ **??** слева от них, файлы, добавленные в отслеживаемые, помечены **A**, отредактированные файлы помечены **M** и так далее.

Git анализирует только отслеживаемые файлы, поэтому, если мы захотим выяснить, какие изменения были внесены в кодовую базу проекта, он выведет изменения только для hello.txt. Воспользуемся командой ​ **git diff**, чтобы в этом убедиться:

**$ git diff**

**--git a/hello.txt b/hello.txt**

**index af5626b..b356b4a 100644**

**--- a/hello.txt**

**+++ b/hello.txt**

**@@ -1 +1,2 @@**

**Hello, world!**

**+Новая строка**

Итак, мы получаем изменения по файлу hello.txt, про файл readme.txt git не упоминает, так как он не отслеживаемый.

Чтобы файл начал отслеживаться, его необходимо добавить в индекс, делается это при помощи команды​ **git add**:

**$ git add .**

Если мы сейчас выполним команду ​**git diff​**, она ничего не вернет:

**$ git diff**

В проекте нет не отслеживаемых файлов, нет изменений вне индекса. Однако, если мы добавим в команду параметр ​**--cached​**, мы можем увидеть изменения, которые сейчас находятся в индексе:

**$ git diff --cached**

**diff --git a/hello.txt b/hello.txt**

**index af5626b..b356b4a 100644**

**--- a/hello.txt**

**+++ b/hello.txt**

**@@ -1 +1,2 @@**

**Hello, world!**

**+Новая строка**

**diff --git a/readme.txt b/readme.txt**

**new file mode 100644**

**index 0000000..9c68326**

**--- /dev/null**

**+++ b/readme.txt**

**@@ -0,0 +1 @@**

**+Новый проект**

Команда ​ **git add .** настолько часто предваряет ​ **git commit**​, что последняя команда снабжается дополнительным параметром **-a**​. Этот параметр автоматически выполняет перевод всех измененных файлов в индексное состояние.

**$ git commit -a**

Можем сразу добавить и закомментировать все изменения одним вызовом ​ **git commit​**:

**$ git commit -am**

**Удаление файлов**

Что делать, если файл добавлен по ошибке и его необходимо удалить? Для этого в git тоже есть команды.

Про неотслеживаемый файл git ничего не знает, поэтому, чтобы от него избавиться, достаточно его просто удалить командой **unlink**:

**$ unlink to\_delete.txt**

Если же git уже осведомлен о файле, то git нам уже подсказывает, при помощи команды **git status**, что делать дальше с файлом:

● либо индексировать при помощи команды ​ **git add​ ,**

● либо откатывать изменения при помощи команды ​ **git restore --staged**

**История изменений**

При создании коммит Git запоминает текущее состояние проекта. Посмотреть историю изменений можно при помощи команды:

**$ git log**

По умолчанию для каждого коммита выводится хэш, автор, время добавления и комментарий к коммиту.

Ограничить количество выводимых коммитов можно при помощи параметра -2.

**$ git log -2**

По умолчанию в отчете команды выводится только метаинформация. Чтобы вывести детальные изменения для каждого из коммитов, команде следует передать параметр -p.

**$ git log -2 -p**

Если нужна статистика для каждого коммита, можно воспользоваться параметром **--stat**:

**$ git log -2 --stat**

Еще больше сократить вывод команды **git log** можно при помощи параметра **- --pretty**, который изменяет формат вывода:

**$ git log --pretty=oneline**

Таким образом можно добиться, чтобы коммиты выводились в одну строку. Можно последовательно почитать комментарии к коммитам в хронологическом порядке.

Если вместо **oneline** использовать **format**, можно сформировать свой собственный формат логов:

**$ git log --pretty=format:"%h - %an, %ar : %s"**

Он выдаст:

**9992e6d - Igor Simdyanov, 13 minutes ago : Изменения в README**

**e0c800c - Igor Simdyanov, 14 minutes ago : Добавляем файл readme.txt**

**dca0e8e - Igor Simdyanov, 20 minutes ago : Comment**

**c24ec38 - Igor Simdyanov, 23 minutes ago : Первый коммит**

Полный список параметров форматирования и их значения можно обнаружить в **документации к git**.

**Поиск коммитов по дате**

Для поиска коммитов в определенном временном интервале можно использовать параметры **--since** и **--until**:

**--until** позволяет задать дату, с которой мы начинаем поиск,

**--since** — задать, до какого момента в прошлом мы ищем коммиты.

Команда ниже позволяет искать коммиты, начиная двумя неделями назад и до текущего момента:

**$ git log --since=2.week**

Следующая команда ищет коммиты с начала проекта до даты недельной давности:

**$ git log --until=1.week**

Можно одновременно использовать оба параметра, чтобы задать интервал поиска в прошлом.

Допускается использовать и точные даты.

**$ git log --since=2019-03-01 --until=2019-03-25**

**Поиск коммита по строке и автору**

При помощи параметра -S можно найти коммит, в котором была добавлена или удалена заданная строка:

**$ git log -SGit -p**

Команда выше ищет коммиты, в которых упоминается **слово Git**.

