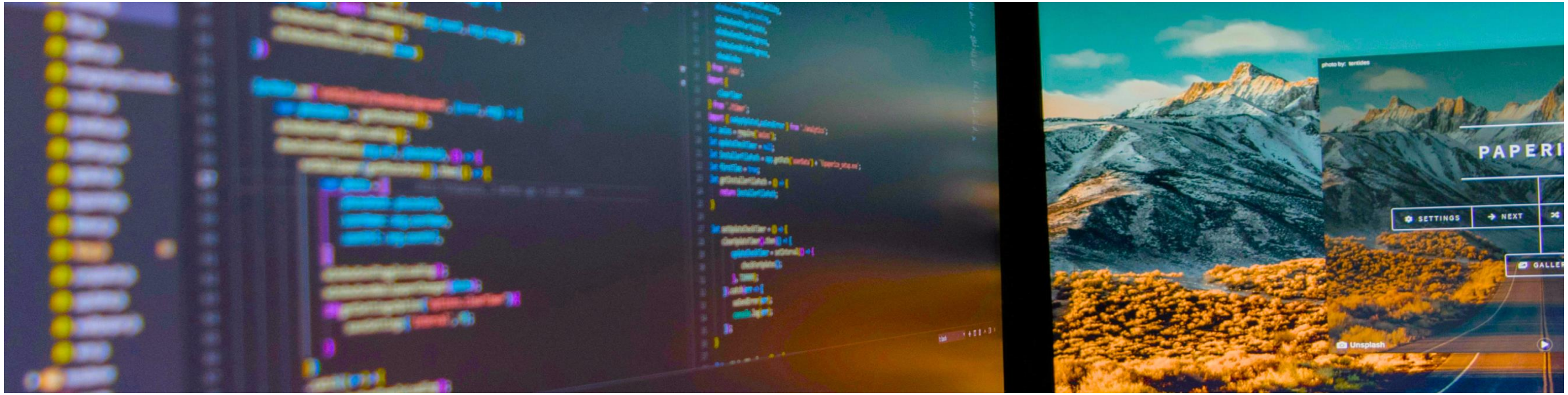
N*



Управление производственным процессом разработки программного обеспечения

Системы контроля версий

ФИТ Лекция №3. 25'

План

1. Системы контроля версий
2. Фундаментальные основы GIT
3. Разбираемся как его использовать

Что такое VCS

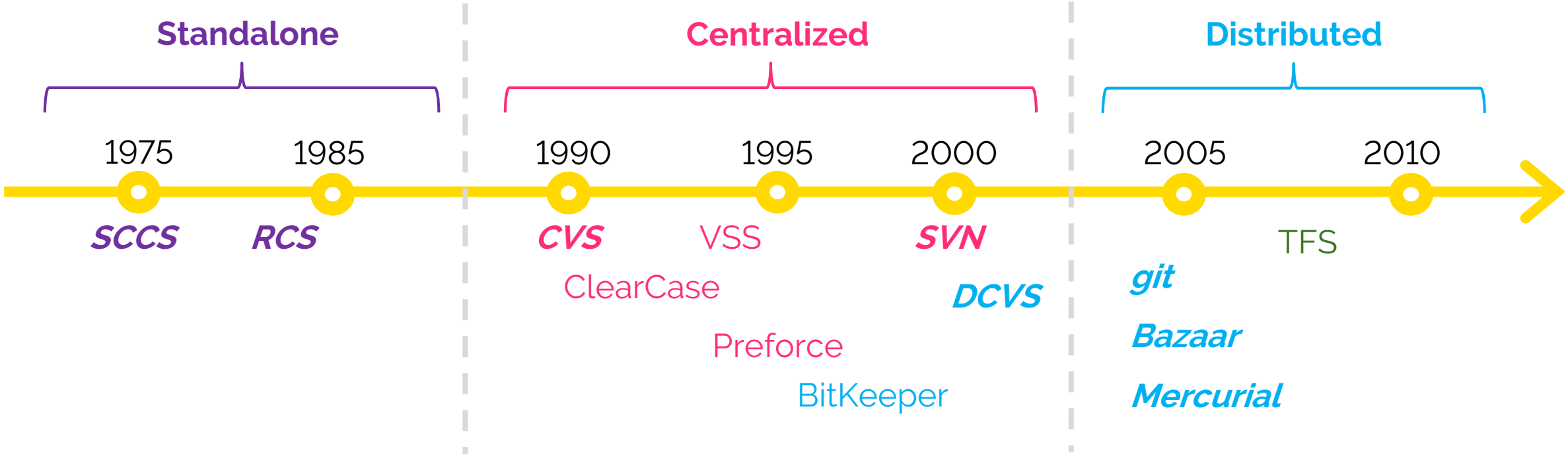
Version Control System (VSC, часто встречается название Revision Control System) –

это системы контроля версий. VCS позволяют хранить несколько версий одного и того же файла, возвращаться к более ранним версиям, отслеживать изменения.

Версией или ревизией (revision) называется конкретное зафиксированное состояние хранилища (репозитория)

Производственный процесс

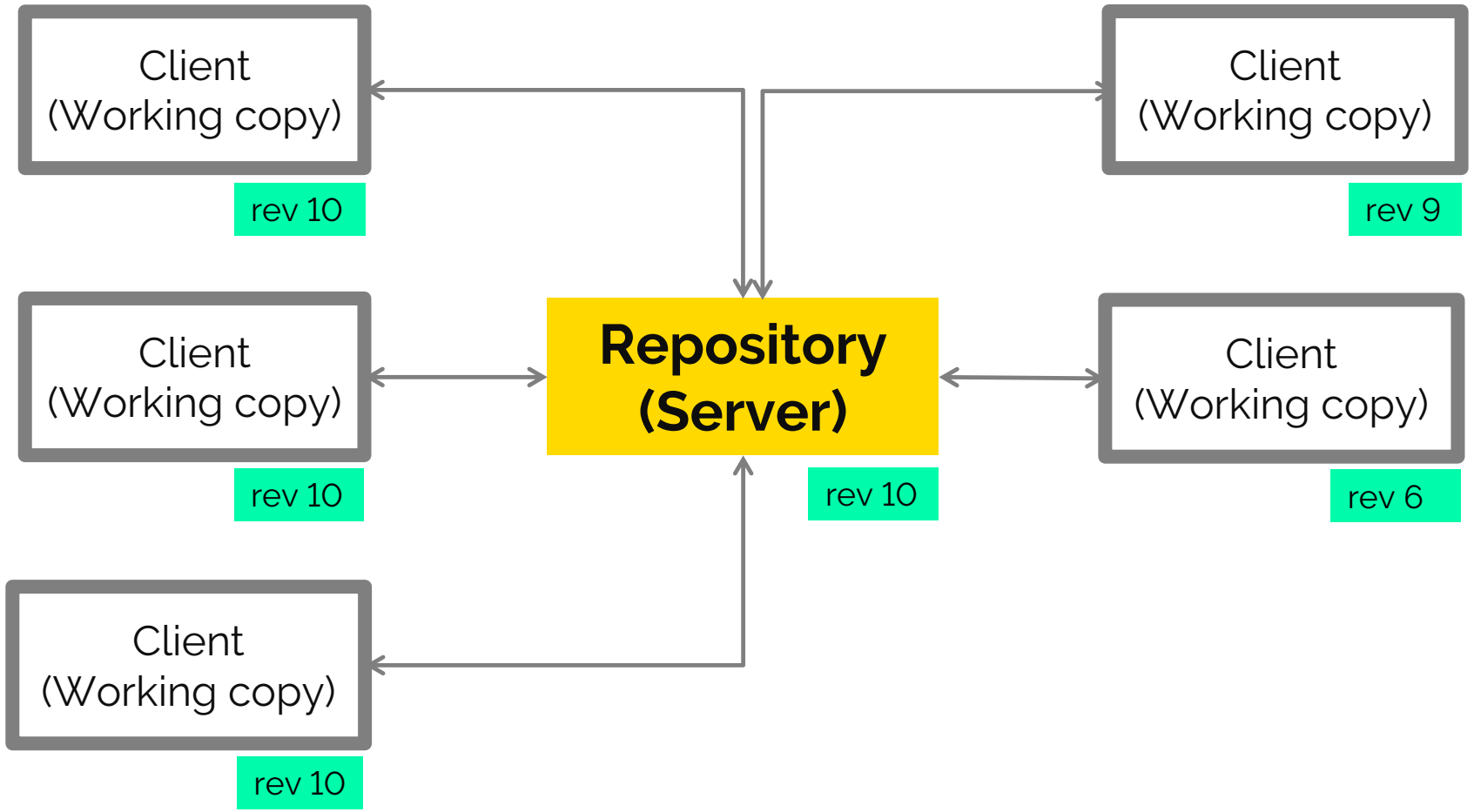
Основные VCS



Требования к VCS

- Single source of truth
- ACID - Атомарность
- ACID - Консистентность
- ACID - Изоляция
- ACID - Устойчивость
- Управление правами
- Возможность вести параллельную разработку
- Работа с историей

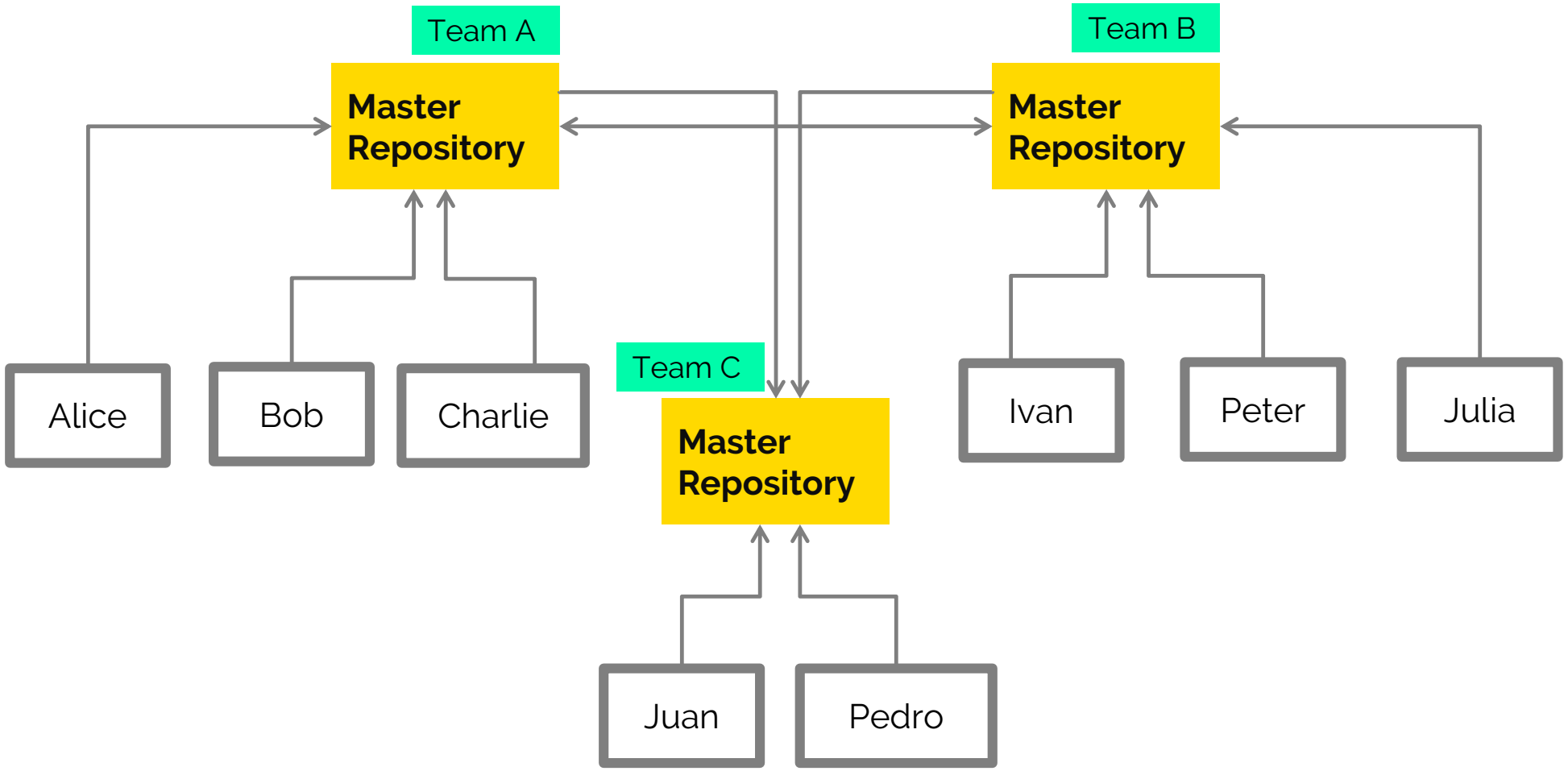
Centralized VCS



Недостатки централизованных VCS

- В большой команде сложно администрировать
- Производительность одного сервера ограничена
- Количество конфликтов зависит от размера команды
- Отказ сервера парализует работу команды

Master-master репликация



Проблемы с master-master репликацей

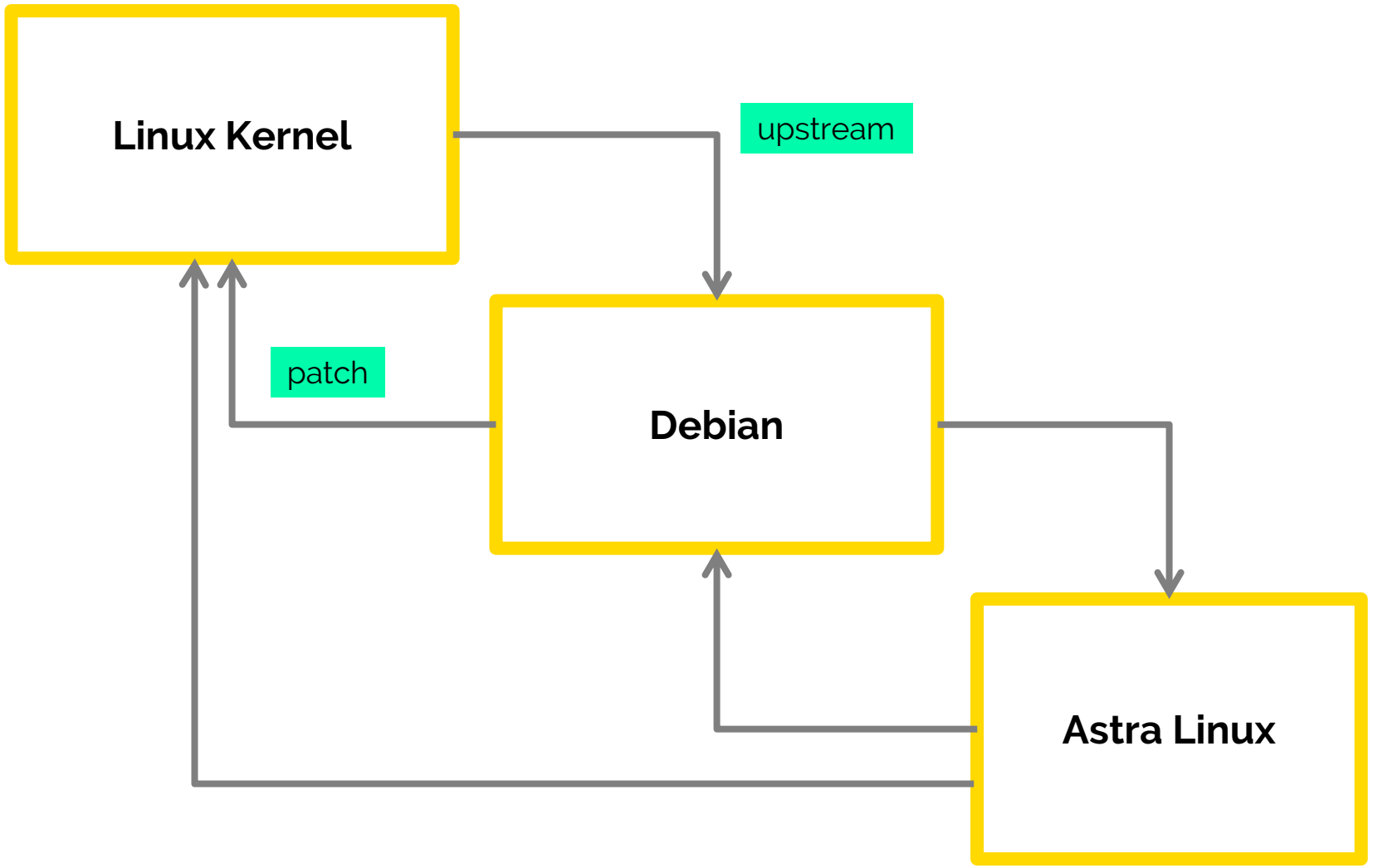
- Сложно обеспечить баланс между консистентностью, производительностью и отказоустойчивостью
- Сложно администрировать
- Сложно интегрировать изменения между командами
- Мало чем отличается от одного большого репозитория

Предпосылки появления GIT

- Лицензионный конфликт с BitKeeper
- Готовность ИТ индустрии к стандартизации
- Много (тысячи) разработчиков
- Много кода (миллионы файлов)
- Длинная история (30 лет)
- Много производных продуктов у которых:
  - собственный цикл релизов
  - собственный набор патчей
  - обмен патчами друг с другом
  - собственная сборка артефактов

Децентрализованные VCS

Trust network



Строим git

- Невозможно создать выделенный «центральный» сервер
- Невозможно обеспечить надежную синхронизацию времени
- Как определить порядок коммитов?
- Как позволить разработчикам взаимодействовать между собой без необходимости отправлять патчи по почте?

