0. О документе

Данный документ является результатом просмотра мной скринкаста по системе контроля версий Git.
Исходный плейлист находится здесь:
https://www.youtube.com/playlist?list=PLDyvV36pndZHkDRik6kKF6gSb0N0W995h

Документ описывает всё то, что автор усвоил во время просмотра вышеуказанного скринкаста.
Можно рассматривать данный документ в качестве пособия по Git, но желательно, хотя и необязательно, посмотреть исходный скринкаст.

Используется старая версия скринкаста, т.к. из нового курса многое полезное было убрано.
Таким образом, данный документ является попыткой воспроизвести материалы скринкаста как можно точнее и объёмнее.

Дополнением к данному гайду служит короткая справка по Vim, она также распространяется под данной лицензией.

Мои контакты:
GitHub - https://github.com/maxt86
ВК - https://vk.com/maxt86
Почта - maxt86@pm.me

*** ОГЛАВЛЕНИЕ ***

---

   4) Зачем нужен индекс?
   5) Коммиты без git add
   6) Удаление и переименование файлов

4. Ветки
   1) Введение
   2) Создание и переключение
   3) Команда checkout при незакоммиченных изменениях
   4) Передвижение веток "вручную"
   5) Состояние отделённой HEAD
   6) Восстановление предыдущих версий файлов
   7) Просмотр истории и старых версий, ~ и :/
   8) Слияние перемоткой
   9) Удаление веток
  10) Лог ссылок: reflog
  11) Сборка мусора

5. Теги
   1) Теги, основные действия с тегами
   2) Использование тегов для экспорта с describe, archive

6. Reset
   1) Жёсткий reset, отмена коммита
   2) Мягкий reset, замена и объединение коммитов
   3) Правка последнего коммита: commit --amend
   4) Смешаный reset, отмена индексации
   5) Жёсткий reset с сохранением изменений: --keep
   6) Особый reset для отмены слияний: --merge
   7) Виды reset – таблица в документации

7. Очистка проекта от изменений

8. Просмотр
   1) Сравнение коммитов, веток и не только: git diff
   2) Сравнение по словам, драйвер diff
   3) Сравнение разных форматов, кастомизация diff
   4) Вывод истории: git log, форматирование коммитов
   5) Диапазоны коммитов для git log и не только
   6) Вывод git log коммитов, меняющих нужный файл
   7) Поиск в истории, фильтры для git log
   8) Кто написал эту строку? git blame

9. Слияние
   1) Истинное слияние и разрешение конфликтов в git merge
   2) Коммит слияния, дальнейшие слияния
   3) Отмена слияния
   4) Семантические конфликты и их разрешение
   5) Слияние с сохранением веток, запрет перемотки --no-ff
   6) Слияние без связи с источником: merge --squash
   7) Слияние без конфликтов через драйвер union, свои драйверы
   8) Стратегии слияния

10. Копирование коммитов
   1) Копирование коммитов: cherry-pick

---

1. Введение

1) Что такое Git?

Git - система контроля версий, позволяющая их отслеживать и вести совместную разработку.
* Особенность Git в том, что это распределённая система контроля версий, т.е. центральный сервер для связи необязателен.

2) Установка Git под Linux и Mac

OS X:
```
brew install git
  или
port install git
```

3) Настройка профиля под Unix-системами

```
cat .profile

export LANGUAGE=en_US

git-completion.bash
git-prompt.sh

GIT_PS1_SHOWDIRTYSTATE=true
```

```
GIT_PS1_SHOWUNTRACKEDFILES=true

PS1="\n\w$(__git_ps1)> "
```

Поиск интеграционных скриптов в поисковике:
  git prompt <Имя оболочки>

https://github.com/magicmonty/bash-git-prompt

4) Установка Git под Windows

Git Bash

Git LFS (Large File Support)

core.autocrlf=[true|input|false]

Git Credential Manager

5) Windows: Git Bash, PowerShell

posh-git

git init

.git

2. Конфигурация

1) Основы

git init

.git

```
git config --global user.name "John Doe"
git config --global user.email john.doe@example.com
```

cat .git/config

```
git config diff.png.textconv ...
  ->
[diff "png"]
      textconv = ...
```

XDG
$XDG_CONFIG_HOME ($HOME/.config)

vvv

```
--system
  /etc/gitconfig
  C:\Program Files\Git\etc\gitconfig
    C:\ProgramData\Git\config - настройки, выбранные при установке Git
^^^

--global
  ~/.gitconfig
  C:\Users\<Имя пользователя>\.gitconfig
    XOR
  $XDG_CONFIG_HOME/git/config
    ~/.config/git/config
^^^

--local (default)
  .git/config
^^^

git config --unset user.name
git config --remove-section user

git config --list

Помощь:
  git config -h
  git help config

git config --global core.pager 'less -RFX'

less:
  / - поиск по регулярному выражению
    (Shift) n - поиск вперёд (назад)
  q - выход


2) Настройка редактора

git config --global --edit (-e)

git config --global core.editor <Редактор>

GIT_EDITOR
EDITOR
VISUAL


3) Алиасы (псевдонимы для команд)

git config --global aliac.c 'config --global'
git config alias.sayhi '!echo "hello"; echo "from git"' ('!git ...; git
...')


4) Проблема с переводами строк Windows/Linux
```

0a (LF: "Line Feed") - Unix
0d 0a (CR: "Carriage Return", LF) - Windows

5) Настройка core.autocrlf для нормализации переводов строк

true - Windows
CRLF -> LF при записи
LF -> CRLF при чтении

input - Unix - односторонняя конвертация
CRLF -> LF при записи

false (по умолчанию)
Если у всех разработчиков одна ОС

convert.c - convert_is_binary - применяется для преобразования переводов строк
xdiff-interface.c - buffer_is_binary

Для файлов можно поставить текстовый или бинарный режим с помощью атрибутов.

6) Атрибуты: .gitattributes, text, eol

.gitattributes позволяет явно сказать, к каким файлам применяются те или иные настройки.

Снятый атрибут - не то же самое, что неуказанный.

Настройки ниже перекрывают те, что выше.

```
# Комментарий
* text=auto # Смотрит на core.autocrlf, потом core.eol, потом на платформу.
*.html text
*.sh eol=lf # Просто говорит, что при чтении файла ничего заменять не надо.
*.sln text eol=crlf
*.png -text
```

binary - псевдоним для "-text -merge -diff"

Денормализация - преобразование переводов строк к родным для данной платформы (Используется при чтении данных при "* text=auto")

Атрибуты для a/b/c.txt:
  <Проект>/.git/info/attributes

  <Проект>/a/b/.gitattributes
  <Проект>/a/.gitattributes
  <Проект>/.gitattributes - лидер

```
  config: core.attributesFile
  (git config --global core.attributesFile ~/.gitattributes)
    (default) $XDG_CONFIG_HOME/git/attributes
      (default) ~/.config/git/attributes

  /etc/gitattributes
  C:\Program Files\Git\etc\gitattributes

git help attributes


7) Игнорирование: .gitignore

# anywhere/Thumbs.db
Thumbs.db

# anywhere/.DS_Store
.DS_Store

# anywhere/my.log, NOT my.logs
*.log

# *.pyc OR *.pyo, NOT *.pyco
*.py[co]

# anywhere/migrate-2077, migrate-2079111.db, NOT migrate-2070...
migrate-207[7-9]*

# anywhere/*.py<any char>, NOT *.py
*.py?

# directory 'build', NOT file 'build'
# the directory itself can be anywhere, not just in the root folder
build/

# Path-Aware Mode

# build/, NOT scripts/build/
/build/

# secret/key, NOT docs/secret/key
secret/key

# doc/file.html, NOT somewhere/doc/file.html
doc/*.html

# my.txt, NOT somewhere/my.txt
/*.txt

# var/www/tmp
# var/www-home/tmp
# NOT var/www/info/tmp
var/www*/tmp
```

```
# users/john/private
# users/alice/private
# NOT users/private
# NOT users/john/project/private
# NOT script/users/john/private
users/*/private

# shop/app/cache
# main/front/app/cache
**/app/cache

# docs/module/generated.html
# docs/general/info/performance.html
# NOT subdir/docs/my.html
docs/**/*.html

# ** can only be used as '**/...', '.../**/...' or '.../**'

# ! - unignore
.*
!.gitattributes
!.gitignore

# DOESN'T WORK
# install/ ignores directory, making its contents unknown to Git
# so we can't 'unignore' anything inside it
/install/
!install/packages.xml

# INSTEAD
# ignore not install/, but each item of its contents
# this way we can 'unignore' it
/install/*
!install/packages.xml

git check-ignore -v install/something

Файлы с игнор-шаблонами:
  <Проект>/.../.gitignore

  <Проект>/.git/info/exclude

  config core.excludesFile
  (git config --global core.excludesFile ~/.gitignore)
    (default) $XDG_CONFIG_HOME/git/ignore
      (default) ~/.config/git/ignore


8) Подключение файлов в конфиг: include

git config (--add) include.path ../gitconfig (.git/config ->
project/gitconfig)
```

--add - доп. инклюды не перезапишут, а дополнят предыдущие; используем, когда нужно подключить несколько файлов

Относительный путь отсчитывается от файла конфигурации.

