--local-branching-on-the-cheap

Search entire site...

- About
  - Branching and Merging
  - Small and Fast
  - Distributed
  - Data Assurance
  - Staging Area
  - Free and Open Source
  - Trademark
- Documentation
  - Reference
  - Book
  - Videos
  - External Links
- Downloads
  - GUI Clients
  - Logos
- Community

This book is available in English.

Full translation available in

azərbaycan dili,
български език,
Deutsch,
Español,
Français,
Ελληνικά,
日本語,
한국어,
Nederlands,
Русский,

Slovenščina,
Tagalog,
Українська
简体中文,

Partial translations available in

Čeština,
Македонски,
Polski,
Српски,
Ўзбекча,
繁體中文,

Translations started for

Беларуская,
فارسی,
Indonesian,
Italiano,
Bahasa Melayu,
Português (Brasil),
Português (Portugal),
Svenska,
Türkçe.

The source of this book is hosted on GitHub.
Patches, suggestions and comments are welcome.

Chapters ▾

1. **1. Введение**

    1. 1.1 О системе контроля версий
    2. 1.2 Краткая история Git

3. 1.3 Основы Git
4. 1.4 Командная строка
5. 1.5 Установка Git
6. 1.6 Первоначальная настройка Git
7. 1.7 Как получить помощь?
8. 1.8 Заключение

2. **2. Основы Git**

1. 2.1 Создание Git-репозитория
2. 2.2 Запись изменений в репозиторий
3. 2.3 Просмотр истории коммитов
4. 2.4 Операции отмены
5. 2.5 Работа с удалёнными репозиториями
6. 2.6 Работа с метками
7. 2.7 Псевдонимы в Git
8. 2.8 Заключение

3. **3. Ветвление в Git**

1. 3.1 О ветвлении в двух словах
2. 3.2 Основы ветвления и слияния
3. 3.3 Управление ветками
4. 3.4 Работа с ветками
5. 3.5 Удалённые ветки
6. 3.6 Перебазирование
7. 3.7 Заключение

4. **4. Git на сервере**

1. 4.1 Протоколы
2. 4.2 Установка Git на сервер
3. 4.3 Генерация открытого SSH ключа
4. 4.4 Настраиваем сервер
5. 4.5 Git-демон
6. 4.6 Умный HTTP
7. 4.7 GitWeb
8. 4.8 GitLab

9. 4.9 Git-хостинг
10. 4.10 Заключение

5. **5. Распределенный Git**

1. 5.1 Распределенный рабочий процесс
2. 5.2 Участие в проекте
3. 5.3 Сопровождение проекта
4. 5.4 Заключение

1. **6. GitHub**

1. 6.1 Настройка и конфигурация учетной записи
2. 6.2 Внесение собственного вклада в проекты
3. 6.3 Сопровождение проекта
4. 6.4 Управление организацией
5. 6.5 Scripting GitHub
6. 6.6 Заключение

2. **7. Инструменты Git**

1. 7.1 Выбор ревизии
2. 7.2 Интерактивное индексирование
3. 7.3 Прибережение и очистка
4. 7.4 Подпись результатов вашей работы
5. 7.5 Поиск
6. 7.6 Исправление истории
7. 7.7 Раскрытие тайн reset
8. 7.8 Продвинутое слияние
9. 7.9 Rerere
10. 7.10 Обнаружение ошибок с помощью Git
11. 7.11 Подмодули
12. 7.12 Создание пакетов
13. 7.13 Замена
14. 7.14 Хранилище учётных данных
15. 7.15 Заключение

3. **8. Настройка Git**

1. 8.1 Конфигурация Git
2. 8.2 Атрибуты Git
3. 8.3 Хуки в Git
4. 8.4 Пример принудительной политики Git
5. 8.5 Заключение

4. **9. Git и другие системы контроля версий**

1. 9.1 Git как клиент
2. 9.2 Миграция на Git
3. 9.3 Заключение

5. **10. Git изнутри**

1. 10.1 Сантехника и Фарфор
2. 10.2 Объекты Git
3. 10.3 Ссылки в Git
4. 10.4 Pack-файлы
5. 10.5 Спецификации ссылок
6. 10.6 Протоколы передачи данных
7. 10.7 Обслуживание репозитория и восстановление данных
8. 10.8 Переменные окружения
9. 10.9 Заключение

1. **A1. Appendix A: Git в других окружениях**

1. A1.1 Графические интерфейсы
2. A1.2 Git в Visual Studio
3. A1.3 Git в Visual Studio Code
4. A1.4 Git в Eclipse
5. A1.5 Git в IntelliJ / PyCharm / WebStorm / PhpStorm / RubyMine
6. A1.6 Git в Sublime Text
7. A1.7 Git в Bash
8. A1.8 Git в Zsh
9. A1.9 Git в Powershell
10. A1.10 Заключение

2. **A2. Appendix B: Встраивание Git в ваши приложения**

1. A2.1 Git из командной строки
2. A2.2 Libgit2
3. A2.3 JGit
4. A2.4 go-git
5. A2.5 Dulwich

3. **A3. Appendix C: Команды Git**

1. A3.1 Настройка и конфигурация
2. A3.2 Клонирование и создание репозиториев
3. A3.3 Основные команды
4. A3.4 Ветвление и слияния
5. A3.5 Совместная работа и обновление проектов
6. A3.6 Осмотр и сравнение
7. A3.7 Отладка
8. A3.8 Внесение исправлений
9. A3.9 Работа с помощью электронной почты
10. A3.10 Внешние системы
11. A3.11 Администрирование
12. A3.12 Низкоуровневые команды

2nd Edition

3.6 Ветвление в Git - Перебазирование

Перебазирование

В Git есть два способа внести изменения из одной ветки в другую: слияние и перебазирование. В этом разделе вы узнаете, что такое перебазирование, как его осуществлять и в каких случаях этот удивительный инструмент использовать не следует.

Простейшее перебазирование

Если вы вернётесь к более раннему примеру из Основы слияния, вы увидите, что разделили свою работу и сделали коммиты в две разные ветки.

