Дата последней сборки документа:
20 декабря 2020 г. в 11:00

Практика использования и наиболее полезные конструкции системы контроля версий `Git`

Содержание

1. Термины и определения

`HEAD` – специальный *указатель* на текущую *локальную ветку*, которая в свою очередь ссылается на последнее зафиксированное состояние, т.е. на *последний* сделанный в ней *коммит*.

2. Настройка строки приглашения в командной оболочке `bash`

Для того чтобы настроить автодополнение команд и строку приглашения в командой оболочке `bash` первым делом нужно скачать файл `contrib/completion/git-completion.bash` и поместить его копию в домашнюю директорию

```
$ curl -o ~/git-completion.bash https://raw.githubusercontent.com\
                /git/git/master/contrib/completion/git-completion.bash
$ echo '. ~/git-completion.bash' >> .bash_profile
```

Затем следует скачать файл `contrib/completion/git-prompt.sh` и сохранить его также в домашней директории

```
$ curl -o ~/git-prompt.sh https://github.com/git/git/blob/master/contrib\
                /completion/git-prompt.sh
$ echo $'. ~/git-prompt.sh\nexport GIT_PS1_SHOWDIRTYSTATE=1\n
                export PS1=\'\w$(__git_ps1 " (%s)")\$ \'' >> .bash_profile
```

Выражение `$'...'` умеет интерпретировать escape-последовательности типа `\n`, `\t` и пр., но выражение `'\w$(__git_ps1 " (%s)")\$ '` стоящее в этой строке как подстрока должно быть заключено в одинарные кавычки, тогда как аргумент функции `__git_ps1` должен быть заключен в двойные кавычки, поэтому одинарные кавычки подстроки приходится экранировать `\'`. Конструкция `$(...)` подставляет то, что возвращает выражение, заключенное в круглые скобки.

Однако строку приглашения можно сделать интереснее если переменной `PS1` передать

.bash_profile

```
export PS1='\[\e[36m\][\u]\[\e[m\]:\[\e[33;1m\]\w$(__git_ps1 " (%s)")\[\e[m\]\$ '
```

Запись `\u` означает имя пользователя, `\w` как и раньше – текущую директорию, а символ `\$` означает «сырой» символ доллара как приглашение командной оболочки.

3. Настройка системы `Git`

Первым делом конфигурационные параметры `Git` берет из файла `/etc/gitconfig`, содержащего значения для всех пользователей системы и всех репозиториев этих пользователей. Па-

раметр `--system`, добавленный к команде `git config`, инициализирует чтение именно из этого файла и запись в него.

Затем `Git` смотрит файл `~/.gitconfig` (или `~/.config/git/config`), привязанный к конкретному пользователю. Чтение и запись этого файла активизирует передача параметра `--global`.

Последним местом поиска конфигурационных параметров является файл в папке используемого в текущий момент репозитория (`.git/config`). Находящиеся в этом файле значения связаны с конкретным репозиторием.

Итак

- системный уровень: `/etc/gitconfig` (флаг `--system`),
- глобальный уровень: `~/.gitconfig` (флаг `--global`),
- локальный уровень (уровень репозитория): `.git/config` (флаг `--local`).

Значения с каждого уровня переопределяют значения, заданные на предыдущем уровне. Например, значения в файле `.git/config` переписывают значения в файле `/etc/gitconfig`, т.е. настройки репозитория имеют приоритет выше, чем настройки для всех пользователей и всех их репозиториев.

Вывести список всех настроек на системном, глобальном или локальном уровнях можно так

```
git config --system --list
git config --global --list
git config --local --list
```

Эти связанные с путями параметры называют Git-атрибутами. Они задаются в файле `.gitattributes` одной из папок (обычно это корневая папка проекта) или в файле `.git/info/attributes`. Последнее делается, когда вы не хотите, чтобы файл с атрибутами включался в коммиты вашего проекта

.git/info/attributes

```
*.pdf binary
```

3.1. Параметр core.excludesfile

Чтобы система `Git` не рассматривала определенные файлы как неотслеживаемые и не пыталась индексировать их при выполнении команды `git add`, следует создать шаблон в файле `.gitignore`. Но иногда возникает необходимость сделать так, чтобы система игнорировала определенные файлы для всех ваших репозиториев. Если вы работаете на компьютере с операционной системой `Mac OS X`, то вам знакомы файлы `.DS_Store` или, например, файлы, которые заканчиваются на `~`.

Параметр `core.exludesfile` позволяет записать файл `.gitignore`, действующий на глобальном уровне. Достаточно создать файл `~/.gitignore_global` в вот таким содержимым

~./gitignore_global

```
*~
.DS_Store
```

Теперь можно передать пусть до этого файла параметру `core.excludesfile`

```
git config --global core.excludesfile ~/.gitignore_global
```

3.2. Внешние инструменты для слияния и индикации изменений

Конфликты слияния можно решать, например, с помощью специальной программы Helix Visual Merge Tool (P4Merge) https://www.perforce.com/downloads/visual-merge-tool.

Создаем сценарий `extMerge.sh` для процедуры слияния, который будет вызывать бинарный файл `p4merge` со всеми переданными аргументами

~/bin/extMerge.sh

```
#!/bin/bash

"/c/Program Files (x86)/Perforce/p4merge" "$@"
```

Оболочка для команды `diff` проверяет наличие всех семи аргументов, которые ей должны передать, и отправляет два из них вашему сценарию слияния. По умолчанию `Git` передает команде `diff` следующие аргументы

```
путь старый-файл старый-хеш старые-права новый-файл новый-хеш новые-права
```

Так как нам требуются только второй и пятый аргументы, то есть имена старого и нового файлов, воспользуемся сценарием-оболочкой, который передаст только нужные сведения

~/bin/extDiff.sh

```
#!/bin/bash

[[ $# -eq 7 ]] && ~/bin/extMerge.sh "$2"
```

Заодно следует убедится, что наши инструменты представляют собой исполняемые файлы

```
chmod +x ~/bin/extMerge.sh
chmod +x ~/bin/extDiff.sh
```

Тепреь можно сделать так, чтобы конфигурационный файл вызывал наши инструменты разрешения конфликтов слияния и индикации добавленных изменений. Для этого потребуется несколько настроек:

- параметр `merge.tool` укажет `Git`, какую стратегию следует использовать,
- параметр `mergetool.*.cmd` определит способ запуска команды,
- параметр `mergetool.trustExitCode` даст `Git` понять, указывает код завершения программы на успешное разрешение конфликта или нет,
- параметр `diff.external` объяснит `Git`, какой командой следует воспользоваться для получения изменений.

Отредактируем файл `~/.gitconfig`

~/.gitconfig

```
...
[merge]
    tool = extMerge.sh
[mergetool "extMerge.sh"]
    cmd = extMerge.sh "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
    trustExitCode = false
[diff]
    external = extDiff.sh
```

После этого можно запустить команду `diff`

```
git diff HEAD~1 HEAD~2
```

или так

```
# git diff было стало
git diff HEAD~1:file_name.txt HEAD~2:file_name.txt
```

Если попытка слияния двух веток оканчивается конфликтом, запустите команду `git mergetool`. В результате откроется программа P4Merge, в которой можно будет разрешить конфликт.

В случае проблем с кодировкой нужно просто указать правильную *File* → *Character Encoding*.

3.3. Форматирование и пробелы

Если вы работатет в операционной системе `Windows`, а ваше коллеги нет, проблемы, связанные с символами конца строки, неизбежны. Дело в том, что в Windows-файлах для новых строк используется как символ перевода строки, так и символ возврата каретки, в то время как в Unix-подобных системах (MacOS X, Linux) – только символ перевода строки.

Для того чтобы `Git` автоматически конвертировал окончания строк типа CRLF в тип LF в момент индексирования файла и производя обратное преобразование при выгрузке кода из репозитория в файловую систему, нужно задать параметр `core.autocrlf`

```
git config --global core.autocrlf true
```

3.4. Хранение учетных данных

Учетные данные хранятся в памяти некоторое время без сохранения на диске (режим кеша)

```
git config --global credential.helper cache --timeout 30000
```

3.5. Псевдонимы

Пример создания псевдонима для команды `status`

```
git config --global alias.st status
```

Теперь этот псевдоним можно в командной оболочке использовать

```
git st -sb
```

Можно создавать псевдонимы и для более сложных команд

```
git config --global alias.last 'log -1 HEAD'
```

3.6. Создание открытого ключа SSH

Многие серверы производят аутентификацию по открытым ключам SSH. Каждый пользователь вашей системы должен сгенерировать себе такой ключ, если он пока отсутствует. Этот процесс происходит одинаково во всех операционных системах. Первым делом нужно убедиться в отсутствии ключа. По умолчанию пользовательские SSH-ключи хранятся в папке `~/.ssh` каждого пользователя.

Нам нужна пара файлов с такими именами, как, к примеру, `id_dsa` или `id_rsa` и соответствующий файл с расширением `.pub`. Файл с расширением `.pub` содержит *открытый ключ*, а второй файл – *закрытый ключ*.

Если таких файлов нет, то их можно создать с помощью утилиты `ssh-keygen`

```
$ ssh-keygen
```

Сначала указывается место хранения ключа (`.ssh/id_rsa`), затем дважды просят ввести парольную фразу. Если вы не хотите вводить пароль, когда используете ключ, пропустите эту операцию.

Теперь можно отослать ключ администратору Git-сервера. Для этого нужно скопировать содержимое файла `.pub` и отправить его по электронной почте. Сам файл выглядит так

```
$ cat ~/.ssh/id_rsa.pub
# выведет
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAA...ABnPxw+6G5Gtp8kJK8LKtPNM= ADM@ADM-HP
```

4. Фундаментальные концепции

4.1. Слежение

При извлечении данных с удаленного репозитория (git fetch/pull) изменения будут объединяться с соответствующими ветками: `master` с `origin/master`, `feature` с `origin/feature`. Ветки объединяются разумеется не по имени, а за счет *upstream tracking*.

4.2. Слияние

При *слиянии* веток снчала нужно перейти в ту ветку, в которую требуется слить данные, а затем применить команду `git merge`, т.е.

```
git checkout master
git merge server
```

4.3. Перемещение

При *перемещении*[1] данных из одной ветки в другую следует сначала перейти в ту ветку, из которой требуется перенести данные, а затем воспользоваться `git rebase`, т.е.

```
git checkout experiment
git rebase master
```

Набор коммитов из ветки, которая перемещается, устанавливается на самый верх ветки, в которую выполняется перемещение.