Условный инклюд:
```
  [includeIf "gitdir:~/company/"]
    path = ~/company/gitconfig
```

3. Основы

1) Создание репозитория, первый коммит

```
git init
<Проект>/.git
```

```
Working Directory - [Index - Repository](.git)
Index: Changes
Repository: FULL HISTORY
```

```
git status
```

```
git add index.html
```

git commit - Все комментарии в редакторе будут вырезаны автоматически

Первая строка - не более 50 символов

Пример:
```
  Create welcome page

  * Add feature A
  * Fix feature B
```

```
create mode 100644 index.html:
  100 - файл
  644 - неисполняемый (755 - исполняемый)
```

Если файловая система не поддерживает отдельное право на выполнение, то при содании репозитория Git автоматически устанавливает:
```
  git config core.fileMode false - говорит, что на права файлов смотреть вообще не надо
```

Если нужно сделать файл исполнимым в Windows:
```
  git update-index --chmod=+x index.html
```
Если файла ещё нет в индексе:
```
  git add --chmod=+x <Имя файла>
```

Посмотреть коммит:
```
  git show <Не менее первых 4 символов хеша коммита>
  git show - текущий коммит
```

```
  git show --pretty=fuller
```

У коммита есть автор и коммиттер

```
git commit --author='John Doe <john.doe@example.com>' --date='...'
```

```
GIT_AUTHOR_NAME
GIT_AUTHOR_EMAIL
GIT_AUTHOR_DATE
GIT_COMMITTER_NAME
GIT_COMMITTER_EMAIL
GIT_COMMITTER_DATE
```

2) Добавление файлов и директорий, git status

Git не умеет работать с пустыми директориями.
.gitkeep обходит это ограничение.

```
git add .
```

```
git reset HEAD <Имя файла или директории> - сбрасывает изменения в индексе
git reset HEAD .idea
```

```
git add --force (-f) .idea/project.iml - добавить, несмотря на игнор
```

```
git commit -m 'Fix issue #123'
```

3) Хороший коммит

Коммит должен выполнять одну вещь, т.е. быть атомарным.
Коммит должен быть консистентным, т.е. логически завершённым.

Commit early. Commit often.

Первая строка сообщения коммита - не более 50 символов, в конце точка НЕ СТАВИТСЯ.

Есть второй стиль:
<Компонент>: <Описание> (компонент с маленькой буквы)

Описание коммита не более 72 символов в ширину

Есть третий стиль:
<Характер выполненной работы>(<Компонент>): <Описание> (характер с маленькой буквы)

4) Зачем нужен индекс?

Чтобы коммитить только определённые файлы.

git add -p - добавить не весь файл, а только некоторые изменения в нём

5) Коммиты без git add

```
git commit --all (-a) -m '...'
git commit -am '...'
```

-a не отслеживает файлы, которые игнорируются Git'ом.

-a вносит в индекс изменения для тех файлов, которые уже там есть. Файлы, которых в индексе нет, не учитываются.

Коммит определённых файлов:
  git commit -m 'Ignore log files' <Пути> - действует то же ограничение, что и для -a

```
git config --global alias.commitall '!git add .; git commit'
git config --global alias.commitall '!git add -A; git commit' - добавить ВСЕ изменения, начиная с корня проекта
git commitall -m '...'
```

6) Удаление и переименование файлов

```
git add . - в том числе и факт удаления файла
git commit -m Cleanup
```

```
git rm <Пути>
git rm -r <Директория>
```

git rm -r src = rm -r src + git add src

git rm -r --cached src - удалит из индекса, но оставит в рабочем каталоге
--cached - операция только с индексом

Если файл модифицирован, то git rm не сработает, т.к. если удалить файл, то несохранённые изменения просто пропадут.

git rm -f index.html - удалить принудительно

Для Git переименование - это удаление старого файла и создание нового.

git mv index.html hello.html

4. Ветки

1) Введение

Ветка - изолированный поток разработки.

Главная ветвь называется 'master'.

Ветки являются тематическимим, т.е. каждая ветка отвечает за свой функционал (feature, another-feature)

Ветки также могут быть версионными (другое название - релизные ветки) (v1.0, v2.0)

Cherry picking - это когда изменения любого коммита применяются к любой ветке.

Есть известное сочетание двух подходов, которое называется git flow.

2) Создание и переключение

Ветка - это специальная ссылка на коммит.

<Проект>/.git/refs/heads/master - содержит хеш коммита

git branch - просмотр ветки
git branch -v

<Проект>/.git/refs/HEAD - хранит ссылку на текущую ветку (может и на коммит, но такое встречается редко)

Коммит, на который указывает ветка, называют её вершиной.

Создание ветки:
  git branch <Имя ветки>

Переключение на ветку:
  git checkout <Имя ветки>

Создание и переключение на ветку:
  git checkout -b <Имя ветки> = git branch + git checkout

3) Команда checkout при незакоммиченных изменениях

git checkout не может переключать на ветку без сохранения изменений (коммит или стэш).
Чекаут ругается только в том случае, если изменения находятся в файле, который различается между ветками. Если файл в обоих ветках одинаковый - чекаут работает несмотря на изменения файла в ветке:

```
~/project fix +1> git checkout master
M       script.js
Switched to branch master

~/project master +1>
```

Есть обходной путь, чтобы переключить ветку без сохранения изменений:
  git checkout --force (-f) <Имя ветки>
  git checkout -f HEAD - отмена всех незакомиченных изменений

git stash - сохранение изменений без коммита
git stash pop - вернуть изменения в ветку

git stash не учитывает ветку, на которой был вызван, поэтому технически её можно вызвать на одной ветке, а git stash pop - на другой. При этом могут возникнуть конфликты наложения изменений.

В реальных проектах не всегда очевидно, что для фичи нужна новая ветка. Мы можем работать на одной ветке, а потом окажется, что работы нужно гораздо больше, и не мешало бы создать новую ветку для нового функционала. В таком случае нам поможет команда git checkout -b, т.к. все незакомиченные изменения в старой ветке попадут в новую ветку.

Это работает только потому, что изменения незакоммиченные. А что, если мы уже сделали несколько коммитов не там по ошибке? (см. следующую тему)

4) Передвижение веток "вручную"

git branch fix

После создания ветки надо перенести старую ветку:
  git branch master 54a4 - создать ветку, указывающую на коммит
  git branch --force (-f) master 54a4 - создание или перенос ветки на коммит, если она существует

Прежде чем менять ветку, надо с неё уйти.

Если хотим отменить перенос:
  git branch -f master fix

git checkout -B master 54a4 - создание или перенос ветки на коммит, если она существует, а потом переключение на неё

5) Состояние отделённой HEAD

git checkout <Хеш коммита> - переключение на любой коммит
При этом возникает состояние отделённой HEAD ('detached HEAD' state)

В <Проект>/.git/HEAD находится не ветка, а ссылка на данный коммит.

Git со временем удаляет недостижимые коммиты.

git cherry-pick <Хеш(и) коммита> - копирует коммиты на текущую ветку

6) Восстановление предыдущих версий файлов

git checkout <Коммит> index.html - достаёт версию файла и добавляет его в индекс в текущей ветке

Убрать файл из индекса:

```
  git reset index.html
```

git checkout HEAD index.html - восстановить последние изменения в файле из репозитория в индексе и рабочей директории
git checkout index.html - то же самое, только из индекса в рабочей директории

Предположим, есть файл master. Как быть с git checkout? Очень просто:
git checkout -- master

7) Просмотр истории и старых версий, ~ и :/

git log - выводит лог от HEAD
git log --oneline

git log <Идентификатор ветки или хеш коммита> - если нужно вывести лог не от HEAD

git show ... - посмотреть конкретный коммит
git show HEAD~ - показать родительский коммит для вершины HEAD
git show HEAD~~ - родитель родителя
git show --quiet - информация без изменений

HEAD~~~ <=> HEAD~3
HEAD <=> @
@~3 <=> HEAD~~~

Под PowerShell для @ нужны кавычки, т.к. это спец. символ.

Посмотреть не изменения, а старый файл целиком:
```
  git show @~:index.html
  git show fix:index.html
  git show :index.html - просмотр текущей проиндексированной версии файла
```

git show :/sayBye - найти самый свежий коммит со словом 'sayBye' в описании
:/ можно использовать в любой команде Git.

Если надо найти все коммиты с такой фразой в описании, то git log с флагами.

8) Слияние перемоткой

git checkout master
git merge fix - слияние fix с master (Fast-forward, перемотка)

При перемотке указатель master передвигается в сторону fix.

git merge при слиянии записывает идентификатор старой вершины в <Проект>/.git/ORIG_HEAD.

Отмена слияния:

```
  git branch -f master ORIG_HEAD

git checkout -B master fix
```

9) Удаление веток

git branch -d fix - удалит fix, только если она объединена с текущей веткой

Если надо удалить необъединённую с текущей ветку:

```
  git branch -D feature
```

Надо трижды подумать, прежде чем удалять ветку последним способ, ведь недостижимые коммиты Git со временем удаляет.

10) Лог ссылок: reflog

Восстановить ветку, если мы забыли хеш коммита, поможет reference log.

При любой операции с изменением ссылок, изменения записываются <Проект>/.git/logs/<Имя ссылки>.

git reflog - вывести рефлог для HEAD
git reflog <Ссылка> - если нужно не для HEAD

git reflog выводит историю от последних действий к более старым.

git reflog выводит более полную историю, чем git log.

