**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ**

**ФЕДЕРАЦИИ**

**Федеральное государственное автономное образовательное учреждение**

**высшего образования**

**«Севастопольский государственный университет»**

**ПРИМЕНЕНИЕ РАСПРЕДЕЛЕННОЙ СИСТЕМЫ КОНТРОЛЯ ВЕРСИЙ GIT В ПРОЦЕССЕ РАЗРАБОТКИ ИС**

**Методические указания**

к лабораторной работе №2

по дисциплине

**“Управление IT-проектами”**

для студентов специальности 09.03.02 –

"Информационные системы и технологии"

всех форм обучения

**Севастополь**

**2015**

1. **Цель работы:** изучить основы локальной работы с системой контроля версий Git.
2. **ОСНОВНЫЕ ПОЛОЖЕНИЯ**
3. **Постановка задачи контроля версий**

Инструменты контроля версий призваны решать задачу **объединения работы** авторов, которые выполняют её на физически разных машинах, что крайне затруднительно (скорее даже невозможно) в случае использования резервных копий. Также системы контроля версий позволяют **легко создавать новые экспериментальные версии**, сохраняя при этом стабильные старые.

Это позволяет легко экспериментировать с изменениями, вливая в стабильную версию часть экспериментальных, отменяя все ненужные изменения и многое другое.

Различают **три поколения систем управления версиями**, кото­рые представлены в таблице 1.1. Самые ранние системы появились более сорока лет назад, первая из них – SCCS, разработанная в Bell Labs в 1972 году.

*Таблица 1.1*

**Системы управления версиями**

|  |  |  |  |
| --- | --- | --- | --- |
| Поколе­ние | Сетевая модель | Операции над файлами | Представители |
| Первое | Нет | Один файл за раз | RCS, sees |
| Второе | Централи­зованная | Несколько файлов за раз | CVS, SVN |
| Третье | Распределённая | Наборы изменений | Git, Mercurial |

Системы контроля версий разделяются на **централизованные** и **распределённые** системы контроля.

**Централизованные системы** имеют сравнительно низкую отказоустойчивость, т.к. всё хранится сосредоточенно в одном месте и аппаратный сбой системы приведёт к потере всей истории изменений. К ним относится популярный продукт SVN (Subversion).

Принципиально другой подход реализуется в **распределённых системах контроля**, таких как Git. Они предполагают хранение всей истории изменений на компьютере каждого автора. Это делает систему максимально устойчивой к сбоям, однако, затрудняет распространение изменений в истории.

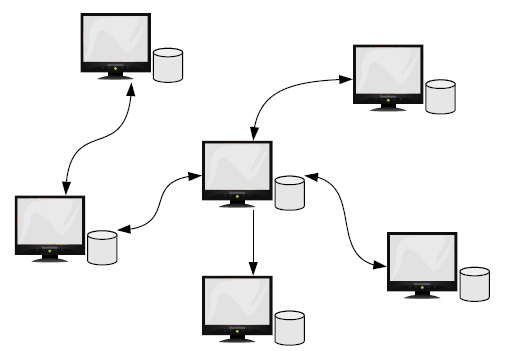


Рис.1.1. Распределенная система управления версиями.

В данной работе будет рассмотрена именно VCS **Git**, т.к. она сейчас набирает популярность, а также предлагает широкий набор возможностей для работы с историей версий.

Благодаря распределённой природе Git, все **изменения производятся локально**, не требуя при этом доступа к центральному репозиторию в случае с централизованными системами контроля версий. Это позволяет легко просматривать историю за большой период*: Git уже скачал всю историю изменений к вам на компьютер и не потребует медленной операции скачивания их с сервера или обсчёта изменений между текущей версией и версией месячной давности, которая сильно нагрузит сервер.*

Работу можно выполнять где угодно, и когда угодно. Как правило, этот плюс Git ставится в противоположность CVCS, в которых локальная работа крайне затруднительна.

Более того, вы можете даже забросить ваши изменения, затем вернуться к ним, обновить историю, предшествующую им, и всё равно нормально влить свои изменения в проект!

*Преимущества Git:*

• *высокая скорость работы* – нет необходимости дожидаться ответа от сервера, чтобы зафиксировать изменения, очень быстро выполняется просмотр всей истории изменений, так как она расположена локально, а не в сети;

• *надёжность* – у каждого разработчика хранится полная копия всех версий, что даёт возможность восстановления исход­ных текстов даже в случае сбоя центрального сервера (в Git все участники сети равноправны, но обычно всегда есть «центральное» хранилище, через которое происходит обмен исходным кодом);

• *гибкость, удобство и мобильность* – в Git очень легко выполнять коммиты и объединения, нет жёстких ограничений на последовательность рабочего процесса, работать с системой управления версиями можно даже без подключения к сети, а её интерфейс построен с учётом прошлого омыта и современных требований к процессу разработки ПО;

• *безопасность* – Git осуществляет контроль целостности данных при помощи контрольных сумм (при любом изменении, фиксируются не только сами изменения, но и их контрольная сумма; это означает, что даже в случае удачной попытки взлома репозитория, Git будет препятствовать внедрению вредоносного кода в исходные тексты программы без ведома разработчиков проекта, ровно как любые изменения в рабочем каталоге не пропадут бесследно).