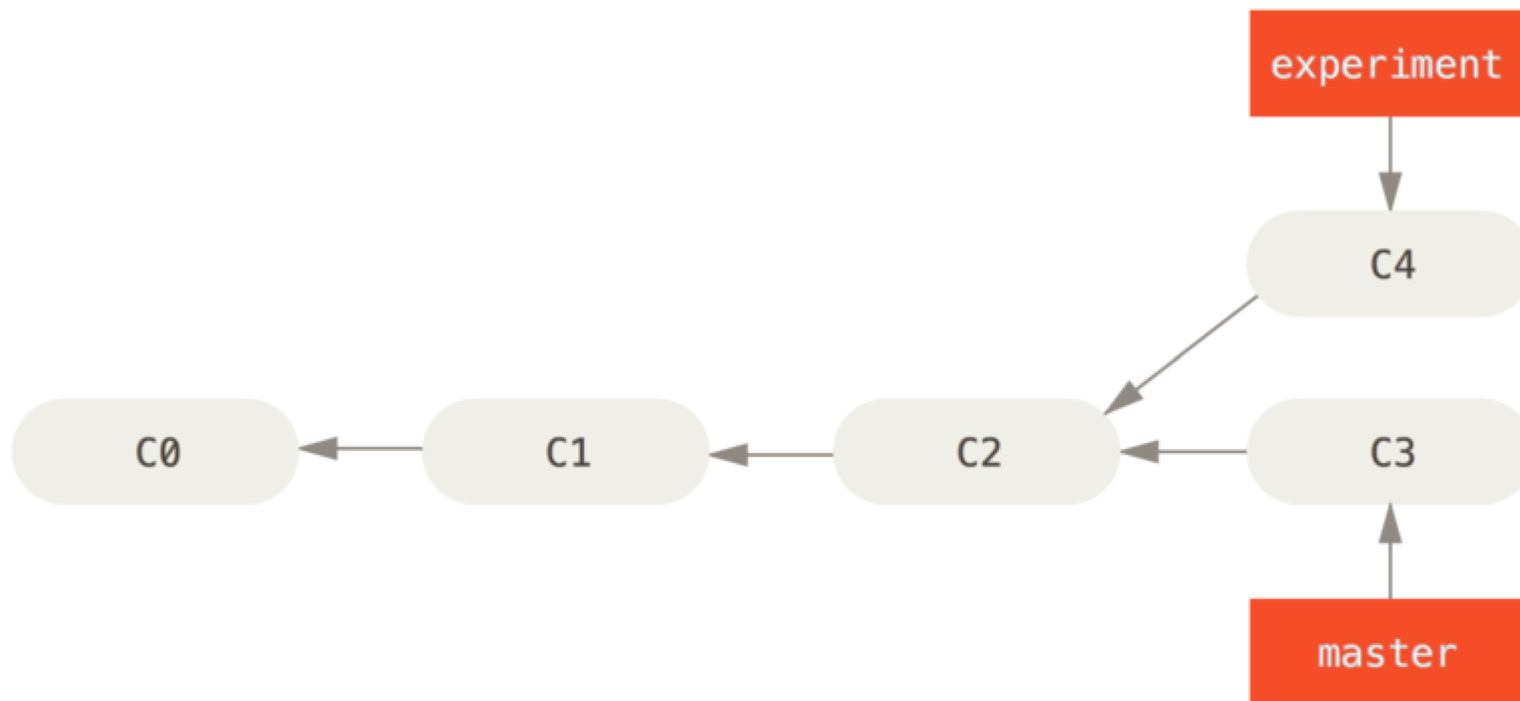

Рисунок 35. История коммитов простого разделения

Как мы выяснили ранее, простейший способ выполнить слияние двух веток — это команда `merge`. Она осуществляет трёхстороннее слияние между двумя последними снимками сливаемых веток (`C3` и `C4`) и самого недавнего общего для этих веток родительского снимка (`C2`), создавая новый снимок (и коммит).

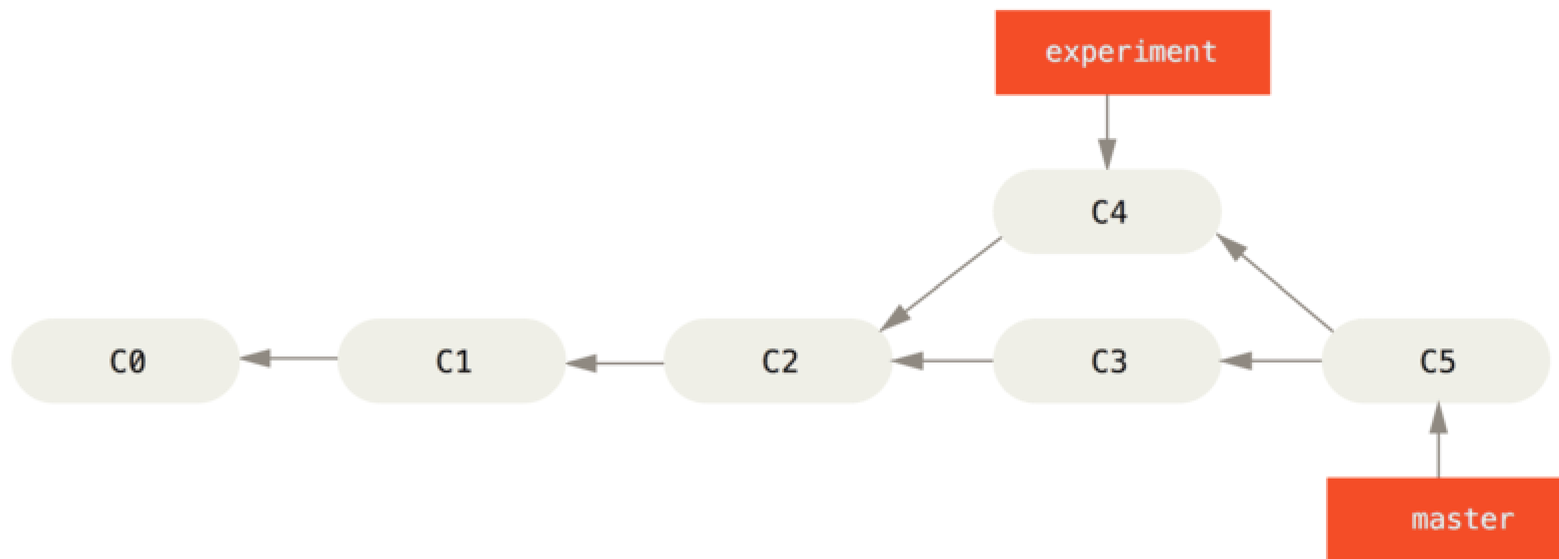

Рисунок 36. Слияние разделённой истории коммитов

Тем не менее есть и другой способ: вы можете взять те изменения, что были представлены в `C4`, и применить их поверх `C3`. В Git это называется *перебазированием*. С помощью команды `rebase` вы можете взять все коммиты из одной ветки и в том же порядке применить их к другой ветке.

В данном примере переключимся на ветку `experiment` и перебазируем её относительно ветки `master` следующим образом:

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

Это работает следующим образом: берётся общий родительский снимок двух веток (текущей, и той, поверх которой вы выполняете перебазирование), определяется дельта каждого коммита текущей ветки и сохраняется во временный файл, текущая ветка устанавливается на последний коммит ветки, поверх которой вы выполняете перебазирование, а затем по очереди применяются дельты из временных файлов.

