Кемеровский государственный университет

Институт фундаментальных наук

**09.03.03 Прикладная информатика в экономике**

Дисциплина: Программирование

**Отчет.** Работа в GIT

Выполнила: Мазанова Ю.Н.

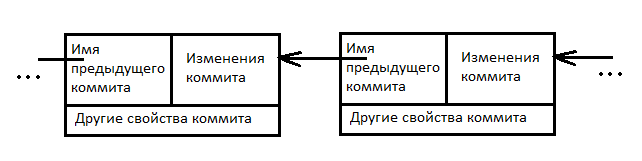
студентка 1-го курса

группы ПИ-211(2)

Преподаватель: Бурмин Л.Н.

Кемерово 2021

**Коммит**

Можно сказать, что коммит это основной объект в любой системе управления версиями. В нем содержится описание тех изменений, которые вносит пользователь в код приложения. В Git коммит состоит из нескольких так называемых объектов. Для простоты понимания можно считать, что коммиты это односвязный список, состоящий из объектов, в которых содержаться измененные файлы, и ссылка на предыдущий коммит.

У коммита есть и другие свойства. Например, дата коммита, автор, комментарий к коммиту и т.п.

В качестве комментария обычно указывают те изменения, которые вносит этот коммит в код, или название задачи, которую он решает.

Git это распределенная система управления версиями. Это значит, что у каждого участника проекта есть своя копия репозитория, которая находиться в папке “. git”, которая расположена в корне проекта. Именно в этой папке хранятся все коммиты и другие объекты Git. Когда вы работаете с Git, он в свою очередь работает с этой папкой.

Завести новый репозиторий очень просто, это делается командой

git init

Таким образом у вас получиться новый пустой репозиторий. Если вы хотите присоединиться к разработке уже имеющегося проекта, то вам нужно будет скопировать этот репозиторий в свою локальную папку с удаленного репозитория. Делается это так:

git clone <url удаленного репозитория>

После чего в текущей папке появляется директория. git в которой и будет содержаться копия удаленного репозитория.

Существует несколько основных областей в которых находиться код.

**Рабочая директория** – это файлы в корне проекта, тот код с которым вы работаете.

**Локальный репозитори**й — она же директория “.git”. В ней хранятся коммиты и другие объекты.

**Удаленный репозиторий** – тот самый репозиторий который считается общим, в который вы можете передать свои коммиты из локального репозитория, что бы остальные программисты могли их увидеть. Удаленных репозиториев может быть несколько, но обычно он бывает один.

Есть еще одна область, с пониманием которой обычно бывают проблемы. Это так называемая область подготовленных изменений (staging area). Дело в том, что перед тем, как включить какие-то изменения в коммит, нужно вначале отметить что именно вы хотите включить в этот коммит. В своей рабочей директории вы можете изменить несколько файлов, но в один конкретный коммит включать не все эти изменения.  
В целом работа с гитом выглядит так: вы меняете файлы в своей рабочей директории, затем добавляете эти изменения в staging area используя команду

git add <имя/файла>

При этом можно использовать маски со звездочкой.

Потом вы делаете коммит в свой локальный репозиторий

git commit –m “Комментарий к коммиту”

Когда коммитов накопиться достаточно много, чтобы ими можно было поделиться, вы выполняете команду

git push

После чего ваши коммиты уходят в удаленный репозиторий.

Если нужно получить изменения из удаленного репозитория, то нужно выполнить команду

git pull

После этого, в вашем локальном репозитории появятся те изменения, которые были отправлены другими программистами.

Код в рабочей области проекта образуется применением тех изменений, которые содержаться в коммитах. У каждого коммита есть свое имя, которое представляет собой результат хеш функции sha-1 от содержимого самого коммита.

Просмотреть коммиты можно при помощи команды

git log

Формат ответа этой команды по дефолту не очень удобен. Вот такая команда выведет ответ в более читаемом виде

git log --pretty=format:"%H [%cd]: %an - %s" --graph --date=format: %c

Что бы закончить просмотр нужно нажать на клавишу q

Посмотреть, что находиться в рабочей директории и staging area можно командой

git status

Рабочую директорию можно переключить на предыдущее состояние выполнив команду

git checkout <hash коммита>

Только перед тем как это делать выполните git status и убедитесь, что у вас нет никаких локальных и не зафиксированных изменений. Иначе Git не поймет, как ему переключаться. git status подскажет вам что можно сделать с локальными изменениями что бы можно было переключиться. Этого правила следует придерживаться и при всяких других переключениях рабочей области.

**Ветка**

Ветка в Git это подвижный указатель на один из коммитов. Обычно ветка указывает на последний коммит в цепочке коммитов. Ветка берет свое начало от какого-то одного коммита. Визуально это можно представить вот так.

