Отчет по лабораторной работе №1

Работа с git

Евсеева Дарья Олеговна

12 февраля, 2022

Содержание

# Цель работы

Целью данной работы является ознакомление с системой контроля версий git и ее основными функциями и изучение основных команд git.

# Задание

Необходимо выполнить ряд команд git для ознакомления с основными функциями системы контроля версий git. Требуется создать проект и поработать с выполнением коммитов, внесением и отменой изменений, также освоить создание веток и их изменение, слияние и сброс, и осуществить работу с удаленным репозиторием и его ветками с помощью клонированного репозитория.

# Теоретическое введение

Система контроля версий git — это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии.

Описание основных команд git

|  |  |
| --- | --- |
| Имя команды | Описание команды |
| git add | Добавляет содержимое рабочей директории в индекс для последующего коммита |
| git status | Показывает состояния файлов в рабочей директории |
| git commit | Берет все данные, добавленные в индекс, и сохраняет их слепок во внутренней БД |
| git reset | Используется для отмены изменений. Изменяет указатель HEAD и, опционально, состояние индекса |
| git rm | Используется для удаления файлов из индекса и рабочей директории |
| git mv | Позволяет переместить файл |
| git clean | Используется для удаления мусора из рабочей директории |
| git branch | Используется для перечисления веток, их создания, удаления или переименования |
| git checkout | Используется для переключения веток и выгрузки их содержимого в рабочую директорию |
| git merge | Используется для слияния одной или нескольких веток в текущую |
| git log | Используется для просмотра истории коммитов |
| git tag | Позволяет задать постоянную метку на какой-либо момент в истории проекта |
| git fetch | Связывается с удаленным репозиторием и забирает из него все изменения |
| git pull | Забирает изменения из удаленного репозитория и пытается слить их с текущей веткой |
| git push | Связывается с удаленным репозиторием и передает в него локальные изменения, которые в нем отсутствуют |
| git remote | Служит для управления списком удаленных репозиториев |

# Выполнение лабораторной работы

## 1.1 Подготовка

**1.1.1 Установка имени и электронной почты**

При первой установке git необходимо задать свое имя и электронную почту. Это можно сделать с помощью следующих команд:

git config --global user.name "Your Name"  
git config --global user.email "your\_email@whatever.com"

В нашей системе git уже был установлен, поэтому сразу переходим к разделу окончания строк.

**1.1.2 Параметры установки окончаний строк**

Настройка core.autocrlf с параметрами true и input делает все переводы строк текстовых файлов в главном репозитории одинаковыми. core.autocrlf true - git автоматически конвертирует CRLF->LF при коммите и обратно LF->CRLF при выгрузке кода из репозитория на файловую систему (используют в Windows). core.autocrlf input - конвертация CRLF в LF только при коммитах (используют в Mac/Linux).

Если core.safecrlf установлен в true или warm, git проверяет, если преобразование является обратимым для текущей настройки core.autocrlf. core.safecrlf true - отвержение необратимого преобразования lf<->crlf.

Мы работаем в системе Mac, поэтому выполним соответствующие команды.

git config --global core.autocrlf input  
git config --global core.safecrlf true

[Окончания строк](screenshots/112.png)

**1.1.3 Установка отображения unicode**

По умолчанию, git будет печатать не-ASCII символы в именах файлов в виде восьмеричных последовательностей.

Установим правильное отображение.

git config --global core.quotepath off

[Установка отображения](screenshots/113.png)

## 1.2 Создание проекта

**1.2.1 Создание страницы «Hello, World»**

Начнем работу в пустом рабочем каталоге с создания пустого каталога с именем hello, затем войдем в него и создадим там файл с именем hello.html.

mkdir hello  
cd hello  
touch hello.html  
echo "Hello, World!" > hello.html

[Создание каталога и файла](screenshots/121.png)

**1.2.2 Создание репозитория**

Создадим git репозиторий из этого каталога.

git init

[Создание репозитория](screenshots/122.png)

**1.2.3 Добавление файла в репозиторий**

Добавим файл в репозиторий.

git add hello.html  
git commit -m "Initial Commit"

[Добавление файла в репозиторий](screenshots/123.png)

**1.2.4 Проверка состояния репозитория**

Проверим текущее состояние репозитория.

git status

[Проверка текущего состояния](screenshots/124.png)

Команда проверки состояния сообщит нам, что коммитить нечего. Это означает, что в репозитории хранится текущее состояние рабочего каталога, и нет никаких изменений, ожидающих записи.

## 1.3 Внесение изменений

Добавим HTML-теги к нашему приветствию. Изменим содержимое файла hello.html.

[Добавление HTML-тегов](screenshots/1311.png)

Проверим состояние рабочего каталога.

git status

[Проверка текущего состояния](screenshots/1312.png)

git знает, что файл hello.html был изменен, но при этом эти изменения еще не зафиксированы в репозитории.

