



BUILDING GIT

JAMES COGLAN

Contents

Change history	xviii
1. (1.0.0) February 25th 2019	xviii
2. (1.0.1) May 22nd 2019	xviii
3. (1.0.2) October 31st 2019	xviii
4. (1.0.3) April 10th 2020	xix
5. (1.0.4) June 18th 2021	xix
License and acknowledgements	xx
1. Introduction	1
1.1. Prerequisites	2
1.2. How the book is structured	2
1.3. Typographic conventions	3
1.4. The Jit codebase	4
2. Getting to know .git	5
2.1. The .git directory	5
2.1.1. .git/config	6
2.1.2. .git/description	7
2.1.3. .git/HEAD	7
2.1.4. .git/info	7
2.1.5. .git/hooks	8
2.1.6. .git/objects	8
2.1.7. .git/refs	9
2.2. A simple commit	9
2.2.1. .git/COMMIT_EDITMSG	11
2.2.2. .git/index	11
2.2.3. .git/logs	12
2.2.4. .git/refs/heads/master	12
2.3. Storing objects	12
2.3.1. The cat-file command	13
2.3.2. Blobs on disk	14
2.3.3. Trees on disk	16
2.3.4. Commits on disk	18
2.3.5. Computing object IDs	19
2.3.6. Problems with SHA-1	21
2.4. The bare essentials	24
I. Storing changes	25
3. The first commit	26
3.1. Initialising a repository	26
3.1.1. A basic init implementation	27
3.1.2. Handling errors	28
3.1.3. Running Jit for the first time	29
3.2. The commit command	30
3.2.1. Storing blobs	31
3.2.2. Storing trees	36
3.2.3. Storing commits	39
4. Making history	45
4.1. The parent field	45

4.1.1. A link to the past	46
4.1.2. Differences between trees	46
4.2. Implementing the parent chain	47
4.2.1. Safely updating .git/HEAD	49
4.2.2. Concurrency and the filesystem	52
4.3. Don't overwrite objects	53
5. Growing trees	55
5.1. Executable files	55
5.1.1. File modes	55
5.1.2. Storing executables in trees	58
5.2. Nested trees	60
5.2.1. Recursive trees in Git	61
5.2.2. Building a Merkle tree	63
5.2.3. Flat or nested?	70
5.3. Reorganising the project	71
6. The index	73
6.1. The add command	73
6.2. Inspecting .git/index	74
6.3. Basic add implementation	77
6.4. Storing multiple entries	82
6.5. Adding files from directories	84
7. Incremental change	87
7.1. Modifying the index	87
7.1.1. Parsing .git/index	87
7.1.2. Storing updates	92
7.2. Committing from the index	93
7.3. Stop making sense	96
7.3.1. Starting a test suite	96
7.3.2. Replacing a file with a directory	98
7.3.3. Replacing a directory with a file	100
7.4. Handling bad inputs	104
7.4.1. Non-existent files	105
7.4.2. Unreadable files	107
7.4.3. Locked index file	108
8. First-class commands	111
8.1. Abstracting the repository	112
8.2. Commands as classes	114
8.2.1. Injecting dependencies	116
8.3. Testing the commands	122
8.4. Refactoring the commands	127
8.4.1. Extracting common code	127
8.4.2. Reorganising the add command	127
9. Status report	131
9.1. Untracked files	131
9.1.1. Untracked files not in the index	133
9.1.2. Untracked directories	135
9.1.3. Empty untracked directories	139
9.2. Index/workspace differences	141

9.2.1. Changed contents	142
9.2.2. Changed mode	144
9.2.3. Size-preserving changes	146
9.2.4. Timestamp optimisation	149
9.2.5. Deleted files	151
10. The next commit	156
10.1. Reading from the database	156
10.1.1. Parsing blobs	158
10.1.2. Parsing commits	158
10.1.3. Parsing trees	159
10.1.4. Listing the files in a commit	162
10.2. HEAD/index differences	162
10.2.1. Added files	163
10.2.2. Modified files	166
10.2.3. Deleted files	167
10.3. The long format	169
10.3.1. Making the change easy	171
10.3.2. Making the easy change	173
10.3.3. Orderly change	176
10.4. Printing in colour	177
11. The Myers diff algorithm	182
11.1. What's in a diff?	182
11.2. Time for some graph theory	184
11.2.1. Walking the graph	186
11.2.2. A change of perspective	190
11.2.3. Implementing the shortest-edit search	193
11.3. Retracing our steps	195
11.3.1. Recording the search	198
11.3.2. And you may ask yourself, how did I get here?	199
12. Spot the difference	202
12.1. Reusing status	202
12.2. Just the headlines	205
12.2.1. Unstaged changes	206
12.2.2. A common pattern	210
12.2.3. Staged changes	212
12.3. Displaying edits	215
12.3.1. Splitting edits into hunks	218
12.3.2. Displaying diffs in colour	226
12.3.3. Invoking the pager	228
II. Branching and merging	233
13. Branching out	234
13.1. Examining the branch command	236
13.2. Creating a branch	239
13.3. Setting the start point	242
13.3.1. Parsing revisions	243
13.3.2. Interpreting the AST	245
13.3.3. Revisions and object IDs	248
14. Migrating between trees	255

14.1. Telling trees apart	256
14.2. Planning the changes	261
14.3. Updating the workspace	264
14.4. Updating the index	266
14.5. Preventing conflicts	267
14.5.1. Single-file status checks	268
14.5.2. Checking the migration for conflicts	269
14.5.3. Reporting conflicts	272
14.6. The perils of self-hosting	274
15. Switching branches	276
15.1. Symbolic references	278
15.1.1. Tracking branch pointers	279
15.1.2. Detached HEAD	280
15.1.3. Retaining detached histories	282
15.2. Linking HEAD on checkout	283
15.2.1. Reading symbolic references	284
15.3. Printing checkout results	285
15.4. Updating HEAD on commit	289
15.4.1. The master branch	291
15.5. Branch management	292
15.5.1. Parsing command-line options	293
15.5.2. Listing branches	295
15.5.3. Deleting branches	299
16. Reviewing history	302
16.1. Linear history	302
16.1.1. Medium format	304
16.1.2. Abbreviated commit IDs	305
16.1.3. One-line format	306
16.1.4. Branch decoration	307
16.1.5. Displaying patches	310
16.2. Branching histories	314
16.2.1. Revision lists	315
16.2.2. Logging multiple branches	316
16.2.3. Excluding branches	322
16.2.4. Filtering by changed paths	329
17. Basic merging	335
17.1. What is a merge?	335
17.1.1. Merging single commits	335
17.1.2. Merging a chain of commits	337
17.1.3. Interpreting merges	339
17.2. Finding the best common ancestor	342
17.3. Commits with multiple parents	344
17.4. Performing a merge	346
17.5. Best common ancestors with merges	348
17.6. Logs in a merging history	352
17.6.1. Following all commit parents	352
17.6.2. Hiding patches for merge commits	353
17.6.3. Pruning treesame commits	353