4.4. Общая схема работы в небольшой команде

Общая схема работы в небольшой команде:

- Некоторое время вы работаете в тематической ветке (например, `issue54`), и когда приходит время, сливаете результаты своего труда в ветку `master`

[1] Т.е. чтобы повторить изменения из одной ветки в другой

```
git checkout master
git merge issue54
```

- Решив, что пришло время поделиться своими наработками с коллегами, вы скачиваете данные с сервера (`git fetch origin`), и если там появились изменения, сливаете к себе ветку `origin/master`, т.е. `git merge origin/master`,
- После чего содержимое ветки `master` можно отправить на сервер `git push origin master`.

4.5. Типовые сценарии работы над проектом

Можно выделить следующие типовые сценарии:

- Выпус релизов (release),
- Исправление ошибок (bug fixing),
- Срочные исправления (hot fix),
- Разработка нескольких фич одновременно (features development),
- Выпуск определенных фич или выпуск по готовности.

В ветке `master` обычно хранится последняя выпущенная версия и она обновляется только от релиза к релизу. Не будет лишним сделать и tag с указанием версии, т.к. ветка может обновится.

Понадобиться еще одна ветка для основной разработки, та самая в которую будет все сливаться и которая будет продолжением ветки `master`. Ее будут тестировать и по готовности, во время релиза, все изменения будут сливать в `master`. Назовем ее `dev`.

Если требуется выпустить ветку `hotfix`, можно вернуться к состоянию `master` или к определенному тегу, создать там ветку `hotfix/version` и начать работу по исправлению критических изменений. Это не затрагивает проект и текущую разработку.

Для разработки фич удобно будет использовать ветки «feature/feature_name». Начинать ветку лучше с самых последних изменений и периодически подтягивать изменения из `dev` к себе. Чтобы сохранить историю простой и линейной, ветку лучше перестраивать на `dev` (`git rebase dev`).

Исправление мелких багов может идти напрямую в `dev`. Для мелких багов и изменений рекомендуется создавать локально временные ветки. Их не надо отправлять в глобальный репозиторий. Это только ваши временные ветки.

Схема исправления багов:

- Исправление в ветке `dev` или отдельной,
- Получение изменений из `origin` (`git fetch origin dev`),
- Перестройка изменений на последнюю версию (`git rebase -i origin/dev`),
- Передача изменений в `origin` (`git push origin dev`).

Пример работы с фичами (это может быть большой баг, над которым работают несколько человек).

4.5.1. Коротко-живущие ветки

Обычно ветка создается с самого последнего варианта кодовой базы. В нашем случае с ветки `dev`

```
(dev)$ git fetch origin dev  # извлекаем изменения
```

Создаем ветку `feature/feature1` на базе ветки `origin/dev`, но не связываем их

```
(dev)$ git branch --no-track feature/feature1 origin/dev
```

Отправляем новую ветку в общий репозиторий, чтобы она была доступна другим

```
(dev)$ git push origin feature/feature1
```

Теперь работу над `feature1` можно вести в ветке `feature/feature1`. Спустя некоторое время в нашей ветке будет много коммитов, и работа над `feature1` будет закончена. При этом в `dev` тоже будет много изменений. И наша задача отдать наши изменения в `dev`.

Когда работа закончена, один разработчик должен предупредить другого, что он собирается «слить» изменения в `dev` и, например, удалить ветку

```
(feature/feature1)$ git fetch origin  # обновиться на всякий случай
(feature/feature1)$ git rebase -i origin/dev  # перестроить ветку feature/feature1 на ветке
    origin/dev. Нужно писать origin/dev, а не dev потому что после обновления ветка ушла вперед,
     а локальная ветка осталась на месте
(feature/feature1)$ git checkout dev
(dev)$ git merge feature/feature1  # сливаем в ветку dev
```

Делаем `push` и убираем ненужное

```
(dev)$ git push origin dev
(dev)$ git push origin :feature/feature1  # удаляем удаленную ветку
(dev)$ git branch -d feature/feature1  # удаляем локальную ветку
```

4.5.2. Долго-живущие ветки

Сложность долго-живущих веток в их поддержке и актуализации. Если делать все как описано выше то, вероятно быть беде: время идет, основная ветка меняется, проект меняется, а ваша фича основана на очень старом варианте. Когда настанет время выпустить фичу, она будет настолько выбиваться из проекта, что объединение может быть очень тяжелым или, даже, невозможным.

Именно поэтому, ветку нужно обновлять.

4.6. Команда `git reset`

В случае конструкции `git reset HEAD~` команда `git reset` на *первом шаге* переместит текущую локальную ветку на один элемент назад, то есть в данном случае ветка `master` будет указывать на предпоследний коммит. И в случае команды `git reset --soft` этот же шаг окажется последним. Другими словами, команда `git reset --soft HEAD~` просто перемещает ветку на один коммит назад (*отменяет последний коммит*, т.е. отменяет действие команды `git commit`), не затрагивая ни *область индексирования*, ни *рабочую папку*. Если сейчас посмотреть статус `git status`, то файлы, которые вошли в отмененный коммит, будут проиндексированны и готовы к фиксации. Этим можно воспользоваться например так: создадим новый файл, проиндексируем его (`git add`), а затем все зафиксируем (`git commit`) вместе со старыми проиндексированными файлами.

Следующим действием команды `git reset` станет *обновление области индексирования* путем добавления туда содержимого снимка, на который нацелен указатель `HEAD`. Если команда вызывается без аргументов (т.е. `git reset HEAD~`), то на этом шаге работа команды заканчивается.

Команда `git reset HEAD~` не только отменяет последний коммит, но и убирает из области индексирования все находившиеся там файлы. То есть этот вариант команды `git reset` отменяет как `git commit`, так и `git add`.

Третьим действием команды `git reset` станет приведение рабочей папки к виду, который имеет область индексирования. До этой стадии команда работает при наличии параметра `--hard`. То есть `git reset --hard HEAD~` убирает последний коммит, результаты работы команд `git add` и `git commit` и все наработки из рабочей папки. Важно понимать, что только параметр `--hard` делает команду `git reset` по настоящему опасной. Это один из немногочисленных случаев, когда `Git` реально удаляет данные.

Заключение команда `git reset` в определенном порядке переписывает три дерева, останавливаясь в указанном месте:

- перемещает ветку, на которую нацелен указатель `HEAD` (и останавливается при наличии `--soft`),
- приводит вид области индексирования в соответствие с данными, на которые нацелен указатель `HEAD` (и без параметра `--hard` на этом останавливается),
- приводит вид рабочей папки в соответствие с видом области индексирования.

Кроме того команде `git reset` можно передавать путь. В этом случае команда пропускает первый этап и ограничивает свою работу определенным файлом или набором файлов.

Такое поведение имеет смысл, ведь указатель `HEAD` не может быть нацелен частично на один коммит, а частично на другой. А вот частичное обновление области индексирования и рабочей папки – вполне реальная операция, поэтому команда `git reset` сразу переходит к этапам 2 и 3.

Команда `git reset file_name` обратна по смыслу команде `git add file_name`, т.е. `git add` индексирует файл, а `git reset` отменяет индексирование файла.

Можно явно указать коммит, из которого следует брать версию файла, например, `git reset eb43bf`.

4.7. Команда `git fetch`

Команда `git fetch` извлекает данные из удаленного репозитория без слияния. Общий синтаксис команды `git fetch` выглядит следующим образом

```
git fetch [<опции>] [<имя_удаленного_репозитория> [<источник>:<цель>...]]
```

Здесь выражение `<источник>:<цель>` – *спецификация ссылок*, где `<источник>` – шаблон для ссылок на сервере, а `<цель>` – место, куда эти ссылки будут записываться локально.

Посмотреть спецификацию ссылок, которая используется в проекте, можно так

```
$ cat .git/config
# выведет
[core]
    repositoryformatversion = 0
    filemode = false
    bare = false
    ...
    ignorecase = true
[remote "origin"]
    url = https://github.com/LeorFinkelberg/Cheat_sheet_bash.git
    fetch = +refs/heads/*:refs/remotes/origin/*  # <--
[branch "origin"]
```

```
    remote = origin
    merge = refs/heads/master
```

Строка, которая начинается с `fetch =`, обеспечивает проецирование имен в удаленном репозитории на имена в вашей локальной папке `.git`.

По сути, она сообщает системе `Git`: «Все, что в *удаленном* репозитории находится по адресу `refs/heads`, должно попасть в папку моего *локального* репозитория `refs/remotes/origin`»

Например, если требуется извлечь данные из *удаленной* ветки `master` *удаленного* репозитория с именем `origin` в *локальную* ветку `mymaster`, то можно воспользоваться конструкцией

```
git fetch origin master:refs/remotes/origin/mymaster
```

Можно задать набор спецификаций. Извлечь данные из нескольких веток посредством командой строки можно так

```
git fetch origin master:refs/remotes/origin/mymaster \
                 topic:refs/remotes/origin/topic
```

Посмотреть коммиты новой ветки можно, указав путь

```
git log --oneline refs/remotes/origin/mymaster
```

Задать множественную спецификацию ссылок для извлечения данных можно и в конфигурационном файле. Чтобы все время получать обновления из веток `master` и `experiment`, добавьте следующие строки в конфигурационный файл

.git/config

```
...
[remote "origin"]
    url = https://github.com/LeorFinkelberg/Cheat_sheet_bash.git
    fetch = +refs/heads/master:refs/remotes/origin/master
    fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

Если группа тестирования работает в некоторых ветках, а вам нужно получить данные из ветки `master` и из всех веток, используемой этой группой, добавьте в раздел конфигурации следующие строки

.git/config

```
...
[remote "origin"]
    url = https://github.com/LeorFinkelberg/Cheat_sheet_bash.git
    fetch = +refs/heads/master:refs/remotes/origin/master
    fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

4.8. Команда `git push`

Здорово, что извлекать данные можно с помощью ссылок, разделенных по пространствам имен, но сначала нужно дать группе тестирования возможность отправлять свои ветки в пространство имен `qa/`. Эта задача решается через спецификации ссылок для команды `push`.

Вот как отправить ветку `master` (*локальную*), принадлежащую группе тестирования, в ветку `qa/master` на *удаленном* сервере

```
git push origin master:refs/heads/qa/master
```

Если нужно, чтобы система `Git` автоматически делала это при каждом запуске команды `git push origin`, добавьте в конфигурационный файл значения переменной `push`

.git/config

```
...
[remote "origin"]
    url = https://github.com/LeorFinkelberg/Cheat_sheet_bash.git
    fetch = +refs/heads/master:refs/remotes/origin/master
    push = +refs/heads/master:refs/heads/qa/master
```

В результате команда `git push origin` по умолчанию будет отправлять *локальную* ветку `master` в ветку `qa/master` на *удаленном* сервере.