Ключевым понятием в системах управления версиями являет­ся **репозиторий** хранилище, в котором содержатся исходные тексты программы, история их изменения и другая служебная информация. История изменений репозитория накапливается путём создания коммитов, которые вручную формируются раз­работчиком.

***Репозиторий***– это рабочая директория проекта. В случае с Git помимо файлов и директорий проекта она содержит в себе скрытую папку **.git**, *которая хранит в себе всю необходимую информацию о проекте, полную историю, набор скриптов и указателей Git, которые используются для навигации по истории.*

В Git файлы могут находиться в одном из трёх состояний:

* зафиксированном,
* изменённом,
* подготовленном.

**Зафиксированный** значит, что файл уже сохранён в вашей локальной базе.

К **изменённым** относятся файлы, которые поменялись, но ещё не были зафиксированы.

**Подготовленные** файлы – это изменённые файлы, отмеченные для включения в следующий коммит.

Таким образом, в проекте с использованием Git есть **три части**:

* рабочий каталог (working directory),
* область подготовленных файлов (staging area),
* каталог Git (Git directory).

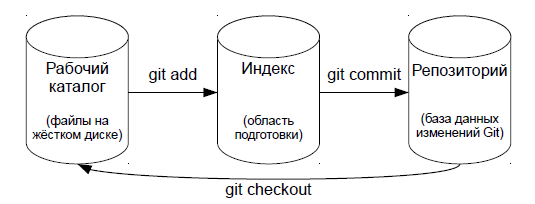


Рис.1.2. Схема взаимодействия репозитория, рабочего каталога и индекса.

Эти файлы достаются из сжатой базы данных в каталоге Git и помещаются на диск для того, чтобы вы их просматривали и редактировали.

**Рабочий каталог** – список файлов, которые фактически хранятся на жёстком диске. Именно в рабочем каталоге происхо­дит изменение исходного текста программы и манипуляции над файлами проекта.

**Область подготовленных файлов** – это обычный файл, обычно хранящийся в каталоге Git, который содержит информацию о том, что должно войти в следующий коммит.

**Каталог Git** – это место, где Git хранит метаданные и базу данных объектов вашего проекта. Это наиболее важная часть Git.

**Индекс** – виртуальная область подготовки изменений. Данная область не существует ни в виде физического репозитория, ни в виде физических файлов – она лишь сохраняет список файлов, которые попадут в коммит. В процессе своей работы разработчик изменяет файлы на жёстком диске и добавляет новые. Для того чтобы эти изменения попали в репозиторий, необходимо сначала поместить все изменённые файлы в область подготовки, а потом сделать коммит.

**Коммит** (англ, commit) – фиксация изменений в репозито­рии, или так называемая контрольная точка. Коммит фиксирует текущее состояние исходного кода, время фиксации, автора и со­провождается комментарием. Со временем коммиты накаплива­ются и формируют историю изменений.

**Ветка** – независимое от других направление разработки. С помощью механизма веток можно организовать развитие функциональности программы но нескольким направлениям, не ме­шающим друг другу. Ветка может начинаться с любого коммита любой другой ветки, таким образом, вся история разработки пред­ставляет собой дерево коммитов. Коммит, из которого начинается другая ветка, называют точкой ветвления, а коммит, в котором две ветки были объединены, объединением.

Стандартный **рабочий процесс с использованием Git** выглядит примерно так:

1. Вы изменяете файлы в вашем рабочем каталоге.

2. Вы подготавливаете файлы, добавляя их слепки в область подготовленных файлов.

3. Вы делаете коммит. При этом слепки из области подготовленных файлов сохраняются в каталог Git.

**Git работает** не с файлами по отдельности, а **с репозиторием целиком**, поэтому в Git при фиксации изменений просто копируется актуальный на данный момент сле­пок файловой системы. Впрочем, это не означает, что репозиторий Git будет занимать очень много дискового пространства: **в Git используются очень эффективные алгоритмы сжатия данных**, в результате чего Git не уступает аналогичным системам.

Чтобы понять, **как работает Git, рассмотрим следующий при­мер**.

**Разработчик создаёт программу**, и каждый день сохраняет резервную копию каталога со всеми её файлами.

В **первый день** работа происходит над программой в каталоге program\_1.

На **второй день** создаётся копия каталога program\_1 с именем program\_2, в котором разработчик продолжает создание программы, при этом все изменения, выполненные за предыдущий день, сохранены в каталоге program\_1.

На **следующий день** создаётся каталог program\_3 и т. д.

Дополнительно, рядом с каждым каталогом **создаётся файл, описывающий изменения за день**: info\_l.txt, info\_2.txt.

Таким образом, **можно отследить историю разработ­ки**, вернуться на день или два назад для тестирования программы или её отладки. Конечно, создавать полную копию всех файлов неэффективно с точки зрения занимаемого пространства на диске. В данном случае можно не копировать те файлы, которые не менялись со вчерашнего дня, а просто сохранять ссылки на них.