17.6.4. Following only treesame parents	354
17.7. Revisions with multiple parents	354
18. When merges fail	356
18.1. A little refactoring	356
18.2. Null and fast-forward merges	358
18.2.1. Merging an existing ancestor	358
18.2.2. Fast-forward merge	359
18.3. Conflicted index entries	361
18.3.1. Inspecting the conflicted repository	362
18.3.2. Stages in the index	363
18.3.3. Storing entries by stage	365
18.3.4. Storing conflicts	366
18.4. Conflict detection	368
18.4.1. Concurrency, causality and locks	369
18.4.2. Add/edit/delete conflicts	371
18.4.3. File/directory conflicts	375
19. Conflict resolution	378
19.1. Printing conflict warnings	378
19.2. Conflicted status	381
19.2.1. Long status format	382
19.2.2. Porcelain status format	383
19.3. Conflicted diffs	384
19.3.1. Unmerged paths	384
19.3.2. Selecting stages	385
19.4. Resuming a merge	387
19.4.1. Resolving conflicts in the index	387
19.4.2. Retaining state across commands	387
19.4.3. Writing a merge commit	389
20. Merging inside files	392
20.1. The diff3 algorithm	394
20.1.1. Worked example	394
20.1.2. Implementing diff3	396
20.1.3. Using diff3 during a merge	402
20.2. Logging merge commits	403
20.2.1. Unifying hunks	409
20.2.2. Diffs during merge conflicts	412
20.2.3. Diffs for merge commits	413
21. Correcting mistakes	415
21.1. Removing files from the index	415
21.1.1. Preventing data loss	416
21.1.2. Refinements to the <code>rm</code> command	419
21.2. Resetting the index state	422
21.2.1. Resetting to a different commit	425
21.3. Discarding commits from your branch	426
21.3.1. Hard reset	428
21.3.2. I'm losing my <code>HEAD</code>	432
21.4. Escaping from merges	433
22. Editing messages	436

22.1. Setting the commit message	437
22.2. Composing the commit message	438
22.2.1. Launching the editor	440
22.2.2. Starting and resuming merges	442
22.3. Reusing messages	445
22.3.1. Amending the HEAD	448
22.3.2. Recording the committer	450
23. Cherry-picking	454
23.1. Cherry-picking a single commit	457
23.1.1. New types of pending commit	459
23.1.2. Resuming from conflicts	461
23.2. Multiple commits and ranges	464
23.2.1. Rev-list without walking	464
23.2.2. Conflicts during ranges	466
23.2.3. When all else fails	471
24. Reshaping history	475
24.1. Changing old commits	475
24.1.1. Amending an old commit	475
24.1.2. Reordering commits	476
24.2. Rebase	478
24.2.1. Rebase onto a different branch	480
24.2.2. Interactive rebase	481
24.3. Reverting existing commits	485
24.3.1. Cherry-pick in reverse	487
24.3.2. Sequencing infrastructure	488
24.3.3. The revert command	490
24.3.4. Pending commit status	494
24.3.5. Reverting merge commits	495
24.4. Stashing changes	497
III. Distribution	501
25. Configuration	502
25.1. The Git config format	502
25.1.1. Whitespace and comments	503
25.1.2. Abstract and concrete representation	505
25.2. Modelling the .git/config file	507
25.2.1. Parsing the configuration	508
25.2.2. Manipulating the settings	511
25.2.3. The configuration stack	515
25.3. Applications	517
25.3.1. Launching the editor	517
25.3.2. Setting user details	518
25.3.3. Changing diff formatting	518
25.3.4. Cherry-picking merge commits	520
26. Remote repositories	524
26.1. Storing remote references	525
26.2. The remote command	526
26.2.1. Adding a remote	527
26.2.2. Removing a remote	529

26.2.3. Listing remotes	529
26.3. Refspecs	531
26.4. Finding objects	534
27. The network protocol	540
27.1. Programs as ad-hoc servers	540
27.2. Remote agents	542
27.3. The packet-line protocol	543
27.4. The pack format	547
27.4.1. Writing packs	548
27.4.2. Reading from packs	552
27.4.3. Reading from a stream	556
28. Fetching content	561
28.1. Pack negotiation	561
28.1.1. Non-fast-forward updates	563
28.2. The <code>fetch</code> and <code>upload-pack</code> commands	565
28.2.1. Connecting to the remote	567
28.2.2. Transferring references	568
28.2.3. Negotiating the pack	569
28.2.4. Sending object packs	572
28.2.5. Updating remote refs	573
28.2.6. Connecting to remote repositories	577
28.3. Clone and pull	578
28.3.1. Pulling and rebasing	581
28.3.2. Historic disagreement	583
29. Pushing changes	585
29.1. Shorthand refspecs	585
29.2. The <code>push</code> and <code>receive-pack</code> commands	587
29.2.1. Sending update requests	589
29.2.2. Updating remote refs	594
29.2.3. Validating update requests	599
29.3. Progress meters	601
30. Delta compression	608
30.1. The XDelta algorithm	608
30.1.1. Comparison with diffs	611
30.1.2. Implementation	612
30.2. Delta encoding	616
30.3. Expanding deltas	620
31. Compressing packs	623
31.1. Finding similar objects	623
31.1.1. Generating object paths	626
31.1.2. Sorting packed objects	627
31.2. Forming delta pairs	630
31.2.1. Sliding-window compression	632
31.2.2. Limiting delta chain length	635
31.3. Writing and reading deltas	638
32. Packs in the database	644
32.1. Indexing packs	645
32.1.1. Extracting TempFile	646

32.1.2. Processing the incoming pack	648
32.1.3. Generating the index	649
32.1.4. Reconstructing objects	652
32.1.5. Storing the index	653
32.2. A new database backend	657
32.2.1. Reading the pack index	659
32.2.2. Replacing the backend	662
32.3. Offset deltas	668
33. Working with remote branches	669
33.1. Remote-tracking branches	669
33.1.1. Logging remote branches	669
33.1.2. Listing remote branches	671
33.2. Upstream branches	672
33.2.1. Setting an upstream branch	673
33.2.2. Safely deleting branches	678
33.2.3. Upstream branch divergence	679
33.2.4. The <code>@{upstream}</code> revision	682
33.2.5. Fetching and pushing upstream	683
34. ...and everything else	685
IV. Appendices	687
A. Programming in Ruby	688
A.1. Installation	688
A.2. Core language	689
A.2.1. Control flow	689
A.2.2. Error handling	692
A.2.3. Objects, classes, and methods	693
A.2.4. Blocks	699
A.2.5. Constants	700
A.3. Built-in data types	701
A.3.1. <code>true</code> , <code>false</code> and <code>nil</code>	701
A.3.2. <code>Integer</code>	701
A.3.3. <code>String</code>	702
A.3.4. <code>Regexp</code>	703
A.3.5. <code>Symbol</code>	703
A.3.6. <code>Array</code>	703
A.3.7. <code>Range</code>	705
A.3.8. <code>Hash</code>	706
A.3.9. <code>Struct</code>	707
A.4. Mixins	707
A.4.1. <code>Enumerable</code>	708
A.4.2. <code>Comparable</code>	710
A.5. Libraries	711
A.5.1. <code>Digest</code>	711
A.5.2. <code>FileUtils</code>	711
A.5.3. <code>Forwardable</code>	712
A.5.4. <code>Open3</code>	712
A.5.5. <code>OptionParser</code>	712
A.5.6. <code>Pathname</code>	712

