Инструкция Github.

Содержание:

1. ВВЕДЕНИЕ
   1. О системе контроля версий
   2. Локальные системы контроля версий
   3. Централизованные системы контроля версий
   4. Распределенные системы контроля версий
2. ОСНОВНЫЕ ОПЕРАЦИИ
   1. Репозиторий
   2. Лес git’a
   3. Ветвление
   4. Создание новой ветки
   5. Переключение между ветками
   6. Слияние веток
   7. Фичи git’a

1. Введение

Эта глава о том, как начать работу с Git. Вначале изучим основы систем контроля версий, затем перейдём к тому, как запустить Git на вашей ОС и окончательно настроить для работы. В конце главы вы уже будете знать, что такое Git и почему им следует пользоваться, а также получите окончательно настроенную для работы систему.

* 1. О системе контроля версий

Что такое «система контроля версий» и почему это важно? Система контроля версий — это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии. Для контроля версий файлов в этой книге в качестве примера будет использоваться исходный код программного обеспечения, хотя на самом деле вы можете использовать контроль версий практически для любых типов файлов.

Если вы графический или web-дизайнер и хотите сохранить каждую версию изображения или макета (скорее всего, захотите), система контроля версий (далее СКВ) — как раз то, что нужно. Она позволяет вернуть файлы к состоянию, в котором они были до изменений, вернуть проект к исходному состоянию, увидеть изменения, увидеть, кто последний менял что-то и вызвал проблему, кто поставил задачу и когда и многое другое. Использование СКВ также значит в целом, что, если вы сломали что-то или потеряли файлы, вы спокойно можете всё исправить. В дополнение ко всему вы получите всё это без каких-либо дополнительных усилий.

* 1. Локальные системы контроля версий

Многие люди в качестве метода контроля версий применяют копирование файлов в отдельную директорию (возможно даже, директорию с отметкой по времени, если они достаточно сообразительны). Данный подход очень распространён из-за его простоты, однако он невероятно сильно подвержен появлению ошибок. Можно легко забыть, в какой директории вы находитесь, и случайно изменить не тот файл или скопировать не те файлы, которые вы хотели.

Для того, чтобы решить эту проблему, программисты давным-давно разработали локальные СКВ с простой базой данных, которая хранит записи о всех изменениях в файлах, осуществляя тем самым контроль ревизий.



Рисунок 1. Локальный контроль версий.

Одной из популярных СКВ была система RCS, которая и сегодня распространяется со многими компьютерами. RCS хранит на диске наборы патчей (различий между файлами) в специальном формате, применяя которые она может воссоздавать состояние каждого файла в заданный момент времени.

* 1. Централизованные системы контроля версий

Следующая серьёзная проблема, с которой сталкиваются люди, — это необходимость взаимодействовать с другими разработчиками. Для того, чтобы разобраться с ней, были разработаны централизованные системы контроля версий (ЦСКВ). Такие системы, как CVS, Subversion и Perforce, используют единственный сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища. Применение ЦСКВ являлось стандартом на протяжении многих лет.



Рисунок 2. Централизованный контроль версий.

Такой подход имеет множество преимуществ, особенно перед локальными СКВ. Например, все разработчики проекта в определённой степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще администрировать ЦСКВ, чем оперировать локальными базами данных на каждом клиенте.

Несмотря на это, данный подход тоже имеет серьёзные минусы. Самый очевидный минус — это единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений, над которыми работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные бэкапы отсутствуют, вы потеряете всё — всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков. Локальные СКВ страдают от той же самой проблемы: когда вся история проекта хранится в одном месте, вы рискуете потерять всё.

* 1. Распределённые системы контроля версий

Здесь в игру вступают распределённые системы контроля версий (РСКВ). В РСКВ (таких как Git, Mercurial, Bazaar или Darcs) клиенты не просто скачивают снимок всех файлов (состояние файлов на определённый момент времени) — они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, умрёт, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных.



Рисунок 3. Распределённый контроль версий.

Более того, многие РСКВ могут одновременно взаимодействовать с несколькими удалёнными репозиториями, благодаря этому вы можете работать с различными группами людей, применяя различные подходы единовременно в рамках одного проекта. Это позволяет применять сразу несколько подходов в разработке, например, иерархические модели, что совершенно невозможно в централизованных системах.

1. ОСНОВНЫЕ ОПЕРАЦИИ

Начнем, естественно, с загрузки. Надеемся, какая у вас операционная система, вы знаете. И сразу предупредим новичков: не путайте git и GitHub — это разные вещи. Нас интересует именно git, а GitHub (или ему подобные сервисы вроде Bitbucket или GitLab) — это по сути хостинг для проектов, использующих git.

* 1. Репозиторий

Итак, вот у вас уже есть git. Теперь нужно создать хранилище версий для него. Запомните, это хранилище называется репозиторий (англ. repository) — при случае можете вставить где-нибудь это словечко. В зависимости от того, какая у вас оболочка, соответствующей командой создайте новую директорию, откройте ее (в командной строке, она же оболочка, а не проводником или чем-то подобным) и выполните:

git init

Все, локальный репозиторий в этой папке создан. То, что здесь сейчас хранится, будет бекапом, поэтому, чтобы его не испортить, создадим рабочую копию (англ. check out) локальной версии:

git clone [url]

Где [url] — это путь до клонируемого репозитория. Мы разбираем сейчас случай, когда вы создаете рабочую копию собственного репозитория, поэтому в качестве [url] здесь вам нужно указать путь до директории, для которой мы выполняли git init.

