24. Reshaping history

In the last chapter we built the `cherry-pick` command, which lets us copy a set of commits from their original place in the history graph onto the tip of another branch. We'll now see how this ability can be used to carry out most of Git's other history manipulation commands.

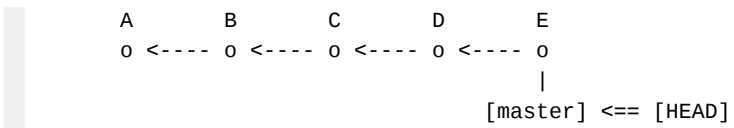
24.1. Changing old commits

The `reset` and `commit` commands by themselves allow us to replace a sequence of commits at the end of a branch. But, what if we wanted to change commits that are few steps behind the current `HEAD`? For example, we might want to amend the content or message of an old commit, or place commits in a different order, or drop them from the history entirely. Let's see how we can use cherry-picking to accomplish these tasks.

24.1.1. Amending an old commit

Imagine we have the following history, where `master` is checked out:

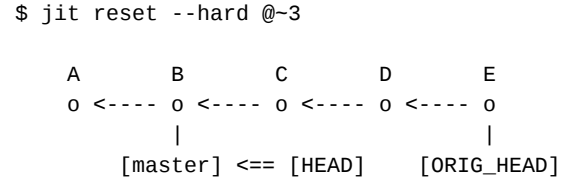
Figure 24.1. Sequence of five commits



It turns out that the code we committed in commit *B* was lacking tests, and we'd like to go back and add some, so the code and its tests are recorded in the same place. The aim is to produce a copy of the above history, in which *B* has been altered. That is, we want to produce a commit that has *A* as its parent, and is followed by copies of *C*, *D* and *E*. These copies will introduce the same *changes* as *C*, *D* and *E*, rather than having the same *content*; they should include whatever new content we introduce into the modified copy of *B*.

We'll begin by performing a hard reset to commit *B*. This moves the current branch pointer to *B* and updates the index and workspace to match. The repo now reflects the content stored in *B*, and `ORIG_HEAD` points at *E*.

Figure 24.2. Resetting `HEAD` to the target commit



Since we now have commit *B* checked out, we can add the tests we want, add the updated files to the index, and use `commit --amend` to replace `B` with a modified version, *B'*.

Figure 24.3. Amending the target commit

```
$ jit add .
$ jit commit --amend

    A       B       C       D       E
    o <---- o <---- o <---- o <---- o
     \                              |
      \                          [ORIG_HEAD]
       \ B'
        o
        |
    [master] <== [HEAD]
```

We can now replay the rest of the commits on top of *B*' using `cherry-pick`; since we originally reset to revision `@~3`, the range `ORIG_HEAD~3..ORIG_HEAD` will give us the commits we want, producing commits *C*', *D*' and *E*' which contain the changes d_{BC}, d_{CD} and d_{DE} respectively.

Figure 24.4. Cherry-picking the remaining history

```
$ jit cherry-pick ORIG_HEAD~3..ORIG_HEAD

    A       B       C       D       E
    o <---- o <---- o <---- o <---- o
     \
      \
       \ B'     C'      D'      E'
        o <---- o <---- o <---- o
                                |
                            [master] <== [HEAD]
```

The current branch pointer now refers to the tip of the modified history. Remember that since each commit contains its parent's ID, replacing an old commit means generating new copy of all the downstream commits. Above, we need a copy of *C* with its `parent` field replaced by the ID of *B*', a copy of *D* whose `parent` is *C*', and so on. Even if we only changed the message of *B* and not its tree, we'd need to make copies of all the downstream commits, because changing a commit's `parent` will change that commit's own ID.

So, although it may only appear that we've modified commit *B*, we have in fact generated a whole new history that diverges from the parent of *B*. The original history still exists, but may no longer have any refs pointing at it. It's important to remember this distinction between commits that have the same content or diff, but are actually distinct objects in the history, as it affects what happens when you come to merge such modified branches later.

24.1.2. Reordering commits

Part of managing your Git history involves arranging commits so they tell a clear story of the project, so people can see how and why the code has been changed over time. For this reason you may want to reorder the commits on a branch before sharing that branch with your team.

Let's say we have the following history containing six commits.

Figure 24.5. Sequence of six commits

```
    A       B       C       D       E       F
    o <---- o <---- o <---- o <---- o <---- o
                                            |
                                        [master] <== HEAD
```

We've decided that commits *B* and *C* should be in the opposite order. When we were writing these commits, we had a lot of uncommitted work and then added it in small pieces, breaking it into lots of small commits. However it turns out that the code in *B* relies on a function that wasn't committed until *C*, so this version of the codebase won't build and may confuse anyone reading the history later. We'd like to effectively swap *B* and *C* so their changes appear in a workable order.

For this workflow, we won't rely on `ORIG_HEAD`, because we'll need to use `cherry-pick` multiple times, and if we decide to abort, that will overwrite `ORIG_HEAD`. So rather than a hard reset, we'll check out a branch at the commit before the ones we want to swap. `HEAD` now points at *A* and the index and workspace reflect that.

Figure 24.6. Creating a fixup branch

```
$ jit branch fixup @~5
$ jit checkout fixup

    A       B       C       D       E       F
    o <---- o <---- o <---- o <---- o <---- o
    |                                       |
 [fixup] <== [HEAD]                      [master]
```

We want to end up with a history where *B* and *C* have swapped places, which we can do by cherry-picking *C*, then *B*, then the rest of the commits after *C*. Note that we can't do this by running `cherry-pick master~3 master~4 master~3..master`, because the use of the range in the last argument means first two arguments won't actually get picked. We need to pick individual commits, and then a range to finish things off.

Figure 24.7. Cherry-picking the reordered commits

```
$ jit cherry-pick master~3 master~4

    A       B       C       D       E       F
    o <---- o <---- o <---- o <---- o <---- o
     \                                      |
      \                                  [master]
       \
        \ C'     B'
         o <---- o
                 |
              [fixup] <== [HEAD]
```

This cherry-pick has effectively swapped *B* and *C*, producing *C'* and *B'*. When we reorder commits, we'll get a conflict if they don't commute[1]. But, even if the commits do commute textually, reordering might result in versions of the codebase that don't run, because one commit functionally depended on another. Always make sure your commits continue to build after amending the history.

Next, we cherry-pick the rest of the commits using a range, beginning with the latest commit we reordered:

[1] Section 18.4.1, "Concurrency, causality and locks"

Figure 24.8. Cherry-picking the remaining history

```
$ jit cherry-pick master~3..master

    A       B       C       D       E       F
    o <---- o <---- o <---- o <---- o <---- o
     \                                      |
      \                                   [master]
       \
        \ C'     B'      D'      E'      F'
         o <---- o <---- o <---- o <---- o
                                         |
                                      [fixup] <== [HEAD]
```

And finally, we can point our original branch at this new history by checking it out, resetting to the `fixup` branch, and then deleting that branch as we no longer need it.

