

История Git

Git был создан в 2005 году Линусом Торвальдсом, создателем Linux, как распределенная система контроля версий. Основной целью разработки было обеспечить надежный и быстрый инструмент для управления версиями, который бы позволял нескольким разработчикам эффективно работать над одним проектом, особенно учитывая масштабное сообщество Linux.

До появления Git, разработчики Linux использовали проприетарную систему BitKeeper, но в 2005 году права на её бесплатное использование были отозваны, что поставило сообщество перед необходимостью поиска замены. Тогда Торвальдс решил создать новую систему, которая бы удовлетворяла нескольким ключевым требованиям: скоростью, эффективностью для больших проектов, безопасностью данных, а также полной распределенностью, позволяющей каждому разработчику работать с полной копией репозитория, включая его историю. Вскоре к Линусу присоединился Джунио Хамано, который в дальнейшем стал основным разработчиком и мейнтейнером Git.

Git сразу отличался несколькими важными особенностями:

- Распределённостью — у каждого разработчика хранится полная копия всей

истории проекта, что

минимизирует зависимость от центрального сервера.

- Эффективным слиянием — Git поддерживает ветвления и слияния, что упрощает параллельную работу и

разработку новых функций без риска конфликтов.

- Высокой скоростью — операции выполняются быстро даже на крупных проектах благодаря алгоритму,

разработанному Торвальдсом.

На протяжении нескольких лет Git стал стандартом для систем контроля версий, особенно в условиях

работы распределенных команд. Появление GitHub в 2008 году, который предоставил удобный

веб-интерфейс для Git и позволил разработчикам делиться проектами публично, сыграло огромную роль

в популяризации Git. Сегодня Git используется миллионами разработчиков по всему миру для управления

исходным кодом и совместной разработки, став одной из основ современного программирования.

GIT

Установка и конфигурация

Установка

**[https://git-scm.com/book/ru/v2/Введение-
Установка-Git](https://git-scm.com/book/ru/v2/Введение-Установка-Git)**

Конфигурация I

- Если Mac или Linux: **откройте терминал**
- Если Windows: **откройте командную строку**

`git help` - помощь, документация

`git help название_команды` - документация конкретной команды

Конфигурация II

git config --global user.name “Имя Фамилия”

git config --global user.email “Ваш email”

git config --global color.ui true

Конфигурация III

Создание проекта

mkdir название_проекта

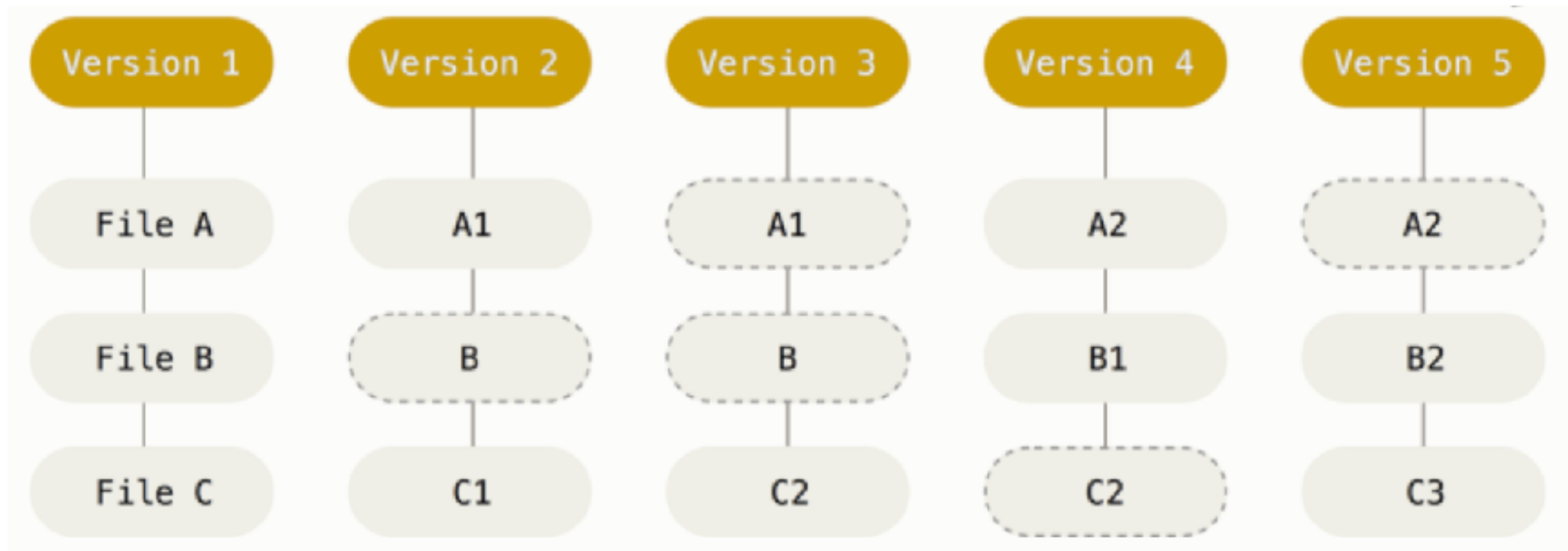
cd название проекта

git init

GIT

ОСНОВЫ

Как GIT хранит изменения?



В виде снимков проекта во времени

Базовая работа с GIT (создание файла)

Вы создали файл

Статус “**untracked**”

git add

Статус “**staged**”
(подготовленный)

git commit

Статус “**committed**”
(зафиксированный)

Базовая работа с GIT (изменение файла)

Вы изменили файл

Статус **“modified”**

git add

Статус **“staged”**
(подготовленный)

git commit

Статус **“committed”**
(зафиксированный)

Базовая работа с GIT

12:45

Создал файл Test.java



Подготавливаем файл к commit'у - **git add Test.java**



Делаем commit (фиксируем, “делаем снимок”) - **git commit**

13:00

Изменил файл Test.java, добавил файл Hello.java



Подготавливаем файлы к commit'у - **git add Test.java Hello.java**
(есть другие варианты, например **git add *.java**)



Делаем commit (фиксируем, “делаем снимок”) - **git commit**

Команды

git status - узнать текущий статус репозитория

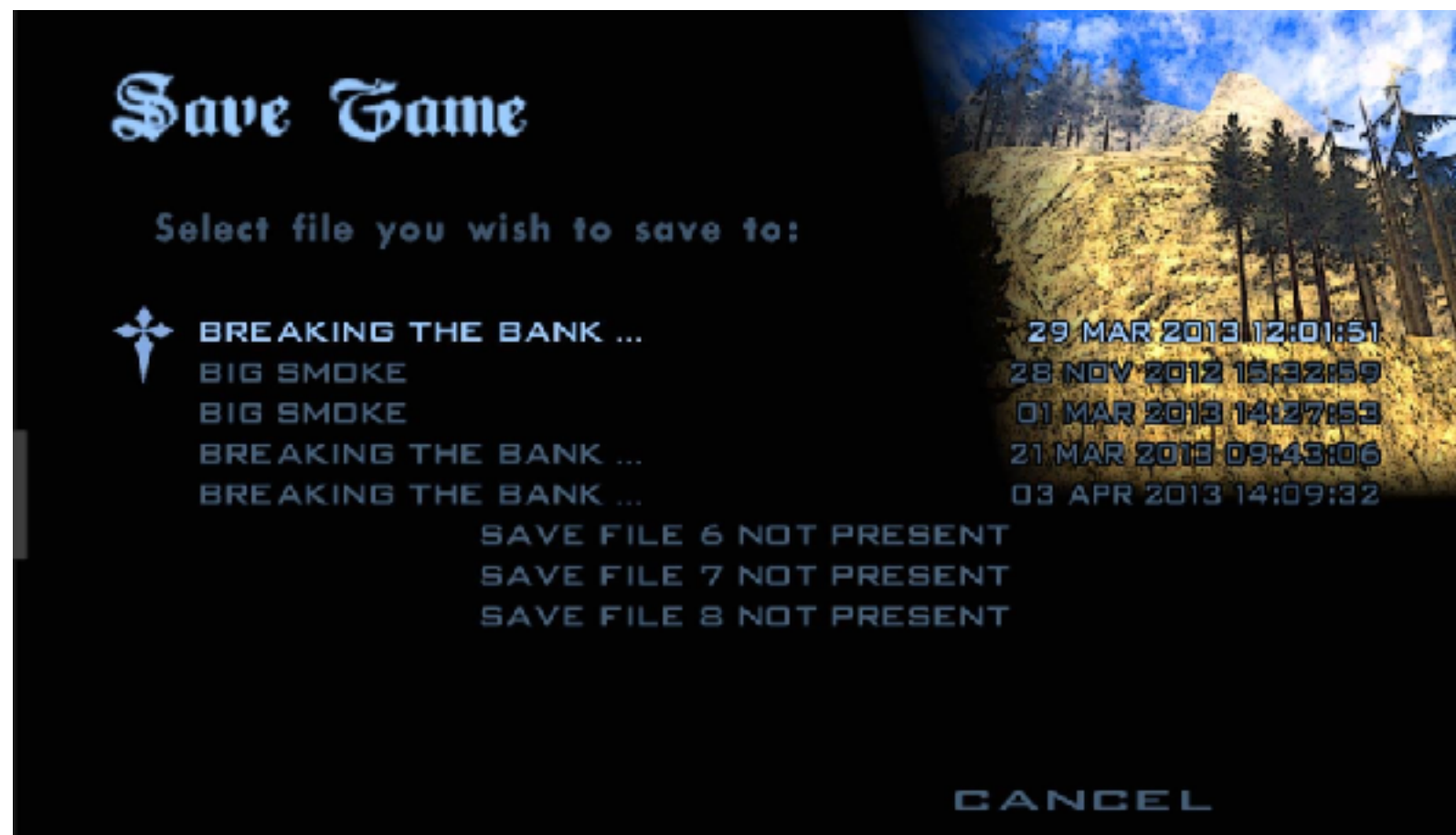
git add - подготовить файл(-ы) к commit'у

git commit -m “Сообщение снимка” - сделать commit

Метафора для commit'a

“Слепок” репозитория

“Снимок” репозитория



Разные способы сделать git add

`git add <список файлов>` - `git add file1 file2`

`git add .` - добавить все файлы в текущей папке

`git add *.java` - добавить все файлы в текущей папке с расширением .java

`git add someDir/*.java` - добавить все файлы в папке someDir с расширением .java

`git add someDir/` - добавить все файлы в папке someDir

`git add "*.java"` - добавить все файлы в проекте с расширением .java

git log



Позволяет посмотреть всю историю commit'ов

GIT

git reset

git reset

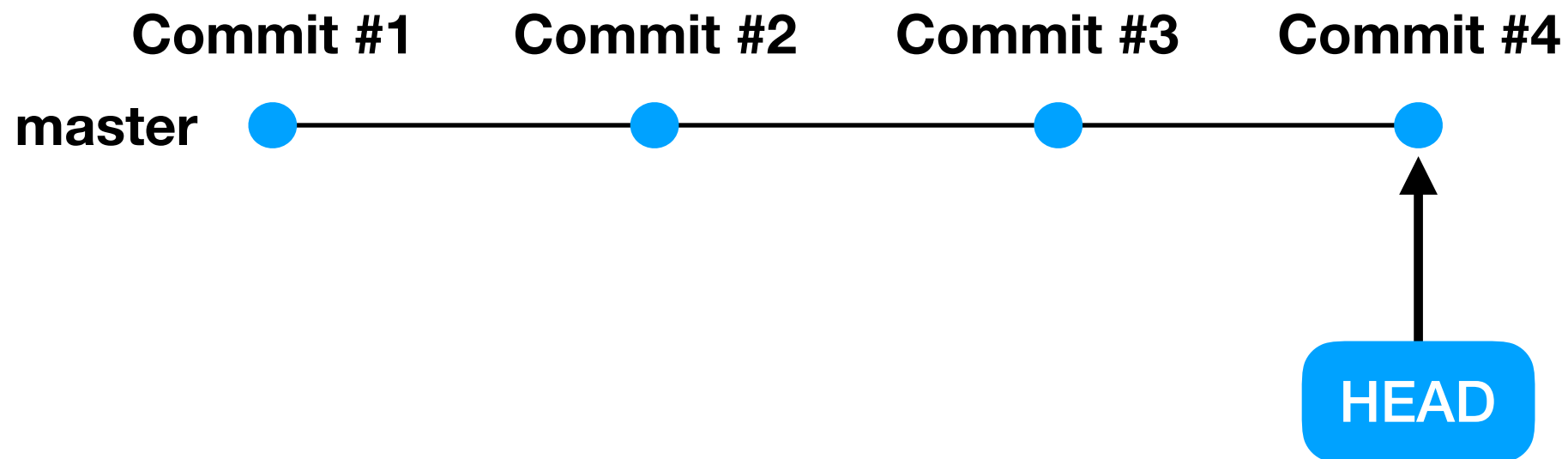
! Опасная команда, может быть использована для переписывания истории

reset (рус. сброс)

Предназначена для отмены каких - либо изменений в проекте, откату проекта к какому - то снимку.