HEAD@{123} - обращение к записям из рефлога

git reflog <=> git reflog show
Но это не единственная возможная опция.

git reflog show <=> git log --oneline -g
Как следствие, множество флагов git log применимо и к git reflog.

git reflog --date=iso

HEAD@{<Дата>} - обращение к записям рефлога по дате

Для обращений по ...@... нужна запись в рефлоге, записи не хранятся бесконечно:

```
  gc.reflogExpire="90 days ago"
  gc.reflogExpireUnreachable="30 days ago"
```

git checkout @{-1} - предыдущая ветка, с которой был чекаут на данную
Данная команда просматривает рефлог для выяснения нужной ветки.
checkout: moving from ... to ...
Мы переключимся на ту ветку, что перед from в рефлоге.

git checkout @{-1} <=> git checkout -

Некоторые команды, но не все, поддерживают второй вариант.

git reflog --no-decorate

11) Сборка мусора

Автоматически запускается при командах слияния и загрузки данных с сервера.

Мы можем вызвать её вручную:
  git gc

Восстановление данных:

  Если коммит не достижим и его даже нет в рефлоге:
    gc.pruneExpire="2 weeks ago" - относится к возрасту коммита

  git fsck --unreachable - список всех недостижимых объектов Git

  Пока жив коммит, живы и его родители.

Удаление данных:

  git filter-branch - заменяет коммиты ветки на новые, полученные из старых с помощью определённого фильтра (в т.ч. удаляющего ненужные файлы и директорией)
  Аналог данной команды - BFG.

  git reflog expire --expire=now --all
  git gc --prune=now - --prune не считается с gc.pruneExpire

5. Теги

1) Теги, основные действия с тегами

git tag v1.0.0 1913

git show --quiet v1.0.0

git log --oneline
git log --oneline v1.1.0

Тег - это просто метка.
Тег, как ветка - только он никуда не перемещается и всегда указывает на один и тот же коммит.
Часто теги используются для маркировки релизов.

git tag - посмотреть список тегов
git tag --contains 54a4 - список релизов, содержащих нужный коммит

git tag -n[<Кол-во строк>, def=1] - вывести сообщение коммита, на который указывает тег.
git tag -n -l 'v1.1.*' - вывести только соответствия маске
git tag -d v1.0.0 v1.1.0 - удалить тег(и)

git help tag

Теги бывают двух видов:
  - лёгкие теги
  - теги с аннотацией

git tag -a -m 'Аннотация' v1.0.0 1913 - создать тег с аннотацией

В большинстве случаев рекомендуется использовать аннотированные теги.

2) Использование тегов для экспорта с describe, archive

git describe - по любому коммиту получить его описание на основе ближайшего тега
(по умолч. используются только аннотированные теги)

==========
> git describe 1913
v1.0.0

> git describe
v1.1.0-2-g2c11f12   < (ближ. тег)-(на сколько коммитов мы впереди)-(g с сокр. идентификатором описываемого коммита)
==========

git archive -o /tmp/v1.1.0-2-g2c11f12.zip HEAD - архивация содержимого репозитория на момент нужного коммита

6. Reset

1) Жёсткий reset, отмена коммита

git reset --hard @~ - жёсткий reset

Жёсткий reset передвигает текущую ветку на указанный коммит, а также обновляет рабочую директорию и индекс.

git reflog master - все предыдущие значения master

cat .git/ORIG_HEAD - reset записывает предыдущее значение HEAD в данный файл - данная ссылка не влияет на сборку мусора и нужна только для удобства

git reset --hard ORIG_HEAD - вернёт всё, как было

Если в файлах были незакомиченные изменения, то при жёстком reset они пропадут! (выход: git reset --keep или git stash)
Но если наша цель - откат последних изменений, то такое поведение скорее в плюс.

Жёсткий reset используется для полной отмены последних коммитов или отката незакомиченных изменений.

2) Мягкий reset, замена и объединение коммитов

git reset --soft @~ - мягкий reset

Мягкий reset переносит текущую ветку на указанный коммит, НО рабочую директорию и индекс не трогает.
Мягкий reset как бы отменяет коммит, при этом оставляя все подготовленные для него данные.
На практике он используется, чтобы переделать неудачные коммиты.

git reset --soft ORIG_HEAD - вернуться обратно

git commit -c ORIG_HEAD - скопировать описание из ORIG_HEAD при новом коммите с внесёнными поправками.
-C - без редактора
При использовании данной опции имя автора и коммитера и дата в сообщении могут отличаться!
git commit --reset-author - сделать автором себя

git commit --amend - внести поправки в коммит (делает в один шаг мягкий reset и коммит с поправками)

git show --quiet --pretty=fuller - вывод Commit и CommitDate

3) Правка последнего коммита: commit --amend

- Правка
- git add index.html
- git commit --amend

git commit --amend - git reset --soft @~ + git commit -c ORIG_HEAD

git commit --amend --reset-author
git commit --amend --no-edit - отменяет вызов редактора

git commit --amend - редактирование описания коммита
git commit --amend -m '...'

(git rebase - переписывание истории)

4) Смешаный reset, отмена индексации

Смешанный reset - reset по умолчанию.

```
git reset --mixed @~
git reset @~
```

Смешанный reset переносит текущую ветку на указанный коммит и обновляет индекс, НО рабочую директорию не трогает.

```
--------------------------
 WD  index *branch - soft
 WD *index *branch - mixed
*WD *index *branch - hard
--------------------------
```

Смешанный reset более удобный, чем мягкий, если мы хотим сформировать индекс заново.
Он также может применяться для очистки индекса.

git reset index.html - сбросить только определённый файл в индексе - эта команда является антиподом git add

git reset 54a4 index.html - обновить файл в индексе в соответствии с указанным коммитом (НО в репозитории файл не меняется!)
Лучше использовать git checkout 54a4 index.html - достаёт файл также и в рабочую директорию.

5) Жёсткий reset с сохранением изменений: --keep

git reset --keep @~ - жёсткий reset с сохранением изменений

Если файл не менялся между коммитами, то файл остаётся в рабочей директории.
Это очень похоже на поведение checkout.

НО reset --keep убирает незакоммиченные изменения из индекса.

git diff --name-only @~ - какие файлы менялись между коммитами

Если файл между коммитами менялся, то reset --keep выдаст ошибку, т.е. это безопасная команда, в отличие от reset --hard
(Если надо и сохранить изменения, и отресетиться, то git stash/git stash pop)

6) Особый reset для отмены слияний: --merge

Нужен для отмены неудачных слияний.

Напоминает reset --keep с тем отличием, что удаляет изменения, которые внесены в индекс.

Как и --keep, основан на жёстком reset.

А если у файла есть и проиндексированные, и непроиндексированные изменения, то он выдаст ошибку.

git reset --merge
Проиндексированные изменения удаляются, а непроиндексированные остаются. Это нужно для разрешения конфликтов слияния.

7) Виды reset – таблица в документации

git help reset - внизу есть таблица сравнений разных reset

7. Очистка проекта от изменений

(
  git checkout -f
  git reset --hard
) - при чистке проекта данные команды никак не влияют на неотслеживаемые Git'ом файлы

git clean -dxf - очистка проекта от хлама
-d - удалить и директории
-x - удалить в т.ч. файлы, которые игнорируются через .gitignore
-f - мы уверены, что мы хотим всё почистить

8. Просмотр

1) Сравнение коммитов, веток и не только: git diff

Данная команда используется такими командами, как git log и git show.

git diff <Коммит1> <Коммит2> - посмотреть различия двух коммитов

Для каждого файла есть отдельный блок.

```
====================================
diff --git a/index.html b/index.html   < a и b - кодовые названия веток
index da9736c..f4952f9 100644          < <контрольная сумма файла в первом источнике..во втором> <тип объекта>(100 - файл, 644 - неисполнимый)
--- a/index.html                       < заголовки diff
+++ b/index.html                       < файлы могут быть переименованы, удалены, созданы и т.д.
@@ -5,14 +5,10 @@                      < если в исх. файле взять 14 строк, начиная с 5, и применить изменения, то в целевом получится 10 строк, начиная с 5 (второе число def=1)
     <script src="script.js"></script>
   </head>
```

```
   <body>
-    <script>
-      sayHi();
-    </script>
-
     Git rules!

     <script>
-      sayBye();
+      work();
     </script>
   </body>
  </html>
=====================================

...
@@ -2,6 +2,10 @@ function sayHi() { < заголовок ханка (начиная с
function) - работает с помощью рег. выражений
...

git diff master feature
git diff master..feature - одно и то же

git diff master...feature - что именно изменилось в feature с момента её
отхождения от master, т.е. сравнивает последний коммит в feature с
коммитом перед разветвлением в master (такой синтаксис порождает поиск
общего предка двух веток)

git diff 2fad - сравнит содержимое рабочей директории с содержимым
репозитория на момент данного коммита
git diff HEAD (или просто git diff, только git diff сравнивает рабочую
директорию с индексом, а не с репозиторием) - обе версии игнорируют
неотслеживаемые файлы

Когда создаётся новый файл:
...
new file mode 100644
index 0000000..fa4de51
--- /dev/null
+++ b/NEWS
...

git diff --cached(--staged) - сравнивать нужно не рабочую директорию, а
индекс

git commit -v - в редакторе сообщением по умолчанию будет вывод diff, но
только если убрать строку-ножницы:
# --- >8 ---

Если нужно по умолчанию вызывать git commit с флагом -v, то:
  git config --global commit.verbose true

git diff index.html
```

git diff . - сравнивает содержимое рабочей директории с индексом, но только для указанных файлов (можно указать текущую директорию точкой)

git diff master feature index.html

git diff --name-only master feature - покажет только имена файлов

git diff -- index.html - рекомендуется вызывать git diff так

```
    1         2
WD <=> Index <=> Repo (HEAD)
^                ^
|                | - 3
+----------------+
```