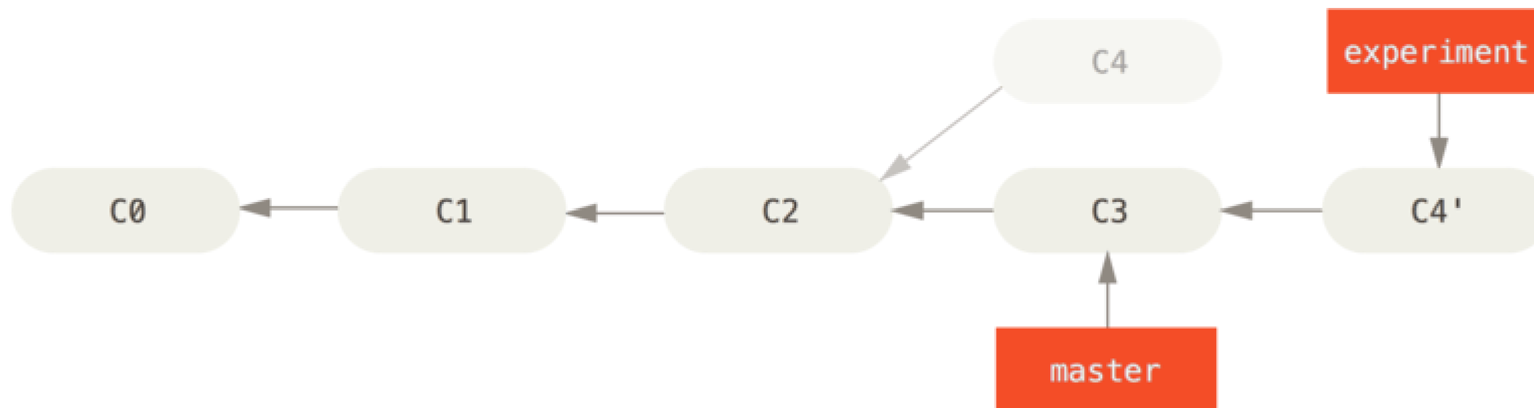

Рисунок 37. Перебазирование изменений из `C4` поверх `C3`

После этого вы можете переключиться обратно на ветку `master` и выполнить слияние перемоткой.

```
$ git checkout master
$ git merge experiment
```

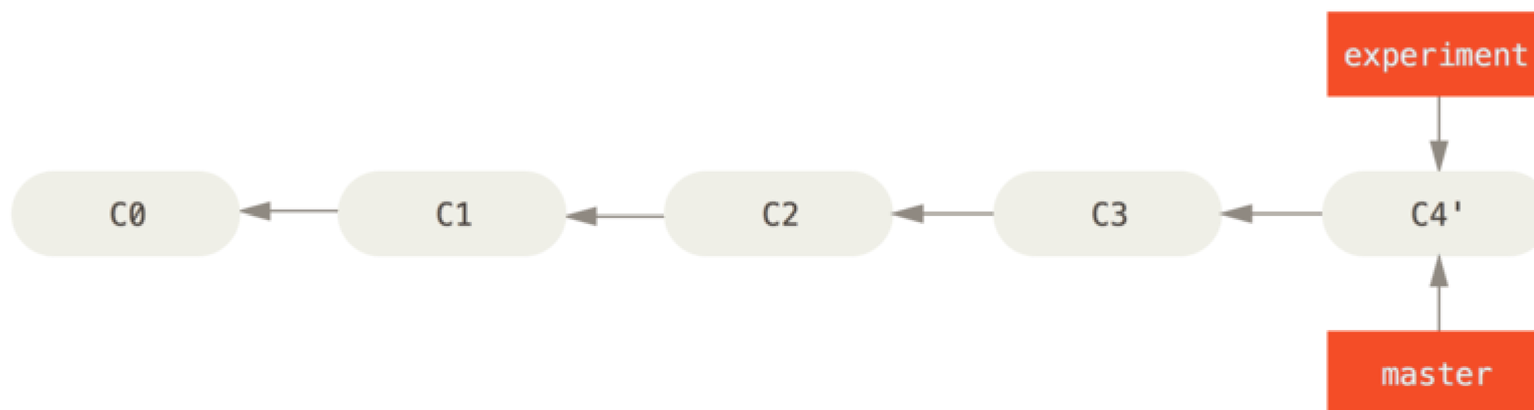

Рисунок 38. Перемотка ветки master

Теперь снимок, на который указывает `C4'` абсолютно такой же, как тот, на который указывал `C5` в [примере с трёхсторонним слиянием](). Нет абсолютно никакой разницы в конечном результате между двумя показанными примерами, но перебазирование делает историю коммитов чище. Если вы взглянете на историю перебазированной ветки, то увидите, что она выглядит абсолютно линейной: будто все операции были выполнены последовательно, даже если изначально они совершались параллельно.

Часто вы будете делать так для уверенности, что ваши коммиты могут быть бесконфликтно слиты в удалённую ветку — возможно, в проекте, куда вы пытаетесь внести вклад, но владельцем которого вы не являетесь. В этом случае вам следует работать в своей ветке и затем

перебазировать вашу работу поверх `origin/master`, когда вы будете готовы отправить свои изменения в основной проект. Тогда владельцу проекта не придётся делать никакой лишней работы — всё решится простой перемоткой или бесконфликтным слиянием.

Учтите, что снимок, на который ссылается ваш последний коммит — является ли он последним коммитом после перебазирования или коммитом слияния после слияния — в обоих случаях это один и тот же снимок, отличаются только истории коммитов. Перебазирование повторяет изменения из одной ветки поверх другой в том порядке, в котором эти изменения были сделаны, в то время как слияние берет две конечные точки и сливает их вместе.

Более интересные перемещения

Также возможно сделать так, чтобы при перебазировании воспроизведение коммитов применялось к совершенно другой ветке. Для примера возьмём История разработки с тематической веткой, ответвлённой от другой тематической ветки. Вы создаёте тематическую ветку `server`, чтобы добавить в проект некоторую функциональность для серверной части, и делаете коммит. Затем вы выполнили ответвление, чтобы сделать изменения для клиентской части, и создали несколько коммитов. Наконец, вы вернулись на ветку `server` и сделали ещё несколько коммитов.