Такая схема разработки напоминает линейную разработку программы с использованием Git. Работа с репозиторием подра­зумевает фиксацию изменений – коммит. Когда пользователь делает коммит, Git автоматически создаёт «слепок» рабочего каталога, фактически создавая новую копию исходных текстов программы. Для тех файлов, которые изменились, создаётся копия, а на остальные файлы создаётся ссылка в предыдущую версию. Однако, в отличие от примера с каталогами, Git автома­тически контролирует все изменения и для каждого изменения сохраняет короткое сообщение комментарий. Таким образом, пользователь работает только с текущей версией программы, a Git сохраняет все её предыдущие варианты.

Со временем коммиты образуют цепочку, отражающую хроно­логическую последовательность разработки программы.

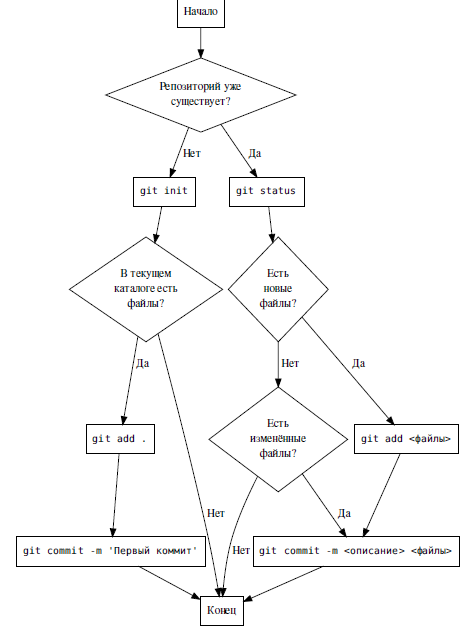


Рис.1.2. Алгоритм повседневной работы с Git.

Большинство профессиональных разработчиков пред­почитают управлять репозиториями Git при помощи **командной строки**.

1. **Основы работы с Git**

Формат команд Git следующий:

$ git <команда> [<параметры>]

Например:

$ git commit -m "first commit" file.txt

* + 1. **Получение справочной информации**

Все команды снабжены подробной справкой. Для получения справки но Git, нужно выполнить команду man git. Для получе­ния справки по одной из команд Git, нужно выполнить команду git help <команда>, например, следующие команды выполняют одно действие – отображают справку но команде commit:

$ git help commit

$ git commit --help

$ man git-commit

Список доступных команд Git можно посмотреть, выполнив git с параметром help:

$ git help

* + 1. **Первичная настройка**

Система Git идентифицирует пользователя но его имени и ад­ресу электронной почты. Перед началом работы с Git необходимо указать свои данные, система запомнит их, и впоследствии все изменения будут подписаны [этим именем и адресом электронной почты.

$ git config --global user.name "Ваше Имя"

$ git config --global user.email [your@e-ma.il](mailto:your@e-ma.il)

Проверить текущие значения можно при помощи тех же команд без параметров:

$ git config --global user.name

$ git config --global user.email

Во многих системах Git устанавливается без **поддержки ки­риллицы**, что может привести к некорректному отображению имён файлов и коммитов, содержащих русские символы. Для того чтобы устранить эту ошибку, нужно выполнить следующую команду:

$ git config --global core.quotepath false

Это **выключит перекодирование русских символов** в Git, остав­ляя их без изменений. После этого можно будет отправлять и просматривать коммиты на русском языке.

После переустановки системы (при сохранении пользователь­ских данных) первичную настройку повторять не нужно, но её необходимо выполнить на всех компьютерах, на которых пла­нируется использовать Git (например, настольный компьютер и ноутбук).

* + 1. **Создание нового проекта**

Создать новый проект Git очень просто. Для этого достаточно создать новый каталог, перейти в него, и в нём выполнить команду git init:

$ mkdir project

$ cd project

$ git init

Выполнить git init можно даже в уже существующем проек­те – в каталоге с исходными текстами программы за исключением случая, когда этот каталог уже принадлежит какому-либо Git-репозиторию.

Например, некоторое время происходила разработка ПО одним разработчиком и уже имеется какой-то накопленный объём исход­ного кода, но к нему присоединяется ещё несколько разработчиков, в результате чего возникает необходимость использования систе­мы управления версиями. В данном случае достаточно перейти в каталог с программой и выполнить там git init.

Каталог, внутри которого была выполнена команда git init. становится рабочим каталогом. Контроль версий осуществляется только для файлов и каталогов, находящихся внутри рабочего каталога. В рабочем каталоге создаётся скрытый подкаталог .git, внутри которого Git хранит всю необходимую служебную информацию. Если каталог .git удалить, рабочий каталог станет обычным каталогом в файловой системе, а накопленная история изменений исчезнет.

Важно понимать, что команда git init только создаёт репо­зиторий, который на данный момент пуст, даже если git init выполняется в каталоге с файлами – индекс и репозиторий в данный момент пусты. Файлы проекта необходимо добавлять в Git вручную.