1. git diff
2. git diff --cached(--staged)
3. git diff HEAD (git diff @)

---

git diff --cached <-> git commit -v (commit.verbose = true)

git diff коммит1:путь1 коммит2:путь2 - сравнение индивидуальных файлов
git diff master:one.html feature:two.html

git diff --no-index путь1 путь2 - сравнивает любые два файла на диске изолированно от Git

2) Сравнение по словам, драйвер diff

git diff --word-diff - сравнение по словам, а не по строкам

```
...
{-that show-}{+can improve+}
...
```

git diff --word-diff=color
git diff --color-words - выделять слова только цветом

Словом Git считает любую послед. символов без пробела.
Чтобы изменить данную установку для, скажем, HTML-файла, то в .gitattributes пишем:

```
*.html diff=html < использовать для HTML-файлов драйвер diff для HTML
```

---

git help attributes - тут есть встроенные в Git драйверы

- Создание драйвера
Идём в .git/config и пишем:

```
[diff "markdown"]                    < markdown - имя драйвера
  xfuncname = "^#+[[:space:]]+.+$"  < ханк
```

```
  wordRegex = "\\*+|[^[:space:]*]+" < рег. выражение, описывающее слово
---------------------------------
Для драйверов используются POSIX extended regular expressions.

Рег. выражение для ленивых:
...
  wordRegex = .
...

Рег. выражения также можно передать через --word-diff-regex.

Встроенные языковые драйверы:
  userdiff.c - PATTERNS(name, pattern, word_regex) и IPATTERN(name,
pattern, word_regex)
  ------------------------------------------------------
  word_regex "![^[:space:]]|[\xc0-\xff][\x80-\xbf]+" } < word_regex +
один произвольный непробельный символ с учётом Unicode
  ------------------------------------------------------ < \n используется
для разделения рег. выражений внутри Git, а ! используется, чтобы
совпадения с последующим рег. выражением исключить


3) Сравнение разных форматов, кастомизация diff

Можно кастомизировать diff для работы с бинарными файлами.

.git/config:
-----------------------------------------
[diff "image"]
  textconv = identify -format '%wx%h %b\\n' < identify - утилита из
пакета ImageMagick
  cachetextconv = true                      < вызовы textconv будут
запоминаться, кэш привязан к тексту команды в textconv выше
  binary = true                             < core.binFileThreshold
-----------------------------------------

---------------
[diff "binary"]
  binary = true
---------------

-----------------------------------
[diff "icdiff"]
  command = sh -c 'icdiff "$1" "$4"' < если хочется использовать свою
команду
-----------------------------------
$1 и $4, т.к. Git передаёт имя файла для diff и три аргумента, связанных
с каждым файлом (можно убедиться, выставив command = echo)
Чаще всего используют git difftool.

git config diff.image.textconv "identify -format '%wx%h %b\n'"
\n обязателен!

Кэш хранится в виде отдельного коммита.
```

В нём находятся результаты вызова textconv.
Ссылка на данный коммит находится тут:
  .git/refs/notes/textconv/image

Чтобы удалить кэш:
  git update-ref -d refs/notes/textconv/image

.gitattributes:
----
*.png diff=image
*.svg diff=binary < есть также встроенный драйвер: *.svg -diff
----------------- < для бинарных файлов есть атрибут binary, это синоним для -text -merge -diff
(
  -text - не преобразовывать концы строк
  -merge - отмена текстового слияния
)

4) Вывод истории: git log, форматирование коммитов

git log - для просмотра коммитов в обратном хронологическом порядке

Логи выводятся в обратном порядке по дате коммита, хоть в них и указывается имя автора, а не коммитера.

git log --pretty=format:'%cd' --date=(short|format:'%F %R')
%F - дата
%R - время - эти обозначения относятся не к Git, а к strftime

Флаги:
--pretty - форматирование вывода (oneline, medium, а также fuller)
--pretty=format:'...' - задание строки форматирования
(
  %h - сокр. ид. коммита
  %cd - дата коммита (CommitDate), %cr (relative) - отн. дата (3 months ago)
  %s - заголовок коммита
  %d - декорирование (т.е. какие ссылки указывают на данный коммит)
  %an - имя автора
  %H - хэш коммита
  %a<буква> - данные по автору < (git commit --date=... --author=...)
  %c<буква> - по коммитеру
  %C(yellow) - жёлтый цвет (применяется к остатку строки) (справка по цветам: git help config)
  %C(#rrggbb)
  %C(reset) - сброс цвета
)
--abbrev-commit - сокращённые идентификаторы коммитов
--oneline - сокращение для --pretty=oneline и --abbrev-commit
--decorate=short - какие ссылки указывают на какие коммиты (HEAD -> master)
--no-decorate - отключить декорирование (см. выше)
--patch(-p) - к каждому коммиту добавляет diff того, что было сделано

git config --global pretty.my format:'...'

git config --global format.pretty my - установит my по умолчанию
git config --global log.date short|relative|format:'...'|format-local:'...'(локальная таймзона)

5) Диапазоны коммитов для git log и не только

Можно задать git log коммиты, которые он будет выводить (def=достижимые из HEAD)

git log <идентификатор(ы)> - коммиты, достижимые из <идентификатора(ов)>

git log --graph - нарисовать структуру коммитов (полезно, если указано несколько веток)

git log --all - коммиты, достижимые из всех ссылок
Когда репозиторий большой, имеет смысл использовать графическую утилиту для отрисовки его структуры.

git log feature ^master - коммиты с момента отхождения feature от master (т.е. все коммиты в feature кроме находящихся также и в master)
git log master..feature (именно 2 точки) - аналогичный синтаксис
git log HEAD..feature => git log ..feature (feature..HEAD => feature..)
--boundary - включить пограничный коммит в вывод, т.е. коммит, который стал началом отхождения одной ветки от другой
git log master...feature - симметрическая разность (т.е. коммиты, которые достижимы из master и feature, но не из обеих веток)

git help revisions - различные способы задания коммитов и диапазонов коммитов

В git diff смысл двух и трёх точек абсолютно другой, чем в остальных командах Git.

6) Вывод git log коммитов, меняющих нужный файл

git log index.html - коммиты, в которых менялся файл index.html
--follow - продолжать следить за файлом, если он был переименован

git log feature..master (--) index.html script.js dir1 dir2

7) Поиск в истории, фильтры для git log

git log --grep Run - все коммиты, в описании которых есть слово Run, с учётом текущей ветки
git log --grep Run feature - то же самое, только из ветки feature

git log --grep Run --grep sayHi

git log --grep Run --grep sayHi --all-match - коммиты и с Run, и с sayHi в описании (после --grep идёт рег. выражение)

Дж. Фридл "Регулярные выражения"

git log --grep 'say(Hi|Bye)' -P - использовать Perl-совместимые рег. выражения
git config --global grep.patternType perl

git log -F - отключить рег. выражения, т.е. считать аргумент после --grep строкой

git log -i - case-insensitive поиск

git log -G<рег. выражение> - поиск по изменениям в коммитах
git log -GsayHi -p index.html - интересен только index.html
git log -G'function sayHi\('

git log -L 3,6:index.html - все коммиты, в которых были изменения с 3 по 6 строку
git log -L '/<head>/','/<\/head>/':index.html - все коммиты, где был изменён HTML-тег <head>

git log -L :sayHi:script.js - посмотреть коммиты с изменениями функции sayHi (любая строка, начинающаяся с буквы, _ или $ - это объявление функции; текстом функции Git считает весь текст до следующей функции) - работает на рег. выражениях

git log -L '/^function sayHi/','/^}/':script.js - рекомендуется использовать такую форму, а не git log -L :sayHi:script.js

git log --author=... - ищет коммиты по автору (в т.ч. его эл. почте)
git log --committer=... - по коммиттеру
git log --before '3 months ago' - поиск по дате (до '3 мес. назад')
--before '2017-09-13'|'2017-09-13 08:30:00 +02' (более подробно о датах в соотв. пункте)
git log --after - аналогично, только не до, а после указ. даты

8) Кто написал эту строку? git blame

git blame <имя файла> - посмотреть, какие строки кем были добавлены
Для каждой строки файла слева находится информация о последнем коммите, который её менял.

Такого разнообразия, как с git log, нет, но некоторые флаги работают:
```
git blame install --date=short -L 5,8
```

Зная коммит, можно его посмотреть:
```
git show 98e7
```

9. Слияние

1) Истинное слияние и разрешение конфликтов в git merge

Истинное слияние - создастся специальный коммит, в котором будут изменения и из feature, и из master.

git status - перед слиянием желательно, чтобы статус был чистый
git diff --name-only master feature - незакомиченных изменений в выведенных файлах быть не должно

git merge - слияние веток, которые не являются предшествующими друг для друга
git merge feature

Алгоритм найдёт общего предка для обеих веток.
git merge-base master feature - вывод общего предка

Алгоритм для каждого файла сравнивает три его версии:
  - в общем предке (base)
  - на текущей ветке (ours)
  - на второй ветке (theirs)

base + (our changes) + (their changes) -> merge

Если в обеих ветках один файл поменяли по-разному в одном и том же месте, то возникает конфликт слияния (merge conflict).
Состояние, в которое мы при этом попадаем, называется прерванным слиянием.