Иногда необходимо найти коммиты, которые принадлежат конкретному разработчику. Сделать это можно при помощи параметра **--author**:

**$ git log --author="Evgeny Kirillin"**

Иногда стоит обратная задача, нужно, наоборот, выяснить, кто автор этой строки в файле и в каком коммите она в последний раз менялась. Для этого предназначена команда **git blame**:

**$ git blame readme.txt**

В качестве параметра команда принимает имя файла. В результате выводится содержимое файла, в котором каждая строка помечена хэшем коммита, автором и датой последнего изменения.

**Игнорирование**

Все, что помещается в **.git-репозиторий**, хранится в истории проекта. Если в коммит попал гигабайтный файл, последующее удаление его не приводит к удалению файла из истории.

Пользователи, выполняющие клонирование репозитория, будут вынуждены каждый раз выгружать всю историю, включая этот гигантский файл. Чтобы Git игнорировал нежелательные файлы, предусмотрен специальный конфигурационный **файл .gitignore**, который помещается в корень проекта.

Внутри файла прописывается шаблон: файлы, которые удовлетворяют им, игнорируются Git.

**\*.log**

**tmp/\***

Например, так мы будем игнорировать все файлы, **которые завершаются расширением log**, а также **файлы в папке tmp**.

**Ресурс http://gitignore.io/** позволяет автоматически формировать содержимое **.gitignore-файла**.

Достаточно указать ключевые слова, например Sublime, Yii2, vim, и на их основе будет предложена **заготовка для .gitignore**, которая исключает временные и вспомогательные файлы.

**Пустые папки**

Git ориентируется на файлы, папки попадают под отслеживание, если только в них находятся файлы, **пустые папки — игнорируются**.

В результате папка не попадает в индекс. **Это не всегда удобно**, иногда папки нужны для функционирования проекта. Например, приложение может ожидать наличие папки log, но так как мы запретили log-файлы, такая папка рассматривается как пустая и не попадает в индекс git-проекта.

**Чтобы избежать этого, в папку часто добавляют файл:**

**.keep**

Это просто обычный пустой файл, он не является частью экосистемы git и вместо него можно использовать любой другой файл. Файл

начинается с точки, поэтому в UNIX-подобных операционных системах он рассматривается как скрытый. В Windows это не так, однако это название получило широкое распространение в тех случаях, когда важно сохранить пустую папку в git-проекте.

**Ветки**

Система контроля версий мало бы отличалась от системы резервного копирования, если бы позволяла только делать снимки состояния проекта. Особенностью таких систем является возможность создания нескольких веток, которые, в свою очередь, являются основой для командной работы над проектом.

Сразу после создания проекта появляется **основная ветка master**:

**$ git branch**

**\* master**

В этом можно убедиться при помощи команды **git branch**. Данная команда выводит список доступных веток, помечая текущую звездочкой.

Если после ключевого слова **branch** указать название, будет создана новая ветка:

**$ git branch feature**

**$ git branch**

Для переключения на новую ветку можно воспользоваться командой **git checkout** или **git switch**:

**$ git checkout feature**

**$ git branch**

**feature**

**\* master**

Процесс создания новой ветки можно сократить до одной команды **git checkout**, если воспользоваться ключом **-b**:

**$ git checkout -b test**

**Switched to a new branch 'test'**

**$ git branch**

**feature**

**master**

**\* test**

**Переименование ветки**

Переименовать ветку можно при помощи команды **git branch**, для этого ей необходимо передать параметр **-m**, после которого указывается имя ветки и ее новое имя.

**$ git branch -m feature to\_delete**

**$ git branch**

**master**

**\* test**

**to\_delete**

**Мы только что переименовали ветку feature в to\_delete.**

Для удаления ветки следует воспользоваться параметром **-D**:

**$ git branch -D to\_delete**

**Deleted branch to\_delete (was beefeb1).**

**$ git branch**

**master**

**\* test**

Как видим, **to\_delete** из списка веток пропала. После создания и переключения новые коммиты будут размещаться в текущей ветке, не затрагивая историю изменений основной ветки **master**.

**Слияние веток**

Коммиты одной ветки можно добавить в другую ветку, такая процедура называется слиянием.

Выполнить слияние можно при помощи команды **git merge**.

**В момент слияния мы должны находиться в ветке, куда будем сливать изменения**, поэтому переключаемся в ветку master:

**$ git checkout master**

Помимо **checkout** можно использовать **switch** (то же самое).

**Смержим (git merge)** изменения из ветки test в master:

**$ git merge test**

Если мы сейчас выполним команды **git log** с параметром **graph**, можем увидеть слияние веток, отрисованное псевдографикой:

**$ git log --pretty=format:"%h -%d %s %cd" --graph**

**Псевдонимы команд**

Для объемных команд можно использовать короткие запоминающиеся псевдонимы. Создать псевдоним можно при помощи команды **git config**, добавив **свойство alias**, после него следует указать имя псевдонима.

Давайте введем **псевдоним gg** **для команды git log с параметром --graph**: (все пишется в одну строку очевидно)

**$ git config --global alias.gg 'log --pretty=format:"%C(yellow)%h%Creset -%C(green)%d%Creset %s %C(bold blue)(%cn) %C(green)%cd%Creset" --graph --date=format:"%d.%m.%Y %H:%M"'**

Элементы в лог-выводе мы подсвечиваем разными цветами. **Теперь можем воспользоваться командой:**

**$ git gg**

**Разрешение конфликтов**

Git прекрасно справляется с объединением изменений в разных файлах и даже в разных частях одного файла. Однако при командной работе неизбежны конфликтные ситуации, когда изменения одного разработчика противоречат тем, которые вносит другой разработчик.