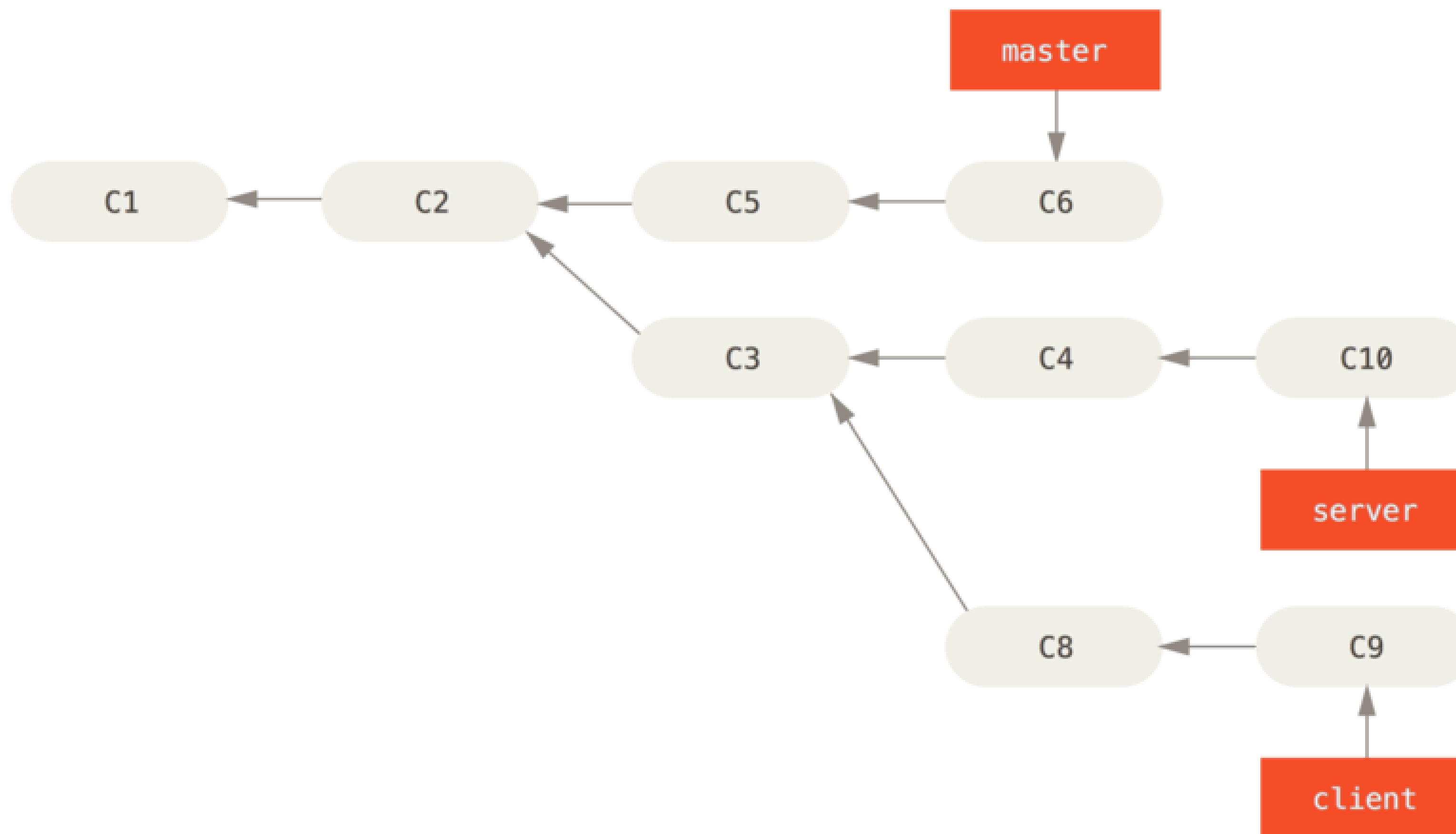

Рисунок 39. История разработки с тематической веткой, ответвлённой от другой тематической ветки

Предположим, вы решили, что хотите внести изменения клиентской части в основную линию разработки для релиза, но при этом не хотите добавлять изменения серверной части до полного тестирования. Вы можете взять изменения из ветки `client`, которых нет в `server` (`C8` и `C9`), и применить их на ветке `master` при помощи опции `--onto` команды `git rebase`:

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

В этой команде говорится: “Переключись на ветку `client`, найди изменения относительно ветки `server` и примени их для ветки `master`”. Несмотря на некоторую сложность этого способа, результат впечатляет.

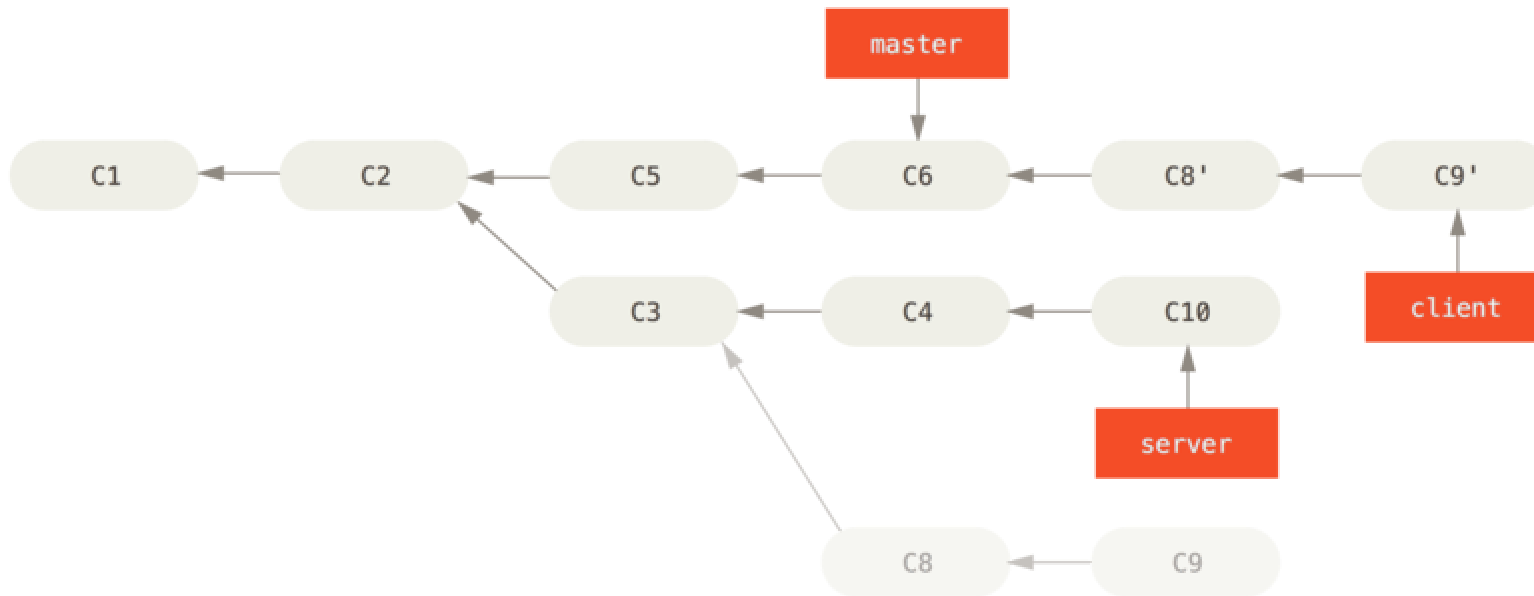

Рисунок 40. Перемещение тематической ветки, ответвлённой от другой тематической ветки

Теперь вы можете выполнить перемотку (fast-forward) для ветки `master` (см [Перемотка ветки `master` для добавления изменений из ветки `client`]()):

```
$ git checkout master
$ git merge client
```

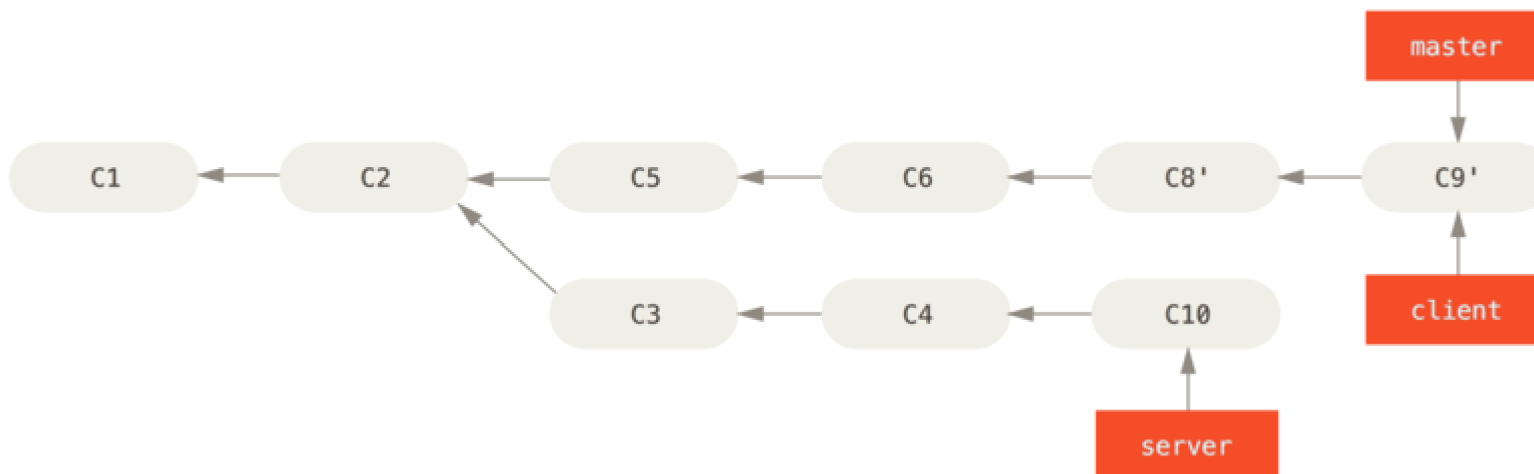

Рисунок 41. Перемотка ветки `master` для добавления изменений из ветки `client`

Представим, что вы решили добавить наработки и из ветки `server`. Вы можете выполнить перебазирование ветки `server` относительно ветки `master` без предварительного переключения на неё при помощи команды `git rebase <basebranch> <topicbranch>`, которая извлечёт тематическую ветку (в данном случае `server`) и применит изменения в ней к базовой ветке (`master`):

```
$ git rebase master server
```