По сути мы просто копируем файл `master` из `refs/heads/`, который содержит хеш-код последнего коммита текущей ветки, в `refs/heads/qa/`.

4.9. Ликвидация ссылок

Спецификацию ссылок, кроме всего прочего, можно использовать для удаления ссылок с сервера. Это может выглядеть так

```
git push origin :topic
```

Так как спецификация ссылок задается в формате `<источнк>:<цель>`, опустив часть `<источник>`, мы по сути, скажем, что тематическую ветку на удаленном сервере следует сделать пустой, и это приведет к ее ликвидации.

Удалить удаленную ветку после перемещения

```
git push origin :feature/feature1
```

И удалить эту ветку локально

```
git branch -d feature/feature1
```

5. Конфликты слияния

5.1. Создание коммитов слияния

Перед потенциально конфликтным слиянием первым делом следует по возможности очистить рабочую папку. Незаконченные наработки желательно либо *зафиксировать* во временной ветке, либо *скрыть*. Если на момент слияния в рабочей папке присутствуют несохраненные данные, есть риск их потерять.

При слиянии веток, истории коммитов которых не имеют общей базы, потребуется специальный аргумент `--allow-unrelated-histories`, которые разрешает сливать такие ветки

```
git merge --allow-unrelated-histories
```

Если вы не ожидали конфликта слияния и не хотите занимается его разрешением, можно просто отменить результат слияния командой (в рабочей папке не должно быть незафиксированных изменений!)

```
git merge --abort
```

Затем можно выполнить команду (флаг `-b` дополнительно выводит имя текущей ветки)

```
git status -sb
```

Команда `git merge --abort` пытается вернуть все в состояние, которое имело место до попытки слияния. Но если перед слиянием в рабочей папке находились *незафиксированные* изменения, возвращение в исходное состояние может оказаться невозможным. Во всех остальных случаях оно прекрасно работает.

Если конфликт возникает преимущественно по причине пробельных символов, то может помочь специальный флаг `-Xignore-all-space` или `-Xignore-space-change`.

Еще бывает удобно сливать файлы в интерактивном режиме с помощью `--patch`

```
git checkout --patch otherbranch file_name.txt
```

Затем будет выведена строка

```
(1/1) Apply this hunk to index and worktree [y,n,q,a,d,s,e,?]?
```

В простейшем случае можно применить все изменения сразу, выбрав `y`, или отказаться от них, выбрав `n`. Но еще можно попросить систему разбить блок изменений (так называемый ханк) на атомарные изменения с помощью `s`, а затем принимать или отклонять изменения по отдельности.

Рассмотрим пример. Предположим, у нас есть пара долгоживущих веток (`master` и `mundo`), в каждой из которых присутствует несколько коммитов, но при этом попытки их слияния не удаются по причине конфликта.

При попытке слить содержимое ветки получаем конфликт

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge faild; fix confilcts and then commit the result.
```

Хотелось бы понять, что именно стало причиной проблемы. Открыв файл, мы увидим примерно такую картину

Так будет выглядеть файл hello.rb после ‘git merge mundo‘

```
#! /usr/bin/env ruby

# this function prints out A GREETING!
def hello
<<<<<<< HEAD
    puts 'hello Python'
=======
    puts 'HELLO MUNDO'
>>>>>>> whitespace
end
```

Теперь этот файл можно аккуратно изучить и поправить строки (удалить те, которые не нужны, и оставить те, которые нужны), а затем зафиксировать изменения с помощью `git commit`.

На обеих ветках в этот файл добавлялось некое содержимое, но некоторые коммиты модифицировали одни и те же строки, что и стало причиной конфликта.

На помощь приходит команда `git checkout --conflict`, которая повторно выгружает содержимое файла и заменяет маркеры конфликта слияния. Это может пригодится в ситуации, когда требуется сбросить маркеры и снова попробовать разрешить конфликт.

Параметру `--conflict` можно передать значения `diff3` и `merge` (последнее используется по умолчанию). Значения `diff3` заставляет `Git` выводить помимо «их», «нашей» еще и «базовую» версию файла

```
git checkout --confilct=diff3 hello.rb
```

После применения этой команды (применяется она сразу после `git merge`) файл будет выглядеть так

Так будет выглядеть файл hello.rb после 'git checkout –conflict'

```
#! /usr/bin/env ruby

# this function prints out A GREETING!
def hello
<<<<<<< ours
    puts 'hello Python'
||||||| base
    puts 'hello mundo'
=======
    puts 'HELLO MUNDO'
>>>>>>> theirs
end
```

Теперь файл можно поправить в ручную, удалив ненужные строки. Но требуется быть очень внимательным, потому как в этом режиме не все конфликты изменений будут отображаться. В отношении некоторых изменений `Git` сама принимает решение.

Команда `git checkout` также можно добавить параметры `--ours` и `--theirs`, которые позволяют быстро выбрать одну из версий файла, не выполняя слияния. Это особенно полезно в случае конфликта бинарных файлов, при которых можно выбрать одну из сторон.

Получить список всех уникальных коммитов, сделанных в любой из участвующих в слиянии веток можно следующим образом

```
git log --oneline --left-right HEAD...MERGE_HEAD
```

Можно вывести информацию об изменениях по каждому коммиту из сливаемых веток

```
git log --oneline --left-right -p HEAD...whitespace
```

Получить список только тех коммитов с каждой стороны слияния, которые касаются только файла, являющегося в данный момент причиной конфликта, можно, добавив параметр `--merge`

```
git log --oneline --left-right --merge
```

Если добавить параметр `-p`, то получим список изменений, внесенных в файл, в котором возник конфликт

```
git log -p --oneline --left-right --merge
```

5.2. Отмена результатов слияния

Если нежелательный *коммит слияния* присутствует только в вашем *локальном* репозитории, то проще и лучше всего переместить ветки, чтобы они указывали туда, куда нужно. Отменить последний коммит, а точнее отменить результат действия команды `git commit` (перемещаем ветку

на один коммит назад), команды `git add` (обновляем область индексирования, удаляя из области индексирования файлы отмененного коммита) и удалить файл из рабочей папки, можно с помощью `git reset` со специальным ключом `--hard`

```
git reset --hard HEAD~
```

Недостатком этого метода является внесения изменений в историю, что может привести к проблемам при наличии репозитория общего доступа. Если другие пользователи работают с коммитами, которые вы собираетесь переписать, от использования команды `git reset` лучше отказаться.

Кроме того, данный подход не будет работать, если с момента слияния был создан хотя бы один новый коммит, – перемещение ссылок, по сути, приведет к потере внесенных изменений.

Если перемещение указателя ветки в вашей ситуации не помогает, система `Git` дает возможность выполнить новый коммит, отменяющий все изменения, внесенные предыдущим коммитом. Эта операция называется восстановлением `revert`

```
git revert -m 1 HEAD
```

Флаг `-m 1` указывает, какой из предков является «основной веткой» и подлежит сохранению. После слияния в ветку, на которую нацелен указатель `HEAD`, у нового коммита будет два предка: первый с указателем `HEAD` (родитель 1) и второй – вершина сливаемой ветки (родитель 2).

Мы хотим отменить все изменения, внесенные слиянием родителя 2, сохранив все содержимое родителя 1.

Содержимое нового коммита `^M` полностью совпадает с содержимым коммита ветки `master`, то есть вы можете начать работу, как будто слияния никогда не было, только коммиты, не входившие в слияние, *все еще присутствуют в истории перемещений указателя* `HEAD` [1, стр. 282].

Лучше всего решить эту проблему отменой возвращения исходного слияния, то есть нужно добавить изменения, которые были отменены, и создать новый коммит слияния

```
git revert ^M
```

Иногда вместо решения конфликта было бы лучше, чтобы система `Git` просто выбрала одну из версий, проигнорировав все остальное. В этом случае к команде `git merge` следует добавить параметр `-Xours` или `-Xtheirs`

```
git merge -Xours mundo
```

В данном случае вместо того чтобы добавить маркеры конфликта в файл система просто выберет файл из соответствующей ветки. При этом все прочие изменения, не приводящие к конфликту, сливаются успешно.

Еще для того чтобы «откатить» успешное слияние (т.е. такое слияние, при котором не создается коммита слияния, а просто вносятся изменения в файлы) бывает очень удобно воспользоваться командой `git reflog`, чтобы посмотреть историю изменений, а затем можно воспользоваться той же командой `git reset --hard`[2], что бы перейти к нужному моменту истории, например

```
git reset --hard HEAD@{1}
```

Посмотреть как был разрешен конфликт можно с помощью

[2]Ключ `--hard` нужен для обновления рабочей папки, т.е. в файлах отмененного коммита будут отменены все изменения, связанные с этим коммитом. Другими словами файлы будут возвращены в состояние до появления отмененного коммита

```
git log --cc -p
```

Команда `git checkout -conflict` позволяет вернуть файл в состояние конфликта

```
git checkout --conflict=merge hello.rb
```

5.3. Слияние поддеревьев

Идея слияния поддеревьев состоит в том, что один из проектов проецируется во вложенную папку другого, и наоборот.

Рассмотрим пример добавления нового проекта в уже существующий, при этом код из первого проекта записывается в подпапку второго.

Первым делом добавим к нашему проекту приложение Rack. Мы присоединим его как удаленный репозиторий (`rack_remote`) и выгрузим его содержимое в отдельную ветку (`rack_branch`)

```
git remote add rack_remote https://github.com/rack/rack
git fetch rack_remote
git checkout -b rack_branch rack_remote/master
```

Теперь корень проекта Rack находится в ветке `rack_branch`, в то время как наш собственный проект находится в ветке `master`.

В рассматриваемой ситуации мы хотим выгрузить содержимое проекта Rack в подпапку нашего основного проекта. В `Git` это реализуется командной `git read-tree`. Она считывает корень дерева одной ветки в текущую область индексирования и рабочую ветку.

Мы просто вернемся в ветку `master` и извлечем содержимое `rack_branch` в подпапку `rack` нашей ветки `master` основного проекта

```
git checkout master
git read-tree --prefix=rack/ -u rack_branch
```

При фиксации все это будет выглядеть как результат добавления во вложенную папку всех файлов из проекта Rack – как будто мы просто скопировали их из архива. Все обновления проекта Rack можно получить, перейдя на его ветку и выполнив скачивание данных из удаленного репозитория

```
git checkout rack_branch
git pull
```

Скачанные изменения можно слить в нашу ветку ветку `master`. Чтобы извлечь изменения и предварительно заполнить сообщение фиксации, используем параметры `--squash` и `--no-commit` вместе с параметром *стратегии слияния поддеревьев* `-s`