Git запоминает коммит, в котором мы осуществляем слияние, в файле ".git/MERGE_HEAD".

git show 54a4:script.js
git show master:script.js
git show feature:script.js - показать три версии файла

Изменения файла берутся только из той ветки, где они присутствуют.
Например, если в master изменения есть, а в feature - нет, то результирующим файлов будет версия файла из ветки master.
Если изменения есть в обеих ветках, Git *пытается* применить к базовой версии изменения и из одной ветки, и из другой.
Но если в двух ветках изменена одна и та же строка, то возникнет конфликт слияния.

git diff -U0 54a4 master index.html - посмотреть изменения файла в master
-U0 - не показывать строки в контексте, а показывать только изменения (def = 3)

В текстовом редакторе подсвечиваются маркеры конфликта.

Маркеры конфликта:

<<<<<<< HEAD

```
      sayBye();
=======
      sayHi();
>>>>>>> feature
------------------
```

git checkout --ours index.html - вытащит тот файл, что на master
git checkout --theirs index.html - версия из feature
git checkout --merge index.html - версия с маркерами конфликта

git reset --hard - прекратит слияние и очистит все незакомиченные изменения, включая конфликты (в итоге мы перейдём на чистый статус в ветке master)
git reset --merge - сброс до master, оставляя незакомиченные изменения в файлах, которые не участвовали в слиянии, т.е. которые в обеих ветках одинаковы

```
-----------------
WD       Index
-----------------
cpu      cpu
*memory  memory
*disk    disk
*network *network
-----------------
После git reset --merge:
-----------------
cpu      cpu
*memory  memory
*disk    disk
network  network
-----------------
```

git reset --merge полезен для ситуаций, когда слияние происходит не на чистом статусе.

git merge --abort - то же, что и git reset --merge, но его проще запомнить

Если конфликтный файл хочется посмотреть не только с ours- и theirs-версиями, но и с версией base, то:
git checkout --conflict=diff3 --merge index.html

Файл с --conflict=diff3:

```
---------------------------------
<<<<<<< ours
    <script>
      sayBye();
    </script>
||||||| base
    Let's have some fun with git.
=======
      sayHi();
>>>>>>> theirs
```

Если хотим включить данный стиль, чтобы он был всегда:
git config --global merge.conflictStyle diff3

Если мы исправили конфликтный файл, то мы не сможем его закоммитить, т.к. при конфликтах в индексе хранится информация сразу о трёх версиях файла (base, ours, theirs).

git show :1:index.html - 1 - номер стадии (1 - общий предок, 2 - наша ветка (ours), 3 - theirs)

Чтобы закоммитить файл, надо обновить индекс:
git add index.html

git commit
  или
git merge --continue - работает только из состояния слияния - аналогично git commit, сделано для красоты

2) Коммит слияния, дальнейшие слияния

У т.н. коммита слияния 2 родителя.

git log --oneline --all --graph:
==
|\
==

git show:
======================
Merge: 4594f10 2c11f12 < old HEAD, merge HEAD
======================

git diff показывает комбинированный сжатый diff (combined condensed diff):
========================
@@@ -12,7 -8,7 +12,8 @@@
+   < 1 родитель
 +   < 2 родитель
++   < строки не было в обоих родительских коммитах
========================
(с минусами аналогично)

Показываются только те изменения, где возникали конфликты => сжатый (condensed)

git show --first-parent - покажет только первого родителя
git show -m - отличия от всех родителей в обычном формате

git diff HEAD^ (HEAD^1) - первый родитель
git diff HEAD^2 - второй родитель
git diff HEAD^2^ - первый родитель второго родителя

^n - переход к n-ному родителю
~n - переход по первому родителю несколько раз подряд

^n используется только при работе с коммитами слияния.

git show @^2

git branch --merged - показывает ветки, объединённые с текущей
git branch --no-merged

git merge feature --no-edit - слияние веток без вызова редактора
git merge feature --log - добавить лог всех сливаемых коммитов с ветки feature (можно указать число: --log=5 (def=20))

git log master --oneline --first-parent - идти только по первому родителю вниз

git log master --first-parent - полезно, когда merge происходил с --log

3) Отмена слияния

git reset --hard @~

git reflog -4 - 4 последние записи

4) Семантические конфликты и их разрешение

Автоматическое слияние не всегда удобно.

Семантические конфликты возникают, когда нужно использовать либо старый синтаксис, либо новый.

git merge feature --no-edit

git reset --hard @~

git merge feature --no-commit - слияние и остановка непосредственно перед коммитом
Данная команда нужна, чтобы перед коммитом исправлять семантические ошибки, возникшие в процессе слияния.

git add index.html

5) Слияние с сохранением веток, запрет перемотки --no-ff

При слиянии перемоткой непонятно, где заканчивается одна ветка и начинается другая, т.к. после слияния всё становится одной сплошной (линейной) веткой. Линейная структура получившейся ветки не предусматривает места для истории слияния веток.

git commit --no-ff feature - вместо перемотки сделать коммит слияния (no fast-forwarding)

Данная ситуация с нежелательной перемоткой настолько распространена, что есть даже специальный параметр конфигурации:
git config merge.ff false

Также:
git config branch.<имя ветки>.mergeoptions '--no-ff'
Если данный параметр установлен, то мы можем явно включить перемотку:
git merge --ff feature
Флаги в команде переопределяют флаги mergeoptions.

6) Слияние без связи с источником: merge --squash

Когда в ветке были дикие эксперименты, но результат экспериментирования - хороший, то иногда хочется данную ветку объединить с другой, при этом историю ветки включать не нужно.

git merge --squash feature - проделает изменения ветки feature на текущей ветке, чтобы когда мы вызвали git commit, то во вторую ветку добавился коммит со всеми изменениями в первой; коммит при этом обычный, с одним родителем

Данная команда полезна с короткими ветками, а также ветками, историю разработки в которых не хочется выставлять напоказ.

Внимание! При использовании флага --squash .git/MERGE_HEAD НЕ создаётся! Поэтому с такими командами, как git merge --abort и --continue, могут быть проблемы.
Вместо --abort: git reset --merge
Вместо --continue: git commit
git commit может пожаловаться на конфликт (git add index.html после изменений, чтобы git commit не ругался).

7) Слияние без конфликтов через драйвер union, свои драйверы

Драйвер слияния определяет способ слияния файлов.
Драйвер по умолчанию задаётся в .gitattributes:
==========
*.txt merge=text   < драйвер по умолчанию для текстовых файлов
*.png merge=binary < для бинарных файлов
==========

Драйвер union не разъединяет, а автоматически объединяет конфликтующие изменения.
Это удобно для файлов, которые заполняются всеми членами команды, и все изменения актуальны (NEWS.md)
.gitattributes:
==========
/NEWS.md merge=union
==========

union особен тем, что если в одной ветке мы удалим строку из файла, а во второй - поменяем существующую, то при попытке слияния объединённая версия будет включать в себя строки, которые есть хотя бы в одном файле. При слияние сначала идут наши строки, а потом сливаемые:
==========
* The project fails now
* The project builds successfully now < некорректный порядок строк, проблема
==========

Поэтому union используется только для тех файлов, которые только дополняются данными, а также в которых небольшое изменения порядка строк не имеет значения.

Можно поставить свой драйвер слияния, он задаётся в конфиге:
==========
[merge "po"]
  name = gettext merge driver
  driver = git-merge-po.sh %O %A %B < O - путь к файлу из общего предка, A - нашей ветки, B - сливаемой ветки
==========
(подробнее: git help attributes)

Драйвер должен выходить с кодом 0 при успешном слиянии и ненулевым - при ошибке.

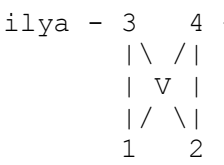
После создания драйвера в .gitattributes:
==========
*.po merge=po
==========

8) Стратегии слияния

В Git есть стратегии слияния. Стратегия по умолчанию - рекурсивная.

Стратегии:
  - recursive
  - octopus
  - ours
  - resolve
  - subtree

Перемотка не является полноценным слиянием.

Рекурсивная стратегия перед слиянием верхних коммитов осуществляет слияние родителей в т.н. виртуального предка:
==========


```
        \ /
        ...
==================
```
V выступает базой слияния для 3 и 4. Он не сохраняется и существует только в момент слияния.

У стратегии recursive есть ряд опций:
- ours - при конфликтах выбирается наш вариант
- theirs - выбирается сливаемый вариант

- renormalize - нормализует все концы строк
Полезна, если ветку прислали с иными концами строк, чем у нас

- ignore-all-space... - заставляет слияние игнорировать различие в пробелах

- no-renames - отключает магию, связанную с переименованием файлов
Нужен для того, чтобы не возникало конфликтов переименования одного и того же файла в двух ветках (потом при слиянии будет два файла, и они будут помечены как конфликтные) (переименованный файл - файл, отличающийся от другого по содержимому не более 50%)
- find-renames=n - для более точного поиска переименований (n=50 def)

- subtree=path - позволяет замёржить одну ветку под директорию другой
Это полезно если у нас есть ветка обновления какой-то сторонней библиотеки и папка с данной библиотекой в нашем проекте.
subtree без указания пути позволяет Git'у попробовать самому определить нужную папку, куда замёржить ветку. Он ищет нужную папку на основе файловой структуры, т.е. уже имеющихся файлов.
* Пакетные менеджеры не всегда удобны, если нужно постоянно обновлять сторонние библиотеки
* В Git есть команда git subtree, которая делает операции с поддеревьями более удобными
* Также есть механика подмодулей вместо поддеревьев (git submodule) (подмодуль - когда в директорию помещается не сторонняя ветка, а репозиторий целиком; при этом никаких слияний не происходит)
git merge -Xsubtree=plugin --allow-unrelated-histories plugin
(произведёт слияние, предположив, что общим предком является пустой коммит)

Опции задаются так:
```
git merge -Xours (-Xtheirs)
git merge -Xfind-renames=80
```

Стратегия octopus нужна, когда необходимо объединение более двух веток. Эта стратегия является просто объединением двух слияний в одно.
```
git merge feature1 feature2
```

Если в одной ветке ошибка, то отменить слияние именно с ней будет непросто - в этом минус стратегии octopus.
Кроме того, если есть конфликт слияния, затрагивающий более двух веток, то octopus попросту не сработает.