* + 1. **Добавление новых файлов в проект**

Git устроен таким образом, что все файлы из рабочего каталога нужно вручную добавлять в репозиторий. Если файл не был включён в репозиторий, Git игнорирует его и не отслежи­вает изменения в этом файле. Добавить файл, находящийся вне рабочего каталога, в общем случае нельзя.

Одна из самых распространённых ошибок начинающих поль­зователей при работе с Git – либо случайное добавление ненуж­ных файлов (временные или промежуточные бинарные файлы) в репозиторий, либо наоборот, попытка фиксации изменений для файлов, которые не были добавлены в репозиторий. Для тот, чтобы избежать таких ошибок, рекомендуется проверять статус репозитория как можно чаще (git status).

Для добавления файла в репозиторий используется команда **git add**.

Данная команда помещает файлы в область подготовки (ин­декс), а с последующим коммитом эти файлы перемещаются в репозиторий. Таким образом, файлы проделывают путь из рабочего каталога в репозиторий через индекс.

Добавить все файлы текущего каталога в репозиторий при помощи следующей команды:

$ git add .

Добавить все файлы полезно при создании нового репозитория, однако с данной командой нужно обращаться с осторожностью, так как в индекс могут случайно попасть лишние файлы.

Желательно перед тем, как выполнить команду git add ., посмотреть, какие файлы будут добавлены. Для этого выполните следующую команду:

$ git add –n .

Данная команда просто покажет список файлов, но фактиче­ски не будет добавлять их в индекс. Если всё нормально, можно будет выполнить команду, реально добавляющую файлы в индекс.

В процессе работы не рекомендуется добавлять файлы при помощи команды git add ., желательно всегда указывать какой конкретно файл нужно добавить. Если нужно добавить несколько файлов, их можно перечислить в командной строке, как показано в следующем примере:

$ git add main.с functions.с \*.h

Данная команда добавляет в индекс все файлы с расширени­ем .h. а также файлы main.с и functions.с.

Практически всегда в проекте существуют файлы, для которых не нужна система управления версиями, но они лежат в том же каталоге, что и репозиторий.

**Примеры файлов, для которых не нужно включать версионный контроль:**

• файлы с объектным кодом (расширение .о), скомпилирован­ный исполняемый файл программы;

• временные файлы (\*~, \*.bak);

• файлы, создаваемые автоматическими системами сборки: Makefile, configure (вид этих файлов зависит от используемой cистемы сборки);

• архивы с исходным кодом;

• скомпилированные файлы документации (pdf, chm).

Для исключения лишних объектов из версионного контроля в Git предусмотрен файл .gitignore, в котором описываются все файлы, которые не должны попадать в репозиторий. Данный файл нужно положить в каталог с проектом и перечислить в нём все имена файлов, которые нужно игнорировать – каждый файл с новой строки.

Наряду с добавлением файлов, следует отметить команды удаления и перемещения файлов. Для удаления файлов из репозитория используется команда git rm. для перемещения и переименования – git mv, аналогичные, соответственно, ко­мандам rm и mv. Преимуществом данных команд является то, что они автоматически перемещают обработанные файлы в область подготовки.

* + 1. **Проверка состояния репозитория**

В процессе работы с репозиторием иногда можно забыть, в каком состоянии он находится: какие файлы изменились, какие появились новые, а какие были удалены. Команда проверки состояния репозитория Git – git status – призвана помочь в этом, и является наиболее часто используемой командой в Git.

Формат команды:

$ git status [<файл(ы)>]

Основная информация, которую можно получить с помощью git status:

• текущая ветка, с которой происходит работа в данный мо­мент;

• какие файлы уже находятся в области подготовки (индексе) команда git commit -а затронет именно эти файлы;

• какие файлы были изменены в рабочем каталоге, но для них не был выполнен коммит и/или они не были добавлены в индекс;

• какие файлы, находящиеся в репозитории, были удалены из рабочего каталога;

• какие файлы появились в рабочем каталоге, но не внесены в индекс репозитория;

• список файлов, которые ожидают ручного разрешения кон­фликтов.

**Всегда делайте git status перед тем.? как что-либо изменить в репозитории.**

Команда **git difftool**, которая поможет сравнить версию из индекса с последним коммитом.

Например, с помощью git status удалось выяснить, что **файл** util.sh неделю назад **претерпел** какие-то **изменения**, но они не отмечены в индексе. Вспомнить, какие изменения были внесены в файл util.sh неделю назад, практически невозможно. С помощью коман­ды **git difftool** **можно установить**, что **в файл** util.sh **случайно попала лишняя пустая строка**. Очевидно, что эти изменения лучше отменить, чтобы не загрязнять историю разработки.

Формат команды:

$ git difftool -у [<файл>]

Данная команда отображает изменения, которые были внесены в указанный файл с момента предыдущего коммита.

В примере команда вычисляет различия между определёнными коммитами.

