Основы Git

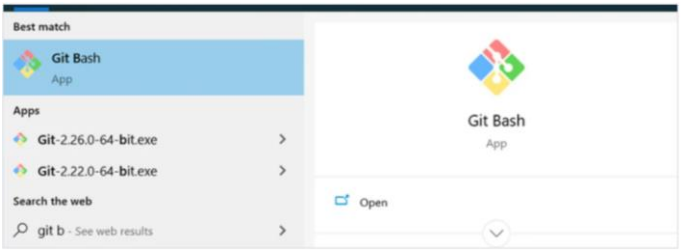
Git — это распределенная система контроля версий нашего кода. Зачем она нам? Для распределенных команд нужна какая-то система управления работы. Нужна, чтобы отслеживать изменения, которые происходят со временем. То есть шаг за шагом мы видим, какие файлы изменились и как. Особенно это важно, когда анализируешь, что было проделано в рамках одной задачи: это дает возможность возвращаться назад. Представим себе ситуацию: был работающий код, всё в нем было хорошо, но мы решили что-то улучшить, там подправить, сям подправить. Все ничего, но такое улучшение поломало половину функционала, сделало невозможным работу. И что дальше? Без Гита нужно было бы часами сидеть и вспоминать, как же все было изначально. А так мы просто откатываемся на коммит назад — и все. Или что делать, если есть два разработчика, которые делают одновременно свои изменения в коде? Без Гита это выглядит так: они скопировали код из оригинала, сделали что нужно. Наступает момент, и оба хотят добавить свои изменения в главную папку. И что делать в этой ситуации?.. Я даже не берусь оценить время, чтоб проделать эту работу. Таких проблем не будет вовсе, если пользоваться Гитом.

Установка Git

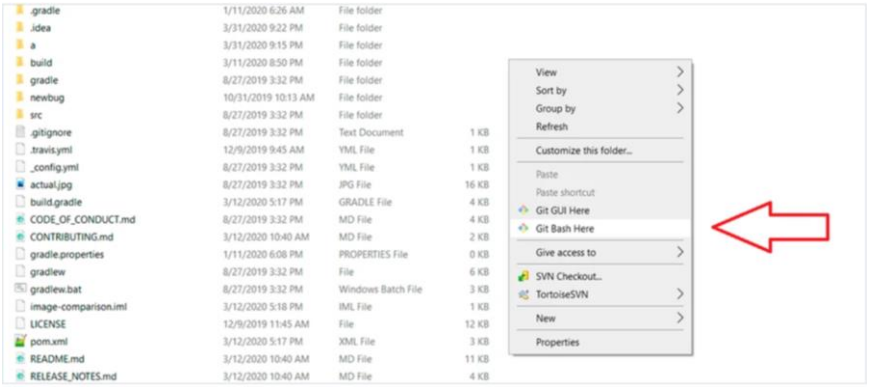
Установим гит на компьютер. Я понимаю, что у всех разные OS, поэтому постараюсь описать для нескольких случаев.

Установка для Windows

Как обычно, нужно скачать exe файл и запустить его. Здесь все просто: жмем на первую ссылку гугла, устанавливаем и всё. Для работы будем использовать bash консоль, которую они предоставляют. Чтобы работать в виндоусе, нужно запустить Git Bash. Вот как он выглядит в меню пуск:



И это уже консоль, в которой можно работать. Чтобы не переходить каждый раз в папку с проектом, чтобы там открыть гит, можно в папке правой кнопкой мыши открыть консоль с нужным нам путем:



Установка для Linux

Обычно git уже установлен и есть в дистрибутивах линукса, так как это инструмент, первоначально написанный для разработки ядра линукса. Но бывают ситуации, когда его нет. Чтобы проверить это, нужно открыть терминал и прописать: git --version. Если будет вразумительный ответ, ничего устанавливать не нужно. Открываем терминал и устанавливаем. Я работаю на Ubuntu, поэтому могу сказать, что писать для нее: sudo apt-get install git. И все: теперь в любом терминале можно пользоваться гитом.

