Дорожная карта

0. Приступаем
1. Введение в Git
2. Начало работы с Git
3. Просмотр истории
4. Ветвление
5. Слияние
6. Отмена изменений
7. Рабочий процесс 👈

Дорожная карта (продолжение)

8. Работа в команде
9. Метки
10. Последние штрихи
11. Завершаем

Стиль git message

1. Отделяйте заголовок и тело сообщения пустой строкой
2. Ограничивайте заголовок до 50 символов
3. Начинайте заголовок большой буквы
4. В конце заголовка не ставьте знаки припинания
5. В заголовке используйте повелительное наклонение
6. Ограничивайте строки тела сообщения 72 символами
7. В теле сообщения описываете *Что* и *Почему*, но не *Как*

Пример git message

```
Краткое (50 символов или меньше) описание изменений

Текст более детального описания, если необходим. Старайтесь
не превышать длину строки в 72 символа. В некоторых случаях
первая строка подразумевается как тема письма, а всё остальное -
тело письма. Пустая строка, разделяющая сообщение, критически важна
(если существует детальное описание) для некоторых команд Git.

Последующие параграфы должны отделяться пустыми строками.

- Списки тоже подходят
- Обычно, элементы списка обозначаются с помощью тире или звёздочки,
  предваряются одиночным пробелом, а разделяются пустой строкой, но
  соглашения могут отличаться
```

Создание псевдонимов

```
$ git config --global alias.co checkout
$ git config --global alias.unstage "reset HEAD --"
$ git config --global alias.su "submodule update --init --recursive"
$ git config --global alias.lg = "log --pretty=oneline --decorate=auto --graph --abbrev-commit"
```

```
[alias]
  co = checkout
  unstage = reset HEAD --
  su = submodule update --init --recursive
  lg = log --pretty=oneline --decorate=auto --graph --abbrev-commit
```

```
$ git unstage fileA
$ git reset HEAD -- fileA
```

Игнорирование файлов

`.gitignore` скрывает файлы, которые не должны попасть в репозиторий

К шаблонам в файле `.gitignore` применяются следующие правила:

- Пустые строки, а также строки, начинающиеся с `#`, игнорируются.
- Можно использовать стандартные glob шаблоны.
- Можно начать шаблон символом слэша `/` чтобы избежать рекурсии.
- Можно заканчивать шаблон символом слэша `/` для указания каталога.
- Можно инвертировать шаблон, использовав восклицательный знак `!` в качестве первого символа.

Игнорирование файлов (продолжение)

```
$ cat .gitignore
# no .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the root TODO file, not subdir/TODO
/TODO

# ignore all files in the build/ directory
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .txt files in the doc/ directory
doc/**/*.txt
```

Игнорирование файлов (продолжение)

- Лучше подготовить `.gitignore` заранее, чтобы избежать случайного добавления ненужных файлов.
- Существует много заготовок в Интернете, например, https://github.com/github/gitignore.

Игнорирование файлов (продолжение)

Glob-шаблоны представляют собой упрощённые регулярные выражения, используемые командными интерпретаторами:

- символ `*` соответствует 0 или более символам;
- последовательность `[abc]` - любому символу из указанных в скобках (в данном примере `a`, `b` или `c`);
- знак вопроса `?` соответствует одному символу;
- и квадратные скобки, в которые заключены символы, разделённые дефисом `[0-9]`, соответствуют любому символу из интервала (в данном случае от `0` до `9`).
- Можно использовать две звёздочки, чтобы указать на вложенные директории: `a/**/z` соответствует `a/z`, `a/b/z`, `a/b/c/z`, и так далее.

Чистка репозитория

Время от времени Git выполняет автоматическую сборку мусора. Можно запустить сборку мусора вручную:

```
$ git gc --auto
```

Сборщик мусора:

- переупаковывает и оптимизирует базу Git;
- удаляет недостижимые объекты, хранящиеся дольше нескольких месяцев.

Чистка рабочего каталога

Команда `git clean` удаляет только неотслеживаемые файлы, которые не добавлены в список игнорируемых.
Любой файл, который соответствует шаблону в `.gitignore`, удалён не будет.

- Опция `-x` позволяет удалить файлы из `.gitignore`.
- Опция `-d` позволяет также удалить каталоги.
- Опция `-n` или `--dry-run` позволяет включает режим имитации удаления.