Лучше использовать последовательные слияния с каждой веткой по отдельности.
Но применение есть: octopus используется в больших проектах, когда надо влить много веток сразу, и эти ветки касаются разных частей проекта, работают с заведомо разными файлами и, очевидно, не конфликтуют между собой.
Т.е. упрощение истории слияний - единственный плюс использования octopus.

Стратегия ours - полностью игнорировать содержимое сливаемой ветки.
Отличие от recursive с ours в том, что файлы, добавленные в сливаемую ветку, не добавляются при слиянии (рекурсивная стратегия с ours оставляет данные файлы, но в diff одних и тех же файлов выбирает наш вариант).
ours только формально объединяет две ветки, но никаких изменений в нашей ветке не производит. Таким образом, данная стратегия является просто формальным объединением веток в истории. Изменений в нашей ветке нет.

Стратегию resolve полностью заменяет рекурсивная стратегия, поэтому первую использовать не принято.

Стратегия subtree - то же самое, что и опция subtree в рекурсивном слиянии, но в неё нельзя передавать путь.
Она тоже существует по историческим причинам, как и resolve.

git merge -s <стратегия> - выбор стратегии

10. Копирование коммитов

1) Копирование коммитов: cherry-pick

Если есть две версии файла в разных ветках и ошибка обнаружена и там, и там, то в ход идёт команда cherry-pick.
Исправляем ошибку в одной ветке и копируем на другую.

diff коммита применяется к текущей ветке, на ней создаётся новый коммит.
Данные два коммита эквивалентны, т.к. там присутствуют одни и те же изменения.
В документации Git эквивалентные коммиты называются через штрих: D и D'.

git cherry-pick <коммит> - копирование коммита

cherry-pick не занимается слиянием веток.

Лучше этим не злоупотреблять - возможно создание новой ветки:
git checkout -b fix <самый ранний коммит с ошибкой>
Исправление ошибки
Дальнейшее слияние с обеими ветками

cherry-pick лучше использовать тогда, когда в одной ветке ошибка уже исправлена другим разработчиком (коммит с исправленной ошибкой может находиться даже в середине ветки)

git cherry-pick -x - добавит в новый коммит информацию о том, откуда он был скопирован

cherry-pick умеет копировать несколько коммитов, даже целые ветки.

git cherry-pick master..feature - все коммиты, что откололись от master и принадлежат feature

При копировании нескольких коммитов может возникнуть конфликт.
Вернуть всё, как было до cherry-pick:
git cherry-pick --abort
Продолжить:
git cherry-pick --continue
Остановиться там, где мы сейчас, и сбросить запомненное состояние:
git cherry-pick --quit (у git merge данного флага нет!) (cherry-pick запоминает, какие коммиты откатить в случае --abort и какие продолжить коммитить в случае --continue - это и называется состоянием)

.git/CHERRY_PICK_HEAD - спец. файл, который использует данная команда

Если завершённый cherry-pick нужно отменить, то применяем жёсткий reset.
git reset --hard @~

Чтобы скопировать изменения, но без коммита (если нужно последующее редактирование):
git cherry-pick --no-commit (-n)

2) Просмотр эквивалентных коммитов: cherry, cherry-mark

Можно обмениваться коммитами и произвольными изменениями (т.н. патчами) в текстовом виде через форумы, чаты, эл. почту и т.д.
Делается это с помощью таких команд, как git patch, git apply и др.

Когда один разработчик применяет у себя изменения другого разработчика, то также образуется копия коммита с другим идентификатором.

Бывают и другие ситуации, при которых в разных ветках появляются эквивалентные коммиты.

Как сравнивать ветки, чтобы эквивалентные коммиты были видны?

git cherry-pick master..feature - скопировали на master разработку с ветки feature
Но в некоторых коммитах возник конфликт. Т.е. какие-то из коммитов на master не эквивалентны коммитам на feature.

git cherry master feature - выведет коммиты из feature, пометив те из них, для которых есть эквивалентные в master
- обозначает коммит с копией в master, а + - коммит без копии (немного странная запись, но как есть)

git cherry master feature -v - вывести также описание коммитов

git cherry feature - второй аргумент равен HEAD по умолчанию

Хотя конфликты при cherry-pick возникают редко, данная команда может быть полезна для отлова коммитов, для которых эквивалентных им коммитов нет.

git cherry существует по историческим причинам, сейчас её заменяет git log, просто раньше во второй команде не было данного функционала.

git log feature...master - симметрическая разность (... - коммиты которые есть в первой ветке, либо во второй, но не являются для данных веток общими)

git log --cherry-pick feature...master - удаляет из вывода эквивалентные коммиты
git log --cherry-mark feature...master - помечает эквивалентные коммиты знаком "="
git log --cherry-mark --left-right feature...master - чтобы было понятно, какой из эквивалентных коммитов из какой ветки (> - коммиты из правой ветки (master), < - левой (feature))
git log --cherry-mark --left-only feature...master - оставит коммиты только из левой ветки (feature)
Аналогично с --right-only.
git log --cherry-mark --right-only --no-merges feature...master - дополнительно удаляет из вывода коммиты слияния
Аналогом команды выше служит git log --cherry (--cherry = --cherry-mark --right-only --no-merges)

11. Перемещение коммитов

1) Перебазирование веток: rebase

git rebase позволяет переписывать историю, объединять, редактировать, менять местами коммиты.
У данного инструмента очень много применений, rebase является эдаким швейцарским ножом.

Самое базовое применение - перенос (перебазирование) веток.

git rebase master - перенесёт начало текущей ветки на вершину master (предком первого коммита в feature будет являться вершина master)

rebase работает, как cherry-pick, т.е. копирует коммиты, но когда копирование завершено, то ссылка на текущую ветку переносится на самый верхний скопированный коммит. Т.е. rebase производит дублирование коммитов, а потом текущая ветка ссылается на самый верхний из скопированных коммитов.
Старые коммиты всё ещё в базе, но они недостижимы и со временем будут удалены.

HEAD при этом передвигается на верхний коммит master (не на саму ветку, а на коммит, чтобы сама ветка master не сдвигалась при копировании коммитов).
А позже мы находимся в состоянии отделённой HEAD, поэтому git reset --hard не поможет, т.к. HEAD он не передвинет.
Для отмены: git rebase --abort
git rebase --quit - удалит служебную информацию о перебазировании, отменит коммиты, но HEAD останется там же (данная команда используется довольно редко)

При перебазировании могут возникнуть конфликты, при этом rebase останавливается, чтобы мы могли конфликт разрешить.
Состояние запоминается.

git rebase --skip - пропустить коммит
Кстати говоря, м.б. ситуация, что копируемый коммит не привносит никаких изменений, т.е. ошибки уже были исправлены в master.
Коммит, который не привносит изменения, называют пустым.
Пустые коммиты пропускаются rebase автоматически.

git rebase --continue - продолжить перебазирование

git add index.html

После перебазирования HEAD указывает на feature (не на master), если мы перебазируем feature.

git rebase создаёт .git/ORIG_HEAD, может понадобиться, если мы хотим отменить уже совершённое перебазирование.
git reset --hard ORIG_HEAD

git rebase не гарантирует сохранность ORIG_HEAD (например, если при разрешении конфликтов была допущена операция reset или при перебазировании слияний)
Поэтому в простых случаях жёсткий reset на ORIG_HEAD работает, а в более сложных случаях рефлог в помощь
(git reflog feature, удобно смотреть именно в рефлог ветки, а не HEAD, т.к. в процессе перебазирования HEAD переносится много раз, от коммита к коммиту, а ветка переносится только в конце перебазирования)
(поэтому в рефлоге ветки меньше записей - нужную запись проще найти)
(git reset --hard feature@{n} после просмотра рефлога помогает отменить перебазирование)

git rebase master feature - перебазировать feature на master
git rebase master feature = git checkout feature + git rebase master

git rebase - перебазировать текущую ветку, а куда - можно задать через конфигурацию Git (как правило, перебазирование выполняется на удалённую (remote) ветку)

2) Rebase против merge: сравнение подходов

git rebase master - перебазировать ветку feature

git merge master - слияние master с feature

Плюс перебазирования - упрощение истории разработки.

Бывает такое, что разработчик feature постоянно на всякий случай мёржит master, при этом создавая бесполезные мусорные коммиты слияния, которые ничего не содержат.
Такие коммиты только лишь усложняют чтение истории.

Однако и у rebase есть определённые ограничения и недостатки.