Установка на macOS

Здесь также для начала нужно проверить, есть ли уже гит (смотри выше, как на линуксе). Если все же нет, самый простой путь — это скачать отседова последнюю версию. Если установлен XCode, то гит уже точно будет автоматически установлен.

Настройка гита

У гита есть настройка пользователя, от которого будет идти работа. Это разумная и необходимая вещь, так как когда создается коммит, гит берет именно эту информацию для поля Author. Чтобы настроить имя пользователя и пароль для всех проектов, нужно прописать следующие команды:

git config --global user.name ”Ivan Ivanov”

git config --global user.email ivan.ivanov@gmail.com

Если есть необходимость для конкретного проекта поменять автора (для личного проекта, например), можно убрать --global, и так получится:

git config user.name ”Ivan Ivanov”

git config user.email ivan.ivanov@gmail.com

Немного теории

Чтобы быть в теме, желательно добавить в свое обращение несколько новых слов и действий… А то говорить будет не о чем. Конечно это некий жаргон и калька с английского, поэтому я буду добавлять значения на английском. Какие слова и действия?

- гит репозиторий (git repository);
- коммит (commit);
- ветка (branch);
- смерджить (merge);
- конфликты (conflicts);
- спулить (pull);
- запушить (push);
- как игнорировать какие-то файлы (.gitignore).

И так далее.

Состояния в Гит

У Гита есть несколько состояний, которые нужно понять и запомнить:

- неотслеживаемое (untracked);
- измененное (modified);
- подготовленное (staged);

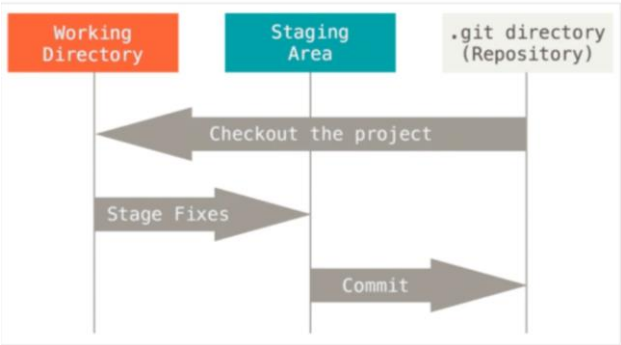
- закомиченное (committed).

Как это понимать?

Это состояния, в которых находятся файлы из нашего кода. То есть, их жизненный путь обычно выглядит так:

1. Файл, который создан и не добавлен в репозиторий, будет в состоянии untracked.
2. Делаем изменения в файлах, которые уже добавлены в гит репозиторий — находятся в состоянии modified.
3. Из тех файлов, которые мы изменили, выбираем только те (или все), которые нужны нам (например, скомпилированные классы нам не нужны), и эти классы с изменениями попадают в состояние staged.
4. Из заготовленных файлов из состояния staged создается коммит и переходит уже в гит репозиторий. После этого staged состояние — пустое. А вот modified еще может что-то содержать.

Выглядит это так (картиночка из официальной доки, так что можно верить)):



Что такое коммит

Коммит — это основной объект в управлении контроля версий. Он содержит все изменения за время этого коммита. Коммиты связаны между с собой как односвязный список. А именно: Есть первый коммит. Когда создается второй коммит, то он (второй) знает, что идет после первого. И таким образом можно отследить информацию. Также у коммита есть еще своя информация, так называемые метаданные:

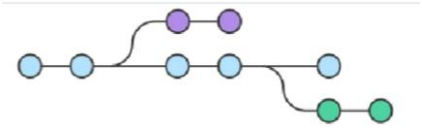
- уникальный идентификатор коммита, по которому можно его найти;
- имя автора коммита, который создал его;
- дата создания коммита;
- комментарий, который описывает, что было сделано во время этого коммита.

Вот как это выглядит:

```
commit 5a37824fa89b4f7650637637e022188d0ffba322
Author: romankh3 <roman.beskrovnyy@gmail.com>
Date:   Thu Feb 6 15:47:58 2020 +0200

    added order.xls file for using drools rules engine.
```

Что такое ветка



Ветка — это указатель какого-то коммита. Так как коммит знает, какой коммит был до него, когда ветка указывает на какой-то коммит, к ней относятся и все те предыдущие. Исходя из этого можно сказать, что веток, указывающих на один и тот же коммит, может быть сколько угодно много. Работа происходит в ветках, поэтому когда создается новый коммит, ветка переносит свой указатель на более новый коммит.