```
git checkout master
git merge --squash -s subtree --no-commit rack_branch
```

Ветки можно хранить в нашем репозитории с другими проектами, периодически сливая их в наш проект как поддеревья. В некотором смысле это удобно: например, фиксация состояния всего кода происходит в одном месте.

Для просмотра разницы содержимого подпапки `rack` и кода в ветке `rack_branch` (это нужно, чтобы понять, не пришло ли время выполнить слияние) следует выполнить команду

```
git diff-tree -p rack_branch
```

6. Подмодули

7. Пакеты

Система `Git` умеет «упаковывать» данные в один файл. Существует разные сценарии, в которых такое поведение востребовано. Например, если в текущий момент нет доступа к общему серверу, а нужно переслать кому-то по электронной почте свои наработки, не передавая 40 коммитов командой `format-patch`.

Именно в таких случаях на помощь приходит команда `git bundle`. Она упакует в бинарный файл все данные, которые в обычной ситуации вы отправили бы на сервер командой `git push`, и этот файл можно будет без проблем переслать по электронной почте или записать на флешнакопитель, а затем распаковать в целевой хранилище

```
git bundle create repo.bundle HEAD master
```

Если репозиторий предназначен для клонирования на новом месте, в пакет добавляется еще и указатель `HEAD`, как и было сделано в данном случае.

Появится файл `repo.bundle` со всем необходимыми для воссоздания ветки `master` вашего репозитория. Теперь этот файл можно отправить по почте или записать на диск.

Чтобы восстановить репозиторий из бинарного файла его можно клонировать в папку

```
$ git clone repo.bundle repo
$ cd repo
$ git log --oneline
```

Если указатель `HEAD` не входит в список ссылок, при распаковке нужно будет указать `-b master` или другую присутствующую в пакете ветку, в противном случае система не будет знать, в какую ветку ей следует перейти, т.е.

```
git clone -b master repo.bundle repo
```

Если нужно передать не все коммиты, а только несколько последних, то можно указать диапазон интересующих коммитов. Для этого мы снова воспользуемся командой `git bundle create`, передав ей *имя* будущего *пакета* и *диапазон коммитов*, которые следует в этот пакет включить

```
git bundle create commits.bundle 389b3f2..master
```

В данном случае в пакет будут включены коммиты, начиная с первого потомка коммита `389b3f2` и заканчивая последним коммитом ветки.

Теперь в нашей папке появился файл `commits.bundle`. Если его отправить коллеге, тот сможет импортировать наши коммиты в исходный репозиторий, даже если там параллельно велась некая работа.

Перед импортированием пакета в репозиторий можно посмотреть его содержимое

```
git bundle verify commits.bundle
```

Команда проверяет корректность пакета и наличие всех предков коммитов, необходимых для правильного восстановления.

Чтобы посмотреть, с чем нам придется работать можно извлечь ветку `master` из переданного репозитория `commits.bundle` в другую ветку `other-master`

```
git fetch commits.bundle master:other-master
```

а затем

```
git checkout other-master
git log --oneline
```

8. Переменные среды

Git всегда работает внутри командной оболочки bash и пользуется для задания своего поведения некоторыми доступными в этой оболочке переменными среды.

Описание наиболее полезных переменных среды можно найти в [1, стр. 451]

8.1. Отладка

Git может выступать и как система слежения.

Переменная GIT_TRACE контролирует протоколирование действий, не поддающих ни в одну из заданных категорий

```
$ GIT_TRACE=true git lga
```

Переменная GIT_TRACE_PERFORMANCE отвечает за регистрацию сведений о производительности. Выводимые данные показывают, сколько времени занимали те или иные действия

```
$ GIT_TRACE_PERFORMANCE=true git gc
18:25:54.205230 trace.c:475             performance: 0.002114576 s: git command: git pack-refs
    --all --prune
18:25:54.233231 trace.c:475             performance: 0.009720393 s: git command: git reflog
    expire --all
18:25:54.278234 read-cache.c:2308       performance: 0.000061270 s:  read cache .git/index
Enumerating objects: 220, done.
Counting objects: 100% (220/220), done.
Delta compression using up to 8 threads
Compressing objects: 100% (87/87), done.
Writing objects: 100% (220/220), done.
Total 220 (delta 128), reused 220 (delta 128), pack-reused 0
18:25:54.536249 trace.c:475             performance: 0.272991152 s: git command: git
    pack-objects --local --delta-base-offset .git/objects/pack/.tmp-23424-pack
    --keep-true-parents --honor-pack-keep --non-empty --all --reflog --indexed-objects
    --unpack-unreachable=2.weeks.ago
18:25:54.576251 trace.c:475             performance: 0.326025363 s: git command: git repack -d
    -l -A --unpack-unreachable=2.weeks.ago
18:25:54.615253 read-cache.c:2308       performance: 0.000111965 s:  read cache .git/index
18:25:54.622254 trace.c:475             performance: 0.011453979 s: git command: git prune
    --expire 2.weeks.ago
18:25:54.640255 trace.c:475             performance: 0.001036295 s: git command: git worktree
    prune --expire 3.months.ago
18:25:54.658256 trace.c:475             performance: 0.000840357 s: git command: git rerere gc
18:25:54.694258 trace.c:475             performance: 0.517178791 s: git command: 'C:\Program
    Files\Git\mingw64\bin\git.exe' gc
```

Переменная GIT_TRACE_SETUP показывает информацию, собранную системой Git о репозитории и среде, с которой взаимодействует

```
$ GIT_TRACE_SETUP=true git status
22:06:55.450824 trace.c:375             setup: git_dir: .git
22:06:55.451824 trace.c:376             setup: git_common_dir: .git
22:06:55.451824 trace.c:377             setup: worktree: E:/[WorkDirectory]/[LaTeX_project]
    /Cheat_sheet_Git
22:06:55.451824 trace.c:378             setup: cwd: E:/[WorkDirectory]/[LaTeX_project]
    /Cheat_sheet_Git
22:06:55.451824 trace.c:379             setup: prefix: (null)
22:06:55.451824 chdir-notify.c:65       setup: chdir from 'E:/[WorkDirectory]/[LaTeX_project]
    /Cheat_sheet_Git' to 'E:/[WorkDirectory]/[LaTeX_project]/Cheat_sheet_Git'
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
    (use "git add <file>..." to update what will be committed)
    (use "git restore <file>..." to discard changes in working directory)
        modified:   cheat_sheet_git.pdf
        modified:   cheat_sheet_git.tex
        modified:   style_packages/podvoyskiy_article_extended.sty

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

9. Конструкции Git

9.1. Создать новый репозиторий

Создать новый репозиторий в текущей директории

```
git init
```

Создать новый репозиторий в указанной директории

```
git init path_to_dir
```

9.2. Клонирование репозитория

Клонировать удаленный репозиторий в директорию с именем folder_name

```
git clone https://github.com/.../git-commands.git folder_name
```

9.3. Настройка Git

Задать глобальные настройки можно следующим образом

```
git config --global user.name "[name]"
git config --global user.email "[email address]"
```

Для того чтобы Git при слияниях, которые сопроваждаются разрешением конфликтов, использовал кэш следует воспользоваться конструкцией

```
git config --global rerere.enabled true
```

9.4. Информация о снимке

Вывести информацию о текущем снимке `HEAD`

```
git cat-file -p HEAD
```

Вывести список файлов, попавших в родительский снимок `HEAD`, с рекурсией по поддиректориям

```
git ls-tree -r HEAD~
```

Вывести список директорий

```
git ls-tree -d HEAD
```

Вывести текущее содержимое индекса (снимок, предложенный для следующего коммита)

```
git ls-files -s
```

Вывести список модифицированных файлов

```
git ls-files -m
```

Вывести список «прочих» файлов, т.е. файлов, находящихся в текущей директории, но не вошедших в снимок

```
git ls-files -o
```

9.5. Добавление файлов в область индексирования

```
git add file_name.py
git add .
```

9.6. Фиксация изменений

Зафиксировать измененное состояние

```
git commit -m 'Initial commit'
```

Зафиксировать измененное состояние, пропустив область индексирования

```
git commit -a -m 'Some comment'
```

Исправить комментарий последнего коммита. Комментарий последнего коммита будет перезаписан

```
git commit -m 'New some comment' --amend
```

Чтобы исправить комментарий коммита (или комментарии нескольких коммитов), созданного некоторое время назад (т.е. речь идет не о последнем коммите) следует перейти в интерактивный режим с помощью команды

```
git rebase -i HEAD~15
```

затем в открывшемся файле заменить «pick» на «reword» (изменить комментарий коммита), сохранить файл и закрыть его.

Далее для каждого коммита (помеченного «reword») можно будет исправить комментарий. После следует сохранить файл и закрыть его. В завершении требуется залить данные на удаленный сервер в принудительном режиме, т.е.

```
git push --force
```

9.7. Удаление файлов

Удалить файл из *области индексирования* и заодно удалить указанный файл из рабочей папки. Чтобы система `Git` перестала работать с файлом, его нужно удалить из числа отслеживаемых (точнее, убрать из области индексирования) и зафиксировать данное изменение

```
$ git rm file_name.py
```

Удалить файл из области индексирования[3], но оставить его в рабочей папке. Данная команда в отличие от `git reset HEAD file_name.py` может использоваться как до первой фиксации (`git commit`), так и после

```
git rm --cached file_name.py
```

Удалить все файлы с расширением `.log`[4] из директории `log/`

```
git rm log/\*.log
```

9.8. Переименование файлов

Переименовать файл

```
git mv old_file_name new_file_name
```

Переименовать файл с использованием `{..}`

```
git mv test_file{,_new}.py
```

9.9. Просмотр истории коммитов

Вывести историю коммитов

```
git log
```

Вывести историю коммитов, ограничившись последними двумя, с указанием разницы, которую внес каждый коммит

```
git log -p -2
```

Вывести историю коммитов с краткой статистикой

[3] `Git` перестает следить за файлом, т.е. он становится *неотслеживаемым*!
[4] Символ `*` экранируется

```
git log --stat
```

Вывести историю коммитов с указанием сокращенного варианта хеш-кода коммита и комментария

```
git log --pretty=format:'%h %s'
```

Вывести историю коммитов за последние 2 недели

```
git log --since=2.week
```

Вывести историю коммитов с захватом интересующего слова в коммите, ограничившись последними двумя

```
git log --grep='key word' -2
```

Вывести историю коммитов, которые попали в заданный временной диапазон

```
git log --since='2020-03-01 10:00' --before ='2020-03-01 11:00'
```

Вывести историю коммитов с указанием сокращенного хеш-кода коммита, тегов, текущей векти и собственно коммита

```
git log --oneline
```

Вывести историю коммитов, показывая места расположения указателей и точек расхождения

```
git log --oneline --decorate --all --graph
```

Отобразить только те не подвергавшиеся слиянию коммиты из ветки `origin/master`, которых нет в ветке `issue54`

```
git log --no-merges issue54..origin/master
```

Провести слияние веток только с помощью перемотки вперед[5] (fast-forward)

```
git merge --ff-only branch_name
```

Вывести информацию о том чем ветка `origin/master` будет отличаться от ветки `master`[6] (каких коммитов нет в ветке `origin/master`)

```
git log origin/master..master -p
```

Еще данный синтаксис часто используется для просмотра информации, которую вы собираетесь отправить на удаленный сервер

```
git log origin/master..HEAD
```

или короткий вариант

```
git log origin/master..
```

[5]Перемотка вперед возможна, если на графе коммитов существует прямой путь от последнего коммита более короткой ветки до последнего коммита более длинной ветки

[6]Этот прием бывает полезен тогда, когда требуется предварительно посмотреть данные, которые будут слиты в ветку

так как `Git` вместо пропущенного фрагмента подставляет `HEAD`.