A.5.7. Set	713
A.5.8. Shellwords	713
A.5.9. StringIO	713
A.5.10. StringScanner	714
A.5.11. Time	714
A.5.12. URI	714
A.5.13. zlib	714
B. Bitwise arithmetic	715

List of Figures

2.1. Contents of .git after running <code>git init</code>	6
2.2. Contents of .git after a single commit	10
2.3. Files generated by the first commit	11
2.4. Creating a shell alias	15
2.5. Decimal, hexadecimal and binary representations of numbers	18
2.6. Minimum viable Git repository	24
5.1. Interpreting the bits of a file mode	56
5.2. Octal digit file permissions	57
5.3. Jit project file layout	60
5.4. Project layout with <code>jit</code> in a directory	61
5.5. Tree containing a flat list of entries	65
5.6. Tree containing a nested tree	65
5.7. Project organised into directories	72
11.1. Three equivalent diffs	183
11.2. Interleaving deletions and insertions	184
11.3. Aligning diff changes with logical code blocks	184
11.4. Edit graph for converting ABCABBA into CBABAC	185
11.5. Initial graph exploration state	186
11.6. Exploration state after a single move	186
11.7. Partial exploration state for two moves	186
11.8. Complete exploration state after two moves	187
11.9. First explorations for the third move	187
11.10. Second explorations for the third move	187
11.11. Discarding the worse result	188
11.12. Complete exploration state after three moves	188
11.13. First explorations for the fourth move	188
11.14. Second explorations for the fourth move	189
11.15. Complete exploration state after four moves	189
11.16. Partial exploration state for five moves	189
11.17. Reaching the bottom-right of the graph	190
11.18. Exploration state in $d\text{-}k$ space	190
11.19. Picking a move from the higher x value	191
11.20. Picking a rightward move when both predecessors have equal x	191
11.21. Exploration state with redundant data removed	192
11.22. State array after each move	192
11.23. Completed exploration state for $\text{diff(ABCABBA, CBABAC)}$	195
11.24. Backtracking choices for $(d, k) = (5,1)$	196
11.25. Choosing the previous highest x value	196
11.26. Backtracking from $(d, k) = (4,2)$	197
11.27. Backtracking from $(d, k) = (3,1)$	197
11.28. Backtracking to the starting position (0,0)	198
12.1. List of edits in a diff	221
12.2. Scanning to find the first change	222
12.3. Setting up to begin a hunk	222
12.4. Capturing the first context line	223
12.5. Expanding the hunk on reading a change	224

12.6. Detecting the end of a group of changes	224
13.1. Commits forming a linked list	234
13.2. Moving <code>HEAD</code> to an older commit	234
13.3. Commits diverging from the original history	234
13.4. <code>master</code> branch referring to the original history	234
13.5. Named branches with diverging histories	235
13.6. Two copies of the same history	235
13.7. Chains of commits built on a common history	235
13.8. Branches from a shared history	236
13.9. Alice and Bob merging each other's changes	236
13.10. <code>.git/logs</code> and <code>.git/refs</code> for a single <code>master</code> branch	237
13.11. <code>.git/logs</code> and <code>.git/refs</code> after creating the topic branch	237
14.1. Chain of commits	255
14.2. New branch pointing to the last commit	255
14.3. New branch pointing to an older commit	255
14.4. Moving <code>HEAD</code> to a different branch	255
14.5. Trees of two adjacent commits	257
14.6. Top level of trees from two commits	257
14.7. Trees for the <code>lib</code> sub-tree of two commits	257
14.8. Trees for the <code>lib/models</code> sub-tree of two commits	258
15.1. Two branch pointers into a commit history	276
15.2. Moving <code>HEAD</code> to point at an older commit	276
15.3. Committing moves the <code>HEAD</code> pointer	276
15.4. Committing a chain of commits	277
15.5. Moving <code>HEAD</code> to another branch	277
15.6. Branch pointer following a commit chain	277
15.7. Two branch pointers into a commit history	278
15.8. <code>HEAD</code> referring to a branch pointer	279
15.9. Branch tracking newly created commits	280
15.10. Switching <code>HEAD</code> to point at a different branch	280
15.11. Detached <code>HEAD</code> , pointing directly to a commit	281
15.12. Branch formed while in detached <code>HEAD</code> mode	282
15.13. Reattaching <code>HEAD</code> to a new branch pointer	283
15.14. <code>HEAD</code> attached to a branch	289
15.15. Detaching <code>HEAD</code> by making a commit	289
15.16. Branch pointer following a new commit	290
16.1. Single chain of commits	303
16.2. History with two branches	316
16.3. Starting commits for the history search	317
16.4. Reading the start commits to find their parents	317
16.5. Continuing along the topic branch	317
16.6. Continuing along the <code>master</code> branch	318
16.7. Locating the common ancestor	318
16.8. Starting commits for a new log search	318
16.9. Processing the most recent commit	319
16.10. Processing the second youngest commit	319
16.11. Processing the third youngest commit	319
16.12. History with two branches	322

16.13. History reachable from <code>master</code>	322
16.14. History reachable from <code>topic</code>	323
16.15. History reachable from <code>master</code> and not from <code>topic</code>	323
16.16. History containing merge commits	323
16.17. Initial state for a range search	324
16.18. Processing an excluded commit	324
16.19. Processing an included commit	324
16.20. Processing an uninteresting commit	324
16.21. Stopping the search on an all-uninteresting queue	325
16.22. Commit graph with a long branch	327
16.23. Loaded commits part-way through a graph search	328
17.1. Two branches changing different files	336
17.2. History following a successful merge	337
17.3. Two branches changing multiple files	338
17.4. History after merging a chain of commits	339
17.5. Alice and Bob's trees before the merge	340
17.6. Calculating Alice and Bob's changes since they diverged	340
17.7. Symmetric merge result	341
17.8. History leading to a merge commit	341
17.9. Merged history with further commits	342
17.10. Merged history without merged parent links	342
17.11. History following multiple merges	342
17.12. Branch containing fork/merge bubbles	347
17.13. Branching/merging history	348
17.14. History with many candidate common ancestors	349
18.1. Merging branches changing different files	356
18.2. Merging branches changing the same file	356
18.3. History with <code>topic</code> already merged into <code>master</code>	358
18.4. History where <code>HEAD</code> is an ancestor of another branch	359
18.5. Fast-forwarding the <code>HEAD</code> branch	360
18.6. Flattened fast-forward result state	360
18.7. Forced merge commit created using <code>--no-ff</code>	360
18.8. Merging branches changing the same file	362
18.9. Bitwise breakdown of the index entry flag bytes	365
18.10. Merging branches changing different files	369
18.11. Sequential application of changes from parallel branches	369
18.12. Reversed sequential application of changes from parallel branches	369
18.13. Merging branches changing the same file	370
18.14. Sequential application of changes to the same file	370
18.15. Reversed sequential application of changes to the same file	370
20.1. Merging branches changing the same file	392
20.2. Initial file contents	393
20.3. Inserting a new line and deleting another	393
20.4. Deleting a line and then inserting one	393
20.5. Original ingredient list	394
20.6. Alice and Bob's changes to the ingredient list	394
20.7. Alice's diff against the original	395
20.8. Bob's diff against the original	395