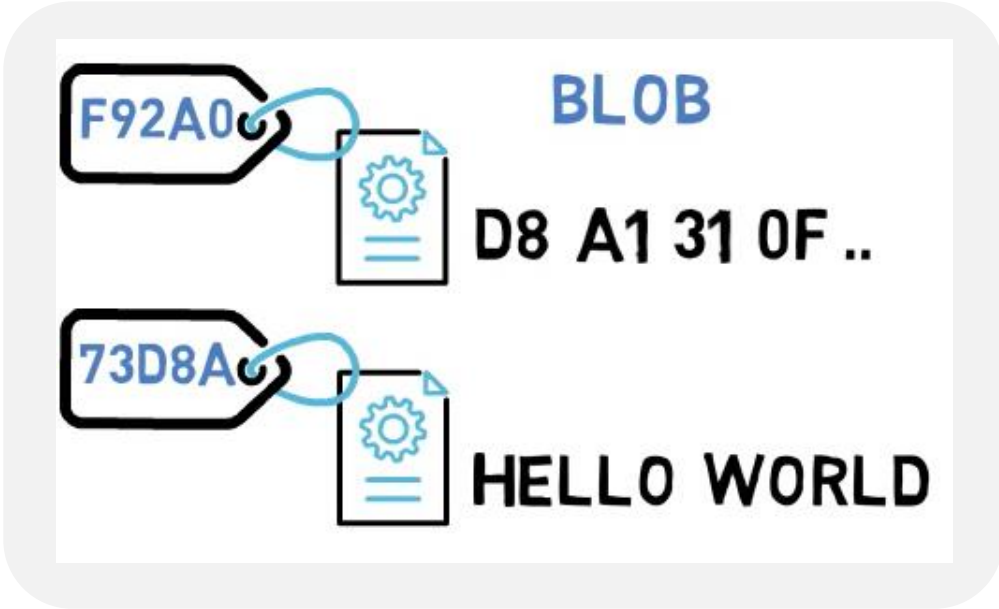
Базовые понятия: BLOB

BLOB

SHA-1 хэш функция:

git hash-object

BLOB - Binary Large OBject

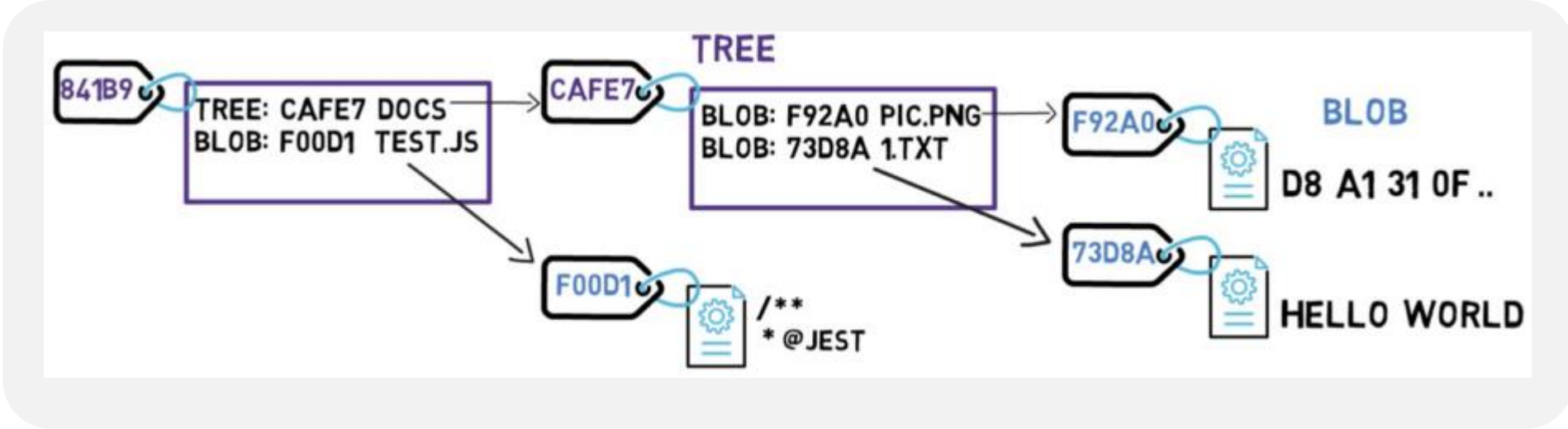


Images from https://www.freecodecamp.org/news/git-internals-objects-branches-create-repo/

Базовые понятия: TREE

Tree

Tree - дерево файлов с хэшами

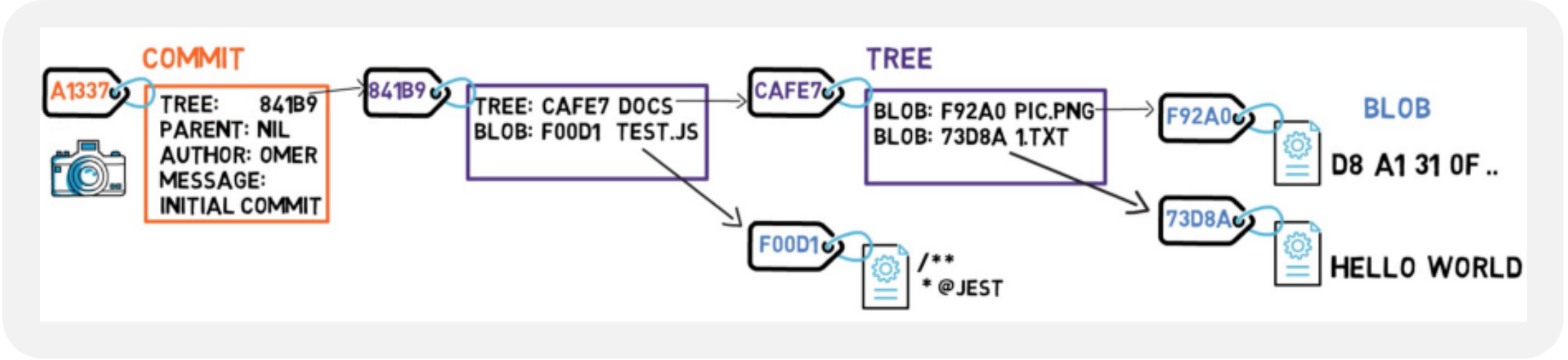


Базовые понятия: Snapshot

Commit

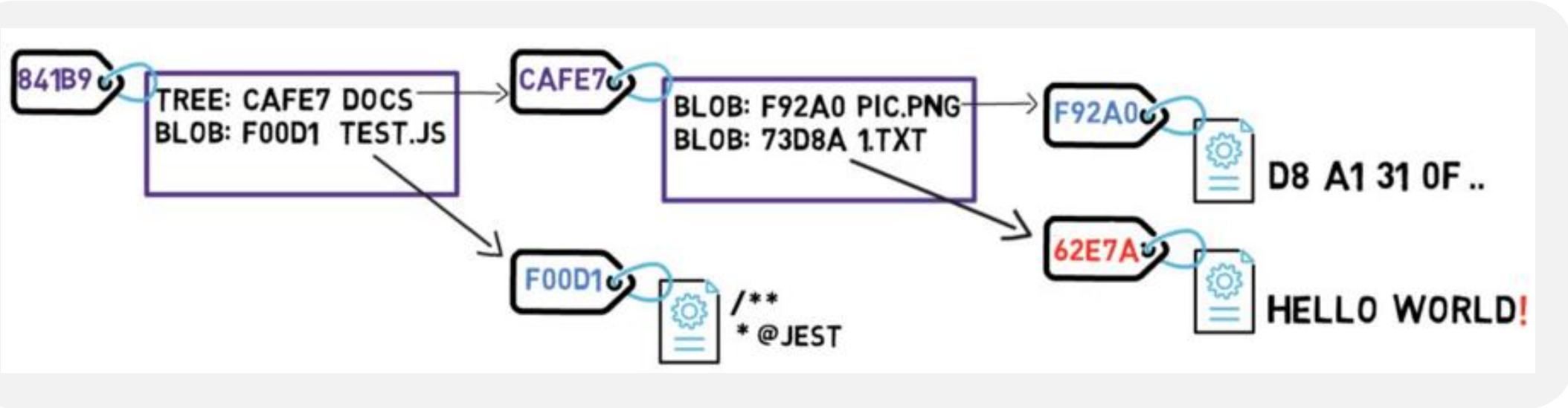
Commit - упорядоченный в истории слепок дерева

```
git write-tree
```



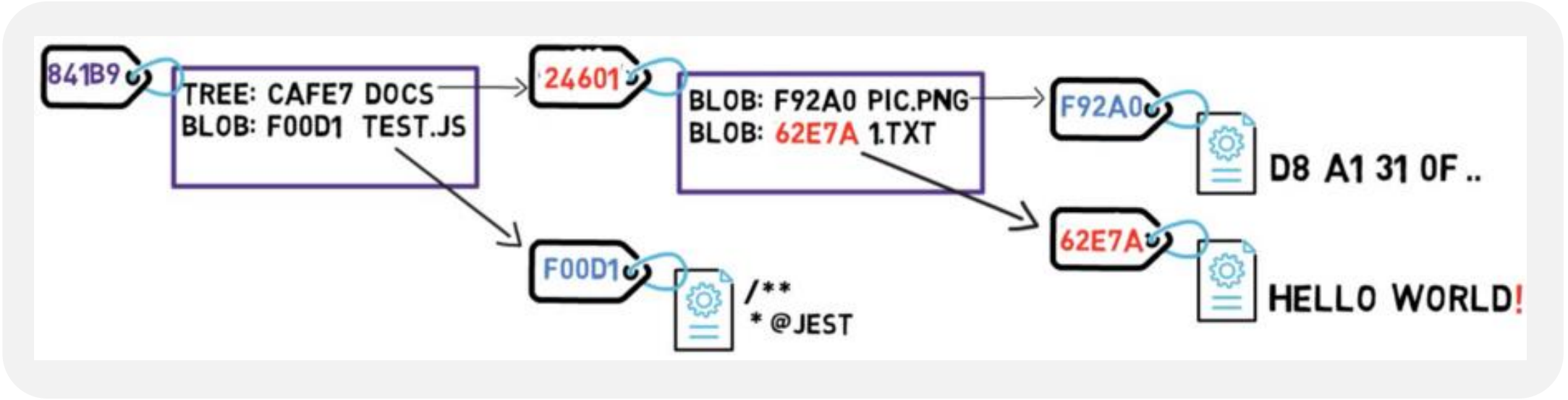
Вносим изменения: файл

Commit



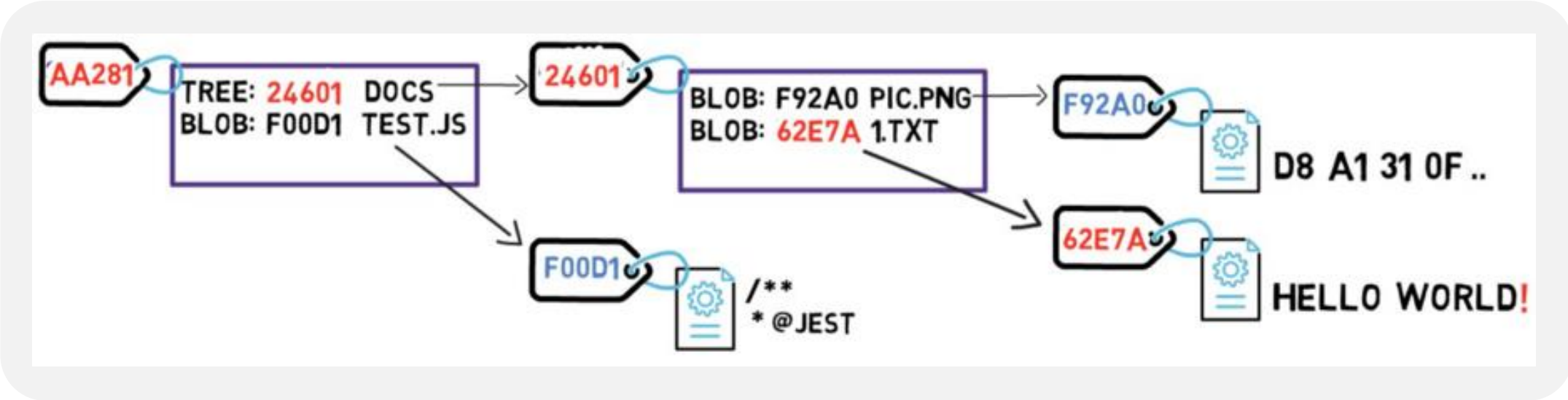
Вносим изменения: дерево

Commit



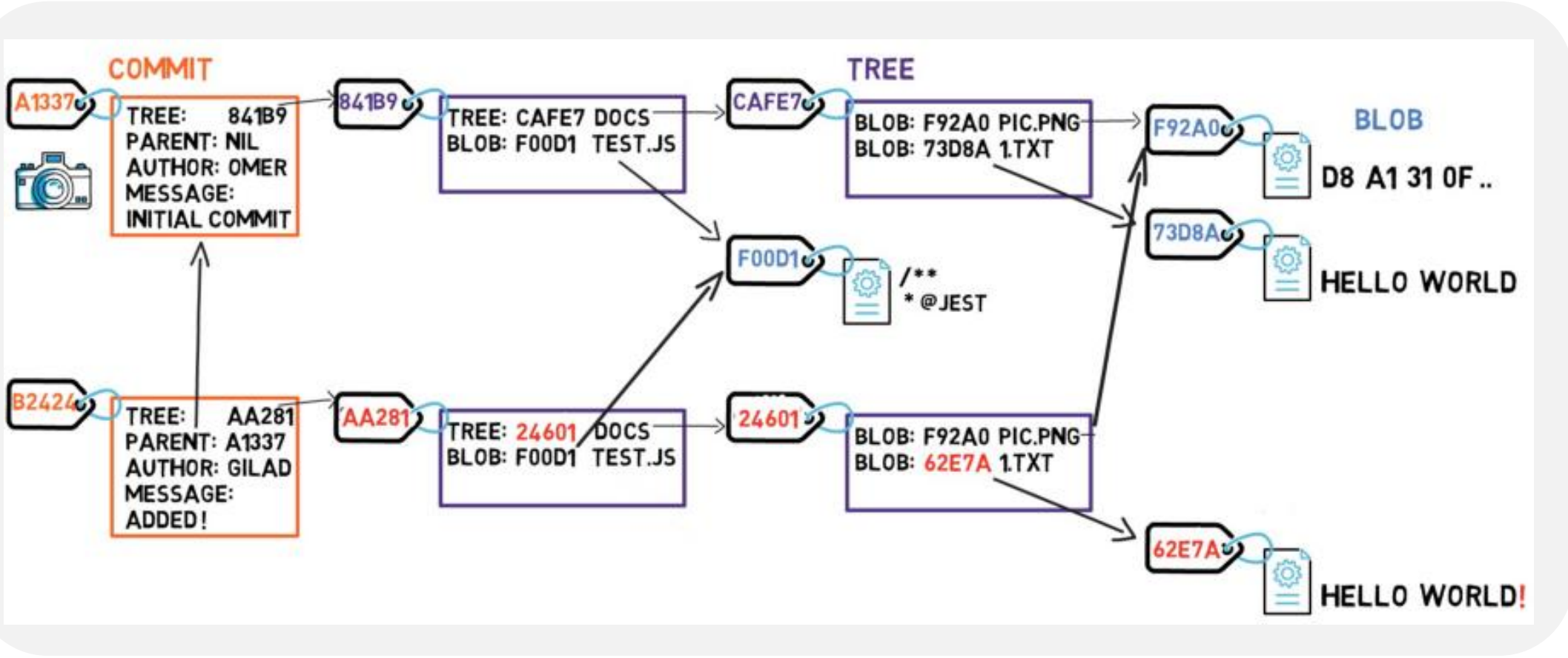
Вносим изменения: доходим до корневого элемента