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

Чистка рабочего каталога (продолжение)

- Опция `-f` подтверждает удаление:

```
$ git clean -d
fatal: clean.requireForce defaults to true and neither -i, -n, nor -f given; refusing to clean

$ git clean -d -f
Removing test.o
Removing tmp/
```

- Опция `-i` задействует интерактивный режим:

```
$ git clean -x -i
Would remove the following items:
  build.TMP  test.o
*** Commands ***
    1: clean                2: filter by pattern
    3: select by numbers    4: ask each
    5: quit                 6: help
What now>
```

Прятание изменений (Прибрежение изменений; stash)

```
$ git stash list
$ git stash [push]
$ git stash show
$ git stash pop
$ git stash apply
$ git stash apply stash@{2}
```

Создание ветки из спрятанных изменений:

```
$ git stash branch <branch>
```

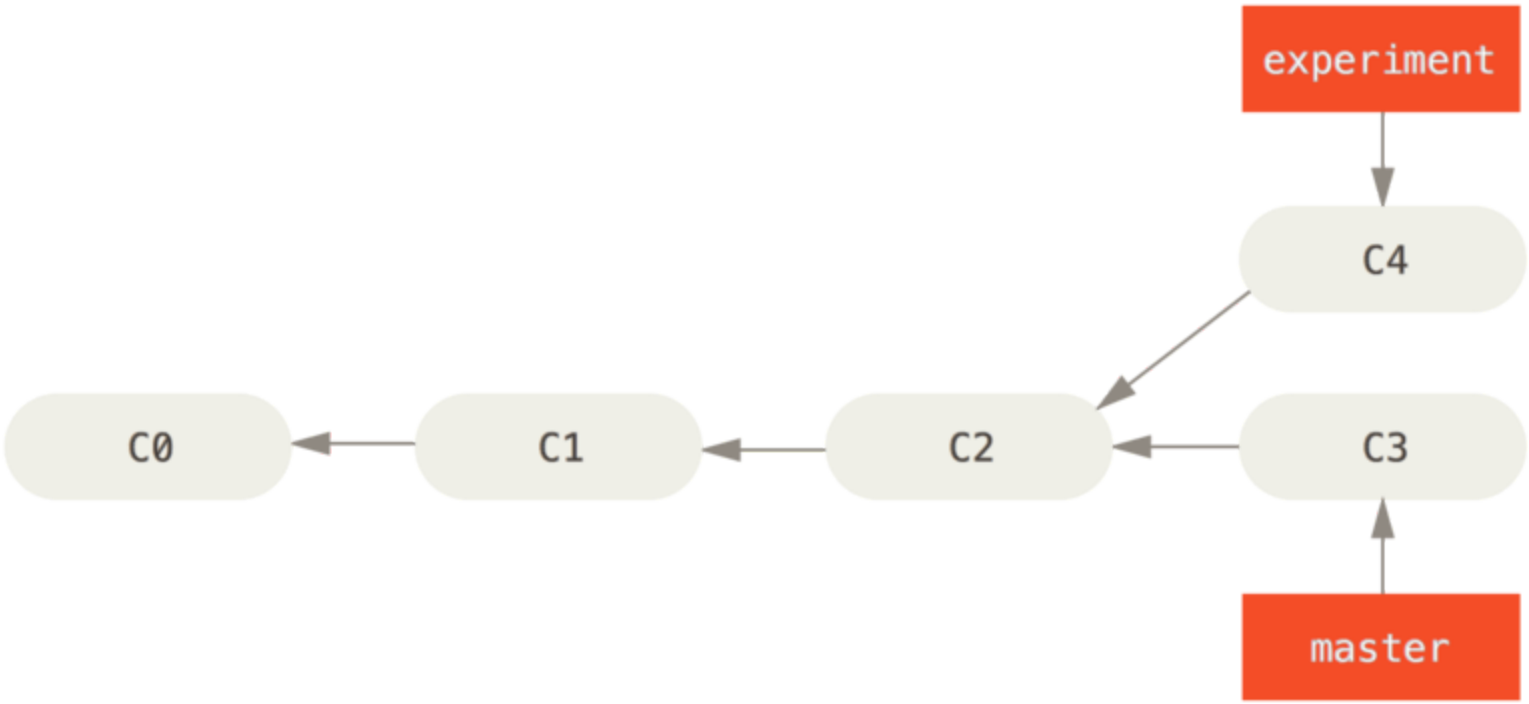
Повторное применение коммита

- Команда `git cherry-pick` позволяет применить изменения указанного коммита к текущей ветке.
- Как правило, изменения копируются из другой ветки.
- Может оказаться полезным чтобы забрать парочку коммитов из другой ветки без полного слияния с той веткой.

