Git Concepts: Demystified

Taylor Blau
CSE 374 (23wi)

Taylor Blau

Staff Software Engineer, GitHub

Now: open-source Git project, Git “at scale”, etc.
Previously: CSE student at UW (‘20)

CSE 374

- Took an extremely similar class as this one when I was an undergrad
    - CSE 391 - System and Software Tools
    - CSE 374 - Programming Concepts & Tools
- These classes “fill in the gaps”: familiarize you with common tools
- “Learning how to learn”: learn how to become familiar with new tools that don’t exist yet!

Git

- Git is an extremely powerful and ubiquitous distributed version control system (DVCS)
- You’ve covered the basics:
    - repositories
    - `git clone`
    - `git branch`
    - `git add`
    - `git log`
- Today’s plan: talk about some Git “mysteries”
    - …or at least a few concepts that seem tricky at first, but will seem non-tricky when seen through the right lens

Git preliminaries

- Git is a distributed version control system (DVCS)
    - Everybody has their own copy of a repository
- Git repositories consist of a set of objects and references
    - Objects: the individual files, directories, and commits that make up your project
    - References: the branches/tags in your repository and which commits they point to
- Objects:
    - Blobs: represents an individual file, contains the content of that file
    - Tree: represents an individual directory, contains a list of `(name, object_id)` pairs
    - Commit: represents a snapshot of your repository, contains:
        - A message describing your changes
        - Author/committer information (e.g., `Taylor Blau <ttaylorr@github.com>`)
        - The "root tree"'s object identifier
        - Zero or more parent commit identifiers

Git commits

- Commit: represents a snapshot of your repository, contains:
  - A message describing your changes
  - Author/committer information (e.g., `Taylor Blau <ttaylorr@github.com>`)
  - The “root tree”’s object identifier
  - Zero or more parent commit identifiers
- Parents?
  - Zero parents: the root of history
  - One parent: an individual element of history
  - More than one parent: a merge between several points in history

```
$ git cat-file -p HEAD
tree 9354315c17f67f9abfb5007076a508f80b77f654
parent 5048df67b295baeaaa6dafb16ff712bd2a62731a
author Junio C Hamano <gitster@pobox.com> 1677106559 -0800
committer Junio C Hamano <gitster@pobox.com> ...

The seventeenth batch

Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

commit internals

- A message describing your changes
- Author/committer information
- The “root tree”’s object identifier
- Zero or more parent commit identifiers

```
$ git cat-file -p HEAD
tree 9354315c17f67f9abfb5007076a508f80b77f654
parent 5048df67b295baeaaa6dafb16ff712bd2a62731a
author Junio C Hamano <gitster@pobox.com> 1677106559 -0800
committer Junio C Hamano <gitster@pobox.com> ...

The seventeenth batch

Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

commit internals

- A message describing your changes
- Author/committer information
- The “root tree”’s object identifier
- Zero or more parent commit identifiers

```
$ git cat-file -p HEAD
tree 9354315c17f67f9abfb5007076a508f80b77f654
parent 5048df67b295baeaaa6dafb16ff712bd2a62731a
author Junio C Hamano <gitster@pobox.com> 1677106559 -0800
committer Junio C Hamano <gitster@pobox.com> ...

The seventeenth batch

Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

commit internals

- A message describing your changes
- Author/committer information
- The "root tree"'s object identifier
- Zero or more parent commit identifiers

```
$ git cat-file -p HEAD
tree 9354315c17f67f9abfb5007076a508f80b77f654
parent 5048df67b295baeaaa6dafb16ff712bd2a62731a
author Junio C Hamano <gitster@pobox.com> 1677106559 -0800
committer Junio C Hamano <gitster@pobox.com> ...

The seventeenth batch

Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

commit internals

- A message describing your changes
- Author/committer information
- The “root tree”’s object identifier
- Zero or more parent commit identifiers

```
$ git cat-file -p HEAD
tree 9354315c17f67f9abfb5007076a508f80b77f654
parent 5048df67b295baeaaa6dafb16ff712bd2a62731a
author Junio C Hamano <gitster@pobox.com> 1677106559 -0800
committer Junio C Hamano <gitster@pobox.com> ...

The seventeenth batch

Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

commit internals

- A message describing your changes
- Author/committer information
- The “root tree”’s object identifier
- Zero or more parent commit identifiers

Git Mysteries

`git cherry-pick`

or: how to move commit(s) from one branch to another

`git rebase`

or: how to re-apply a range of history on a new base

`git stash`

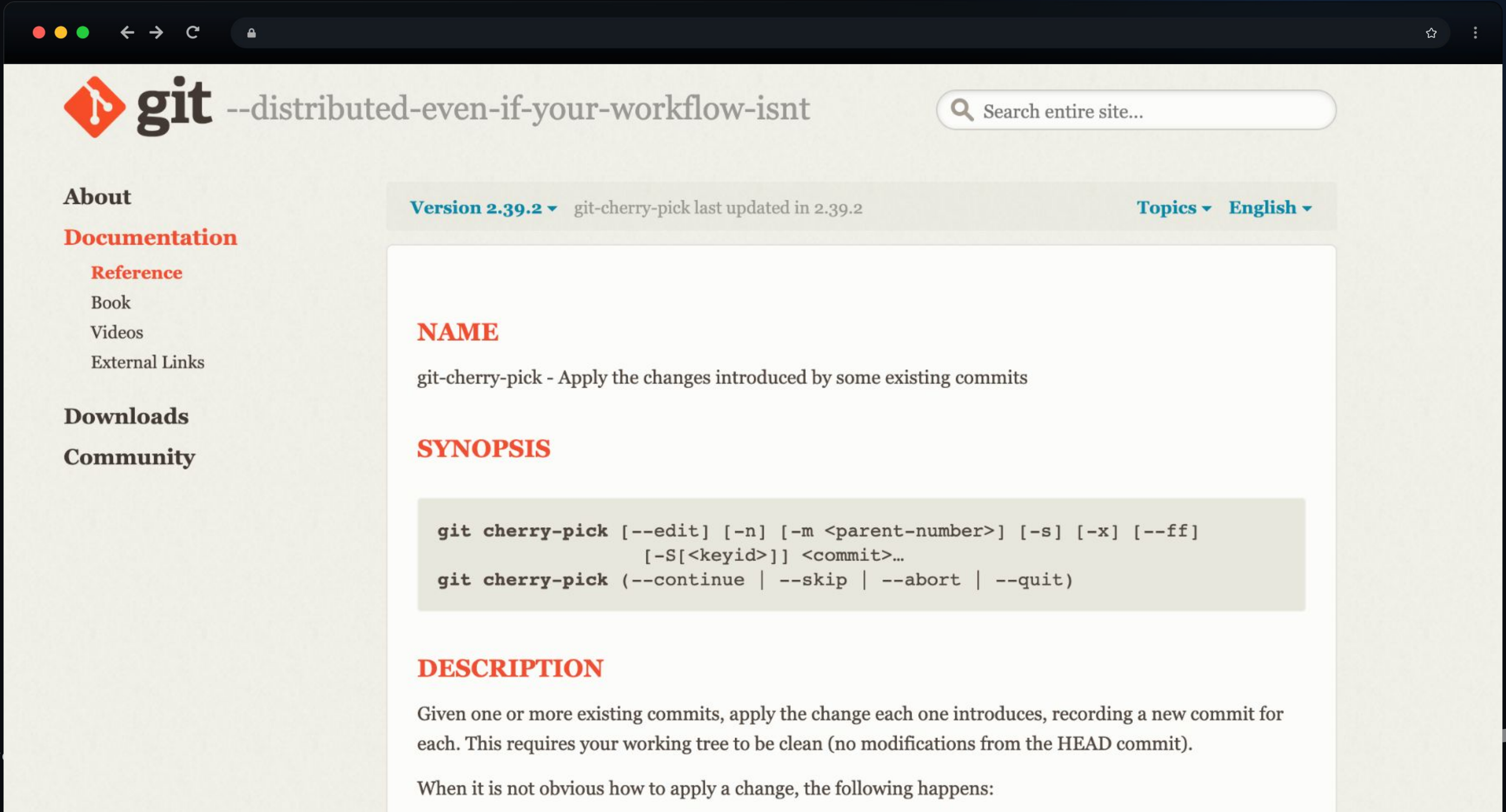
or: how to save your work for later

git cherry-pick

or: how to move commit(s) from one branch to another

git cherry-pick

- Recall branches and commits: say you’re working on a homework assignment with a partner
- You solve some component of that assignment, and commit your work before you’re ready to push it up
- Oops! You committed while on the wrong branch.
- What do you do?



NAME

git-cherry-pick - Apply the changes introduced by some existing commits

SYNOPSIS