Commit



Записываем новое дерево

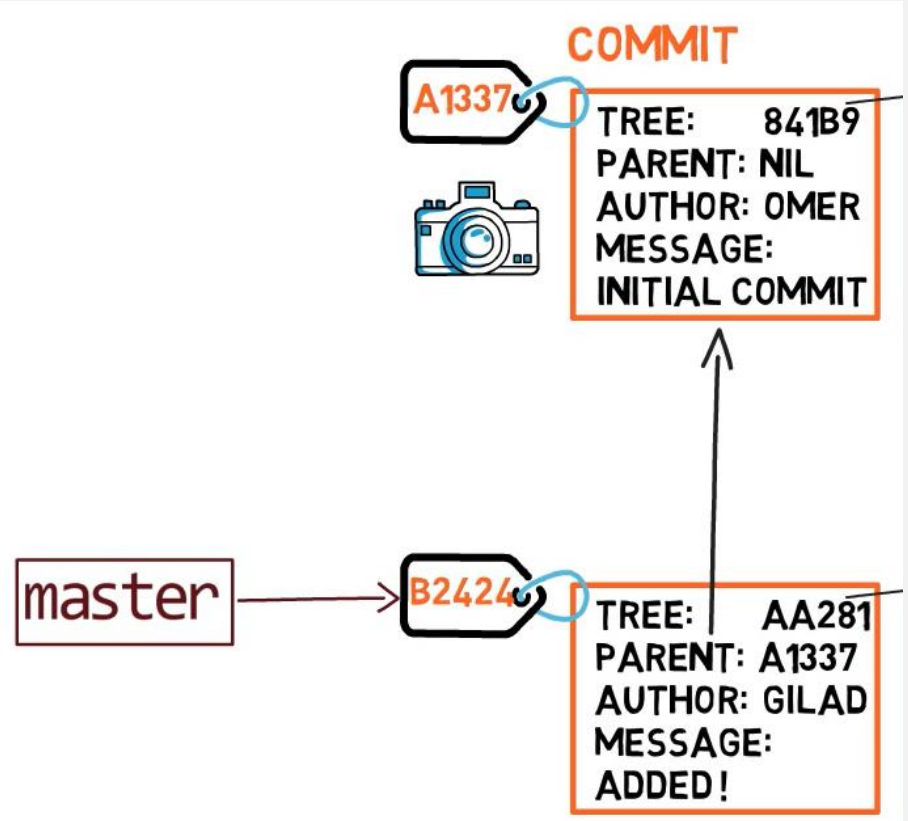
Commit



Ветви в GIT

Refs

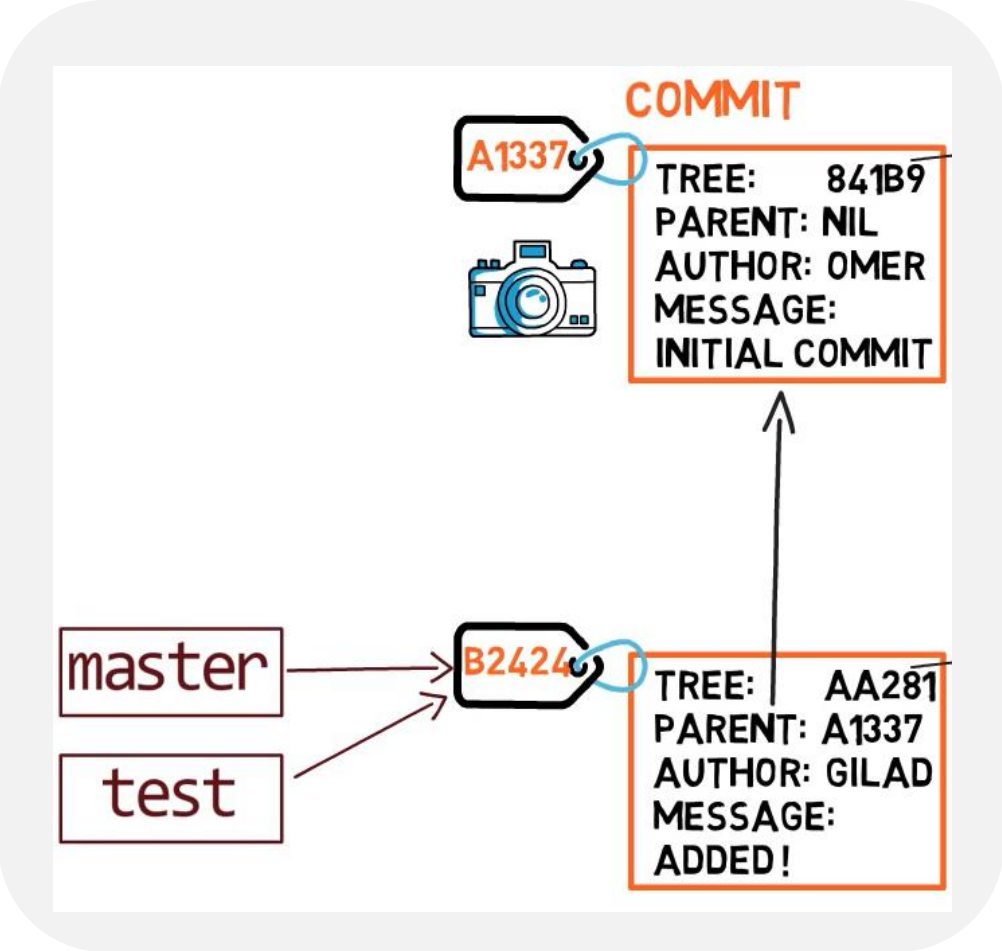
Ref - ссылка на коммит



Строим git

Ветки

Ветка – это всего лишь ссылка на коммит



Зачем нужны ветви?

Ветки

- **Release branch**. Параллельная разработка и поддержка старых версий
- **Feature branch**. Разработка нового функционала без риска сломать работающий код
- Частые коммиты в собственную ветку

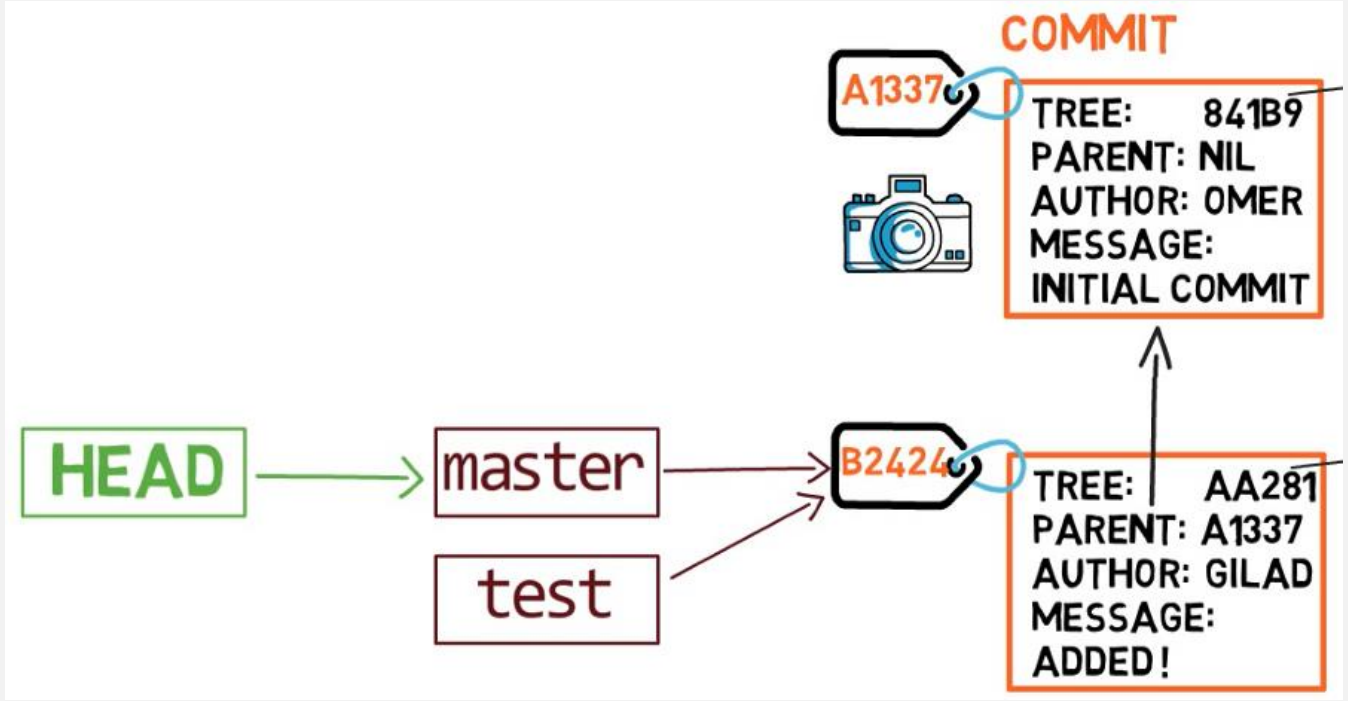
К сожалению, у ветвей есть существенный недостаток – операция слияния веток часто требует **ручной работы**

Указатель HEAD

Ветки

Специальный указатель HEAD
указывает на рабочую ветку

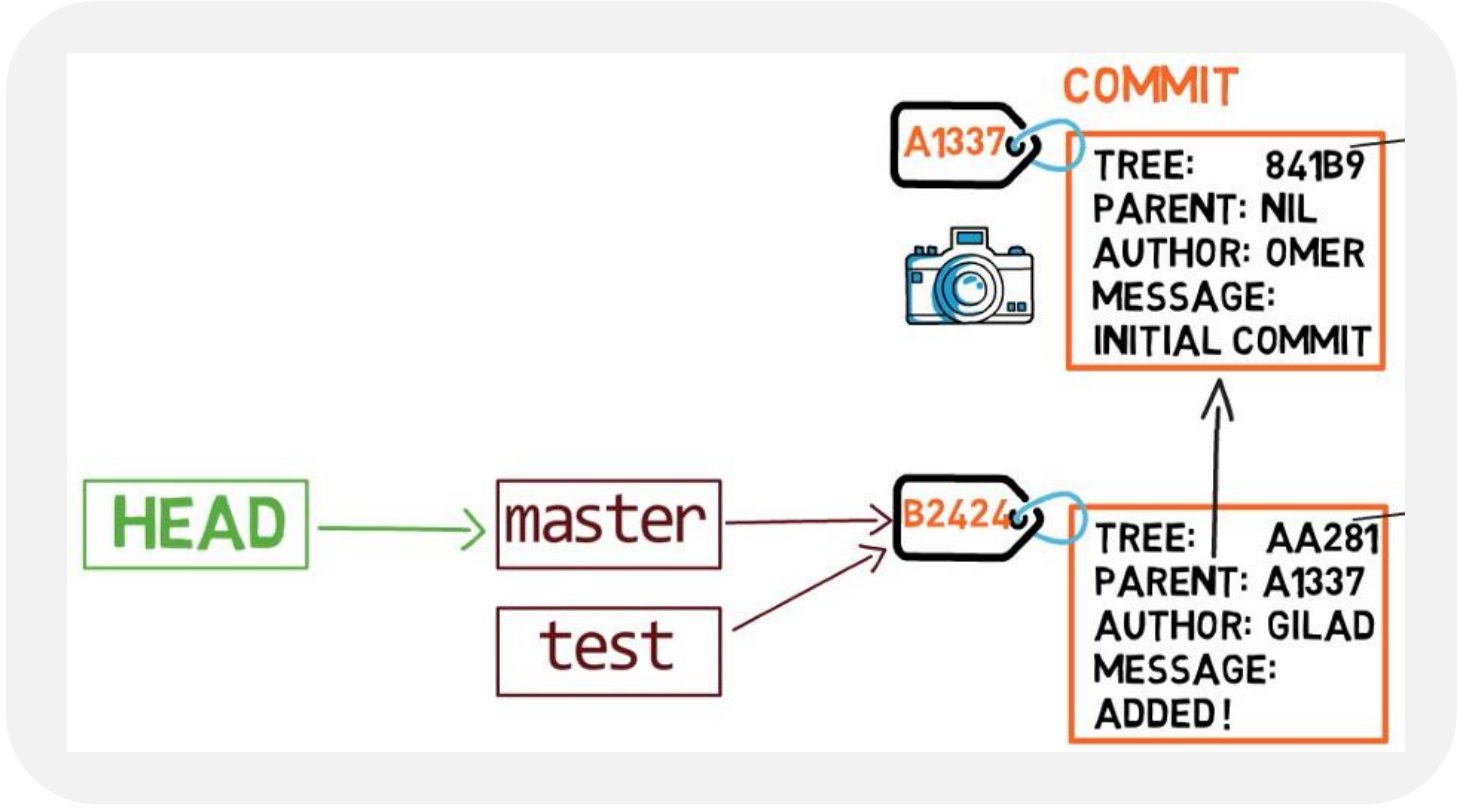
```
HEAD
objects
└──54
        f6e651fe2728e9972323dac32bf2192f17af
refs
└──heads
```