**Команда очень мощная и многофункциональная
(вызывается с разными параметрами)**

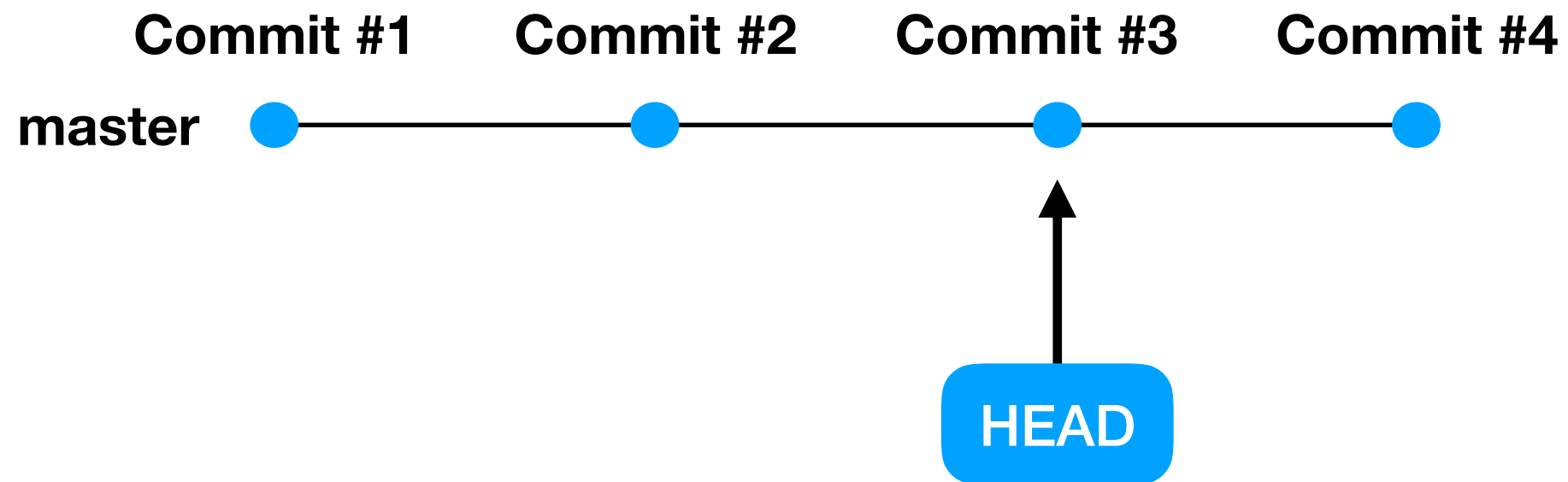
Указатель HEAD



**Обычно указывает на последний (текущий)
КОММИТ**

Указатель HEAD

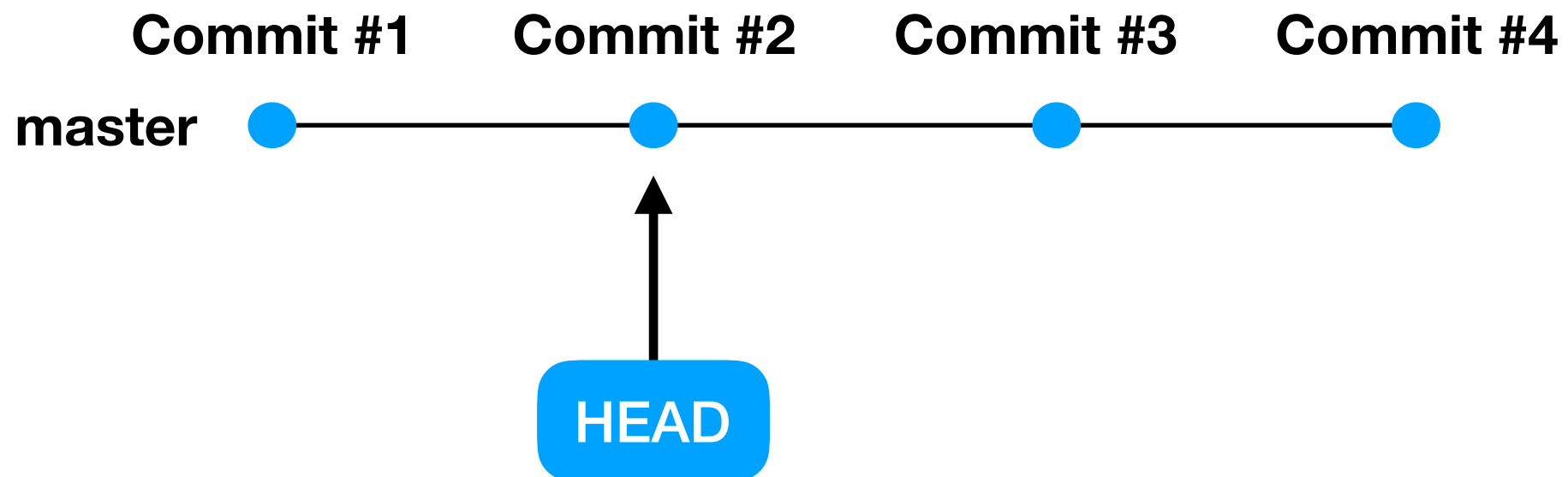
HEAD^



HEAD можно смещать

Указатель HEAD

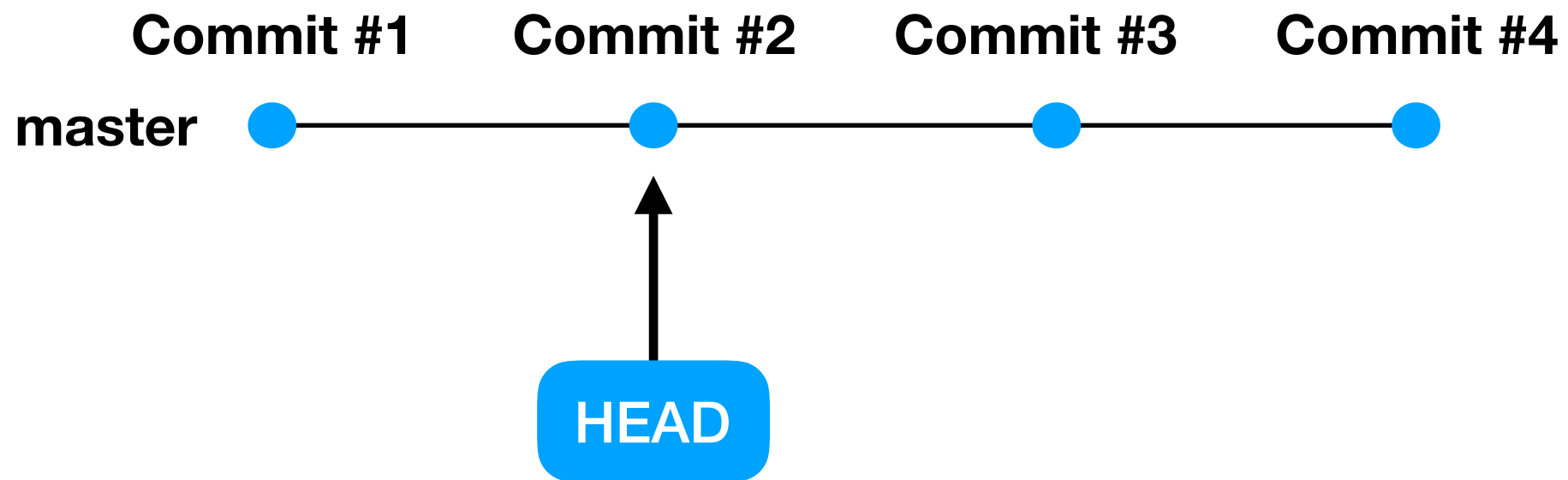
HEAD^^



HEAD можно смещать

Указатель HEAD

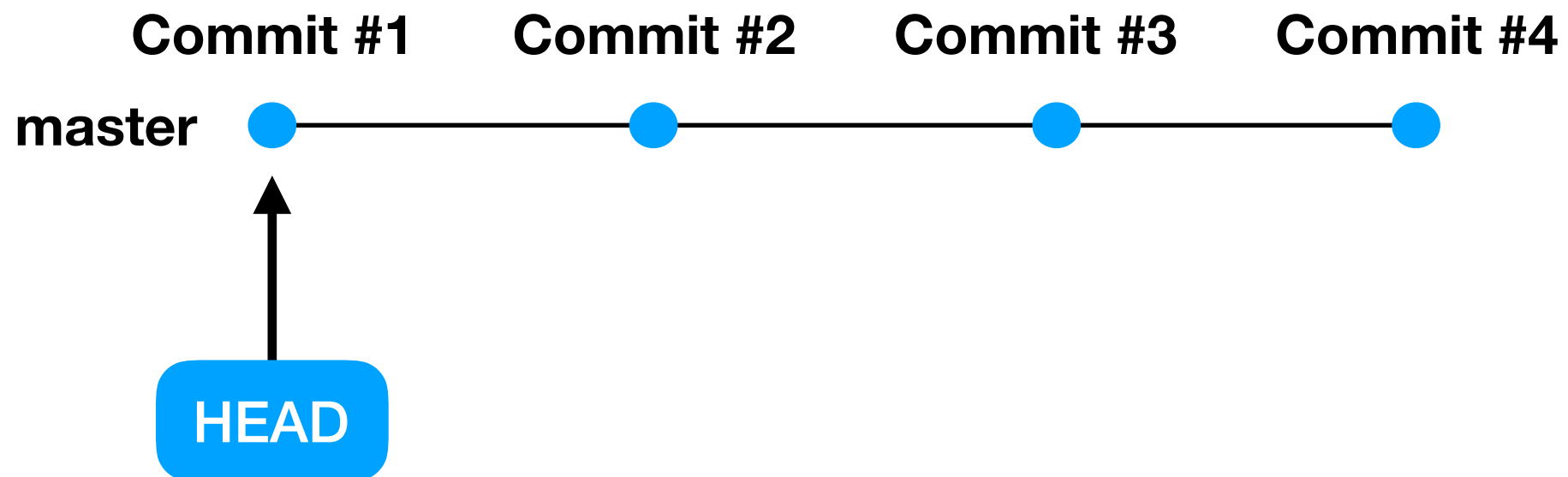
HEAD~2



HEAD можно смещать

Указатель HEAD

HEAD~3



HEAD можно смещать

git reset

Имеет 3 режима (в зависимости от радикальности отката к указанному коммиту):

- **--soft**
- **--mixed**
- **--hard**

git reset (синтаксис вызова)

Режим **Желаемый коммит**

git reset [--soft | --mixed | --hard] [commit]

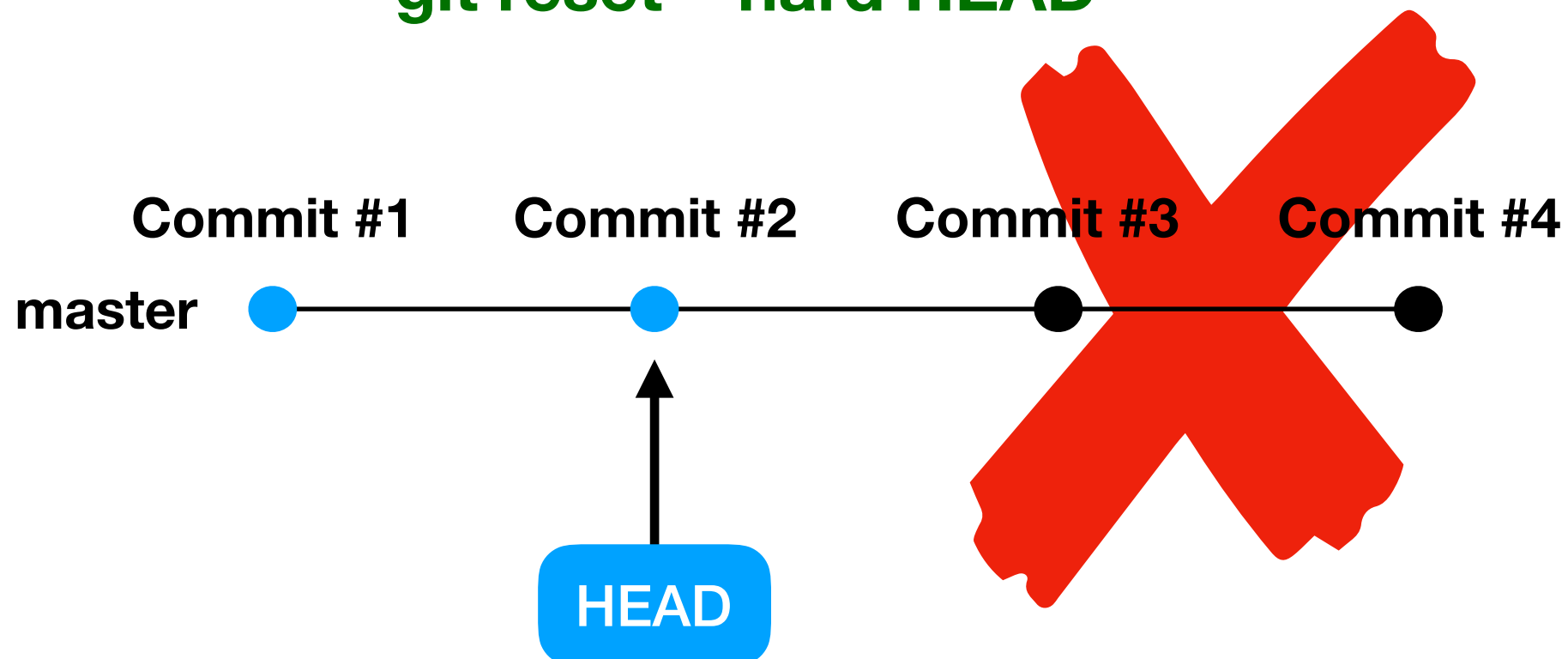
↑
Может быть хэш
коммита (уникальный
идентификатор) или
различные вариации с
HEAD

git reset --hard

Возвращает проект к указанному коммиту, при этом полностью удаляет все коммиты после указанного.

Самая "сильная" вариация git reset. Удаляет коммиты безвозвратно!