## 1.4 Индексация изменений

Теперь выполним команду git, чтобы проиндексировать изменения и проверим состояние.

git add hello.html  
git status

[Индексация изменений](screenshots/14.png)

Изменения файла hello.html были проиндексированы. Это означает, что git теперь знает об изменении, но изменение пока не записано в репозиторий. Следующий коммит будет включать в себя проиндексированные изменения.

Отдельный шаг индексации в git позволяет нам продолжать вносить изменения в рабочий каталог, а затем, в момент, когда мы захотим взаимодействовать с версионным контролем, git позволит записать изменения в малых коммитах, которые фиксируют то, что мы сделали. Разделяя индексацию и коммит, мы имеем возможность с легкостью настроить, что идет в какой коммит.

**1.4.1 Коммит изменений**

Когда мы ранее использовали git commit для коммита первоначальной версии файла hello.html в репозиторий, мы включили метку -m, которая делает комментарий в командной строке. Команда commit также позволяет нам интерактивно редактировать комментарии для коммита. Если мы опустим метку -m из командной строки, git перенесет нас в редактор.

Сделаем коммит.

git commit

[Коммит изменений](screenshots/1411.png)

Откроется редактор. В первой строке введем комментарий: «Added h1 tag». Сохраним файл и выйдем из редактора.

[Ввод комментария](screenshots/1412.png)

Теперь проверим состояние.

git status

[Проверка состояния](screenshots/1413.png)

Рабочий каталог чистый, можно продолжить работу.

**1.4.2 Добавление стандартных тегов страницы**

Изменим страницу «Hello, World», чтобы она содержала стандартные теги html и body.

[Добавление тегов html и body](screenshots/1421.png)

Теперь добавим это изменение в индекс git.

git add hello.html

[Добавление изменения](screenshots/1422.png)

Теперь добавим заголовки HTML (секцию head) к странице «Hello, World».

[Добавление секции head](screenshots/1423.png)

Проверим текущий статус.

git status

[Проверка статуса](screenshots/1424.png)

Мы видим, что hello.html указан дважды в состоянии. Первое изменение (добавление стандартных тегов) проиндексировано и готово к коммиту. Второе изменение (добавление заголовков HTML) является непроиндексированным. Если бы мы делали коммит сейчас, заголовки не были бы сохранены в репозиторий.

Произведем коммит проиндексированного изменения, а затем еще раз проверим состояние.

git commit -m "Added standard HTML page tags"  
git status

[Коммит и проверка состояния](screenshots/1425.png)

Состояние команды говорит о том, что hello.html имеет незафиксированные изменения, но уже не в буферной зоне. Теперь добавим второе изменение в индекс, а затем проверим состояние. В качестве файла для добавления используем текущий каталог (.).

git add .  
git status

[Добавление второго изменения и проверка состояния](screenshots/1426.png)

Второе изменение было проиндексировано и готово к коммиту. Сделаем коммит второго изменения.

git commit -m "Added HTML header"

[Коммит второго изменения](screenshots/1427.png)

**1.4.3 История**

Получим список произведенных изменений.

git log

[Получение списка изменений](screenshots/1431.png)

Также историю можно вывести в однострочном формате.

git log --pretty=oneline

[Вывод истории в однострочном формате](screenshots/1432.png)

Существует много вариантов отображения лога. Например, можно также использовать следующие команды:

git log --pretty=oneline --max-count=2  
git log --pretty=oneline --since='5 minutes ago'  
git log --pretty=oneline --until='5 minutes ago'  
git log --pretty=oneline --author=<your name>  
git log --pretty=oneline --all

**1.4.4 Получение старых версий**

Получим хэши предыдущих версий.

git log

[Вывод истории для получения хэшей](screenshots/1441.png)

Изучим данные лога и найдем хэш для первого коммита. Используем этот хэш-код в команде checkout. Затем проверим содержимое файла hello.html.

git checkout <hash>  
cat hello.html

[Использование хэша и проверка файла](screenshots/1442.png)

Вернемся к последней версии в ветке master (master — имя ветки по умолчанию). Снова проверим содержимое файла hello.html.

git checkout master  
cat hello.html

[Возвращение в последнюю версию и проверка файла](screenshots/1443.png)

**1.4.5 Работа с тегами версий**

Назовем текущую версию страницы hello первой (v1). Для этого создадим тег первой версии.

git tag v1

[Создание тега первой версии](screenshots/1451.png)

Теперь текущая версия страницы называется v1.