Если мы **сливаем (merge)** изменения одного и того же файла из ветки test в master, то у нас возникает конфликтная ситуация:

**$ git merge test**

**Auto-merging hello.php**

**CONFLICT (add/add): Merge conflict in hello.php**

**Automatic merge failed; fix conflicts and then commit the result.**

Git не смог автоматически слить изменения. Из отчета команды можно видеть, что конфликтная ситуация возникала в файле hello.php.

Давайте отправимся в редактор и разрешим ее вручную (что мы примерно увидим в редакторе):

**<?php**

**<<<<<<< HEAD**

**echo "Hello, world!";**

**=======**

**session\_start();**

**echo "Hello, world!";**

**>>>>>>> test**

Конфликты обозначаются угловыми скобками, чтобы их было хорошо видно в коде. В верхней части приводится код текущей ветки, master, в нижней — той ветви, которая сливается (в данном случае это ветка test).

Нам нужно выбрать правильный вариант или написать новый, который учтет изменения из обеих частей:

**<?php**

**session\_start();**

**echo "Hello, world!";**

**Оставляем вариант из ветки test.** Конфликт разрешен. Теперь необходимо вернуться в консоль и зафиксировать изменения.

**$ git status**

**$ git diff**

(Проверили, все в порядке)

Добавим изменения в индекс:

**$ git add .**

И создадим коммит:

**$ git commit -m 'Fix confict'**

Теперь можно быть уверенными, что учтены изменения из обеих веток.

**Удаленный Git-репозиторий**

Удалённый репозиторий представляет собой компьютер с git, расположенный в интернете или ещё где-то в сети.

Для того, чтобы начать пользоваться удаленным git-репозиторием, проще всего начать с готовых **git-хостингов**, например, **GitHub** или **BitBucket**. Если **git-хостинги** по каким-то причинам не подходят, можно развернуть свой собственный git-репозиторий: для этого достаточно виртуального сервера с SSH-доступом.

В случае, если помимо консольного доступа необходим веб-интерфейс, можно воспользоваться **GitLab**, который тоже можно развернуть на собственном сервере. **GitLab** предоставляет веб-интерфейс для управления git-репозиториями, правами доступа и предоставляет схожий с Git-хостингами функционал.

**Git-хостинг GitHub**

Чтобы получить доступ к созданию собственных репозиториев, необходимо зарегистрироваться на сайте **http://github.com**. Далее следует перейти в редактирование профиля на странице https://github.com/settings/profile. В разделе SSH Key следует загрузить свой публичный ключ.

После этого можно приступать к созданию собственных проектов. В верхнем меню следует найти значок + и выбрать в выпадающем списке пункт **New repository**, на странице создания будет предложено назвать проект.

Пусть проект будет называться hello. Открывшаяся пустая страница проекта предлагает два варианта инициализации: развертывание проекта с нуля или подключение уже готового репозитория.

Если у нас уже есть репозиторий, то лучше воспользоваться вторым вариантом:

**$ git remote add origin** [**git@github.com:igorsimdyanov/hello.git**](mailto:git@github.com:igorsimdyanov/hello.git)

Здесь **git** — имя пользователя на сервере, **github.com** — адрес сервера, а **igorsimdyanov/hello.git** — путь к репозиторию на сервере.

**$ git push -u origin master**

Команда **git push** отправляет изменения на git-сервер. При первом обращении может быть задан вопрос, хотите ли вы подсоединиться к новому ssh-серверу. На него следует ответить утвердительно, набрав yes. При последующих выполнениях команды **git push** данный вопрос задаваться уже не будет.

Если мы сейчас перезагрузим страницу репозитория, то сможем увидеть файлы нашего проекта на github.com.

Команда **git remote add origin** регистрирует удаленный сервер в файле **.git/config.**

Можно видеть, что после выполнения команды в нем появляется секция **remote** с адресом удаленного Git-репозитория. Файл **.git/config** можно исправлять вручную.

Название **origin**, хотя и предлагается по умолчанию, не является обязательным — его можно изменить. В этом случае в команде **git push** вместо **origin** следует указывать соответствующее название репозитория.

Давайте добавим псевдоним **github**:

**$ git remote add github git@github.com:igorsimdyanov/hello.git**

Теперь отправлять изменения в репозиторий можно при помощи команды:

**$ git push github master.**

Т. е., после команды **git push** сначала указываем имя удаленного репозитория, а затем название ветки.

Список доступных удаленных репозиториев можно узнать при помощи команды:

**$ git remote**

**github**

**origin**

**Клонирование репозитория**

После того, как проект развернут в удаленном Git-репозитории, его можно клонировать при помощи команды git clone.

Набираем команду git clone:

$ git clone git@github.com:igorsimdyanov/hello.git

После выполнения команды у нас будет две папки: test и hello. Они позволят нам смоделировать ситуацию командной работы.