Figure 24.9. Resetting the original branch

```
$ jit checkout master
$ jit reset --hard fixup
$ jit branch -D fixup

    A       B       C       D       E       F
    o <---- o <---- o <---- o <---- o <---- o
     \                                      |
      \                                  [ORIG_HEAD]
       \
        \ C'     B'      D'      E'      F'
         o <---- o <---- o <---- o <---- o
                                         |
                                      [master] <== [HEAD]
```

You can use this technique to arbitrarily reorder commits, drop commits by not cherry-picking them into the new history, amend old commits, and so on. Next we'll look at how Git's other history manipulation tools can be built on top of this operation.

24.2. Rebase

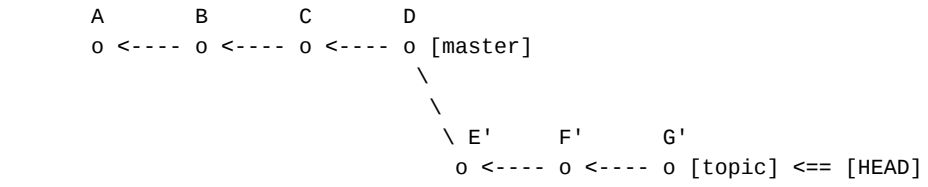
Git's `rebase` command is used to perform all sorts of changes to a project's history. It is highly configurable, but in its default form its job is to take a history that looks like this:

Figure 24.10. History with two divergent branches

```
    A       B       C       D
    o <---- o <---- o <---- o [master]
             \
              \
               \ E      F       G
                o <---- o <---- o [topic] <== [HEAD]
```

And reshape it so that your current branch effectively forks off from the end of some other branch, rather than its original start point. For example, running `git rebase master` on the above history will produce this outcome:

Figure 24.11. Branch after rebasing to the tip of `master`

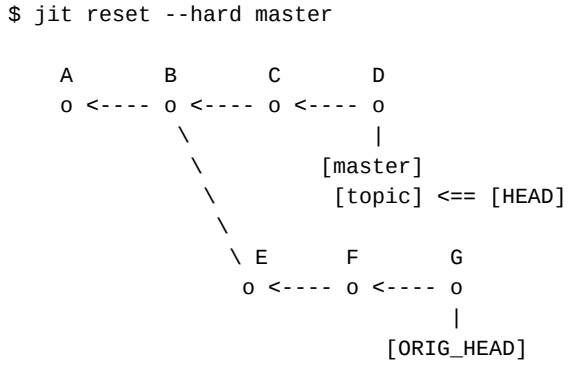


It's called *rebasing* because you are literally changing the commit your branch is based on; detaching it from its original start point and attaching it at the end of a branch so its history incorporates all the changes from that branch. This is often done to keep the history clean (avoiding merge commits that don't add any meaningful information), or to sort out any potential merge conflicts with another branch before merging into it.

The documentation[2] will tell you that the `rebase` command saves the details of all the commits on your branch (`topic` in our example) that aren't on the *upstream* branch — that's the branch you're rebasing onto, `master` in the above example. Then it resets your current branch to the tip of the upstream branch, and replays the saved commits on top of it. With our knowledge of the building blocks we have so far, we can translate this into a couple of commands.

First, from our starting state, we'll do a hard reset to point our branch at the revision we want to rebase onto. This leaves `ORIG_HEAD` pointing at the original tip of our branch.

Figure 24.12. Resetting the current branch to the upstream branch



Then we want to replay the commits from `topic` that aren't on `master`, on top of `master`. Since `HEAD` is now pointing at the upstream branch, we can select the required commits with the range `..ORIG_HEAD`, and we can use `cherry-pick` to replay these commits.

[2] https://git-scm.com/docs/git-rebase

Figure 24.13. Cherry-picking the branch onto the upstream

```
$ jit cherry-pick ..ORIG_HEAD

    A       B       C       D       E'      F'      G'
    o <---- o <---- o <---- o <---- o <---- o <---- o
             \              |                       |
              \         [master]                [topic] <== [HEAD]
               \
                \
                 \ E      F       G
                  o <---- o <---- o
                                  |
                             [ORIG_HEAD]
```

And hey presto, we now have an equivalent of our original branch, but incorporating *C* and *D* rather than diverging at *B*. As usual, the original commits still exist and can be accessed via the `ORIG_HEAD` reference.

24.2.1. Rebase onto a different branch

A common variation is to use the `--onto` option to transplant the commit range onto a different starting commit. For example, say we have the following history in which we forked off from the `topic` branch to fix a bug, which we did using commits *G* and *H*.

Figure 24.14. History with three chained branches

```
    A       B       C       D
    o <---- o <---- o <---- o [master]
             \
              \
               \ E      F
                o <---- o [topic]
                         \
                          \
                           \ G      H
                            o <---- o [bug-fix] <== [HEAD]
```

This bug fix doesn't actually depend on the work in `topic` and we'd like to transplant it onto `master`. The `--onto` option can do just that:

Figure 24.15. Rebase onto a different branch

```
$ git rebase --onto master topic

    A       B       C       D
    o <---- o <---- o <---- o [master]
             \               \
              \               \
               \ E      F      \ G'     H'
                o <---- o       o <---- o
                        |               |
                     [topic]         [bug-fix] <== [HEAD]
```

In general, `rebase --onto <rev1> <rev2>` makes Git reset the current branch to `<rev1>`, and then replay all the commits on the original branch that aren't reachable from `<rev2>`.

The revision arguments don't have to be branch names, they can be any revision specifier. And so, this is equivalent to running `reset --hard <rev1>` followed by `cherry-pick <rev2>..ORIG_HEAD`. First we do a hard reset:

Figure 24.16. Resetting to the target branch

```
$ jit reset --hard master

    A       B       C       D
    o <---- o <---- o <---- o [master]
             \                [bug-fix] <== [HEAD]
              \
               \ E      F
                o <---- o [topic]
                         \
                          \
                           \ G      H
                            o <---- o [ORIG_HEAD]
```

Then we cherry-pick the required commits:

Figure 24.17. Cherry-picking the original branch

```
$ jit cherry-pick topic..ORIG_HEAD

    A       B       C       D
    o <---- o <---- o <---- o [master]
             \                \
              \                \
               \ E      F       \ G'     H'
                o <---- o        o <---- o [bug-fix] <== [HEAD]
                       / \
                 [topic]  \
                           \ G      H
                            o <---- o [ORIG_HEAD]
```

So we can see that `cherry-pick` can be used to arbitrarily transplant a range of commits to any other point in the graph. The real `rebase` command can do much more than this and deal with other complications in the history, for example discarding or preserving merge commits, and dropping commits whose changes are already present in the target branch. However, the core functionality in most cases can be done with a `reset` and a `cherry-pick`.