Теперь создадим тег для версии, которая идет перед текущей версией и назовем его v1-beta. В первую очередь нам надо переключиться на предыдущую версию. Вместо поиска до хэш, мы будем использовать ^, обозначающее «родитель v1».

git checkout v1^  
cat hello.html

[Переход к предыдущей версии](screenshots/1452.png)

Это версия c тегами html и body, но еще пока без head. Сделаем ее версией v1-beta.

git tag v1-beta

[Создание тега предшествующей версии](screenshots/1453.png)

Теперь попробуем попереключаться между двумя отмеченными версиями.

git checkout v1  
git checkout v1-beta

[Переключение между версиями](screenshots/146.png)

Просмотрим доступные теги.

git tag

[Просмотр доступных тегов](screenshots/1471.png)

Мы также можем посмотреть теги в логе.

git log master --all

[Просмотр тегов в логе](screenshots/1472.png)

Мы можем видеть теги (v1 и v1-beta) в логе вместе с именем ветки (master). Кроме того, HEAD показывает коммит, на который мы переключились (на данный момент это v1-beta).

## 1.5 Отмена локальных изменений (до индексации)

Убедимся, что мы находимся на последнем коммите ветки master, прежде чем продолжить работу.

git checkout master

[Переход к ветке master](screenshots/151.png)

**1.5.1 Изменение hello.html**

Внесем изменение в файл hello.html в виде нежелательного комментария.

[Внесение нежелательного комментария](screenshots/152.png)

Проверим состояние рабочего каталога.

git status

[Проверка состояния](screenshots/153.png)

Мы видим, что файл hello.html был изменен, но еще не проиндексирован.

**1.5.2 Отмена изменений в рабочем каталоге**

Используем команду git checkout для переключения версии файла hello.html в репозитории.

git checkout hello.html  
git status  
cat hello.html

[Отмена изменений](screenshots/154.png)

Команда git status показывает нам, что не было произведено никаких изменений, не зафиксированных в рабочем каталоге.

## 1.6 Отмена проиндексированных изменений (перед коммитом)

**1.6.1 Изменение файла и индексирование изменений**

Внесем изменение в файл hello.html в виде нежелательного комментария.

[Внесение нежелательного комментария](screenshots/1611.png)

Проиндексируем это изменение.

git add hello.html

[Индексирование изменения](screenshots/1612.png)

Проверим состояние нежелательного изменения.

git status

[Проверка состояния](screenshots/162.png)

Состояние показывает, что изменение было проиндексировано и готово к коммиту.

**1.6.2 Сброс буферной зоны**

Теперь произведем отмену проиндексированного изменения.

git reset HEAD hello.html

[Отмена проиндексированного изменения](screenshots/163.png)

Команда git reset сбрасывает буферную зону к HEAD. Это очищает буферную зону от изменений, которые мы только что проиндексировали.

Переключим версию коммита и проверим состояние.

git checkout hello.html  
git status

[Переключение версии коммита и проверка состояния](screenshots/164.png)

Наш рабочий каталог опять чист.

## 1.7 Отмена коммитов

Есть несколько способов отмены неверных коммитов, мы будем использовать самый безопасный. Мы отменим коммит путем создания нового коммита, отменяющего нежелательные изменения.

**1.7.1 Изменение файла и выполнение коммита**

Изменим файл hello.html.

[Изменение файла](screenshots/1721.png)

Далее сделаем коммит.

git add hello.html  
git commit -m "Oops, we didn't want this commit"

[Произведение коммита](screenshots/1722.png)

**1.7.2 Отмена коммита**

Чтобы отменить коммит, нам необходимо сделать коммит, который удаляет изменения, сохраненные нежелательным коммитом.

git revert HEAD

[Удаление нежелательных изменений](screenshots/1731.png)

Мы перейдем в редактор, где можем отредактировать коммит-сообщение по умолчанию или оставить все как есть. Сохраним и закроем файл.

[Редактор коммит-сообщения](screenshots/1732.png)

Так как мы отменили самый последний произведенный коммит, мы смогли использовать HEAD в качестве аргумента для отмены. Мы можем отменить любой произвольной коммит в истории, указав его хэш-значение.

**1.7.3 Проверка лога**

Проверка лога показывает нежелательные и отмененные коммиты в наш репозиторий.

git log

[Проверка лога](screenshots/174.png)

## 1.8 Удаление коммиттов из ветки

git revert является мощной командой, которая позволяет отменить любые коммиты в репозиторий. Однако, и оригинальный и «отмененный» коммиты видны в истории ветки (при использовании команды git log).

**1.8.1 Команда git reset**

При получении ссылки на коммит (т.е. хэш, ветка или имя тега), команда git reset:

* перепишет текущую ветку, чтобы она указывала на нужный коммит;
* опционально сбросит буферную зону для соответствия с указанным коммитом;
* опционально сбросит рабочий каталог для соответствия с указанным коммитом.

**1.8.2 Проверка истории**

Сделаем проверку нашей истории коммитов.

git log

[Проверка истории коммитов](screenshots/182.png)

Мы видим, что два последних коммита в этой ветке — «Oops» и «Revert Oops». Удалим их с помощью сброса.