**Обмен изменениями**

Перейдем, например, в папку hello, создадим коммит с файлом index.php и напишем в нем:

<?php echo "Hello world!";

Зафиксируем изменения в виде коммита:

**$ git add index.php**

**$ git commit -m "Индексный файл проекта"**

Отправить изменения на сервер можно при помощи команды **git push**, первый параметр в которой соответствует удаленному репозиторию, а второй параметр — ветке.

**$ git push origin master**

Если удаленный репозиторий у нас один, а работа идет с основной веткой master, параметры можно опустить и записать команду в короткой форме:

**$ git push**

На странице проекта в GitHub можно убедиться, что файл успешно добавлен в репозиторий.

Теперь перейдем в проекте в папку test, создадим коммит с файлом phpinfo.php и напишем в нем:

<?php

phpinfo();

Переходим в папку test:

**$ cd ../test**

Запрашиваем статус проекта:

**$ git status**

Добавляем файл в индекс:

**$ git add .**

И формируем коммит

**$ git commit -m "Добавление phpinfo-отчета"**

Давайте попробуем отправить коммит на сервер:

**$ git push origin master**

Попытка завершается неудачей, сервер сообщает, что в удаленном репозитории есть незагруженные изменения, и предлагает их предварительно загрузить. **Git предотвращает конфликтные ситуации: если при слиянии возникнет конфликт, Git предложит их разрешить.**

Давайте выполним команду **git pull** и скачаем изменения:

**$ git pull origin master**

Вот теперь можно отправлять свои изменения на сервер:

**$ git push origin master**

Если теперь перейти на страницу проекта в GitHub и перезагрузить страницу, можно увидеть, что файл успешно доставлен в удаленный репозиторий.

Команда **git pull**, как правило, применяется для загрузки лишь одной указанной ветки, для того, чтобы получить все изменения с удаленного сервера, следует воспользоваться командой **git fetch**.

**$ cd ../hello**

**$ git fetch**

**Управление удаленными ветками**

При работе с репозиторием ветка может располагаться локально и на удаленном репозитории. Пока вы не отправите коммиты ветки при помощи команды **git push**, они не появятся на удаленном сервере. Загрузить изменения из удаленного репозитория можно при помощи команды **git pull**.

Создадим локальную ветку fix\_error:

**$ git switch -c fix\_error**

**Switched to a new branch 'fix\_error'**

Запросим список текущих веток:

**$ git branch**

**\* fix\_error**

**master**

Изменение локальной ветки не отражается на удаленной, пока мы не отправим изменения на сервер. Давайте в этом убедимся, перейдя в GitHub на страницу branches.

На удаленном сервере у нас находится лишь одна ветка master, о новой ветке fix\_error GitHub не подозревает, так как она не была еще отправлена на сервер.

Список удаленных веток можно посмотреть и в консоли. Для этого команде **git branch** следует добавить параметр **-r**:

**$ git branch -r**

**origin/HEAD -> origin/master**

**origin/master**

Как видим, в отчете команды тоже только одна удаленная ветка — **master**. Обратите внимание, что она имеет суффикс в виде имени удаленного репозитория, у нас это **origin**.

Давайте отправим текущую ветку в удаленный репозиторий:

**$ git push origin fix\_error**

Для этого воспользуемся командой **git push**, после которой указываем имя репозитория **origin** и название ветки **fix\_error**.

Изменения отправлены, давайте посмотрим список удаленных веток.

**$ git branch -r**

**origin/HEAD -> origin/master**

**origin/fix\_error**

**origin/master**

Итак, новая ветка появилась на сервере.

Что будет, если мы переименуем или удалим ветку локально? Давайте переименуем ветку fix\_error в to\_delete:

**$ git branch -m fix\_error to\_delete**

**$ git branch**

**master \***

**to\_delete**

**$ git branch -r**

**origin/HEAD -> origin/master**

**origin/fix\_error**

**origin/master**

Как видно, ветка была переименована только локально, изменения не коснулись удаленного репозитория. Мы можем попробовать отправить ветку **to\_delete** на удаленный сервер:

**$ git push origin to\_delete**

**$ git branch -r**

**origin/HEAD -> origin/master**

**origin/fix\_error**

**origin/master**

**origin/to\_delete**

В результате на сервере появится сразу три ветки. Более того, мы **можем даже удалить локальную ветку** **to\_delete**:

**$ git switch master**

**Switched to branch 'master'**

**Your branch is ahead of 'origin/master' by 1 commit.**

**(use "git push" to publish your local commits)**

Давайте переместимся на ветку **master**. У нас не получится удалить выбранную ветку, нужно это делать с какой-то другой:

**$ git branch -D to\_delete**

**Deleted branch to\_delete (was 9e02299).**

**$ git branch**

**\* master**

**$ git branch -r**

**origin/HEAD -> origin/master**

**origin/fix\_error**

**origin/master**

**origin/to\_delete**

Опять же, изменения затрагивают только локальные ветки, ветки удаленного репозитория остаются незатронутыми.