$ git difftool -у а6cbl cf513

$ git difftool -y HEAD~2

Здесь a6cbl и cf513 – хэши коммитов, между которыми нужно посмотреть различия. Ключевое слово HEAD~2 означает «два коммита назад от текущего».

В Git любой коммит имеет **уникальный идентификатор**. Од­нако, в отличие от других систем управления версиями, где коммитам просто назначались порядковые номера, данный **иден­тификатор** представляет собой **контрольную сумму коммита или, сокращённо, хэш**.

Git сохраняет все данные, используя для их хранении контрольную сумму SHA-1, вычисленную на основе файла или каталога. **Длинна хэша составляет 40 символов**. Он используется Git в качестве индекса при обращении к файлам в своей базе данных.

Git рассчитывает контрольную сумму SHA-1 коммита на основе сразу всей истории изменений, из-за чего каждый коммит получает свой уникальный номер. Это важный аспект защиты данных внутри Git, благодаря которому их практически невоз­можно испортить и невозможно незаметно изменить историю разработки.

Примеры уникальных идентификаторов:

673 c5e7c6d4d5dbdf1650cd437f90d7bcl2ele81

ed894fа551180a2fe69a294fЬ1ае43а684ае6784

blfc8f6fal417af10dddbl8eb85e39edbbd96d6d

Однако, такой идентификатор имеет один **недостаток**: *его слож­но набрать на клавиатуре без ошибок или запомнить*. Для того чтобы упростить работу с контрольными суммами коммитов, **Git позволяет обращаться** лишь к **первым нескольким символам** хэша (не меньше 4-х)

* + 1. **Создание коммита**

Формат команды:

$ git commit **-m** ’<Комментарий>’ [<файл(ы)>]

Опция **-m**, позволяет добавлять **краткое описание внесенных изменений**, что в дальнейшем позволяет локализовать ошибки и упрощает разработку.

При выполнении коммита **желательно** **указывать файлы, к ко­торым он относится**. Если файлы не указаны, Git сканирует весь проект и в репозиторий попадут все изменённые файлы, находящиеся в области подготовки. Внимательно читайте сообщения об ошибках: в них система сообщает, что пошло не так.

Пример:

$ git commit -m ’Реализация сортировки’ sort . с

Если комментарий не указан (параметр -m), Git откроет ре­дактор vi, в котором попросит ввести комментарий к коммиту. Коммит без комментария создать невозможно.

$ git commit с опцией -a

$ git commit **-a** -m "Added new function to main.cpp"

git commit **-а** выполнит коммит не только для файлов, находящихся в области подготовки, но и для файлов, которые уже были добавлены ранее в репозиторий и изменились в рабочем каталоге. Иными словами, git commit **-а** **делает коммит для всех файлов, кроме тех, для которых явно не выполнили команду git add**.

$ git commit <файл(ы)>

$ git add file1.txt file2.cpp file3.tex

автоматически помещает перечис­ленные файлы в индекс, если они ещё не были добавлены. Данная команда является самой безошибочной, поэтому на этапе изучения Git всегда рекомендуется явно указывать, для каких файлов нужно сделать коммит.

**2.1.7 Откат коммита**

Когда коммит уже был выполнен, существует альтернативный метод его отмены – откат коммита. Данная операция создаёт ещё один коммит, отменяя действие предыдущего. Этот вариант удобен, когда с репозиторием работает много разработчиков и другие методы отмены коммита могут привести к серьёзным ошибкам в программе.

Формат команды:

$ git revert <коммит>

Здесь <кокмит> – хэш/тэг коммита или относительный адрес коммита.

Следующая команда очень полезна в повседневной работе: она отменяет только что выполненный коммит:

$ git revert HEAD

Эта команда отменяет коммит но заданному хэшу:

$ git revert Ь6а23

* + 1. **Быстрая отмена изменений**

Часто возникает ситуация, когда какой-то файл был испорчен/удалён, но коммит ещё не выполнен. В таком случае Git позволяет быстро отменить изменения, о чём сообщает команда

Например, используя команду git status, пользователь просматривает текущие изменения

$ git status

…

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: algorithm.c

Ненужные изменения можно отменить, используя команду git checkout

$ git checkout -- algorithm.c

Команда сбросит все изменения в файле algorithm.с. Эту возможность удобно использовать для временных экспериментов с исходным кодом без риска что-либо испортить.

**Внимание!** Если выполнить коммит после git checkout, все последующие коммиты будут утеряны! Для того чтобы избежать порчи репозитория, рекомендуется сразу создать новую ветку.

* + 1. **Отмена изменений перед коммитом**

Это самый простой метод отмены изменений.

Например, если случайно удалить файл на диске и выполнить коммит, данный файл исчезнет и из репозитория. Но приучившись выполнять git status, легко заметить неладное до того, как будет выполнен коммит (очевидно, что нужно воспользоваться этим полезным советом всегда делать git status перед git commit). Таким образом, можно всегда вовремя восстановить удалённые или повреждённые файлы.