Сделать новую ветку и переключиться на нее можно выполнив команд

git pull

git checkout –b <имя новой ветки>

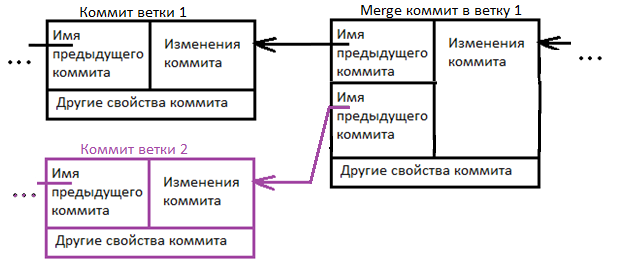
Просто сделать ветку, не переключаясь на нее можно командой

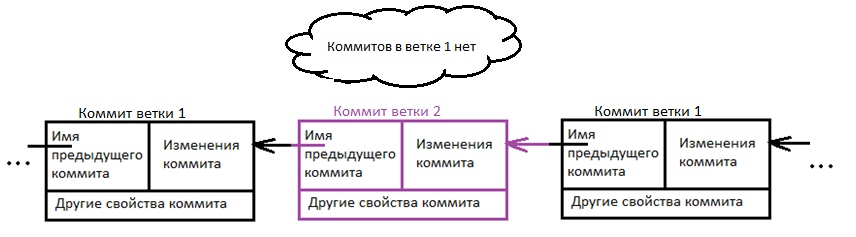
git branch <имя ветки>

переключиться на ветку

git checkout <имя ветки>

Важно понимать, что ветка берет свое начало не от ветки, а от последнего коммита который находиться в той ветке, в которой вы находились.

Ветка обычно заканчивается специальным merge коммитом, который говорит, что ветку нужно объединить с какой-то другой веткой. В merge коммите содержатся две ссылки на два коммита которые объединяются в одну ветку.

Существует другая ситуация при объединении веток, в которой merge может произойти без merge commit. Дело в том, что если в одной из веток не произошло никаких изменений, то необходимость в merge commit с двумя предками отпадает. В таком случае, при слиянии веток, Git просто сделает пометку о том, что дальше будут идти коммиты той ветки с которой эта ветка была объединена. Такая схема merge называется слияние-перемотка (fast-forward merge), визуально это можно представить вот так.

Во всех этих случаях, после того как ветка объединяется с другой веткой, все коммиты сделанные в ней, попадают в ветку, с которой она была объединена. Так же важно понимать, что merge это не двунаправленная операция. Если смержить ветку задачи в мастер ветку, то в мастер ветке появится код, который находился в ветке задачи, а в ветке задачи не появиться новый код из мастера ветки. Если нужно что бы это произошло, нужно смержить мастер ветку в ветку задачи.

Что бы смержить одну ветку в другую нужно вначале переключиться на ту ветку, в которую вы хотите смержить

git checkout <имя ветки>

Потом получить последние изменения сделанные в этой ветке выполнив

git pull

Затем выполнить команду

git merge <имя ветки>

Так выглядит работа с ветками в общих чертах.

Обратите внимание на то, что перед тем как заводить новую ветку нужно выполнить git pull. Делается это по нескольким причинам.

Другой программист мог изменить код, в том числе внести такие изменения, которые повлияют на решение задачи, для которой вы заводите новую ветку. Эти изменения могут вам пригодиться при решении своей задачи.

Из-за этих изменений вы можете получить конфликт при мерже.

Больше шанс что у вас получится merge commit. Это не так плохо, как два предыдущих пункта. Но если можно избежать лишнего коммита, то почему бы этого не сделать?

**Популярные схемы работы с ветками в Git**

Теперь можно описать популярные схемы работы с ветками в гите.  
Ветки нужны для того, чтобы программисты могли вести совместную работу над проектом и не мешать друг другу при этом. При создании проекта, Git создает базовую ветку. Она называется master веткой. Она считается центральной веткой, т.е. в ней содержится основной код приложения.

**Классическая схема работы с ветками**

Обычно перед тем как взяться за решение какой-то задачи, программист заводит новую ветку от последнего рабочего коммита мастер ветки и решает задачу в этой новой ветке. В ходе решения он делает ряд коммитов, после этого тестирует код непосредственно в ветке задачи. А после того как задача решена, делают merge обратно в мастер ветку. Такую схему работы часто используют с юнит тестами и автоматизированным деплоем. Если юнит тесты будут покрывать весь код, то можно настроить деплой так, что вначале будут прогоняться все тесты в ветке задачи. А после этого, если они прошли успешно, будет происходить merge и деплой. При такой схеме можно добиться полной автоматизации при тестировании и деплои.

**Именная ветка**

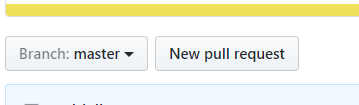
Неопытные программисты заводят себе именную ветку и работают всегда в ней. Они решают по одной задачи за раз, и когда заканчивают решение одной из задач, делают новый Pull запрос через Web интерфейсе (об этом чуть ниже). Недостаток этого подхода в том, что так можно решать только одну задачу и нельзя быстро переключиться на решение другой задачи. Еще один недостаток в том, что ветки так со временем будут все сильнее расходиться и код в ветке программиста рано или поздно устареет относительно мастер ветки и его придется обновить. Для этого можно либо смержить мастер ветку в ветку программиста, либо завести новую ветку для этого программиста от последнего рабочего состояния в мастер ветке. Правда к тому времени, как это произойдет программист уже может освоить гит в достаточной мере что бы перейти на “классическую” схему работы. Таким образом эта схема имеет место быть для неопытных пользователей Git.