Для того, чтобы управлять удаленными ветками, следует воспользоваться отдельным набором команд. Например, для удаления ветки на сервере, следует воспользоваться командой **git push**:

**$ git push origin :to\_delete**

**To github.com:igorsimdyanov/hello.git**

**- [deleted] to\_delete**

Однако **имя ветки следует предварить двоеточием**. Запрашиваем список удаленных веток, чтобы убедиться, что ветки **to\_delete** больше не существует:

**$ git branch -r**

**origin/HEAD -> origin/master**

**origin/fix\_error**

**origin/master**

Если потребуется переименовать ветку, то сделать это одной командой довольно сложно. Проще всего создать на удаленном сервере копию ветки, а старую ветку удалить. Давайте **переименуем ветку fix\_error в to\_delete**. Для этого создаем локально копию удаленной ветки с именем **to\_delete**:

**$ git branch to\_delete origin/fix\_error**

**Branch 'to\_delete' set up to track remote branch 'fix\_error' from 'origin'.**

**$ git branch**

**\* master**

**to\_delete**

**Такой командой мы как бы создаем новую ветку и называем ее to\_delete, хотя она полностью копирует ветку с хостинга (вроде бы).**

Забрасываем локальную ветку **to\_delete** на удаленный сервер:

**$ git push origin to\_delete**

Теперь старую ветку **fix\_error** можно удалить:

**$ git push origin :fix\_error**

Таким образом ветка будет переименована:

**$ git branch**

**\* master**

**to\_delete**

**Статус и навигация по git-проекту**

Выяснить, на какой ветке мы сейчас находимся, можно при помощи команды **git branch** или **git status**. Если в качестве командной оболочки используется bash, можно воспользоваться решением git-aware-promt (https://github.com/jimeh/git-aware-prompt), который встраивает название текущей ветки в подсказку командной строки. Обратите внимание, что проект тоже расположен на GitHub, чтобы им воспользоваться, нам придется его клонировать.

Давайте создадим в домашнем каталоге **папку .bash**, в которой будут располагаться скрипты. Здесь мы для этого воспользуемся консольной командой **mkdir**, однако вы можете создать папку любым удобным для вас способом:

**$ mkdir .bash**

Переходим в папку:

**cd .bash**

Теперь нам необходимо в нее клонировать репозиторий с проектом git-aware-promt. Возвращаемся в git-репозиторий проекта, копируем ссылку для клонирования. **Обратите внимание, что ссылка для клонирования может начинаться с префикса https или git@ — это разные протоколы общения с git-сервером.** Использование **git@** потребует наличие на git-сервере открытого ключа. Этот способ удобен при работе с вашими собственными репозиториями. При использовании протокола **https://** мы можем читать репозиторий без ограничений. Именно эту ссылку мы и копируем, чтобы клонировать проект. Однако если мы захотим закинуть свои изменения в чужой проект, нам либо откажут, либо запросят логин и пароль.

Клонируем проект при помощи команды **git clone**:

**$ git clone** [**https://github.com/jimeh/git-aware-prompt.git**](https://github.com/jimeh/git-aware-prompt.git)

Чтобы проект заработал, нам потребуется внести изменения в файл **.bash\_profile** домашнего каталога. Этот файл выполняется всякий раз при инициализации терминала.

**$ source "~/.bash/git-aware-prompt/main.sh**

При помощи команды **source** выполняется скрипт **main.sh** из проекта **git-aware-prompt**.

Кроме того, модифицируем переменную окружения **PS1** таким образом, чтобы в нее добавлялось название текущей ветки:

**export PS1="\w \[\$txtcyn\]\$git\_branch\[\$txtred\]\$git\_dirty\[\$txtrst\]\$ "**

**Переменная окружения PS1 в Unix-мире несет ответственность за формирование подсказки командной строки.**

Открываем новый терминал и переходим внутрь проекта:

**$ cd GitProject**

Как видно, подсказка командной строки изменяется, теперь она содержит название текущей ветки: **GitProject (master)$**

Если мы переключимся на ветку **to\_delete**:

**$ git switch to\_delete**

В подсказке командной строки появится название ветки to\_delete: **GitProject (to\_delete)$**

Давайте внесем изменения в файл **README.md** (добавим строку Hello, world!):

**GitProject (to\_delete) \*$**

Как видно, к подсказке добавляется красная звездочка, которая сообщает о том, что имеются неиндексированные изменения.

**Как скрывать изменения**

Если в ходе работы над веткой мы внесли какие-то изменения в файл, но вынуждены срочно прерваться, чтобы переключиться в другую ветку, в лучшем случае незафиксированные изменения перейдут в эту новую ветку.

**$ git switch master**

Чтобы этого не происходило, следует либо выполнить команду **git commit**, либо отложить изменения:

**$ git switch to\_delete**

Для того, чтобы отложить изменения на потом, нам потребуется добавить их в индекс:

**$ git add .**

**$ git status**

После этого можно выполнить команду **git stash**:

**$ git stash**

Обратите внимание, что красная звездочка исчезла, мы очистили текущую ветку от изменений:

**$ git status**

**Однако они не потеряны, а спрятаны в stash-списке.** Давайте их посмотрим при помощи команды **git stash list**:

**$ git stash list**

Теперь мы можем переключиться на другую ветку и поработать с ней:

**$ git switch master**

**$ touch hello.txt**

**$ git add .**

**$ git commit -m 'New file hello.txt'**

Чтобы вернуться к ветке to\_delete и незавершенным изменениям, переключаемся на нее при помощи команды switch:

**$ git switch to\_delete**

**$ git stash list**

Теперь можно вернуть данные из stash, для этого воспользуемся командой **git stash apply**:

**$ git stash apply**

Изменения, которые были спрятаны в stash-списке, снова доступны для работы. Команду **git stash** можно выполнять несколько раз подряд в разных ветках. **Каждый раз в stash-список будет добавляться новая запись**:

**$ git add .**

**$ git stash**

**$ git stash list**

Они пронумерованы цифрами 0, 1 и т. д.