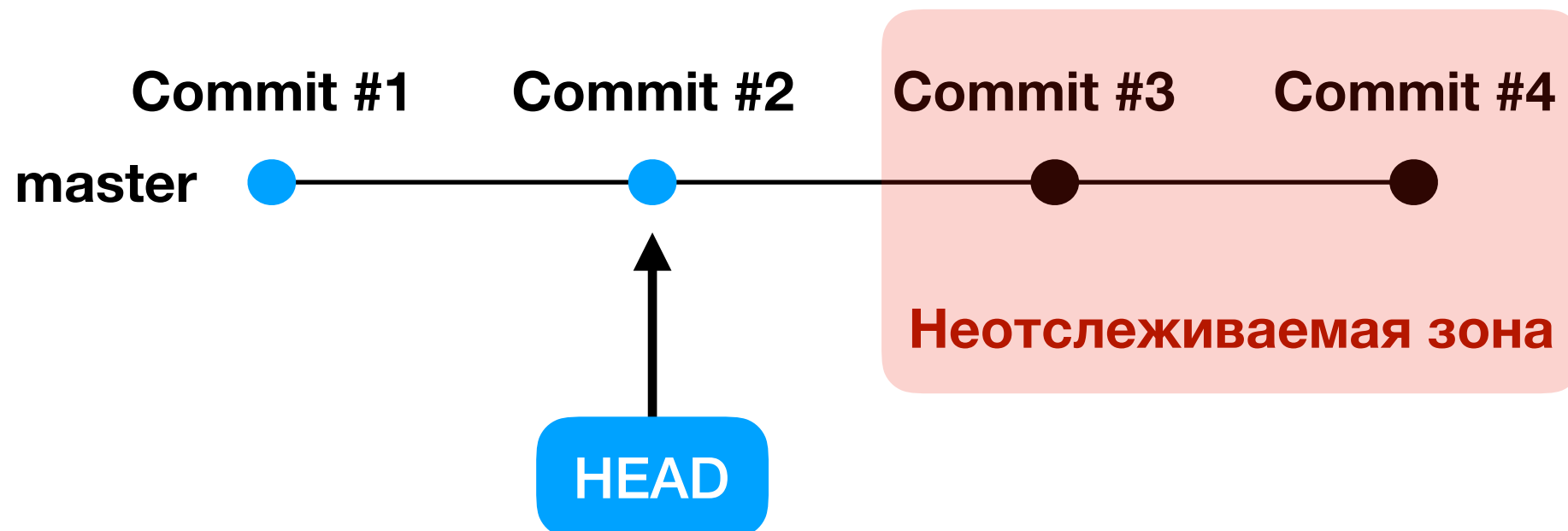
git reset --hard HEAD^^



git reset --mixed

Возвращает проект к указанному коммиту, при этом переводит все коммиты после указанного в **неотслеживаемую (unstaged)** зону.

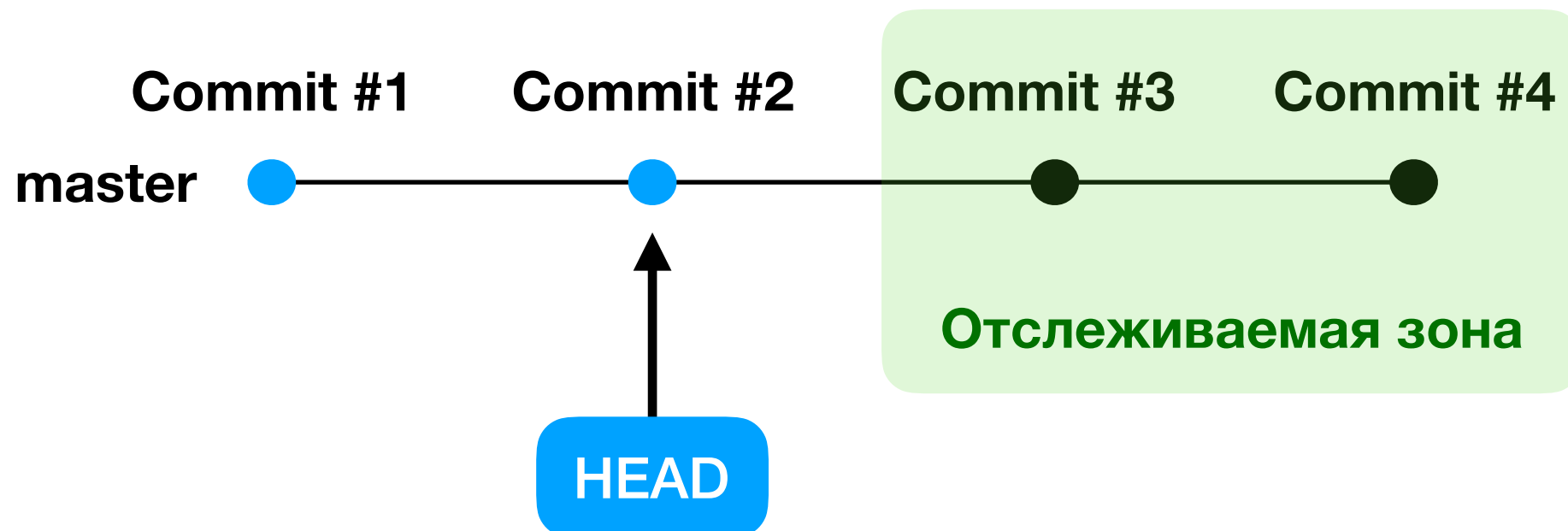
git reset --mixed HEAD^^



git reset --soft

Возвращает проект к указанному коммиту, при этом переводит все коммиты после указанного в **отслеживаемую (staged)** зону.

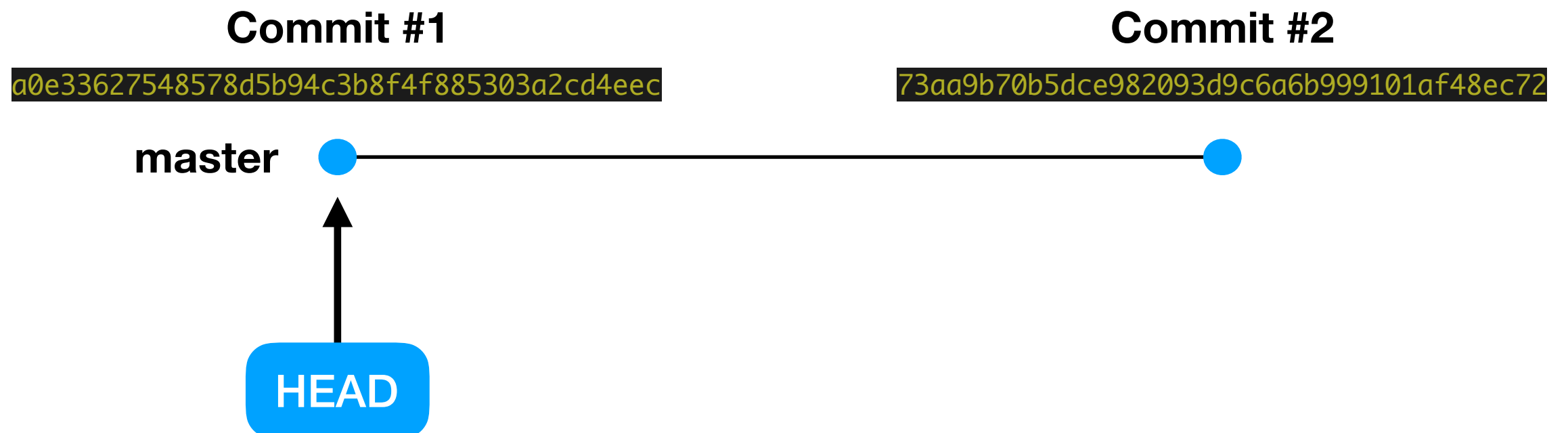
git reset --soft HEAD^^



Можно и по хэшу коммита

Возвращает проект к указанному коммиту

```
git reset [--soft | --mixed | --hard] a0e33627548578d5b94c3b8f4f885303a2cd4eec
```



git reset по-умолчанию

- При вызове без указания параметра [--soft | --mixed | --hard] по-умолчанию используется --mixed.

git reset HEAD^^ = git reset --mixed HEAD^^

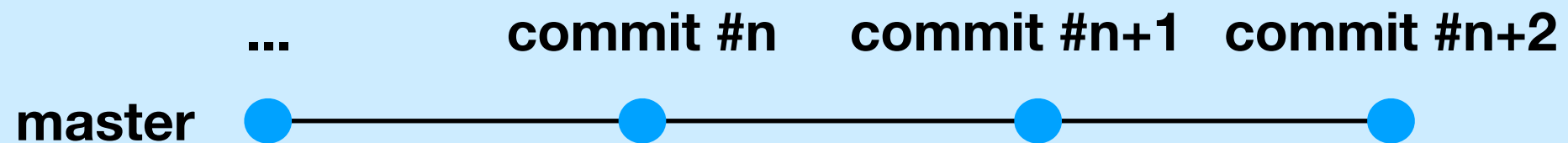
- При вызове без указания желаемого коммита по-умолчанию используется HEAD.

git reset = git reset --mixed HEAD

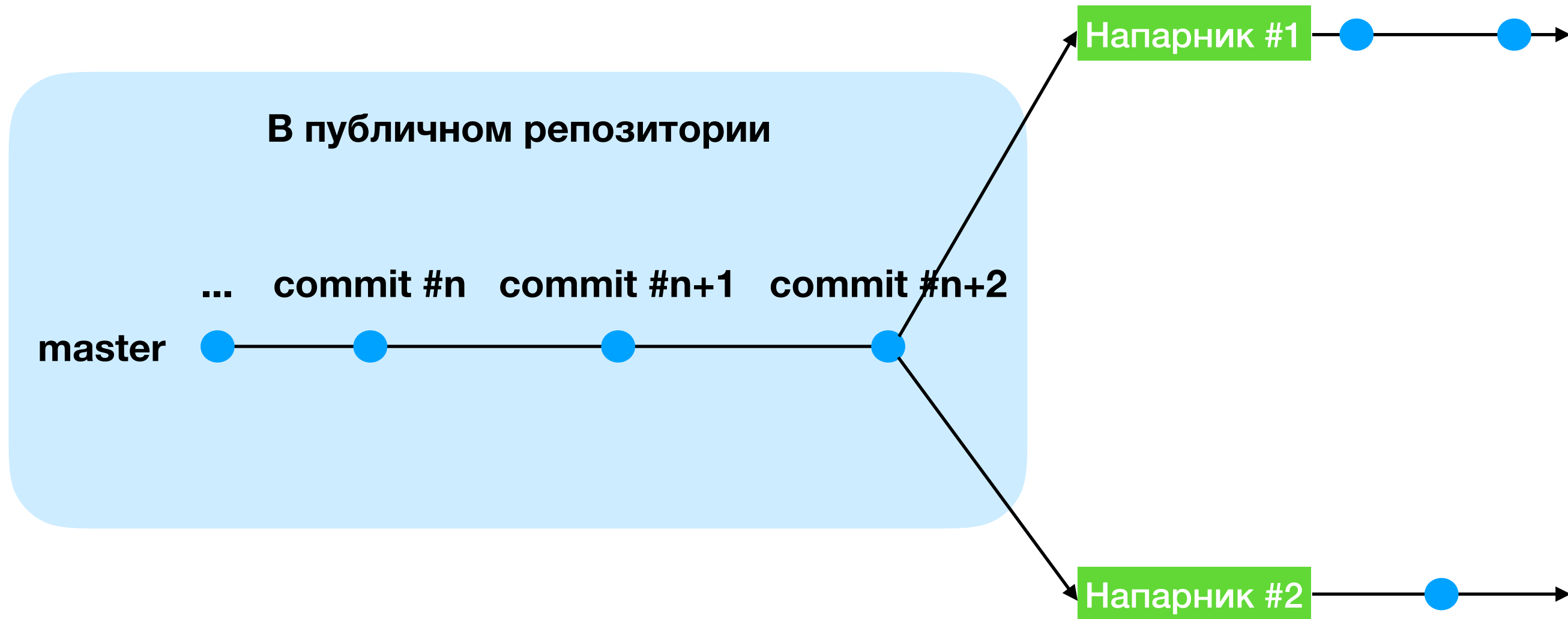
Опасность git reset

Никогда не делайте `git reset <commit #n>` после того, как вы опубликовали какие-либо КОММИТЫ после `<commit #n>`

В публичном репозитории

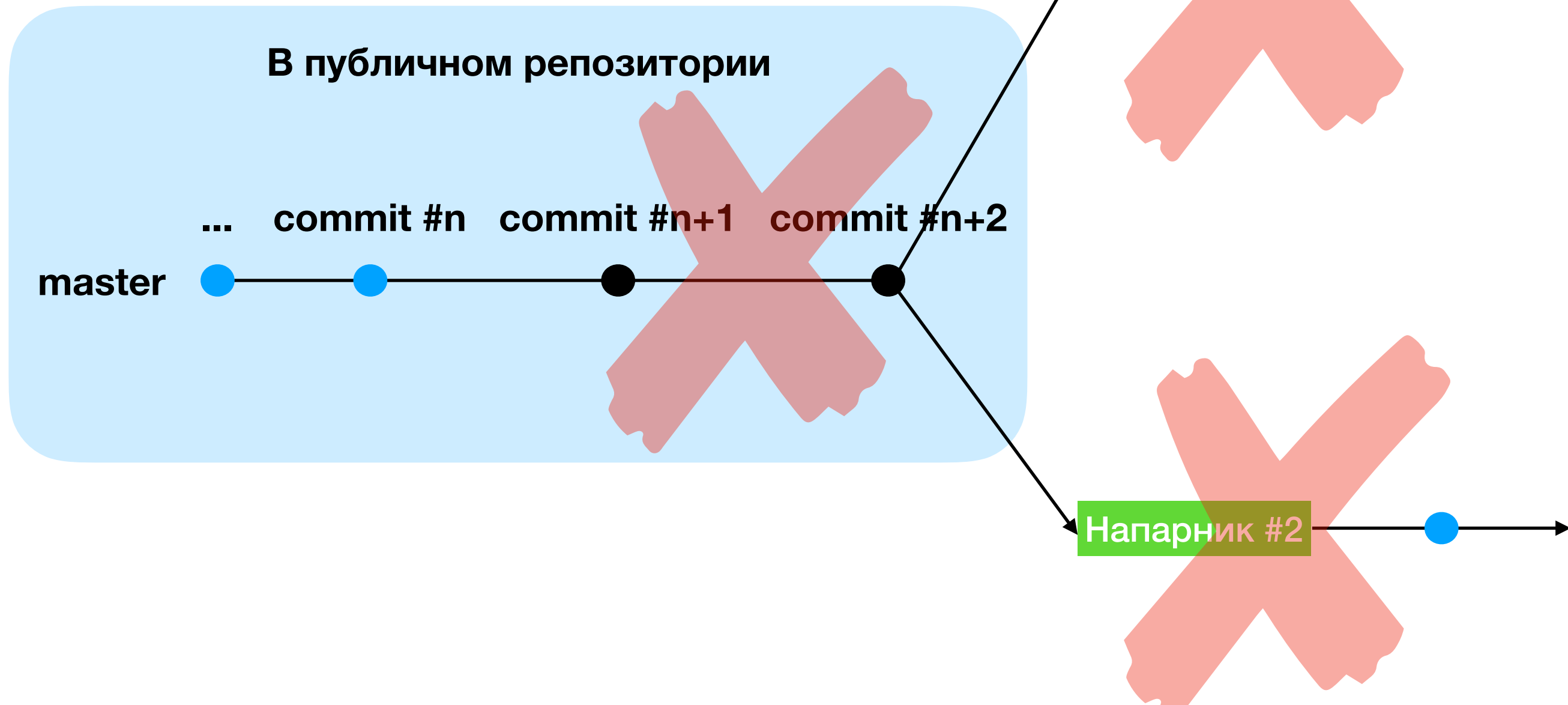


Опасность git reset



Опасность git reset

`git reset --hard <commit #n>`



GIT

git checkout (часть 1)