Начало работы с Гитом

Можно работать и только с локальный репозиторием, и с удаленным. Для отработки нужных команд можно воспользоваться только локальным репозиторием. Он хранит всю информацию только локально в проекте в папке .git. Если говорить об удаленном, то вся информация хранится где-то на удаленном сервере: локально хранится только копия проекта, изменения которой можно запушить (git push) в удаленный репозиторий. Здесь и далее будем обсуждать работу с гитом в консоли. Конечно, можно пользоваться какими-то

графическими решениями (например, в Intellij IDEA), но сперва нужно разобраться, какие команды происходят и что они значат.

Работа с гитом в локальном репозитории

Далее я предлагаю вам проделать все те шаги, которые проделал я, в то время как будете читать статью. Это улучшит ваше понимание и усвоение материала. Так что приятного аппетита :) Чтобы создать локальный репозиторий, нужно написать:

```
git init
```

После этого будет создана папка .git в том месте, где находится консоль. .git — это папка, которая хранит всю информацию о гит репозитории. Ее удалять не нужно ;) Далее, добавляются файлы в этот проект, и их состояние становится Untracked. Чтобы посмотреть, какой статус работы на данный момент, пишем:

```
git status
```

Мы находимся в master ветке, и пока мы не перейдем в другую, так все и останется. Таким образом видно, какие файлы изменены, но еще не добавлены в состояние staged. Чтобы добавить их в состояние staged, нужно написать git add. Здесь может быть несколько вариантов, например:

- git add -A — добавить все файлы из состояния в staged;
- git add . — добавить все файлы из этой папки и все внутренних. По сути тоже самое, что и предыдущее;
- git add <имя файла> — добавляет только конкретный файл. Здесь можно пользоваться регулярными выражениями, чтобы добавлять по какому-то шаблону. Например, git add *.java: это значит, что нужно добавить только файлы с расширением java.

Ясно, что первые два варианта простые, а вот с добавлением будет интереснее, поэтому пишем:

```
git add *.txt
```

Чтобы проверить статус, используем уже известную нам команду:

git status

Отсюда видно, что регулярное выражение отработало верно, и теперь test_resource.txt находится в staged состоянии. И, наконец, последний этап (при локальном репозитории, с удаленным будет еще один ;)) — закоммитить и создать новый коммит:

git commit -m “all txt files were added to the project”

Далее есть отличная команда, чтобы посмотреть на историю коммитов в ветке. Воспользуемся ею:

git log

Здесь уже видно, что появился наш первый коммит с текстом, который мы передали. Очень важно понять, что текст, который мы передаем, должен максимально точно определять то, что было проделано за этот коммит. Это в будущем будет помогать множество раз. Пытливый читатель, который еще не уснул, может сказать: а что случилось с файлом GitTest.java? Сейчас узнаем, используем для этого:

git status

Как видим, он так и остался в состоянии untracked и ждет своего часа. А может мы вовсе не хотим его добавлять в проект? Бывает и такое. Далее, чтобы стало интереснее, попробуем изменить наш текстовый файл test_resource.txt. Добавим туда какой-то текст и проверим состояние:

git status

Здесь хорошо видна разница между двумя состояниями — untracked и modified. GitTest.java находится в состоянии untracked, а test_resource.txt находится в modified. Теперь, когда уже есть файлы в состоянии modified, мы можем посмотреть на изменения, которые были произведены над ними. Сделать это можно при помощи команды:

git diff

То есть здесь хорошо видно, что я добавил в наш текстовый файл hello world! Добавляем изменения в текстовом файле и коммитим:

git add test_resource.txt

git commit -m “added hello word! to test_resource.txt”