**1.8.3 Сброс коммитов к предшествующему коммиту Oops**

Прежде чем удалить коммиты, отметим последний коммит тегом, чтобы потом можно было его найти.

git tag oops

[Отметка коммита тегом](screenshots/183.png)

Глядя на историю лога, мы видим, что коммит с тегом «v1» является коммитом, предшествующим ошибочному коммиту. Сбросим ветку до этой точки.

Поскольку ветка имеет тег, мы можем использовать имя тега в команде сброса (если она не имеет тега, мы можем использовать хэш-значение).

git reset --hard v1  
git log

[Сброс ветки до коммита v1](screenshots/184.png)

Наша ветка master теперь указывает на коммит v1, а коммитов Oops и Revert Oops в ветке уже нет. Параметр –hard указывает, что рабочий каталог должен быть обновлен в соответствии с новым head ветки.

**1.8.4 Поиск ошибочных коммитов**

Однако, ошибочные коммиты все еще находятся в репозитории, и мы все еще можем на них ссылаться.

Посмотрим на все коммиты.

git log --all

[Просмотр коммитов](screenshots/185.png)

Мы видим, что ошибочные коммиты не исчезли. Они все еще находятся в репозитории. Просто они отсутствуют в ветке master. Если бы мы не отметили их тегами, они по-прежнему находились бы в репозитории, но не было бы никакой возможности ссылаться на них, кроме как при помощи их хэш имен. Коммиты, на которые нет ссылок, остаются в репозитории до тех пор, пока не будет запущен сборщик мусора.

**1.8.5 Опасность сброса**

Сброс в локальных ветках, как правило, безопасен. Последствия любой «аварии» как правило, можно восстановить простым сбросом с помощью нужного коммита. Однако, если ветка «расшарена» на удаленных репозиториях, сброс может сбить с толку других пользователей ветки.

## 1.9 Удаление тега oops

Удалим тег «oops» и коммиты, на которые он ссылался, сборщиком мусора.

git tag -d oops  
git log --all

[Удаление тега и коммитов](screenshots/191.png)

Тег «oops» больше не будет отображаться в репозитории.

## 1.10 Внесение изменений в коммиты

**1.10.1 Внесение изменений в страницу**

Добавим в страницу комментарий автора (вставим свою фамилию).

[Добавление комментария автора](screenshots/11011.png)

Выполним коммит.

git add hello.html  
git commit -m "Add an author comment"

[Выполнение коммита](screenshots/11012.png)

Обновим страницу hello, включив в нее email.

[Добавление email в комментарий](screenshots/1102.png)

**1.10.2 Изменение предыдущего коммита**

Давайте изменим предыдущий коммит, включив в него адрес электронной почты, чтобы не создавать для этого отдельный коммит.

git add hello.html  
git commit --amend -m "Add an author/email comment"

[Изменение предыдущего коммита](screenshots/1103.png)

Просмотрим иторию.

git log

[Просмотр истории](screenshots/1104.png)

Мы можем увидеть, что оригинальный коммит «автор» заменен коммитом «автор/email». Этого же эффекта можно достичь путем сброса последнего коммита в ветке, и повторного коммита новых изменений.

## 1.11 Перемещение файлов

**1.11.1 Перемещение файла hello.html в каталог lib**

Сейчас мы собираемся создать структуру нашего репозитория. Перенесем страницу в каталог lib.

mkdir lib  
git mv hello.html lib  
git status

[Перенос файла в каталог](screenshots/1111.png)

Перемещая файлы с помощью git mv, мы информируем git о следующих вещах:

* Что файл hello.html был удален.
* Что файл lib/hello.html был создан.
* Оба эти факта сразу же проиндексированы и готовы к коммиту. Команда git status сообщает, что файл был перемещен.

## 1.12 Второй способ перемещения файлов

Ту же задачу с перемещением файла можно было бы решить следующим набором команд:

mkdir lib  
mv hello.html lib  
git add lib/hello.html  
git rm hello.html

**1.12.1 Коммит в новый каталог**

Сделаем коммит этого перемещения:

git commit -m "Moved hello.html to lib"

[Коммит перемещения файла](screenshots/01121.png)

## 1.13 Подробнее о структуре

**1.13.1 Добавление index.html**

Добавим файл index.html в наш репозиторий.

[Файл index.html](screenshots/011311.png)

Добавим файл и сделаем коммит.

git add index.html  
git commit -m "Added index.html."

[Добавление файла и коммит](screenshots/011312.png)

Теперь при открытии index.html, вы увидим кусок страницы hello в маленьком окошке.