Эта команда добавит изменения ветки `server` в ветку `master`, как это показано на Перебазирование ветки `server` на основании ветки `master`.

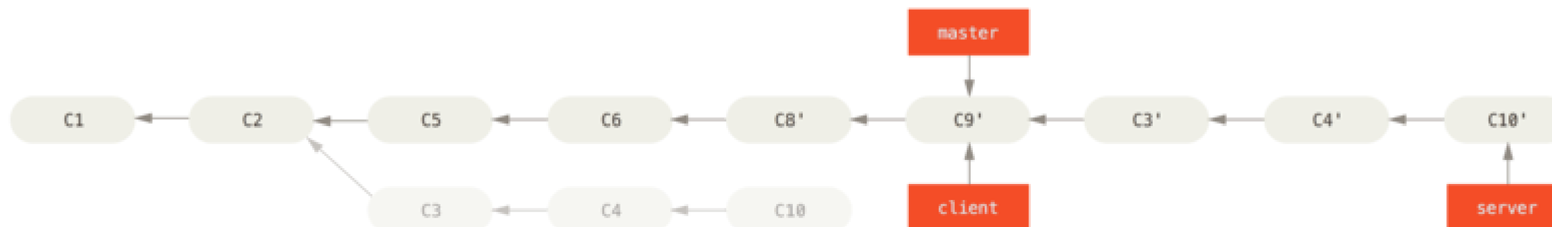

Рисунок 42. Перебазирование ветки `server` на основании ветки `master`

После чего вы сможете выполнить перемотку основной ветки (`master`):

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

Теперь вы можете удалить ветки `client` и `server`, поскольку весь ваш прогресс уже интегрирован и тематические ветки больше не нужны, а полную историю вашего рабочего процесса отражает рисунок Окончательная история коммитов:

```
$ git branch -d client
$ git branch -d server
```

Рисунок 43. Окончательная история коммитов

Опасности перемещения

Но даже перебазирование, при всех своих достоинствах, не лишено недостатков, которые можно выразить одной строчкой:

Не перемещайте коммиты, уже отправленные в публичный репозиторий

Если вы будете придерживаться этого правила, всё будет хорошо. Если не будете, люди возненавидят вас, а ваши друзья и семья будут вас презирать.

Когда вы что-то перемещаете, вы отменяете существующие коммиты и создаёте новые, **похожие** на старые, но являющиеся другими. Если вы куда-нибудь отправляете свои коммиты и другие люди забирают их себе и в дальнейшем основывают на них свою работу, а затем вы переделываете эти коммиты командой `git rebase` и выкладываете их снова, то ваши коллеги будут вынуждены заново выполнять слияние для своих наработок. В итоге, когда вы в очередной раз попытаетесь включить их работу в свою, вы получите путаницу.

Давайте рассмотрим пример того, как перемещение публично доступных наработок может вызвать проблемы. Предположим, вы склонировали репозиторий с сервера и сделали какую-то работу. И ваша история коммитов выглядит так:

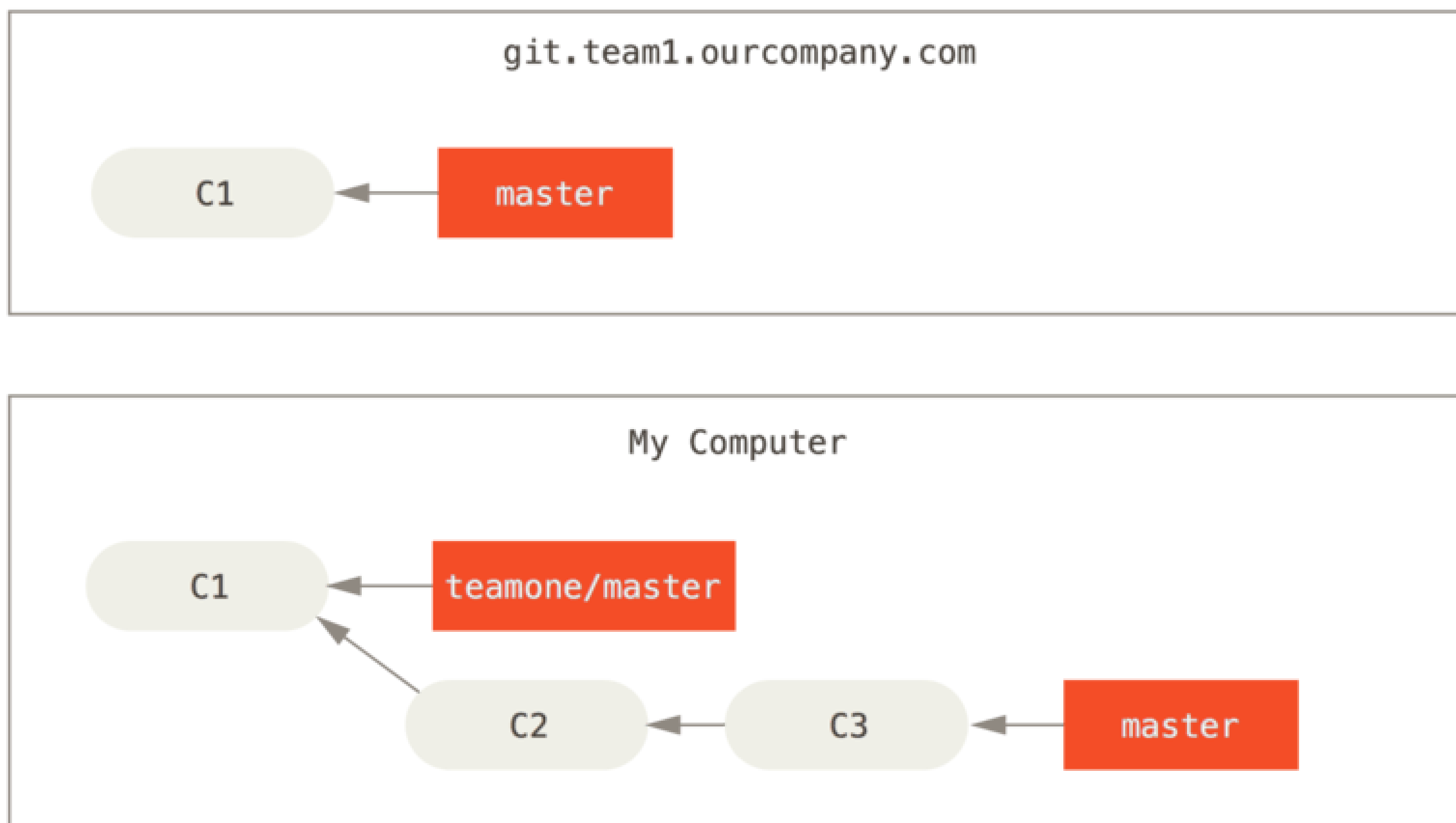

Рисунок 44. Клонирование репозитория и выполнение в нём какой-то работы

Теперь кто-то другой внёс свои изменения, слил их и отправил на сервер. Вы стягиваете их к себе, включая новую удалённую ветку, что изменяет вашу историю следующим образом:

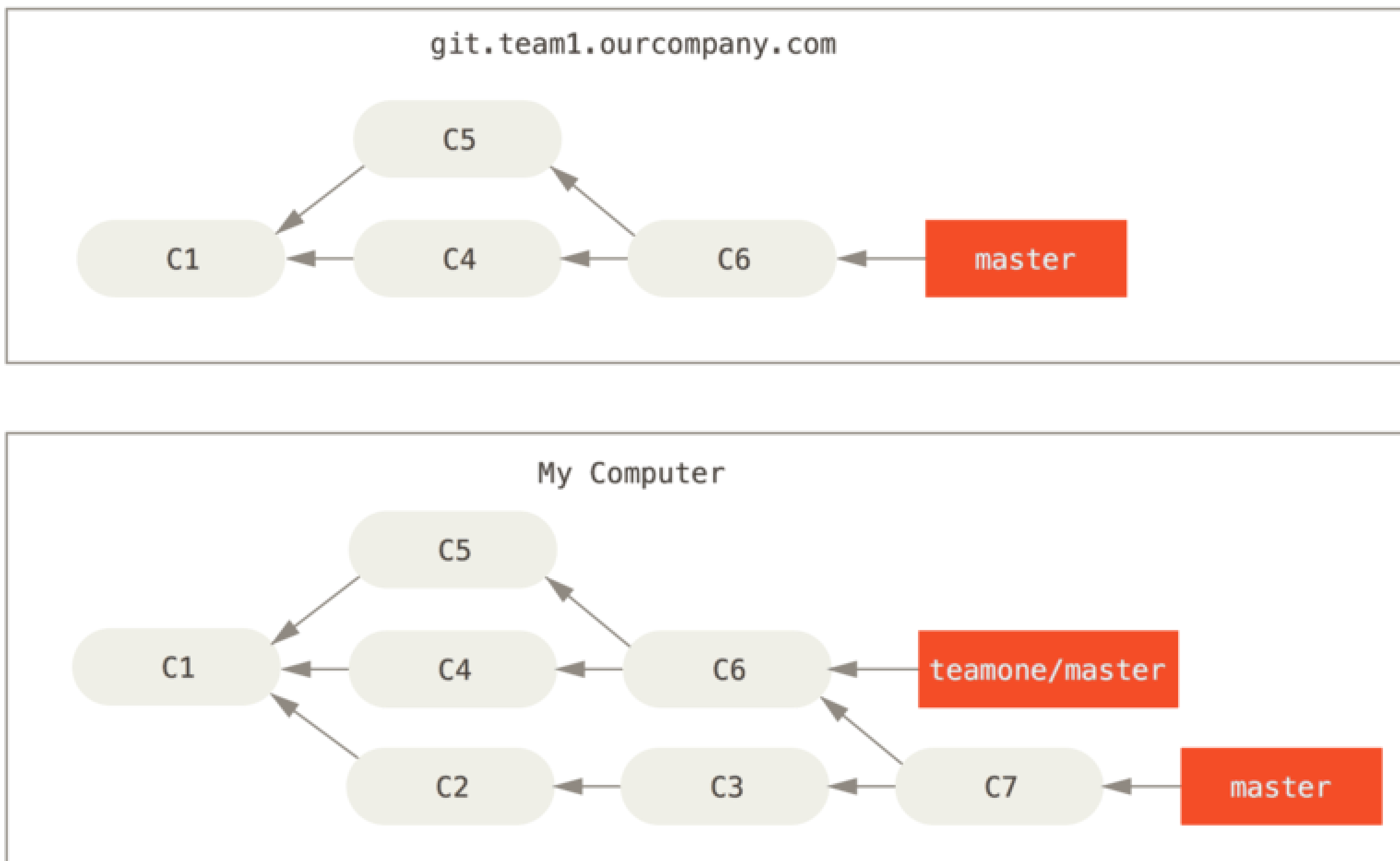

Рисунок 45. Извлекаем ещё коммиты и сливаем их со своей работой

Затем автор коммита слияния решает вернуться назад и перебазировать свою ветку; выполнив `git push --force`, он перезаписывает историю на сервере. При получении изменений с сервера вы получите и новые коммиты.