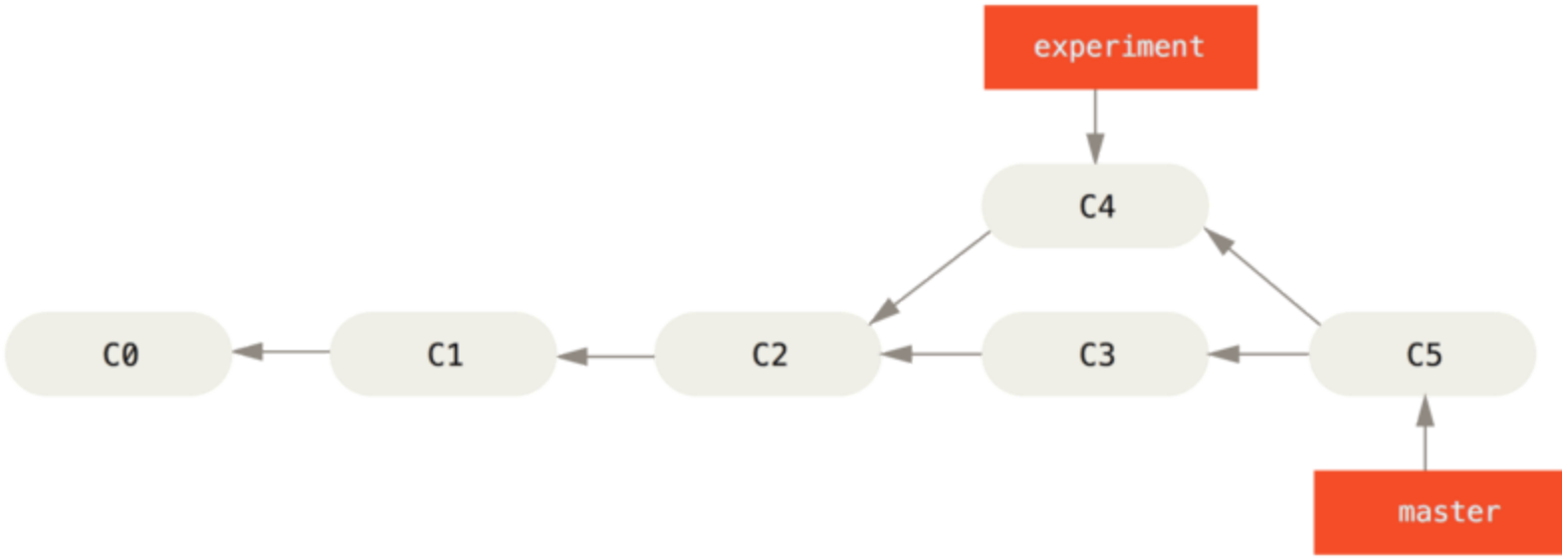
```
$ git cherry-pick <commit>
```

- (является основной для команд `git merge` и `git rebase`)