Строим git

Индекс

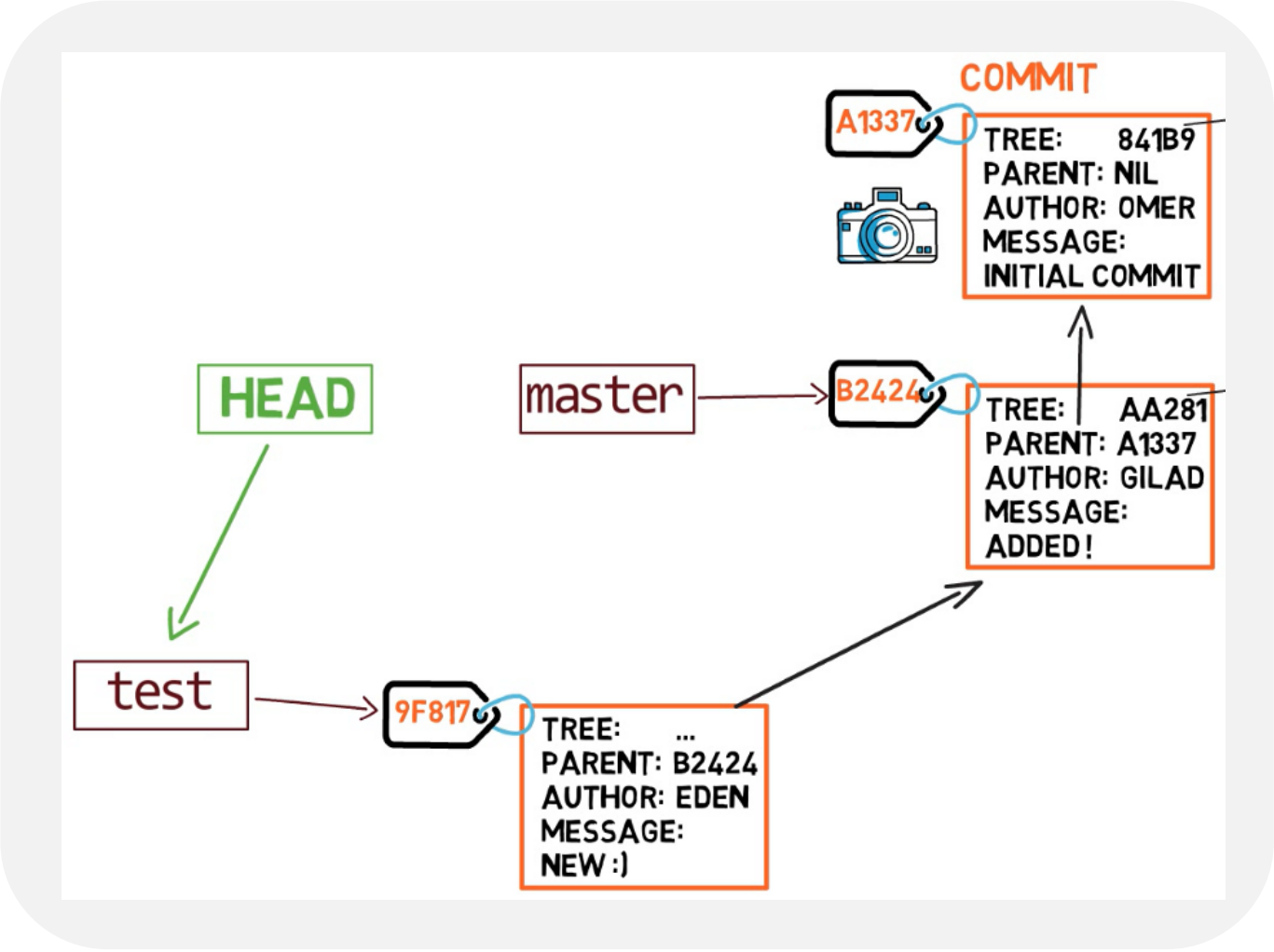
```
$ git checkout -b test
```



Строим git

Индекс

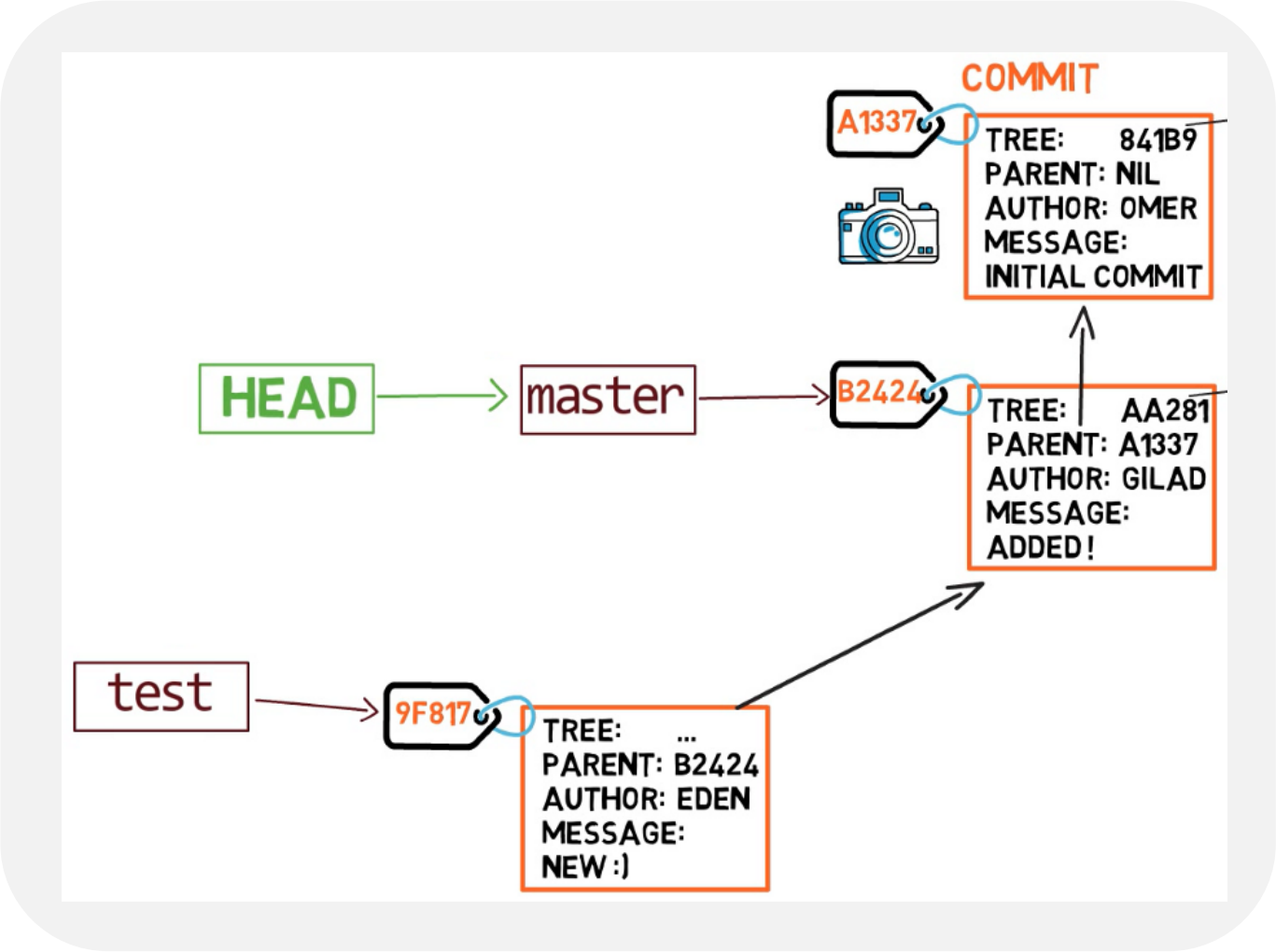
```
$ git commit -m "NEW :)"
```



Строим git

Индекс

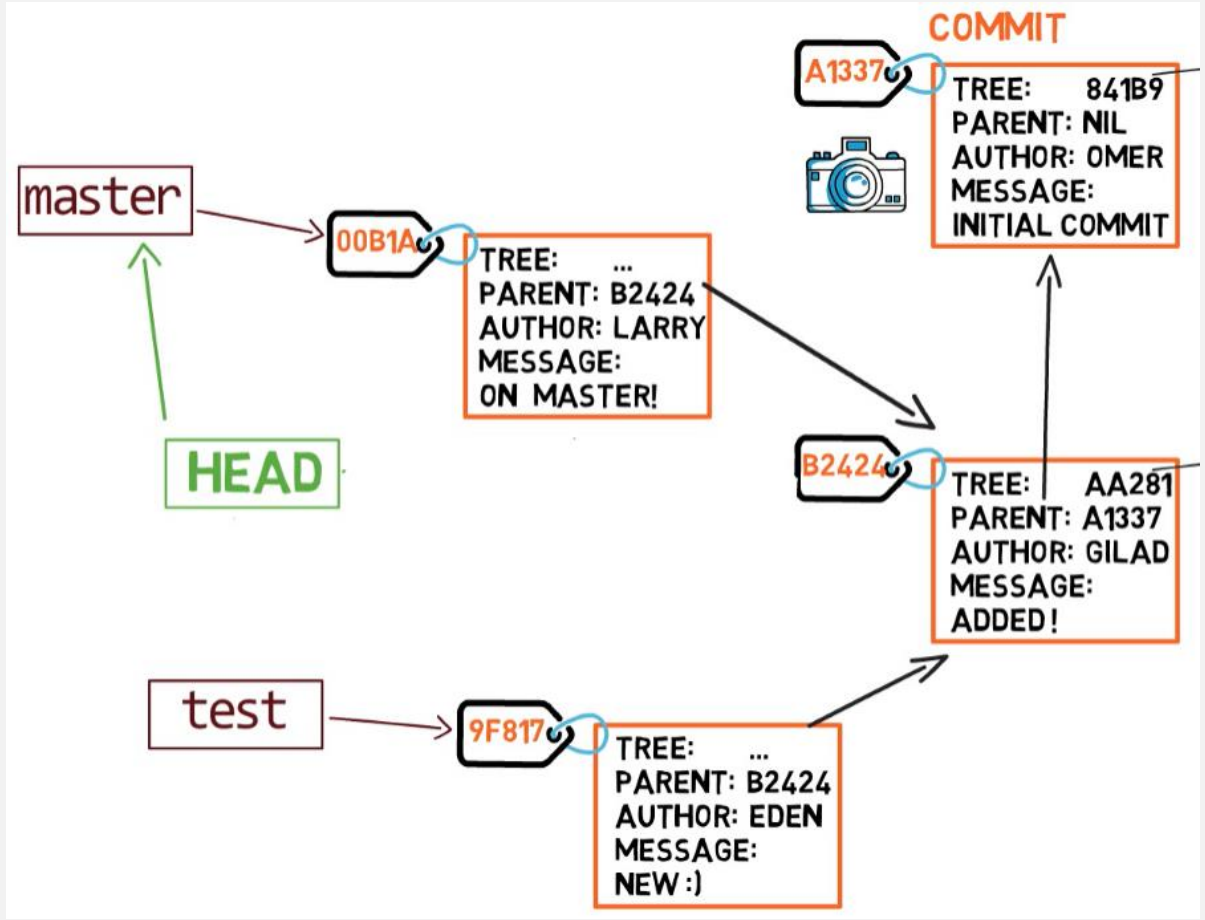
```
$ git checkout master
```



Строим git

Индекс

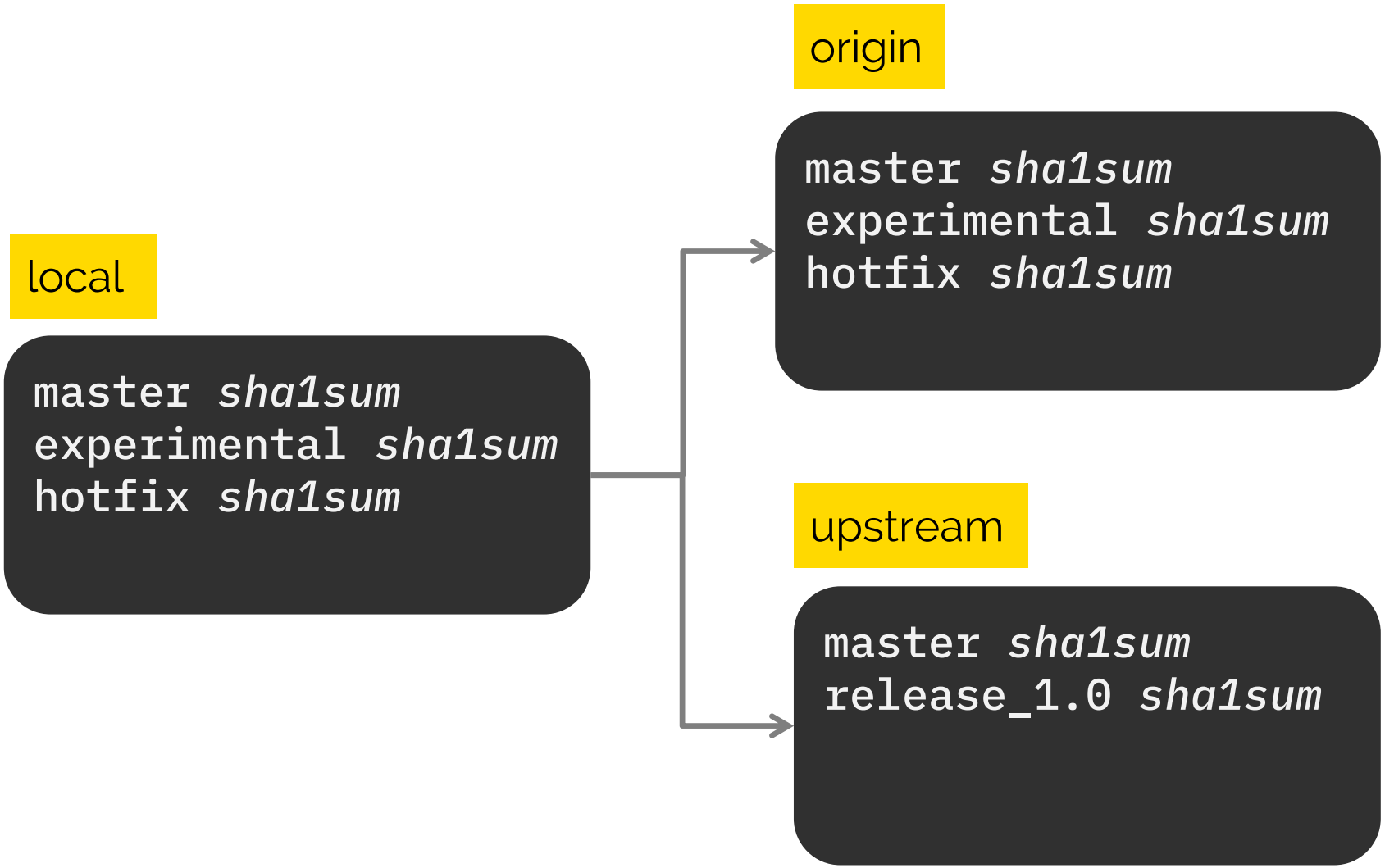
```
$ git commit -m "ON MASTER!"
```



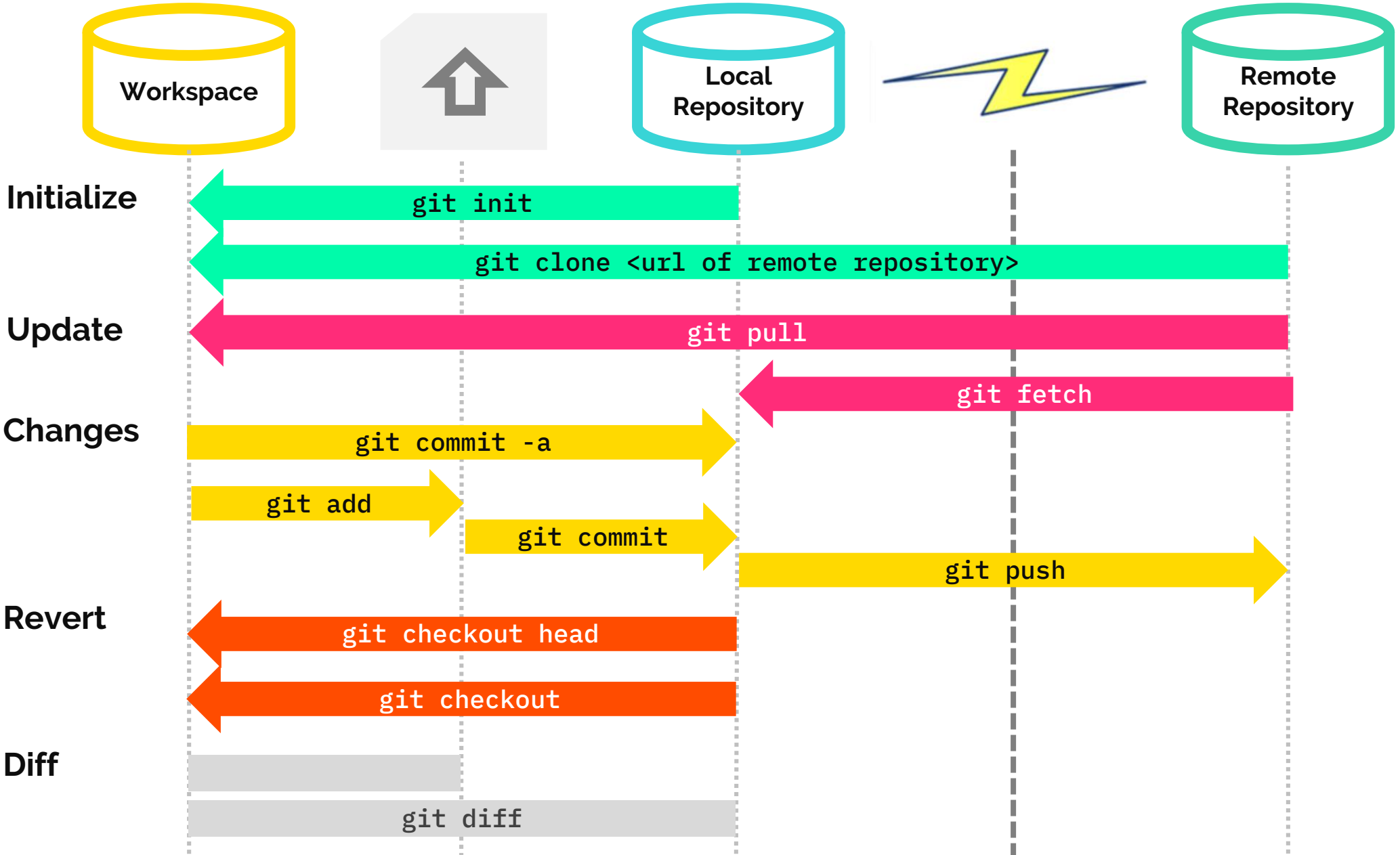
Git: что внутри

1. Файловая система с поддержкой версионирования
2. Упаковка объектов под капотом (delta compression)
3. Собственный протокол для синхронизации через HTTP/SSH
4. Сервер отличается от клиента только отсутствием рабочей копии и HEAD (bare repository)