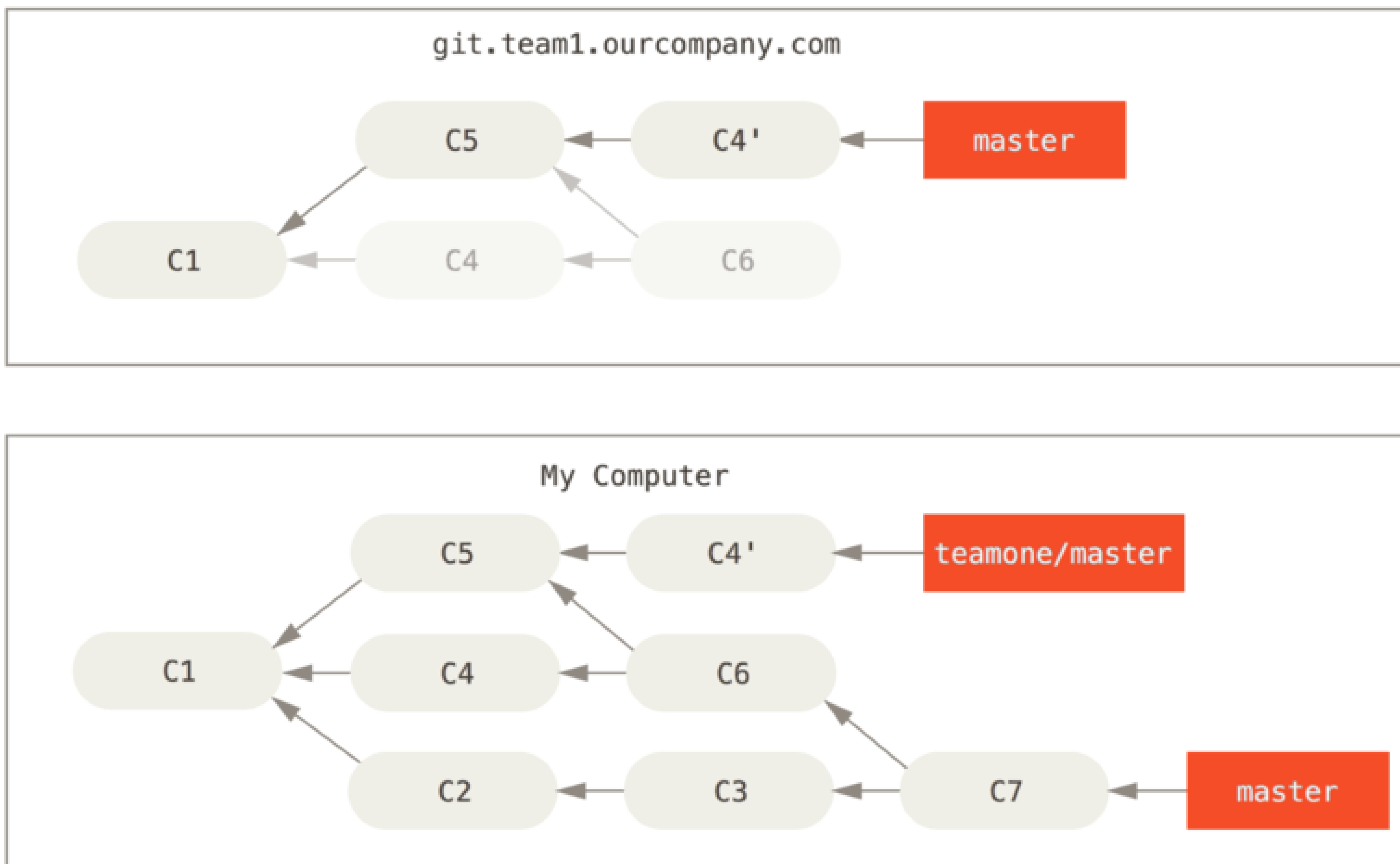

Рисунок 46. Кто-то выложил перебазированные коммиты, отменяя коммиты, на которых основывалась ваша работа

Теперь вы оба в неловком положении. Если вы выполните `git pull`, вы создадите коммит слияния, включающий обе линии истории, и ваш репозиторий будет выглядеть следующим образом:

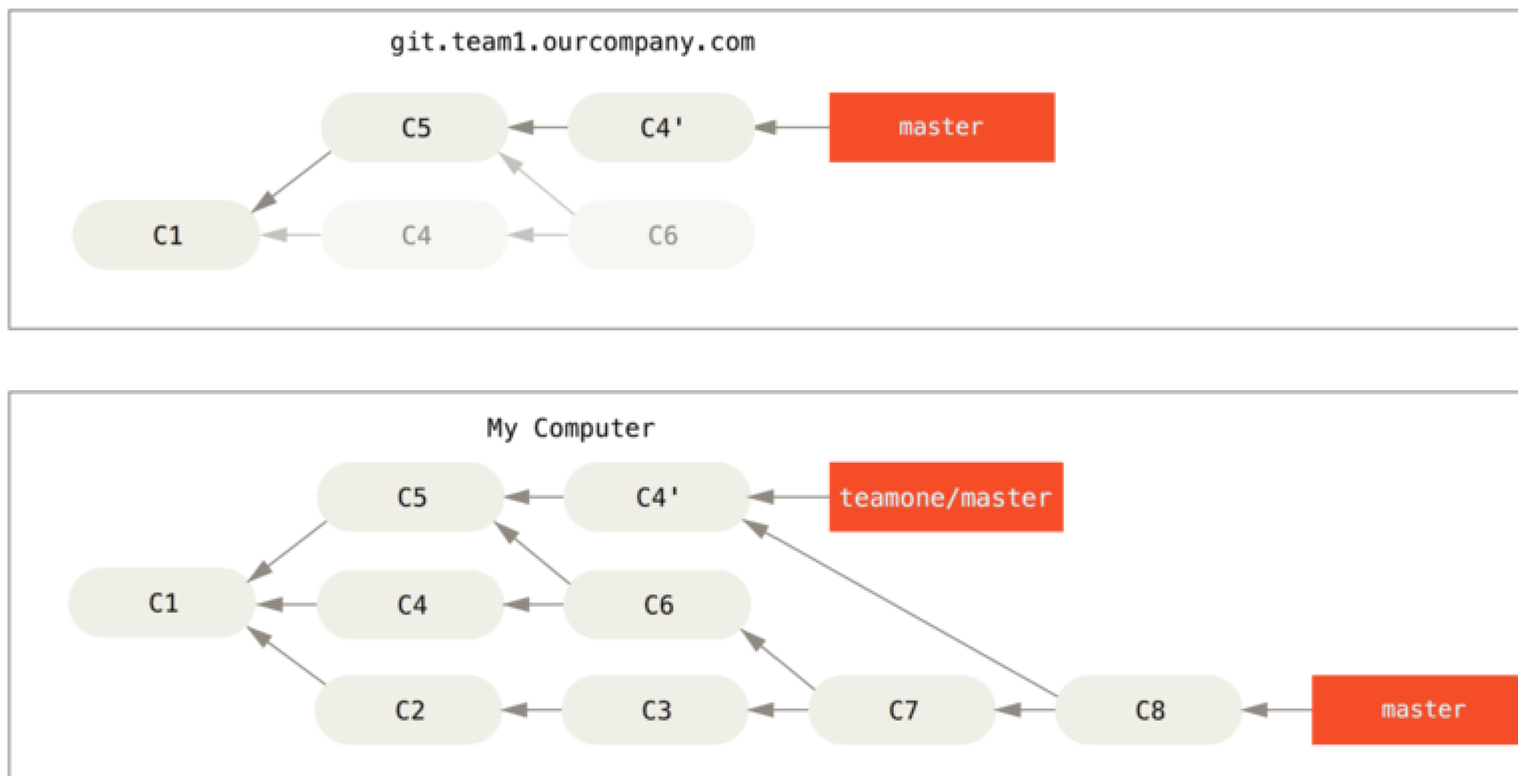

Рисунок 47. Вы снова выполняете слияние для той же самой работы в новый коммит слияния

Если вы посмотрите `git log` в этот момент, вы увидите два коммита с одинаковыми авторами, датой и сообщением, что может сбить с толку. Помимо этого, если вы отправите свою историю на удалённый сервер в таком состоянии, вы вернёте все эти перебазированные коммиты на сервер, что ещё больше всех запутает. Логично предположить, что разработчик не хочет, чтобы `C4` и `C6` были в истории, и именно поэтому она перебазируется в первую очередь.

Меняя базу, меняй основание

Если вы попали в такую ситуацию, у Git есть особая магия чтобы вам помочь. Если кто-то в вашей команде форсирует отправку изменений на сервер, переписывающих работу, на которых базировалась ваша работа, то ваша задача будет состоять в определении того, что именно было ваше, а что было переписано **ими**.

Оказывается, что помимо контрольной суммы коммита SHA-1, Git также вычисляет контрольную сумму отдельно для патча, входящего в этот коммит. Это контрольная сумма называется “patch-id”.

Если вы скачаете перезаписанную историю и перебазируете её поверх новых коммитов вашего коллеги, в большинстве случаев Git успешно определит, какие именно изменения были внесены вами, и применит их поверх новой ветки.

К примеру, если в предыдущем сценарии вместо слияния в Кто-то выложил перебазированные коммиты, отменяя коммиты, на которых основывалась ваша работа мы выполним `git rebase teamone/master`, Git будет:

- Определять, какая работа уникальна для вашей ветки (C2, C3, C4, C6, C7)
- Определять, какие коммиты не были коммитами слияния (C2, C3, C4)
- Определять, что не было перезаписано в основной ветке (только C2 и C3, поскольку C4 — это тот же патч, что и C4')
- Применять эти коммиты к ветке `teamone/master`

Таким образом, вместо результата, который мы можем наблюдать на Вы снова выполняете слияние для той же самой работы в новый коммит слияния, у нас получилось бы что-то вроде Перемещение в начало force-pushed перемещённой работы.