По умолчанию git использует последнюю запись, если вы хотите воспользоваться более ранним вариантом, можно явно указать метку stash-записи после команды **git stash apply**:

**$ git stash apply stash@{1}**

Впрочем, держать много записей в стеше не стоит, удалить записи можно при помощи команды **git stash drop**:

**$ git stash drop**

**$ git stash list**

**$ git stash drop**

**$ git stash list**

**Версионирование, теги**

Версии программного обеспечения, как правило, состоит из трех цифр, первая из которых называется мажорной, вторая — минорной, а третья — патч-версией.

Например, 1.4.0.

Мажорная версия меняется довольно редко, как правило, в случае кардинальной переработки программного продукта.

Минорная версия отвечает за регулярные релизы, в рамках которых добавляется новый функционал и исправляются ошибки.

В рамках патч-версии изменений в возможностях программного обеспечения не производится. Как правило, в рамках такой версии закрываются критические уязвимости.

Это не единственно возможный способ версионирования, например, можно ориентироваться на даты выхода продукта. Так поступают во многих дистрибутивах Linux, например Ubuntu, и в линейке интеграционных сред JetBrains.

Какая бы модель версионирования не была выбрана, в системе контроля версий придется отметить коммит, который отвечает за версию, чтобы в любой момент времени иметь возможность собрать нужную версию продукта.

**Теги**

Для специальной пометки коммитов git предоставляет механизм тэгов или, как еще говорят, меток.

История изменений может включать тысячи коммитов, осуществлять поиск и навигацию по которым может быть затруднительно.

Тем не менее, нам может потребоваться быстро найти коммит, который соответствует стабильной версии приложения. Эту проблему решают метки, т. е., короткое запоминающееся название, по которому коммит легко найти среди других.

Более того, метки — это тоже тэги, которые назначаются коммитам автоматически. Одному коммиту мы можем назначить любое количество тэгов.

Для того, чтобы создать тэг, можно воспользоваться командой **git tag**, после которого указывается название тэга:

**$ git tag v1.3.0**

При попытке создания уже существующего тэга git сообщает, что такой тэг уже существует.

Тэги можно снабжать комментариями:

**$ git tag -a v1.4.0 -m 'Новый тэг'**

Для этого метку тэга добавляем при помощи параметра -а (а маленькое), а комментарий добавляем при помощи параметра -m (м маленькое):

**$ git tag**

**v1.3.0**

**v1.4.0**

Для просмотра списка доступных тэгов можно вызвать команду **git tag** без параметров.

Чтобы отфильтровать вывод, можно воспользоваться поиском по шаблону:

**$ git tag -l v1.1.\***

**v1.1.0**

Символ звездочки означает любое количество других символов. Если потребуется удалить тэг, команде **git tag** следует добавить параметр **-d**:

**$ git tag -d v1.3.0**

**Deleted tag 'v1.3.0' (was 089a6e8)**

Убедимся, что тэг исчез, для этого воспользуемся командой **git tag** без параметров:

**$ git tag**

**v1.4.0**

Как видим, тэга v1.3.0 больше не существует.

**Связь коммита и тэга**

Тэг — это всегда ссылка на коммит. На один и тот же коммит можно навесить множество тэгов. Более того, хэш тоже можно рассматривать как тэг, просто его сложнее запомнить. Поэтому мы вводим свои собственные тэги, которые человеку запомнить гораздо проще.

Посмотрим историю коммитов при помощи команды, которую мы сконструировали из команды **git log** на одном из прошлых уроков:

**$ git gg**

**\* 089a6e8 - (HEAD -> master, tag: v1.1.0) New line in hello.php (Evgeny Kirillin) 05.05.2019 13:15**

**\* 316d747 - (origin/to\_delete, origin/master, origin/HEAD) Merge branch 'master' of github.com:Glebasher57/hello (Evgeny Kirillin) 05.05.2019 12:00**

**...**

Посмотреть описание текущего коммита можно при помощи команды **git show v1.0.0** (любая другая версия):

**$ git show v1.0.0**

**commit 5219e99c0cc4799b6f9119a99e86c374d8a54979 (HEAD -> master, tag: v1.0.0)**

**Author: Evgeny Kirillin <jenek96@gmail.com>**

**Date: Wed Feb 12 21:24:39 2020 +0300**

**Коммит файла pwd.txt во время теста git stash**

**diff --git a/pwd.txt b/pwd.txt**

**new file mode 100644**

**index 0000000..e69de29**

Вместо тэга v1.1.0 можно использовать хэш:

**$ git show 5219e9**

**Тэг HEAD**

Помимо наших собственных тэгов и хэш-кодов, git предоставляет несколько дополнительных тэгов. Один из них HEAD, указывает на последний коммит:

**$ git gg**

**$ git show** **HEAD**

Как видим, мы всегда можем использовать тэг HEAD для того, чтобы сослаться на последний коммит.