24.2.2. Interactive rebase

Git's `rebase --interactive` command[3] provides the ability to make more complicated changes to the history. It will select the commit range to be transplanted and then present this list to you in your editor, letting you choose what should be done with each commit. The commits can be arbitrarily reordered, and a range of commands can be applied to each one. A few of these commands are straightforward to replicate using what we already know.

For example, `pick` just cherry-picks the given commit. `drop` does nothing; the given commit is not cherry-picked, and this is equivalent to deleting it from the list. `reword` cherry-picks a

[3] https://git-scm.com/docs/git-rebase#_interactive_mode

commit but opens the editor for you to change the commit message. `edit` cherry-picks the commit but then pauses to let you make arbitrary changes — amending the `HEAD` commit, adding some new commits of your own, etc. — before continuing with the rest of the list.

There are a couple of commands that are a little more complicated, but can still be replicated using our existing tools: `squash` and `fixup`.

We have already seen that a combination of `reset` and `commit` can be used to squash the last few commits on a branch[4]. The `squash` command in `rebase` works very similarly, it just lets you deal with commits that are deeper in the history, so a little extra work is needed. Let's say we have the following history and want to squash commit *C* into *B*.

Figure 24.18. History before squashing

```
A       B       C       D
o <---- o <---- o <---- o
                        |
                    [master] <== [HEAD]
```

What this means is that we want commits *B* and *C* to be effectively replaced by a single commit containing the same tree as *C*, with the rest of the history following on after. So first, we need to get the index into the state of commit *C*, and we'll do that by checking out a temporary branch at that position.

Figure 24.19. Checking out the desired tree

```
$ jit branch squash @^
$ jit checkout squash

    A       B       C       D
    o <---- o <---- o <---- o
                    |       |
                    |   [master]
                    |
                [squash] <== [HEAD]
```

Now we proceed as before: a soft reset points `HEAD` at the previous commit, but leaves the index unchanged, so it still contains the tree of *C*.

Figure 24.20. Resetting to the parent commit

```
$ jit reset --soft @^

    A       B       C       D
    o <---- o <---- o <---- o
            |               |
            |           [master]
            |
        [squash] <== [HEAD]
```

Now, we can use `commit --amend` to replace *B* with a new commit whose tree is that of *C*, but we're given the message from *B* as a starting point in the editor.

[4] Section 22.3, "Reusing messages"

Figure 24.21. Amending the parent commit to contain the squashed changes

```
$ jit commit --amend

    A       B       C       D
    o <---- o <---- o <---- o
     \                      |
      \                 [master]
       \
        \ B+C
         o
         |
      [squash] <== [HEAD]
```

We can then cherry-pick the rest of the branch to complete the history.

Figure 24.22. Cherry-picking the remaining history

```
$ jit cherry-pick master

    A       B       C       D
    o <---- o <---- o <---- o
     \                      |
      \                 [master]
       \ B+C    D'
        o <---- o
                |
            [squash] <== [HEAD]
```

Finally, we reset our original branch to point at this new history, and delete the temporary branch.

Figure 24.23. Cleaning up branch pointers after squashing

```
$ jit checkout master
$ jit reset --hard squash
$ jit branch -D squash

    A       B       C       D
    o <---- o <---- o <---- o
     \                      |
      \                [ORIG_HEAD]
       \ B+C    D'
        o <---- o
                |
            [master] <== [HEAD]
```

Git's `squash` command actually combines the messages of *B* and *C* when creating the *B+C* commit, but I'll leave that as an exercise for the reader. The above illustrates what is happening at the level of the history and the contents of commits.

The `fixup` command is described as being just like `squash`, except it only keeps the first commit's message, not that of the squashed commit. However I tend to use it slightly differently; I usually reach for it when I notice a commit far back in the history needs to be changed. I'll make a commit on top of the current `HEAD` that includes the additional changes, and then use `rebase` to move this fix-up commit back through the history so it follows the commit I want to change. I can then combine them with the `fixup` command. You can absolutely use

`squash` to do this, it's just a slightly different use case to combining two commits that are already adjacent.

This workflow combines a reordering with the squash technique we've just seen. Let's start with our familiar linear history:

Figure 24.24. History before a fix-up

```
A       B       C
o <---- o <---- o
                |
            [master] <== [HEAD]
```

Imagine that we've noticed that commit *B* isn't quite what we want, and we'd like to change its contents. We can begin by writing a new commit *D* than contains the amendments we'd like to apply to *B*.

Figure 24.25. Creating a fix-up commit

```
$ jit commit --message "D"

    A       B       C       D
    o <---- o <---- o <---- o
                            |
                        [master] <== [HEAD]
```

Now, we want to combine commits *B* and *D*, which means creating a commit whose tree is $T_B + d_{CD}$ — the tree from *B* plus the change introduced by *D*. Another way to think of this is that we want to squash *D* into *B*, but first we need to relocate *D* so it's next to *B*.

Let's start a new branch at *B*:

Figure 24.26. Starting a fix-up branch

```
$ jit branch fixup @~2
$ jit checkout fixup

    A       B       C       D
    o <---- o <---- o <---- o
            |               |
            |           [master]
            |
         [fixup] <== [HEAD]
```

Then, we can cherry-pick *D* onto this branch. The tree of this commit *D'* will be $T_B + d_{CD}$ as required.

Figure 24.27. Cherry-picking the fix-up commit

```
$ jit cherry-pick master

    A       B       C       D
    o <---- o <---- o <---- o
             \              |
              \ D'      [master]
               o
               |
            [fixup] <== [HEAD]
```

We can now squash *B* and *D'* together using the procedure we used above. We soft-reset so that `HEAD` points at *B* but the index retains the content of *D'*. We then use `commit --amend` to commit this tree with *A* as the parent, keeping the message from *B*.

Figure 24.28. Creating a squashed commit containing the fix-up

```
$ jit reset --soft @^
$ jit commit --amend

    A       B       C       D
    o <---- o <---- o <---- o
     \       \               |
      \       \ D'       [master]
       \       o
        \
         \ B+D'
          o
          |
       [fixup] <== [HEAD]
```

Finally, we cherry-pick the remaining history — commit *C* — onto our temporary branch, reset our original branch to this new history, and delete the temporary branch.

Figure 24.29. History following a relocated fix-up commit

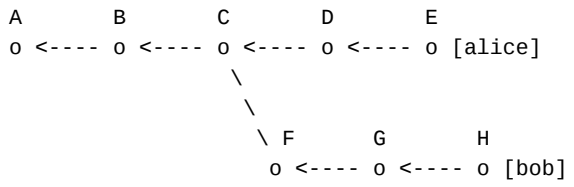
```
$ jit cherry-pick master^
$ jit checkout master
$ jit reset --hard fixup
$ jit branch -D fixup

    A       B       C       D
    o <---- o <---- o <---- o
     \       \               |
      \       \ D'     [ORIG_HEAD]
       \       o
        \
         \ B+D'   C'
          o <---- o
                  |
              [master] <== [HEAD]
```

The `rebase` command is a much more direct way of performing these changes and can do much more besides, but conceptually most of its behaviour can be replicated with commands we already have, if a little laboriously. This process illustrates that although commits primarily store *content* rather than *changes*, it is possible to treat a commit as the implied difference between its content and that of its parent. These changes can be arbitrarily recombined using the merge machinery to make complex changes to the project history.