```
git cherry-pick [--edit] [-n] [-m <parent-number>] [-s] [-x] [--ff]
                [-S[<keyid>]] <commit>…
git cherry-pick (--continue | --skip | --abort | --quit)
```

DESCRIPTION

Given one or more existing commits, apply the change each one introduces, recording a new commit for each. This requires your working tree to be clean (no modifications from the HEAD commit).

When it is not obvious how to apply a change, the following happens:

Mar

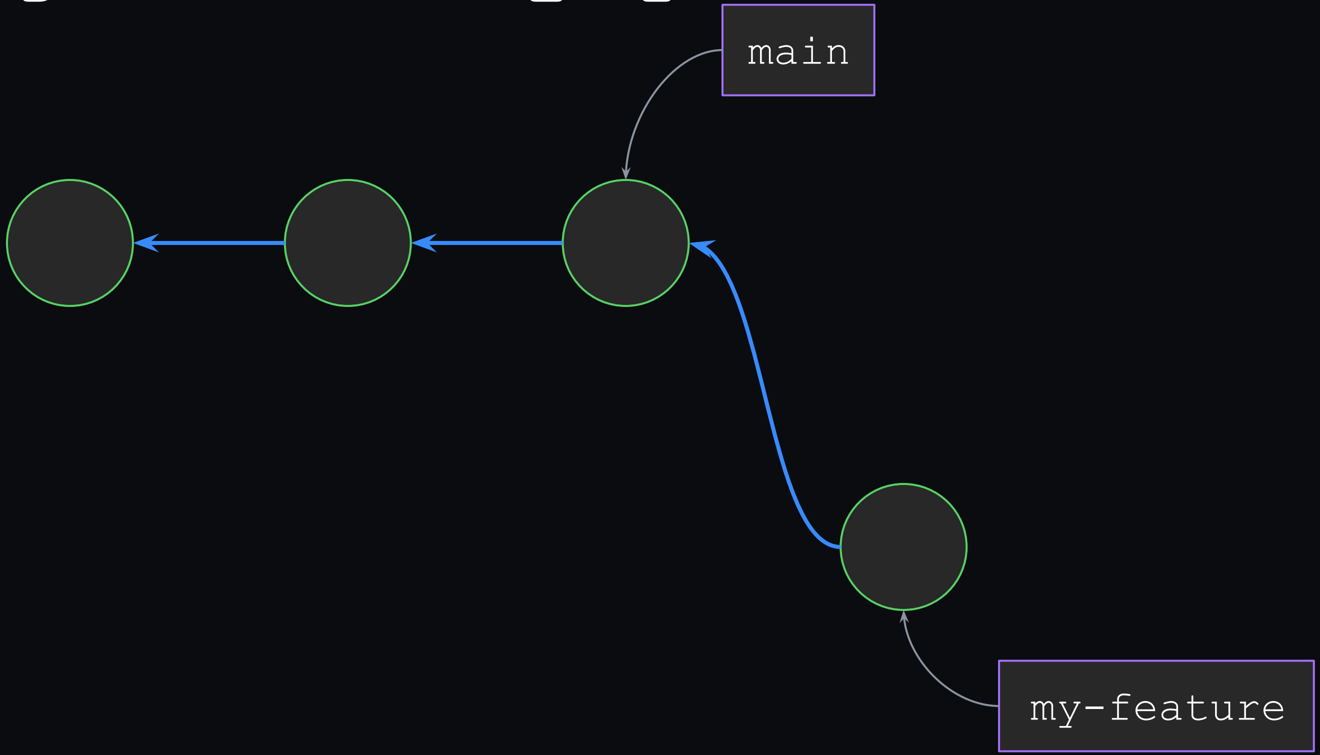
git cherry-pick

- `git cherry-pick $SOME_COMMIT`

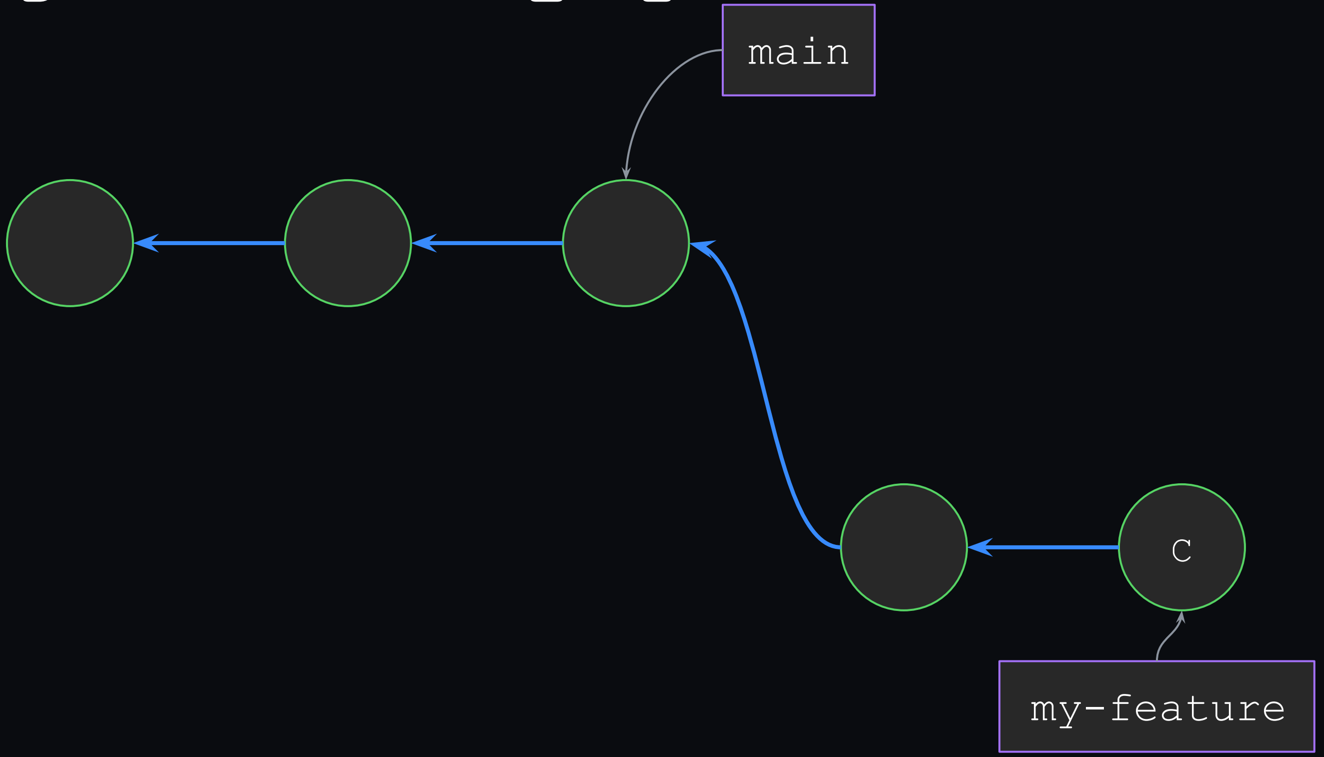
…What's going on here?

1. Instructs Git to create a new commit
2. Git makes a new commit object with the same root tree, and author as `$SOME_COMMIT`
   a. The committer is you (regardless of whether or not you committed `$SOME_COMMIT`)
   b. The parent commit is the last thing that was on your branch
3. Your branch is updated to point at the commit that was just created by cherry-pick
4. Your working copy is updated to reflect any change(s) made by the copy of `$SOME_COMMIT`

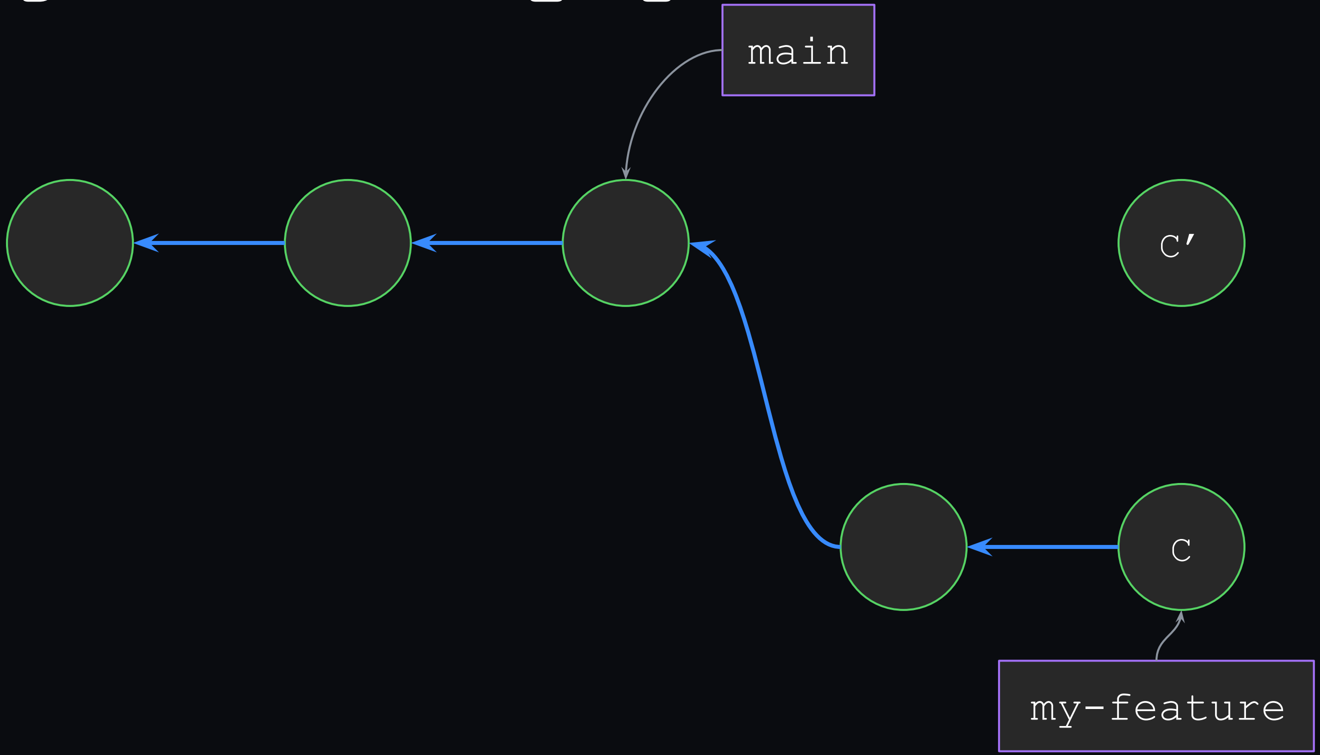
git cherry-pick



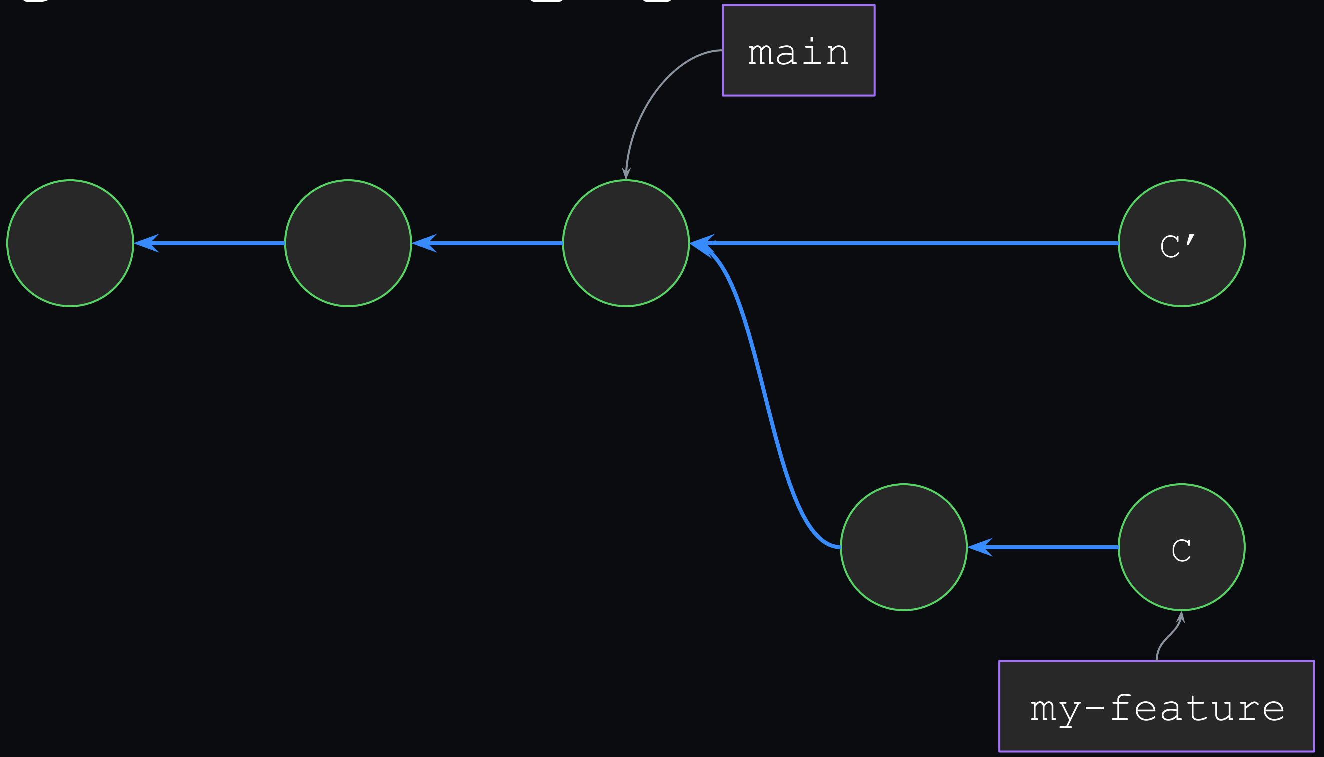
git cherry-pick



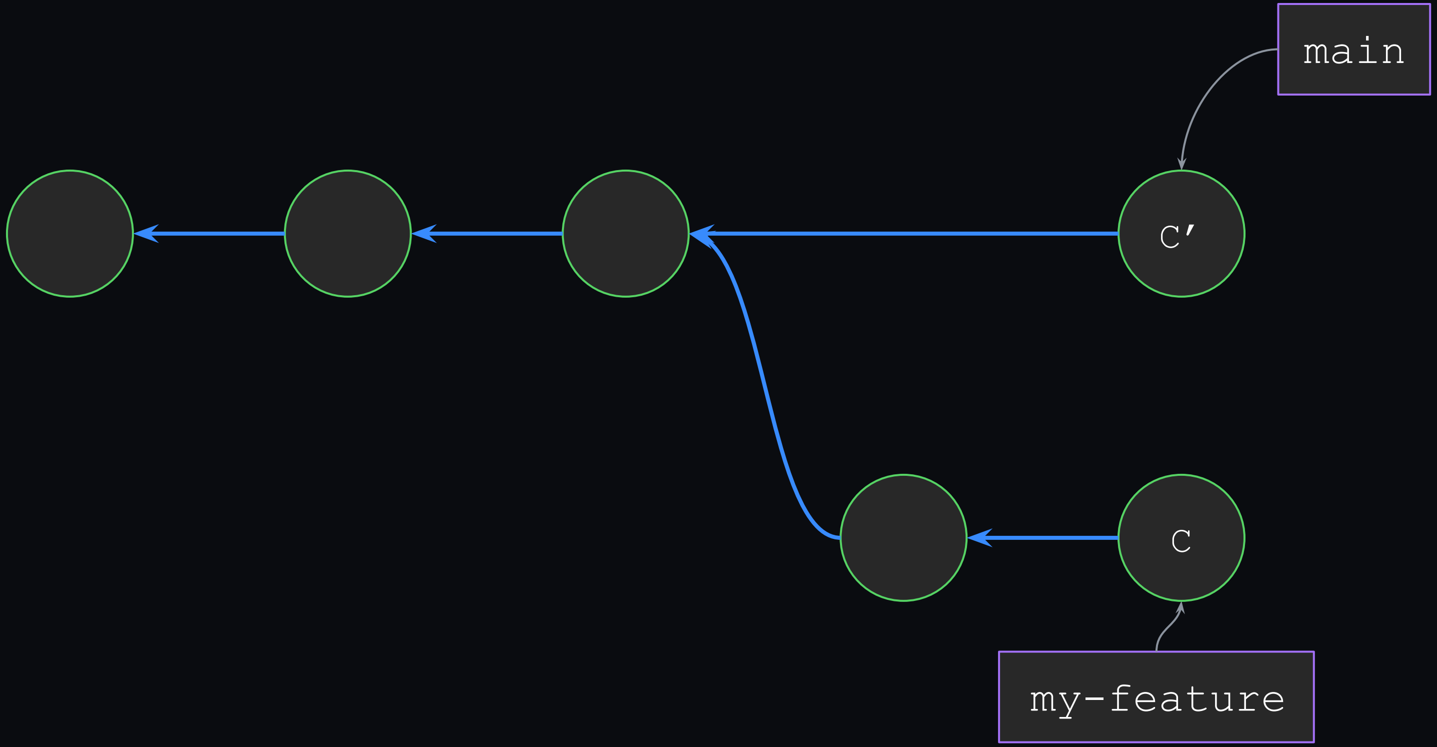
git cherry-pick



git cherry-pick



git cherry-pick



`git cherry-pick` pro-tips

- `git cherry-pick -x` to remember where your commit came from
- `git cherry-pick -e` to edit your (new) commit message before applying
- `git cherry-pick -m<N>` to pick which parent is "mainline" when cherry-picking a merge
  - Replay your changes relative to the Nth parent
  - In other words: which of the two parents captures the state of the branch you're merging into prior to applying that merge?
  - …almost always reasonable to write `-m1` if you're not sure

…Lots more available at: https://git-scm.com/docs/git-cherry-pick

git rebase

or: how to re-apply a range of history on a new base

git rebase -i