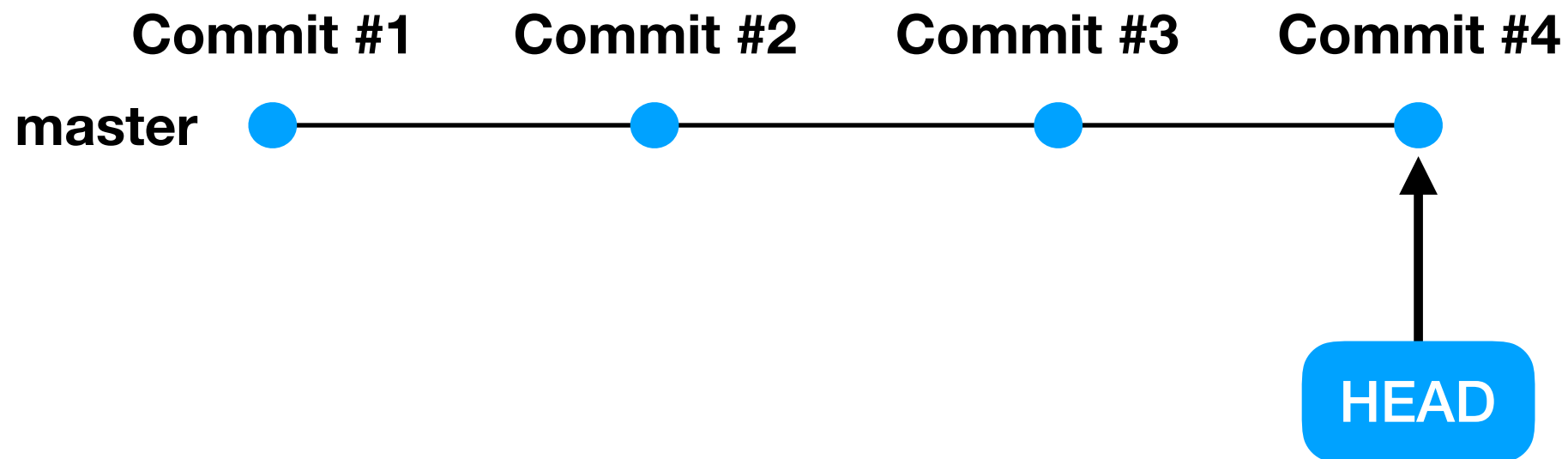
git checkout

(часть функционала)

- **Команда очень мощная и многофункциональная (как и git reset)**
- **Используется для перемещения между коммитами, версиями отдельных файлов и ветками**

git checkout

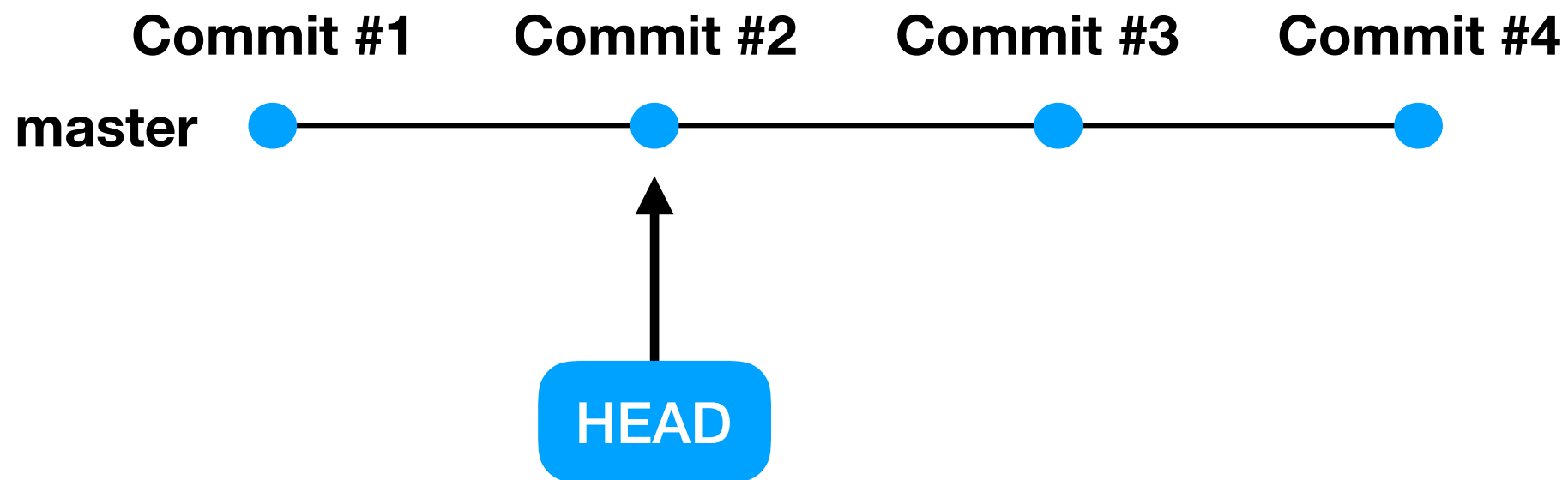
(перемещение между коммитами)



Хотим посмотреть, как выглядел проект в каком-то снимке в прошлом

git checkout

(перемещение между коммитами)



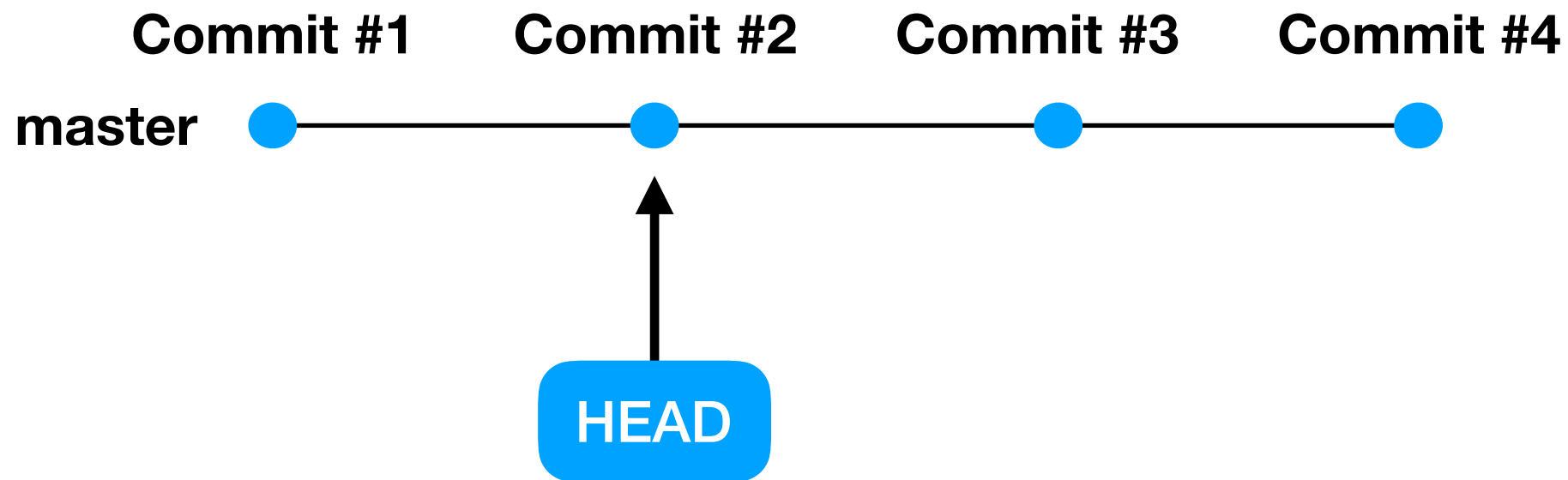
git checkout <хэш commit #2>

git checkout HEAD^^

git checkout HEAD~2

git checkout

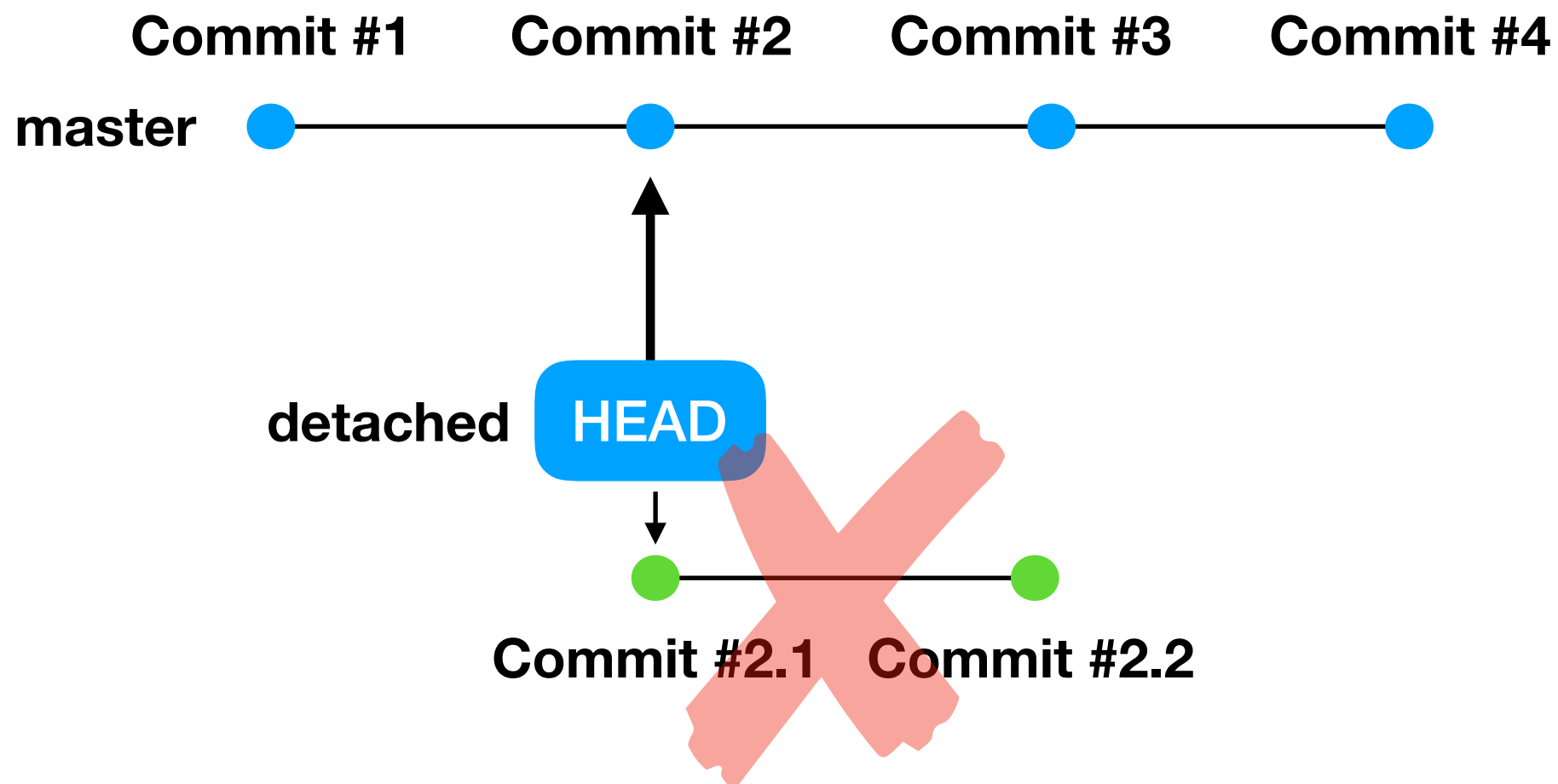
(перемещение между коммитами)



- Состояние проекта полностью вернулось к указанному снимку. При этом никакие коммиты не удалились. Мы в любой момент можем перенестись обратно в актуальную версию.
- Указатель HEAD находится в состоянии DETACHED (рус. отделенный). Он отделен от актуальной версии проекта. Любые изменения или коммиты сделанные в этом состоянии удаляются сборщиком мусора при переходе к другому коммиту.