20.9. Aligning the three versions on unchanged lines	395
21.1. Chain of commits with <code>HEAD</code> pointing to the branch tip	426
21.2. Checking out an earlier commit, leaving the branch pointer in place	426
21.3. Deleting the old branch pointer and restarting it at the current <code>HEAD</code>	427
21.4. Checking out the moved branch	427
21.5. Chain of unreachable commits	432
21.6. <code>ORIG_HEAD</code> pointing to the previous <code>HEAD</code> position	432
22.1. Commit chain before a reset	447
22.2. Commit chain after a reset	447
22.3. Commit graph with an ‘amended’ commit	447
22.4. Chain of commits to be squashed	448
22.5. Desired squashed history	448
22.6. <code>HEAD</code> reset to before the commits to be squashed	448
22.7. Squashed history	448
22.8. The <code>commit</code> command creates a new commit	448
22.9. The <code>commit --amend</code> command replaces an existing commit	449
23.1. History with two branches	454
23.2. Merging part of the topic branch into <code>master</code>	454
23.3. Cherry-picking a single commit into <code>master</code>	454
23.4. Imagined history graph for a cherry-pick merge	455
23.5. Merge input commits with history links removed	455
23.6. Merge commit pointing to the two input commits	456
23.7. Merge commit following a previous merge	456
23.8. Cherry-pick commit unconnected to its source	456
23.9. History with conflicting changes	466
23.10. Partially completed range cherry-pick	466
23.11. Partially completed range cherry-pick with conflicts resolved	467
24.1. Sequence of five commits	475
24.2. Resetting <code>HEAD</code> to the target commit	475
24.3. Amending the target commit	476
24.4. Cherry-picking the remaining history	476
24.5. Sequence of six commits	476
24.6. Creating a fixup branch	477
24.7. Cherry-picking the reordered commits	477
24.8. Cherry-picking the remaining history	478
24.9. Resetting the original branch	478
24.10. History with two divergent branches	478
24.11. Branch after rebasing to the tip of <code>master</code>	479
24.12. Resetting the current branch to the upstream branch	479
24.13. Cherry-picking the branch onto the upstream	480
24.14. History with three chained branches	480
24.15. Rebase onto a different branch	480
24.16. Resetting to the target branch	481
24.17. Cherry-picking the original branch	481
24.18. History before squashing	482
24.19. Checking out the desired tree	482
24.20. Resetting to the parent commit	482
24.21. Amending the parent commit to contain the squashed changes	483

24.22. Cherry-picking the remaining history	483
24.23. Cleaning up branch pointers after squashing	483
24.24. History before a fix-up	484
24.25. Creating a fix-up commit	484
24.26. Starting a fix-up branch	484
24.27. Cherry-picking the fix-up commit	484
24.28. Creating a squashed commit containing the fix-up	485
24.29. History following a relocated fix-up commit	485
24.30. History with concurrent edits	486
24.31. Alice removes a shared commit from the history	486
24.32. Merging reintroduces a dropped commit	486
24.33. Committing to undo earlier changes	487
24.34. Merging does not reintroduce the removed content	487
24.35. History with two files	487
24.36. Reverting an old change	488
24.37. History with non-commutative commits	491
24.38. Reverting the last two commits	491
24.39. Branched and merged history	496
24.40. History with reverted merge	496
24.41. Attempting to re-merge a reverted branch	497
24.42. Cherry-picking reverted changes	497
24.43. Reverting a reverted merge	497
24.44. Stored pair of stash commits	498
24.45. Initial work state	498
24.46. Checking out the stash branch	498
24.47. Committing the index state	498
24.48. Committing the workspace state	499
24.49. Checking out the original branch	499
24.50. Adding more commits to master	499
24.51. Cherry-picking the stash commits	500
24.52. Regenerating the uncommitted changes	500
26.1. Alice's history	532
26.2. Bob's repository after fetching from Alice	532
26.3. Alice's extended history	532
26.4. Bob's updated repository	532
26.5. Alice's amended history	532
26.6. Bob's repository after an unforced fetch	532
26.7. Bob's repository after a forced fetch	533
26.8. Forking history	535
28.1. Alice's initial repository	561
28.2. Bob's copy of Alice's repository	561
28.3. Alice extends the master branch	561
28.4. Bob's updated database	562
28.5. Bob's updates his remote reference	562
28.6. Alice's rewritten history	563
28.7. Alice's database with unreachable commits removed	564
28.8. Bob's repository after fetching from Alice	579
28.9. Bob fetches new commits from Alice	579

28.10. Bob performs a fast-forward merge	579
28.11. Bob fetches divergent history	580
28.12. Bob merges Alice's changes into his <code>master</code> branch	580
28.13. Alice's history	580
28.14. Alice fetches from Bob	580
28.15. Alice fast-forwards to Bob's merge commit	581
28.16. Alice merges Bob's changes with her own	581
28.17. Bob's repository after fetching from Alice	582
28.18. Bob resets his <code>master</code> branch to Alice's latest commit	582
28.19. Bob rebases his branch	582
28.20. Alice's repository after fetching Bob's rebased commits	582
28.21. Alice fast-forwards her <code>master</code> branch	583
28.22. Bob's repository after fetching from Alice	583
28.23. Bob fetches Alice's rewritten history	583
28.24. Bob merges Alice's old and new histories	583
28.25. Bob's repository after fetching Alice's revert commits	584
29.1. Repository in sync with the remote	591
29.2. Local history diverges from the remote	592
29.3. Merging the remote's changes	592
29.4. Alice and Bob's combined history	595
29.5. Alice's update is accepted	595
29.6. Bob's repository after merging Alice's changes	595
30.1. First few iterations of the matching loop	610
30.2. Finding the first matching block	610
31.1. Empty sliding window	630
31.2. Sliding window with one element	630
31.3. Sliding window with two elements	631
31.4. Sliding window with three elements	631
31.5. Sliding window with one slot empty	631
31.6. Sliding window after becoming full	631
31.7. Sliding window overwriting old elements	631
31.8. Delta dependencies from compressing three commits	636
31.9. Delta dependencies with compression size limits	638
32.1. Files stored end-to-end	644
32.2. Files stored in fixed-size blocks	644
B.1. Bitwise-and and bitwise-or of two numbers	715
B.2. Setting a <code>File.open</code> mode with bitwise-or	715
B.3. Checking a <code>File.open</code> mode with bitwise-and	716
B.4. Checking a non-matching <code>File.open</code> mode	716
B.5. Packing values into a byte	716
B.6. Decimal, hexadecimal and binary representations of numbers	717

Change history

1. (1.0.0) February 25th 2019

- The first edition.