-
  ```
  $ git fetch origin main
  $ git rebase -i main --onto origin/main
  ```

…What's going on here?

1. Instructs Git to create new commits from the ones between `main..HEAD` on top of some new base (in this case, `origin/main`)
2. Git makes new commit object(s) with the same root tree, and author each commit in range
3. Your branch is updated to point at the commit that was just created by rebase
4. Your working copy is updated to reflect any change(s) made

…similar to `git cherry-pick`!

`git rebase -i`

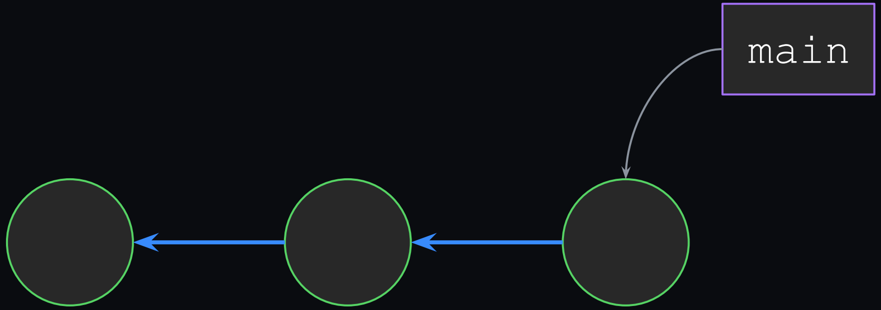
…similar to `git cherry-pick`!

Key insight:

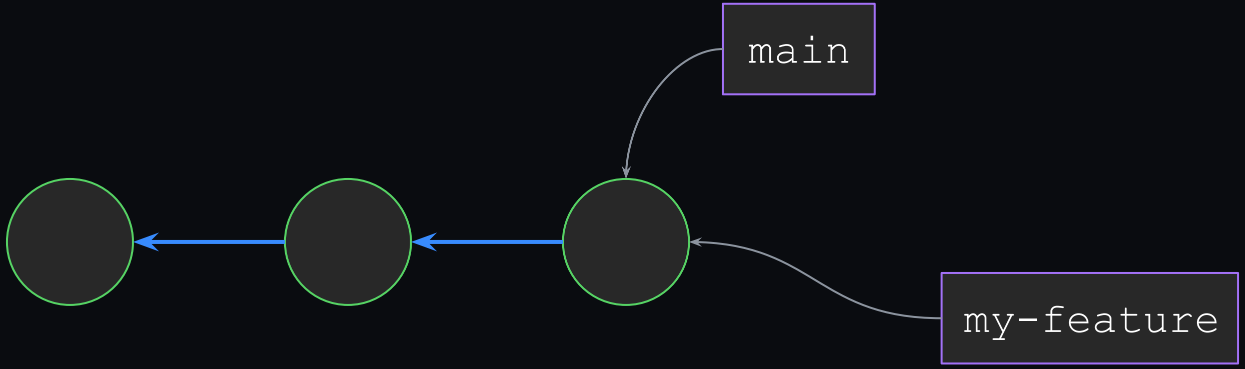
- Many “hard” Git concepts (like `cherry-pick` and `rebase -i`) can be explained by two simple ideas:
    a. Commits are snapshots, not diffs
    b. “Moving” a commit consists of creating a new commit with the same contents, and attaching it at the right point(s) in your history

`cherry-pick` does this once for a single commit, the new base is whatever branch you’re on
`rebase -i` does this for a *range* of commits, the new base is whatever you specify

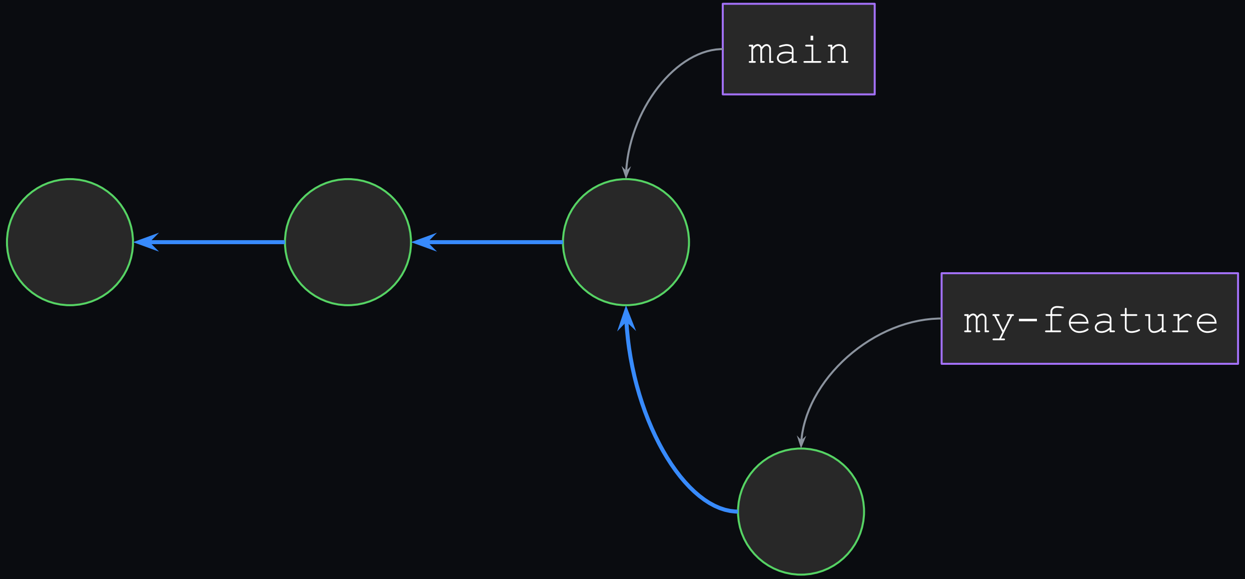
git rebase -i



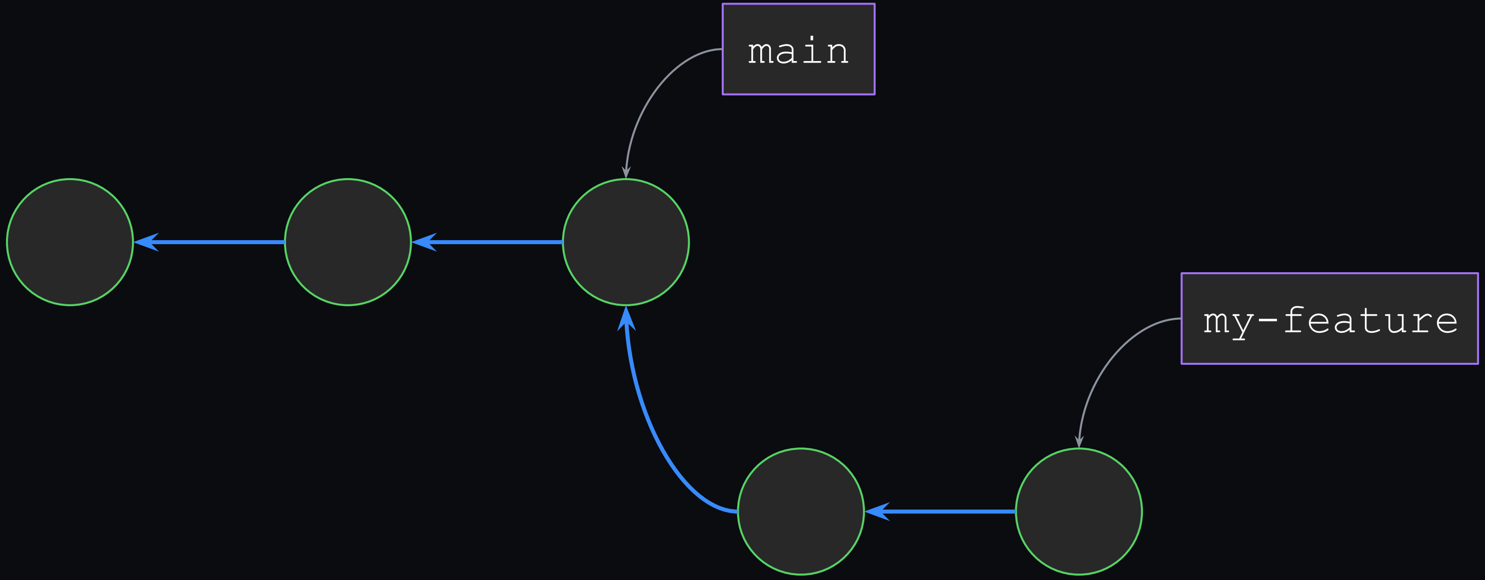
git rebase -i



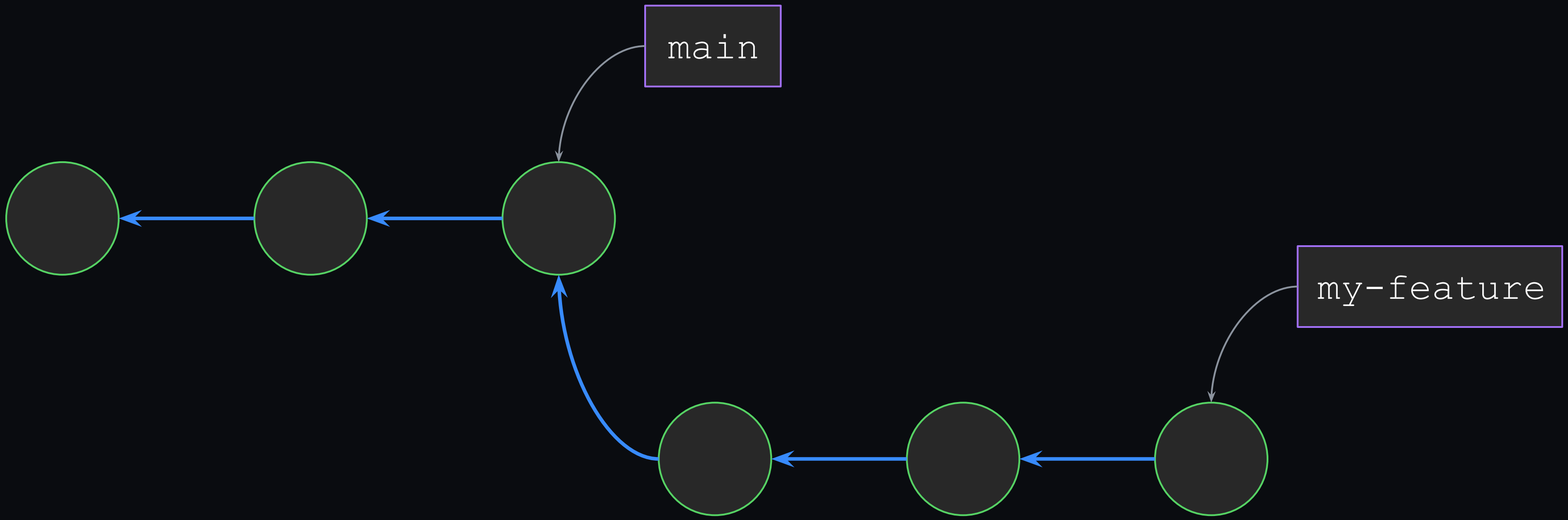