Вывести информацию из журнала ссылок[7]

```
git log -g master
```

9.10. Отмена индексирования

Отменить индексирование файла (файл удаляется из области индексирования). Данная команда может применяться только после первой фиксации (`git commit`)

```
git reset HEAD file_name.py
```

9.11. Отмена коммитов и перемещение по истории

Замечание

Все коммиты, которые были отправлены в удаленный репозиторий, должны отменяться ТОЛЬКО новыми коммитами (`git revert`), чтобы избежать проблем с историей разработки у других разработчиков

Создать новый коммит, отменяющий изменения последнего коммита без запуска редактора

```
git revert HEAD --no-edit
```

Создать новый коммит, отменяющий изменения, внесенные коммитом с указанным хеш-кодом

```
git revert b9533bb --no-edit
```

Все команды, приведенные ниже можно выполнять ТОЛЬКО, если коммиты еще *не были отправлены* в удаленный репозиторий

```
git commit --amend -m 'New commit-message' # создать новый коммит на основе обновленной области
    индексирования и задать новое сообщение фиксации
git reset --hard @~  # передвинуть ветку на предыдущий коммит, обновить область индексирования и
     рабочую папку; эквивалентно git reset --hard HEAD~
git reset --hard 75e2d51  # то же самое, но на коммит с указанным хеш-кодом
```

9.12. Восстановление изменений

Восстановить в рабочей директории указанный файл на момент указанного коммита

```
git checkout 34534534 index.html
```

Скопировать на текущую ветку изменения из указанного коммита и *зафиксировать* эти изменения

```
git cherry-pick 454535
```

Скопировать на текущую ветку изменения из ветки `master` (2 последних коммита)

```
git cherry-pick master~2..master
```

Скопировать на текущую ветку изменения из указанного коммита, но *не фиксировать*

[7] Этот способ работает только для данных, которые все еще находятся в журнале ссылок, поэтому его невозможно исопльзовать для просмотра коммитов, возраст которых превышает несколько месяцев

```
git cherry-pick -n 3453455
```

Прервать конфликтный перенос коммитов

```
git cherry-pick --abort
```

Продолжить конфликтный перенос коммитов

```
git cherry-pick --continue
```

9.13. Работа с удаленными репозиториями

Добавить удаленный репозиторий под коротким именем `pb`. Теперь вместо полного URL можно использовать имя `pb`

```
git remote add pb https://github.com/paulboone/ticgit
```

Извлечь данные из удаленного репозитория. Эта команда связывается с удаленным проектом и извлекает оттуда все пока отсутствующие в локальном репозитории данные. Она *не выполняет* автоматического слияния с ветками, и вообще никак не затрагивает эти ветки

```
git fetch origin
```

Отправить данные локальной ветки `master` на удаленный репозиторий `origin`

```
git push origin master
```

Передать данные от локальной ветки `serverfix` в ветку `awesomebranch` на удаленном репозитории

```
git push origin serverfix:awesomebranch
```

Вывести информацию о конкретном удаленном репозитории `origin`

```
git remote show origin
```

Изменить имя удаленного репозитория с `pb` на `paul`. Теперь к ветке `pb/master` нужно будет обращаться по имени `paul/master`

```
git remote rename pb paul
```

Удалить ссылку на удаленный репозиторий

```
git remote rm paul
```

9.14. Работа с тегами

Вывести список доступных тегов

```
git tag
```

Вывести список тегов, отвечающих поисковому шаблону

```
git tag -l 'v1.8.*'
git tag -l 'v0.2*.*'
```

Создать тег с комментарием. Тег привязывается к последнему коммиту

```
git log -a v1.4 -m 'My version 1.4'
```

Вывести информацию по тегу

```
git show v1.4
```

Создать легковесный тег (просто не указываются `-a`, `-s`, `-m`)

```
git tag v1.4-lw
```

Отправить все теги на удаленный репозиторий. По умолчанию команда `git push` не отправляет теги на удаленный репозиторий

```
git push origin --tags
```

9.15. Работа с ветками

Вывести список существующих веток

```
git branch
```

Показать все имеющиеся ветки (в том числе на удаленных репозиториях)

```
git branch -a
```

Показать список веток и последние коммиты в каждой из них

```
git branch -vv
```

Переименовать локально ветку `old_branch_name` в `new_branch_name`

```
git branch -m old_branch_name new_branch_name
```

Переименовать локально текущую ветку в `new_branch_name`

```
git branch -m new_branch_name
```

Переименовать ветку в удаленном репозитории

```
git push origin :old_branch_name new_branch_name
```

Завершить процесс переименования

```
git branch --unset-upstream
```

Создать новую ветку

```
git branch testing
```

Переключиться на новую ветку

```
git checkout testing
```

Создать новую ветку и тут же переключитсья на нее

```
git checkout -b iss53
```

Внедрить внесенные изменения в готовый код

```
git merge hotfix
```

Удалить ветку

```
git branch -d hotfix
```

Вывести ветки, НЕ объединенные с текущей веткой

```
git branch --no-marged
```

Создать *локальную копию ветки* `serverfix` на основе *удаленной ветки* `origin/serverfix`. В результате будет получена локальная ветка, которая начинается там же, где и ветка `origin/serverfix`

```
git checkout -b serverfix origin/serverfix
```

или альтернативный вариант

```
git checkout --track origin/serverfix
```

Создать локальную копию ветки с именем `sf` на основе удаленной ветки `origin/serverfix`. Теперь локальная ветка `sf` поддерживает автоматический обмен данными с удаленной веткой `origin/serverfix`

```
git checkout -b sf origin/serverfix
```

Вывести только те коммиты, которых нет в первой ветке (ветка `master`)

```
git log master..contrib
```

или так

```
git log contrib --not master
```

или так

```
git log ^master contrib
```

Вывести только те наработки из *тематической ветки*, которые появились там после расхождения с веткой `master`

```
git diff master...contrib
```

Вывести изменения, которые присутствуют только в ветке `master`

```
git diff origin/master..master
```

Вывести информацию по различиям файлов в разных коммитах

```
git diff HEAD:file_name.txt HEAD~:file_name.txt
```

Вывести информацию по непроиндексированным изменениям в конкретном файле

```
git diff file_name.txt
```

Для обращения к существующей ветке можно использовать краткую форму `@{u}`. К примеру, если мы следим из ветки `master` за веткой `origin/master`, то для краткости можно писать так

```
git merge @{u}
```

вместо

```
git merge origin/master
```

Вывести список веток *наблюдения*. Все цифры представляют собой показатели, зафиксированные в момент последнего скачивания данных с каждого сервера. Данная команда не обращается к серверам, а просто сообщает локальные данные из кэша. Для получения актуальной информации о количестве новых коммитов на локальных и удаленных ветках следует извлечь данные со всех удаленных серверов и только затем воспользоваться этой командой, т.е.

```
git fetch --all
git branch -vv
 iss53 7e424c3 [origin/iss53: ahead 2] forgot the brackets
 master 1ae2a45 [origin/master] deploying index fix
 serverfix 5ea463a [teamone/server-fix-good: ahead 3, behind] this should do it
 ...
```

9.16. Работа со ссылками

Вывести список всех ссылок локального репозитория

```
git show-ref
```

9.17. Отправка данных на удаленный репозиторий

Для того чтобы отправить данные из локального репозитория на удаленный следует использовать конструкцию

```
git push origin master
```

но предварительно необходимо слить данные из удаленного репозитория с помощью команды `git pull origin master` (однако делать надо это очень осторожно и в общем случае лучше воспользоваться сначала командной `git fetch`, а затем уже `git merge`). Кроме того, может потребоваться специальный флаг `--allow-unrelated-histories`, разрешающий слияние несвязанных историй (то есть историй коммитов, не имеющих общей базы)

```
git pull origin master --allow-unrelated-histories
```

9.18. Перемещение данных

Изменения, зафиксированные в одной ветке, повторить в другой ветке (в `Git` это называется *перемещением*). Например, чтобы повторить изменения из ветки `experiment` в ветке `master`, следует сначала перейти в ту ветку, из которой требуется перенести изменения (ветка `experiment`), а затем воспользоваться командой `git rebase`[8]

```
git checkout experiment
git rebase master
```

[8] Работает это следующим образом: ищется общий предок двух веток (текущей ветки и ветки, в которую выполняется перемещение), вычисляется разница, вносимая каждым коммитом текущей ветки, и сохраняется во временных файлах. После этого текущая ветка сопоставляется тому же коммиту, что и ветка, в которую осуществляется перемещение, и одно за другим происходят все изменения

Внести изменения клиентской части (ветка `client`) в окончательную версию кода (ветка `master`), оставив изменения серверной части (ветка `server`) для дальнейшего тестирования. Другими словами, взять изменения клиентской части, не связанные с изменениями на серверной стороне, и воспроизвести их в ветке `master` можно следующим образом[9]

```
git rebase --onto master server client
```

Общий синтаксис команды

```
git rebase --onto <на_какую_ветку_перестроить>
                      <с_какого_коммита_или_ветки> <по_какой_коммит_или_ветку>
```

Переместить изменения из ветки `server` в ветку `master`, вне зависимости от того, в какой ветке вы находитесь, позволяет команда `git rebase [main_branch] [topic_branch]`. Эта команда переключает на тематическую ветку (в данном случае – на ветку `server`) и воспроизводит ее содержимое в основной ветке (`master`)

```
git rebase master server
```