Рассмотрим пример. Имеется репозиторий, в нём содержится несколько файлов:

$ ls

file1 file2 file3

Испортим наш репозиторий и удалим один файл:

$ rm file2

$ ls

filel file3

Теперь посмотрим, что произошло с репозиторием:

$ git status

# On branch master

# Changed but not updated:

# (use "git add/rm <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

# deleted: file2

#

no changes added to commit (use "git add" and/ or "git commit -a")

Видно, что из каталога с репозиторием был удалён один файл. Удобно, что Git сразу предлагает варианты дальнейших действий.

В нашем случае:

$ git checkout -- file2

$ ls

filel file2 file3

Файл вернулся на место.

* + 1. **Тэги**

Git не обладает понятием «версия программы», вместо этого он предоставляет более гибкий инструмент – возможность помечать коммиты.

**Тэг** (англ, tag – метка) – уникальная строка, которая будет соответствовать данному коммиту. Именно при помощи тэгов разработчики обновляют в Git версию своей программы, хотя на самом деле версией будет просто какой-то конкретный коммит.

Тэг назначается отдельной командой git tag, которая поме­чает предыдущий коммит. Если набрать команду без аргументов, она выведет список тэгов:

$ git tag

v 1.0

v 1.1

v l.2

Для того чтобы отметить предыдущий коммит, достаточно просто вызвать команду:

$ git tag <тэг>

Например, если нужно изменить версию программы на 1.4:

$ git tag "1.4"

* + 1. **Просмотр истории изменений**

Для этого служит команда

$ git log

Кроме того, у этой команды есть подробный режим пошагового просмотра изменений

$ git log -p

Чтобы посмотреть список коммитов в компактном виде, используйте команду:

$ git log –oneline

fd750f4 new feature

laf5079 new file

a73fdd0 test

По умолчанию git log выдаёт отчёт в развёрнутом виде, одна­ко его можно сократить, используя параметр --format.

$ git log --format = "\*/.h \*/.ai \*/.s"

fd750f4 2013-10-16 16:47:04 +0400 new feature

laf5079 2013-10-16 16:34:51 +0400 new file

a73fddO 2013-09-25 12:57:58 +0400 test

или

$ git log --format = "\*/.an \*/.s"

Иванов Иван new feature

Иванов Иван new file

Иванов Иван test

* + 1. **Ветви**

Механизм ветвей позволяет проводить эксперименты, не нарушая основной линии развития программы. Редактирование отдельной ветви программы не затрагивает исходного кода ветви **master**, поэтому безопасно. Для создания ветви следует воспользоваться командой

$ git branch experimental

После этого можно проверить, сколько в данный момент ветвей в программе при помощи команды

$ git branch

experimental

\* master

При этом текущая «рабочая» ветвь помечена звездочкой. Для смены ветви необходимо воспользоваться командой git checkout:

$ git checkout experimental

После завершения редактирования можно сохранить изменения и вернуться на основную ветвь

$ git commit -a -m "Added some experiments"

$ git checkout master

Для внесения изменений из экспериментальной ветви в основную ветвь следует воспользоваться командой git merge

$ git merge experimental

Для удаления ветви (с проверкой внесения изменений в основную ветвь) используется команда

$ git branch -d experimental

Удаление ветви без внесения изменений в основную ветвь делается командой

$ git branch -D experimental

* + 1. **Очистка репозитория**

Git в процессе работы накапливает временные файлы, необхо­димые для ускорения доступа к репозиторию и хранения другой служебной информации. Со временем эти файлы становятся не нужны, но Git не удаляет их автоматически до тех пор, пока это не будет сделано явно.

Для того чтобы очистить репозиторий, выполните команду:

$ git gc –prune

Команда никак не влияет на состояние репозитория, за исклю­чением уменьшения занимаемого репозиторием дискового про­странства.

1. **ПРАКТИЧЕСКАЯ ЧАСТЬ**

**Ход работы**

* 1. Запустите Git.
  2. Установите имя и электронную почту

git config --global user.name "Your Name"

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

* 1. Начните работу в пустом рабочем каталоге

mkdir <имя каталога>

* 1. Выберите каталог

cd <имя каталога>

* 1. Создайте там текстовый файл

touch <имя файла. >

* 1. Создайте репозиторий

git init

* 1. Добавьте страницу в репозиторий

git add hello.html

* 1. Создайте коммит

git commit -m "First Ccmmit"

* 1. Проверьте состояние репозитория

git status

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

* 1. Внесите изменение в исходный файл
  2. Проверьте состояние

git status

Первое, что нужно заметить, это то, что git знает, что созданный ранее файл  был изменен, но при этом эти изменения еще не зафиксированы в репозитории.

Также обратите внимание на то, что сообщение о состоянии дает вам подсказку о том, что нужно делать дальше. Если вы хотите добавить эти изменения в репозиторий, используйте команду git add. В противном случае используйте команду git сheckout для отмены изменений.

* 1. Добавьте изменения

git add <имя файла. >

* 1. Проверьте состояние

git status

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

Если вы решили, что не хотите коммитить изменения, команда состояния напомнит вам о том, что с помощью команды git reset можно снять индексацию этих изменений.