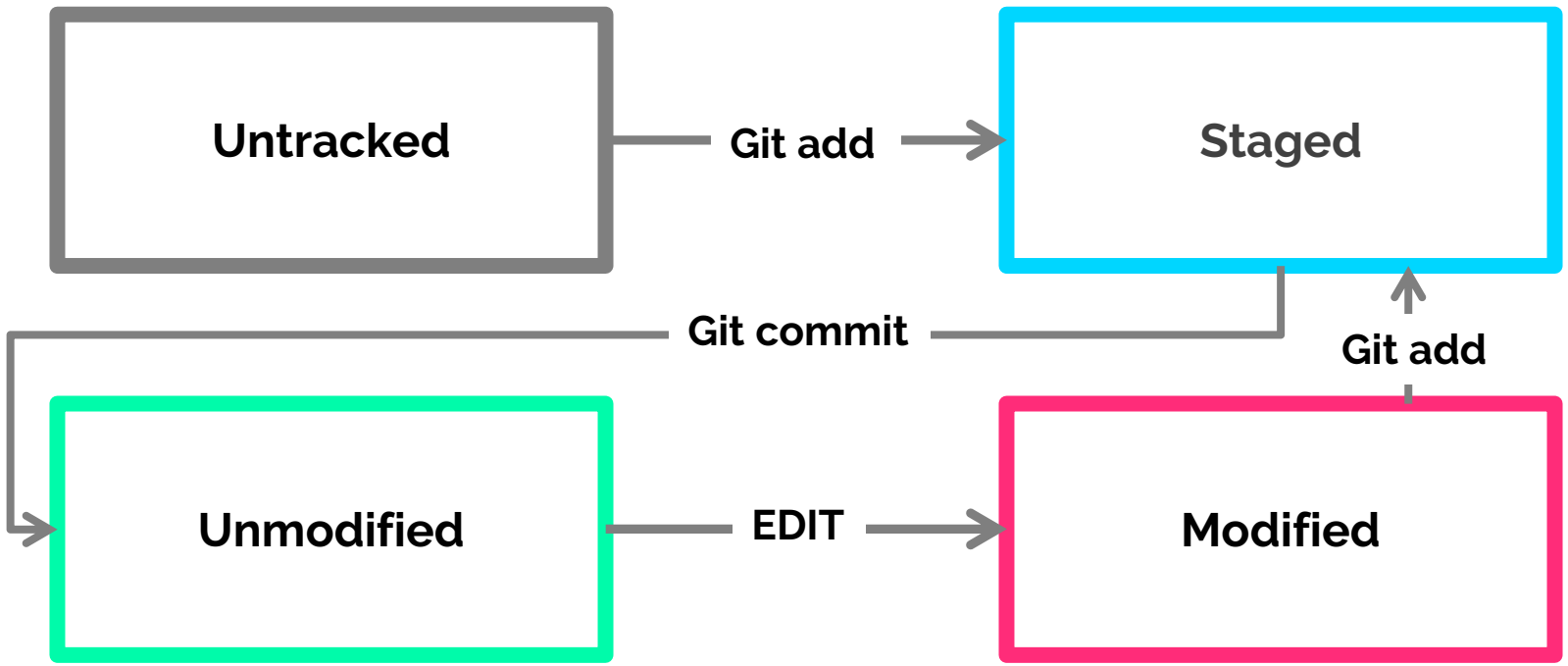
Удаленный репозиторий



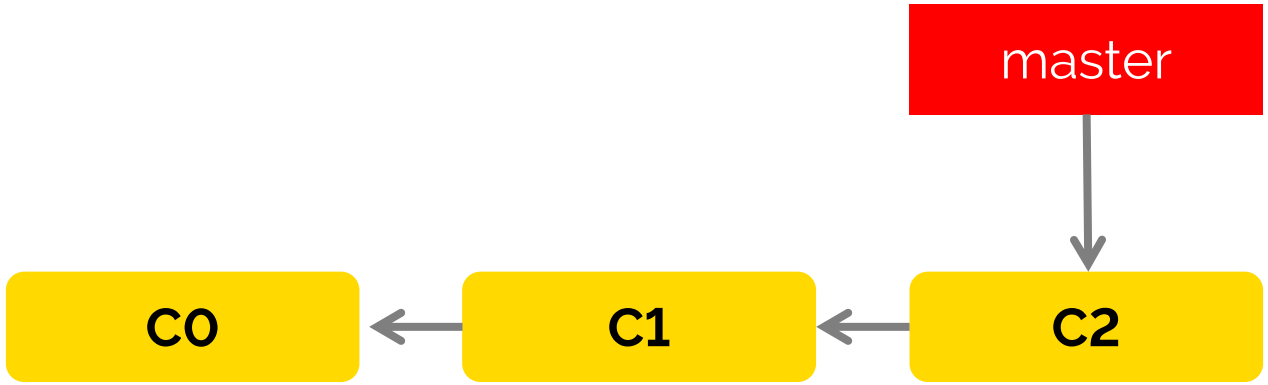
Staging area



Статус файла

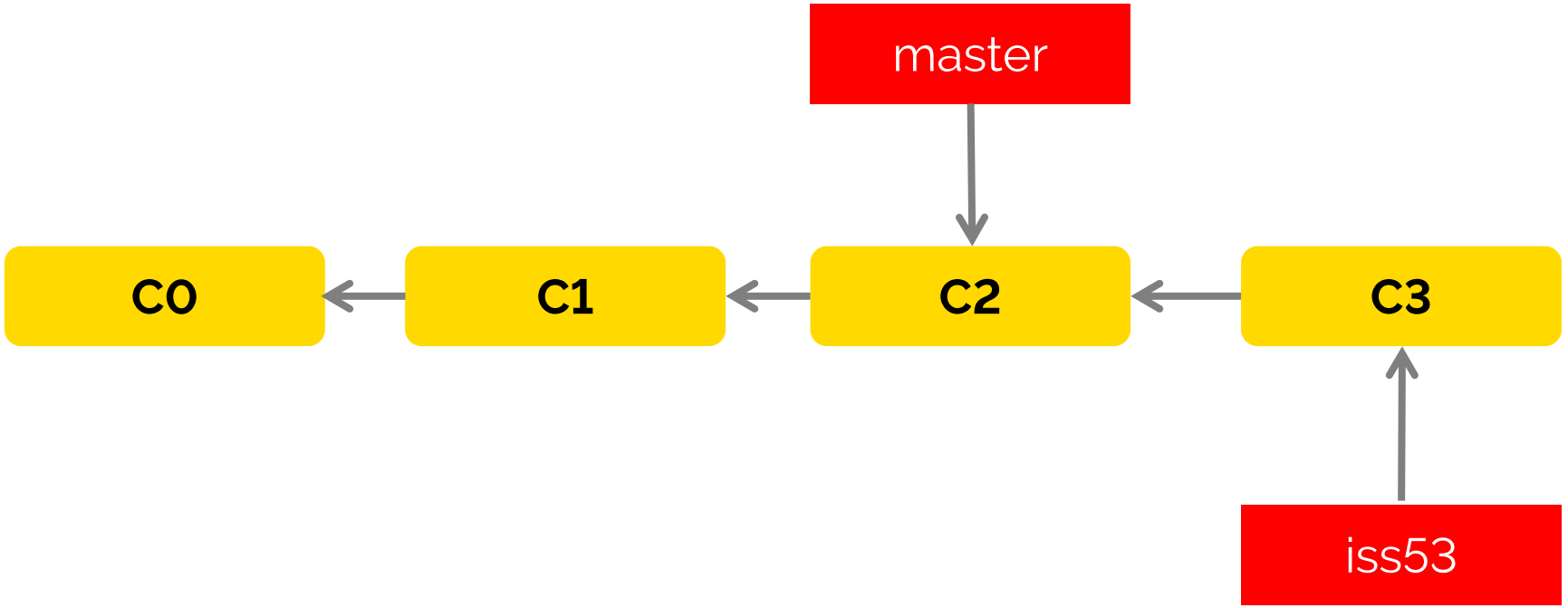


Начали разработку



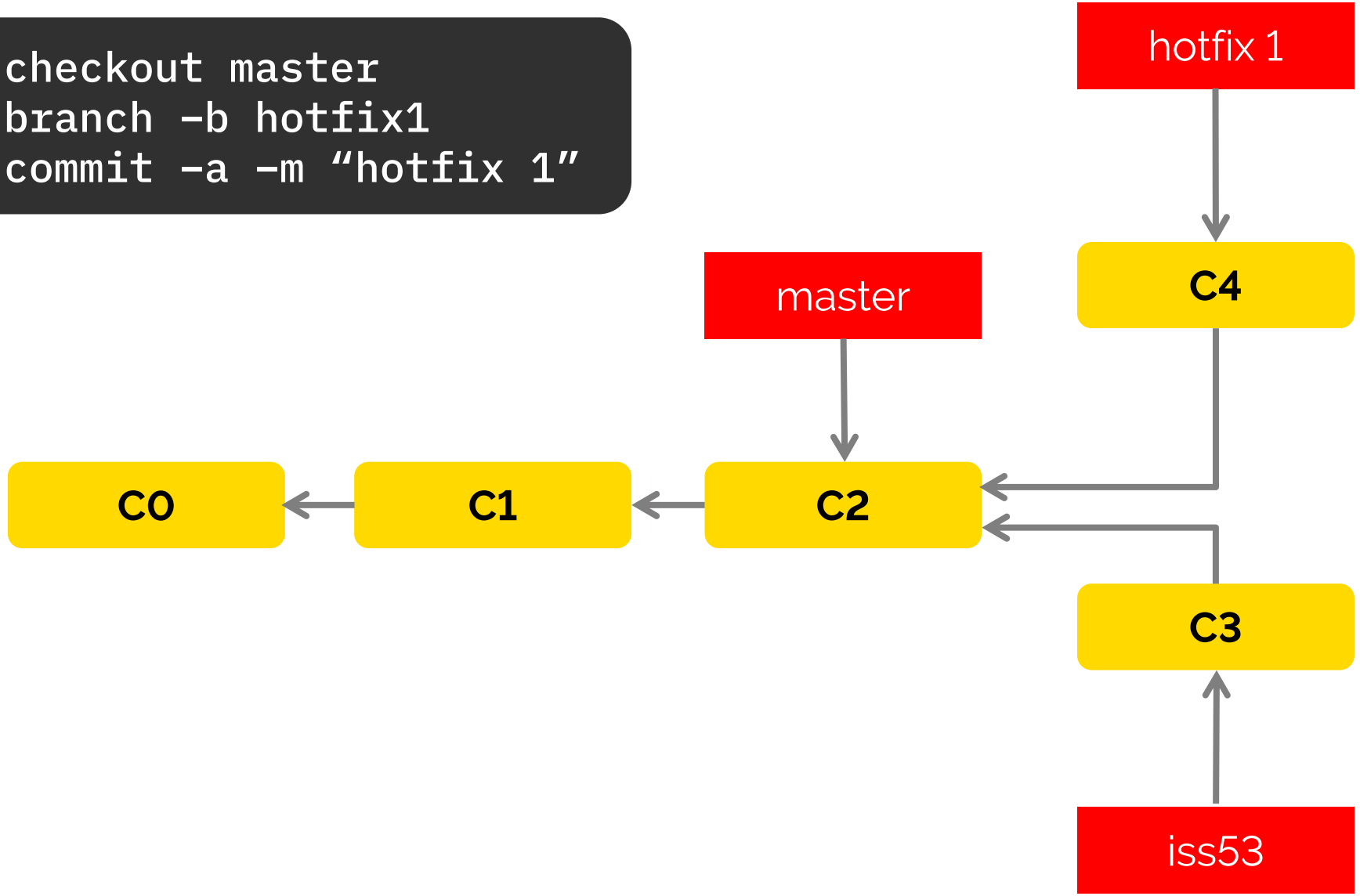
Branch: Стартуем работать над новой задачей

```
$ git checkout master
$ git branch -b iis53
$ git commit -a -m “Issues 53”
```



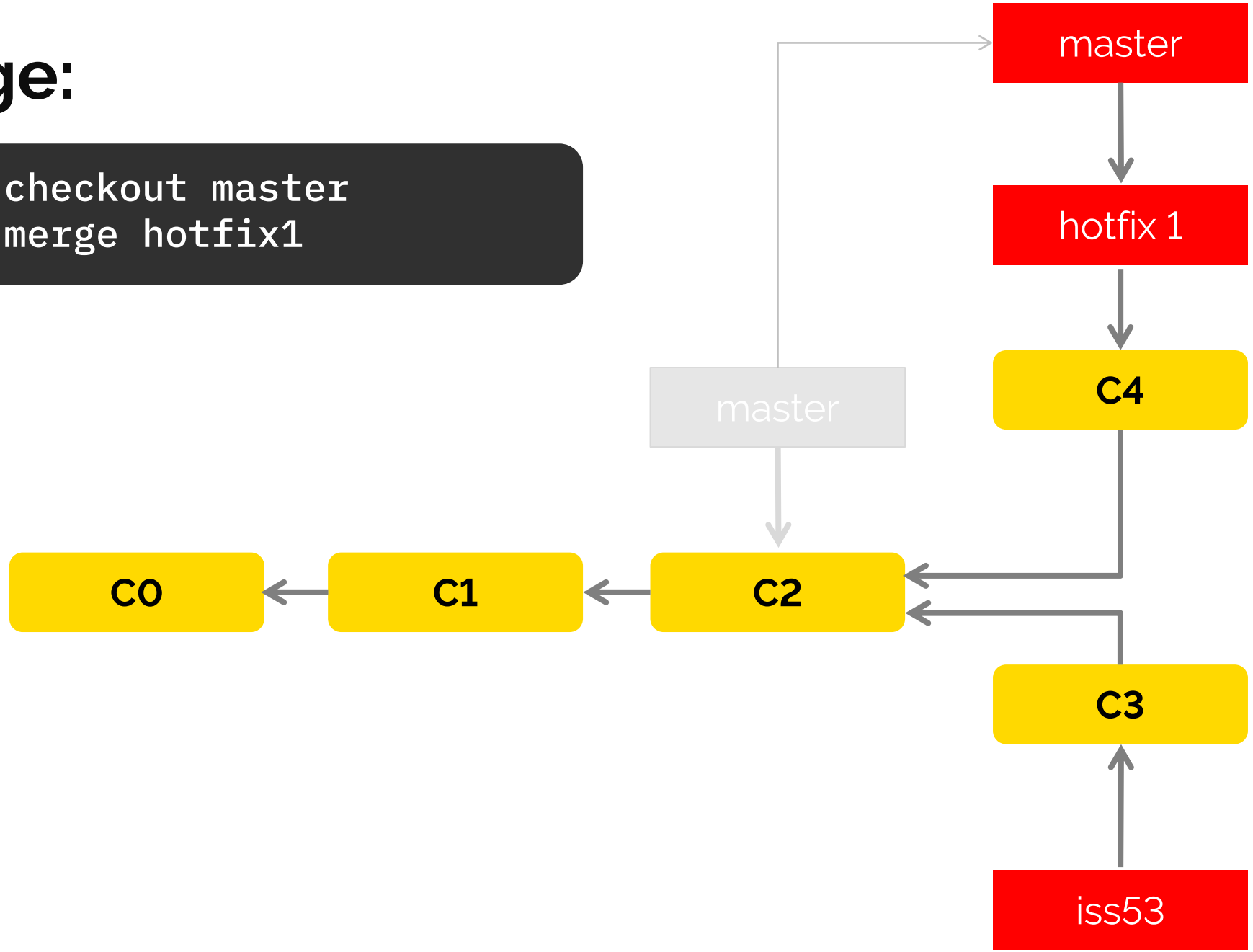
Branch: Нужно сделать исправление в prod

```
$ git checkout master
$ git branch -b hotfix1
$ git commit -a -m “hotfix 1”
```