2. (1.0.1) May 22nd 2019

- Explains the use of subscripts to indicate bases for numbers in the introduction.
- Mentions that `RUBYOPT="--disable gems"` must not be used for running tests.
- Clarifies that index entries must have at least one trailing null byte.
- Makes some minor stylistic changes and corrects various typographical errors.

3. (1.0.2) October 31st 2019

- Removes the `Database::Tree#each_entry` method and uses the exposed `entries` attribute for iterating the tree's contents.
- Adjusts the `Refs` class's method for creating missing parent directories for new branches, and cleans up empty parent directories when branches are deleted.
- Simplifies the `RevList#mark_parents_uninteresting` method's algorithm for visiting all the loaded commits.
- Redesigns the way that `Database::TreeDiff` filters the diff by path, by introducing a new class called `PathFilter`, using a data structure that makes the implementation simpler.
- Fixes `Command::Status#print_pending_type` so that it works during a cherry-pick or revert that paused due to a merge conflict.
- Fixes the `Config#subsections` method so that it does not return the empty string `""` as a subsection name.
- Refactors the `Pack::Numbers::PackedInt56LE` module's implementation to make it easier to read, and more like the other `Pack::Numbers` modules.
- Corrects the examples for how object IDs are looked up in a pack index fanout table.
- Makes the `branch -vv` command always print the upstream branch name, even if the local branch is up-to-date with the remote.
- Uses an explicit caller for all class methods that were implicit called on `self`, for example `Revision.parse` instead of just `parse` in the `Revision` class, to clarify what's being called.
- Makes a few other minor stylistic refactorings and typographical corrections.

4. (1.0.3) April 10th 2020

- Changes `Tree::ENTRY_FORMAT` in the first commit to `Z*H40` and removes the confusing statement that the mode of tree entries is seven bytes, which is not true once nested trees are introduced.
- Reduces the amount of changes required to `Tree.build` and `Tree#add_entry` by implementing `parent_directories` and `basename` on the original `Entry` class, so it behaves like the later `Index::Entry` class.
- Fixes a bug in `Index` caused by deleting elements from the sets in `@parents` while iterating over them.
- Improves the `Config` class so that it can handle multi-line values, where the line break is escaped with a backslash.
- Corrects an example of the `push` command by setting `receive.denyCurrentBranch` to `false` in the remote repository.
- Fixes a couple of grammatical errors.

5. (1.0.4) June 18th 2021

- Includes a version of `Index#initialize` that was previously omitted from the text.
- Adds the ability for the `diff` command to compare any two revisions, by invoking the command with two arguments. Some logic is refactored out of `Command::Log` into `Command::PrintDiff` to enable this.
- Fixes `Refs#short_name` so that it handles refnames that don't have one of the well-known directories as a prefix.
- Checks for a leading slash in branch names in `Revision#valid_ref?` and checks for valid refs in the `receive-pack` command, to prevent clients using path traversal to write outside `.git/refs`.
- Tweaks how `Pack::Compressor` sorts objects to make the code simpler and the resulting packs slightly smaller.
- Includes a before/after comparison of the effect of enabling delta compression when sending a pack via the `push` command.
- Reloads the list of `Packed` stores held by `Database::Backends` at the end of `ReceiveObjects#recv_packed_objects`, so that objects in the received pack can be loaded.

License and acknowledgements

Building Git

Copyright © 2019–2021 James Coglan. All rights reserved.

The program Jit, and all source code contained in this book, is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

Git™ and the Git logo are either registered trademarks or trademarks of Software Freedom Conservancy, Inc., corporate home of the Git Project, in the United States and/or other countries.

Neither this book nor its author is affiliated with or endorsed by any person or organisation associated with the Git Project or with Software Freedom Conservancy, Inc.

This book is produced using free and open source software. The text is written using the AsciiDoc¹ language, formatted using DocBook XSL² and Pygments³, and compiled using xsltproc⁴, Calibre⁵ and Apache FOP⁶.

I owe a huge debt of gratitude to my partner, Rachel Souhami, who helped me discuss a lot of the ideas in this book, provided feedback on the text, and was endlessly encouraging to me throughout the writing process. Aanand Prasad, Murray Steele and Paul Mucur reviewed the Ruby tutorial and provided me with valuable feedback. I would also like to thank Brian Kung, Chris Jones, Gabe Jackson, Ivan Taney, Jan Lehnardt, Julien Kirch, Kushal Kumaran, Misty De Meo and Pierre Jambet for submitting corrections and suggesting improvements.

¹<https://www.methods.co.nz/asciidoc/>

²<http://docbook.sourceforge.net/>

³<http://pygments.org/>

⁴<http://xmlsoft.org/XSLT/xsltproc2.html>

⁵<https://calibre-ebook.com/>

⁶<https://xmlgraphics.apache.org/fop/>

1. Introduction

Over the last decade, Git has risen to become the dominant version control and collaboration tool both for open source software, and within private companies. It ushered in a move away from centralised version control towards a decentralised model, in which every developer has a complete self-contained repository on their local workstation. This radically changed the process of software development, making many common operations faster and enabling completely new workflows for managing change across software teams.

Git has always had a reputation for being hard to learn and use. This is partly because of this paradigm change, but also because its user interface is confusing. It provides many ways of doing the same thing, and has unclear separation of responsibilities between different commands. The ad-hoc nature of its design process means it's accumulated more than its fair share of rabbit holes for new users to fall down. In some cases the user interface is strictly more complicated than the functionality it sits on top of. It is a long-running complaint that in order to use Git effectively, you have to know what it's doing behind the scenes.

Unfortunately, determining what it's really doing can be a daunting task. We can get a certain amount of visibility by examining what it's doing with the files on disk, but for some things there's no substitute for reading the source code. At the time of writing, the current version of Git is v2.15.0, and its codebase is approaching 200,000 lines of C code. Due to the large number of options some commands have, a lot of this code is very complicated and it can be hard to see the essential functionality among all the different code paths.

In this book, we'll be building our own version of Git, from scratch, to learn how it works. To make this more manageable than reading the original, we'll take two simplifying steps. First, we won't be implementing anywhere near the complete command interface from Git, only those parts that are essential to making a usable replica that demonstrates the core concepts. Second, we'll be using a high-level language with a rich standard library, so we'll need much less code than if we were rewriting in C. To the extent possible, this implementation will be informed by direct observation of Git itself, either by observing its effects on files, by logging its system calls, or by study of the original source code.

To keep ourselves honest, we won't rely on the original Git implementation in our codebase—instead, all the commits we add to our project will be generated by our own software. That means we won't be able to commit anything until we have enough code to generate a valid Git commit, but as it turns out, we won't need to write much code to get the project off the ground. This process of *self-hosting*¹ provides excellent motivation for adding new features, as we'll quickly notice where the lack of a feature is limiting the tool's utility. As we add features, our implementation will become more and more capable, and each new command will unlock opportunities for further improvements.