24.3. Reverting existing commits

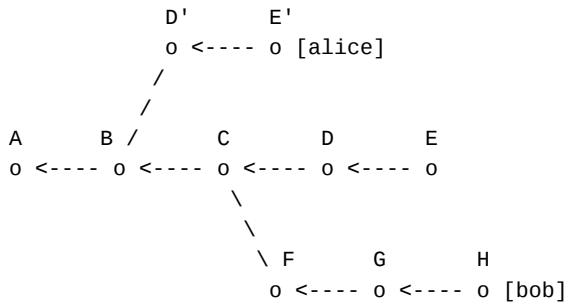
It's certainly useful to be able to edit the history of your branch, but it becomes a problem once you've shared your history with other teammates. As an example, let's imagine two coworkers, Alice and Bob, that are each working on their own branch of a project. The last commit each developer has in common is *C*.

Figure 24.30. History with concurrent edits



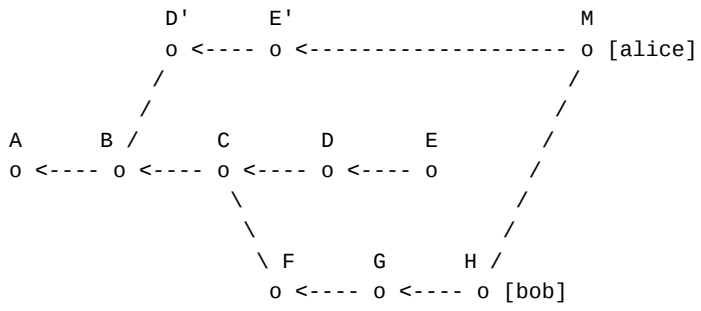
Now, suppose Alice decides *C* should not have been committed, and wants it removed from the history. She resets to *B* and then cherry-picks *D* and *E* to construct the history she actually wants.

Figure 24.31. Alice removes a shared commit from the history



The problem is that *C* has already been shared with another developer; Bob has this commit in his history, and his branch now diverges from Alice's at *B*. When Alice goes to merge in Bob's changes, this is the resulting history:

Figure 24.32. Merging reintroduces a dropped commit



The base of merge *M* is commit *B*, and so the changes from commits *C*, *F*, *G* and *H* will be incorporated. Alice ends up with a commit that includes content she thought she'd removed!

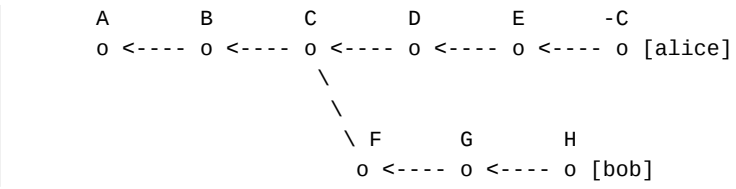
It's important to note at this point that Git is not doing anything wrong here. Git is merely a tool for tracking and reconciling concurrent edits to a set of files, and if the history says that *C* is a concurrent edit on one branch, rather than a commit shared by both branches, then that's how Git will treat it. The problem is that the history is not a good representation of Alice's intentions and understanding of the project. If she wants all the developers to agree that *C* should be removed, she either needs to ask everyone to rebase their branches, or include this information in a better way in the history.

This type of problem arises whenever you rebase commits that have already been fetched by other developers. If you change the history of commits you've pushed, everyone needs to migrate their changes onto your replacement commits, and this is really hard to get right. It's

much easier to manage if you commit on top of the existing shared history, with commits that effectively undo the commits you wanted to remove.

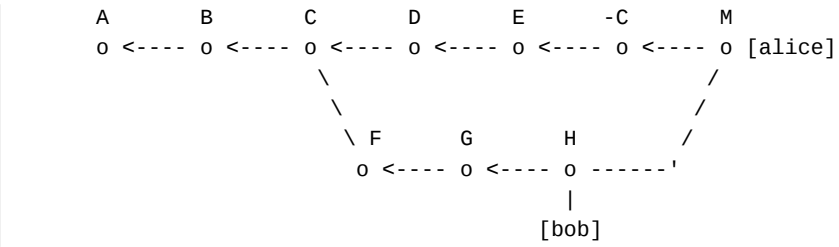
Let's now imagine that Alice does this. She writes a commit, which we'll call *–C*, that removes the content that was added in *C*.

Figure 24.33. Committing to undo earlier changes



Now, when Alice merges from Bob's branch, the base is *C*, their original shared commit, and so only commits *F*, *G* and *H* are incorporated. The changes from *C* are not reintroduced into the project.

Figure 24.34. Merging does not reintroduce the removed content



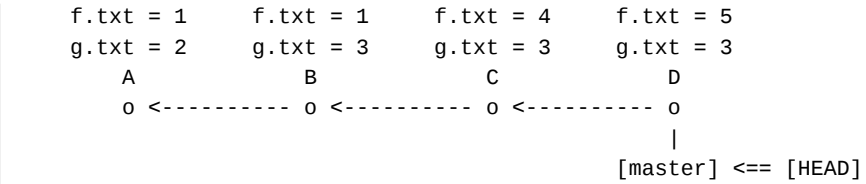
There are many reasons you'd want to make a change like this. Maybe some temporary code was added to facilitate a long refactoring, and can now be removed. Maybe a feature flag was used to incrementally roll a feature out, and is no longer necessary. Maybe you added someone's public key to your config management repository and they have since left the company. These are all situations where you don't want to expunge the content from history, you just want it not to exist in your latest version. Since this is a fairly common requirement, Git includes a command so that you don't have to construct this *inverse commit* by hand: the `revert` command.

24.3.1. Cherry-pick in reverse

The `revert` command is much like `cherry-pick`, it just applies the inverse of the changes in the given commits. In fact if you read their documentation you'll see that both commands take the same options and look almost identical in functionality. This is not a coincidence; in fact the two commands are deeply similar and differ only in how they use the `Merge` module to apply changes.

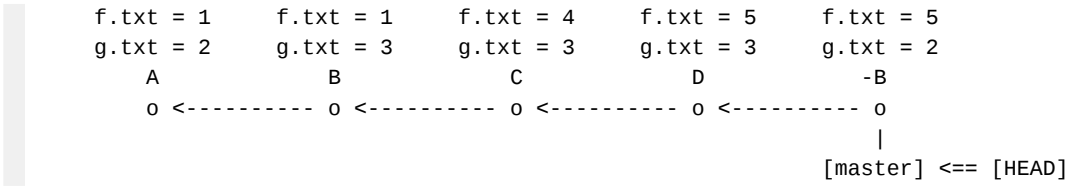
Consider the following history in which each commit modifies one of two files.