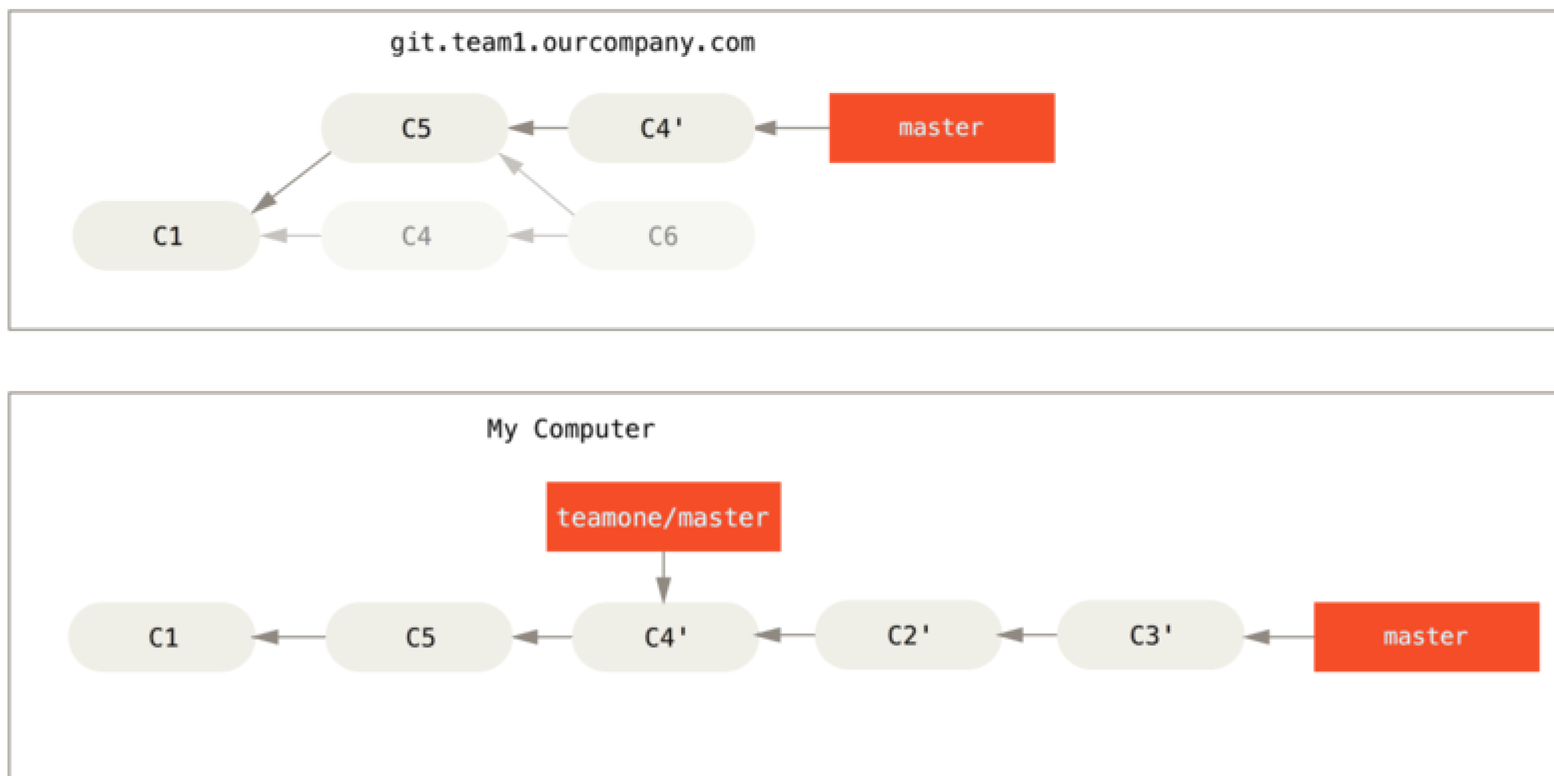

Рисунок 48. Перемещение в начало force-pushed перемещённой работы

Это возможно, если `C4` и `C4'` фактически являются одним и тем же патчем, который был сделан вашим коллегой. В противном случае `rebase` не сможет определить дубликат и создаст ещё один патч, подобный C4 (который с большой вероятностью не удастся применить чисто, поскольку в нём уже присутствуют некоторые изменения).

Вы можете это упростить, применив `git pull --rebase` вместо обычного `git pull`. Или сделать это вручную с помощью `git fetch`, а затем `git rebase teamone/master`.

Если вы используете `git pull` и хотите использовать `--rebase` по умолчанию, вы можете установить соответствующее значение конфигурации `pull.rebase` с помощью команды `git config --global pull.rebase true`.

Если вы рассматриваете перебазирование как способ наведения порядка и работаете с коммитами локально до их отправки или ваши коммиты никогда не будут доступны публично — у вас всё будет хорошо. Однако, если вы перемещаете коммиты, отправленные в публичный репозиторий, и есть вероятность, что работа некоторых людей основывается на этих коммитах, то ваши действия могут вызвать существенные проблемы, а вы — вызвать презрение вашей команды.

Если в какой-то момент вы или ваш коллега находите необходимость в этом, убедитесь, что все знают, как применять команду `git pull --rebase` для минимизации последствий от подобных действий.

Перемещение vs. Слияние

Теперь, когда вы увидели перемещение и слияние в действии, вы можете задаться вопросом, что из них лучше. Прежде чем ответить на этот вопрос, давайте вернёмся немного назад и поговорим о том, что означает история.

Одна из точек зрения заключается в том, что история коммитов в вашем репозитории — это **запись того, что на самом деле произошло**. Это исторический документ, ценный сам по себе, и его нельзя подделывать. С этой точки зрения изменение истории коммитов практически кощунственно; вы *лжёте* о том, что на самом деле произошло. Но что, если произошла путаница в коммитах слияния? Если это случается, репозиторий должен сохранить это для потомков.

Противоположная точка зрения заключается в том, что история коммитов — это **история того, как был сделан ваш проект**. Вы не публикуете первый черновик книги или инструкции по поддержке вашего программного обеспечения, так как это нуждается в тщательном редактировании. Сторонники этого лагеря считают использование инструментов `rebase` и `filter-branch` способом рассказать историю проекта наилучшим образом для будущих читателей.

Теперь к вопросу о том, что лучше — слияние или перебазирование: надеюсь, вы видите, что это не так просто. Git — мощный инструмент, позволяющий вам делать многое с вашей историей, однако каждая команда и каждый проект индивидуален. Теперь, когда вы знаете, как работают оба эти приёма, выбор — какой из них будет лучше в вашей ситуации — зависит от вас.

При этом, вы можете взять лучшее от обоих миров: использовать перебазирование для наведения порядка в истории ваших локальных изменений, но никогда не применять его для уже отправленных куда-нибудь изменений.

prev | next

About this site
Patches, suggestions, and comments are welcome.
Git is a member of Software Freedom Conservancy