We'll focus on covering the essential components as quickly as possible, rather than fully implementing each command and all its options. The ultimate aim is to have a program that can store and accurately retrieve its own Git history, and push itself to a remote repository hosted on GitHub².

¹<https://en.wikipedia.org/wiki/Self-hosting>

²<https://github.com/>

Within a tool like Git, numerous programming concepts come into play and studying it is a great way to broaden your knowledge. As we work on our implementation, we'll explore the theory behind each component and why it works the way it does. So, as well as gaining a better understanding of Git, you'll learn a great many programming concepts along the way that you can apply to other problems.

You'll learn Unix system concepts like how to work with files, how processes work and how they communicate, and how to present output in a terminal. You'll learn about the data structures that Git uses to keep the commit history compact and efficient to search, both on disk and when it sends content over the network. You'll learn the algorithms behind diffing and merging files, and how to find data shared between files so they can be compressed. And, you'll learn how Git implements a form of distributed database, how it supports concurrent editing, why merge conflicts occur and how it prevents race conditions when users share their changes to a project.

1.1. Prerequisites

I have tried to target this text at an audience that has some experience of programming already, without assuming they know any particular language or topic in detail. Our Git implementation will use Ruby, a dynamic object-oriented language, but this choice is somewhat arbitrary and the code should be approachable if you have experience in a similar language such as Python, JavaScript, or C#.

If you don't already know Ruby, or need to brush up, you can find everything you need to know about it for this book in Appendix A, *Programming in Ruby*. This includes installation instructions, an explanation of the core language and its built-in libraries.

One of the aims of this book is to teach Unix programming concepts, and as such I recommend following the code on a Unix-like system such as macOS or Linux. Some of the code will not be compatible with Windows — essentially, anything that interacts directly with the filesystem, starts child processes, or performs I/O would need further work to run on that platform. However, if you are a Windows user, this knowledge is still likely to be useful to you, and there is plenty about this project that is platform-agnostic.

1.2. How the book is structured

This book charts the development of our own version of Git. Many of the features we'll develop in the course of the project rely on tools developed in earlier chapters, and so it will make most sense if you approach the book in the order it is presented. Although each chapter functions as a useful reference for an element of Git's behaviour, the implementations will rely on code studied earlier in the text.

The content has been grouped into three parts, covering broad conceptual chunks of the problem of distributed version control. Part I, “Storing changes” focuses on the initial challenge of storing the history of a project as a sequence of commits, and helping the author see the current state of their project. Part II, “Branching and merging” introduces the idea of concurrent editing via branches, and examines how commits on separate branches are merged together. This includes a discussion of how the operation of merging can be applied to a wide variety of

operations to alter the project history. Part III, “Distribution” discusses how Git sends content over the network between repositories, how it stores data efficiently, and how it prevents data loss in shared repositories.

1.3. Typographic conventions

All code listings in this book are in Ruby, unless otherwise indicated. Code listings that form part of the codebase are shown with their filename at the top, for example:

```
# lib/command/diff.rb

module Command
  class Diff < Base

    include PrintDiff

    def define_options
      @options[:patch] = true
      define_print_diff_options

      @parser.on "--cached", "--staged" do
        @options[:cached] = true
      end
    end

    #
    # ...
    #

  end
end
```

The line `# ...` indicates that part of a method or class has been elided, either because it is unchanged from a previous version, or because it will appear in a later code listing.

When a code listing includes lines beginning `>>`, these are examples typed into Ruby’s interactive shell, IRB. Lines beginning with `=>` show the result of the previous statement, for example:

```
>> 1 + 1
=> 2
```

Listings in which lines begin with `$` are commands entered into a Bash terminal. Lines not beginning with `$` in such listings are output from the last command.

```
$ cat .git/HEAD
ref: refs/heads/master
```

When discussing numbers in various bases, numerals are presented with a subscript indicating which base they are in. For example, 123_{10} is a decimal number, one hundred and twenty-three. The same number may be presented as the hexadecimal numeral $7B_{16}$, or the binary numeral 1111011_2 . In Ruby, numeric literals beginning with `0x` are hexadecimal, those starting with `0b` are binary, and those with a leading `0` are octal (base-8). Unless otherwise indicated, you should assume numerals are in base-10.

1.4. The Jit codebase

Over the course of this book we'll build an alternative implementation of Git, which I've named Jit. You should have received a copy of the Jit repository along with this book. This should contain a working copy of the Jit codebase along with its history in the `.git` directory.

```
$ cd Jit

$ ls
LICENSE.txt Rakefile    bin          lib          test

$ tree .git
.git
├── HEAD
├── config
├── index
└── objects
    └── pack
        ├── pack-32295bd84e83a270b366bd809565fe250082f4bb.idx
        └── pack-32295bd84e83a270b366bd809565fe250082f4bb.pack
└── refs
    └── heads
        └── master
```

All the content in the `.git` directory was generated using Jit itself, not the canonical Git software. The files in `.git/objects/pack` contain all the code that generated them!

You can run Jit from this directory via the file `bin/jit`, if you have Ruby installed. For example, running this command should show you the complete history of the project:

```
$ ./bin/jit log --patch
```

If you want to run `jit` from other locations on your computer, add its `bin` directory to your PATH:

```
$ export PATH="$PWD/bin:$PATH"
```

Jit has also been published as a *gem*, which is Ruby's system for sharing packages. You can install it using `gem`:

```
$ gem install jit
```

The text of the book tracks the commit history of Jit, and you can follow along with the changes described in the book by using the `log` command to view all the commits and their diffs. To retrieve an old version of the codebase you can run `./bin/jit checkout`, but do bear in mind that since this will change the code that's in the workspace, you will not be able to get back to the `master` commit if you check out a commit before the `checkout` command was added! But, since the repository is a valid Git repository, you can also use the usual `git` commands to navigate it.

2. Getting to know .git

Our aim in this book is to build a working copy of Git, one that's capable enough to be used in place of the real thing. This applies right from the outset: we'll proceed without using Git at all to store our new codebase, only creating our first commit when our version is capable of writing one itself.

We'd like to get to that point as soon as possible, and so our first task is to get acquainted with the structure of a Git repository, and determine which parts of it are essential to replicate in our first commit.

2.1. The .git directory

The first thing we run to create new repositories is `git init`. Without any additional arguments, this turns the current working directory into a Git repository, placing a directory called `.git` inside it. Before we go any further, we should familiarise ourselves with what this directory contains, and attempt to figure out which pieces of it are essential for getting to our first commit.