Figure 24.35. History with two files



We'd like to revert commit *B*, that is, undo the change of `g.txt` so that it contains its original value `2`.

Figure 24.36. Reverting an old change



What we're doing here is taking the change introduced by *B*, that is d_{AB} = `{ g.txt => [2, 3] }`, and inverting it to get `{ g.txt => [3, 2] }`. This is applied to commit *D* to undo the change to `g.txt`. If commit *C* or *D* had changed `g.txt` again to some other value, then this change would not apply, and we should get a conflict.

The tree we want to end up with in the revert commit $-B$ is $T_{-B} = T_D - d_{AB}$, that is the tree from *D* with the effect of *B* removed. Thus far we've only considered *adding* changes to trees, so what does it mean to remove them? Well, if d_{AB} is the change required to transform T_A into T_B, then the inverse change $-d_{AB}$ should be the change that converts T_B into T_A. That is, $-d_{AB} = d_{BA}$. To calculate the inverse change from a commit, we can just swap the order of the arguments when generating the tree diff.

Whereas `cherry-pick` applies d_{AB} by performing a merge between *B* and *D* with *A* as the base, `revert` merges *A* and *D* with *B* as the base. If that sounds weird, remember from the previous chapter that `Merge::Resolve` does not care about the historical relationship between its inputs, so we are free to 'undo' a commit by using it as the base for a merge involving its parent.

In this case, the differences on each side of the merge are d_{BA} = `{ g.txt => [3, 2] }` and d_{BD} = `{ f.txt => [1, 5] }`. These do not overlap and so the end result is $T_{-B} = T_D + d_{BA}$ = `{ f.txt => 5, g.txt => 2 }`.

24.3.2. Sequencing infrastructure

Since the `revert` and `cherry-pick` commands are so similar and differ only in some minor details, we can reuse much of the `Command::CherryPick` class so that both commands share the same implementation. As a matter of fact, the `CherryPick` class is already separated into a set of methods that are specific to cherry-picking, and many that aren't. I'm going to extract the latter set into a module called `Command::Sequencing` that we can use as a base for both commands. It contains the option definitions and the `run` method that describes the overall flow of the command, and various glue methods that execute the `Merge::Resolve` given a set of inputs, handle stopping the command on conflicts, and so on.

```
# lib/command/shared/sequencing.rb

module Command
  module Sequencing

    CONFLICT_NOTES = <<~MSG
      ...
    MSG
```

```
      def define_options
      def run
      def sequencer
      def resolve_merge(inputs)
      def fail_on_conflict(inputs, message)
      def finish_commit(commit)
      def handle_continue
      def resume_sequencer
      def handle_abort
      def handle_quit

    end
  end
```

This leaves a few methods in `CherryPick` that are specific to this command. `merge_type` is used by `fail_on_conflict`, `handle_abort` and `handle_quit` to trigger the right commit type in `PendingCommit`. `store_commit_sequence` expands the arguments into a list of commits and stores them, and this will differ for `revert`, as we'll see. `pick` and `pick_merge_inputs` deal with constructing the inputs for `Merge::Resolve` and building the resulting commit.

```
  # lib/command/cherry_pick.rb

  module Command
    class CherryPick < Base

      include Sequencing
      include WriteCommit

      private

      def merge_type
      def store_commit_sequence
      def pick(commit)
      def pick_merge_inputs(commit)

    end
  end
```

To support the `revert` command, the sequencer needs a bit of extra functionality. In `cherry-pick`, each line in `.git/sequencer/todo` begins with the word `pick`, whereas in `revert`, each line begins with `revert`; when we resume the sequencer we need to know which command to use on each commit. Let's add a method to `Sequencer` called `revert`, and make the `pick` and `revert` methods store the name of the command, not just the given commit.

```
  # lib/repository/sequencer.rb

      def pick(commit)
        @commands.push([:pick, commit])
      end

      def revert(commit)
        @commands.push([:revert, commit])
      end
```

The `dump` and `load` methods need to be similarly updated to parse the command at the beginning of each line, rather assuming all of them will be `pick`.

```
# lib/repository/sequencer.rb

    def dump
      return unless @todo_file

      @commands.each do |action, commit|
        short = @repo.database.short_oid(commit.oid)
        @todo_file.write("#{ action } #{ short } #{ commit.title_line }")
      end

      @todo_file.commit
    end

    def load
      open_todo_file
      return unless File.file?(@todo_path)

      @commands = File.read(@todo_path).lines.map do |line|
        action, oid, _ = /^(\S+) (\S+) (.*)$/.match(line).captures

        oids   = @repo.database.prefix_match(oid)
        commit = @repo.database.load(oids.first)
        [action.to_sym, commit]
      end
    end
```

To complete the sequencing infrastructure, the `resume_sequencer` in `Command::Sequencing` needs to inspect the action name on each command it fetches, and call either `pick` or `revert` depending on the result.

```
# lib/command/shared/sequencing.rb

    def resume_sequencer
      loop do
        action, commit = sequencer.next_command
        break unless commit

        case action
        when :pick   then pick(commit)
        when :revert then revert(commit)
        end
        sequencer.drop_command
      end

      sequencer.quit
      exit 0
    end
```

We're now ready to add the `revert` command itself.

24.3.3. The `revert` command

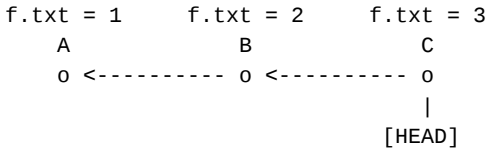
The `Command::Revert` class needs to follow the same template set by `Command::CherryPick` in order to work with the `Sequencing` module. We need to define `merge_type`, `store_commit_sequence`, and then a `revert` method to be called by `resume_sequencer`.

```
# lib/command/revert.rb
```

```
module Command
  class Revert < Base

    include Sequencing
    include WriteCommit

    private

    def merge_type
      :revert
    end
```

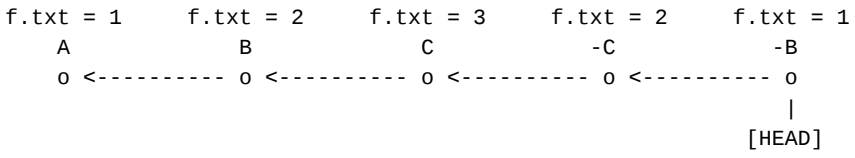
First, let's tackle storing the commit sequence. `cherry-pick` iterates over the input range from the oldest commit forwards, so the changes are applied in the same order they were originally. Since the `revert` commit reverses the changes, they need to be done in the reverse order from how they were originally committed. For example, consider this history:

Figure 24.37. History with non-commutative commits



Running `revert @~2..` should revert the latest two commits. The relevant diffs here are d_{BA} = `{ f.txt => [2, 1] }` and d_{CB} = `{ f.txt => [3, 2] }`. A diff will only apply on top of `HEAD` if its pre-image is the same as the state of `HEAD`; d_{BA} cannot be applied because its pre-image for `f.txt` is `2`, whereas `HEAD` has `f.txt = 3`. We need to apply d_{CB} to replace `3` with `2`, and then d_{BA} to replace `2` with `1`.