[Открытие файла](screenshots/011313.png)

## 1.14 Git внутри: Каталог .git

**1.14.1 Каталог .git**

Просмотрим каталог .git, в котором хранится вся информация git.

ls -C .git

[Просмотр .git](screenshots/01141.png)

**1.14.2 База данных объектов**

Далее просмотрим .git/objects.

ls -C .git/objects

[Просмотр .git/objects](screenshots/01142.png)

Мы видим набор каталогов, имена которых состоят из 2 символов. Имена этих каталогов являются первыми двумя буквами хэша sha1 объекта, хранящегося в git.

**1.14.3 Углубление в базу данных объектов**

Далее посмотрим в один из каталогов с именем из 2 букв.

ls -C .git/objects/<dir>

[Просмотр каталога из .git/objects](screenshots/01143.png)

Мы увидим файлы с именами из 38 символов. Это файлы, содержащие объекты, хранящиеся в git.

**1.14.4 Config File**

Посмотрим содержимое .git/config.

cat .git/config

[Просмотр файла конфигураций](screenshots/01144.png)

Это файл конфигурации, создающийся для каждого конкретного проекта. Записи в этом файле будут перезаписывать записи в файле .git/config нашего главного каталога.

**1.14.5 Ветки и теги**

Просмотрим .git/refs и его внутренние файлы.

ls .git/refs  
ls .git/refs/heads  
ls .git/refs/tags  
cat .git/refs/tags/v1

[Просмотр .git/refs и внутренних файлов](screenshots/01145.png)

Каждый файл в подкаталоге tags соответствует тегу, ранее созданному с помощью команды git tag. Его содержание — это хэш коммита, привязанный к тегу. Каталог heads практически аналогичен, но используется для веток, а не тегов. На данный момент у нас есть только одна ветка, так что мы увидим в этом каталоге только ветку master.

**1.14.6 Файл HEAD**

Просмотрим содержимое .git/HEAD.

cat .git/HEAD

[Просмотр содержимого .git/HEAD](screenshots/01146.png)

Файл HEAD содержит ссылку на текущую ветку, в данный момент это ветка master.

## 1.15 Работа непосредственно с объектами git

Просмотрим последний коммит в репозиторий.

git log --max-count=1

[Просмотр последнего коммита](screenshots/01151.png)

**1.15.1 Вывод информации с помощью SHA1 хэша**

Выведем информацию о последнем коммите.

git cat-file -t <hash>  
git cat-file -p <hash>

[Информация о последнем коммите](screenshots/01152.png)

Мы можем вывести дерево каталогов, ссылка на которое идет в коммите. Используем SHA1 хэш из строки «дерева» из списка, полученного выше.

git cat-file -p <treehash>

[Вывод дерева](screenshots/01153.png)

Выведем каталог lib.

git cat-file -p <libhash>

[Вывод каталога](screenshots/01154.png)

Выведем файл hello.html.

git cat-file -p <hellohash>

[Вывод файла](screenshots/01155.png)

**1.15.2 Исследование репозитория**

Исследуем git репозиторий и найдем оригинальный файл hello.html с самого первого коммита вручную по ссылкам SHA1 хэша в последнем коммите.

[Путь до первого коммита(1)](screenshots/011561.png)

[Путь до первого коммита(2)](screenshots/011562.png)

## 1.16 Создание ветки

Переместим изменения в отдельную ветку, чтобы изолировать их от изменений в ветке master.

Назовем нашу новую ветку «style».

git checkout -b style  
git status

[Создание новой ветки](screenshots/01161.png)

Обратим внимание, что команда git status сообщает о том, что мы находимся в ветке «style».

**1.16.1 Добавление файла стилей style.css**

Создадим файл стилей.

touch lib/style.css

[Создание файла стилей](screenshots/011621.png)

[Содержимое файла стилей](screenshots/011622.png)

Добавим файл и сделаем коммит.

git add lib/style.css  
git commit -m "Added css stylesheet"

[Добавление и коммит](screenshots/011623.png)

**1.16.2 Изменение файлов**

Обновим файл hello.html, чтобы использовать стили style.css.

[Обновление содержания файла](screenshots/011631.png)

Выполним коммит.

git add lib/hello.html  
git commit -m "Hello uses style.css"

[Выполнение коммита](screenshots/011632.png)

Обновим файл index.html, чтобы он тоже использовал style.css.

[Изменение файла](screenshots/011641.png)

Выполним коммит.

git add index.html  
git commit -m "Updated index.html"

[Выполнение коммита](screenshots/011642.png)

## 1.17 Навигация по веткам

Теперь в нашем проекте есть две ветки. Просмотрим лог.

git log --all

[Просмотр лога](screenshots/0117.png)

Переключимся между ветками.

git checkout master  
cat lib/hello.html

[Переключение ветки](screenshots/01171.png)

Сейчас мы находимся на ветке master. Это заметно по тому, что файл hello.html не использует стили style.css.