* 1. Закоммитьте изменения

git commit

Если вы опустите метку -m из командной строки, git перенесет вас в редактор по вашему выбору. Редактор выбирается из следующего списка (в порядке приоритета):

* переменная среды GIT\_EDITOR
* параметр конфигурации core.editor
* переменная среды VISUAL
* переменная среды EDITOR
  1. В первой строке редактора введите комментарий: «Added h1 tag». Сохраните файл и выйдите из редактора (для этого в редакторе по-умолчанию (Vim) вам нужно нажать клавишу ESC, ввести :wq и нажать Enter).
  2. Проверьте состояние

git status

Рабочий каталог чистый, можете продолжить работу.

* 1. Измените созданный ранее файл
  2. Добавьте это изменение в индекс git.

git add <имя файла>

* 1. Добавьте ещё одно изменение в исходный файл
  2. Проверьте текущий статус

git status

Обратите внимание на то, что файл указан дважды в состоянии.

Первое изменение проиндексировано (git add) и готово к коммиту.

Второе изменение является непроиндексированным. Если бы вы делали коммит сейчас, заголовки не были бы сохранены в репозиторий.

* 1. Произведите коммит проиндексированного изменения (значение по умолчанию)

git commit -m "Added …"

* 1. Еще раз проверьте состояние

git status

Состояние команды говорит о том, что файл имеет незафиксированные изменения, но уже не в буферной зоне.

* 1. Добавьте второе изменение в индекс

git add .

В качестве файла для добавления, использовался текущий каталог («.»). Это самый краткий и удобный путь для добавления всех изменений в файлы текущего каталога и его подкаталоги. Но поскольку он добавляет все, не лишним будет проверить состояние перед запуском add, просто чтобы убедиться, что вы не добавили какой-то файл, который добавлять было не нужно.

* 1. Проверьте состояние с помощью команды git status. Второе изменение было проиндексировано и готово к коммиту.
  2. Сделайте коммит второго изменения

git commit -m. "Added …"

* 1. Просмотрите историю проекта

git log

вы увидите список всех коммитов в репозиторий, которые были совершены.

27. Используйте однострочный формат

git log --precty=oneline

* 1. Выведите несколько вариантов истории

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

git log --pretty=format: "%h %ad | %s%d [%an]" --graph --date=short

* --pretty="..." – определяет формат вывода.
* %h – укороченный хэш коммита
* %d – дополнения коммита («головы» веток или теги)
* %ad – дата коммита
* %s – комментарий
* %an – имя автора
* --graph – отображает дерево коммитов в виде ASCII-графика
* --date=short – сохраняет формат даты коротким и симпатичным
  1. Просмотрите историю проекта

gitk

* 1. Настройте алиасы и шоркаты для команд Git

git config --global alias.со checkout

git config --global alias.ci commit

git config --global alias.st status

git config --global alias.br branch

git config --global alias.hist 'log --pretty=format:"%h %ad | %s%d [%an]" --graph --date=short'

Теперь вы можете вводить git st там, где раньше приходилось использовать git status. Аналогичным образом, пишем git co вместо git checkout и git ci вместо git commit.

* 1. Используйте новую команду для просмотра состояния

Git st

* 1. Создайте алиасы для следующих команд

alias gs='git status '

alias ga='git add '

alias gb='git branch'

alias gc='git commit'

alias gd='git diff'

alias go='git checkout '

Сокращение go для команды git checkout особенно полезно. Оно позволяет вводить: go <branch>

* 1. Просмотрите историю проекта

git hist

* 1. Изучите данные лога и найдите хэш для первого коммита. Он должен быть в последней строке данных git hist. Используйте этот хэш-код (достаточно первых 7 знаков) в команде ниже.

git checkout <hash>

cat <имя файла. >

<hash> – это первые семь знаков хэш-кода, например $ git checkout 911e8c9

* 1. Проверьте содержимое созданного файла.
  2. Вернитесь к последней версии в ветке master

git checkout master

cat <имя файла. >

«master» – имя ветки по умолчанию. Переключая имена веток, вы попадаете на последнюю версию выбранной ветки.

* 1. Внесите изменение в файл.

Иногда случается, что вы изменили файл в рабочем каталоге, и хотите отменить последние коммиты. Для этого используется команда checkout.

* 1. Проверьте состояние

git status

Видим, что файл  был изменен, но еще не проиндексирован.

* 1. Отмените изменения в рабочем каталоге.

git checkout <имя файла. >

* 1. Проверьте состояние

git status

Команда status показывает, что не было произведено никаких изменений, не зафиксированных в рабочем каталоге. И нежелательное изменение больше не является частью содержимого файла.

cat <имя файла. >

* 1. В очередной раз измените файл.
  2. Проиндексируйте изменения

git add <имя файла. >

* 1. Проверьте состояние нежелательного изменения.

git status

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

* 1. Выполните сброс буферной зоны.

git reset HEAD <имя файла. >

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

Команда reset (по умолчанию) не изменяет рабочий каталог. Поэтому рабочий каталог все еще содержит нежелательный комментарий.