Если над feature работают два человека, то один может удивиться, если он работал на старой версии ветки, внёс свои изменения, а другой разработчик перебазировал feature и работал уже над ней. Очень неприятная ситуация, в этом минус git rebase для групповой работы.

Общее правило такое: пока ветка только у одного разработчика на компьютере, её можно перебазировать, сколько душе влезет, но если ветка публичная, т.е. с ней работают другие люди, то переписывание истории, включая rebase, запрещены.

Вторая проблема перебазирования - коммиты копируются, при этом один коммит до перебазирования м.б. рабочим, а после - сломанным, при этом конфликтов даже может не возникнуть.
Такая ситуация возникает, если в master что-то было изменено, например, имя функции, а в feature она вызывается по-старому.
Перебазирование сломает такой коммит, хотя и может пройти без сучка и задоринки.
При этом сломаться может не только один коммит, но и вся ветка вообще.
Даже если добавить в перебазированный feature коммит с исправлениями, то мы получим в результате пачку битых коммитов в истории разработки, что не есть хорошо.
И не дай бог потом откатиться на битый коммит!

При слиянии такой проблемы не существует, т.к. сломан будет только коммит слияния, но не сами коммиты из веток.
Принципиальное отличие здесь в том, что операция слияния безопасна.
Слияние не трогает предыдущие коммиты - в этом его огромный плюс.
Потом нам просто будет легче разобраться в происшедшем.
И исправлять надо будет всего-то один коммит.

Но проблема сломанных коммитов возникает редко, далеко не при каждом rebase.
Избегать проблемных ситуаций помогают автоматизированные тесты.

git rebase -x '...' master - запуск команды (обычно это выполнение тестов) (команда будет выполняться после каждого перебазирования отдельного коммита) (если команда завершится с ненулевым статусом, то перебазирование приостановится)

Перебазирование:
- \+ Упрощение истории
- \- Только для приватных веток
- \- Возможны ошибки в коммитах

Но перебазирование всё равно является нужной и полезной вещью.
Хорошо, если в команде разработчиков есть Code Review.

Возможно также использовать интерактивное перебазирование для корректировки истории.

Подход с Code Review удобен и приятен разработчикам, поэтому перебазирование используется командами довольно часто.
Кроме того, есть ряд случаев, где без перебазирования не обойтись, например, когда коммиты созданы не на той ветке по ошибке.

3) Тесты при rebase, rebase -x

git rebase -x 'node feature.mjs' master

При перебазировании неправильный коммит остаётся, HEAD указывает на него, поэтому его надо заменить:
git add feature.mjs
git commit --amend --no-edit
git rebase --continue

4) Перенос части ветки, rebase --onto

Бывают ситуации, когда новые коммиты мы создаём не там, где нужно.
Например, мы создали ветку fix на feature и хотим её перебазировать на master.
Но вот вопрос: как перенести на master только fix, исключая коммиты из feature?
Ведь fix является частью feature!
Но хотим мы перенести только fix, а не feature...
Для таких случаев существует флаг --onto, он указывается перед веткой, куда надо перенести.

Итак, мы на fix.
git rebase --onto master feature - перенесёт на master текущую ветку (fix), начиная с конца feature (вершина feature не в счёт, т.к. не принадлежит ветке fix)
git rebase --onto master feature fix - то же самое, только длиннее (= git checkout fix + git rebase --onto master feature)

Иногда удобнее бывает cherry-pick - она нужна, если нам надо перенести только определённые коммиты с ветки, не перенося саму ветку.
git rebase --onto feature @~2 - перенесёт 2 коммита с master на feature, да ещё и сам master перенесёт, что нам не очень нужно.
Гораздо проще и удобнее:
git checkout feature
git cherry-pick master~2..master
Передвигаем после этих команд master на два коммита назад:
git branch -f master master~2
Вуаля!

Таким образом, rebase и cherry-pick иногда заменяют и дополняют друг друга.

5) Перебазирование слияний, rebase -p

Предположим, что в ветке, которую мы хотим перебазировать, есть коммит слияния.
По умолчанию rebase пропускает коммиты слияния и делает историю ветки линейной.

Для перебазирования ветки feature команда rebase сперва ищет все коммиты, которые есть в feature, но которых нет в master.
Ветке принадлежат все предки её вершины, в т.ч. и вторые родители при слиянии.

Т.к. коммит слияния пропускается, то, если в нём есть изменения, например, в процессе разрешения конфликта, они не будут применены.
Это м.б. не совсем то, что нам нужно.

(git rebase копирует коммиты, основываясь на его diff'е с родителем)

git rebase --preserve-merges (-p) master - учитывать слияния при перебазировании
При таком подходе копируются коммиты слияния, а также все коммиты по их первому родителю, включая самого родителя.
Т.е. вторые родители коммита слияния не копируются (хотя в скопированном коммите слияния вторым родителем они выступают)

Из-за внутренней реализации работы git rebase -p изменения, которые были в коммите слияния, при копировании данного коммита игнорируются.
*** В Git идёт работа над данной проблемой, но на момент середины 2018 года это так, поэтому желательно избегать перебазирования веток со слияниями ***

Если всё же нужно выполнить перебазирование с учётом слияний, то нужно учитывать особенности, о которых рассказано в данном пункте.

6) Интерактивное перебазирование, rebase -i

Git позволяет в любой момент подредактировать историю с помощью т.н. интерактивного перебазирования.
Обычно оно совершается перед публикацией ветки или когда она доделана и хочется интегрировать её в master.
Другой случай - когда мы хотим предложить изменения в проект с открытым исходным кодом. Разработчикам проекта нужна красивая и понятная ветка, что и является отличной причиной использовать интерактивное перебазирование.

git rebase -i master - интерактивное перебазирование

Команды:
p, pick = use commit = скопировать коммит

r, reword = use commit, but edit the commit message = использовать коммит, но отредактировать его описание
e, edit = use commit, but stop for amending = выбрать коммит, но остановиться для его изменения
s, squash = use commit, but meld into previous commit = выбрать коммит, но объединить его с предыдущим, включая сообщение
f, fixup = like "squash", but discard this commit's log message = squash, но отбросить сообщение коммита
x, exec = run command (the rest of the line) using shell = исполнение команды (до конца строки)
d, drop = remove commit - удалить коммит

pick b4500d8 Create work

При удалении и перемещении коммитов могут возникать конфликты.

git rebase -x '...' -i - ставит exec '...' после каждого коммита

drop и удаление строки - одно и то же, но второе отключают, чтобы случайно не выстрелить себе в ногу.
git config rebase.missingCommitsCheck warn/error

git config rebase.abbreviateCommands true - при исходной генерации списка вместо полных команд использовать только первые буквы (pick - p, reword - r, ...)

git show
git reset @~

Нюанс: команда edit срабатывает после копирования коммита.

git commit --amend
git rebase --continue (прогресс rebase специальным образом хранится в директории .git)

git rebase --edit-todo - посмотреть и даже можно поправить, какие действия идут дальше

Интерактивное перебазирование также используют просто для редактирования, без перемещения ветки.

git reset --hard feature@{1}

git rebase -i @~3 - интерактивное перебазирование 3 последних коммитов ветки feature, т.е. простая их правка (если feature состоит минимум из 4 коммитов)
Особенность git rebase -i в том, что она копирует коммиты начиная с момента первого изменения (эдакий аналог перемотки)

Если хочется заставить rebase сделать копии всех коммитов от точки перебазирования - неважно, изменены они или нет - то есть специальный флаг:
git rebase --no-ff
Но на практике он нужен очень редко.

7) Коммиты-заплатки: rebase autosquash

Как изменить не вершинный коммит?
Выполнить интерактивное перебазирование.

Но есть доп. механизм, называющийся autosquash.

Например, в коммите есть опечатка.
*** Коммит должен быть непубличным, т.е. подходящим для перебазирования ***
Создаём коммит, где исправляем ошибку.
Коммитим:
git commit -a --fixup=@~ - к заголовку коммита добавится "fixup! "
(git commit --squash - использовать squash вместо fixup)

git rebase -i --autosquash - автоматически изменяет коммиты, у которых есть коммит-фиксап ("fixup! " + описание коммита, который нужно пофиксить)
Теперь у коммитов с "fixup! " в начале будет автоматически прописана команда fixup вместо pick.
После перебазирования два коммита будут объединены в один: старый коммит + заплатка.

git config --global rebase.autoSquash true - включить autosquash по умолчанию

12. ReReRe - Авторазрешение повторных конфликтов

Бывают ситуации, когда уже разрешённый конфликт приходится разрешать ещё раз (*)
Например, при повторении прерванного перебазирования или отменённого слияния.
Или же мы сперва подтягивали изменения в ветку при помощи постоянных слияний, а позже решили ветку перебазировать.

Повторно разрешать конфликты неохота - нужна какая-то возможность разрешать те же конфликты автоматически.
И такая возможность есть: ReReRe (Reuse Recorded Resolution).
Git будет запоминать, как разрешились конфликты, и, если повторно возникнет такая же ситуация, он автоматически разрешит эти конфликты так же, как и запомнил при первом их разрешении.