Чтобы посмотреть на все коммиты, пишем:

git log

Как видим, уже есть два коммита. Таким же образом добавляем и GitTest.java. Теперь без комментариев, просто команды:

git add GitTest.java

git commit -m “added GitTest.java”

git status

Работа с .gitignore

Ясно, что мы хотим хранить только исходный код и ничего другого в репозитории. А что может быть еще? Как минимум, скомпилированные классы и/или файлы, которые создают среды разработки. Чтобы гит их игнорировал, есть специальный файл, который нужно создать. Делаем это: создаем файл в корне проекта с названием .gitignore, и в этом файле каждая строка будет шаблоном для игнорирования. В этом примере гит игнор будет выглядеть так:

```
*.class
target/
*.iml
.idea/
```

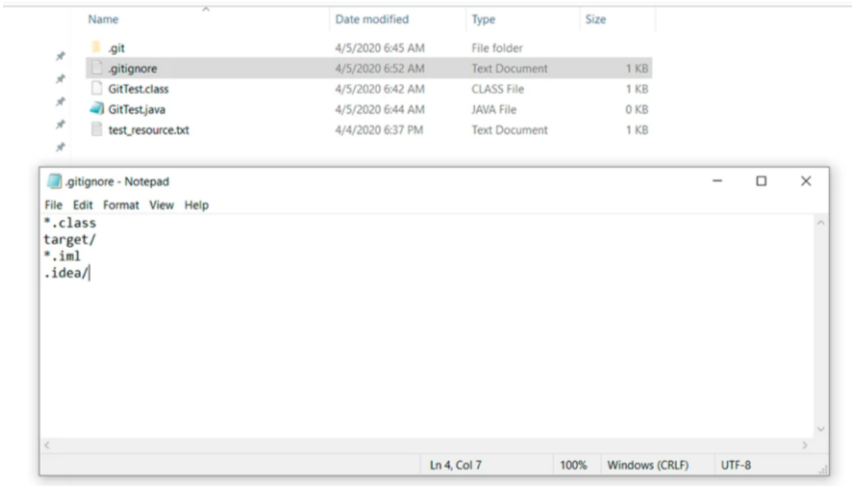
Смотрим теперь:

- первая строка — это игнорирование всех файлов с расширением .class;
- вторая строка — это игнорирование папки target и всего, что она содержит;
- третья строка — это игнорирование всех файлов с расширением .iml;
- четвертая строка — это игнорирование папки .idea.

Попробуем на примере. Чтобы посмотреть как это работает, добавим скомпилированный класс GitTest.class в проект и посмотрим статус проекта:

git status

Ясно, что мы не хотим как-то случайно (если использовать git add -A) добавить скомпилированный класс в проект. Для этого создаем .gitignore файл и добавляем все, что описывалось ранее: Теперь добавим новым коммитом гит игнор в проект:



git add .gitignore

git commit -m “added .gitignore file”

И теперь момент истины: у нас есть в untracked состоянии скомпилированный класс GitTest.class, который мы не хотели добавлять в гит репозиторий. Вот здесь-то и должен заработать гит игнор:

git status

Все чисто) Гит игнору +1)

Работа с ветками и иже с ним

Разумеется, работать в одной ветке неудобно одному и невозможно, когда в команде больше одного человека. Для этого существует ветвление. Как я уже говорил, ветка — это просто подвижный указатель на коммиты. В этой части рассмотрим работу в разных ветках: как смерджить изменения одной ветки в другую, какие могут возникнуть конфликты и многое другое. Чтобы посмотреть список всех веток в репозитории и понять, на какой находишься, нужно написать:

git branch -a

Видно, что у нас только одна ветка master, и звездочка перед ней говорит, что мы находимся на ней. К слову, чтобы узнать, на какой ветке мы находимся, можно воспользоваться и проверкой статуса (git status). Далее есть несколько вариантов создания веток (может их и больше, я использую эти):

- создать новую ветку на основе той, на которой находимся (99% случаев);
- создать ветку на основе конкретного коммита (1%).