* 1. Удалите нежелательные изменения в рабочем каталоге, используя команду checkout из задания.
  2. Просмотрите изменения

git status

Иногда вы понимаете, что новые коммиты являются неверными, и хотите их отменить. Есть несколько способов решения этого вопроса, далее мы будем использовать самый безопасный.

Отмените коммит путем создания нового коммита, отменяющего нежелательные изменения.

* 1. Измените файл.
  2. Проиндексируйте его и создайте коммит.

git add <имя файла. >

git commit -m "Oops, we didn't want this commit"

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

git revert HEAD -–no-edit

* 1. Проверьте историю

git hist

* 1. Внесите изменение в файл (добавьте автора проекта).
  2. Выполните

git add <имя файла. >

git commit -m "Add an author comment"

После совершения коммита вы понимаете, что любой хороший комментарий должен включать электронную почту автора. Обновите файл, включив в него email автора.

* 1. Выполните

git add <имя файла. >

git commit --amend -m "Add an author/email comment"

* 1. Просмотрите историю

git hist

Видно, что оригинальный коммит «автор» заменен коммитом «автор/email»

* 1. Переместите созданный ранее файл в другой каталог

mkdir lib

git mv <имя файла. > lib

* 1. Просмотрите состояние

git status

* 1. Сделайте коммит этого перемещения

git commit -m "Moved <имя файла. > to lib"

1. **СОДЕРЖИМОЕ ОТЧЁТА**

Отчёт должен содержать в себе:

1. Титульный лист.

2. Краткое описание работы с локальным репозиторием Git.

3. Практическая работа над выбранным ранее проектом с использованием СКВ Git.

Логическая последовательность выполнения команд должна быть сохранена (commit не должен идти раньше init). Также должен быть список использованных команд и их описание.

3. Вывод о проделанной работе (полученные навыки, замеченные трудности или, наоборот, улучшения в работе).

4. Список использованных источников

1. **КОНТРОЛЬНЫЕ ВОПРОСЫ**
2. Для чего предназначена система управления версиями исходного кода программы?
3. Что такое репозиторий?
4. Из чего формируется история изменений в репозитории?
5. Для чего используется механизм ветвления?
6. Что такое система Git? Какова её область применения?
7. В чём заключается ключевое отличие системы Git от систем управления версиями предыдущего поколения?
8. В чём различие централизованных и распределённых систем управления версиями?
9. Смогут ли разработчики продолжить работу с репозиторием (просмотр истории изменений, сравнение версий, создание новых коммитов) после выхода из строя центрального хранилища исходных текстов в случае централизованной системы управления версиями? В случае распределённой? Ответ поясните.
10. Требуется ли в системе Git для выполнения коммитов и просмотра истории изменений доступ в сеть?
11. Что представляет собой интерфейс системы Git?
12. Как получить справку по командам системы Git?
13. Какие первичные настройки нужно выполнить перед началом работы с Git? Нужно ли дублировать первичную настройку на другом компьютере?
14. Можно ли создать репозиторий Git в уже существующем проекте с исходным кодом?
15. Если в рабочем каталоге создать новый репозиторий, в каком состоянии будут находиться индекс и репозиторий?
16. В каком случае нужно выполнить коммит сразу после того, как был создан новый репозиторий? Ответ поясните.
17. Как добавить в репозиторий файл из рабочего каталога?
18. Как избежать случайного добавления ненужных файлов в репозиторий?
19. Как добавить в репозиторий несколько файлов? Все файлы рабочего каталога?
20. Возможно ли выполнить команду git commit, не указывая комментарий (параметр -m)? Что произойдёт в данном случае?
21. Был создан новый каталог, в него положили новый файл и инициализировали репозиторий. Какие действия необходимо выполнить перед коммитом, чтобы этот файл оказался в репозитории?
22. В Git есть команда, которая проверяет состояние репозитория и выводит соответствующие подсказки – git status. Приведите как минимум два примера, в каких ситуациях данная команда может быть полезна.
23. Какую полезную информацию о репозитории можно получить с помощью команды git status?
24. **РЕКОМЕНДУЕМЫЕ ИНТЕРНЕТ-ИСТОЧНИКИ**
25. Git How То – интерактивный тур по Git на русском языке: <http://githowto.com/ru>.
26. Интерактивный учебник Git: <http://tty.github.com/>.
27. Интерактивный учебник по ветвлению Git, работающий в браузере: <http://pcottle.github.io/learnGitBranching/>.
28. Сайт github.com является хорошим справочником по Git. Справочная система GitHub находится по адресу: <http://help.github.com/>.
29. Интерактивный учебник "Pro Git" <http://progit.org/book/>
30. Интерактивный учебник "Git Community Book" <http://book.git-scm.com/>
31. Книга "Git Magic" <http://www-cs-students.stanford.edu/~blynn/gitmagic/index.html>,

перевод: <http://habrahabr.ru/blogs/Git/80909/>

1. Сравнение команд Git-SVN <http://git.or.cz/course/svn.html>
2. Редактирование истории в git <http://gq.net.ru/2009/12/16/git-history-rewrite/>