**Переход к произвольному тэгу**

По умолчанию команда **git tag** устанавливает метку на текущий коммит. Однако есть возможность установить тэг на совершенно произвольный коммит. Для этого в конце команды указывается хэш коммита:

**$ git tag -a v1.2.0 -m 'Old version' beefeb1**

Если мы теперь посмотрим историю, можно обнаружить, что тэг v1.2.0 установлен на более ранний коммит:

**$ git gg**

Если необходимо поработать с проектом в том виде, в котором он был на момент создания тэга v1.2.0, можно отпочковать ветку от коммита, на который указывает этот тэг:

**$ git switch -c old\_project** **v1.2.0**

**Switched to a new branch 'old\_ project'**

Для этого после команды создания ветки указывается тэг. Вместо тэга можно использовать хэш коммита. Теперь мы находимся в новой ветке, которая отпочкована от коммита, на который указывает тэг v1.2.

**Отправка тэгов на удаленный репозиторий**

По умолчанию команда **git push** не отправляет тэги на удаленный git-сервер. Для отправки тэгов необходимо явно указать его имя после команды **git push**:

**$ git push origin v1.2.0**

Используется та же команда, что и для отправки веток, только вместо названия ветки мы указываем название тэга.

Если перейти на GitHub на страницу релизов, можно обнаружить, что тэг v1.2.0 появился на удаленном репозитории. Мы отправили лишь один тэг, но если их много и всех их необходимо отправить на удаленный репозиторий, можно воспользоваться командой **git push**, передав ей параметр **--tags**:

**$ git push origin --tags**

Для того, чтобы запросить тэги на удаленном сервере, можно воспользоваться командой **git ls-remote**:

**$ git ls-remote --tags origin**

Для того, чтобы запросить тэги на удаленном сервере, можно воспользоваться командой **git ls-remote**:

**$ git ls-remote --tags origin**

Для того, чтобы удалить тэг на удаленном сервере, можно поступить так же, как и в случае ветки.

**$ git push origin :v1.2.0**

**To github.com:igorsimdyanov/hello.git**

**- [deleted] v1.2.0**

Мы используем команду **git push**, после которой указываем название удаленного репозитория, в данном случае **origin**, и название тэга, которое предваряем двоеточием:

**$ git ls-remote --tags origin**

(запрашиваем список тэгов с удаленного репозитория для проверки)

**Слияние и переносы**

Давайте смоделируем ситуацию, когда мы отпочковались от ветки master и создали ветку feature, а в ней — ряд коммитов. При этом ветка master ушла вперед. Теперь нам необходимо добавить в ветку feature новые изменения. Самый простой способ добавить изменения, это воспользоваться слиянием (merge), ранее о нем уже говорилось. Но есть и другой способ решить такую задачу.

**Git rebase**

Вместо того, чтобы вливать в ветку feature изменения из ветки master, коммиты ветки feature перемещаются таким образом, чтобы ветка feature начиналась там, где заканчивается ветка master. В результате ветка feature содержит все коммиты master плюс новые коммиты.

Добиться этого можно при помощи операции git rebase.

Благодаря такому переносу веток все изменения в master будут доступны в ветке feature. При этом читать историю коммитов становится гораздо проще. Если над проектом работает сразу много человек, параллельных веток может быть много. Операция переноса позволяет вытянуть историю в более или менее линейную последовательность.

Операция git rebase считается сложной, к тому же, изменяющей историю git, поэтому во многих компаниях она запрещена. Тем не менее, знать о ней необходимо.

Переключаемся в ветку master:

**$ git checkout master**

Давайте попробуем снова воспроизвести ситуацию с веткой **feature**, добавив в ветку **master** новый коммит. Добавляем в файл hello.txt строку 'Hello, git rebase!':

**$ echo 'Hello, git rebase!' > hello.txt**

Cоздаем коммит в ветке master:

**$ git add .**

**$ git commit -m 'Adding new hello git rebase line master'**

Для того, чтобы совершить процедуру rebase, перемещаемся в ветку feature:

**$ git switch feature**

Выполняем команду git rebase:

**$ git rebase master**

**First, rewinding head to replay your work on top of it...**

**Applying: Adding new line feature**

В конце команды указываем название ветки master. Теперь ветка feature перемещена вперед.

Если мы попробуем сейчас закинуть ветку на сервер, у нас ничего не получится:

**$ git push origin feature**

Это связано с тем, что на сервере по-прежнему хранится старый вариант ветки **feature**. Чтобы забросить новый вариант, после команды **rebase** потребуется принудительно перезаписать ветку:

**$ git push origin feature -f**

Для этого команде **git push** передается параметр **-f**, от английского **force**. Считается, что такое принудительное перезаписывание ветки потенциально опасно и при неумелом использовании может повредить проект. Каждая команда договаривается, пользоваться ли **rebase** или избегать ее.