Figure 24.38. Reverting the last two commits



To have the best chance of a clean merge each time, we revert commits from the latest commit backwards. `RevList` iterates commits in this order, so whereas `CherryPick` iterates `RevList` in reverse, `Revert` uses the normal iteration order.

```
    # lib/command/revert.rb

    def store_commit_sequence
      commits = RevList.new(repo, @args, :walk => false)
      commits.each { |commit| sequencer.revert(commit) }
    end
```

Next, we define the `revert` method that `resume_sequencer` will call to revert each commit. This is similar to the `pick` method, with a few differences. It uses its own helper method `revert_merge_inputs` to construct the inputs for the `Merge::Resolve`, and it constructs a new

commit message rather than reusing the one from the picked commit. It also lets the user edit the message before saving the commit.

```
# lib/command/revert.rb

    def revert(commit)
      inputs  = revert_merge_inputs(commit)
      message = revert_commit_message(commit)

      resolve_merge(inputs)
      fail_on_conflict(inputs, message) if repo.index.conflict?

      author  = current_author
      message = edit_revert_message(message)
      picked  = Database::Commit.new([inputs.left_oid], write_tree.oid,
                                     author, author, message)

      finish_commit(picked)
    end
```

The `revert_merge_inputs` method embodies the core difference between `revert` and `cherry-pick`. It's almost the same as `pick_merge_inputs`, but it uses the picked commit as the base of the merge, and its parent as the `right` input. Swapping these two arguments is all it takes to undo a commit.

```
# lib/command/revert.rb

    def revert_merge_inputs(commit)
      short = repo.database.short_oid(commit.oid)

      left_name  = Refs::HEAD
      left_oid   = repo.refs.read_head
      right_name = "parent of #{ short }... #{ commit.title_line.strip }"
      right_oid  = commit.parent

      ::Merge::CherryPick.new(left_name, right_name,
                              left_oid, right_oid,
                              [commit.oid])
    end
```

The `revert_commit_message` and `edit_revert_message` helpers construct the default revert commit message, and invoke the editor to let the user change it if desired.

```
# lib/command/revert.rb

    def revert_commit_message(commit)
      <<~MESSAGE
        Revert "#{ commit.title_line.strip }"

        This reverts commit #{ commit.oid }.
      MESSAGE
    end

    def edit_revert_message(message)
      edit_file(commit_message_path) do |editor|
        editor.puts(message)
        editor.puts("")
        editor.note(Commit::COMMIT_NOTES)
```

```
      end
    end
```

Having completed the `Revert` command class, we need to adjust a few bits of supporting code so that we can resume a revert if it causes a merge conflict. Just as the `cherry-pick` command stores pending commits in the file `.git/CHERRY_PICK_HEAD`, `revert` uses `.git/REVERT_HEAD`. We just need to add an entry to `PendingCommit::HEAD_FILES` to reflect this and pass the type argument `:revert` when we store pending commits.

```
# lib/repository/pending_commit.rb

    HEAD_FILES = {
      :merge       => "MERGE_HEAD",
      :cherry_pick => "CHERRY_PICK_HEAD",
      :revert      => "REVERT_HEAD"
    }
```

We also need to expand the `resume_merge` method in `WriteCommit` so that if the pending commit type is `:revert`, then we call `write_revert_commit`.

```
# lib/command/shared/write_commit.rb

    def resume_merge(type)
      case type
      when :merge       then write_merge_commit
      when :cherry_pick then write_cherry_pick_commit
      when :revert      then write_revert_commit
      end

      exit 0
    end
```

The `write_revert_commit` method does much the same job as the `write_cherry_pick_commit` method, except that we don't reuse the author or message from the reverted commit. This means we can use the `write_commit` method to build and store the commit, rather than constructing it ourselves.

```
# lib/command/shared/write_commit.rb

    def write_revert_commit
      handle_conflicted_index

      parents = [repo.refs.read_head]
      message = compose_merge_message
      write_commit(parents, message)

      pending_commit.clear(:revert)
    end
```

Finally, to handle the `revert --continue` command, the `handle_continue` method in `Sequencing` needs to invoke this `write_revert_commit` method if the pending commit type is `:revert`.

```
# lib/command/shared/sequencing.rb

    def handle_continue
      repo.index.load
```

```
        case pending_commit.merge_type
        when :cherry_pick then write_cherry_pick_commit
        when :revert      then write_revert_commit
        end

        sequencer.load
        sequencer.drop_command
        resume_sequencer

      rescue Repository::PendingCommit::Error => error
        @stderr.puts "fatal: #{ error.message }"
        exit 128
      end
```

This completes the functionality of the `revert` command, and its sharing of the sequencer code from `cherry-pick` means it can be paused and resumed successfully when a merge conflict occurs.

Although `revert` is useful for quickly removing content, and demonstrates the power of the merge system to reverse as well as apply changes, it is not appropriate for removing things like passwords and other sensitive credentials you've accidentally published to a repository. If you revert such a commit, anybody can still retrieve the content by checking out an older commit.

If you accidentally publish a security credential to a repository, you need to immediately change that password so it cannot be used, and then completely remove it from the repository. In this case, you will need to remove the commit using the technique we saw at the beginning of Section 24.3, "Reverting existing commits", then get the rest of your team to fetch your updated history and rebase their own branches onto it. All branch pointers from which the removed commit is reachable must also be removed; this will prevent the object being transmitted when someone fetches from your repository[5].

24.3.4. Pending commit status

Now that we have various commands that can lead to conflicted states, it would be helpful to display this state to the user. Although `status` does list conflicted files, implying a merge is in progress, it can be hard to remember what kind of merge is happening, how to resume it correctly, and how to escape if it's gone wrong. For this reason, Git includes content in the `status` output to tell you what type of merge is happening, and what state it's in.

Let's add a new step to `Command::Status#print_long_format` that prints this information:

```
    # lib/command/status.rb

      def print_long_format
        print_branch_status
        print_pending_commit_status

        # ...
      end
```

`print_pending_commit_status` is essentially a big `case` statement that decides on some text to display based on whether a merge, cherry-pick or revert is pending, and whether there are

[5] Chapter 28, *Fetching content*

any conflicted files. In the case of a cherry-pick or revert, it also tells you which commit is being picked, as these commands run through a sequence of commits rather than performing a single merge operation.

