GeekBrains

# Система контроля версий Git — advanced

## Методичка к уроку 2

## Работа с изменениями

# Оглавление



# Введение

В предыдущей лекции мы говорили о работе с удалёнными репозиториями. Начали с обзора систем GitHub и GitLab, со сравнения их функционала, областей применения, особенностей, преимуществ и недостатков, затем поговорили обо всех возможных способах подключения к удалённым репозиториям, изучили возможности синхронизации веток удалённого репозитория с ветками локального. Кроме того, вы познакомились с механизмом настройки связей с удалёнными репозиториями — просмотра этих связей, их добавления, изменения и удаления, а в конце лекции поговорили о разрешении типичных проблем, которые могут возникать при работе с удалёнными репозиториями и о путях их решения.

В этой лекции мы будем говорить о работе с изменениями и разберём следующие темы:

- **Жизненный цикл изменений в Git.** Изменения, которые вы сделали в файлах вашей рабочей копии, могут находиться в трёх состояниях, и об этом мы поговорим.
- **Сравнение версий и просмотр изменений.** Как сравнивать различные версии программного кода, хранящегося в вашем репозитории, между собой и как просматривать изменения между такими версиями и в отдельных файлах.
- **Игнорирование изменений, gitignore.** Из предыдущего курса вы уже знакомы с файлом .gitignore и с тем, для чего он используется. Но есть ряд тонких моментов, которые вам также будут полезны, и мы с вами их разберём.
- **Отмена несохранённых изменений.** Рассмотрим как можно отменять несохранённые изменения в репозиториях — то есть те изменения, которые ещё не закоммичены.
- **Отмена сохранённых изменений.** Закоммиченные изменения тоже можно отменять. Это можно делать разными способами, с которыми мы с вами также подробно познакомимся в этой лекции.
- **Отмена слияний веток.** Отменять можно не только обычные коммиты, но и слияния веток, причём не только успешные, но и неудавшиеся.
- **Откладывание изменений.** На практике бывает так, что вам нужно отложить всю сделанную работу и не закоммичивать её. И система контроля версий Git также предоставляет такие возможности.
- **Слияние и перемещение изменений.** Поговорим о том, как сливать и перемещать изменения между ветками. Эти знания вам также могут пригодиться в практической работе с системой контроля версий Git.

# Термины, используемые в лекции

**Индекс** — специальная область в рабочей копии Git-репозитория, в которой хранятся изменения файлов, готовые к коммиту.

# Жизненный цикл изменений в Git

Вы уже знаете, что такое коммит. Это, по сути, набор изменений в файлах и папках проекта, который вы сохранили в репозиторий под определённым

комментарием. Так вот файлы в папке, в которой есть Git-репозиторий, могут находиться в трёх состояниях, от которых зависит попадание этих файлов в коммит. Более того, вы можете при необходимости отправлять в коммит не все сделанные вами изменения, а только некоторые, управляя их состояниями.

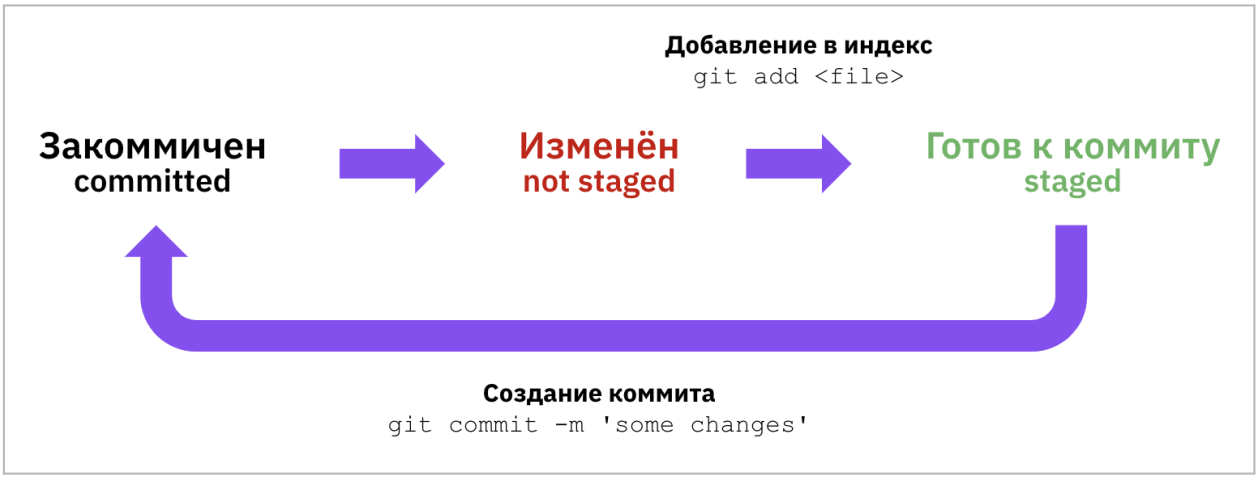
Первое состояние — это состояние, в котором файл, вместе со всеми изменениями, сохранён в репозитории, то есть, закоммичен. Если вы измените существующий файл или добавите новый файл, то он будет находиться в состоянии “изменён”.

Измененные файлы вы можете подготовить к коммиту, но необязательно добавлять в коммит все изменения, которые вы сделали. Например, вы внесли изменения в четыре файла, но оказалось, что для решения задачи вам нужно оставить изменённым только один из них. В этом случае вы готовите к коммиту только его, и только он перейдет в статус "закоммичен" и попадёт в репозиторий, а остальные три так и останутся в состоянии "изменен", и эти изменения не будут находиться в репозитории.

Подготовка файлов к коммиту называется добавлением в индекс и делается командой “git add”, после которой указывается имя файла, который мы в этот индекс добавляем. Ранее вы писали здесь точку, что означало, что в индекс нужно добавить все изменения в текущей папке — ну то есть все изменения, внесённые в проект.

Индекс в Git — это специальная область, в которой хранятся ещё незакоммиченные, но уже готовые к коммиту изменения файлов. То есть готовые к добавлению в репозиторий из рабочей папки. Если вы делаете коммит, то в него попадают только те изменения, которые были добавлены в индекс. Коммитить файлы, как вы уже знаете, можно командой “git commit” с комментарием о том, какие изменения были внесены.

Вот сводная диаграмма, иллюстрирующая переходы изменений в файлах и папках между состояниями:




Текущее состояние файлов в рабочем каталоге можно просматривать командой “git status”. Она отображает и те файлы, которые изменены, и те, которые готовы к коммиту. Изменённые файлы при этом отображаются красным цветом, а готовые к коммиту — зелёным (см. ниже).