**Перенос коммита**

Иногда в ветке может быть много изменений, а сливать одну в другую нельзя. Например, не настало время выкатки изменений, тем не менее, нам может потребоваться перенести один или несколько коммитов из одной ветки в другую. Для этого предназначена команда git cherry-pick.

Давайте создадим коммит в ветке **feature**, который мы потом будем переносить в ветку **master**. Изменим файл readme.txt, добавив в него строку 'Hello, git rebase!' Создаем коммит:

**$ git status**

**$ git add .**

**$ git commit -m 'Adding new line git rebase feature'**

Давайте посмотрим его хэш:

**$ git gg**

Помещаем его в буфер обмена и переключаемся на ветку **master**:

**$ git switch master**

Команда для переноса коммита всегда выполняется в ветке, в которую мы переносим коммит:

**$ git cherry-pick ce268e51**

Выполняем команду **cherry-pick**, указывая ей в качестве параметра **хэш** переносимого коммита.

При попытке слияния у нас возможно возникнет конфликт. Давайте поправим его. Для этого отправляемся в редактор и открываем файл readme.txt **Устраняем конфликт** и отправляемся в консоль:

**$ git status**

Git предлагает нам выполнение команды **cherry-pick**, для этого нужно добавить параметр - **--continue**. Добавляем файл в индекс:

**$ git add .**

И завершаем перенос коммита:

**$ git cherry-pick --continue**

В ветку **master** был перенесен коммит, однако его хэш поменялся. Несмотря на то, что вносимые изменения совпадают с изменениями исходного, это новый коммит со своим собственным хэшем.

**Объединение коммитов**

Иногда коммитов, которые необходимо перенести из одной ветки в другую, довольно много. Переносить их поодиночке командой **cherry-pick** может быть утомительно.

В этом случае несколько коммитов ветки предварительно объединяют в один, который затем и переносят в другую ветку при помощи команды **cherry-pick**.

Для операции объединения нескольких коммитов в один используется команда **git rebase -i**. Давайте переключимся на ветку **feature**:

**$ git checkout feature**

Создадим еще какой-то коммит. Добавляем в файл readme.txt строку 'Hello, squash!' и создаем коммит:

**$ git status**

**$ git add .**

**$ git commit -m 'Adding new line'**

Теперь выполняем команду **git cherry**, чтобы выяснить, насколько ветка **feature** **отличается** **от ветки** **master**:

**$ git cherry -v master**

Как видно, она отличается на три коммита. Мы знаем, что сослаться на последний коммит ветки можно при помощи готового тэга **HEAD**:

**$ git show HEAD**

Если нам потребуется коммит, который предшествует последнему, можно через тильду добавить значение 1:

**$ git show HEAD~1**

Увеличивая цифру после тильды, можно перебрать все коммиты ветки:

**$ git show HEAD~2**

Давайте снова выполним команду **git cherry**:

**$ git cherry -v master**

Так как разница составляет три коммита, начальный коммит — **HEAD~3**:

**$ git show HEAD~3**

Для того, чтобы объединить коммиты, начиная с этого коммита до последнего, выполняем команду **git rebase -i**:

**$ git rebase -i HEAD~3**

Открывается файл с перечислением всех коммитов и описанием, что можно с ними сделать. Если перед каждым коммитом мы оставляем значение **pick — они остаются**, если **squash — объединяются**. У вас должен остаться хотя бы один коммит, помеченный как **pick**. Сохраняем файл. Как результат, git склеивает коммиты в один. Давайте в этом убедимся:

**$ git cherry -v master**

У нас ветка feature отличается от master только одним коммитом. Если мы сейчас попробуем отправить ветку на сервер, у нас ничего не получится:

**$ git push origin feature**

Мы воспользовались перезаписыванием истории проекта при помощи команды **git rebase**, поэтому нам потребуется перезаписать ветку при с использованием параметра **-f** или **--force**:

**$ git push origin feature --force**

Или в сокращенном варианте — просто **-f**:

**$ git push origin feature -f**

Несколько коммитов объединены в один. Теперь можно переносить коммит в другую ветку при помощи команды **git cherry pick** или просто смержить ветку.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**pwd** - выводит точный путь до каталога, в котором мы находимся.

**ls** - выводит в командную строку список файлов директории/каталога в которой мы сейчас находимся.

**ls -la** - будут выведены в том числе скрытые файлы, которые в UNIX-подобных операционных системах начинаются с точки.

**cd /User** и т.д. - переход между директориями/каталогами (в данном случае будет переход в каталог User).

**cd ..** - возвращает в предыдущий каталог (на уровень выше получается).

**mkdir ГГ** - создает новый каталог/директорию в том месте, где мы сейчас находимся (в данном случае создаст папку ГГ).

**touch hello.txt** - создаст файл hello.txt в том месте, где мы сейчас находимся.

**echo `Привет`** - выведет слово "Привет" в командную строку.

**echo 'Hello, world!' > hello.txt** - Запишет Hello, world! в файл Hello.txt если, таким же образом, повторно ввести данные, то прошлые будут перезаписаны.

**echo 'Hello, world!' >> hello.txt** - знак >> используется для добавления данных в конец файла, не перезаписывая прошлые данные.

**cat hello.txt** - выведет содержимое файла hello.txt в командную строку.

**unlink hello.txt** - удалит файл hello.txt