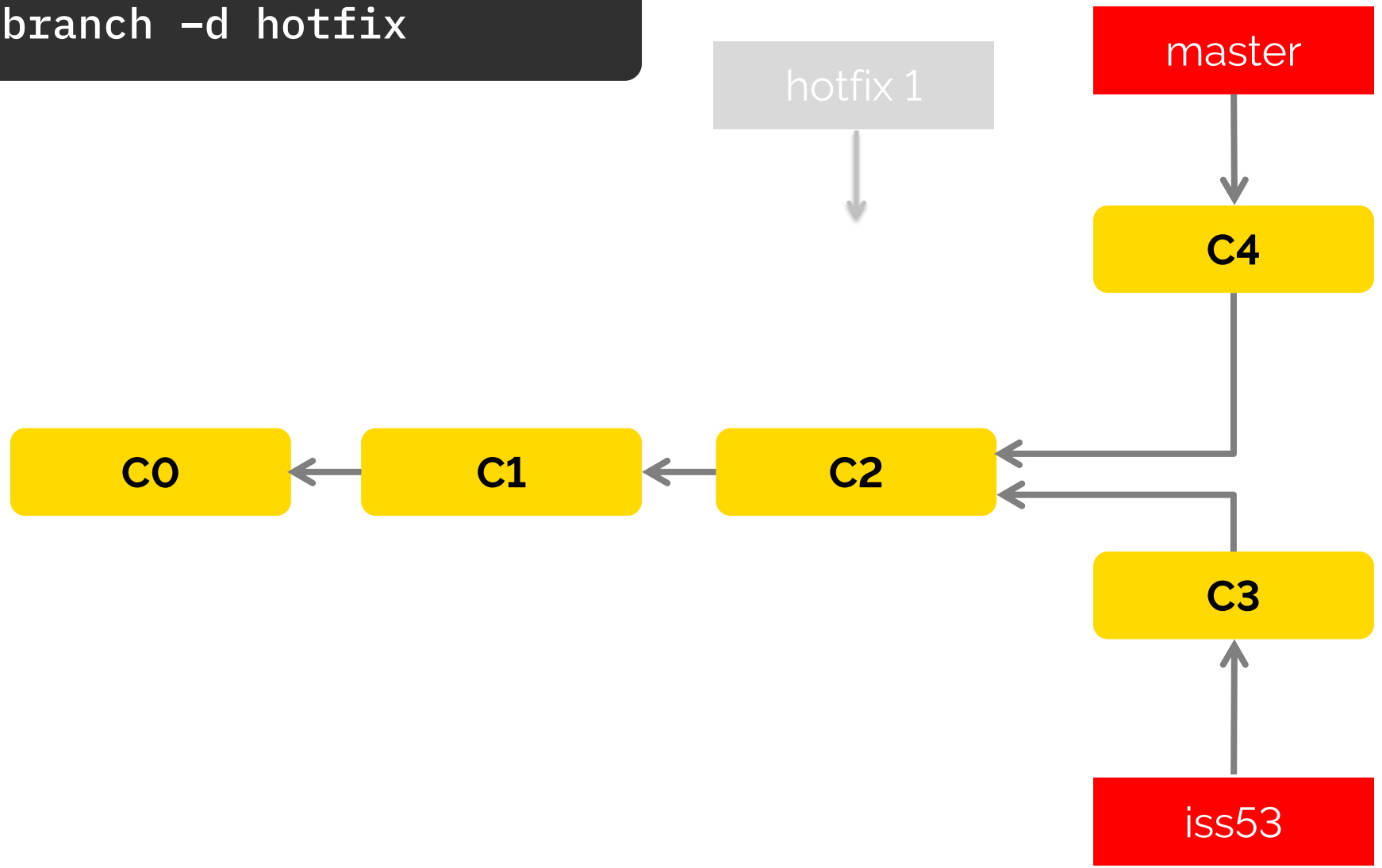
Merge:

```
$ git checkout master
$ git merge hotfix1
```

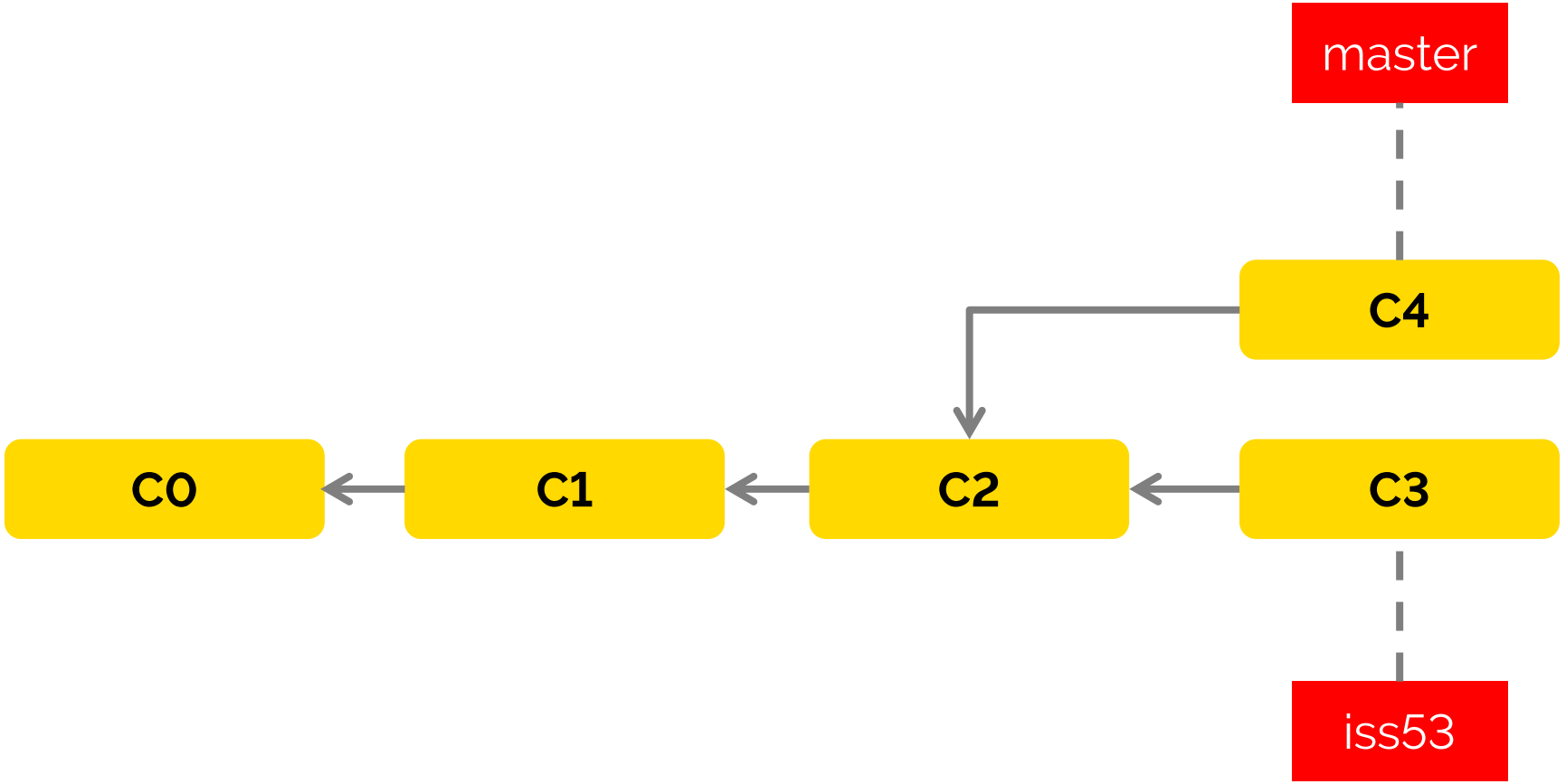


Удаляем ветку

```
$ git branch -d hotfix
```

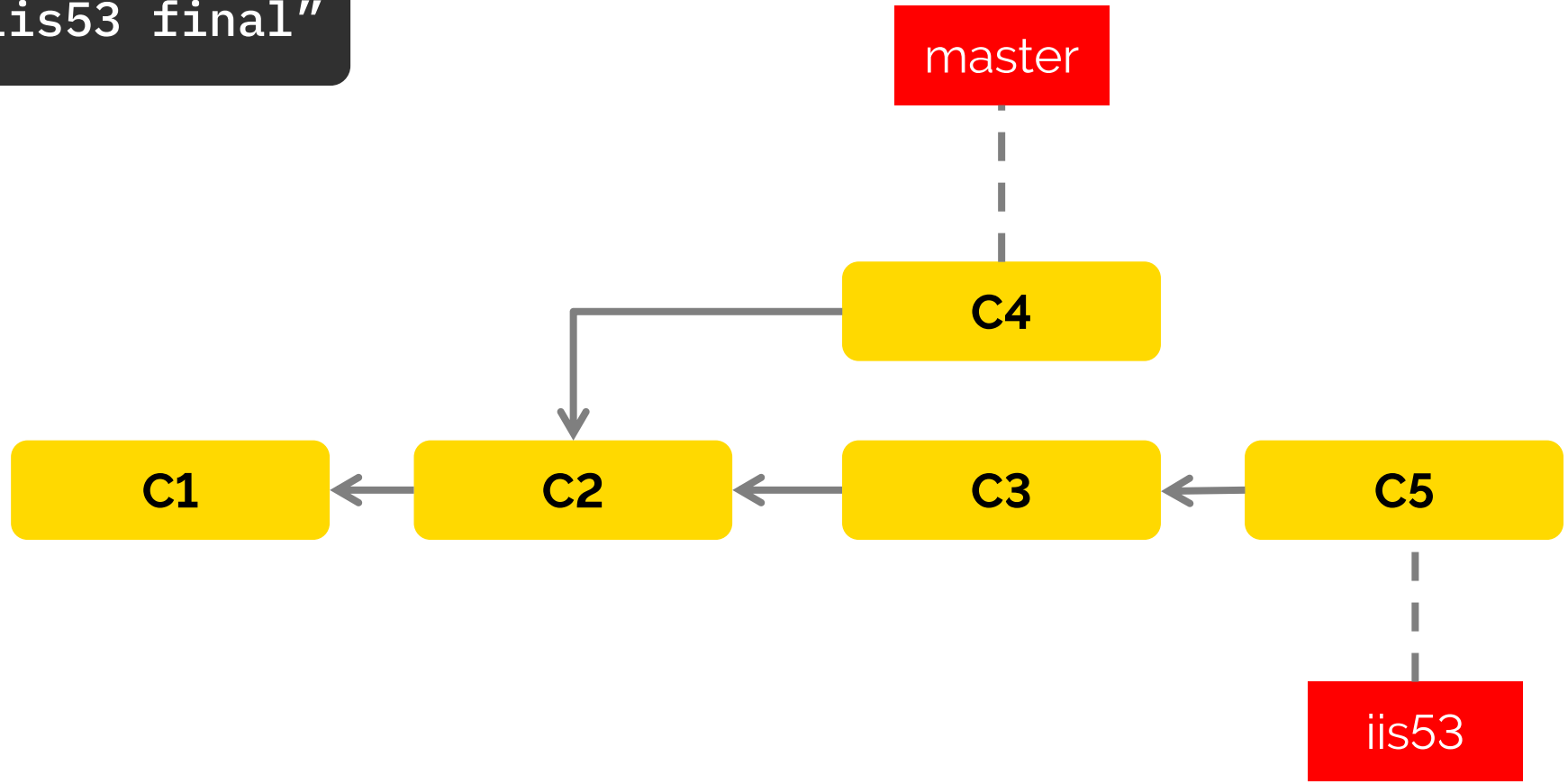


Merge. Расходящиеся ветки



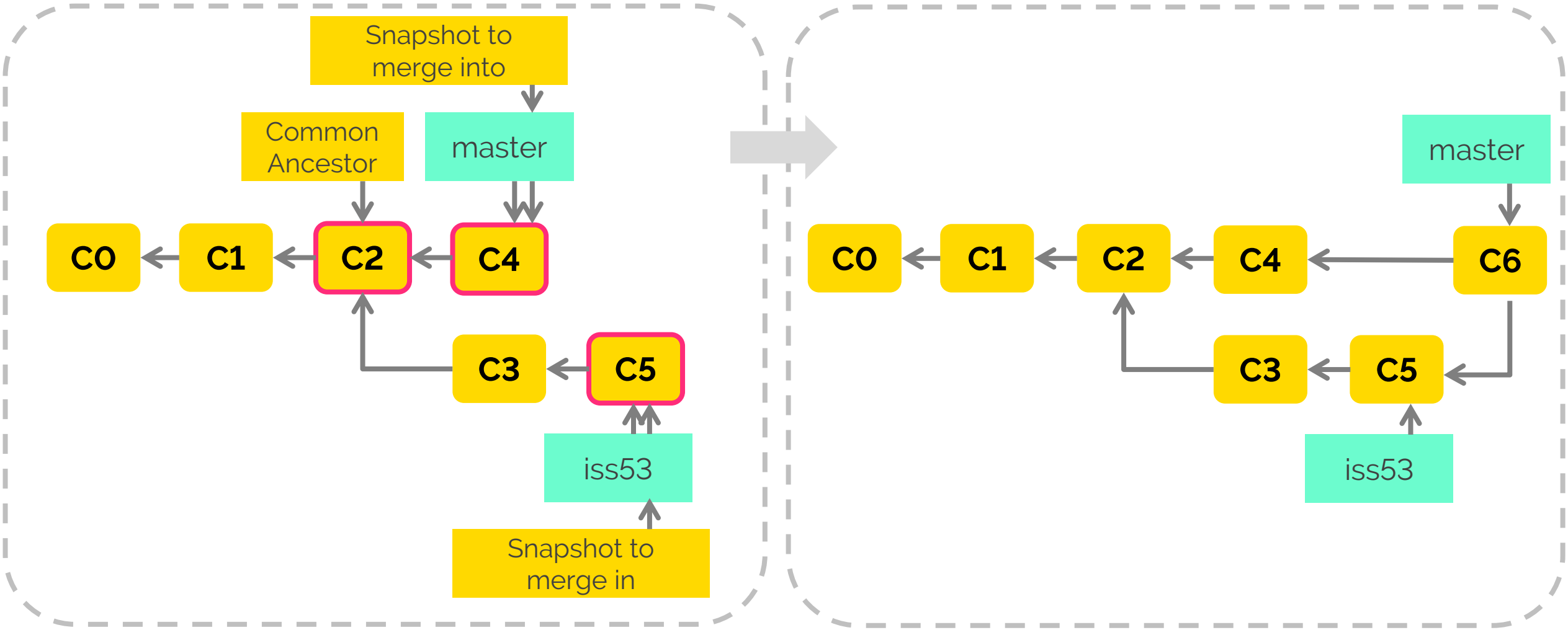
Merge. Расходящиеся ветки

```
$git commit –a –m “iis53 final”
```

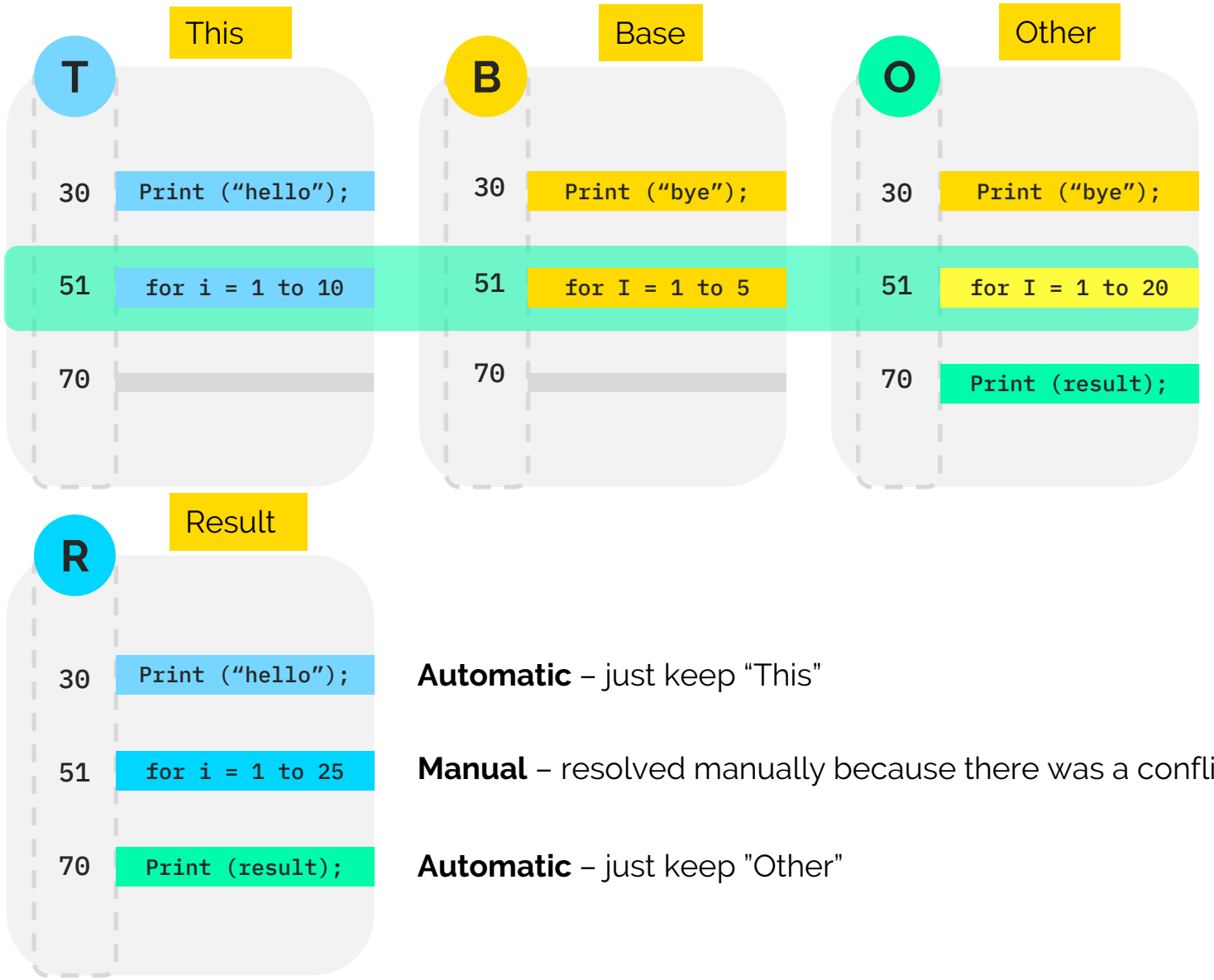


Слияние веток: C6 результирующий снимок

```
git merge iss53
```



3-way merge



Automatic – just keep “This”