Создаем ветку на основе конкретного коммита

Опираться будем на уникальный идентификатор коммита. Чтобы найти его, напишем:

git log

Я выделил коммит с комментарием “added hello world…”. У него уникальный идентификатор — “6c44e53d06228f888f2f454d3cb8c1c976dd73f8”. Я хочу создать ветку development начиная с этого коммита. Для этого напишу:

git checkout -b development 6c44e53d06228f888f2f454d3cb8c1c976dd73f8

Создается ветка, в которой будут только первые два коммита из ветки master. Чтобы проверить это, мы сперва убедимся, что перешли в другую ветку и посмотрим на количество коммитов ней:

git status

git log

И правда: получилось, что у нас два коммита. Кстати, интересный момент: в этой ветке еще нет файла .gitignore, поэтому наш скомпилированный файл (GitTest.class) теперь подсвечивается в untracked состоянии. Теперь можем провести еще раз ревизию наших веток, написав:

git branch -a

Видно, что есть две ветки — master и development — и сейчас стоим на development.

Создаем ветку на основе текущей

Второй способ создания ветки — создание на основе другой. Я хочу создать ветку на основе master ветки, поэтому нужно сперва перейти на нее, а уже следующим шагом — создать новую. Смотрим:

- git checkout master — переходим на ветку master;
- git status — проверяем, точно ли на мастере.

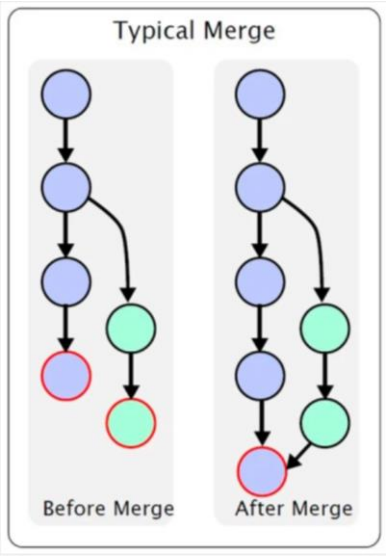
Вот здесь видно, что мы перешли на master ветку, здесь уже работает гит игнор, и скомпилированный класс уже не светится как untracked. Теперь создаем новую ветку на основе master ветки:

```
git checkout -b feature/update-txt-files
```

Если есть сомнения, что эта ветка будет не такой же, как и master, можно это легко проверить, написав git log и посмотреть на все коммиты. Там их должно быть четыре.

Резолвим конфликты

Прежде чем разобраться с тем, что такое конфликт, нужно поговорить о слиянии (смерджывании) одной ветки в другую. Вот такой картинкой можно показать процесс, когда одну ветку мерджат в другую:



То есть, есть главная ветка. От нее в какой-то момент создают второстепенную, в которой происходят изменения. Как только работа сделана, нужно слить одну ветку в другую. Я не буду описывать разные особенности: я

хочу донести в рамках этой статьи только понимание, а уже детали узнаете сами, если будет нужно. Так вот, на нашем примере, мы создали ветку feature/update-txt-files. Как написано в имени ветки — обновим текст.

Теперь нужно создать под это дело новый коммит:

```
git add *.txt
git commit -m “updated txt files”
git log
```

Теперь, если мы хотим смерджить feature/update-txt-files ветку в master, нужно перейти в master и написать git merge feature/update-txt-files:

```
git checkout master
git merge feature/update-txt-files
git log
```

Как результат — теперь и в мастер ветке есть коммит, который был добавлен в feature/update-txt-files. Эта функциональность добавлена, поэтому можно удалить фиче (feature) ветку. Для этого напишем:

```
git branch -D feature/update-txt-files
```

Пока понятно, да? Усложняем ситуацию: теперь допустим, что опять нужно изменить txt файл. Но теперь еще и в мастере этот файл будет изменен также. То есть он будет параллельно изменяться, и гит не сможет понять что нужно делать в ситуации, когда мы захотим смерджить в master ветку новый код. Поехали! Создаем новую ветку на основе master, делаем изменения в text_resource.txt и создаем коммит под это дело:

```
git checkout -b feature/add-header
... делаем изменения в файле
```

git add *.txt

git commit -m “added header to txt”

Переходим на master ветку и также обновляем этот текстовый файл в той же строке, что и фиче ветка:

git checkout master

… обновили test_resource.txt

git add test_resource.txt

git commit -m “added master header to txt”