git rebase -i



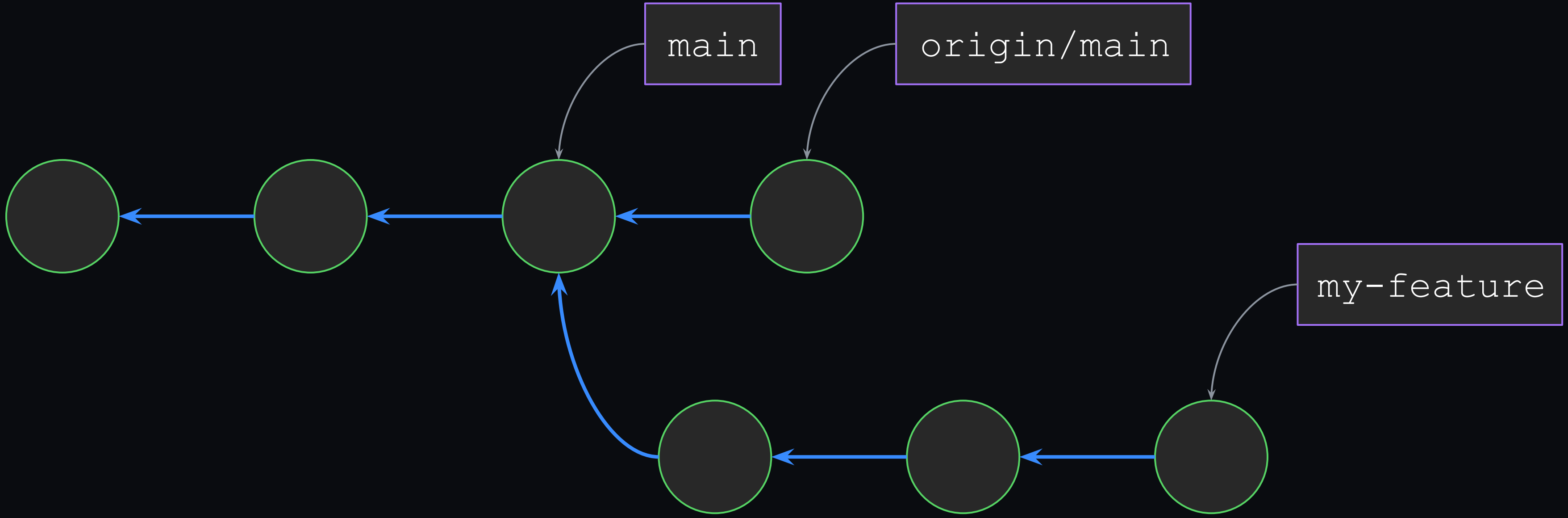
git rebase -i



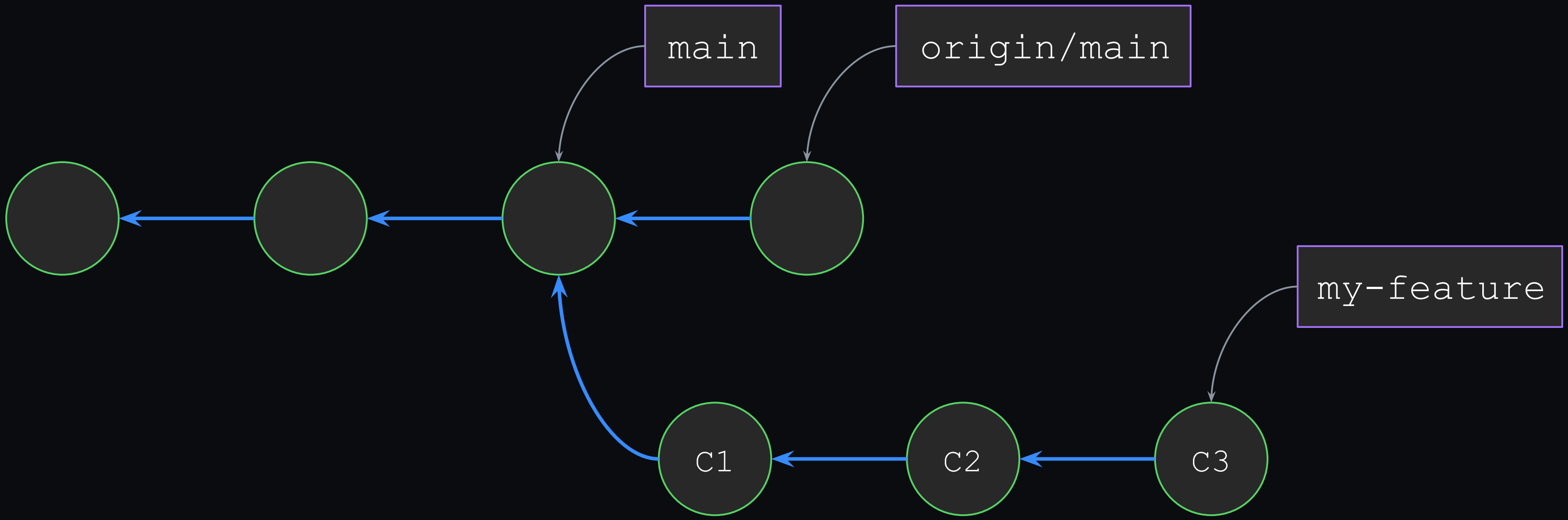
git rebase -i



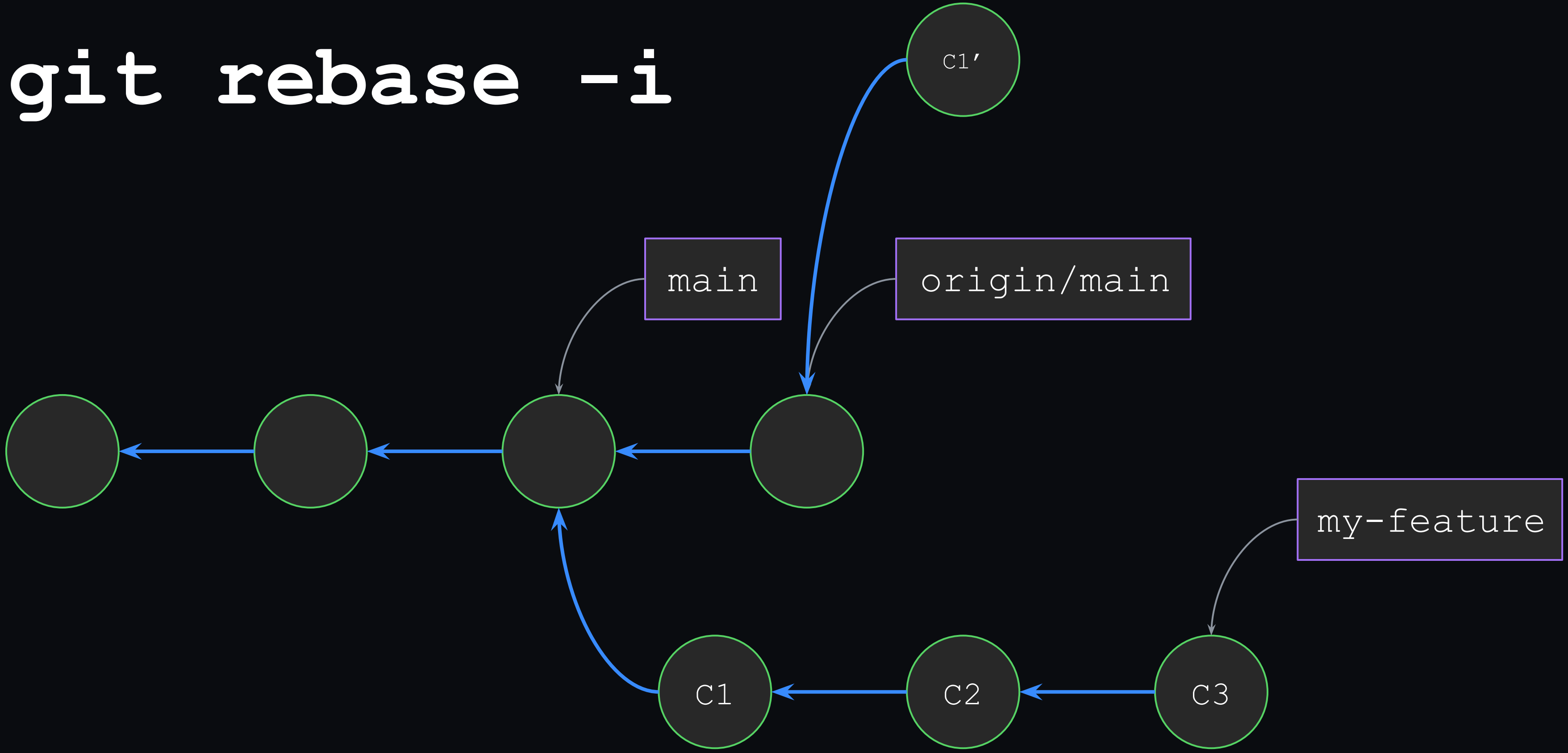
git rebase -i



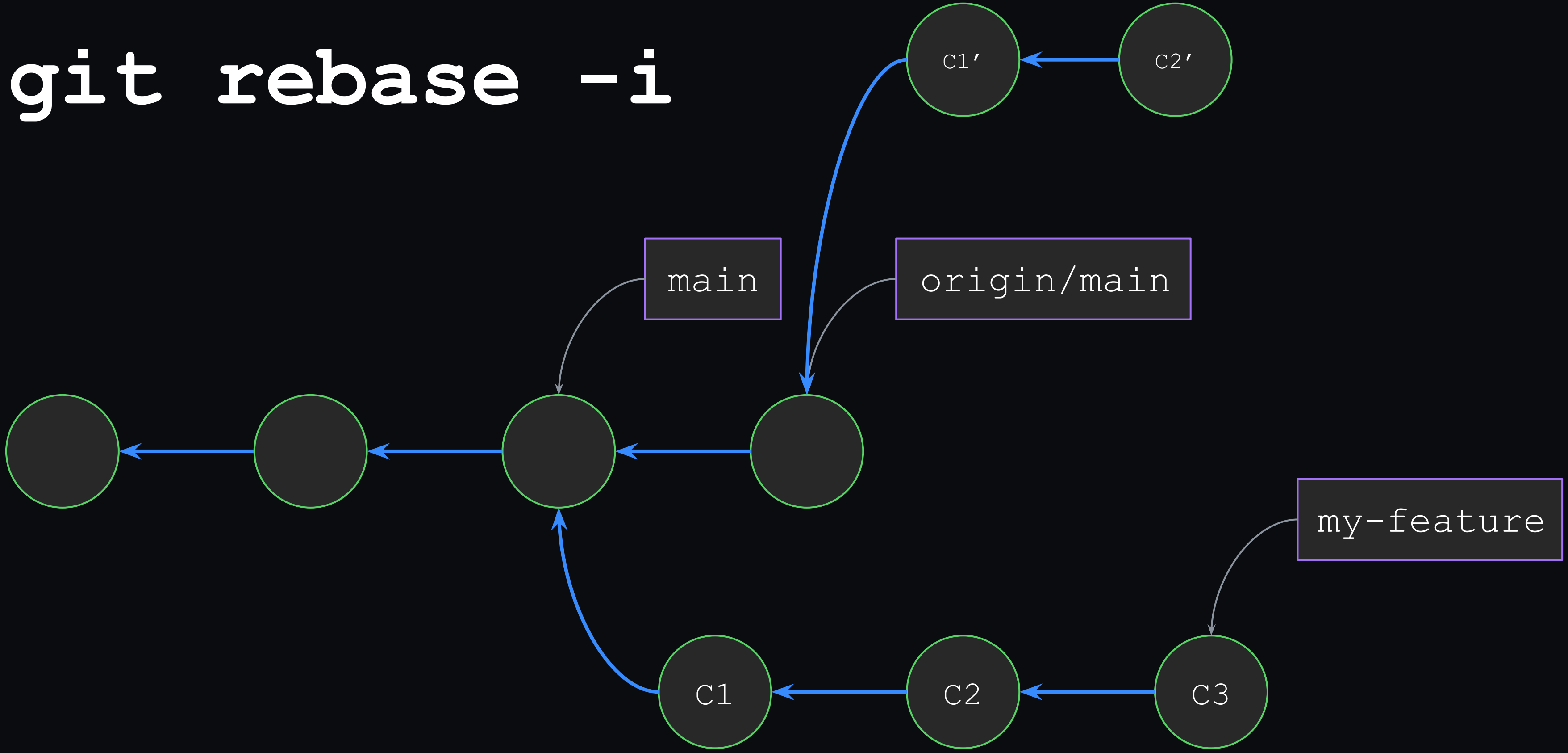
git rebase -i



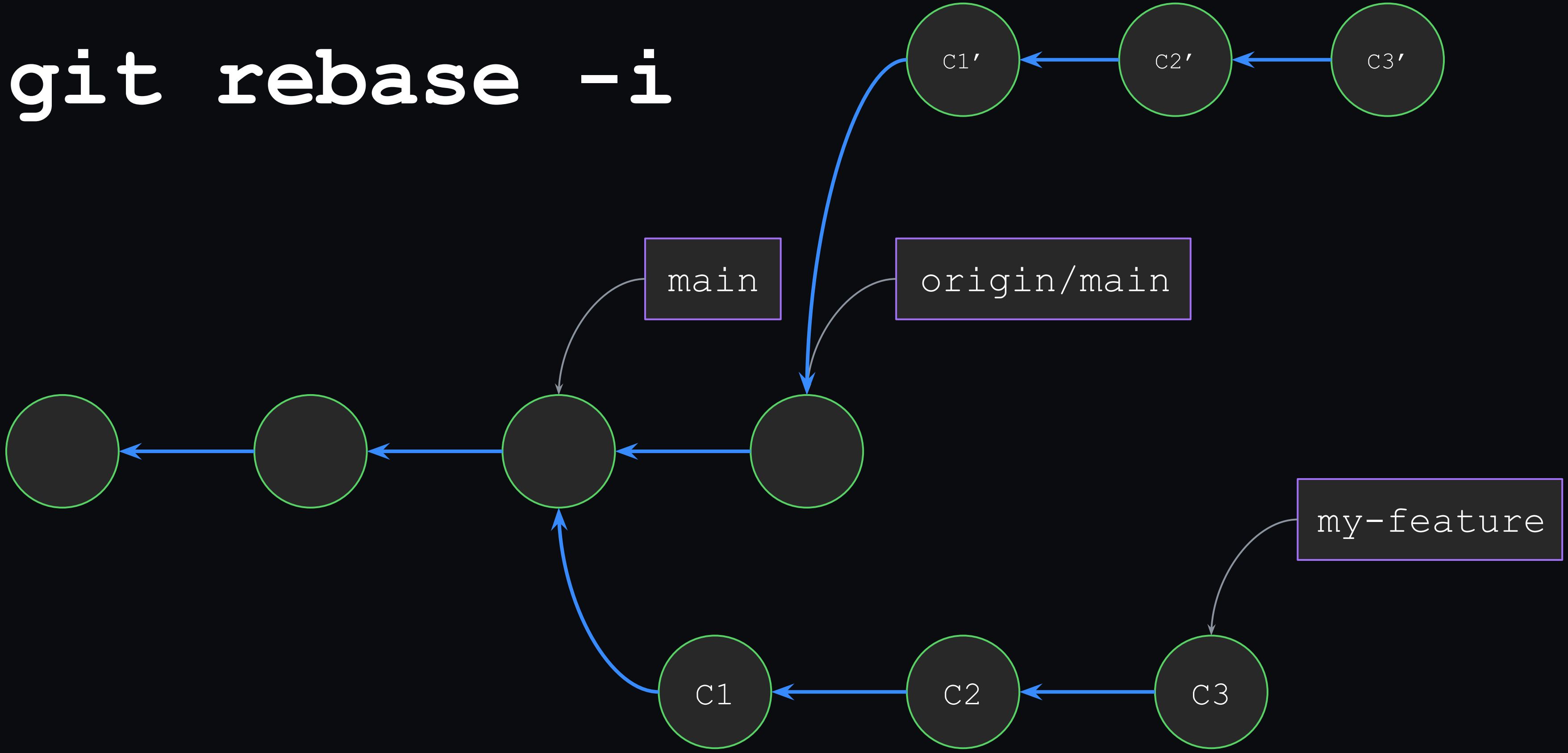
git rebase -i



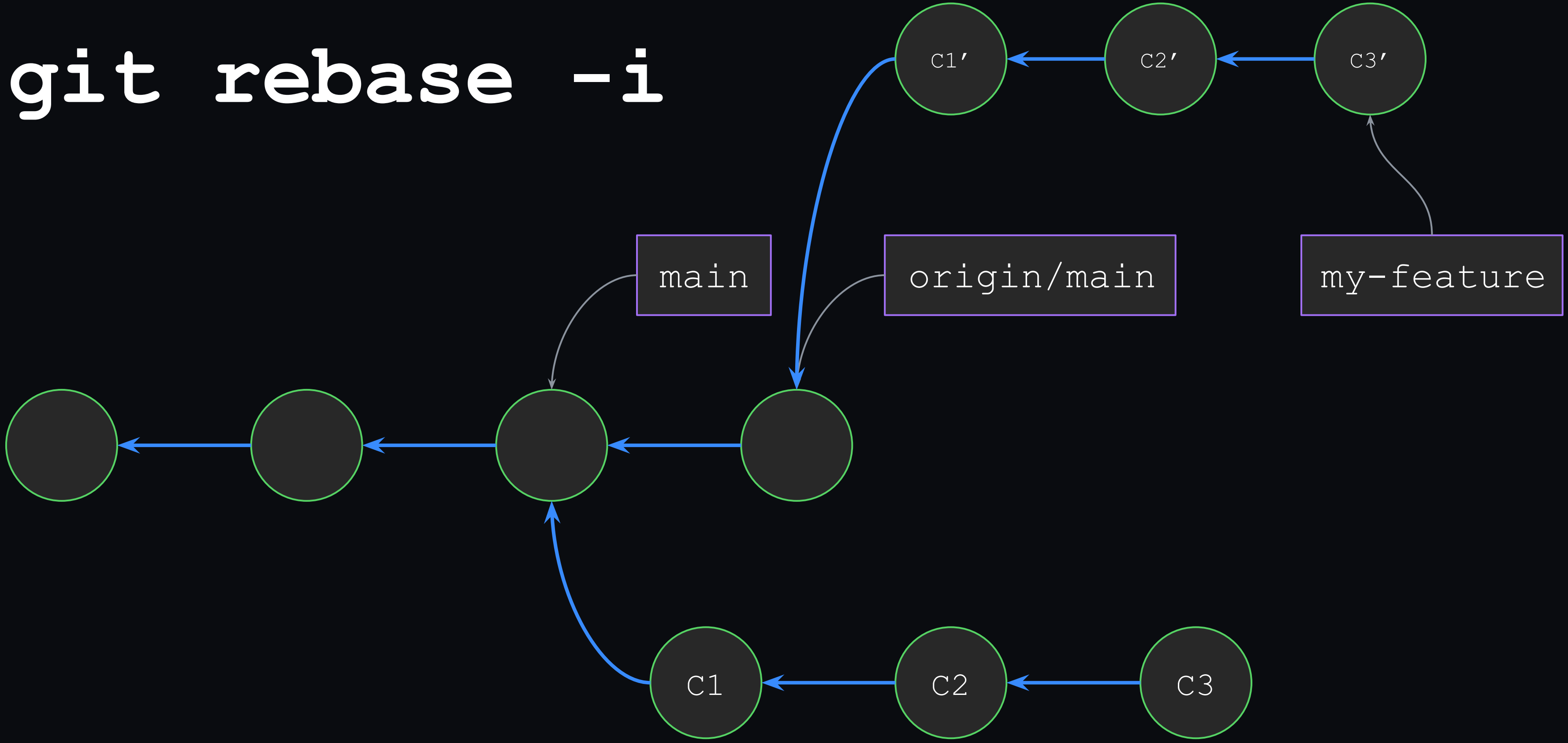
git rebase -i



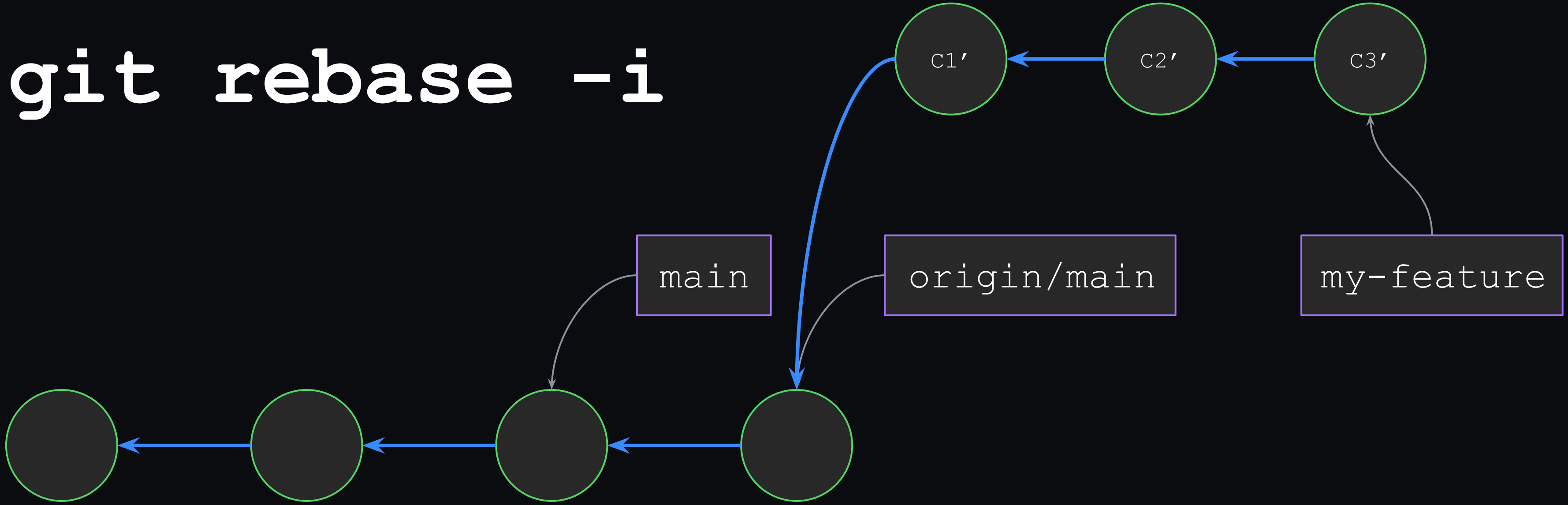
git rebase -i



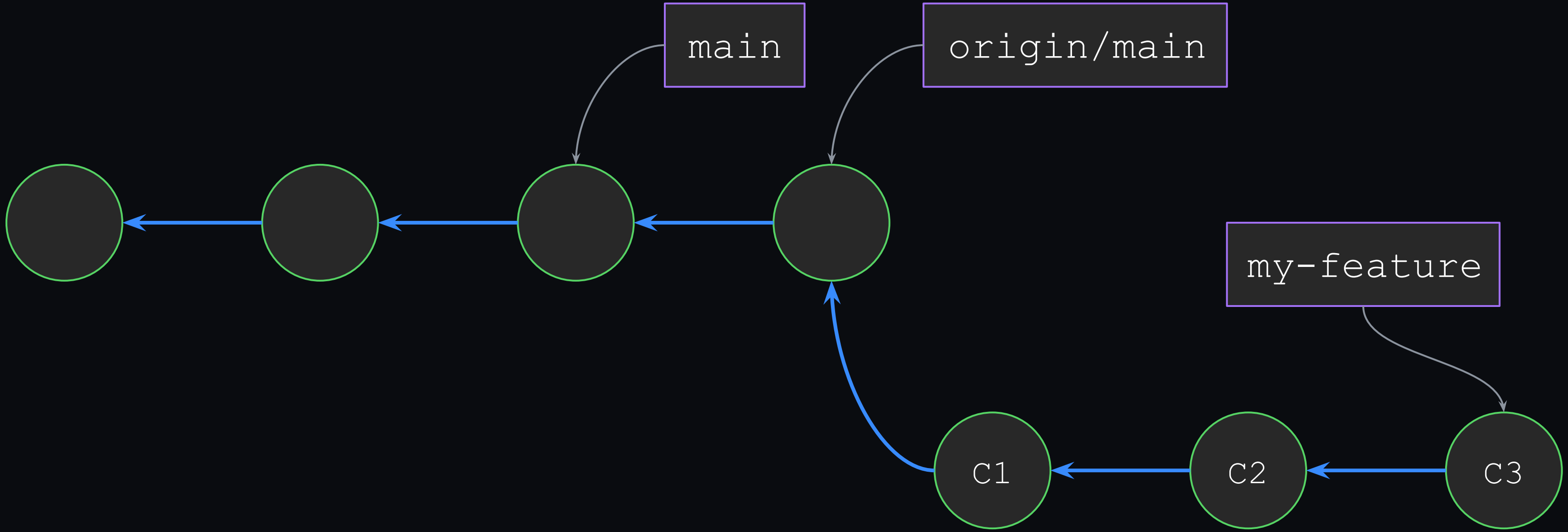
git rebase -i



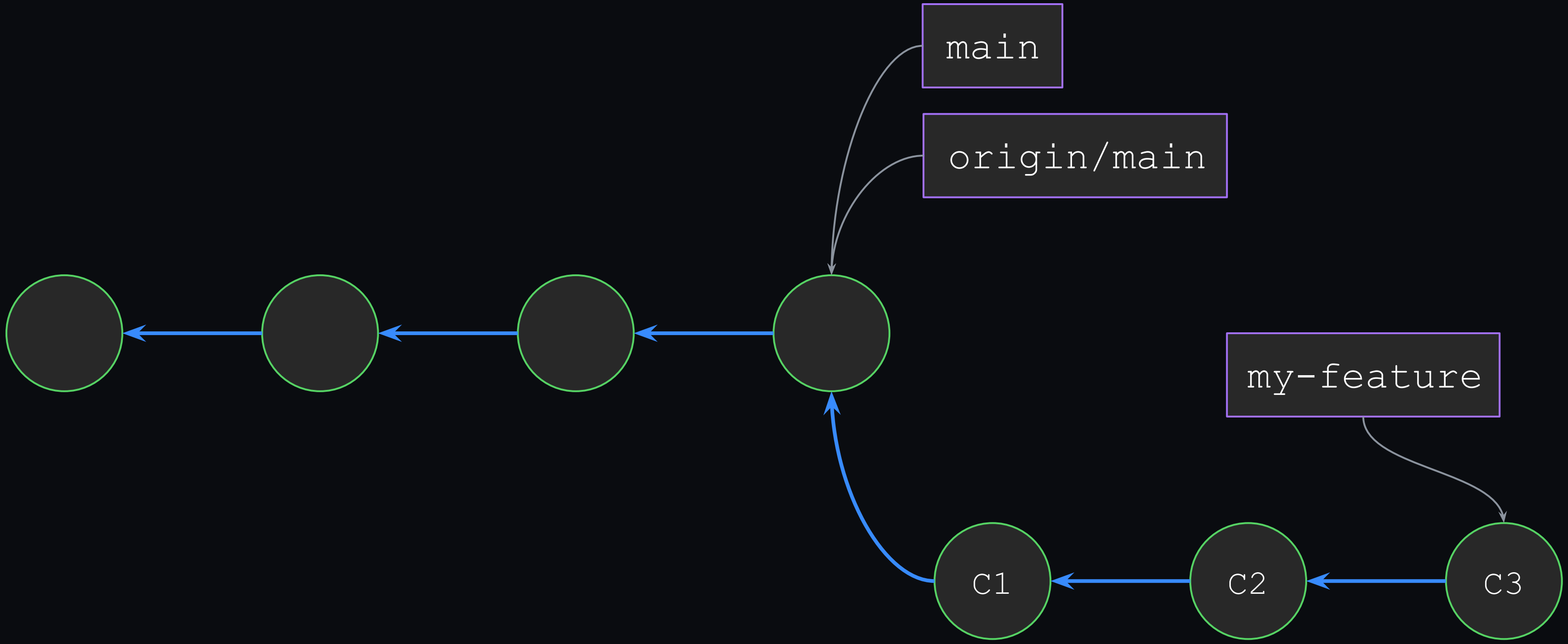
git rebase -i



git rebase -i



git rebase -i



`git rebase -i`

To summarize:

- Specified a *range* of commits we wanted to move
- …and an existing base for those commits (our copy of `main`)
- …and a proposed new base for those commits (upstream's copy of `main`, `origin/main`)
- Git created new commits, one by one, placing them correctly in history and updating ref(s) accordingly

`git rebase -i`

Lots of other functionality that we're not going over:

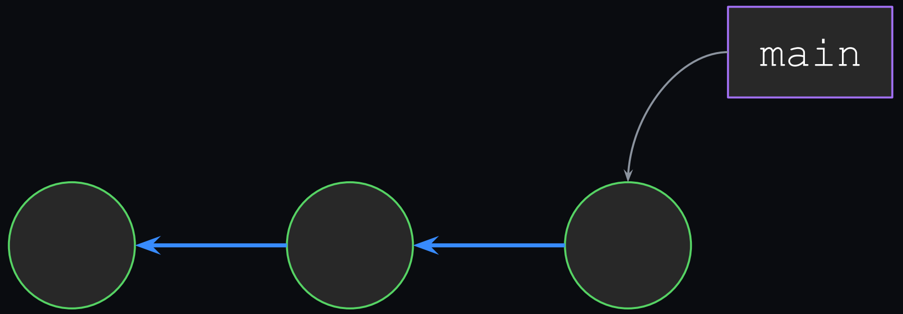
- Dropping/reordering commits
- Stopping in the middle of a rebase, editing your work
- Rewording commit messages
- `--rebase-merges` to preserve more complex (non-linear) structures in history
- …etc :-)