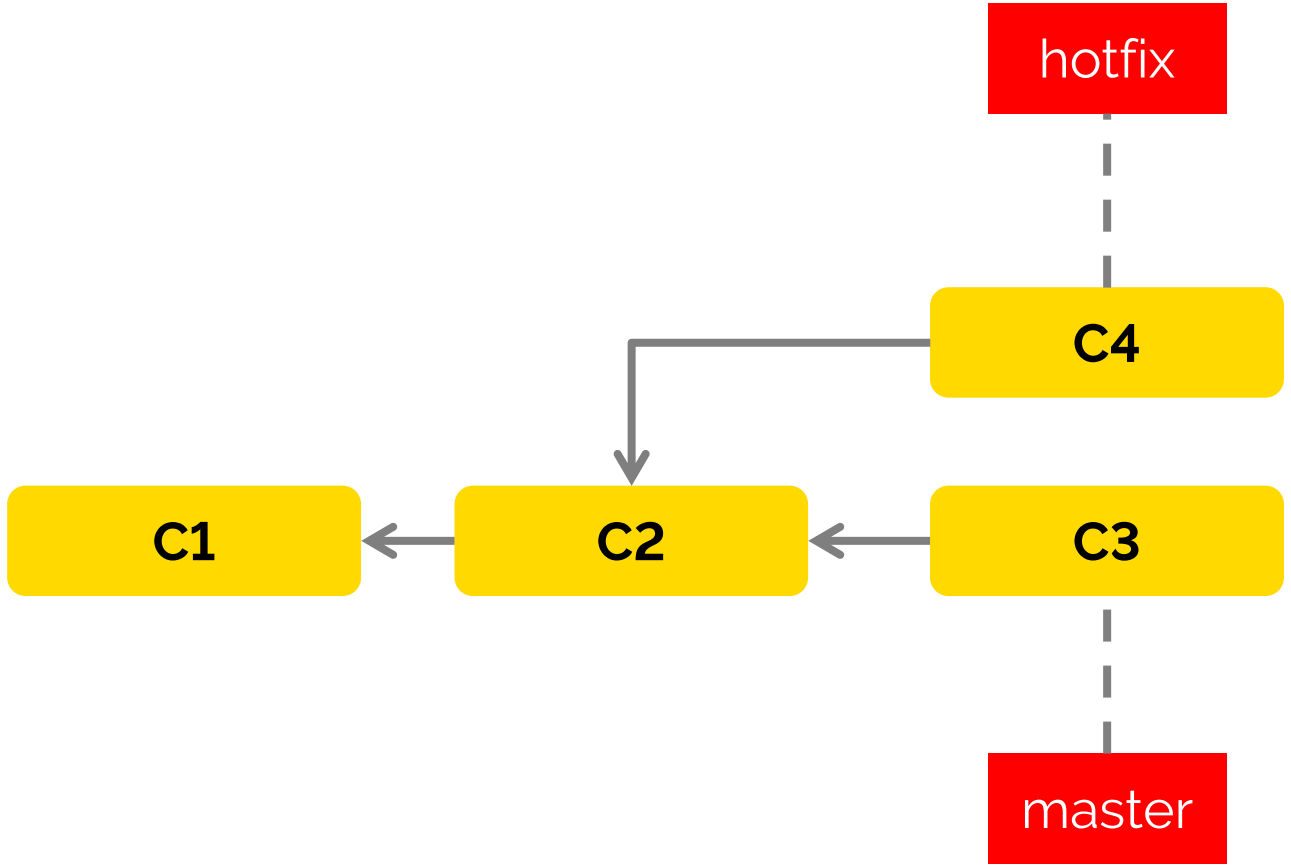
Manual – resolved manually because there was a conflict!

Automatic – just keep ”Other”

Merge

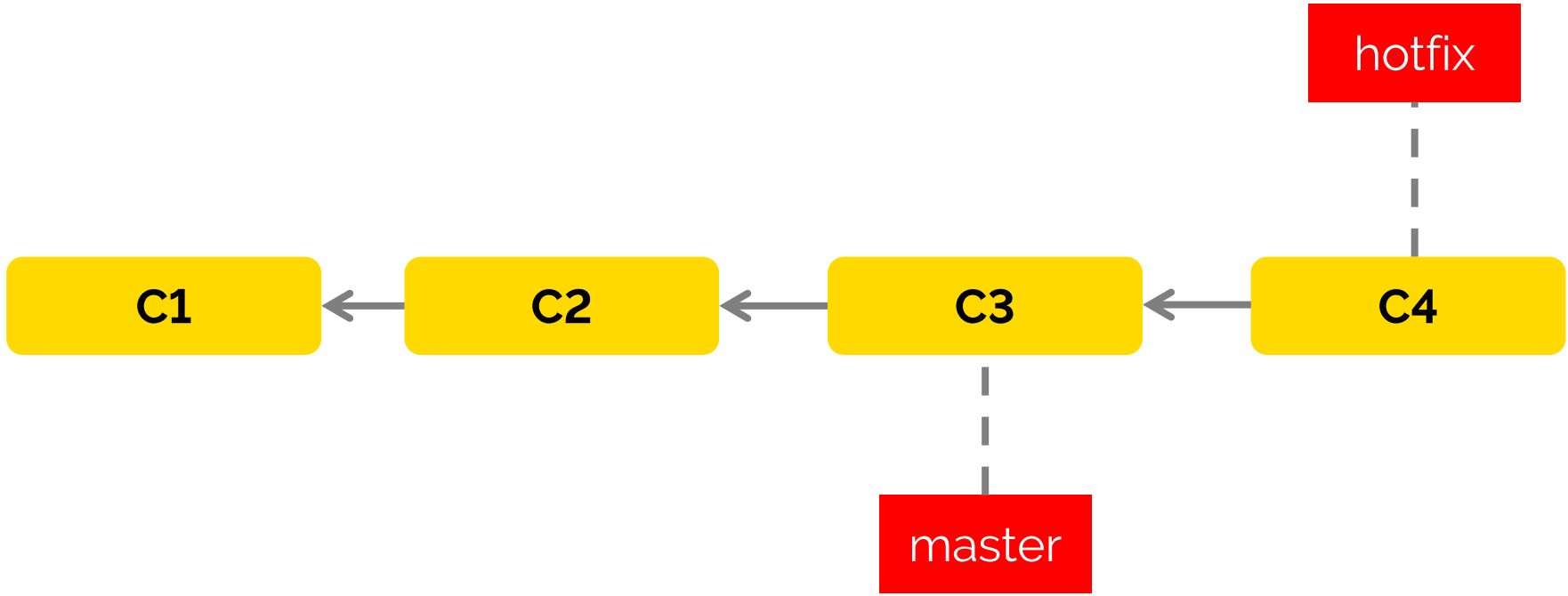
- Разрешаем все конфликты одновременно
- В результате получаем ровно один merge коммит
- История промежуточных коммитов сделанных в ветке hotfix сохраняется
- Не существует способа определить в какой ветке был сделан коммит C4, а в какой C3 кроме как по сообщению в merge коммите C5

Rebase. Расходящиеся ветки

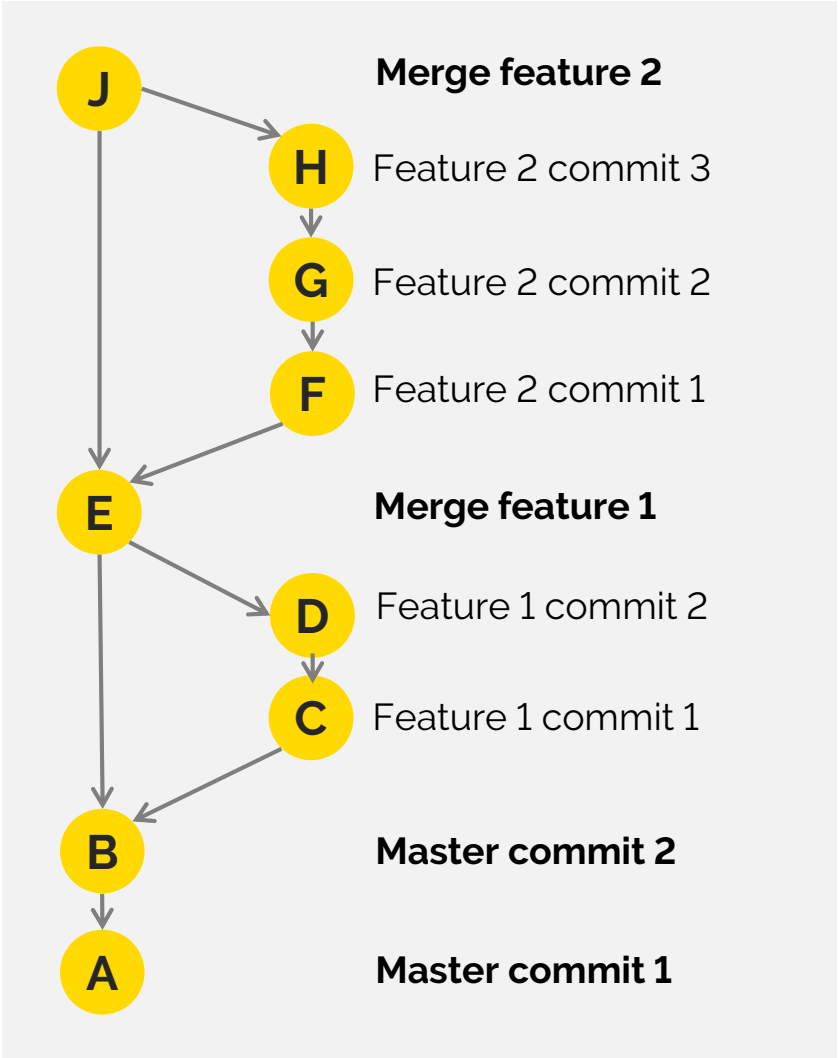


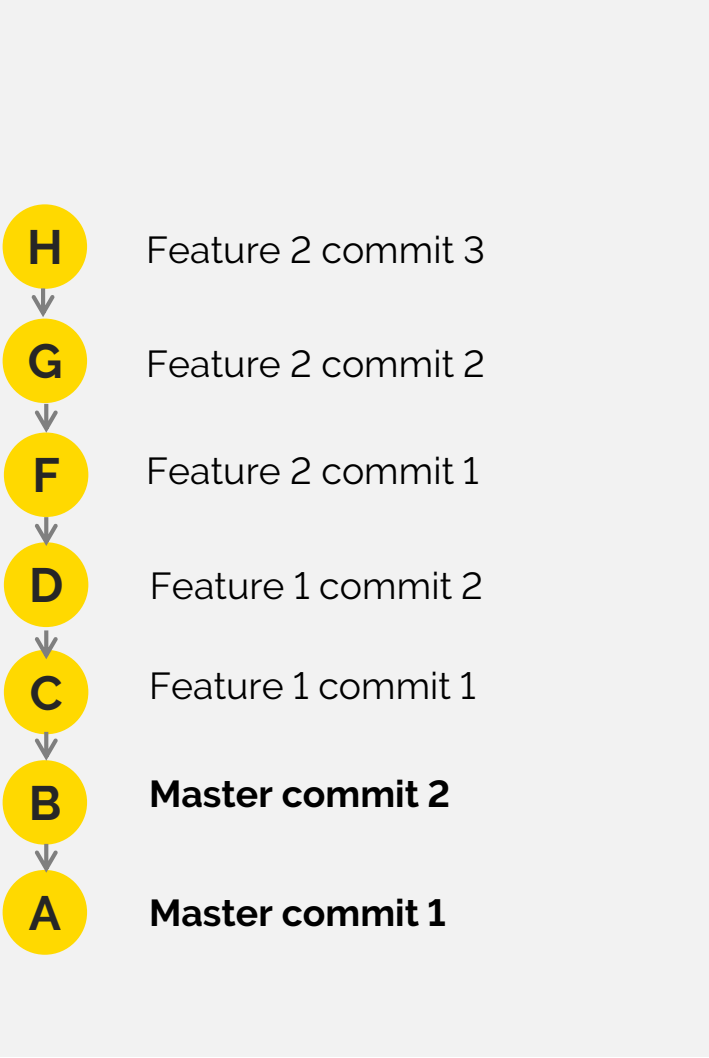
Rebase. расходящиеся ветки

```
$ git checkout hotfix
$ git rebase master
```



Merge VS Rebase



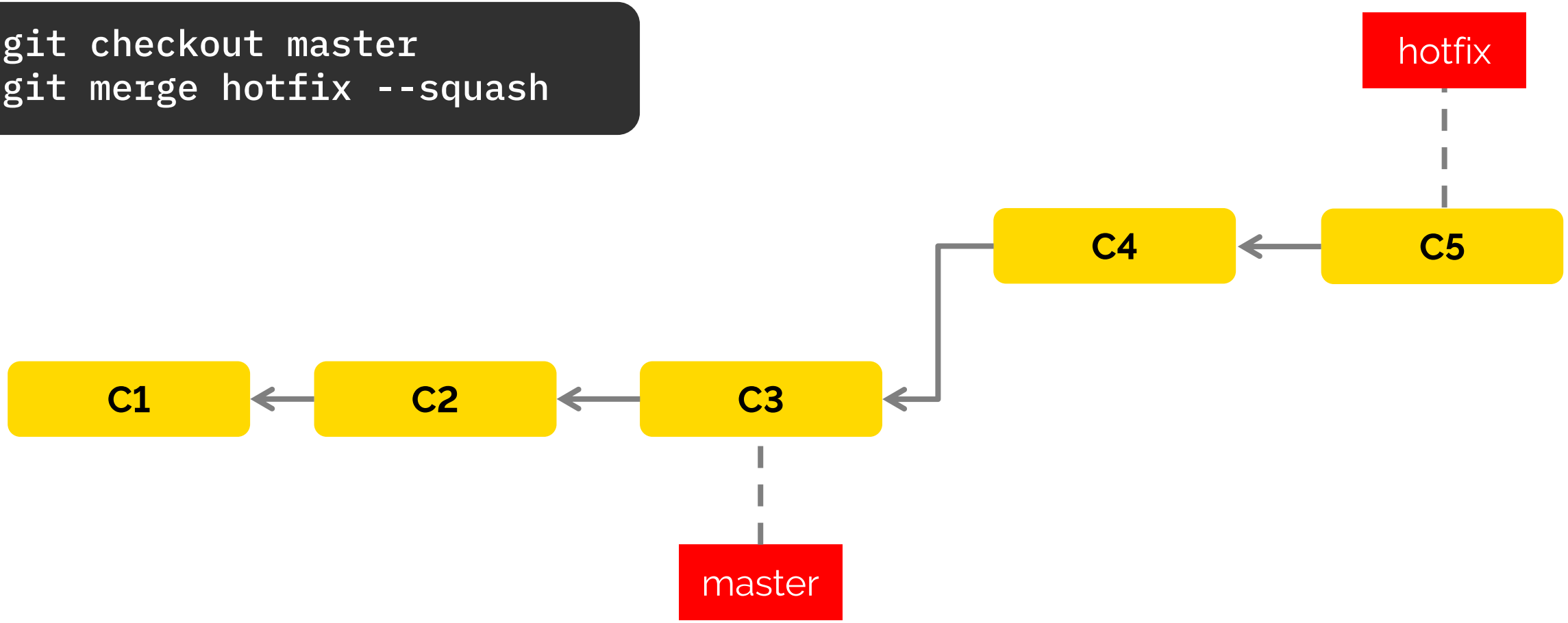


Rebase

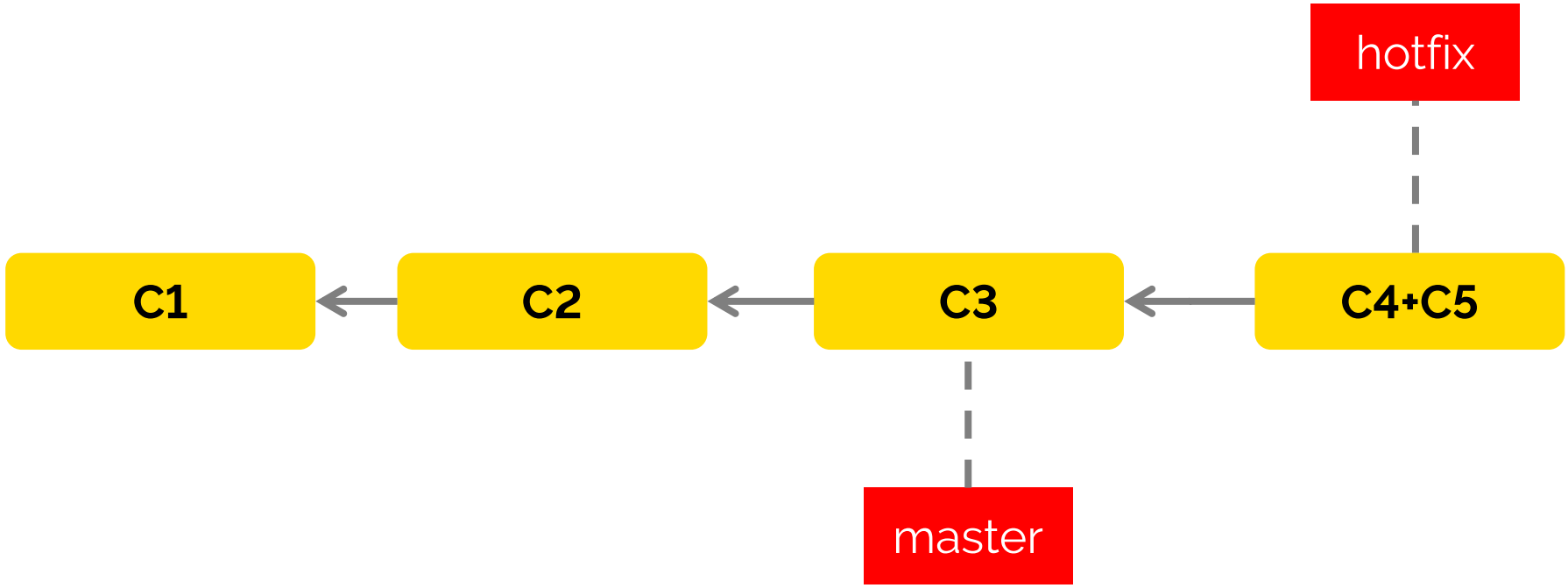
- Rebase меняет всю историю коммитов в ветке до общего предка
- Нельзя ребейзить ветки, которые уже используются другими (без крайней необходимости!)
- Rebase переносит разрешение конфликтов на автора ветки
- Rebase позволяет разрешать конфликты по одному за раз
- Линейная история чище чем нелинейная