Включение:
git config rerere.enabled true

При слиянии (git merge) в состоянии включённого ReReRe в выводе возникает следующие строки:
==========
Recorded preimage for 'index.html' < до разрешения конфликта

```
Recorded resolution for 'index.html < после (git commit / git merge --
continue)
==================================
```

После отмены слияния исправления в файле неважны, если мы только не тронули сам конфликт.

```
Переиспользование:
================================================
Resolved 'index.html' using previous resolution.
================================================
```

ReReRe применяет к файлу запомненное разрешение конфликта, но сам файл не добавляет в индекс.
Чтобы автоматически добавлять файл в индекс после авторазрешения:
`git config rerere.autoUpdate true`
С другой стороны, лучше убедиться, что после авторазрешения конфликтов файл именно такой, какой нужен - поэтому включать данную настройку хоть и можно, но не рекомендуется.
Если файл не такой, как нужно, т.е. прошлый способ авторазрешения конфликта неактуален:
`git checkout --merge index.html` - достаёт файл с конфликтом
`git rerere forget index.html` - забывает способ авторазрешения конфликта
`git commit -a --no-edit`

ReReRe работает не только для слияния (*)

`.git/rr-cache/` - служебная директория, используемая ReReRe
Можно её удалить:
`rm -rf .git/rr-cache`

Старые разрешённые коммиты Git автоматически удаляет при сборке мусора, если они старше такой переменной конфига:
`rerere.rerereResolved`
По умолчанию старше 60 дней.
А старые неразрешённые, если они старше такой переменной:
`rerere.rerereUnresolved`
По умолчанию 15 дней.

Отключить ReReRe:
`git config rerere.enabled false`
  или
`git config --unset rerere.enabled`
`rm -rf .git/rr-cache` - удалять обязательно!

ReReRe имеет скрипт, который запоминает разрешённые конфликты, в т.ч. до подключения ReReRe:
`rerere-train.sh`
Обычно он находится в инсталляции Git:
`/opt/local/share/git/contrib/rerere-train.sh`

Использование:
`rerere-train.sh <коммит>` - коммит, от которого идти (часто используется, если коммит уже недостижим, напр.: `feature@{1}`)

rerere-train.sh --all - идти по всем веткам

13. Обращение коммитов

1) Обратные коммиты, revert

Для отмены коммитов можно использовать жёсткий reset:
git reset --hard @~
Или перебазирование:
git rebase

Но при разработке в команде у этих подходов есть общая проблема: если коммит уже отправлен коллегам, то его уже просто так не отредактировать и не удалить.
Локальную историю мы исправить можем, но если коммит уже получили другие разработчики и, возможно, даже сделали на его основе что-то ещё, то так просто отменить его у всех не получится.
Но выход есть, и это команда git revert.

git revert @ - смотрит, какие изменения есть в коммите, и создаёт коммит с противоположными изменениями
git revert 1913

Сообщение по умолчанию для коммита с заголовком "Create sayBye":
```
======================
Revert "Create sayBye"
======================
```

Обратные коммиты засоряют историю разработки.
С другой стороны, в Git легко обмениваться коммитами, а вот команды отмены коммита не существует.
Поэтому git revert так полезен.
Он просто м.б. единственным выходом из сложившейся ситуации.

Данная утилита может также обращать диапазон коммитов:
git revert A..D

Обратный A коммит обозначается в документации Git вот так: ^A.

Внутренний механизм команды revert и cherry-pick один и тот же.
Разница только в том, что revert создаёт не копию, а обратный коммит.
git revert поддерживает почти все флаги cherry-pick.

2) Отмена слияния через revert

При отмене коммита слияния требуется указать, какие именно изменения отменить, т.е. относительно какого родителя данного коммита.

git revert 38e8 -m 1 - отменить изменения относительно первого родителя коммита слияния

В результате будет создан т.н. коммит отмены.
Тут есть нюанс: когда мы доработали feature и нужно слить её в master, то git merge проанализирует общего предка feature и master, но это будет коммит на середине ветки feature, т.е. коммиты до общего предка, включая и его, пропадут при слиянии доработанной feature с master.

Можно скопировать потерявшиеся коммиты:
```
==========================================
~/project master> git cherry-pick 2702 2c11
==========================================
git cherry-pick 38e8 -m 1
```

А можно отменить коммит отмены:
```
git revert 0cc5
```
Этот вариант надёжнее, т.к. между коммитом слияния и его отменой могли быть какие-то другие коммиты, и при вызове revert возник конфликт, который был разрешён, но в итоге коммит отмены уже не является точной противоположностью коммита слияния. Таким образом, делая отмену отмены, мы вернём то, что было отменено, и только это.

После отмены отмены сливаем ветку заново.

Лучше всё же избегать отмены слияний.
Но если проблема в ветке большая, то всё же иногда приходится выполнить отмену слияния.

Следует заметить, что если повторное слияние делается после перебазирования, то отмена отмены не понадобится.

3) Повторное слияние с rebase

Отменили слияние веток с помощью git revert
Перебазировали ветку feature относительно вершины master:
```
git rebase master feature
```
В результате коммит-родитель для коммита слияния из старой feature после перебазирования будет учтён автоматически, и никаких отмен отмен делать не следует.
Данная хитрость делает повторное слияние с rebase легче.

Но на практике есть нюансы:
1. git rebase --onto master 54a4 feature (= git checkout feature + git rebase --onto master 54a4) (54a4 - коммит, разделяющий поток на две ветки, если забыть про слияние этих веток в будущем) - необходимо использовать --onto, иначе перебазирована будет не вся ветка, а только её часть
2. При копировании некоторых коммитов могут возникнуть конфликты (ReReRe может помочь)

После перебазирования повторно сливаем ветки:
```
git merge --no-ff --no-edit feature
```
При это после перебазирования дополнительных шагов, вроде отмены отмены, не требуется.

Но это удобно только лишь в том случае, когда мы можем спокойно перебазировать ветку и не иметь проблем синхронизации с коллегами.
В другом случае, вариант с отменой отмены вполне рабочий.

Особый случай - перебазирование ветки без её передвижения.
Если feature отходит от 54a4:
git rebase 54a4 --no-ff - если не задать данный флаг, то Git, увидев, что ветка уже отходит от данного коммита, не будет заниматься её копированием; нам же необходимо полностью скопировать ветку на то же место, где она и была - поэтому --no-ff

Итак, мы рассмотрели отмену публичных слияний в Git, а также как потом сделать повторное слияние.
Из предложенных методов можно выбрать подходящий нашему проекту в нашей конкретной ситуации.
Обычно, если ветку можно перебазировать, то её перебазируют, а если нельзя - то делают отмену отмены через revert.

14. Даты в git

1) Передача даты в гит, форматы дат

Время в Git указывается с точностью до секунды.

Форматы:
2018-01-30 12:30:00 -07:00
2018-01-30T12:30:00-07:00
Tue Jan 30 2018 12:30:00 GMT-0700
30 Jan 2018 12:30:00
2018-01-30 12:30
2018-01-30
2018.01.30
30.01.2018
Jan 30, 2018
01/30/2018 - при использовании "/" сперва месяц, потом число
1517315400

git log --pretty='%ci | %s' --before='2018-01-02' - если время не указано, то берётся текущее ('16:00' -> '2018-01-02 16:00')
Если указать только время, то в начале добавится текущая дата (по аналогии)

Формат времени м.б., например, "4pm".

Если временной зоны нет, то берётся текущая.

Часто используется следующий формат:
2018-01-30 12:30 - текущее время
3 weeks = 3 weeks ago = 3.weeks.ago = 2018-01-09 12:30 - 3 недели назад от *текущего* момента (данное обозначение относительна)
hours - days - weeks - months ...

hour = hours
day = days
week = weeks
month = months
...
Точка в относительной записи удобна тем, что не надо ставить кавычек вокруг.
--before=3.days.blablablah - так тоже можно, Git не будет ругаться, т.к. ago - украшательство
1 year 2 months 5 minutes 1 second - 2016-11-30 12:25
yesterday - 2018-01-29 12:30
midnight =~ 00:00 - 2018-01-30 00:00
noon =~ 12:00 - 2018-01-30 12:00
tea =~ 17:00 - 2018-01-29 17:00 (29, а не 30, см. ниже)
Разница между данными тремя словами и прямым указанием времени в том, что если время ещё не настало, то берётся вчера.
Данные слова гарантируют, что дата никогда не будет в будущем, но на практике они используются довольно редко.
last friday - 2018-01-26 12:30
never - 1970-01-01 00:00+00
now - 2018-01-30 12:30
Эти обозначения чаще всего используются для задания даты экспирации чего-либо.
git reflog expire --expire=now --all - удалить все записи рефлога - конкретно в этой команде в Git есть хак: она считает now за очень большую дату в будущем
--expire=all - исторический псевдоним для --expire=now, действующий только в датах экспирации

2) Форматирование для вывода дат

git log --pretty='%cd | %s' --date=<формат>

Форматы:
default - Tue Jan 30 12:30:00 2018 +0300
rfc = rfc2822 - Tue, 30 Jan 2018 12:30:00 +0300
iso = iso8601 - 2018-01-30 12:30:00 +0300
iso-strict - 2018-01-30T12:30:00+0300 - формат для программирования
short - 2018-01-30
unix - 1517304600
raw - 1517304600 +0300 - внутренний формат Git
relative - 2 months ago - относительная дата - данный формат сопровождается потерей точности, но иногда бывает удобен
format:'%F %T %z' - свой формат (обозначения такие же, как в strftime)
*-local - данный суффикс прибавляется к любому формату, кроме raw и unix, и означает, что надо вывод форматировать с учётом текущей временной зоны; например: iso-local, format-local:'%F %T'

Чаще всего используется relative или *-local (format-local:'%F %T')