Замечание

При перемещении изменений из одной ветки в другую, нужно перейти на ту ветку, *из которой* планируется переместить изменения

9.19. Перемещение отдельного коммита

Взять представленные в коммите изменения и попытаться применить их в текущей ветке. Команда извлечет изменения, появившиеся в коммите, но при этом измениться контрольная сумма SHA-1 коммита, так как у него другая дата применения

```
git cherry-pick e43a6fd3e9488...
```

9.20. Удаление коммитов

Для того чтобы удалить послдений коммит следует сначала удалить коммит в локальном репозитории

```
git rebase -i HEAD~2
```

а затем отправить данные в форсированном режиме на удаленный репоизторий

```
git push origin +master --force
```

Замечание

После удаления коммита или после изменения комментария коммита обязательно нужно «залить» обновления на удаленный сервер с помощью `git push origin master --force`

---

[9] По сути, команда приказывает «перейти в ветку `client`, найти исправления от общего предка веток `client` и `server` и повторить их в ветке `master`»

9.21. Просмотр информации по коммитам

Если требуется вывести информацию по коммиту (например, требуется выяснить что было удалено/добавлено в этот коммит), то можно обратиться к коммиту через его хеш-код

```
git show 06e6bbc
```

Информацию по последнему коммиту можно посмотреть следующим образом

```
git show master
```

9.22. Ссылки на предков

Для просмотра *предыдущего коммита* достаточно написать `HEAD^`, что означает «родитель HEAD»

```
git show HEAD^
```

Вывести информацию по коммиту указанного файла на момент предыдущего от HEAD коммита

```
git show @~:file_name.txt
```

Показать самый последний коммит, в описании которого встречает указанная строка

```
git show -s :/'cherry-pick'
```

Посмотреть как выглядит один и тот же файл на разных ветках можно так

```
$ git show master:file_name.py
$ git show alterbranch:file_name.py
```

Для просмотра всех предков указанного коммита следует воспользоваться конструкцией

```
git log j345345^@
```

Другое распространенное обозначение *предка* – символ `~`. Он также соответствует *ссылке на первого родителя*, поэтому записи `HEAD^` и `HEAD~` эквивалентны. А вот если указать номер после символа `~`, то проявятся различия между `~` и `^`.

Например, запись `HEAD~2` означает «первый предок первого предка», при этом происходит переход от заданного предка вглубь указанное число раз, т.е. `HEAD~3` укажет на четвертый[10] от конца ветки коммит.

После символа `^` можно указать число: например, запись `d921970^2` означает «второй предок коммита d921970». Этот синтаксис применяется только в случае *коммитов слияния*, у которых существует несколько предков. *Первый родитель* – это ветка, на которой вы находились в момент слияния, а *второй родитель* – коммит на ветке, которая подверглась слиянию

```
git show d921970^2
```

Указанные обозначения можно комбинировать. К примеру, второго родителя четвертого от конца ветки коммита (при условии, что это коммит слияния) можно получить, написав `HEAD~3^2`.

[10] Так как отсчет ведется, начиная со второго коммита от конца ветки

9.23. Диапазоны коммитов

Вывести все коммиты, достижимые по ссылке `refA` или `refB`, но не достижимые по ссылке `refC`

```
git log refA refB ^refC
git log refA refB --not refC
```

Вывести только те коммиты, которые есть либо в ветке `master`, либо в ветке `experiment`, но не в обеих ветках одновременно

```
git log master...experiment
```

С этой командой часто используют параметр `--left-right`, позволяющий посмотреть, с какой стороны диапазона находится каждый коммит

```
git log --left-right master...experiment
```

9.24. Скрытие и очистка

Часто во время работы над проектом, все еще находится в беспорядочном состоянии, возникает необходимость перейти в другую ветку и поработать над другим аспектом. Проблема в том, что фиксировать работу, сделанную наполовину, чтобы позже к ней вернуться вы не хотите. В такой ситуации на помощь приходит команда `git stash`.

Если, к примеру, вы отредактируете два файла и только один из них проиндексируете без фиксации результатов своей работы, то с помощью команды

```
git stash save
```

можно будет перейти на другую ветку, скрыв наработки в буфере.

Замечание

По умолчанию команда `git stash` сохраняет только файлы из *области индексирования*

Теперь можно легко менять ветки и работать над другими фрагментами проекта – все изменения хранятся в стеке. Увидеть содержимое позволяет команда

```
git stash list
```

Вернуть спрятанные в буфер изменения в рабочее состояние можно командой

```
git stash apply
```

Если требуется вернуться к работе над версией, сохраненной в буфере ранее, следует указать ее номер

```
git stash apply stash@{2}
```

Замечание

Вообще говоря, нет необходимости возвращать содерижмое буфера в чистый рабочий каталог и в ту же ветку, из которой они были сохранены. Можно скрыть изменения одной ветки, перейти в другую и попытаться вставить изменнное состояние туда

После извлечения информации из буфера файлы, которые до помещения в буфер были проиндексированы, автоматически в это состояние не вернуться. Чтобы сразу вернуть данные из буфера в исходное состояние, нужно написать

```
git stash apply --index
```

При этом команда `apply` только возвращает данные в ветку, но из стека они никуда не деваются. Убрать их из стека позволяет команда `git stash drop` с именем удаляемого файла

```
git stash drop stash@{0}
```

Врочем, существует также команда

```
git stash pop
```

которая возвращает сохраненную в буфере информацию в ветку и немедленно удаляет ее из буфера.

9.24.1. Более сложные варианты скрытия

Чтобы не скрывать данные, которые были проиндексированы командой `git add`, следует написать

```
git stash save --keep-index
```

Команда `git stash` по умолчанию сохраняет только данные из области индексирования, но параметр `--include-untracked` или `-u` заставляет систему `Git` сохранять также все *неотслеживаемые файлы*.

Для того чтобы в интерактивном режиме указать `Git` какие файлы нужно скрыть, а какие нет, следует воспользоваться конструкцией

```
git stash save --patch
```

9.25. Отмена скрытых изменений

Может возникнуть ситуация, когда после возвращения изменений из буфера вы выполняете некую работу, а затем хотите отменить изменеия, внесенные из буфера. Сделать это можно следующим образом: сначала нужно извлечь связанные с буфером исправления, а затем применить их в реверсивном виде

```
git stash show -p stash@{0} | git apply -R
```

Замечание

Если скрыть некие наработки, оставить их на некоторое время в буфере, а тем временем продолжить работу в ветке, из которой была скрыта информация, в итоге можно столкнуться с ситуацией, когда просто взять и вернуть данные из буфера не удастся.

Намного проще протестировать скрытые изменения командой `git stash branch branch_name`. Она создает новую ветку, переходит к коммиту, в котором вы находились на момент скрытия работы, копирует в новую ветку содержимое буфера и очищает его, если изменения прошли успешно. Это удобный спсоб легко восстановить скрытые изменения и продолжить работу с ними в новой ветке

9.26. Принудительно перезаписать локальные файлы

Если требуется локальные файлы перезаписать файлами с удаленного сервера, то алгоритм следующий

```
git fetch --all
get reset --hard origin/master
```

Команда `git fetch --all` скачивает отсутствующие файлы с удаленного репозитория без попытки слить или переместить данные, а `git reset --hard origin/master` «сбрасывает» ветку `master`. Опция `--hard` изменяет все файлы в рабочем дереве таким образом, чтобы они совпадали с файлами из `master/origin`.

9.27. Очистка рабочей папки

В некоторых ситуациях лучше не скрывать результаты своего труда или файлы, а избавиться от них. Это можно сделать командой `git clean`. Удаление требуется, чтобы убрать мусор, сгенерированный путем слияния или внешними инструментами, или чтобы избавиться от артефактов сборки в процессе ее очистки.

С этой командой надо быть крайне аккуратным, так как она предназначена для удаления неотслеживаемых файлов из рабочей папки. Даже если вы передумаете, восстановить содержимое таких файлов, как правило, будет невозможно. Безопаснее воспользоваться командой `git stash --all`, скрывающей из папки все содержимое, но с последующим его сохранением в буфере.

Предположим, вы все-таки хотите удалить командой `git clean` мусорные файлы или очистить вышу рабочую папку. Для удаления из этой папки всех *неотслеживаемых* файлов используйте команду `git clean -f -d`[11], которая полностью очищает папку, убирая не только файлы, но и вложенные папки.

Бывает полезно перед действительным удалением посмотреть на результа имитационного удаления. Сделать это можно, добавив ключ `-n`, т.е.

```
git clean -d -n
```

По умолчанию команда `git clean` удаляет только *неотслеживаемые* файлы, не добавленные в список игнорированных, т.е. любой файл, имя которого совпадает с шаблоном в файле `.gitignore`, сохраниться.

Чтобы удалить и их, например убрав все генерируемые в процессе сборки файлы с расширением `.o` с целью полной очистки сборки, нужно добавить параметр `-x`

```
git clean -d -n -x
```

9.28. Подготовка устойчивой версии

Если требуется создать архив последнего состояния кода для тех пользователей, у которых отсутствует система Git, то это можно сделать командой `git archive`

```
# tarball-архив
$ git archive master --prefix='project/' | gzip > $(git describe master).tar.gz
# zip-архив
```

[11] Параметр `-f` означает принудительное удаление

```
$ git archive master --prefix='project/' --format=zip > $(git describe master).zip
```

9.29. Подписи с помощью GPG

9.29.1. Общие сведения

GPG (также известный как GnuPG) создавался как свободная альтернатива несвободному PGP. Утилита GPG может испольоваться для симметричного шифрования, но в основном используется для ассиметричного шифрования информации.

Если кратко, то при симметричном шифровании для шифроваки и расшифровки сообщения используется один ключ, а при ассиметричном шифровании используется два ключа – публичный и приватный. Публичный используется для шифрования и его мы можем дать своим друзьям, а приватный – для расшифровки, и его вы должны хранить в надежном месте.

При такой схеме расшифровать сообщение может только владелец приватного ключа (даже тот, кто зашифровал сообщение, не может произвести обратную операцию).

Замечание

Сообщения, теги и пр. подписывают для того чтобы подтвердить, что сообщение написано именно вами и не изменялось в процессе передачи. Если сообщение будет изменено, то при проверке подписи это будет указано

Чтобы создать ключ, следует запустить утилиту командной строки gpg[12] с аргументом `--full-generate-key` (допустимо и с аргументом `--gen-key`, но в этом случае не будет возможности выбрать несколько важных параметров[13])

```
$ gpg --full-generate-key --expert

pg (GnuPG/MacGPG2) 2.2.17; Copyright (C) 2019 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Please select what kind of key you want:
(1) RSA and RSA (default)
(2) DSA and Elgamal
(3) DSA (sign only)
(4) RSA (sign only)
(7) DSA (set your own capabilities)
(8) RSA (set your own capabilities)
(9) ECC and ECC
...
```

Замечание

Важно иметь в виду, что выбрав вариант (3) DSA (sign only) или (4) RSA (sign only) нельзя будет шифровать сообщения и файлы

Для RSA ключа размером 2048 бит вполне достаточно, но можно выбрать и размер до 4096 бит (а вот использовать ключи размера меньще 2048 бит небезопасно).