To get started, we can run these commands to create a new repository and then change to the directory we just created.

```
$ git init first-repo
Initialized empty Git repository in /Users/jcoglan/src/first-repo/.git/
$ cd first-repo
```

To inspect the contents of `.git`, we can use a program called `tree`¹. This is available through Homebrew² and most Linux package archives. Running `tree` after the above commands prints the following:

¹<https://manpages.ubuntu.com/manpages/bionic/en/man1/tree.1.html>

²<https://brew.sh/>

Figure 2.1. Contents of .git after running git init

```

.git
├── config
├── description
├── HEAD
└── hooks
    ├── applypatch-msg.sample
    ├── commit-msg.sample
    ├── post-update.sample
    ├── pre-applypatch.sample
    ├── pre-commit.sample
    ├── pre-push.sample
    ├── pre-rebase.sample
    ├── pre-receive.sample
    ├── prepare-commit-msg.sample
    └── update.sample
├── info
│   └── exclude
└── objects
    ├── info
    └── pack
└── refs
    ├── heads
    └── tags

```

The function of most of these files is described in the repository-layout documentation³. A repository can actually function without most of these files, and we will introduce them gradually as we progress through our implementation. For now, I'll give a brief description of each file here just so they're not completely mysterious.

2.1.1. .git/config

.git/config contains configuration settings that apply only to this repository, rather than to your use of Git in general. It usually contains only a handful of entries, for example on my system git init generates the following:

```

[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true

```

The meaning of every configuration setting can be found in the config documentation⁴. These settings tell Git that:

- We're using version 0 of the repository file format
- Git should store each file's *mode* (i.e. whether the file is executable) as reported by the filesystem
- The repository is not *bare*, meaning it's a repository where the user will edit the working copy of files and create commits, rather than just a shared location for multiple users to push and pull commits from

³<https://git-scm.com/docs/gitrepository-layout>

⁴<https://git-scm.com/docs/git-config>

- The reflog is enabled, meaning that all changes to files in .git/refs are logged in .git/logs

If you're using macOS, you'll probably also get these settings that help Git handle the text encoding of filenames on that operating system.

```
ignorecase = true  
precomposeunicode = true
```

2.1.2. .git/description

This file contains the name of the repository; it is used by gitweb⁵ to display the repositories hosted on a server. It looks like this by default:

```
Unnamed repository; edit this file 'description' to name the repository.
```

2.1.3. .git/HEAD

HEAD is an important part of Git's data model. It contains a reference to the *current commit*, either using that commit's ID or a *symbolic reference*⁶ (often shortened to *symref*) to the current branch. It will usually contain a symref to the master branch out of the box.

```
ref: refs/heads/master
```

When you create a new commit, the commit referenced by HEAD will be used as the new commit's parent. When you check out a branch or specific commit, HEAD will be changed to refer to the thing you just checked out. When you perform a merge, the requested branch is merged with whatever HEAD points at.

2.1.4. .git/info

This directory is used to store various pieces of metadata about the repository that don't fit into any of the other main directories. For example, the info/refs directory is used to list some of the contents of the repository in a format that's convenient for Git's over-the-wire remote protocols.

On init, Git will usually create .git/info/exclude, which tells Git's user-facing commands like add, rm and status which files should not be checked in. By default, it just contains a few commented-out examples.

```
# git ls-files --others --exclude-from=.git/info/exclude  
# Lines that start with '#' are comments.  
# For a project mostly in C, the following would be a good set of  
# exclude patterns (uncomment them if you want to use them):  
# *.[oa]  
# *~
```

You may be familiar with the .gitignore file for specifying files to ignore. The difference between .git/info/exclude and .gitignore is that the latter is part of your source tree, its

⁵<https://git-scm.com/docs/gitweb>

⁶Section 15.1, "Symbolic references"

contents end up stored in commits and therefore are distributed to people you collaborate with. The exclude file is not part of the commit database and so remains on your machine, so if there are patterns that only apply on your computer and not to anybody else working on the project, put them here.

2.1.5. .git/hooks

The hooks directory contains scripts that Git will execute as part of certain core commands, and so they can be used to customise Git's behaviour. For example, before creating a new commit, Git will execute `.git/hooks/pre-commit` if that file exists. All the scripts that Git creates here on `init` are examples, and you can activate them by removing the extension `.sample` from the filename. For example, here's the contents of `.git/hooks/commit-msg.sample`, which checks the contents of commit messages before the new commit is written.

```
#!/bin/sh
#
# An example hook script to check the commit log message.
# Called by "git commit" with one argument, the name of the file
# that has the commit message. The hook should exit with non-zero
# status after issuing an appropriate message if it wants to stop the
# commit. The hook is allowed to edit the commit message file.
#
# To enable this hook, rename this file to "commit-msg".
#
# Uncomment the below to add a Signed-off-by line to the message.
# Doing this in a hook is a bad idea in general, but the prepare-commit-msg
# hook is more suited to it.
#
# SOB=$(git var GIT_AUTHOR_IDENT | sed -n 's/^\\(.*>)\\.*/$Signed-off-by: \\1/p')
# grep -qs "^$SOB" "$1" || echo "$SOB" >> "$1"
#
# This example catches duplicate Signed-off-by lines.

test "" = "$(grep '^Signed-off-by: ' "$1" |
    sort | uniq -c | sed -e '/^[\t ]*[1-9][\t ]*/d')" || {
    echo >&2 Duplicate Signed-off-by lines.
    exit 1
}
```

2.1.6. .git/objects

This directory forms Git's *database*; it's where Git stores all the content it tracks — your source code and other assets that you put under version control. We'll look more closely at the files stored in here shortly, but for now we'll just make a note of the subdirectories that are created on `init`.

- `.git/objects/pack` is for storing objects in an optimised format⁷. At first, every object added to the database is stored in its own file; these are called *loose objects*. But on certain events, many individual files are rolled up into a *pack*: a single file containing many objects in a compressed format. Those packs, and the index files that describe where to find each object within them, live here.

⁷Chapter 32, *Packs in the database*

- `.git/objects/info` is used to store metadata about the packs for use by some of the remote protocols, and it also stores links to other object stores if the repository refers to content that's stored elsewhere.

2.1.7. `.git/refs`

The `refs` directory stores various kinds of pointers into the `.git/objects` database. These pointers are usually just files that contain the ID of a commit. For example, files in `.git/refs/heads` store the latest commit on each local branch⁸, files in `.git/refs/remotes` store the latest commit on each remote branch⁹, and `.git/refs/tags` stores tags. Stashed changes are also stored as commits in the object database, and references to those commits are placed in `.git/refs/stash`.

2.2. A simple commit

We've examined everything that Git generates when you run `init`. The next step to bootstrapping our system and storing our first commit is to learn a little about how Git stores content, and we'll do that by having it store a single commit and then looking at the resulting files.

Here's a script that will generate a Git repository, write two small text files, and create a single commit containing those files.

```
$ git init git-simple-commit
$ cd git-simple-commit

$ echo "hello" > hello.txt
$ echo "world" > world.txt

$ git add .

$ git commit --message "First commit."
[master (root-commit) 2fb7e6b] First commit.
 2 files changed, 2 insertions(+)
 create mode 100644 hello.txt
 create mode 100644 world.txt
```

Already, Git is telling us some interesting things in its output, in the text that's printed when we run `git commit`. It says we're on the `master` branch, and we've made a *root commit*, that is a commit with no parents. Its abbreviated ID is `2fb7e6b`, and its message is `First commit`. It contains two new files, and two lines of text have been added. Each of those files have mode `100644`, which we'll learn more about shortly when we look at trees.