git checkout

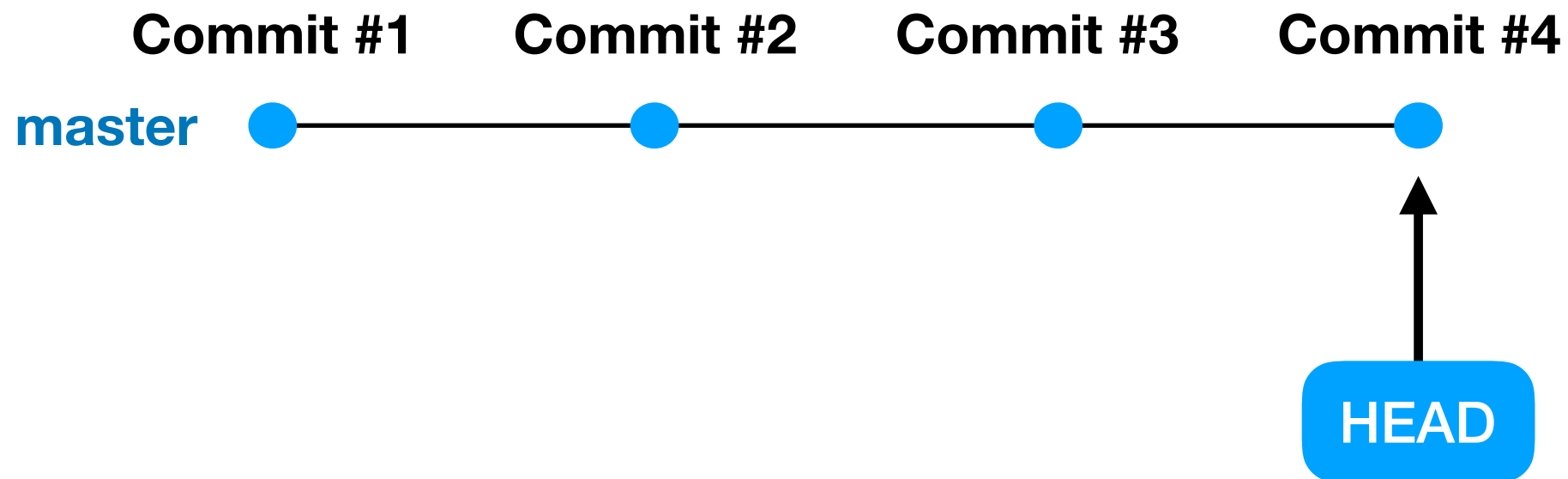
(перемещение между коммитами)



- **Указатель HEAD находится в состоянии DETACHED (рус. отделенный). Он отделен от актуальной версии проекта. Любые изменения или коммиты сделанные в этом состоянии удаляются сборщиком мусора при переходе к другому коммиту.**

git checkout

(перемещение между коммитами)



- **git checkout master** - переход обратно к актуальному коммиту
- **master** - название текущей ветки. О ветвлении позже.

git checkout

(перемещение между версиями файлов)

**Хотим вернуть конкретный файл(-ы) к какой-то
версии в прошлом**

git checkout

(перемещение между версиями файлов)

- `git checkout <указатель коммита> -- путь_до_файла_1 путь_до_файла_2`

Пример: `git checkout a0e33627548578d5b94c3b8f4f885303a2cd4eес -- file1 file2`

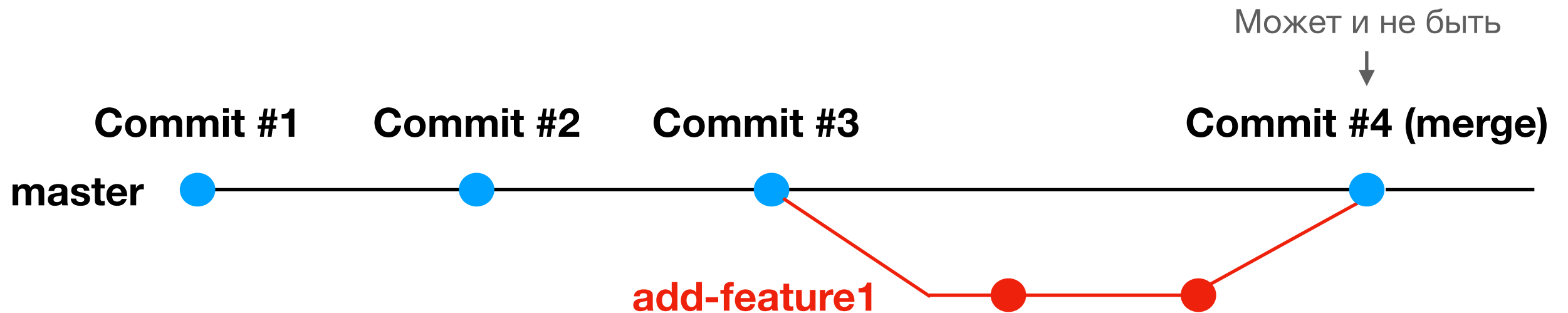
Возвращает два файла **file1** и **file2** к версии, которая была у
НИХ в **указанном** КОММИТЕ

GIT

Ветвление (Branching)

Часть 1

Зачем использовать ветвление?



- Новые функции разрабатываются в отдельных ветках
- Ветка **master** содержит стабильную версию проекта. Можем вернуться на **master** в любой момент.
- Сразу несколько разработчиков могут работать в своих ветках над своими задачами. После завершения работы над задачами, эти ветки "сливаются" в **master** ветку.

git branch название_ветки

Команда для создания новой ветки



git branch some-feature

*вызвали команду, находясь на
Commit #3
новая ветка "отпочковывается"
именно с этого коммита

git branch

Команда для просмотра, на какой ветке мы сейчас находимся

```
[test_project]$ git branch  
* master
```

***На какой ветке
находится указатель
HEAD**

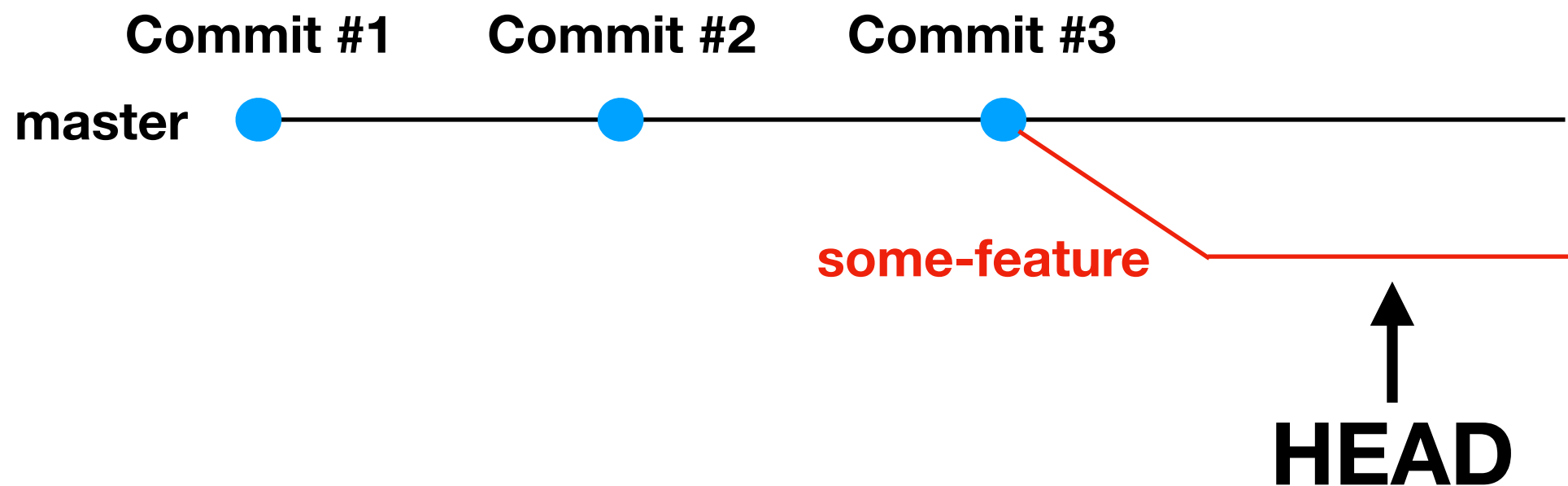
git branch -d название_ветки

Команда для удаления ветки

```
test_project$ git branch -d some-feature  
Deleted branch some-feature (was a0f92b5).
```

Переключение между ветками

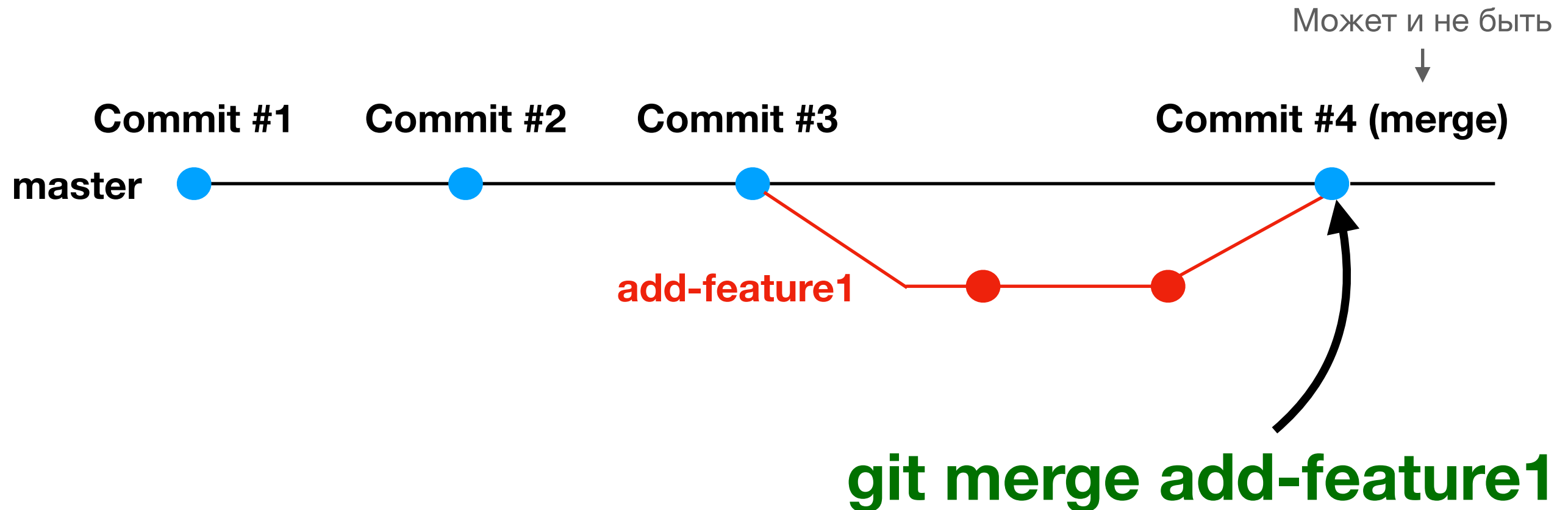
git checkout название_ветки



git checkout some-feature

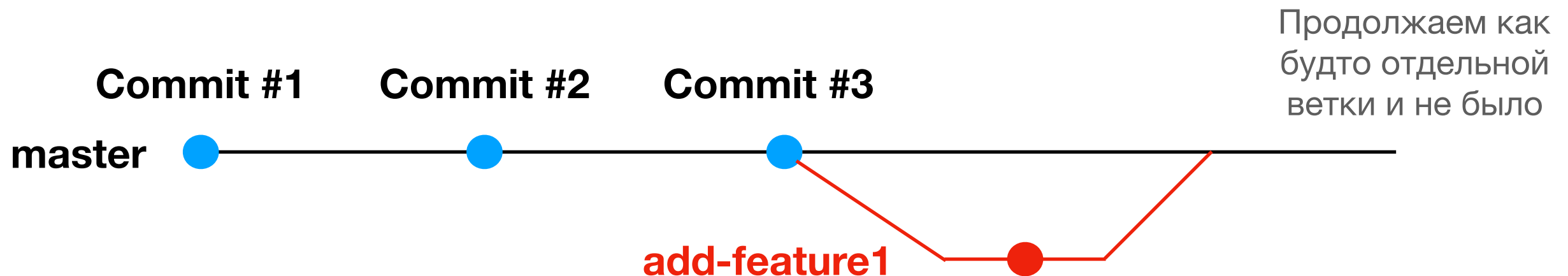
git merge

Сливает одну ветку с другой



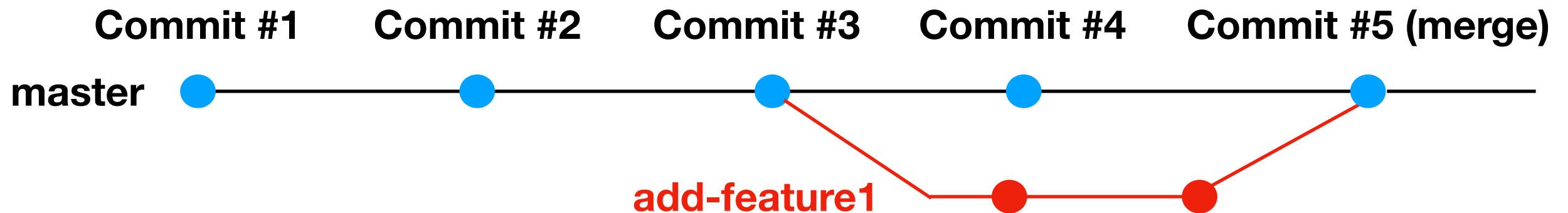
Слили ветку **add-feature1** в ветку master

Fast - Forward merge



- Пока мы работали в **своей** ветке, в ветке master ничего не произошло (не было новых коммитов)
- GIT'у очень легко слить ветку **add-feature1** в master (не может возникнуть конфликтов)
- Не создается отдельный commit для слияния (merge commit)