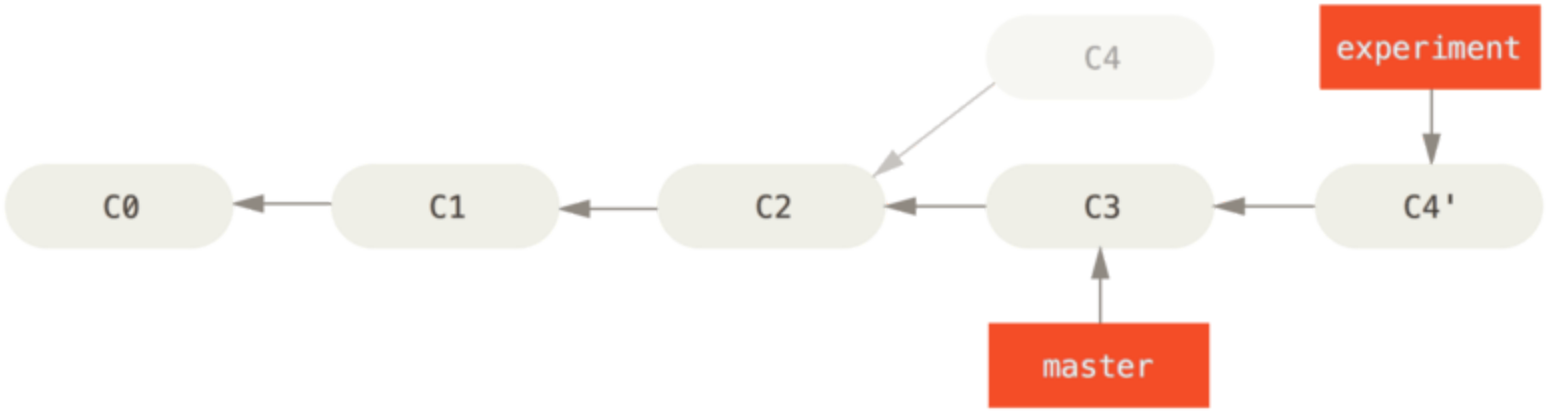
Перебазирование (rebase)



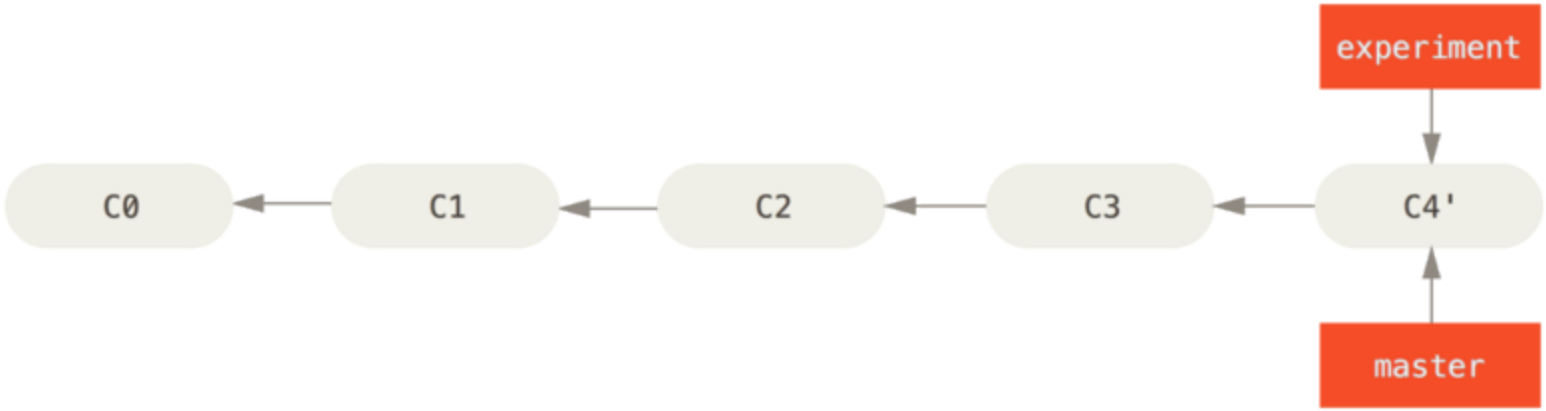
Перебазирование (rebase) (продолжение)



Перебазирование (rebase) (продолжение)



Перебазирование (rebase) (продолжение)



Терминология

Операция	Куда	Что
Слияние	Куда вливаемся (local; ours)	Что вливается (remote; theirs)
Перебазирование	Куда перебазируемся (local; theirs)	Что перебазируется (remote; ours)

ours = где мы находимся (текущая ветка)

theirs = куда придём (что в команде)

Интерактивное перебазирование

Пример

o---o1----o2----p1-----p2

o - master

oX - изменения у коллеги

pX - нащи изменения

задача: надо сжать p1 и p2 в один коммит и переместить его на верхушку мастера

2 решения:

1. интерактивный rebase
2. плющим коммиты и cherry-pick

Поиск проблем бинарным поиском

- Бинарный поиск производится в упорядоченной последовательности.
- При бинарном поиске искомый ключ сравнивается с ключом среднего элемента в массиве.
  - Если они равны, то поиск успешен.
  - В противном случае поиск осуществляется аналогично в левой или правой частях массива.