Вернемся к ветке style.

git checkout style  
cat lib/hello.html

[Возвращение к ветке style](screenshots/01172.png)

Содержимое lib/hello.html подтверждает, что мы вернулись на ветку style.

## 1.18 Изменения в ветке master

**1.18.1 Создание файла README в ветке master**

Перейдем к ветке master.

git checkout master

[Переход к ветке master](screenshots/011811.png)

Создадим файл README.md.

echo "This is the Hello World example from the git tutorial." > README.md

[Создание файла README.md](screenshots/011812.png)

## 1.19 Выполнение коммита изменений README.md в ветку master

Сделаем коммит.

git add README.md  
git commit -m "Added README"

[Выполнение коммита](screenshots/0119.png)

**1.19.1 Просмотр текущих веток**

Теперь у нас в репозитории есть две отличающиеся ветки. Просмотрим ветки и их отличия.

git log --graph --all

[Просмотр веток(1)](screenshots/011911.png)

[Просмотр веток(2)](screenshots/011912.png)

Добавление опции –graph в git log вызывает построение дерева коммитов с помощью простых ASCII символов. Мы видим обе ветки (style и master), и то, что ветка master является текущей HEAD. Общим предшественником обеих веток является коммит «Added index.html». Опция –all гарантированно означает, что мы видим все ветки. По умолчанию показывается только текущая ветка.

## 1.20 Слияние веток

Слияние переносит изменения из двух веток в одну. Вернемся к ветке style и сольем master с style.

git checkout style  
git merge master  
git log --graph --all

[Слияние веток](screenshots/012011.png)

[Просмотр лога](screenshots/012012.png)

Путем периодического слияния ветки master с веткой style мы можем переносить из master любые изменения и поддерживать совместимость изменений style с изменениями в основной ветке.

## 1.21 Создание конфликта

**1.21.1 Возвращение в master и создание конфликта**

Вернемся в ветку master и внесем изменения.

git checkout master

[Возвращение к ветке master](screenshots/012111.png)

[Внесение изменений в файл](screenshots/012112.png)

Выполним коммит.

git add lib/hello.html  
git commit -m 'Life is great'

[Выполнение коммита](screenshots/012113.png)

**1.21.2 Просмотр веток**

Просмотрим лог.

git log --graph --all

[Просмотр лога](screenshots/01212.png)

После коммита «Added README» ветка master была объединена с веткой style, но в настоящее время в master есть дополнительный коммит, который не был слит с style. Последнее изменение в master конфликтует с некоторыми изменениями в style. На следующем шаге мы решим этот конфликт.

## 1.22 Разрешение конфликтов

**1.22.1 Слияние master с веткой style**

Теперь вернемся к ветке style и попытаемся объединить ее с новой веткой master.

git checkout style  
git merge master

[Попытка объединения веток](screenshots/012211.png)

[Просмотр файла](screenshots/012212.png)

Открыв lib/hello.html, мы увидим в нем два раздела. Первый раздел — версия текущей ветки (style). Второй раздел — версия ветки master.

**1.22.2 Решение конфликта**

Нам необходимо вручную разрешить конфликт. Внесем изменения в lib/hello.html.

[Внесение изменений в файл](screenshots/01222.png)

Сделаем коммит решения конфликта.

git add lib/hello.html  
git commit -m "Merged master fixed conflict."

[Выполнение коммита](screenshots/01223.png)

**1.22.3 Перебазирование как альтернатива слиянию**

Рассмотрим различия между слиянием и перебазированием. Для того, чтобы это сделать, нам нужно вернуться в репозиторий в момент до первого слияния, а затем повторить те же действия, но с использованием перебазирования вместо слияния. Мы будем использовать команду reset для возврата веток к предыдущему состоянию.

## 1.23 Сброс ветки style

Вернемся на ветке style к точке перед тем, как мы слили ее с веткой master. Мы можем сбросить ветку к любому коммиту.

Нам необходимо найти последний коммит перед слиянием.

git checkout style  
git log --graph

[Возвращение к ветке style и просмотр коммитов(1)](screenshots/012311.png)

[Возвращение к ветке style и просмотр коммитов(2)](screenshots/012312.png)

Мы видим, что коммит «Updated index.html» был последним на ветке style перед слиянием. Сбросим ветку style к этому коммиту.

git reset --hard <hash>

[Сброс ветки к коммиту](screenshots/012313.png)

Теперь попробуем найти лог ветки style.

git log --graph --all

[Просмотр лога](screenshots/01232.png)

У нас в истории больше нет коммитов слияний.