~~Fast - Forward merge~~



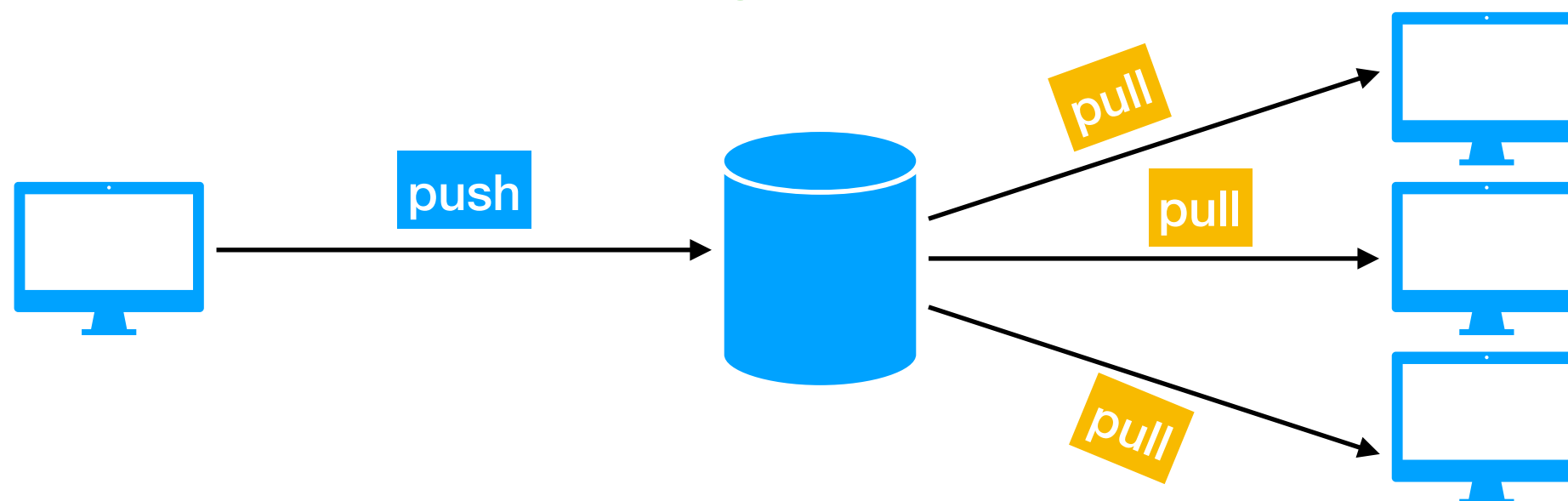
- Пока вы работали в своей ветке, кто-то добавил коммиты в ветку master
- Или вы сами добавили новые коммиты в ветку master (пример - вас попросили исправить какой-нибудь критический баг и запустить на GitHub)
- Могут возникнуть конфликты. Гит попыбует решить их самостоятельно. Если у него не получится, придется решать их вручную.
- Merge commit создается.

GIT

Работа с удаленным репозиторием.
Введение.

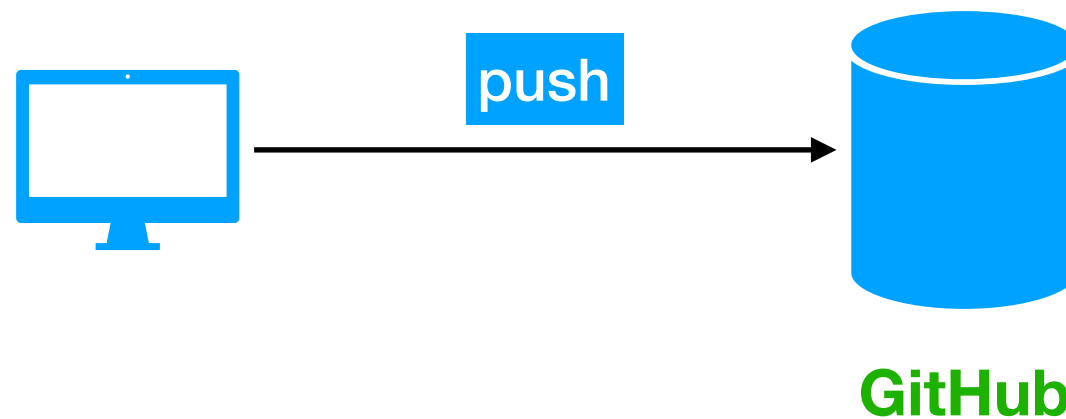
Работа с удаленным репозиторием

- До сих пор вся наша работа была сохранена только локально на нашем компьютере
- С помощью GIT можно отправить нашу работу на удаленный репозиторий - **для дополнительной сохранности** и для того, чтобы **другие люди могли видеть наши КОММИТЫ**



Удаленный репозиторий

- **GitHub**
- **BitBucket**
- **GitLab**



Предоставляет нам всю инфраструктуру для хранения и управления GIT - репозиториев

git remote

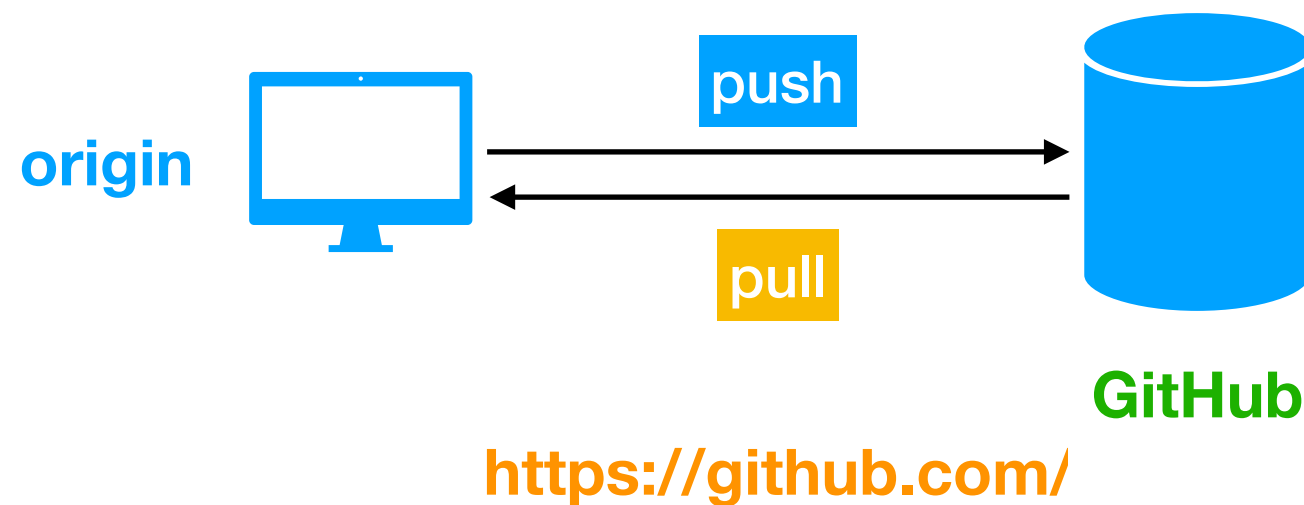
Команда для настройки и просмотра удаленных репозиториев

- **git remote -v** - просмотр списка существующих удаленных репозиториев
- **git remote add НАЗВАНИЕ_РЕПОЗИТОРИЯ АДРЕС_РЕПОЗИТОРИЯ**
- добавить новый удаленный репозиторий, который находится по **указанному** адресу. При этом, на нашем компьютере к удаленному репозиторию мы будем обращаться по его **названию**
- **git remote remove НАЗВАНИЕ_РЕПОЗИТОРИЯ** - удалить репозиторий с указанным **названием**

Добавление удаленного репозитория

Пример:

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



На нашем компьютере хранится только ссылка на удаленный репозиторий
origin - название этой ссылки

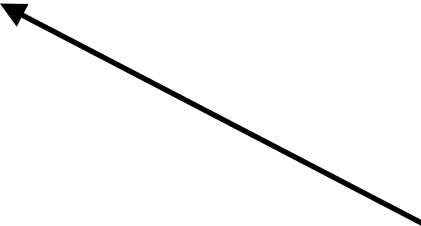
git push

Команда для отправки локального репозитория на удаленный

git push НАЗВАНИЕ_УДАЛЕННОГО_РЕПОЗИТОРИЯ ВЕТКА

Пример:

git push origin master

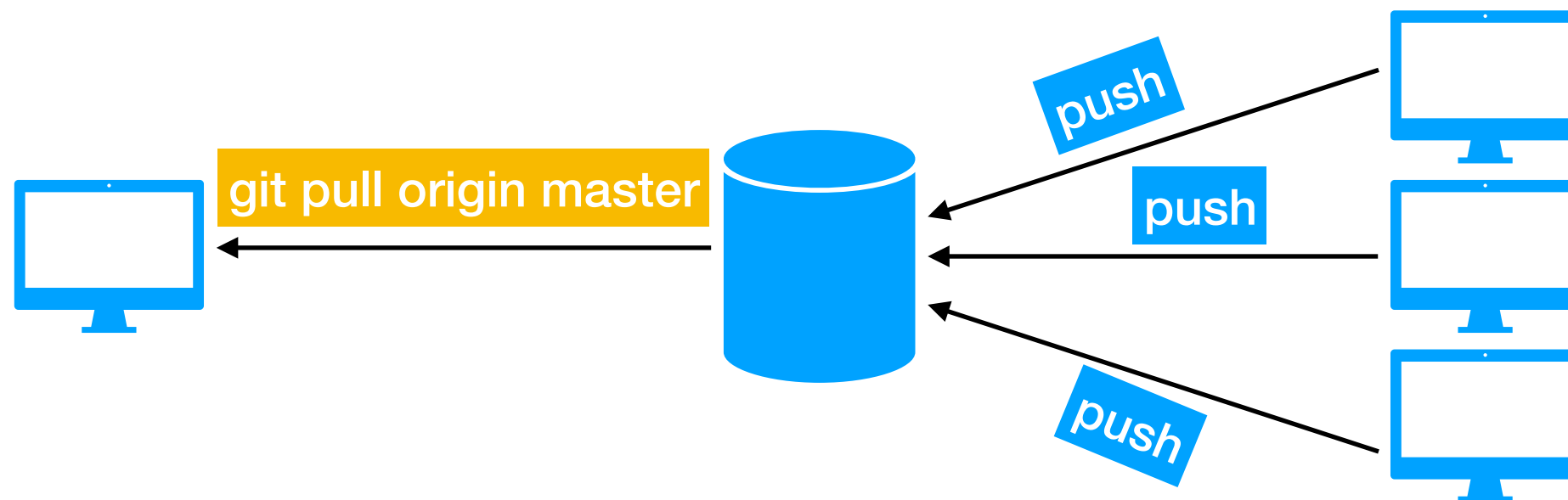


Отправляем на удаленный
репозиторий с именем **origin**
нашу ветку **master**

Все! Наша ветка **master** теперь скопирована на удаленный репозиторий

git pull

Команда для получения обновлений с удаленного репозитория

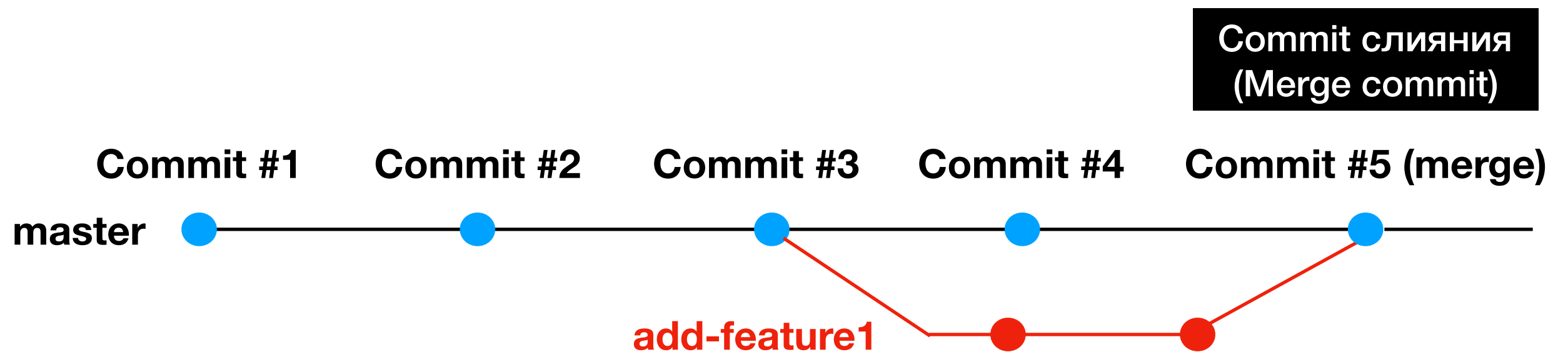
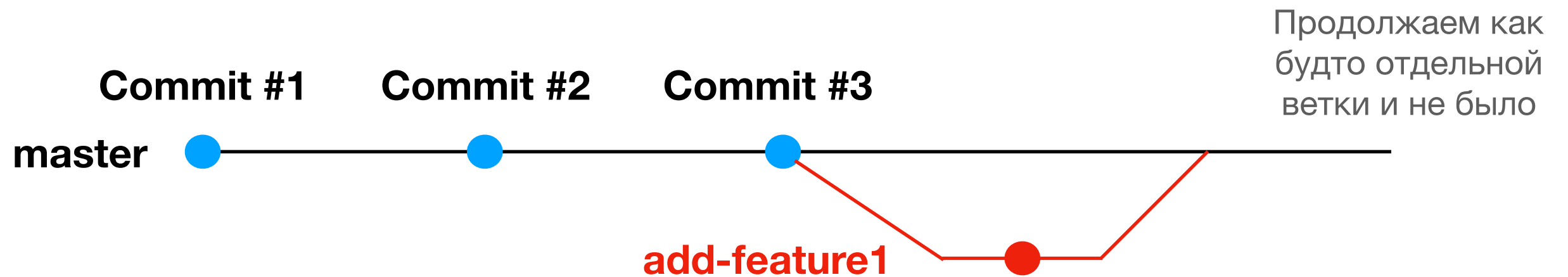


Все! Теперь наш локальный репозиторий синхронизирован с удаленным


GIT


Rebase (Перебазирование)

Мотивация



Мотивация


 master Merge commit branch 'branch5'

 branch5 Merge commit 'branch4' into branch5

Merge commit 'branch5' into master


Merge branch 'branch4' into branch5

Merge branch 'branch6'

 branch6 Some more work.

Merge branch 'branch5'

Sesame snaps toffee caramels.