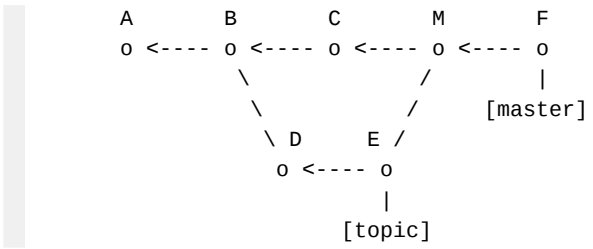
```
# lib/command/status.rb

    def print_pending_commit_status
      case repo.pending_commit.merge_type
      when :merge
        if @status.conflicts.empty?
          puts "All conflicts fixed but you are still merging."
          hint "use 'jit commit' to conclude merge"
        else
          puts "You have unmerged paths."
          hint "fix conflicts and run 'jit commit'"
          hint "use 'jit merge --abort' to abort the merge"
        end
        puts ""
      when :cherry_pick
        print_pending_type(:cherry_pick)
      when :revert
        print_pending_type(:revert)
      end
    end

    def print_pending_type(merge_type)
      oid   = repo.pending_commit.merge_oid(merge_type)
      short = repo.database.short_oid(oid)
      op    = merge_type.to_s.sub("_", "-")

      puts "You are currently #{ op }ing commit #{ short }."

      if @status.conflicts.empty?
        hint "all conflicts fixed: run 'jit #{ op } --continue'"
      else
        hint "fix conflicts and run 'jit #{ op } --continue'"
      end
      hint "use 'jit #{ op } --abort' to cancel the #{ op } operation"
      puts ""
    end

    def hint(message)
      puts "  (#{ message })"
    end
```

24.3.5. Reverting merge commits

Throughout this chapter and the previous one, we've been assuming that every commit being cherry-picked or reverted is a normal commit with a single parent. But it's perfectly possible to cherry-pick merge commits too, and to revert them. Since it uses the `Commit#parent` method, the merge will always use the commit's first parent.

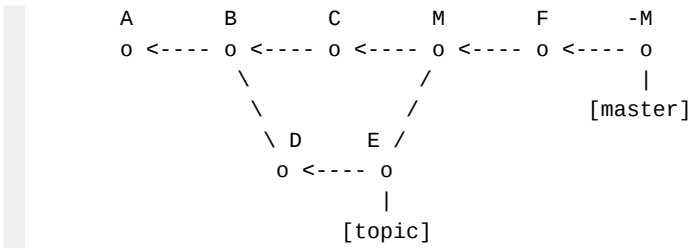
In the case of `revert`, this means the command can be used to effectively undo a merge. Suppose we have the following history in which *M* is a merge commit that was generated by running `merge topic` while `master` was checked out at *C*.

Figure 24.39. Branched and merged history



If we want to undo the merge, we can run `revert master^`, and this will generate a new commit that reverses the effect of *M*.

Figure 24.40. History with reverted merge



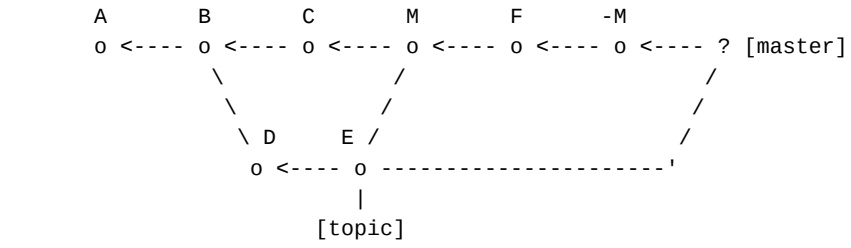
But what is the effect of *M*, and what does that mean for the tree T_{-M}? Well, `revert` will perform a merge between the current `HEAD` (*F*) and the given commit's first parent (*C*), with the commit *M* as the base. So $T_{-M} = T_M + d_{MF} + d_{MC}$, the tree of *M* plus the difference from each side of the merge. Since $T_M + d_{MF} = T_F$ by definition, this simplifies to $T_{-M} = T_F + d_{MC}$. So the effect of $-M$ is to add d_{MC} to T_F.

Now, what is d_{MC}? We know from Section 24.3.1, "Cherry-pick in reverse" that $d_{MC} = -d_{CM}$, the inverse of the change from *C* to *M*. We also know that the merge *M* has $T_M = T_C + d_{BE}$, the tree of *C* plus the net difference from the merged branch. That means d_{BE} is the difference from *C* to *M*, and so $d_{MC} = -d_{BE} = d_{EB}$ — the inverse of the change introduced by the merged branch. That means that reverting *M* undoes the effect of the merge and removes the changes introduced in *D* and *E*, but does *not* remove the change from *C*.

Although for most graph search purposes the order of a commit's parents does not matter, in the case of `cherry-pick` and `revert` they matter a great deal, as they determine which side of the merge we pick changes from. If we revert a commit generated by running `merge <branch>`, the revert undoes the changes from `<branch>` and leaves anything that was reachable from `HEAD` at that point alone.

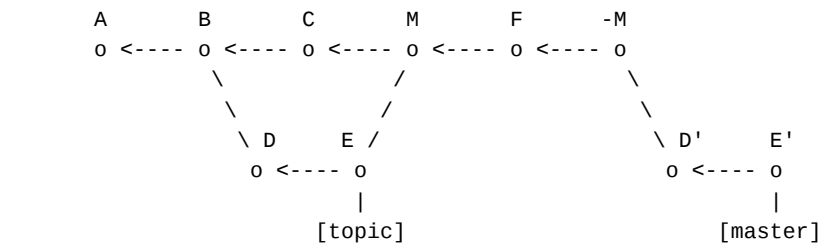
Finally, we saw in the previous chapter that the `cherry-pick` command does not create a `parent` link between the new commit and the one it's derived from, as that would prevent a future true merge from including the whole branch's history. A similar problem occurs if we revert a merge and then later decide we want to bring it back. What happens if we try to run `merge topic` again following our revert commit?

Figure 24.41. Attempting to re-merge a reverted branch



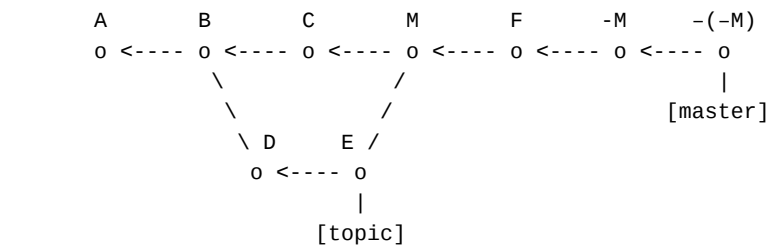
The base of this merge would be the common ancestor of $-M$ and E, which is E itself — E is an ancestor of $-M$. So this merge attempt ends up doing nothing. To get the changes from the `topic` branch back, we can either cherry-pick its commits using a command like `cherry-pick @~3..topic`:

Figure 24.42. Cherry-picking reverted changes



Or, we can revert the commit $-M$: since this commit applies the change d_{EB}, reverting it will produce the change d_{BE}.