## 1.24 Сброс ветки master

Добавив интерактивный режим в ветку master, мы внесли изменения, конфликтующие с изменениями в ветке style.Вернемся в ветке master в точку перед внесением конфликтующих изменений. Это позволяет нам продемонстрировать работу команды git rebase, не беспокоясь о конфликтах.

git checkout master  
git log --graph

[Возвращение к ветке master и просмотр лога](screenshots/012411.png)

Коммит «Added README» идет непосредственно перед коммитом конфликтующего интерактивного режима. Мы сбросим ветку master к коммиту «Added README».

git reset --hard <hash>  
git log --graph --all

[Сброс ветки к коммиту](screenshots/012412.png)

## 1.25 Перебазирование

Используем команду rebase вместо команды merge. Мы вернулись в точку до первого слияния и хотим перенести изменения из ветки master в нашу ветку style. На этот раз для переноса изменений из ветки master мы будем использовать команду git rebase вместо слияния.

git checkout style  
git rebase master  
git log --graph

[Перенос изменений и просмотр результатов](screenshots/0125.png)

**1.25.1 Слияние VS перебазирование**

Конечный результат перебазирования очень похож на результат слияния. Ветка style в настоящее время содержит все свои изменения, а также все изменения ветки master. Однако, дерево коммитов значительно отличается. Дерево коммитов ветки style было переписано таким образом, что ветка master является частью истории коммитов. Это делает цепь коммитов линейной и гораздо более читабельной.

## 1.26 Слияние в ветку master

Мы поддерживали соответствие ветки style с веткой master (с помощью rebase), теперь сольем изменения style в ветку master.

Перейдем к ветке master и произведем слияние.

git checkout master  
git merge style

[Переход к ветке master и слияние веток](screenshots/01261.png)

Поскольку последний коммит ветки master прямо предшествует последнему коммиту ветки style, git может выполнить ускоренное слияние-перемотку. При быстрой перемотке вперед git просто передвигает указатель вперед, таким образом указывая на тот же коммит, что и ветка style. При быстрой перемотке конфликтов быть не может.

Просмотрим лог.

git log

[Просмотр лога](screenshots/01262.png)

Теперь ветки style и master идентичны.

## 1.27 Клонирование репозиториев

Перейдем в рабочий каталог и сделаем клон нашего репозитория hello.

cd ..  
pwd  
ls

[Переход в рабочий каталог](screenshots/01271.png)

Сейчас мы находимся в рабочем каталоге. Здесь есть единственный репозиторий под названием «hello».

Создадим клон репозитория.

git clone hello cloned\_hello  
ls

[Клонирование репозитория](screenshots/01272.png)

В нашем рабочем каталоге теперь два репозитория: оригинальный репозиторий «hello» и клонированный репозиторий «cloned\_hello».

## 1.28 Просмотр клонированного репозитория

**1.28.1 Просмотр содержимого**

Посмотрим на клонированный репозиторий.

cd cloned\_hello  
ls

[Просмотр клонированного репозитория](screenshots/01281.png)

Мы видим список всех файлов на верхнем уровне оригинального репозитория: README.md, index.html и lib.

**1.28.2 Просмотр истории**

Просмотрим историю репозитория.

git log --all

[Просмотр истории репозитория](screenshots/01282.png)

Мы видим список всех коммитов в новый репозиторий. Единственная разница с оригинальным репозиторием в названиях веток.

**1.28.3 Удаленные ветки**

Мы увидели ветку master (HEAD) в списке истории, а также увидите ветки со странными именами (origin/master, origin/style и origin/HEAD).

## 1.29 Что такое origin?

Выполним команду git remote:

git remote

[Получение имени удаленного репозитория](screenshots/01291.png)

Мы видим, что клонированный репозиторий знает об имени по умолчанию удаленного репозитория. Посмотрим, можем ли мы получить более пдробную информацию об имени по умолчанию.

git remote show origin

[Получение более подробной информации](screenshots/01292.png)

## 1.30 Удаленные ветки

Посмотрим на ветки, доступные в нашем клонированном репозитории.

git branch

[Просмотр доступных локальных веток](screenshots/0130.png)

Как мы видим, в списке только ветка master. Команда git branch выводит только список локальных веток по умолчанию.

Теперь просмотрим все ветки.

git branch -a

[Просмотр всех доступных веток](screenshots/01301.png)

Git выводит все коммиты в оригинальный репозиторий, но ветки в удаленном репозитории не рассматриваются как локальные. Если мы хотим собственную ветку style, мы должны сами ее создать.

## 1.31 Изменение оригинального репозитория

Внесем некоторые изменения в оригинальный репозиторий, чтобы затем попытаться извлечь и слить изменения из удаленной ветки в текущую.

**1.31.1 Внесение изменений в оригинальный репозиторий hello**

Вернемся в оригинальный репозиторий.

cd ../hello

[Возвращение в оригинальный репозиторий](screenshots/013111.png)

Внесем изменения в файл README.md.

[Изменение файла README.md](screenshots/013112.png)

Теперь добавим это изменение и сделаем коммит.

git add README  
git commit -m "Changed README in original repo"

[Выполнение коммита внесенного изменения](screenshots/013113.png)

Теперь в оригинальном репозитории есть более поздние изменения, которых нет в клонированной версии. Далее мы извлечем и сольем эти изменения в клонированный репозиторий.