Но если вы крутой чувак и уже работаете с удаленным сервером, то вот такая команда будет для вас в самый раз:

git clone username@host:/path/to/repository

* 1. Лес git’а

Немного теории. Git в своей работе управляет тремя структурами, которые называются деревьями. Первое — это рабочая директория, в ней хранятся файлы, с которыми вы прямо сейчас работаете. Ну, она ж рабочая, логично. Второе — это Index, этакий чек-поинт, который позволяет вам вносить изменения и ничего не портить. А третье — это HEAD, который указывает на последний сделанный вами коммит. (Чтобы вы не запутались в терминологии: коммит (англ. commit) — это сохранение состояния проекта в репозиторий. Короче, считайте, новая версия.)

Так вот, чтобы вы не заблудились в этих трех соснах, запомните две крутые команды: add и commit. Они позволят вашей работе спокойно бродить по git’у, сохраняясь, куда надо. Если вы придумали что-то гениальное и тут же внесли изменение в рабочую копию проекта, то не спешите сразу коммитить! Сначала испытайте в Index’е, для этого выполните:

git add [имя\_файла]

если вы внесли изменение только в один файл, или

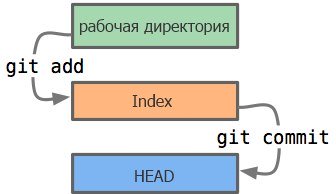
git add \*

если вы хорошо потрудились поменяли сразу кучу исходников. Изменения положительны? Хорошо потестили? Тогда скорее коммитить:

git commit -m "Commit message"

Вы, конечно же, плохой парень и не оставляете комментариев в коде. Но git — другое дело. Не ленитесь оставлять поясняющие сообщения: будьте уверены, вам хватит других проблем, кроме как разбираться, что же поменялось в этом коммите по сравнению с прошлой версией.

Вот вам, кстати, поясняющая картинка:

[](https://s3.tproger.ru/uploads/2016/06/EXcwUk86PWs.jpg)

Теперь файл(-ы) прочно обосновались в HEAD вашей рабочей локальной копии. Оттуда их не выгнать, но в вашем удаленном репозитории их все еще нет. Давайте сунем их еще и туда! Используйте:

git push origin master

Только вместо master напишите название нужной ветки. Ах да, вы же еще не знаете, что такое ветки. Ну ладно, пока что запомните это место, а когда прочитаете про ветвление, вернетесь сюда.

Ах да, для крутых чуваков, работающих с серверами (как раз тут уместно говорить про GitHub, например), команда будет такой:

git remote add origin [сервер]

* 1. Ветвление

По-английски эта штука зовется branching — лучше как следует вникните в этот вопрос и почитайте про ветвление подробнее, я вас с ним только познакомлю. Ветвление используется для одновременной и независимой разработки разных фич (ну, или накопления большего количества багов, ведь исходного кода становится больше). Основной веткой является master — она появляется при создании репозитория. Другие ветки — это песочницы, когда достаточно в них наиграетесь, слейте в единое целое в master. Сейчас поясню, как это делается.

* 1. Создание новой ветки

Вот вы решили проработать какую-нибудь новую фичу. Создайте для нее новую ветку:

git checkout -b [новая\_ветка]

Ах да, фантазия-то у вас, наверное, работает на полную катушку, ну да поумерьте её в деле именования веток: назвать ветку можно только именем, допустимым для переменной в вашем любимом языке.

* 1. Переключение между ветками

Надо сделать перерыв в работе с этой фичей и переключиться на другую ветку? Используйте (если работаете с локальным репозиторием, то указывать его имя не обязательно):

git checkout [репозиторий]/[ветка]

Ну, а если вы уже совсем не хотите с ней работать, то удалите ее совсем:

git branch -d [ветка]

Со своей веткой вы можете творить любые непотребства: ее никто не увидит, пока вы сами ее не пропушите в удаленный репозиторий командой:

git push origin [ветка]

* 1. Слияние веток

Чтобы слить ветку в ту, с которой вы сейчас работаете, используйте:

git merge [ветка]

Но, понятное дело, это все приводит к конфликтам. И это реально проблема. Так что попробуйте исправлять все ручками прямо в директории с репозиторием. Только потом не забудьте пометить, что вы их «слили»:

git add [имя\_файла]

Кстати, ветки можно сравнить:

git diff [одна\_ветка] [другая\_ветка]

Так, теперь приступим к более решительным действиям. Будем обновлять свой репозиторий в соответствии с самым свежим коммитом. Сделать это очень просто (а вот вернуть обратно не очень, поэтому трижды подумайте, прежде чем совершать эту ужасную ошибку):

git pull

Я, конечно, понимаю, что вы слишком круты, чтобы оставлять какие-либо пометки на будущее — все держите в голове — но все-таки рекомендую вам оставлять тэги. И это не моя выдумка, так делают многие:

git tag [tag] [первые\_десять\_символов\_соответствующего\_коммита]

Вы не знаете, какие первые символы у имени нужного коммита? Не беда, смотрите в историю репозитория — его лог:

git log

Там есть куча разных параметров для использования этой полезной штуковины, ну да погуглите их сами. Ах да, кстати, мы уже писали как-то про то как сделать git log более информативным.

* 1. Фичи git’а

Если вы ленивый, и вам не охота по-трупрогерски все писать в оболочке своей ОСи, то можете использовать GUI git’а:

gitk

В источнике найдете еще кучу других GUI-шек.  
Если вам стандартный вывод git’а кажется скучным, раскрасьте его:

git config color.ui true

Ну, и есть еще такая штука — интерактивное индексирование. Когда у вас будет уже достаточно большой проект, то ужать представление index’а в log’е можно будет так:

git add -i

Надеюсь, это руководство поможет вам на ранних этапах не запутаться в работе с git’ом и вы наконец научитесь следить за своими бекапами.