[12] Для операционной системы `MacOS X` https://gpgtools.org/
[13] Выбор расширяется, елси добавить параметр `--expert`

Если выбрать ограниченный срок действия ключа, то по истечению его срока ключ будет признан недействительным[14].

В завершении генерируется ключ и добавляется в связку ключей. В связке ключей может находится множество ключей. Также на этом этапе создается *сертификат отзыва* – файл, с помощью которого созданный ключ можно отозвать (признать недействительным). Рекомендуется хранить его в безопасном месте.

Комметарии к сообщению `gpg`:

- `rsa` – алгоритм шифрования RSA,
- `2048` – длина ключа,
- `1970-01-01` – дата создания ключа,
- `2BB680...E426AC` – отпечаток ключа. Его следует сверять при импортировании чужого публичного ключа – у обоих сторон он должен быть одинаков,
- `uid` – идентификатор (user-id),
- `pub` – публичный ключ,
- `sub` – публичный подключ,
- `sec` – секретный ключ,
- `ssb` – секретный подключ.
- `S` – подпись (signing),
- `C` – подпись ключа (certification),
- `E` – шифрование (encryption),
- `A` – авторизация (authentication).

Файл конфигурации храниться по адресу `~./gnupg/gpg.conf`. Пример файла

gpg.conf

```
keyid-format 0xlong
throw-keyids
no-emit-version
no-comments
```

Здесь `keyid-format 0xlong` – формат вывода идентификатора ключа. У каждого ключа и подключа есть свой идентификатор. По умолчанию он не выводится. Допустимые значения:

- `none` (не выводить),
- `short` (короткая запись),
- `0xshort` (короткая запись с префиксом «0x»),
- `long` (короткая запись с префиксом «0x»),
- `0xlong` (длинная запись с префиксом «0x»).

Далее `throw-keyids` – не включать информацию о ключе в зашифрованное сообщение. Эта опция может быть полезна для анонимизации получателя сообщения.

`no-emit-version` – не вставлять версию GPG в зашифрованное сообщение.

`no-comments` – убирает все комментарии из зашифрованного сообщения.

В файле конфигурации эти опции записываются без префикса `--`.

Команды и опции:

[14] Можно продлить срок действия ключа, пока он не истечет

- `--armor / -a`: созадет ASCII (символьный) вывод. При шифровании GPG по умолчанию создает бинарный вывод. При использовании этой опции GPG кодирует информацию кодировкой Radix-64,
- `--encrypt / -e`: зашифровать сообщение,
- `--recipient / -r`: указать ключ, который будет использоваться для шифрования. Можно использовать информацию о пользователе (имя, почта), идентификатор ключа, отпечаток ключа,
- `--decrypt / -d`: расшифровать сообщение,
- `--sign / -s`: подписать сообщение. Подпись при этом будет располагаться отдельно самого сообщения,
- `--clear-sign / --clearsign`: подписать сообщение. Подпись при этом сохраняется вместе с сообщением,
- `--local-user / -u`: указать ключ, который будет использоваться для подписи. Схож с опцией `--recipient`, но это не одно и то же,
- `--verify`: проверить подпись,
- `--list-keys / -k`: вывести список публичных ключей,
- `--list-secret-keys / -K`: вывести список приватных ключей,
- `--export`: экспортировать публичный ключ в файл, который потом можно куда-нибудь отправить,
- `--import`: импортировать публичный ключ,
- `--edit-key`: редактировать ключ,
- `--expert`: режим «эксперта».

Примеры использования:

Зашфровать файл `decrypted.txt` в файл `encrypted.gpg` ключом `0x12345678`. При этом итоговый файл будет текстовым, а не бинарным

```
gpg -a -r 0x12345678 -e decrypted.txt > encrypted.gpg
```

Расшифровать файл `encrypted.gpg` кючом `0x12345678` и сохранить его в файл `decrypted.txt`

```
gpg -r 0x12345678 -d encrypted.gpg > decrypted.txt
```

Подписать файл `message.txt` ключом `0x12345678` и сохранить подпись в файл `sign.asc`

```
gpg -u 0x12345678 -s message.txt > sign.asc
```

Подписать файл `message.txt` ключом `0x12345678` и записать сообщение с подписью в файл `message.gpg`

```
gpg -r 0x12345678 --clearsign message.txt > message.gpg
```

Проверить подпись файла `message.txt`, которая записана в файле `message.asc`

```
gpg --verify message.asc message.txt
```

Импортировать публичный ключ из файла `pubkey.gpg`

```
gpg --import pubkey.gpg
```

Чтобы заставить `Git` использовать закрытый ключ, следует установить значение конфигурационного параметра `user.signingkey`

```
git config --global user.signingkey 0A46826A
```

Теперь `Git` будет использовать этот ключ по умолчанию для подписи тегов и коммитов.

Вывести информацию по GPG-ключам

```
gpg --list-keys
# выведет
/c/Users/ADM/.gnupg/pubring.kbx
------------------------------
pub   rsa2048 2020-06-06 [SC] [expires: 2022-06-06]
6A2B86D3D3BD58B4661982FAEEDC6C3662F65FD8
uid           [ultimate] Leor Finkelberg <leor.finkelberg@yandex.ru>
sub   rsa2048 2020-06-06 [E] [expires: 2022-06-06]
```

Заставить Git использовать для подписи закрытый ключ можно так

```
git config --global user.signingkey 6A2B86D3D3BD58B4661982FAEEDC6C3662F65FD8
```

Вывести информацию по параметру `signingkey`

```
git config --global --list | grep sign
```

9.29.2. Подписи коммитов

Чтобы подписать отдельный коммит, следует

```
git commit -a -S -m 'signed commit'
```

Для просмотра и проверки таких подписей команда `git log` снабжена параметром `--show-signature`

```
git log --show-signature -1
```

С помощью параметра `--verify-signatures` можно заставить проверять слияния и отклонять их, если коммит не содержит доверенной GPG-подписи.

Если воспользоваться этим параметром при слиянии с веткой, содержащей неподписанные и недействительные коммиты, слияние выполнено не будет

```
git merge --verify-signatures non_verify_branch
```

Если же ветка, с которой осуществляется слияние, содержит только корректно подписанные коммиты, то команда `merge` сначала покажет все проверенные ею подписи, а потом перейдет непосредственно к слиянию

```
git merge --verify-signatures signed_branch
```

Можно также воспользоваться параметром `-S` команды `git merge` для подписи коммита, *образующегося в результате слияния*

```
git merge --verify-signatures -S signed_branch
```

Итоговый коммит слияния получит подпись.

9.30. Поиск

По умолчанию команда `git grep` выполняет поиск среди всех файлов вашей рабочей директории. Параметр `-n` указывает номера строк, в которых была найдена заданная подстрока

```
git grep -n 'section' *.tex
```

Вывести список файлов, в которых встречаются строки, удовлетворяющие поисковому шаблону, и дополнительно для каждого файла указать число совпадений

```
git grep --count 'section' *.tex
```

Вывести список файлов текущей директории (и ее поддиректорий) с указанием строк, удовлетворяющих поисковому шаблону; сводки по каждому файлу разделяютс пустой строкой и отображается заголовок файла

```
git grep --break --heading 'section'
```

Чтобы увидеть историю изменения функции или строки кода в кодовой базе следует исопльзовать конструкцию

```
git log -L :somthing_string:cheat_sheet_git.tex  # какая-то строка
git log -L :E_compute:sou.py  # имя функции
```

9.31. Поиск в Git-журнале

Предположим, что нам нужно определить, когда появилась константа `ZLIB_BUF_MAX`. При помощи параметра `-S` мы попросим Git показать нам только те коммиты, в которых эта строка добавлялась или удалялась

```
git log -SZLIB_BUF_MAX --oneline  # константа ZLIB_BUF_MAX
git log -SE_compute --oneline  # имя функции E_compute
```

9.32. Перезапись истории

Во время работы с `Git` периодически возникает необходимость внести исправления в историю коммитов. Система `Git` примечательна тем, что позволяет вносить изменения в самый последний коммит. Можно скрыть наработки, работу над которыми пока не хотите продолжать или можно внести изменения в сделанные коммиты, придав истории совсем другой вид. И все это делается до выкладывания ваших наработок в общий доступ.

9.32.1. Редактирование последнего коммита

Отредактировать сообщение фиксации последнего коммита очень просто. Эта команда берет область индексироавния и включает в коммит всю обнаруженную там информацию

```
git commit --amend
```

Эту технику нужно применять с осторожностью, так как она меняет контрольную сумму SHA-1 коммита. Как и в случае с небольшим перемещением, нельзя править последний коммит, если вы уже отправили его в общий доступ.

9.32.2. Редактирование нескольких сообщений фиксации

Чтобы отредактировать сообщения последних трех коммитов или сообщения только для некоторых коммитов из этой группы, в качестве аргумента команде `git rebase -i` передается родитель последнего коммита, который вы собираетесь менять, т.е.

```
git rebase -i HEAD~3
```

Эта команда служит для перемещения, то есть будут переписаны все коммиты в диапазоне `HEAD~3..HEAD` вне зависимости от того, меняете вы для них сообщения или нет.

Ни в коем случае не включайте в этот набор коммиты, уже отправленные на центральный сервер, – сделав так, вы запутаете других разработчиков, предоставив им альтернативную версию уже имеющихся изменений.

Важно запомнить, что коммиты перечисляются в обратном порядке. Самый старый коммит отображается сверху, так как именно он будет воспроизводится первым.

Нужно отредактировать сценарий таким образом, чтобы на коммитах, в которые вы хотите внести изменения, он останавливался. Для этого замените слово `pick` на `edit`. Например, для редактирования сообщения фиксации только в третьем коммите в файл следует внести вот такие изменения:

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Когда вы сохраните этот файл и закроете редактор, `Git` перебросит вас к последнему коммиту в списке и откроет для вас командную строку со следующим сообщением

```
git rebase -i HEAD~3
Stopped at 7482e0d... updated the gemspec to hopefully work better
You can amend the commit now, with
git commit --amend
Once you're satisfied with your changes, run
git rebase --continue
```

Введите

```
git commit --amend
```

Измените сообщение фиксации и закройте редактор. Затем запустите команду

```
git rebase --continue
```

Данная команда автоматически применяет остальные два коммита и на этом заканчивает работу.