Figure 24.43. Reverting a reverted merge



Cherry-pick and revert commits are not special, they're exactly the same as any other commit and contain a pointer to a tree. The differences between those trees can always be recombined in arbitrary ways to effect the desired outcome.

24.4. Stashing changes

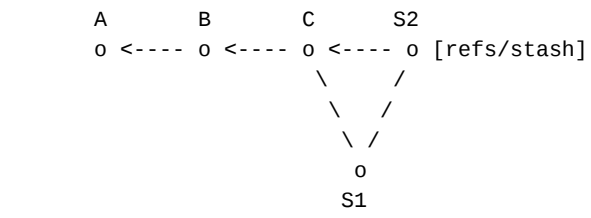
One last application of commits and cherry-picking before we move on. Git includes a command call `stash`[6], which lets you store away uncommitted changes and retrieve them later. We won't implement this command, but it turns out it can easily be simulated with our existing tools.

If you run `git stash`, any changes that have not been committed seem to vanish, and your index and workspace return to the state that matches the current `HEAD`. If you take a look at the file `.git/refs/stash` and the commits it points at, you'll find out that it actually stores the

[6] https://git-scm.com/docs/git-stash

uncommitted state as a pair of commits. Say `HEAD` points at the commit *C* below. If we run `git stash`, the commits S_1 and S_2 will be created, but `HEAD` will remain pointing at *C* and the index and workspace will match it.

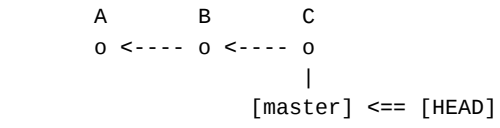
Figure 24.44. Stored pair of stash commits



S_1 has *C* as its parent and has a message like `index on <branch>: <ID> <message>`, while S_2 has both *C* and S_1 as parents and its message says `WIP` (work in progress) instead of `index`. Git has saved that state of the index (your uncommitted changes) and the workspace (your unstaged changes) as commits, and linked them together to show what state they were based on.

We can simulate this process quite straightforwardly. Let's say we begin in the following state, and the index and workspace both differ from `HEAD`.

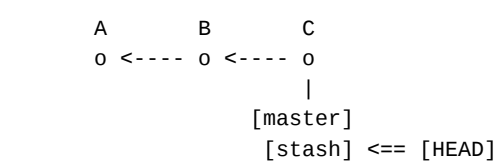
Figure 24.45. Initial work state



First, we'll create a new branch called `stash`, and check it out.

Figure 24.46. Checking out the `stash` branch

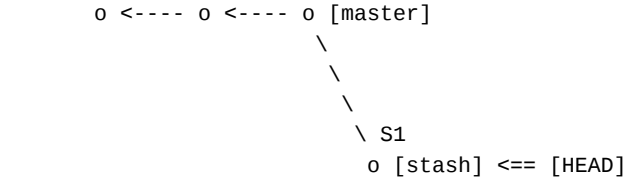
```
$ jit branch stash
$ jit checkout stash
```



Then, we can save the current state of the index by running `commit`, creating the first stash commit.

Figure 24.47. Committing the index state

```
$ jit commit --message "index on master"
```



Next we need to put any unstaged changes into the index and make a second commit. We can't just use `add .` as that would include untracked files. Instead we need to select the files that have been modified and add each one. The following command does just that:

```
$ jit status --porcelain |
    grep "^.M" |
    cut -c 4- |
    xargs jit add
```

As we saw in Chapter 9, *Status report*, `status --porcelain` prints an `M` in the second column for files modified in the workspace. `grep "^.M"` selects lines whose second character matches[7], and `cut -c 4-` selects the fourth to the last characters of each line[8], essentially parsing the filename out of the line. Altogether, this command passes all the files that are modified in the workspace to the `add` command. A similar command, with `D` in place of `M` and `rm` in place of `add` will remove all workspace-deleted files from the index. This state can then be committed to preserve the state of the workspace.

Figure 24.48. Committing the workspace state

```
$ jit status --porcelain | grep "^.M" | cut -c 4- | xargs jit add
$ jit status --porcelain | grep "^.D" | cut -c 4- | xargs jit rm
$ jit commit --message "WIP on master"

    A       B       C
    o <---- o <---- o [master]
                     \
                      \
                       \
                        \ S1    S2
                         o <---- o [stash] <== [HEAD]
```

Finally we want to return to the state `HEAD` was originally in, with the index and workspace in sync, and we can do that by checking out our original branch.

Figure 24.49. Checking out the original branch

```
$ jit checkout master

    A       B       C
    o <---- o <---- o [master] <== [HEAD]
                     \
                      \
                       \
                        \ S1    S2
                         o <---- o [stash]
```

Let's say we've added a couple of commits to `master` and now want to recall our stashed changes. The state of the history is now:

Figure 24.50. Adding more commits to `master`

```
    A       B       C       D       E
    o <---- o <---- o <---- o <---- o [master] <== [HEAD]
                     \
                      \
                       \
                        \ S1    S2
                         o <---- o [stash]
```

[7] https://manpages.ubuntu.com/manpages/bionic/en/man1/grep.1.html
[8] https://manpages.ubuntu.com/manpages/bionic/en/man1/cut.1posix.html

If we cherry-pick the `stash` branch onto the current tip of `master`, that will create two new commits replicating the changes in S_1 and S_2.

Figure 24.51. Cherry-picking the stash commits

```
$ jit cherry-pick ..stash

    A       B       C       D       E      S1'     S2'
    o <---- o <---- o <---- o <---- o <---- o <---- o [master] <== [HEAD]
                     \
                      \
                       \
                        \ S1    S2
                         o <---- o [stash]
```

The `HEAD`, index and workspace will now all match the state of S_2'. To get back to the state we would be in if we ran `git stash apply` or `git stash pop`, we want `HEAD` to point at E, the index should match S_1' and the workspace should match S_2'.

With `HEAD` pointing at S_2', if we run `reset @^` then the `HEAD` and index will change to match S_1', but the workspace will be unaffected, so it still reflects the state of S_2'. If we then run `reset --soft @^`, `HEAD` will be moved to E but nothing else will change; the index will still reflect S_1'. So now the `HEAD` is in the right place, the index reflects the staged changes originally captured in S_1, and the workspace has the unstaged changes from S_2.

Figure 24.52. Regenerating the uncommitted changes

```
$ jit reset @^
$ jit reset --soft @^

    A       B       C       D       E      S1'     S2'
    o <---- o <---- o <---- o <---- o <---- o <---- o
                     \              |
                      \         [master] <== [HEAD]
                       \
                        \ S1    S2
                         o <---- o [stash]
```

At this point, the `stash` branch can be deleted, and the original and cherry-picked stash commits become unreachable.