И теперь самый интересный момент: нужно смерджить изменения из feature/add-header ветки в master. Мы находимся в мастер ветке, поэтому нужно только написать:

git merge feature/add-header

Но мы получим результат с конфликтом в файле test_resource.txt. И здесь мы можем видеть, что гит не смог самостоятельно решить, как смерджить этот код и говорит, что нужно вначале разрезолвить конфликт, а уже потом сделать коммит. Ок, открываем в текстовом редакторе файл, в котором конфликт, и видим:Чтобы понять, что здесь сделал гит, нужно вспомнить, что мы где писали, и сравнить:

1. между “<<<<<<< HEAD” и “=======” находятся изменения мастер, которые были в этой строке в мастер ветке.
2. между “=======” и “>>>>>>> feature/add-header” находятся изменения, которые были в feature/add-header ветке.

Таким образом гит показывает, что в этом месте он не смог понять, как слить воедино этот файл, разделил этот участок на две части из разных веток и предложил решить нам самим. Хорошо, твердою волей решаю убрать все, оставить только слово header:

Посмотрим на статус изменений, описание будет несколько другим. Будет не modified состояние, а Unmerged. Так что смело можно было добавить пятое состояние… Но я думаю, что это излишне, посмотрим:

```
git status
```

Убедились, что это другой случай, необычный. Продолжаем:

```
git add *.txt
```

В описании можно заметить, что предлагают написать только git commit. Слушаем и пишем:

```
git commit
```

И все: таким образом мы сделали это — разрезолвили конфликт в консоли. Конечно, в средах разработки можно это сделать немного проще, например, в Intellij IDEA все настроено так хорошо, что можно выполнять все необходимые действия в ней. Но среда разработки делает много чего “под капотом”, и мы зачастую не понимаем, что именно там произошло. А когда нет понимания, тогда могут возникнуть и проблемы.

Работа с удаленными репозиториями

Последний шаг — разобраться еще с несколькими командами, которые нужны для работы с удаленным репозиторием. Как я уже говорил, удаленный репозиторий — это какое-то место, где хранится репозиторий и откуда можно его клонировать. Какие бывают удаленные репозитории? Примеров тьма:

- GitHub — это крупнейшее хранилище для репозиториев и совместной разработки. Я уже описывал его в предыдущих статьях. Подписывайтесь на мой гитхаб аккаунт. Я часто выставляю там свои наработки в тех сферах, которые изучаю во время работы.
- GitLab — веб-инструмент жизненного цикла DevOps с открытым исходным кодом, представляющий систему управления репозиториями кода для Git с собственной вики, системой отслеживания ошибок, CI/CD пайплайн и другими функциями. После новости о том, что Microsoft купила GitHub, некоторые разработчики продублировали свои наработки в GitLab.
- BitBucket — веб-сервис для хостинга проектов и их совместной разработки, основанный на системе контроля версий Mercurial и Git. Одно время имел большое преимущество перед GitHub в том, что у него были бесплатные приватные репозитории. В прошлом году GitHub также открыл эту возможность для всех бесплатно.
- И так далее…

Первое, что нужно сделать в работе с удаленным репозиторием — клонировать проект себе в локальный. Для этого дела я экспортировал проект, который мы делали локально, и теперь каждый его может себе клонировать, написав:

```
git clone https://github.com/romankh3/git-demo
```

Теперь локально есть полная копия проекта. Чтобы быть уверенным, что локально находится последняя копия проекта, нужно, как говорится, спулить данные, написав:

```
git pull
```

В нашем случае сейчас ничего не изменилось удаленно, поэтому и ответ: Already up to date. Но если я внесу какие-то изменения в удаленном репозитории, локальный обновится после того, как мы их спулим. И, наконец, последняя команда — запушить данные на удаленный репозиторий. Когда мы локально что-то сделали и хотим это передать на удаленный репозиторий, нужно сперва

создать новый коммит локально. Для этого добавим в наш текстовый файл еще что-нибудь.

Теперь уже обыденная для нас вещь — создаем коммит под это дело:

```
git add test_resource.txt
git commit -m “prepated txt for pushing”
```

И теперь команда, чтобы отправить это на удаленный репозиторий:

```
git push
```