Если взять условный проект, изменить в нём два файла, удалить два файла и добавить два файла и потом файлы по одному из каждой группы добавить в индекс, можно посмотреть на полный вывод команды “git status”:

```
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   index.html
        new file:   info.html
        deleted:    styles.css

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   script.js
        deleted:    test.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        README.md
```

Обратите внимание, что в результате выполнения команды “git status” вывелось аж целых три раздела с разными типами изменений файлов:

1. Раздел с файлами, добавленными в индекс. Обозначен как “Changes to be committed”. В этом разделе имена файлов отображаются зелёным цветом.

2. Раздел с изменёнными файлами, изменения в которых отслеживаются Git-ом. Этот раздел обозначен как “Changes not staged for commit”. Здесь имена файлов отображаются красным цветом.

3. Раздел с изменёнными файлами, которые не отслеживаются Git-ом. Этот раздел обозначен как “Untracked files”. И в нём файлы тоже отображаются красным цветом, поскольку эти файлы, как и файлы из раздела 2, находятся в изменённом состоянии и не добавлены в индекс.

Обратите внимание, что в разделах 1 и 2 слева от этих файлов написано, что конкретно с этими файлами произошло: “modified” означает, что файл был изменён, “new file” — что он был добавлен, а “deleted” — удалён.

Ещё раз отмечу, что красным отображаются те изменения, которые не добавлены в индекс и в коммит не попадут, а зелёным — те, которые находятся в индексе и в коммит попадут. Есть выполнить коммит, то после него команда git status будет выводить следующее:

```
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   script.js
        deleted:    test.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        README.md

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

Здесь мы видим, что все изменения из индекса, которые были зелёными, пропали, и в индексе изменений не осталось. В связи с этим Git нам пишет:

```
not changes added to commit (use "git add" and/or "git commit -a")
```

Как мы уже увидели с вами выше, новые файлы, которые только были добавлены в репозиторий, отображаются командой “git status” отдельно в раделе “untracked”. Это те файлы, изменения в которых пока не отслеживаются системой контроля версий Git. Но после того, как мы их добавим в индекс и закоммитим, изменения в них также начнут отслеживаться, как и во всех остальных файлах.

# Сравнение версий и просмотр изменений

Ключевой задачей систем контроля версий, в том числе, системы Git, является отслеживание изменений — в коде, файлах и папках того или иного проекта. И очень часто возникает необходимость увидеть эти изменения — посмотреть, какой код был дописан или исправлен и какие файлы и/или папки добавлены, изменены или удалены.

В этом блоке мы посмотрим, как можно сравнивать различные версии файлов, лежащих в системе Git, и о том, как просматривать изменения, сделанные в этих файлах. Чаще всего при работе в Git требуется просмотреть следующие изменения:

- В коде одного файла перед коммитом
- В проекте целиком перед коммитом
- В файле между двумя коммитами
- В проекте целиком между двумя коммитами
- Между коммитами, находящимися в разных ветках
- В коде файла, лежащего вне Git-репозитория

Если вы измените файл и захотите просмотреть изменения, которые в в нём произошли относительно последней закоммиченной версии, вы можете выполнить команду “git diff”, указав имя этого файла, например:

```
git diff index.html
```

Эта команда отобразит изменения в этом файле и отметит красным цветом удалённые строки, а зелёным — добавленные. Если какая-либо строка была

изменена, то на её месте будут отображены две строки: “старая” красным цветом и “новая” — зелёным. Слева от зелёных строк будет стоят символ “+”, также обозначающий, что эти строки были добавлены, а справа — символ “–”, обозначающий, что соответствующие строки были удалены.

Вот пример вывода команды “git diff” применительно к файлу из которого удалили одну строку, одну строку изменили (убрали пробел и слово “проектами”), а одну строку — добавили:

```
 <html>
     <head>
         <title>Система управления проектами</title>
-        <script src="script.js"></script>
     </head>
     <body>
-        <h1>Система управления проектами</h1>
+        <h1>Система управления</h1>
         <p>Текст описания системы, подробный и детальный, текст
ы, подробный и детальный, текст описания системы, подробный и де
писания системы, подробный и детальный, текст описания системы,
льный, текст описания системы, подробный и детальный, текст опис
дробный и детальный, текст описания системы, подробный и детальн
ия системы, подробный и детальный, текст описания системы, подро
</p>
         <hr>
         <p>Some description text text text text text text</p>
         <p>Some description</p>
+        <button>Новая кнопка</button>
     </body>
 </html>
```

Вот так выглядят изменения в одном файле. Но чаще всё же изменения вносятся сразу в несколько файлов, и Git также предоставляет возможность просмотреть сразу все такие изменения. Для этого нужно вызвать команду без параметров:

```
git diff
```

Так же, как показано выше, будут выведены изменения, но сразу во всех файлах. Стрелками на клавиатуре можно прокручивать эти изменения, если они

не уместились на экране. Чтобы выйти из просмотра изменений, нужно нажать на клавиатуре клавишу “q”.

Просмотр незакоммиченных изменений — это удобно, но чаще всё же требуется просмотреть изменения между двумя коммитами — как в пределах одной ветки, так и между коммитами разных веток. Чтобы просмотреть изменения между коммитами в конкретном файле, необходимо выполнить команду “git diff”, указав хэши этих коммитов и имя файла, изменения в котором вам требуется просмотреть, например:

```
git diff 5562fa353 d5e73e4b2 index.html
```

Хэши коммитов, как вы знаете, можно посмотреть командой “git log”. При этом не обязательно указывать их полностью, можно указать хотя бы несколько первых символов. На момент написания этого текста на компьютере автора команда “git diff” корректно работала при указании первых четырёх символов хэшей каждого из коммитов.

Первым указывается более ранний коммит, вторым — более поздний. И изменения отображаются так, как будто они произошли между первым и вторым указанным коммитом. Если коммиты указать в обратном порядке, то и изменения будут отображаться обратные: строки, которые были добавлены, будут показываться как удалённые, и наоборот.

Также бывает необходимость посмотреть изменения во всём проекте между коммитами. Для этого можно также использовать команду “git diff”, но просто не указывая имя файла:

```
git diff 5562fa353 d5e73e4b2
```

Если вы сравниваете какой-то коммит с последней закоммиченной версией, хэш последнего коммита можно не указывать, достаточно будет только хэша одного коммита:

```
git diff 5562fa353
```

Сравнить коммиты можно даже, если они находятся в разных ветках, также указав их хэши. В рамках одного репозитория хэши всех коммитов разные, и поэтому имена веток указывать не обязательно.

Ну и, наконец, команду “git diff” можно использовать и просто для сравнения двух файлов, которые не находятся в Git-репозитории, для этого нужно выполнить команду “git diff”, указав имена этих файлов через пробел, например, так:

```
git diff index.html index2.html
```

Такая команда будет работать прекрасно, даже несмотря на то, что мы сравниваем файлы, которые не лежат в Git-репозитории. Если имена файлов после команды “git diff” поменять местами, изменения также будут отображаться наоборот, как и в случае с хэшами коммитов.

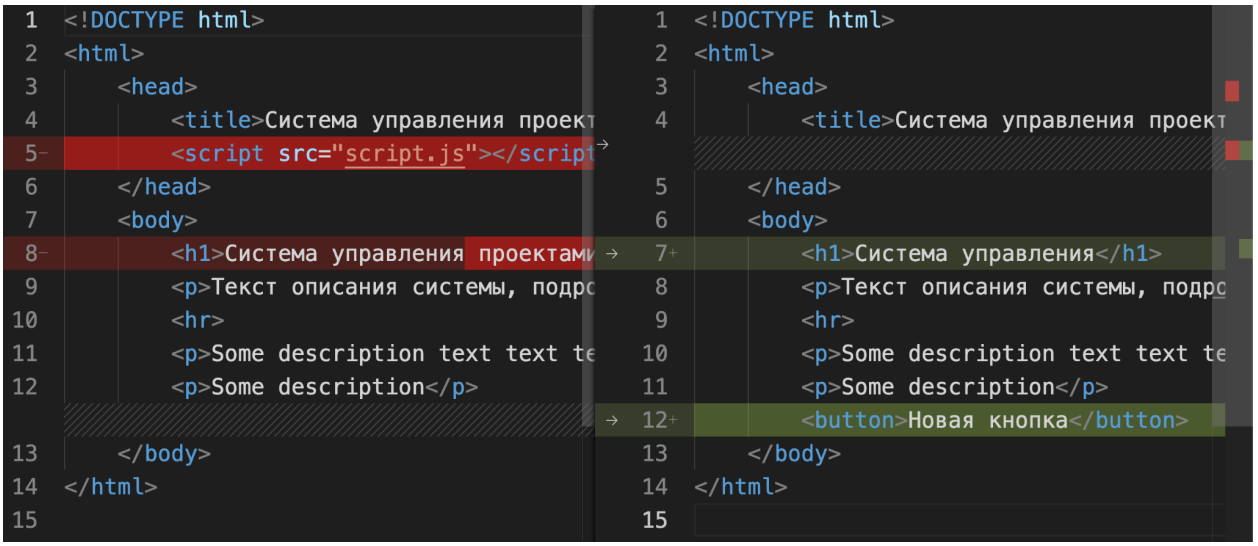
Часто бывает необходимость увидеть, какие изменения какими пользователями и в рамках каких коммитов вносились в тот или иной файл. Это можно сделать, выполнив команду “git blame” и указав имя интересующего нас файла:

```
git blame index.html
```

Эта команда выведет строки файла и напротив каждой строки укажет и коммиты, и время внесения изменений, и имена пользователей, которые их вносили. Вот фрагмент вывода этой команды:

```
fb8570d9 (Daniil Pilipenko  2022-08-17 11:56:45 +0300  1) <!DOCTYPE html>
fb8570d9 (Daniil Pilipenko  2022-08-17 11:56:45 +0300  2) <html>
fb8570d9 (Daniil Pilipenko  2022-08-17 11:56:45 +0300  3)     <head>
fb8570d9 (Daniil Pilipenko  2022-08-17 11:56:45 +0300  4)         <title>Система
```

На самом деле, инструменты такого просмотра есть практически в любой среде разработки, и там они даже более удобны, чем в командной строке:




Это пример того, как в среде разработки Visual Studio Code выглядит сравнение двух версий файла. Здесь аналогичным образом используются красный и зелёный цвета: красным обозначены удалённые строки, а зелёным — добавленные.

# Игнорирование изменений

В реальной работе над проектами часто возникают ситуации, когда вы не хотите, чтобы в репозитории сохранялись определённые файлы и изменения в них. В этом случае вы можете игнорировать изменения в некоторых файлах, используя специальный файл .gitignore. Вы уже знаете, как его использовать, и в этом блоке мы кое-что с вами повторим и несколько расширим ваши знания.

Начнём с того, что некоторые типы файлов вообще не рекомендуется сохранять в Git-репозиториях. Во-первых, потому что это бессмысленно, а во-вторых, они будут занимать в репозитории лишнее место, каждый раз при изменении отображаться красным цветом или даже попадать в коммиты. Вот эти типы файлов. Исключение таких файлов из отслеживания Git-ом сделает работу с репозиторием быстрее и удобнее не только для вас, но и для других разработчиков, которые будут его использовать.

Что же это за файлы? Это, прежде всего, файлы, не содержащие редактируемый программный код. Они могут появиться в проекте при его создании, и тогда вы сможете сразу их исключить, либо могут появиться тогда,

когда проект уже работает, и тогда вы сможете исключить их в процессе. Давайте перечислим основные типы таких файлов и разберём их по отдельности.

**Лог-файлы (логи).** В логах записывается различная информация по тому, как работает конкретное приложение, но в репозитории с кодом им не место.

Во-первых, потому что они постоянно меняются и будут попадать в коммиты, по сути, не имея к этим коммитам никакого отношения. Во-вторых, потому что они нужны только в рамках конкретного развёрнутого где-то проекта: если проект кто-то будет себе клонировать, то он будет его запускать на своём компьютере, где будут записываться новые логи, имеющие отношение только к этому компьютеру. Ну и, в-третьих, логи могут быть очень большие, и они будут просто занимать место в репозитории вместе с историей всех своих изменений.

**Файлы, загруженные пользователями.** Например, загруженные пользователями на сайт аватарки, какие-то другие изображения, PDF-файлы или что-то аналогичное. Такие данные следует хранить отдельно и не в репозитории, поскольку они, как правило, занимают много места и их версии не требуется отслеживать.

**Служебные файлы сред разработки.** Такие файлы обычно содержат различные настройки среды конкретного разработчика. Если новый опытный разработчик скачает такой проект в свою среду, он не захочет, чтобы чужие настройки применялись к его среде. Здесь есть некоторые нюансы, но, в основном, такие файлы всё же исключают из отслеживания Git-ом.

**Внешние библиотеки.** Их тоже не принято хранить в репозитории. Они, как правило, доступны в Интернете, и их всегда можно установить при очередной сборке проекта.

**Файлы локальной конфигурации.** Если в вашем проекте есть конфигурация, зависящая от того, на каком сервере или компьютере он запускается, например, файлы с параметрами доступа к базе данных, то такие файлы локальной конфигурации также принято исключать из репозитория.

В таких случаях можно добавлять файлы конфигурации с незаполненными данными и специальным расширением “dist”, чтобы любой желающий мог себе такой файл скопировать и внести в него параметры, относящиеся к той машине,

на которой данный проект будет разворачиваться и работать. Либо создавать файлы с так называемыми переменными локального окружения, которые также нужно исключать из отслеживания.

**Файлы операционной системы.** Специальные файлы операционной системы, которые иногда автоматически добавляются в папки, но не относятся к программному коду и проекту. Например, файлы desktop.ini в ОС Windows или .DS_store в ОС MacOS.

**Очень большие файлы.** Слишком большие файлы, не содержащие программный код, тоже не следует хранить в репозиториях. Например, какой-нибудь бинарный файл размером в 50 мегабайт, который вы никогда не будете изменять и который будет только увеличивать размер репозитория и время его скачивания.

Давайте теперь посмотрим, как мы можем исключить такие файлы и папки из нашего проекта и делать так, чтобы изменения в них не отслеживались Git-ом, не отображались командой “git status” и не попадали в коммиты.

Как вы уже знаете, делать это мы можем путём добавления таких файлов в файл с именем “.gitignore”. Чтобы не отслеживать просто какой-то отдельный файл, его нужно просто указать в отдельной строке:

```
errors.log
```

Если вам нужно исключить из отслеживания все файлы с каким-нибудь расширением, например, все файл с расширением *.log, вам так и нужно написать в файле “.gitignore”:

```
*.log
```

Символ звёздочки будет обозначать, что вместо неё может быть написано всё, что угодно, но в конце имени должно быть обязательно “.log”, и тогда такой файл не будет отслеживаться Git-ом в этой директории.

Из отслеживания можно также исключать целые папки. Для этого необходимо указать её имя и слэш в конце:

```
logs/
```

Слэш в конце говорит о том, что это именно папка, а не файл. Если слэш убрать, то такая команда будет указывать на то, что нужно игнорировать и папку “logs”, и файл с именем “logs”, если он тут появится. А со слэшем будет игнорироваться только папка.

Ещё один важный элемент синтаксиса файла “.gitignore” — это отрицание паттерна. Часто бывает необходимо задать напрямую правило, указывающее, что конкретный файл или папку игнорировать не следует. К примеру, мы можем начать игнорировать все лог-файлы, лежащие в папке “log”, но какой-то специальный файл исключить из игнорирования, для чего перед его именем нужно поставить восклицательный знак:

```
logs/*.log
!logs/special.log
```

Кстати говоря, если бы мы в “.gitignore” указали всю папку целиком, то потом исключить из неё отдельный файл было бы нельзя. То есть вот так работать не будет:

```
logs/
!logs/special.log
```

Нельзя включить в отслеживание файл из папки, которая исключена целиком.

# Отмена несохранённых изменений

При работе с программным кодом часто бывает так, что вы начали вносить какие-то изменения, а потом решили, что этого делать не нужно. Это может касаться как отдельного файла, так и проекта целиком. В этом блоке мы разберём способы удаления таких изменений.

Итак, представьте, что вы внесли какие-нибудь изменения в какой-нибудь один файл, например, в файл “index.html”. Команда “git status” будет выдавать вам следующее:

```
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:   index.html
```

Здесь будет уже знакомая вам фраза “use "git add <file>..." to update what will be committed”, а после неё будет написано: “use "git restore <file>..." to discard changes in working directory”. То есть, чтобы вернуть файл к исходному состоянию, нужно выполнить команду:

```
git restore index.html
```

Её выполнение приведёт к тому, что сделанные в этом файле изменения исчезнут. Если же файл уже был добавлен в индекс, то его можно вернуть из индекса аналогичной командой, но с параметром “--staged”:

```
git restore --staged styles.css
```

Если же вы изменили в проекте сразу множество файлов и решили удалить все эти изменения, то вызывать команду “git restore” для каждого файла будет уже не так удобно, и вы можете воспользоваться командой:

```
git reset --hard
```

С этой командой и её параметрами мы ещё познакомимся подробнее в одном из следующих блоков, а пока мы показываем вам лишь один из вариантов её работы — удаления всех незакоммиченных изменений. В том числе, кстати, тех изменений, которые были добавлены в индекс.

При этом команда “git reset” не удаляет те файлы, которые не отслеживаются и ещё ни разу не добавлялись в индекс — файлы, которые находятся в состоянии “untracked”. Для того, чтобы их удалить, можно воспользоваться командой:

```
git clean -f
```

🔥 Обращаем ваше внимание на то, что использовать эти команды нужно осторожно, поскольку они удаляют все незакоммиченные изменения безвозвратно. В норме система контроля версий Git работает таким образом, чтобы ничего никогда не было потеряно, и все изменения были сохранены. Но если вы их просто удаляете одним из вышеперечисленных способов, то восстановить их уже не получится.

Также хотелось бы вам показать способ удаления файла из отслеживания системой контроля версий Git. Зачем вам это может пригодится? Представьте, что у вас в проекте есть какой-то файл, который не следует хранить в Git'е, но вы этот файл случайно закоммитили вместе с какими-то другими изменениями в коде. Потом этот файл менялся, и перед очередным коммитом вы, наконец, заметили, что этот файл отслеживается Git'ом и решили добавить его в ".gitignore".

Но это не поможет: файл не исчезнет из отслеживания, и даже если вы сбросите его с помощью "git restore", изменения также продолжат в нём отслеживаться. Чтобы Git перестал отслеживать изменения в этом файле, его нужно удалить из отслеживания командой:

```
git rm --cached file_name
```

В этой команде параметр "rm" расшифровывается как "remove", то есть по-английски "удалить", а "cached" означает, что нужно не удалить файл из рабочего каталога, а только перестать его отслеживать.

# Отмена сохранённых изменений

Теперь вы знаете, как можно удалять незакоммиченные изменения, но всё же, в основном, в Git приходится работать именно с коммитами, и в этом блоке мы покажем вам, как отменять закоммиченные изменения — как в отдельном файле, так и целиком в проекте.

Начнём с файлов. Если вам нужно вернуть файл к конкретной версии на момент определённого коммита, вы можете выполнить уже знакомую нам команду "git checkout", но указать не имя ветки, как раньше, а имя коммита и имя этого файла, например:

```
git checkout b45a983 index.html
```

И этот файл будет возвращён к нужной версии и сразу добавлен в индекс. Вы можете закоммитить эти изменения — то есть зафиксировать эту версию файла, а можете их пока отменить: либо командой “git reset --hard”, которая может быть полезна сразу для нескольких файлов, либо обычно предлагаемой в таких случаях Git’ом командой:

```
git restore --staged index.html
```

Она переведёт изменения из индекса в изменённое состояние. после чего удалить эти изменения совсем можно командой:

```
git restore index.html
```

Если же вы хотите отменить коммит целиком, вы можете воспользоваться командой “git revert”. С помощью неё вы можете отменить как последний коммит, так и любой предыдущий. Пример выполнения команды:

```
git revert b45a983
```

При этом указывать нужно коммит к состоянию которого нам нужно вернуть ветку, а не тот, который нужно отменить. После выполнения этой команды вам, возможно, будет предложено отредактировать сообщение коммита, после чего для сохранения этого сообщения и выхода из редактора vim вы можете выполнить команду:

```
:wq
```

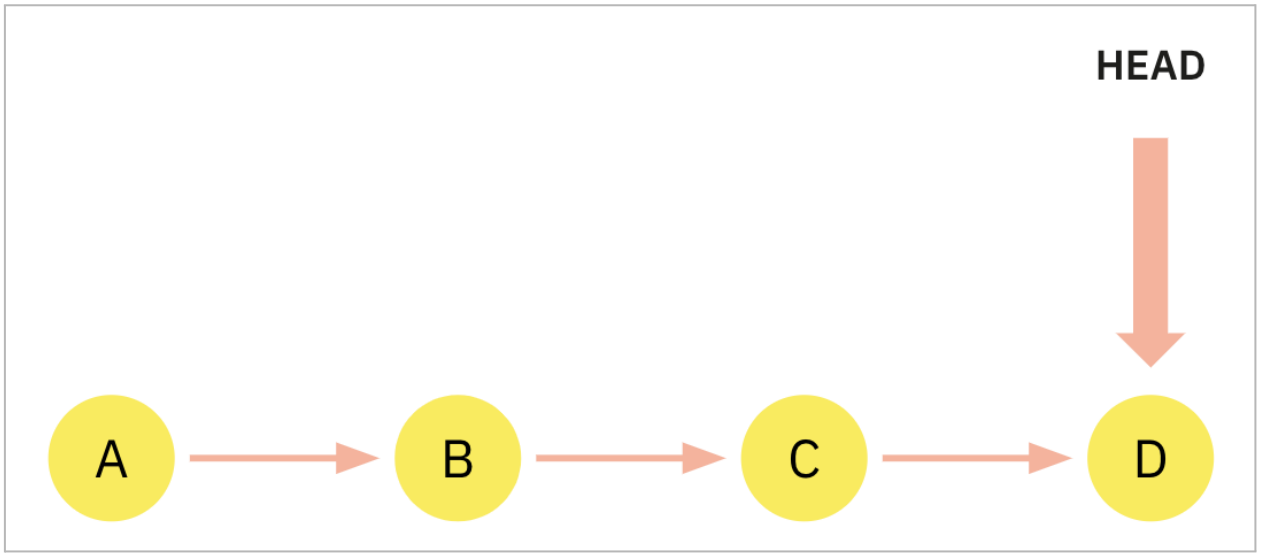
Обратите внимание, что “git revert” осуществляет отмену конкретного коммита, то есть все последующие коммиты будут отменены. Если вам нужно отменить несколько коммитов, то это можно сделать последовательным выполнением команды “git revert”:

```
git revert --no-commit hash3
git revert --no-commit hash2
git revert --no-commit hash1
```

Если указывать при этом параметр “--no-commit”, вы сможете откатиться на несколько версий назад, не делая множество коммитов, а сделав единственный коммит в конце.

Теперь давайте детально рассмотрим отмену — или сброс — самих коммитов. Такой сброс можно делать командой “git reset”, у которой есть три основных режима работы — мягкий, жёсткий и смешанный — soft, hard и mixed.

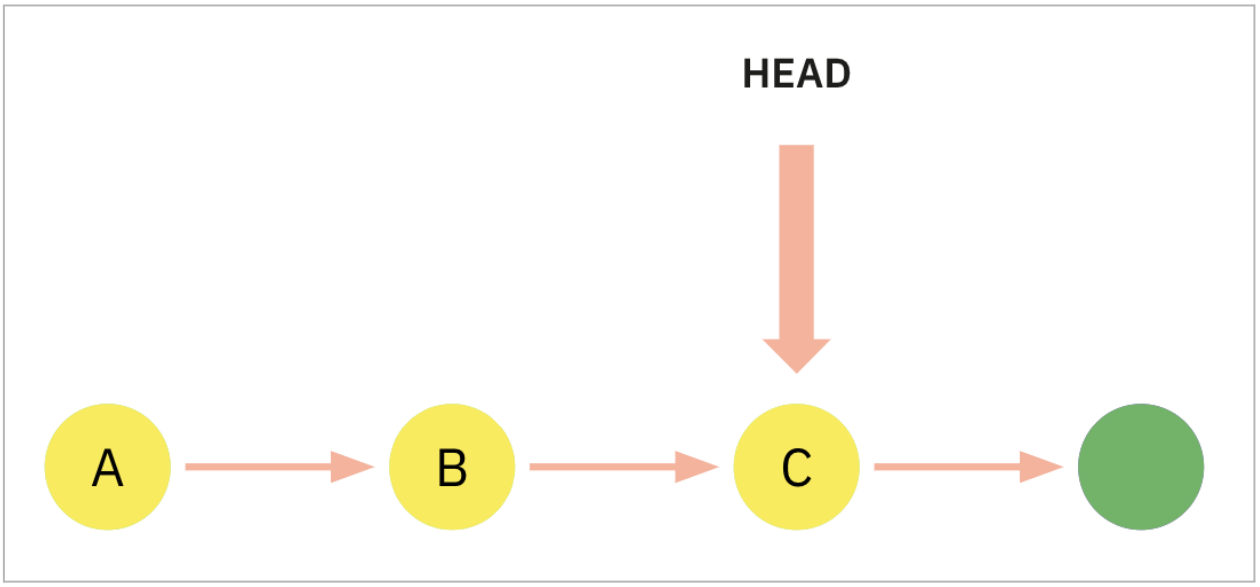
Для того, чтобы разобраться с командой “git reset”, нам нужно познакомиться с важным понятием — понятием указателя HEAD. “Head” в переводе с английского означает “голова”. Представьте, что у вас есть ветка с коммитами, и HEAD — это указатель на последний коммит текущей ветки. То есть если мы ветку переключим на другую, то этот указатель будет указывать уже на её последний коммит:




Если мы выполним команду “git reset” в “мягком варианте” и укажем в качестве параметра предпоследний коммит:

```
git reset --soft C
```

...то мы сбросим изменения последнего коммита и отправим их в индекс:

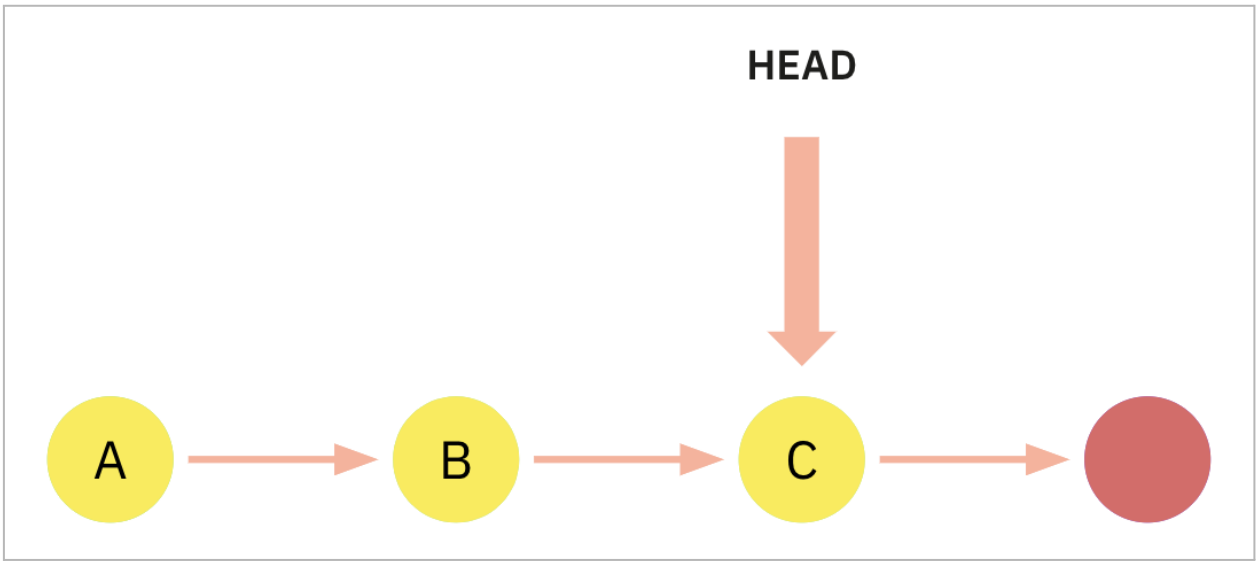



То есть если вы после этого снова сделаете коммит, то этот коммит будет полностью аналогичен коммиту “D”, который вы сбросили.

И если мы выполним команду “git reset” в смешанном варианте — с параметром “--mixed”:

```
git reset --mixed C
```

...то мы снова сбросим изменения до указанного коммита, но только всё, что было сброшено, будет сохранено в виде незакоммиченных и недобавленных в индекс изменений в рабочей директории:



Кстати, “mixed” — это вариант команды “git reset” по умолчанию, поэтому его можно не указывать, и это будет сброс именно по смешанному варианту:

```
git reset C
```

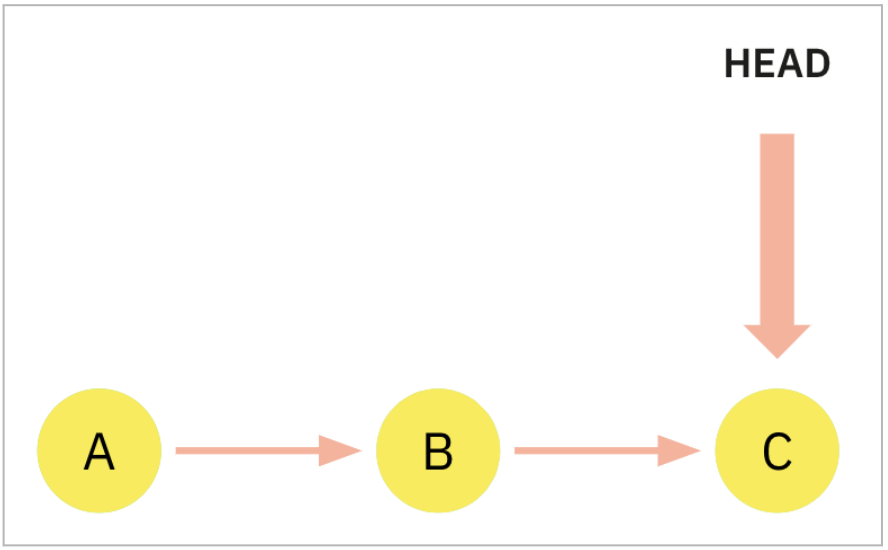
То есть, ещё раз, такой сброс приведёт к тому, что последующие коммиты будут отменены, а все изменения, которые в них содержались, вернутся в рабочий каталог, и они не будут добавлены в индекс. Чтобы сделать коммит, идентичный последнему, нам нужно сначала, как обычно добавить всё в индекс, а затем закоммитить:

```
git add .
git commit -m 'changes'
```

Последний вариант сброса коммитов, который мы разберём — жёсткий. С ним вы уже знакомы из предыдущих тем, но мы выполняли эту команду без параметров и, по сути, сбрасывали изменения до последнего коммита, не отменяя сам коммит. Если вы выполните жёсткий вариант команды “git reset”, с параметром “--hard”, и укажете конкретный коммит:

```
git reset --hard C
```

...то изменения в последующих коммитах просто будут безвозвратно удалены:



Обратите особое внимание на то, что коммиты будут удалены безвозвратно, поэтому использовать этот, самый “зверский”, вариант команды “git reset” следует крайне осторожно и только тогда, когда вы точно уверены в своих действиях.

В заключение этой темы хотелось бы рассказать об изменении последнего коммита, поскольку такая необходимость тоже часто возникает на практике. Например, если вы вдруг поняли, что написали сообщение последнего коммита с ошибкой, вы можете выполнить команду:

```
git commit --amend -m 'new comment'
```

И сообщение коммита изменится на новое “new commet”. Либо если вы вдруг поняли, что забыли добавить какой-то файл в последний коммит, вам после его изменения нужно его просто добавить в индекс:

```
git add .
```

...а затем добавить в последний коммит аналогичной командой:

```
git commit --amend --no-edit
```

Параметр “no-edit” означает, что вы не хотите редактировать сообщение коммита.

# Отмена слияния веток

Отмена слияния веток — это отдельный тип изменений. Часто бывают ситуации, когда вы сделали “merge”, но потом поняли, что сделали его зря. Например, в ветку “master” влили ветку, в которой была выполнена какая-то задача, а потом поняли, что задача недоделана или её вливание сломало всё остальное, что было в ветке “master”. Чтобы отменить такое слияние, вы можете выполнить специально предназначенную для этого команду:

```
git reset --merge hash
```

При этом нужно указать коммит, до которого вам нужно откатиться. В данном случае – коммит, предшествующий коммиту со слиянием, в котором был зафиксирован “merge”.

В случае, если вы выполнили команду “git merge”, но слияние не произошло из-за конфликтов, вы также можете его отменить, но уже при помощи другой команды:

```
git merge --abort
```

Она отменит все незакоммиченные изменения, произошедшие в рабочем каталоге в результате неудачного слияния.

# Откладывание изменений

Система контроля версий Git позволяет откладывать незакоммиченные изменения. Когда это может быть необходимо? Представьте, что вы работаете над какой-то задачей, и тут к вам приходит ваш руководитель и просит срочно сделать другую задачу. Сделать коммит при этом вы не можете, поскольку коммитить принято только законченные задачи. Вы также не сможете переключиться на другую ветку. На этот случай Git предлагает специальный инструмент, позволяющий временно отложить сделанные изменения. Чтобы изменения отложить, вам нужно просто выполнить команду:

```
git stash
```

Если после выполнения этой команды вы посмотрите статус рабочей директории, то в ней не будет никаких незакоммиченных изменений, а её состоянии будет соответствовать последнему коммиту. После этого вы сможете перейти к выполнению другой задачи и, когда её закоммитите, сможете вернуть более ранние изменения из отложенных командой:

```
git stash pop
```

При этом отложенные изменения будут слиты с теми, которые вы только что закоммитили. Возможно, что это произойдёт автоматически, и тогда вы увидите просто сообщение со словом “auto-merging”, либо же произойдёт

конфликт слияния, который вам придётся разрешать вручную аналогично любому другому конфликту слияния, что вы уже умеете делать.

Команда “git stash” позволяет откладывать несколько порций изменений. Вы можете внести изменения, отложить их с помощью “git stash”, потом внести новые изменения, и снова отложить их с помощью “git stash”, и они будут сохранены отдельно. Теперь, если вы будете выполнить команду “git stash pop”, вы будете возвращать изменения в обратном порядке: сначала те, которые были добавлены самой последней командой “git stash”, затем — предпоследней, и так далее.

При необходимости вы также можете отменить последнюю порцию отложенных изменений командой:

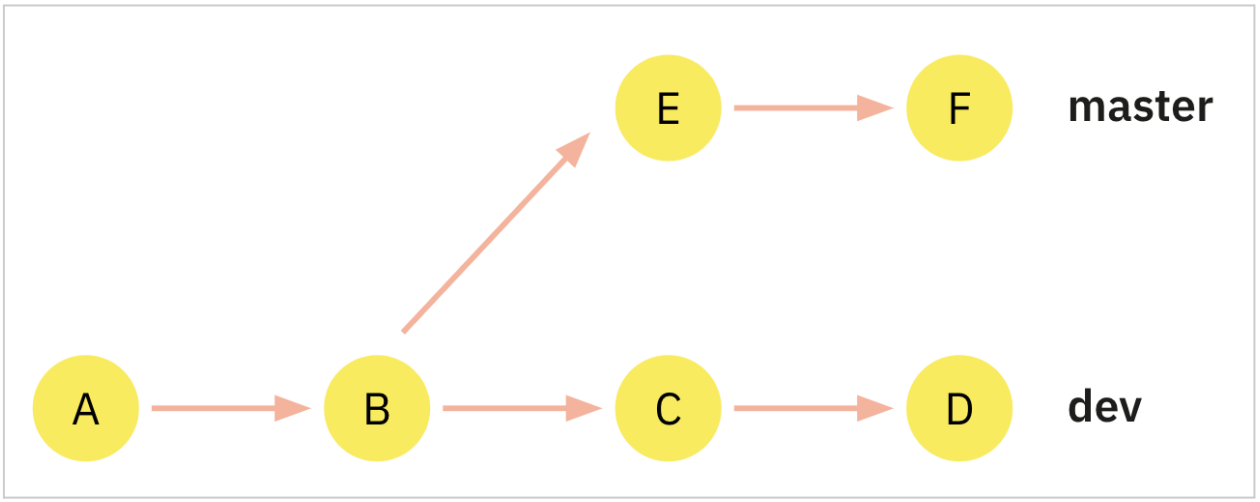
```
git stash drop
```

Вы также можете выполнять эту команду несколько раз, чтобы последовательно отменить все порции изменений, отложенных последовательным выполнением команды “git stash”.

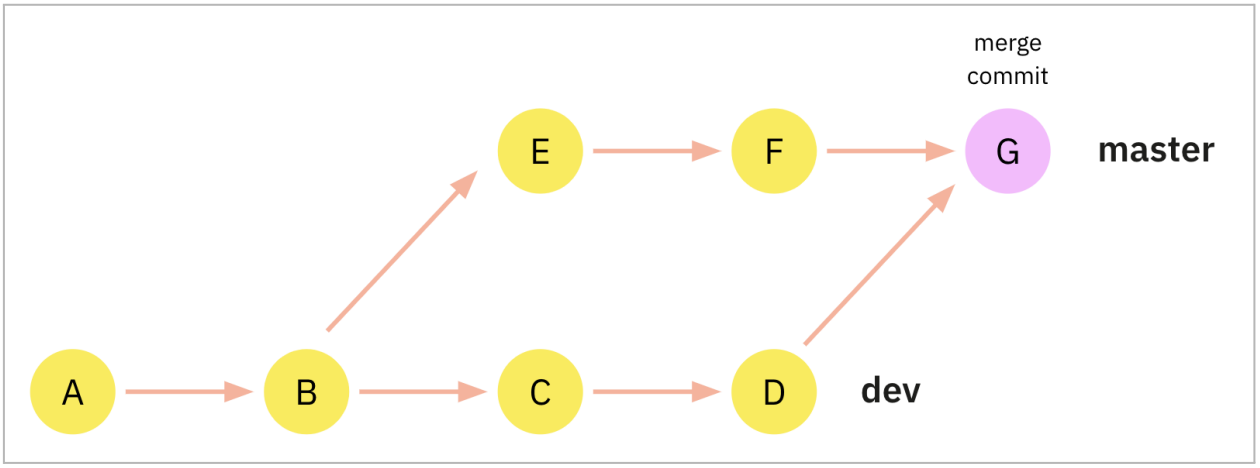
# Перемещение изменений

Набор изменений, то есть сделанных коммитов, вы также при необходимости можете переместить между ветками. Ранее вы нечто подобное делали только при помощи механизма слияния. Давайте вспомним детально, как работает merge в контексте коммитов.

Представьте, что у вас есть две ветки — dev и master. Сначала была создана ветка dev, потом от неё возникла ветка master, которая развивалась как-то отдельно:




И тут решили, что пора бы накопившиеся в ветке dev изменения отправить в ветку master. Это можно сделать при помощи слияния — команды “git merge”. В результате её выполнения у вас в ветке master возникнет новый коммит, содержащий все изменения из ветки dev, которых ещё не было в ветке master:



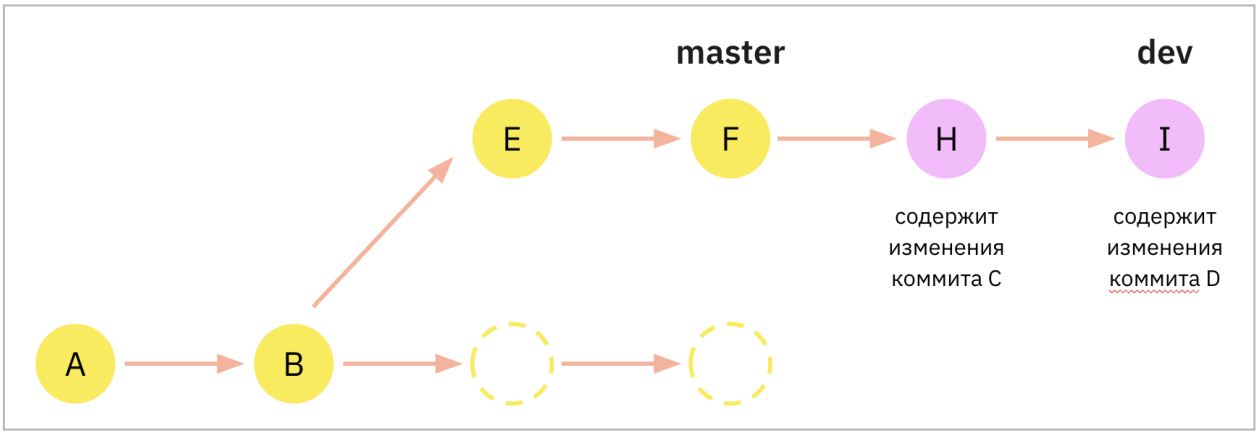

Напомним, что в данном случае эту команду нужно выполнять в ветке “master”, в которую вы вливаете ветку “dev”:

```
git merge dev
```

В примере на изображении выше коммиты, обозначенные буквами C и D, попадут в единый коммит G в ветке master. И в случае удаления ветки dev восстановить историю этих двух коммитов уже не получится. Эта особенность считается одним из минусов механизма слияния. И, если вам нужно сохранить историю коммитов — перенести её целиком в другую ветку, существует специальный механизм, который называется rebase. Команда при этом будет выглядеть так:

```
git rebase dev
```

И если мы выполним rebase, мы изменим родительский коммит нашей ветки. По сути, мы перебазируем нашу ветку на ветку master:



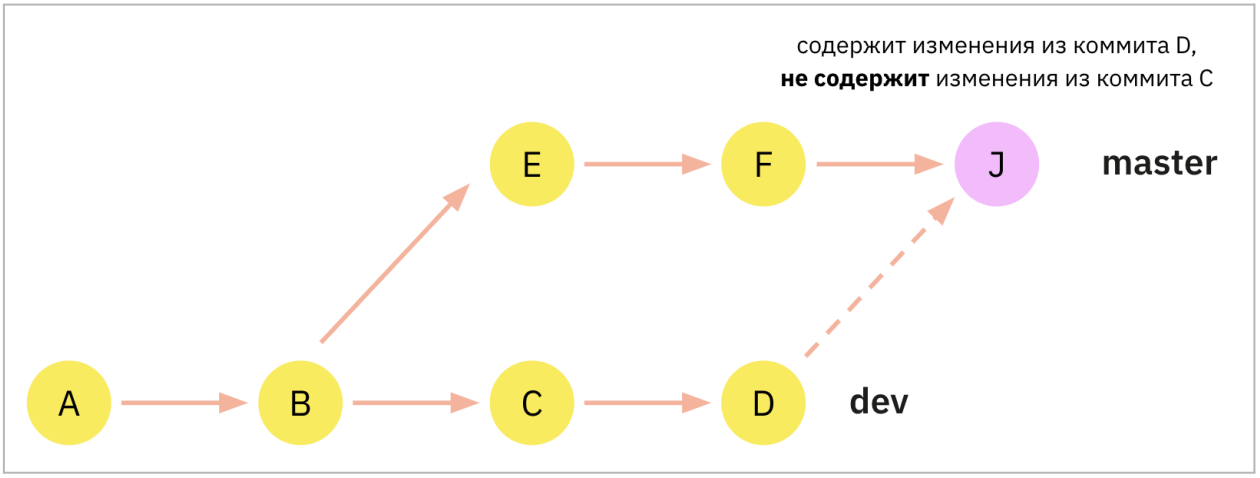

Поскольку мы переносим эти изменения после изменений в ветке master, то это будут уже немного другие коммиты, которые в себе будут содержать изменения перенесённых коммитов, наложенные на изменения из ветки master. И, разумеется, хэши этих коммитов тоже будут другие. Но каждый из них будет содержать в себе изменения из соответствующих перенесённых коммитов.

Преимущество этого способа заключается в том, что история изменений остаётся прямой и удобной для просмотра и анализа, но при этом есть и недостаток: в будущем по такой истории будет достаточно сложно понять, что произошёл именно git rebase.

Третий механизм, который важно знать, это cherry-pick. Этот механизм позволяет перенести какой-то один или несколько коммитов из одной ветки в другую без переноса других изменений. Например, в какой-то ветке в отдельном коммите был исправлен критичный баг, и также сделано много другой работы. Пусть этим коммитом в нашем случае будет коммит, обозначенный буквой D. И вы не хотите вливать эту ветку целиком, не хотите, чтобы коммит C тоже попал, хотите перенести только изменения из коммита D. То есть механизм слияния — git merge — вам не подойдёт. И перебазировать при помощи rebase тоже не хотите, поскольку хотите сохранить все ветки без изменений. Вы хотите только перенести изменения из этого коммита, — коммита D, — в ветку master без предыдущей истории. Это как раз делается при помощи операции cherry-pick:

```
git cherry-pick D
```

Ветка, в которую вы переносите нужный вам коммит, будет содержать все изменения из коммита D, но не будет содержать изменения из коммита C:




# Что можно почитать ещё?

1. Полная документация по git reset — https://git-scm.com/docs/git-reset
2. Полная документация по git stash — https://git-scm.com/docs/git-stash
3. Ещё одно понятное объяснение git rebase — https://www.atlassian.com/ru/git/tutorials/rewriting-history/git-rebase
4. Ещё одно понятное объяснение git cherry-pick — https://www.atlassian.com/git/tutorials/cherry-pick
5. Полная документация по git rebase — https://git-scm.com/docs/git-rebase
6. Полная документация по git cherry-pick — https://git-scm.com/docs/git-cherry-pick