Squash merge

```
$ git checkout master
$ git merge hotfix --squash
```

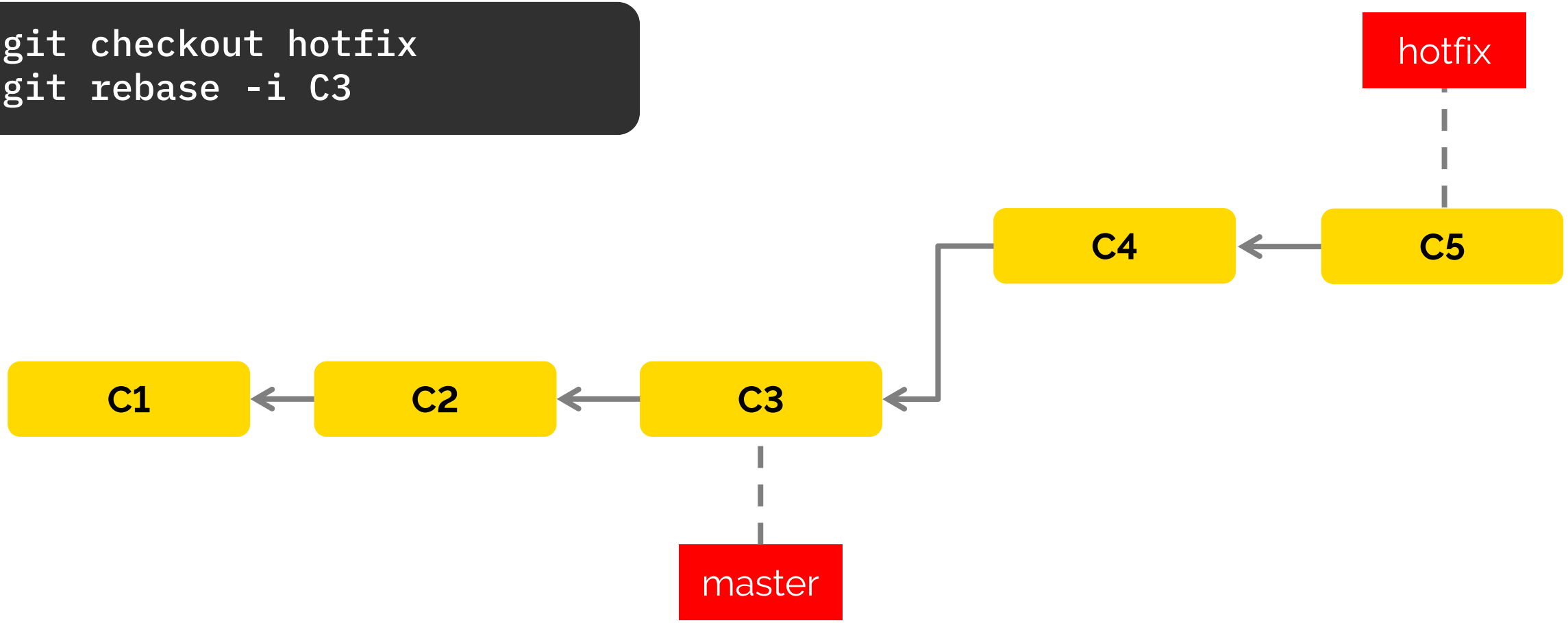


Squash merge



Interactive rebase

```
$ git checkout hotfix
$ git rebase -i C3
```



History grooming

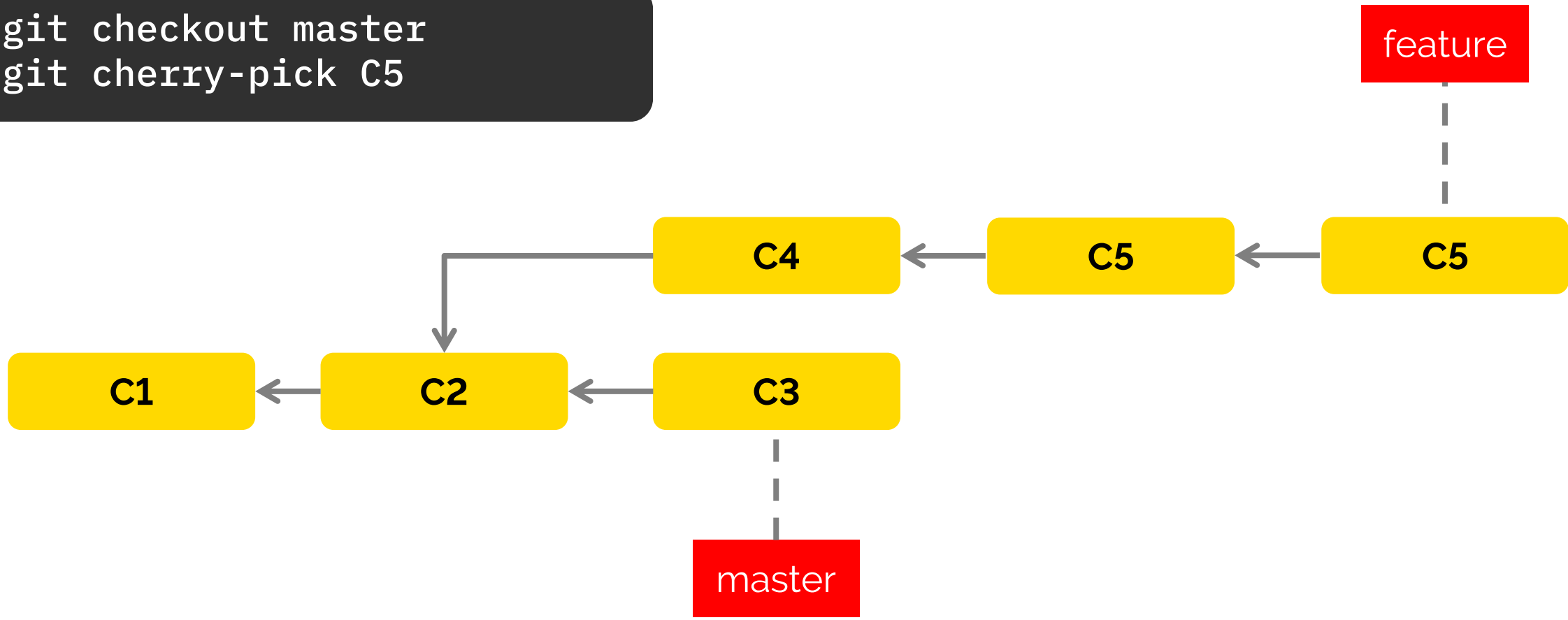
Поскольку git умеет изменять локальную историю, часто используется такой workflow:

1. При разработке изменения коммитятся очень часто и довольно грязно.
2. Как только разработка завершена делается rebase к master
3. Все коммиты сбрасываются и изменения остаются только в рабочей копии
4. С помощью команд add -i и commit изменения группируются по смыслу
5. Уже причесанные изменения заливаются в удаленный репозиторий.

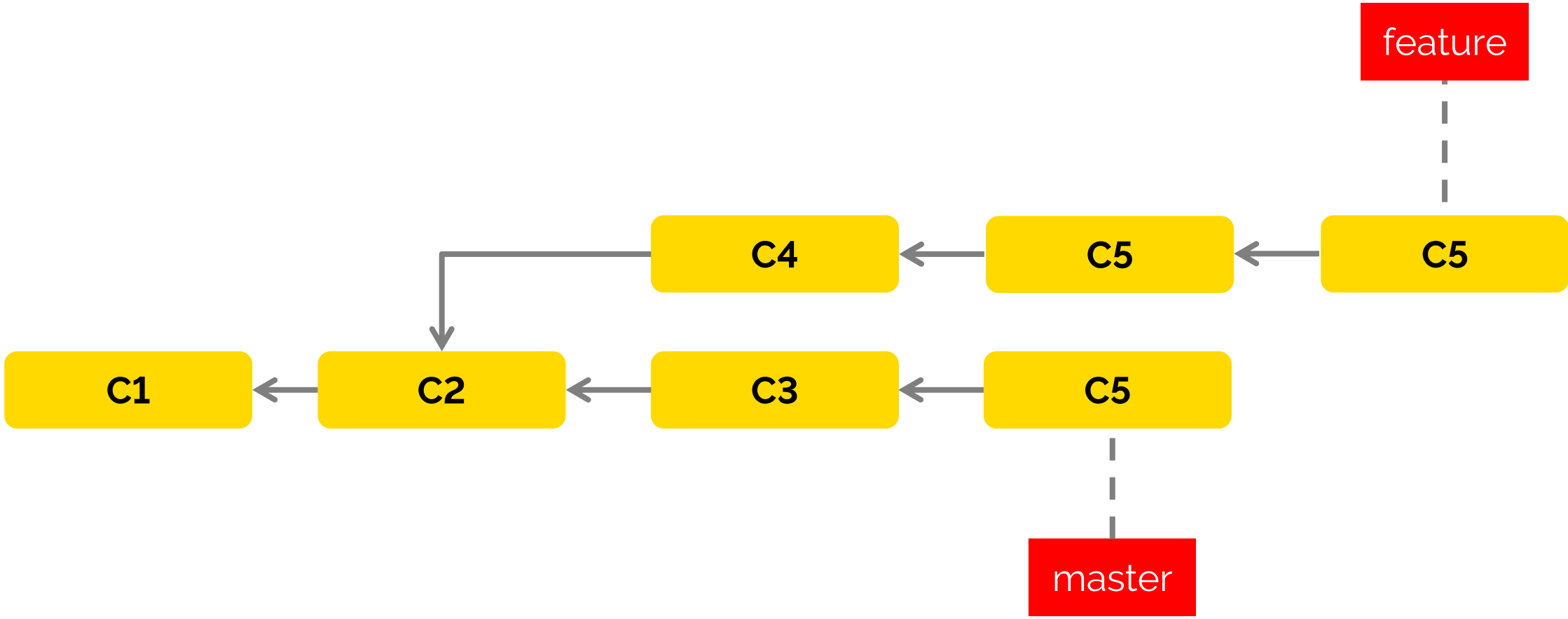
Примерно того же результата можно добиться через
`git merge --squash newFeature && git commit -m 'Your custom commit message'`

Cherry pick

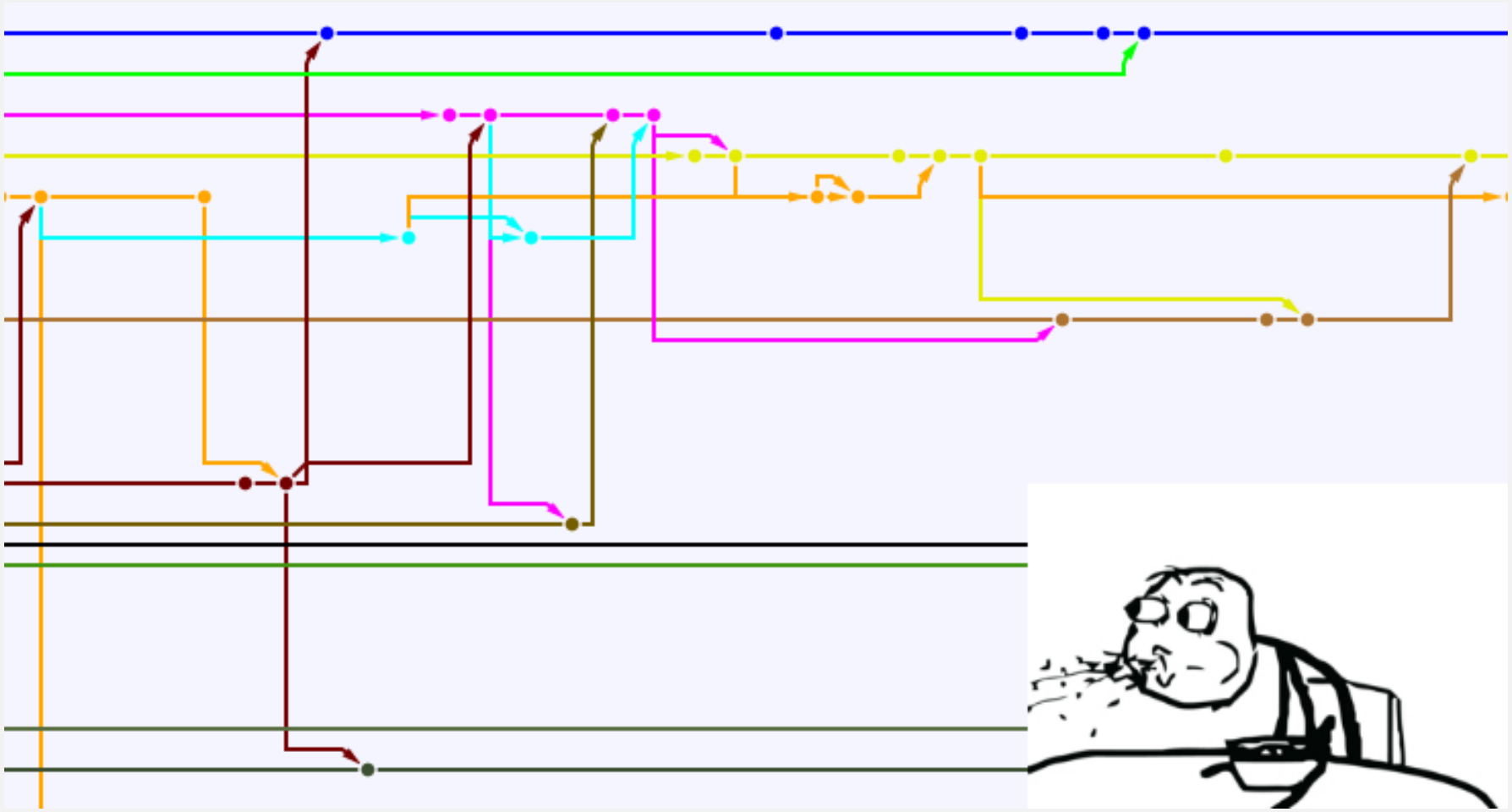
```
$ git checkout master
$ git cherry-pick C5
```



Cherry pick



Merge VS Rebase



Merge VS Rebase

- **Rebase** - **возможность** очистить историю приватных веток от запутанных слияний и множества мелких коммитов

- **Rebase Policy** – **опасность** работы с публичными ветками, другие разработчики вынуждены будут проходить цепочку новых коммитов каждый раз когда они делают git pull

- **Merge Policy** – **простота** использования, **запутанность** истории коммитов

Литература

1. «Pro GIT» aka «The GIT Book», второе издание, 2014 год https://git-scm.com/book/en/v2

2. GitButler (2024) – автоматическое управление изменениями

Вопросы?

Спасибо за внимание