For more: https://git-scm.com/docs/git-rebase

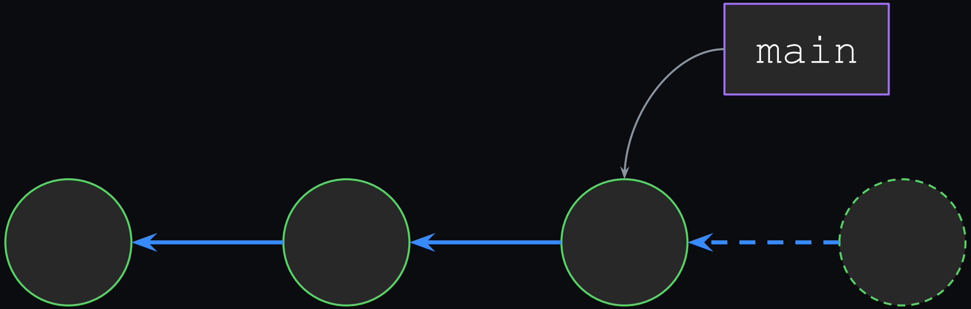
git stash

or: how to save your work for later

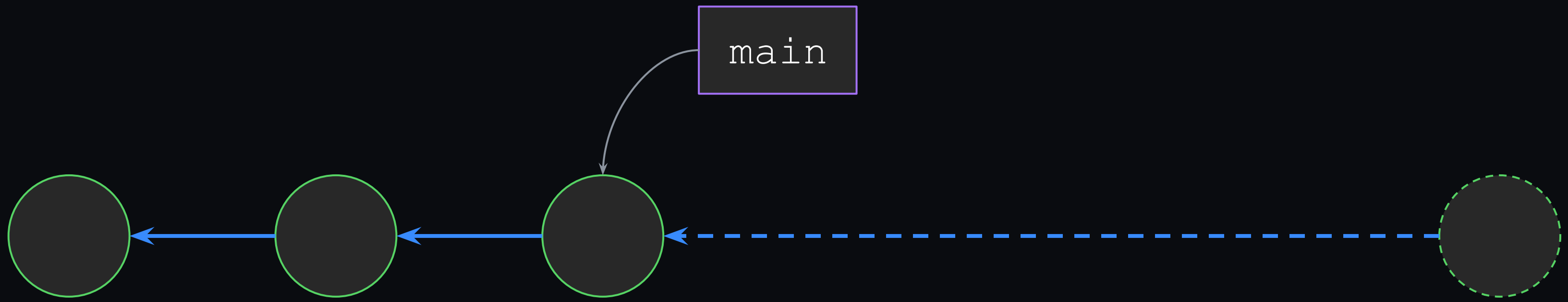
git stash



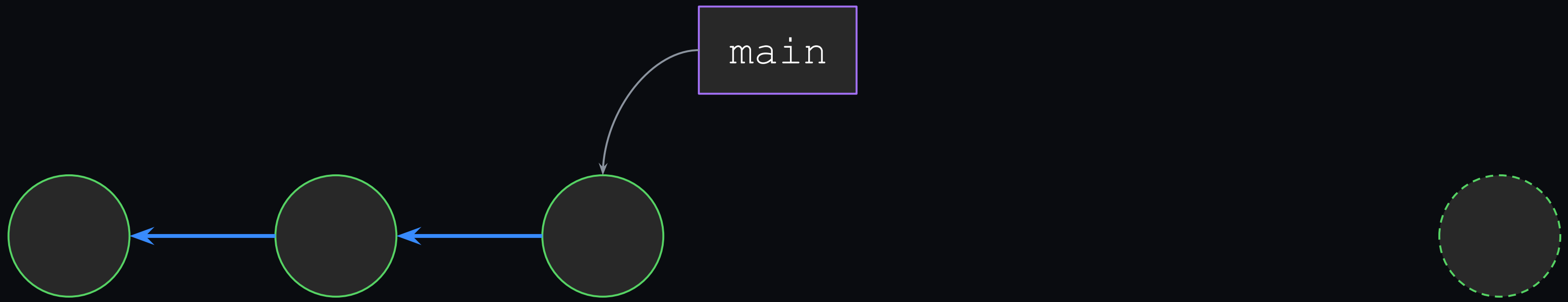
git stash



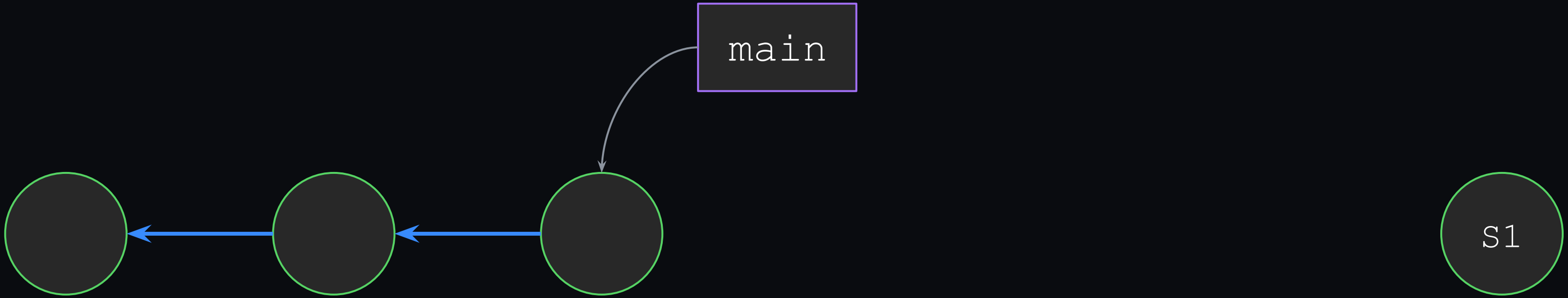
git stash



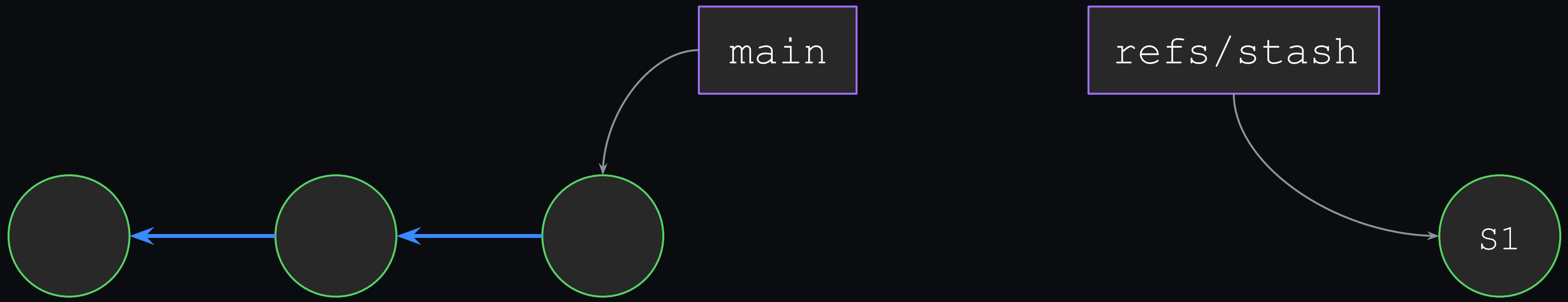
git stash



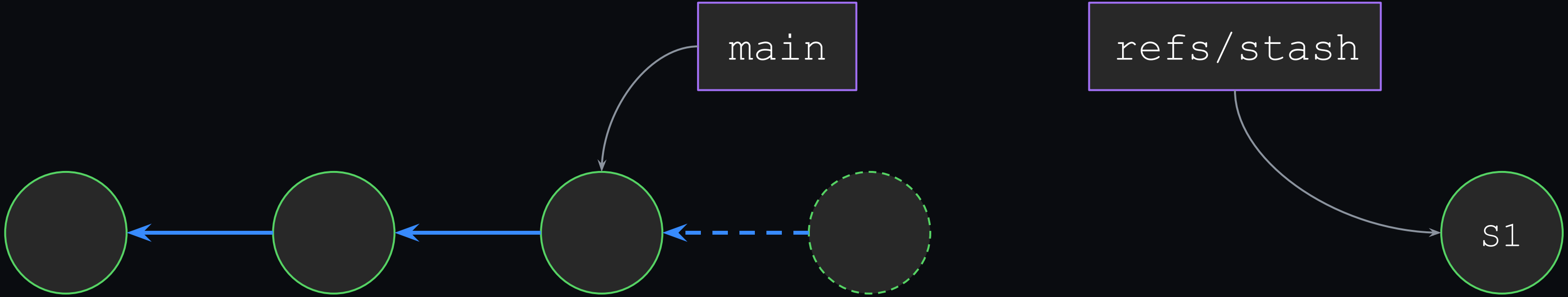
git stash



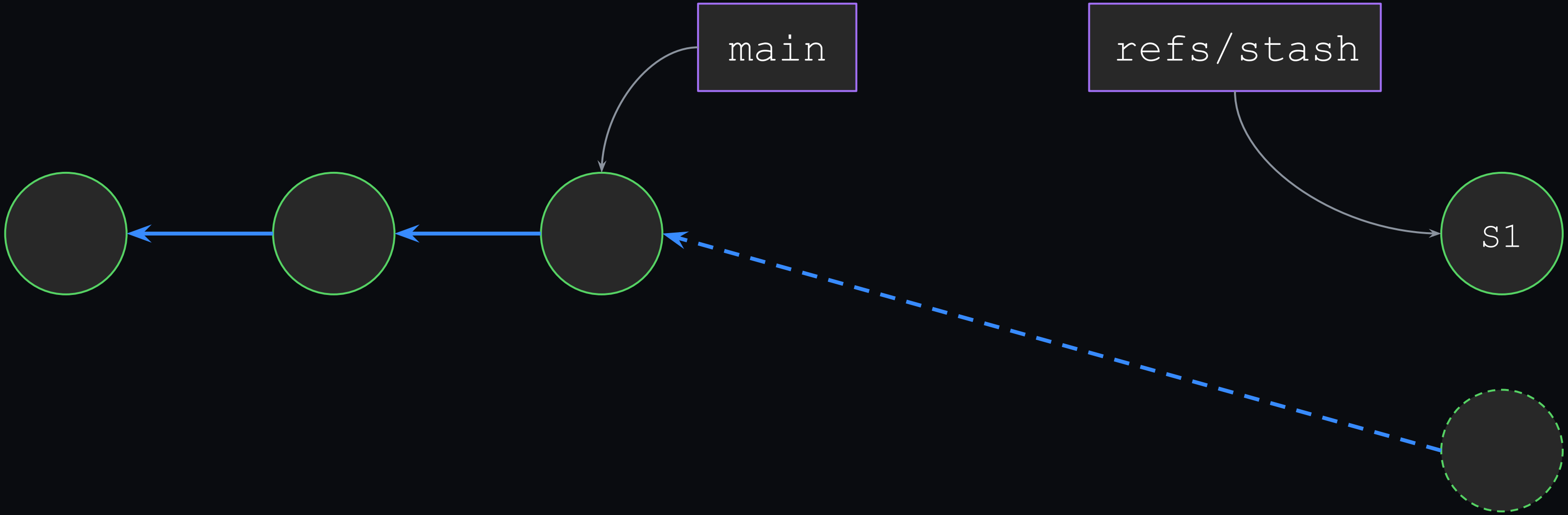
git stash



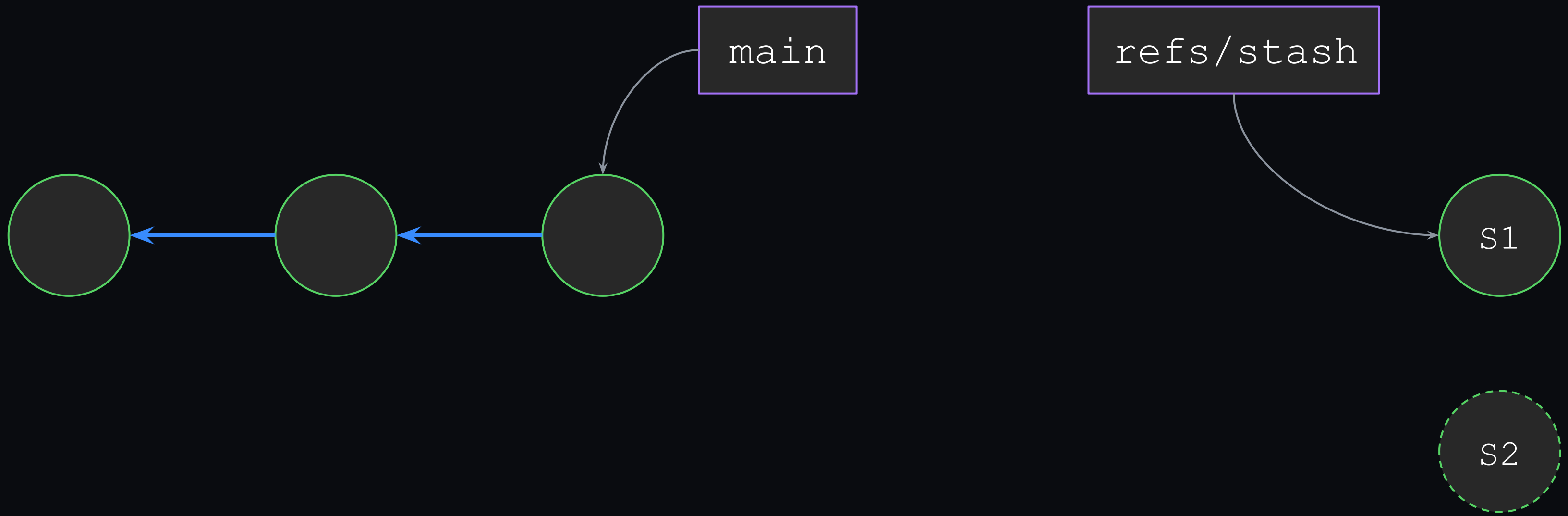
git stash



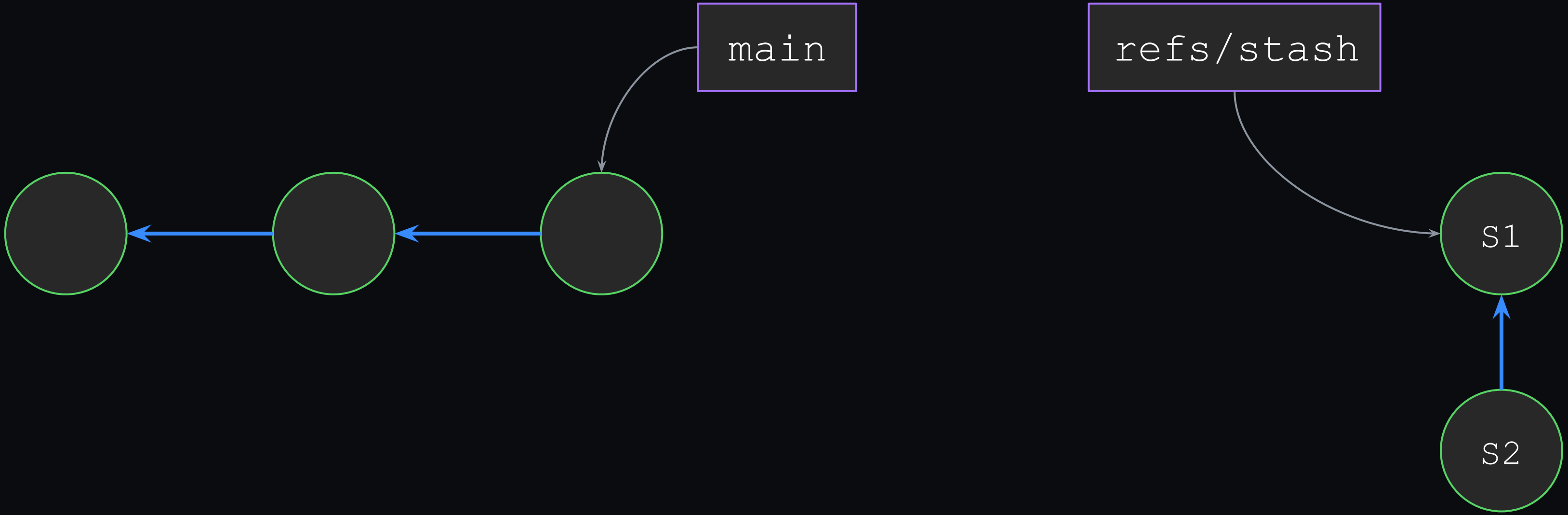
git stash



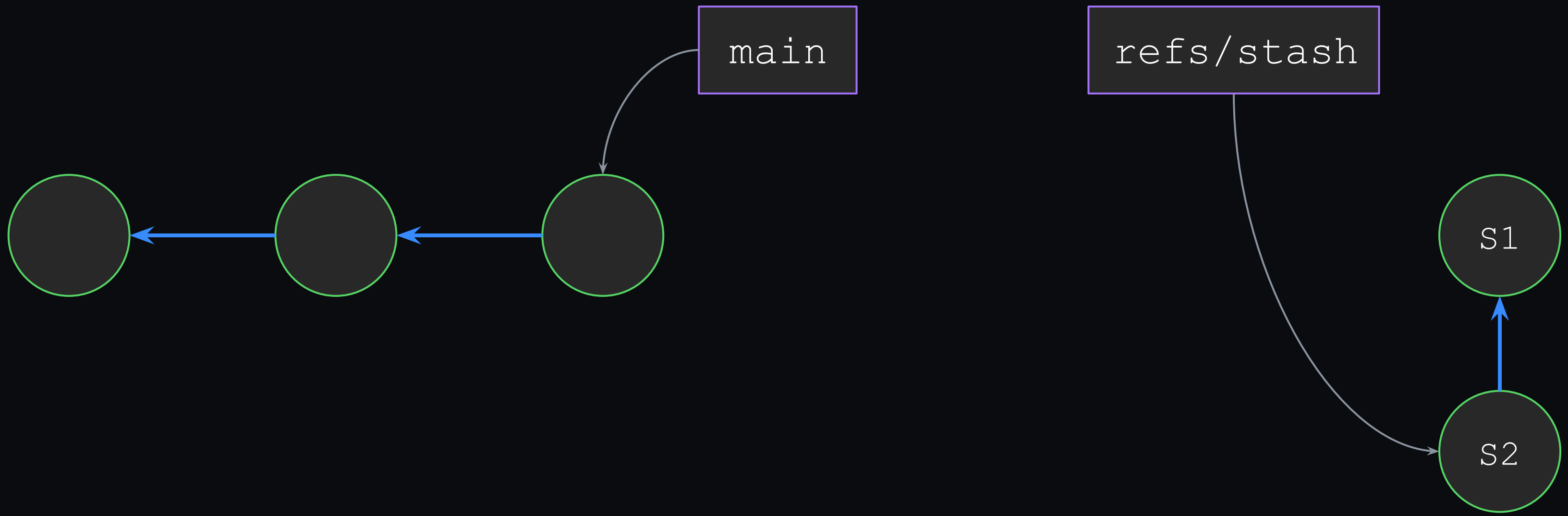
git stash