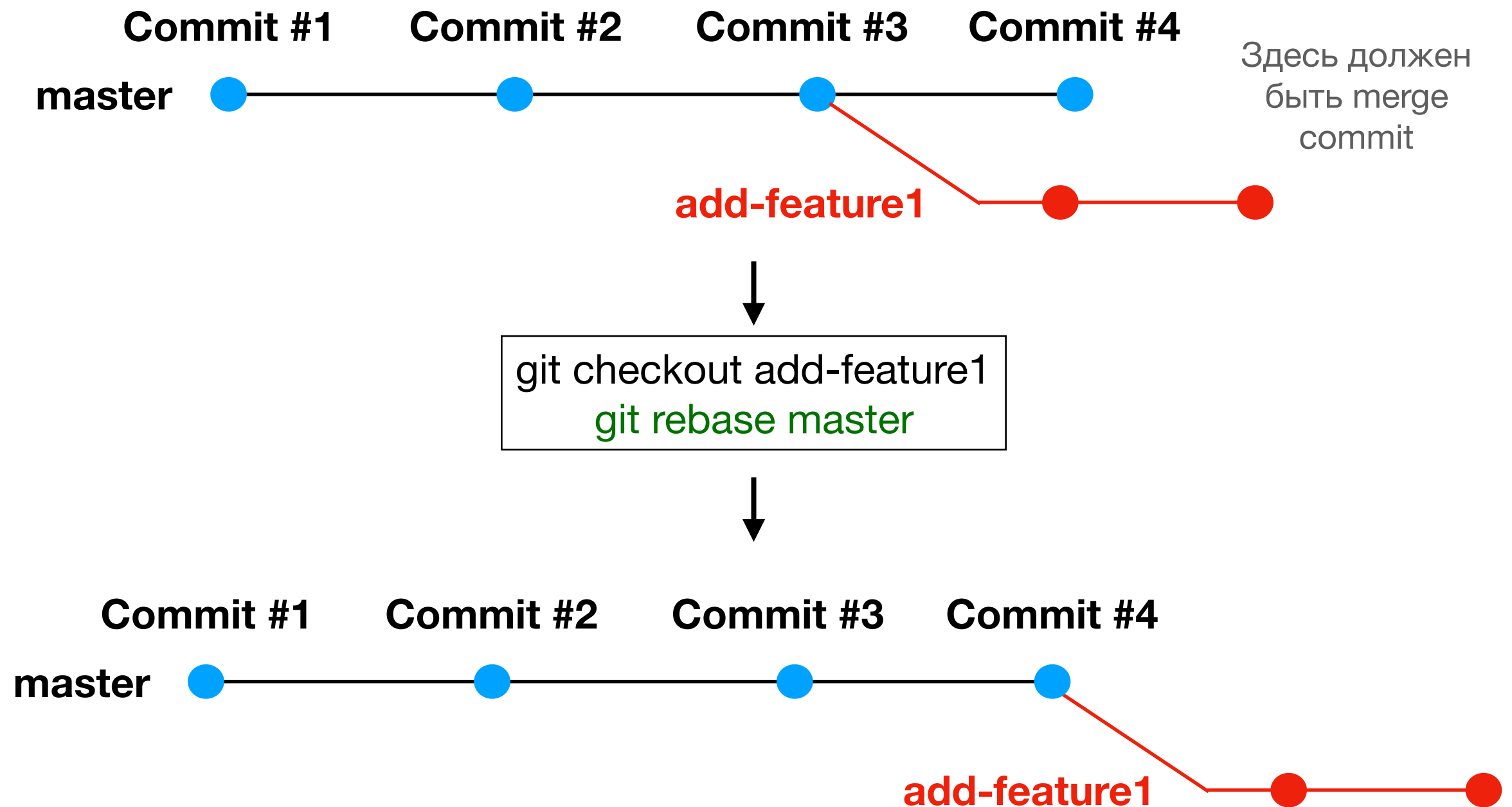
 branch3 Soufflé dessert lemon drops tart.

Sugar plum dessert marzipan.

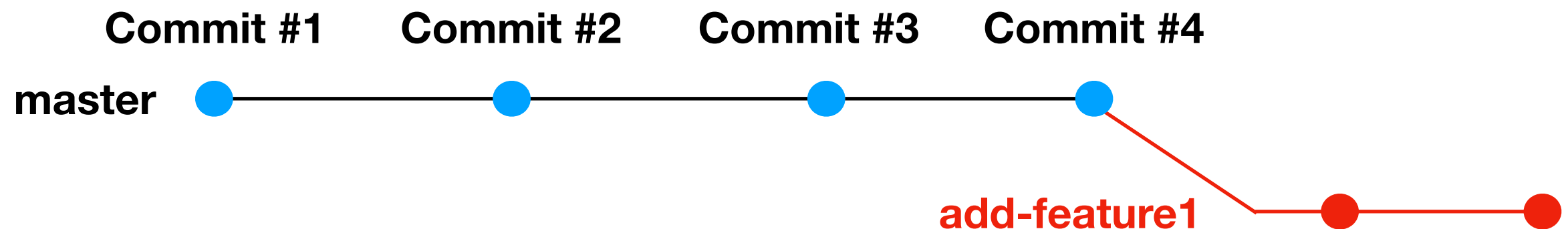
rebase - альтернатива **merge**

- Обе команды делают одно и то же - сливают ветки
- Команда merge может создавать merge commit при слиянии (в случае не fast-forward), команда rebase merge commit'а не создает (дальше увидим как это работает)
- Команда merge безопасней, чем rebase - есть отдельный commit, отображающий слияние.
- Плюс merge - достоверная полная история commit'ов
- Плюс rebase - лаконичная линейная история без лишних коммитов
- Если в ветке долго велась работа и произошло много изменений лучше использовать merge
- Если ветка была недолгая и произошло мало изменений - можно использовать rebase
- Используйте merge, если вас не просят о rebase.

Как работает rebase?

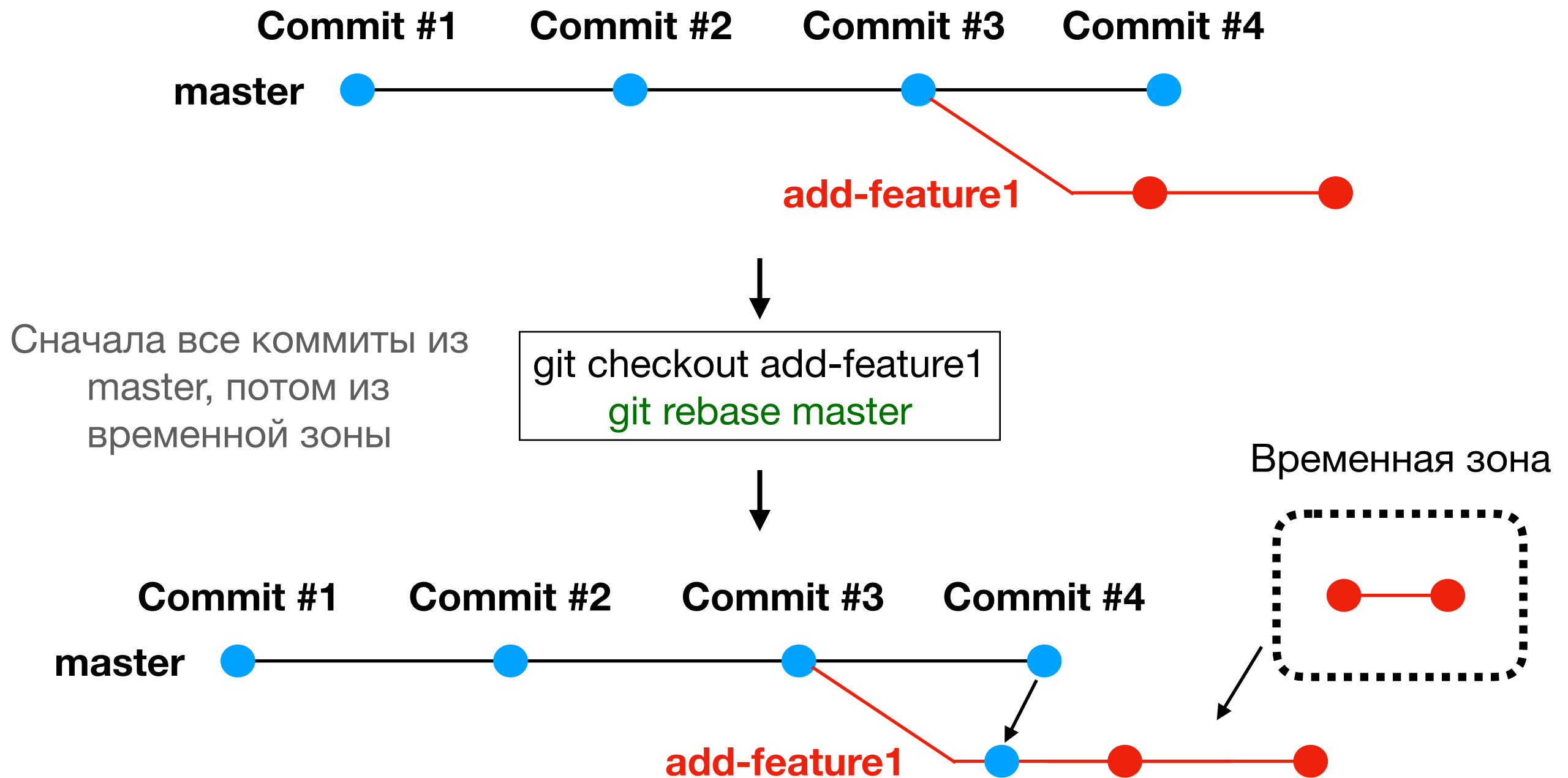


Что произошло?

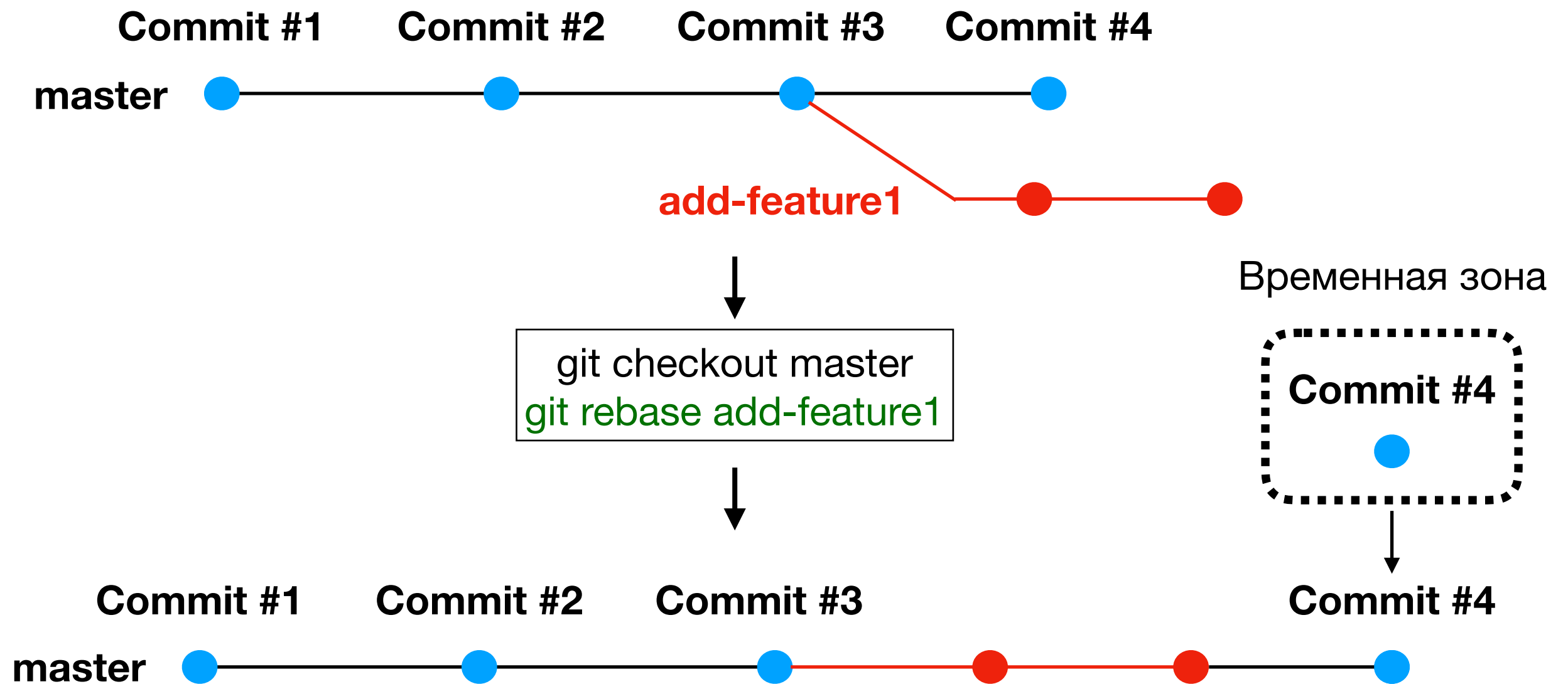


- Ветка **add-feature1** "перебазировалась"
- В ветку **add-feature1** был добавлен Commit #4 из master. Затем, поверх него были добавлены коммиты из **add-feature1**. Теперь можно делать fast-forward слияние без merge commit'a

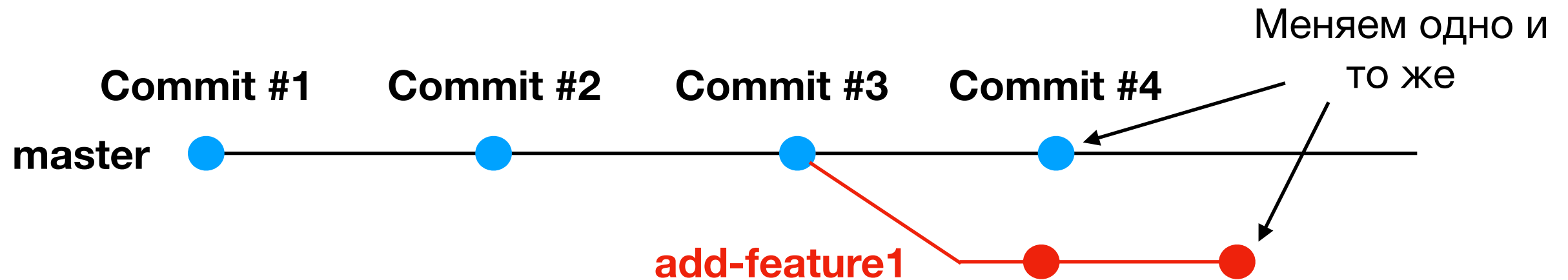
Временная зона



Можно ли наоборот?