Поиск проблем бинарным поиском (продолжение)

- В качестве последовательности можем рассматривать список коммитов.
- Ищем коммит, который вносит ключевое изменение
- Искать можем в терминах good/bad, old/new и в собственных (fast/slow)
  - `good` == `old` - состояние до изменения
  - `bad` == `new` - состояние после изменения

Запуск поиска

```
$ git bisect start
$ git bisect bad          # текущий коммит содержит ошибку
$ git bisect good v2.6    # v2.6 не содержит ошибки
```

или одной командой

```
$ git bisect start HEAD v2.6.13-rc2
```

Проверка текущего коммита

- Если текущий коммит содержит ошибку

  `git bisect bad`
- Если текущий коммит не содержит регрессии

  `git bisect good`
- Если текущий коммит не может быть протестирован:

  `git bisect skip`

Прекращение поиска

Прекращение поиска производится с помощью команды

```
git bisect reset
```

Пример работы команды

```
06:35 $ git log --graph --oneline --abbrev-commit --decorate
- fb094a8 (HEAD -> master) Update revision to 108
- edc90c1 Update revision to 98
...
- 63da586 Update revision to 82
- dff4e44 Update revision to BAD
- 0b0f815 Update revision to 80
...
- 3aac53c Update revision to 14
- 5a7f71f Update revision to 8
$ git bisect start fb094a8 5a7f71f
Bisecting: 9 revisions left to test after this (roughly 3 steps)
[aa33a2d8df71733d9d4f2f820d43d96cb1d6a4b3] Update revision to 70
$ git bisect good
Bisecting: 4 revisions left to test after this (roughly 2 steps)
[42dc6bdb8ad4d369b8d7e238d8df86fa96720b45] Update revision to 84
$ git bisect bad
Bisecting: 2 revisions left to test after this (roughly 1 step)
[0b0f8159db025b4a14d9a6160ad13696d2475665] Update revision to 80
$ git bisect good
Bisecting: 0 revisions left to test after this (roughly 1 step)
[63da586a5893144c9a72872e1540c941ccf3c452] Update revision to 82
$ git bisect bad
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[dff4e444ab2813b0c9b8440ff3606abbe48b06f0] Update revision to BAD
$ git bisect bad
dff4e444ab2813b0c9b8440ff3606abbe48b06f0 is the first bad commit
commit dff4e444ab2813b0c9b8440ff3606abbe48b06f0
Author: Maxim Suslov <MSuslov@luxoft.com>
Date:   Mon Sep 30 00:00:55 2019 +0300

    Update revision to BAD

 revision.txt | 1 +
 1 file changed, 1 insertion(+)
```

В случае ошибки

- Посмотреть историю решений

```
$ git bisect log
# bad: [fb094a8e] Update revision to 108
# good: [5a7f71f6] Update revision to 8
git bisect start 'fb094a8' '5a7f71f'
# good: [aa33a2d8] Update revision to 70
git bisect good aa33a2d8
# bad: [42dc6bdb] Update revision to 84
git bisect bad 42dc6bdb
...
```

В случае ошибки (продолжение)

- Сохранить историю в файл

```
git bisect log > history.txt
```

- Отредактировать решения, изменив файл
- Перезапустить поиск

```
git bisect reset
git bisect replay history.txt
```

- Продолжить выполнять команды `git bisect ...`

Автоматизация поиска

```
$ cat ~/test.sh
#!/bin/sh
make || exit 125                     # this skips broken builds
~/check_test_case.sh                 # does the test case pass?
$ git bisect start HEAD HEAD~10 --   # culprit is among the last 10
$ git bisect run ~/test.sh
$ git bisect reset                   # quit the bisect session
```

То же без файла `test.sh` и используя `borken` / `fixed`:

```
$ git bisect start --term-old broken --term-new fixed
$ git bisect fixed
$ git bisect broken HEAD~10
$ git bisect run sh -c "make || exit 125; ~/check_test_case.sh"
$ git bisect reset
```

Заключение