**1.31.2 Извлечение изменений**

Извлечем изменения из удаленного репозитория.

cd ../cloned\_hello  
git fetch  
git log --all

[Извлечение изменений из удаленного репозитория](screenshots/01312.png)

Сейчас мы находимся в репозитории cloned\_hello. На данный момент в репозитории есть все коммиты из оригинального репозитория, но они не интегрированы в локальные ветки клонированного репозитория. Обратим внимание, что коммит «Changed README in original repo» включает в себя коммиты «origin/master» и «origin/HEAD». Посмотрев на коммит «Updated index.html», мы увидим, что локальная ветка master указывает на этот коммит, а не на новый коммит, который мы только что извлекли.

Выводом является то, что команда git fetch будет извлекать новые коммиты из удаленного репозитория, но не будет сливать их с наработками в локальных ветках.

**1.31.3 Проверка README.md**

Мы можем продемонстрировать, что клонированный файл README.md не изменился.

cat README

[Проверка содержимого файла](screenshots/01313.png)

## 1.32 Слияние извлеченных изменений

**1.32.1 Слияние извлеченных изменений в локальную ветку master**

Сольем извлеченные изменения в ветку master.

git merge origin/master

[Слияние извлеченных изменений](screenshots/01321.png)

**1.32.2 Проверка файла README.md**

Проверим файл README.md еще раз.

cat README.md

[Проверка содержимого файла](screenshots/013221.png)

Хотя команда git fetch не сливает изменения, мы можем вручную слить изменения из удаленного репозитория. Теперь рассмотрим объединение fetch и merge в одну команду. Выполнение команды:

git pull

эквивалентно двум следующим шагам:

git fetch  
git merge origin/master

[Проверка команды pull](screenshots/013222.png)

## 1.33 Добавление ветки наблюдения

Ветки, которые начинаются с remotes/origin являются ветками оригинального репозитория. Обратим внимание, что у нас больше нет ветки под названием style, но система контроля версий знает, что в оригинальном репозитории ветка style была.

Добавим ветку для отслеживания удаленной ветки.

git branch --track style origin/style  
git branch -a  
git log --max-count=2

[Добавление локальной ветки наблюдения](screenshots/001331.png)

Теперь мы можем видеть ветку style в списке веток и логе.

## 1.34 Чистые репозитории

Чистые репозитории (без рабочих каталогов) обычно используются для расшаривания. Обычный git-репозиторий подразумевает, что вы будете использовать его как рабочую директорию, поэтому вместе с файлами проекта в актуальной версии, git хранит все служебные, «чисто-репозиториевские» файлы в поддиректории .git.

## 1.35 Создание чистого репозитория

Создадим чистый репозиторий.

cd ..  
git clone --bare hello hello.git  
ls hello.git

[Создание чистого репозитория и просмотр его содержимого](screenshots/00135.png)

Сейчас мы находимся в рабочем каталоге. Как правило, репозитории, оканчивающиеся на .git являются чистыми репозиториями. Мы видим, что в репозитории hello.git нет рабочего каталога. По сути, это есть не что иное, как каталог .git нечистого репозитория.

## 1.36 Добавление удаленного репозитория

Добавим репозиторий hello.git к нашему оригинальному репозиторию.

cd hello  
git remote add shared ../hello.git

[Добавление чистого репозитория к оригинальному репозиторию](screenshots/00136.png)

## 1.37 Отправка изменений

Так как чистые репозитории, как правило, расшариваются на каком-нибудь сетевом сервере, нам необходимо отправить наши изменения в другие репозитории.

Начнем с создания изменения для отправки. Отредактируем файл README.md и сделаем коммит.

[Редактирование файла](screenshots/001371.png)

git checkout master  
git add README  
git commit -m "Added shared comment to readme"

[Выполнение коммита](screenshots/001372.png)

Теперь отправим изменения в общий репозиторий.

git push shared master

[Отправка изменений в общий репозиторий](screenshots/001373.png)

Общим называется репозиторий, получающий отправленные нами изменения.

## 1.38 Извлечение общих изменений

Извлечем изменения из общего репозитория. Переключимся в клонированный репозиторий и извлечем изменения, только что отправленные в общий репозиторий.

cd ../cloned\_hello  
git remote add shared ../hello.git  
git branch --track shared master  
git pull shared master  
cat README.md

[Извлечение изменений из общего репозитория](screenshots/00138.png)

# Выводы

В результате проделанной работы мы познакомились с ситемой контроля версий git и ее основными функциями и научились пользоваться основными командами git.

# Список литературы

* Методические материалы к лабораторной работе, представленные на сайте “ТУИС РУДН” https://esystem.rudn.ru/