**Схема с dev веткой**

Другая схема очень похожа на классическую, только в ней помимо мастера ветки есть еще девелоперская ветка, которая деплоится на тестовый сервер. Такую ветку обычно называют dev. Схема работы при этом такая. Программист перед выполнением новой задачи заводит для нее ветку от последнего рабочего состояния в мастер ветке. Когда он заканчивает работу над задачей, то мержит ветку задачи в dev ветку самостоятельно. После этого, совместными усилиями задача тестируется на тестовом сервере вместе с остальными задачами. Если есть ошибки, то задачу дорабатывают в той же ветке и повторно мержат с dev веткой. Когда тестирование задачи заканчивается, то ВЕТКУ ЗАДАЧИ мержат с мастер веткой. Важно заметить, что в этой схеме работы с мастер веткой нужно мержить ветку задачи, а не dev ветку. Ведь в dev ветке будут содержаться изменения, сделанные не только в этой задаче, но и в других и не все эти изменения могут оказаться рабочими. Мастер ветка и dev ветка со временем будут расходиться, поэтому при такой схеме работы периодически заводят новую dev ветку от последнего рабочего состояния мастер ветки. Недостатком этого подхода является избыточность, по сравнению с классической схемой. Такую схему работы с ветками часто используют если в проекте нет автоматизированных тестов и все тестирование происходит вручную на сервере разработки.

Так же следует отметить что эти схемы работы можно комбинировать между собой, если в этом есть какая-то необходимость.

**Pull запросы**

С этим понятием имеется путаница. Дело в том, что в Git есть две совершенно разные вещи, которые можно назвать Pull запросом. Одна из них, это консольная команда git pull. Другая это кнопка в web интерфейсе репозитория. На github.com она выглядит вот так

Про эту кнопку и пойдет речь дальше.

Если программист достаточно опытный и ответственный, то он обычно сам сливает свой код в мастер ветку. В противном случае программист делает так называемый Pull запрос. Pull запрос — это по сути дела запрос на разрешение сделать merge. Pull запрос можно сделать из web интерфейса Git, или при помощи команды git request-pull. После того как Pull запрос создан, остальные участники могут увидеть это, просмотреть тот код, который программист предлагает внести в проект, и либо одобрить этот код либо нет. Merge через pull запросы имеет свои плюсы и минусы. Минус в том, что для тесной команды опытных программистов такой подход будет лишним. Это будет только тормозить работу и вносить в нее оттенки бюрократии.

С другой стороны, если в проекте есть не опытные программисты, которые могут сломать код, то Pull запросы могут помочь избежать ошибок, и быстрее обучить этих программистов наблюдая за тем какие изменения они предлагают внести в код.  
Так же Pull запросы подходят для широкого сообщества программистов, работающих с открытым исходным кодом. В этом случае нельзя заранее сказать что-то о компетенции таких разработчиков и о том, что они хотят изменить в коде.

**Конфликты**

Конфликты возникают при мердже веток если в этих ветках одна и та же строка кода была изменена по-разному. Тогда получается, что Git не может сам решить какое из изменений нужно применить и он предлагает вручную решить эту ситуацию. Это замедляет работу с кодом в проекте. Избежать этого можно разными методами.

Например, можно распределять задачи так, чтобы связанные задачи не выполнялись одновременно различными программистами.

Другой способ избежать этого, это договориться о каком-то конкретном стиле кода. Тогда программисты не будут менять форматирование кода и вероятность того, что они изменят одну и ту же строчку станет ниже.

Еще один хороший совет, который поможет вам избежать конфликтов при работе в команде, это вносить минимум изменений в код при решении задач. Чем меньше строчек вы поменяли, тем меньше вероятность что вы измените ту же самую строку что и другой программист в другой задаче.

После того, как в мастер ветке достигается состояние, которое можно считать стабильным оно отмечается тегом с версией этого состояния. Это и есть то что называют версией программы.

Делается это вот так

git tag -a v1.0

Что бы передать ветки в удаленный репозиторий нужно выполнить команду

git push –tags

Теги удобны еще и тем, что можно легко переключиться на то состояние кода которое отмечено тегом. Делается это с помощью все той же команды

git checkout <имя тега>

Различные системы деплоя и автоматизированной сборки используют теги для идентификации того состояния, которое нужно задеплоить или собрать. Так сделано потому, что если мы будем собирать или деплоить код последней версии, то есть риск, что какой-то другой программист в этот момент внесет какие-то изменения в мастер ветку, и мы соберем не то что хотели. К тому же так будет проще переключаться между рабочими и проверенными состояниями проектов.