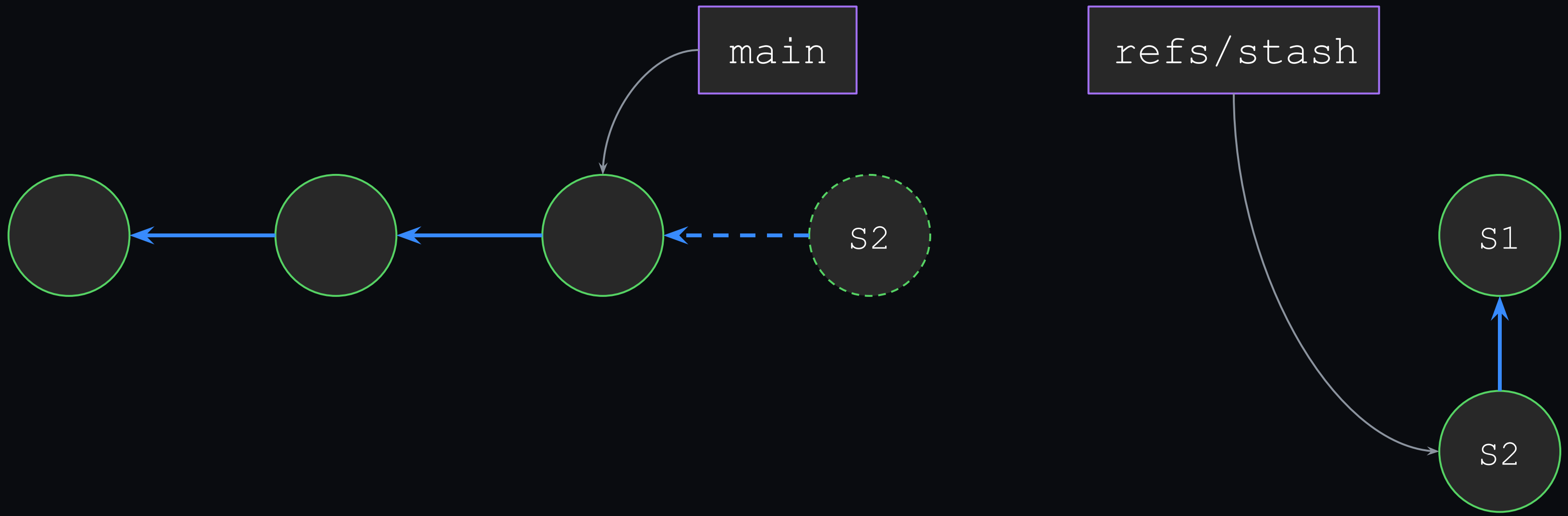
git stash



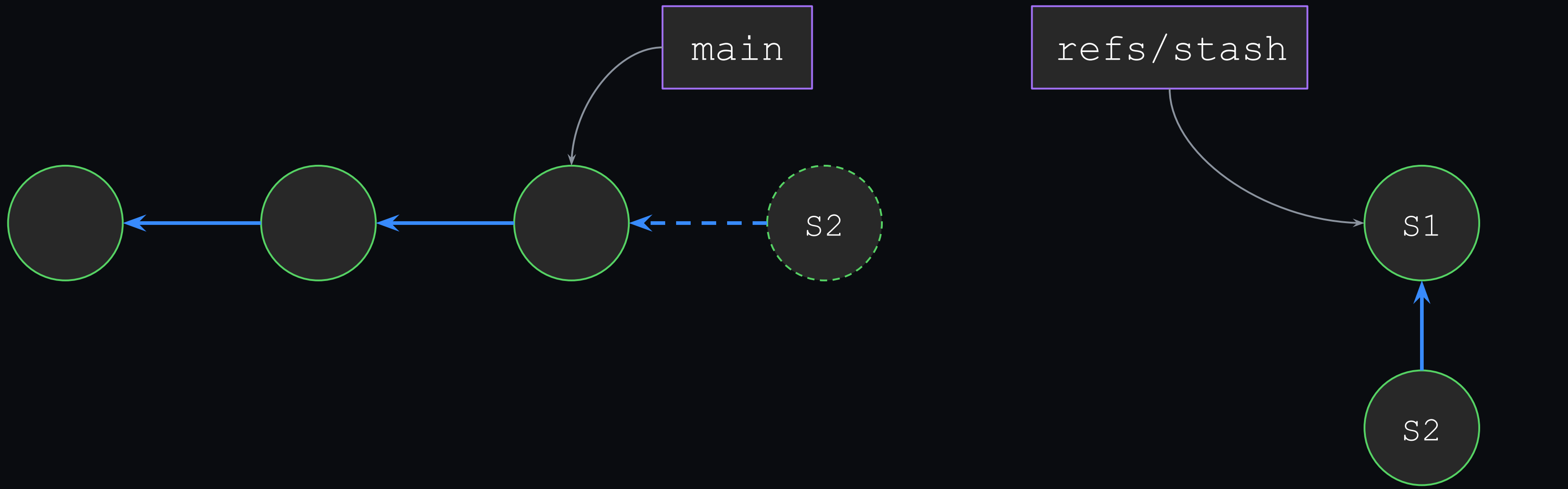
git stash



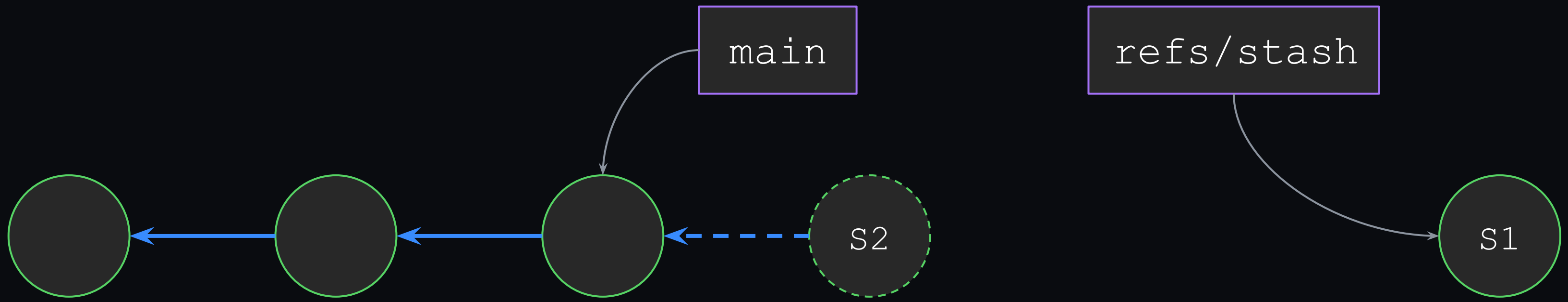
git stash



git stash



git stash



`git stash`

To summarize:

- Takes any pending changes in your working copy
- Creates a new commit containing those changes
- Tags it with a temporary reference called `refs/stash`

Lots of ways to interact with your stash:

- Add to it: `git stash`, or `git stash push`
- Remove from it: `git stash pop`
- List its contents: `git stash list`

...and, as usual, lots more: https://git-scm.com/docs/git-stash :-)

Q&A

<ttaylorr@github.com>