9.32.3. Изменение порядка следования коммитов

Перемещение в интерактивном режиме (`git rebase -i`) может также использоваться для изменения порядка следования коммитов или их удаления. К примеру, чтобы удалить коммит, связанный с добавлением файла `cat-file`, и изменить порядок следования двух оставшихся коммитов, нужно изменить сценарий перемещения. Вот исходный вариант

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

А вот вариант сценария

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

После сохранения новой версии сценария и выхода из редактора система `Git` перемотает ветку до предка этих коммитов, применит коммиты `310154e` и `f7f3f6d`, после чего остановится.

9.32.4. Объединение коммитов

Инструмент интерактивного перемещения позволяет также превратить несколько коммитов в один коммит.

Как обычно работа начинается с команды `git rebase -i` (предварительно нужно определить номер родителя базового коммита, т.е. коммита, на который будет указывать `HEAD~`, но этот коммит не будет отображаться в сценарии перемещения)

```
git rebase -i HEAD~родитель_базового_коммита
```

Если вместо `pick` или `edit` указать `squash`, `Git` применит указанное изменение и непосредственно предшествующее ему изменение и заставит вас объединить сообщения фиксации. После сохранения результатов редактирования появится единственный коммит.

Важно помнить, что коммиты в сценарии перемещения отображаются в обратном порядке и что `squash` воздействует на *предыдущий* коммит.

9.32.5. Разбиение коммита

Процедура разбиения коммита отменяет внесенные им изменения, затем индексирует его по частям и фиксирует столько раз, сколько коммитов вы в итоге хотите получить. Предположим вы хотите разбить средний коммит на две части. Вместо коммита «update README formatting and added blame» вы хотите сделать так, чтобы первый коммит обновлял и форматировал файл `README`, а второй добавлял файл `blame`. Для этого в сценарии `rebase -i` нужно заменить инструкцию для разбиваемого коммита на `edit`

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

После того как сценарий вернет вас в командную строку, вы отмените действие коммита и создадите из этих отмененных изменений нужное количество новых коммитов. Как только вы сохраните сценарий и выйдете из редактора, `Git` перейдет к родителю первого коммита из списка, применит превый коммит (`f7f3f6d`), затем второй (`310154e`) и вернет вас в консоль. Здесь изменения, внесенные этим коммитом, можно отменить командой `git reset HEAD^`, которая эффективно возвращает все в предшествующее состояние, причем модифицированные файлы оказываются *неиндексированными*. После этого можно начинать индексацию (`git add`) и фиксацию (`git commit`) файлов, пока у вас не появится *несколько коммитов*. Затем останется выполить команду `git rebase --continue`

```
git rebase -i HEAD~3
git reset HEAD~
git add README
git commit -m 'updated README formatting'
git add lib/simplegit.rb
git git commit -m 'added blame'
git rebase --continue
```

Git применит последний коммит (a5f4a0d) из этого сценария, и история приобрете такой вид

```
git log -4 --pretty=format:'%h %s'
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

Замечание

Эта процедура меняет контрольные суммы SHA всех коммитов в списке, поэтому важно следить за тем, чтобы список содержал только коммиты, которые еще не отправлялись в общее хранилище

9.32.6. Переписывание истории с помощью filter-branch

Существует еще один способ переписывания истории, к которому прибегают, когда при помощи сценария нужно внести изменения в большое количество коммитов, например, везде поменять ваш адрес электронной почты или убрать какой-то файл из всех коммитов. В таких случаях на помощь приходит команда filter-branch, позволяющая переписывать большие фрагменты истории.

Удаление файла из всех коммитов Часто случается так, что пользователь, необдуманно выполнив команду git add, включил в коммиты огромный бинарный файл и его требуется отовсюду удалить. Или вы можете сами включить в коммит файл, содержащий пароль, а потом решить сделать проект открытым.

Вот так выглядит удаление файла passwords.txt из истории проекта

```
git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
```

Параметр --tree-filter заставляет выполнить указанную команду после перехода к каждой следующей версии проекта, а затем повторно фиксирует результаты. В этом случае вы удаляете файл passwords.txt из *каждого снимка состояния системы* вне зависимости от того, существует он или нет. Еще можно использовать --index-filter, --msg-filter, --commit-filter, --tag-name-filter и пр.

После выполнения этой команды может потребоваться «перезагрузить» историю коммитов с помощью интерактивного перемещения, например так

```
git rebase -i HEAD~3
```

Сохранить и выйти. После этого можно посмотреть, что стало с историей коммитов git log -oneline.

Для удаления всех случайно зафиксированных *резервных копий* файла, созданных текстовым редактором, можно написать[15]

```
git filter-branch --tree-filter 'rm -f *~' HEAD
```

Имена файлов при *одноуровневом резервном копировании* формируются за счет добавления к исходному имени файла знака тильды `~`. При *многоуровневом* создании резервных копий к имени файла добавляется сочетание символов `~n~`, где `n` – это номер следующей резервной копии, начинающийся с единицы [2, стр. 230].

В общем случае применение команды `git filter-branch` рекомендуется применять в тестовой ветке, а затем, если выяснится, что именно такой результат вам и нужен, выполнить полную перезагрузку ветки `master`.

Чтобы команда работала со всеми вашими ветками, добавьте к ней `--all`.

9.32.7. Изменение адресов электронной почты в глобальном масштабе

Также часто возникает ситуация, когда пользователь перед началом работы забывает воспользоваться командой `git config` и указать свой адрес электронной почты.

Изменить адреса электоронной почты можно так

```
git filter-branch --commit-filter '
    if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
    then
         GIT_AUTHOR_NAME = "Scott Chacon";
         GIT_AUTHOR_EMAIL = "schacon@example.com";
         git commit-tree "$@";
    else
        git commit-tree "$@";
    fi ' HEAD
```

Эта команда по очереди переписывает все коммиты, вставляя туда ваш новый адрес электронной почты. Так как коммиты содержат значения SHA-1 своих предков, эта команда поменяет значения SHA *всех* коммитов в истории, а не только тех, в которых был обнаружен указанный вами электронный адрес.

9.33. Спецификация ссылок

Рассмотрим более сложный случай проецирования удаленных веток на локальные

```
git remote add origin https://github.com/LeorFinkelberg/Cheat_sheet_Git.git
```

Эта команда добавляет в файл `.git/config` раздел, задающий имя удаленного репозитория (`origin`), его URL-адрес и *спецификацию ссылок* для получения оттуда данных

```
[remote "origin"]
    url = https://github.com/LeorFinkelberg/Cheat_sheet_Git.git
    fetch = +refs/heads/*:refs/remotes/origin/*
```

Эта спецификация имеет следующий формат: по желанию символ `+`, за которым следует `<источник>:<цель>`, где `<источник>` – шаблон для ссылок на сервере, а `<цель>` – место, куда эти

[15] Эта команда удаляет из истории проекта все файлы, заканчивающиеся на ~

ссылки будут записываться локально. Символ + заставляет Git обновлять ссылки, даже если речь идет не о перемотке.

К слову, значение переменной fetch можно узнать следующим образом

```
git config remote.origin.fetch
```

В случае настроек по умолчанию, которые автоматически используются командой git remote add, Git извлекает все ссылки из папки refs/heads/ *на сервере* и записывает их в *локальную папку* refs/remotes/origin/.

Если на сервере есть ветка master, локальный доступ к ее журналу можно получить так

```
git log origin/master
git log remotes/origin/master
git log refs/remotes/origin/master
```

Все эти команды дают один и тот же результат, так как Git разворачивает каждую из них до состояния refs/remotes/origin/master.

Если нужно, чтобы с удаленного сервера при каждом обращении к нему извлекалось содержимое только ветки master, а не всех подряд, измените соответствующую строку в конфигурационном файле

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

Посмотреть настройки конфигурационного файла на уровне репозитория, связанные с извлечением данных, можно так

```
git config --local --list | grep 'fetch'
```

Эта *спецификация ссылки*, предлагаемой по умолчанию, будет использоваться командой git fetch при обращении к данному удаленному репозиторию.

Если же вы хотите выполнить некое однократное действие, указать спецификацию можно в командной строке. Чтобы извлечь данные из *удаленной* ветки master и поместить их в *локальную* ветку origin/mymaster, напишите

```
git fetch origin master:refs/remotes/origin/mymaster
```

9.34. Информация о файлах

Вывести информацию о файлах и директориях, попавших в заданное дерево объекта, с указанием режима доступа, типа и контрольной суммы

```
git ls-tree HEAD
# выведет
100644 blob 9c8ab46e95aacdaed92b2dbcf6bf69965e115dd1    .gitignore
100644 blob 93ffc9c04dd7b32cdfb14b0db7af3fcf165917a2    README.md
100644 blob 411b170c34aeed793ae7984ccd2f657802a43300    cheat_sheet_git.pdf
100644 blob 9ffd86fa5acc6be4beed24f7b2c0508c1e222edf    cheat_sheet_git.tex
040000 tree 417bb89bd6f65dfa12a8a9d35343bf8c6c223ad0    style_packages
```

Вывести размер файла по его контрольной сумме

```
git cat-file -s 411b170c34aeed793ae7984ccd2f657802a43300 # 737387
```

9.35. Перемещения

Перестроить текущую ветку `dev` на ветке `origin/dev` в интерактивном режиме

```
(dev)$ git rebase -i origin/dev
```

10. Коллекция сценариев

10.1. Откат к состоянию до слияния

Была ветка `fix`, в которой исправляли баг. Исправили, слили изменения из `fix` в `master`, но тут выяснилось, что это исправление ломает приложение. Нужно откатить ветку `master` к состоянию до слияния

```
# находимся в ветке fix, баг "исправлен"
git checkout master
git merge fix
# видим проблему, приложение сломалось
git checkout fix
git branch -f master ORIG_HEAD
```

10.2. Замена одного файла другим без слияния

Если требуется просто заменить один файл другим без слияния (например, один pdf-файл другим), то можно воспользоваться такой констукцией

```
git checkout --ours file_name.pdf   # оставить файл без изменений
git checkout --theirs file_name.pdf # заменить файлом из сливаемой ветки
```

Список литературы

1. *Чакон С.*, *Штрауб Б.* Git для профессионального программиста. – СПб.: Питер, 2020. – 496 с.
2. *Собель М.* Linux. Администрирование и системное программирование. 2-е изд. – СПб.: Питер, 2011. – 880 с.