Конфликты слияния при rebase



Разрешение конфликта такое же, как в случае с merge.

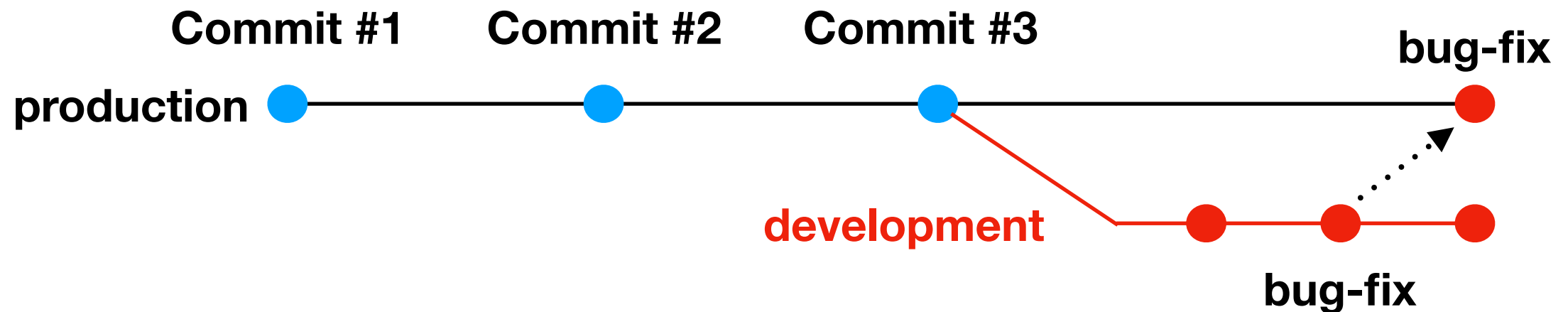
`git rebase --continue`
`git rebase --abort`

GIT

cherry-pick

git cherry-pick

Используется тогда, когда нам надо "взять" один или несколько коммитов из другой ветки в нашу ветку



Нам нужен единственный коммит **[bug-fix]** из ветки **development**

Метафора



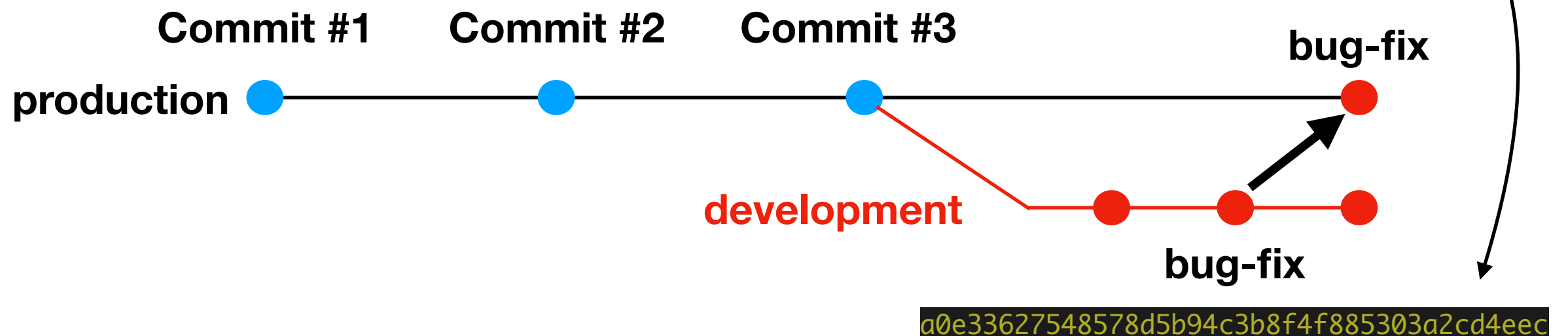
cherry pick - отбирать только хорошие вишенки

git cherry-pick

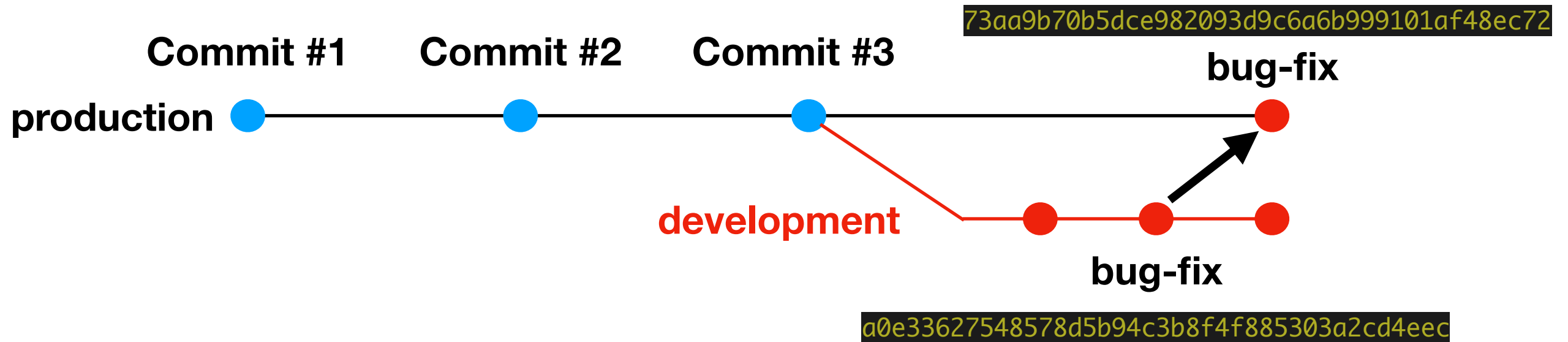
Убеждаемся, что мы в
правильной ветке

git checkout production

git cherry-pick хэш_коммита

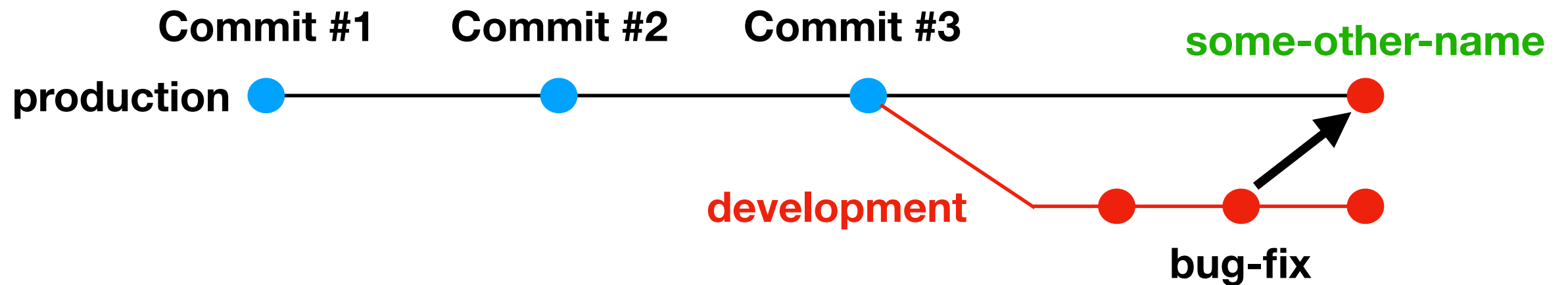


git cherry-pick



- У взятого коммита будет другой хэш, отличный от хэша в оригинальной ветке (то есть формально это **НОВЫЙ КОММИТ**)

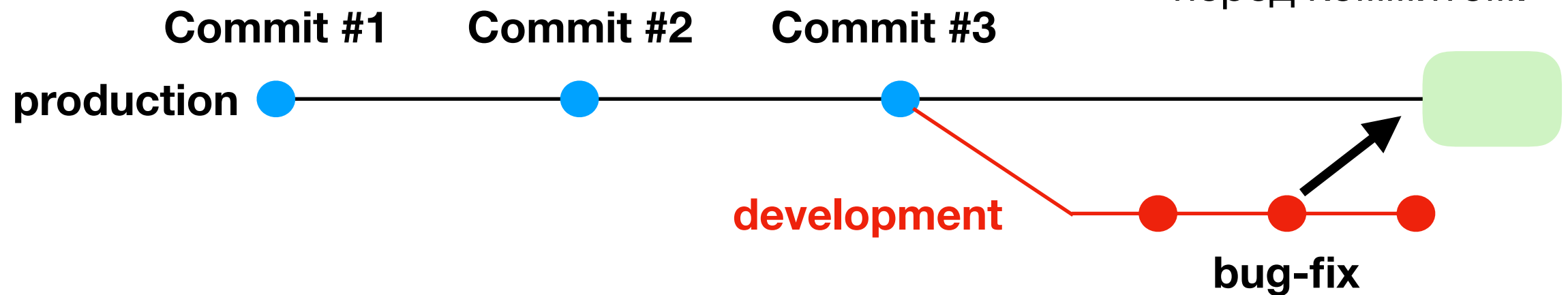
git cherry-pick --edit



- Хотим перенести коммит из другой ветки, но при этом хотим поменять сообщение коммита

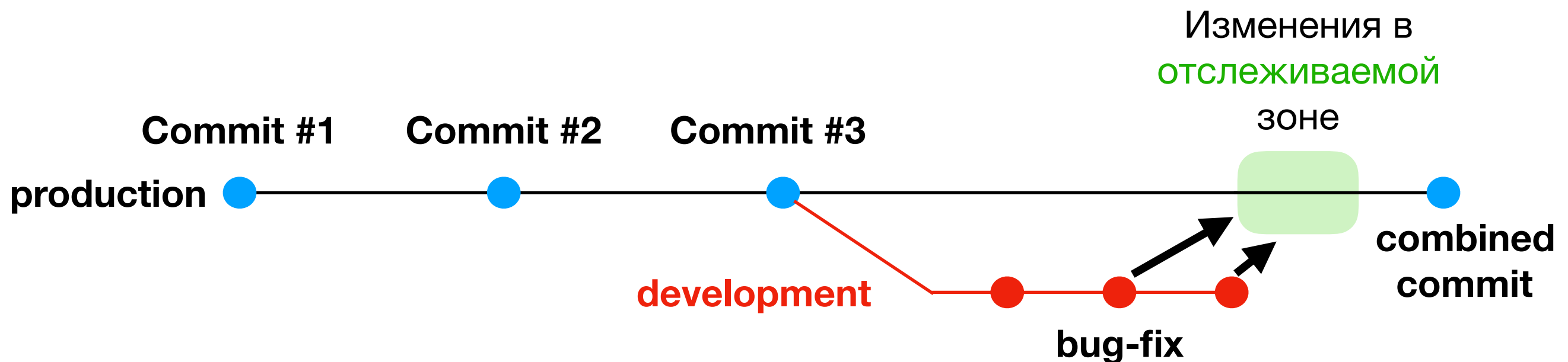
git cherry-pick --no-commit

Изменения находятся в
отслеживаемой зоне.
Можем внести правки
перед коммитом.



- Хотим перенести изменения из коммита из другой ветки, но при этом не хотим делать коммит в нашей ветке. Хотим, чтобы изменения просто попали в отслеживаемую зону.
- Это бывает полезно, если мы хотим внести небольшие правки в тот **КОММИТ**, который мы забираем из другой ветки

git cherry-pick --no-commit



- Другой сценарий - слияние двух коммитов из другой ветки в один коммит

Другие параметры

- **git cherry-pick -x** хэш_коммита
 - Указывает в сообщении коммита хэш того коммита, из которого мы сделали cherry-pick
- **git cherry-pick --signoff** хэш_коммита
 - Указывает в сообщении коммита имя того пользователя, кто совершил cherry-pick

GitFlow — модель ветвления для управления разработкой

GitFlow — это популярная модель ветвления для работы с системой контроля версий Git, разработанная Винсентом Дриессеном. Она предназначена для управления проектами с долгосрочной разработкой и частыми релизами.

Основные принципы GitFlow:

1. Основные ветки:

- **Main (master):** содержит стабильные и протестированные версии кода. Используется только для релизов.

- **Develop:** основная рабочая ветка для интеграции новых фич и изменений. Это "актуальное состояние разработки".

2. Дополнительные ветки:

- **Feature-ветки:** создаются для работы над отдельными функциональностями. После завершения разработки фича объединяется в `develop`.
- **Release-ветки:** используются для подготовки релиза. Сюда входят тестирование, исправление багов и подготовка документации. После завершения изменения

объединяются в `main` и `develop`.

- **Hotfix-ветки:** создаются для срочных исправлений в релизной версии. После исправления изменения вносятся как в `main`, так и в `develop`.

3. Процесс работы:

- Начало работы над новой функцией начинается с создания `feature`-ветки от `develop`.
- После завершения работы над функцией ветка объединяется обратно в `develop`.

- Для подготовки релиза создаётся `release`-ветка от `develop`. После финальной проверки изменения вливаются в `main` и `develop`.
- При обнаружении критических ошибок в релизе создаётся `hotfix`-ветка от `main`.

Преимущества GitFlow:

- Структурированный процесс управления кодом.
- Чёткое разделение рабочих веток и стабильного кода.
- Удобство для работы с большими командами и долгосрочными проектами.

Недостатки:

- . Сложность для небольших проектов или команд.
- . Требуется дополнительных усилий для управления ветками.

GitFlow широко применяется в проектах, где важно поддерживать чёткое разделение этапов разработки, тестирования и релиза.

i