Now, let's look at the files on disk that exist after we've run the above commands. If you run the above script, your files might look a little different to this because Git stores the timestamp at which a commit is created and the name and email address of the author, and these will be different on your system. However, you should see something that looks mostly like the file tree below; run `tree .git` to generate it.

⁸Chapter 13, *Branching out*

⁹Section 26.1, “Storing remote references”

Figure 2.2. Contents of .git after a single commit

```
.git
├── COMMIT_EDITMSG
├── config
├── description
├── HEAD
├── hooks
│   ├── applypatch-msg.sample
│   ├── commit-msg.sample
│   ├── post-update.sample
│   ├── pre-applypatch.sample
│   ├── pre-commit.sample
│   ├── pre-push.sample
│   ├── pre-rebase.sample
│   ├── pre-receive.sample
│   ├── prepare-commit-msg.sample
│   └── update.sample
├── index
├── info
│   └── exclude
└── logs
    ├── HEAD
    └── refs
        └── heads
            └── master
├── objects
│   ├── 2f
│   │   └── b7e6b97a594fa7f9ccb927849e95c7c70e39f5
│   ├── 88
│   │   └── e38705fdbd3608cddbe904b67c731f3234c45b
│   ├── cc
│   │   └── 628ccd10742baea8241c5924df992b5c019f71
│   ├── ce
│   │   └── 013625030ba8dba906f756967f9e9ca394464a
│   ├── info
│   └── pack
└── refs
    ├── heads
    │   └── master
    └── tags
```

Comparing this to Figure 2.1, “Contents of .git after running `git init`”, we can remove the files that were already present after we ran `git init`, and focus just on those files that are new:

Figure 2.3. Files generated by the first commit

```

.git
├── COMMIT_EDITMSG
├── index
└── logs
    ├── HEAD
    └── refs
        └── heads
            └── master
└── objects
    ├── 2f
    │   └── b7e6b97a594fa7f9ccb927849e95c7c70e39f5
    ├── 88
    │   └── e38705fdbd3608cddbe904b67c731f3234c45b
    ├── cc
    │   └── 628ccd10742baea8241c5924df992b5c019f71
    └── ce
        └── 013625030ba8dba906f756967f9e9ca394464a
└── refs
    └── heads
        └── master

```

Just as we did following `git init`, we can take a look at each of these files and see what's inside them, using the `cat`¹⁰ command.

2.2.1. .git/COMMIT_EDITMSG

```
$ cat .git/COMMIT_EDITMSG
First commit.
```

This seems to contain the message we gave for our first commit. This file is used by Git to compose commit messages; if you don't pass `--message` when you run `git commit`, then Git opens your editor to compose the commit message¹¹. The file `COMMIT_EDITMSG` is what the editor is told to open, and when you close your editor Git will read this file to see what you saved there, and use it in the commit. When you do pass `--message`, it still saves the message in this file.

2.2.2. .git/index

If you run `cat .git/index`, you will probably see a lot of what looks like nonsense in your terminal¹²:

```
$ cat .git/index
DIRCY& Y@%
#VFJ  hello.txtY& Y@t+$Y$#+\q world.txtTREE2 0
|s24IP&j`
```

This is because `index` contains binary data, rather than text. However, you might spot the filenames `hello.txt` and `world.txt` in the output. We'll look at `index` in more depth in Chapter 6, *The index*, but for now you can think of it as a cache that stores information about

¹⁰<https://manpages.ubuntu.com/manpages/bionic/en/man1/cat.1posix.html>

¹¹Chapter 22, *Editing messages*

¹²In some cases, this might make your terminal prompt stop printing properly, and text you type in may not display. If this happens, type `reset` and press `ENTER`, which should reload your shell and get things working again.

each file in the current commit and which version of each file should be present. This is updated when you run `git add`, and the information in the index is used to build the next commit.

2.2.3. .git/logs

The files in here are text files, and we can use cat to print them. The trailing backslashes here indicate a run-on line; all the content here actually appears on one line in the files and I've just broken them up this way to fit them on the page.

These files contain a log of every time a *ref*—that is, a *reference*, something that points to a commit, like `HEAD` or a branch name—changes its value. This happens when you make a commit, or change which branch is checked out, or perform a merge or a rebase. Because our `HEAD` is a symref to `refs/heads/master`, the logs for both of these contain the same content. What has been recorded here is one log event, which changed these refs from pointing at nothing—the `0000000...` value—to the commit `2fb7e6b`. Each log event contains a little information about what caused it, which in this case is an initial commit whose author and message have been recorded.

These logs are used by the `reflog` command.

2.2.4. .git/refs/heads/master

The final things in the list of new files are some cryptically named files in `.git/objects`, which we'll discuss shortly, and one new file in `.git/refs`, called `refs/heads/master`. This file simply records which commit is at the tip of the master branch, which `HEAD` is currently referring to.

```
$ cat .git/refs/heads/master  
2fb7e6b97a594fa7f9ccb927849e95c7c70e39f5
```

2.3. Storing objects

The final files that the first commit introduced are four files in `.git/objects` with long hexadecimal names. One of these names might look familiar: we know from the information printed by `git commit` that `2fb7e6b` is the abbreviated form of the ID of the first commit. We can discover its full ID by running `git show`, which prints information about the `HEAD` commit, and examining the first line:

```
$ git show  
commit 2fb7e6b97a594fa7f9ccb927849e95c7c70e39f5
```

You'll notice that one of the new files in .git/objects is .git/objects/2f/b7e6b97a594fa7f9ccb927849e95c7c70e39f5, that is the ID of this commit with the first two

characters forming a directory name. The files with hexadecimal names are where Git stores all the objects that make up the content you've stored under version control, and commits are one such object.

2.3.1. The cat-file command

You can ask Git to show you an object from its database by running `git cat-file -p` with the ID of the object you're interested in. The `-p` flag is short for ‘pretty-print’. Let's try that with our commit:

```
$ git cat-file -p 2fb7e6b97a594fa7f9ccb927849e95c7c70e39f5  
tree 88e38705fdbd3608cddbe904b67c731f3234c45b  
author James Coglan <james@jcoglan.com> 1511204319 +0000  
committer James Coglan <james@jcoglan.com> 1511204319 +0000
```

First commit.

This gives us some idea of how Git represents the commit on disk. The first line begins with `tree` and is followed by another long hexadecimal number; this is the ID of a *tree*, which represents your whole tree of files as they were when this commit was made. Then we have some lines listing the author and committer, which each take the form of a person's name, their email address, and a Unix timestamp¹³ and timezone offset for when the commit was created. Finally there is a blank line and then the commit message, which is free text.

This commit refers to a tree with ID `88e38705fdbd3608cddbe904b67c731f3234c45b`. This ID should be the same if you run this example on your computer; Git only stores timestamps and author information on commits, not trees, and the same file contents should get the same ID no matter when or where you store it in Git. Let's take a look at that object, again using `git cat-file`:

```
$ git cat-file -p 88e38705fdbd3608cddbe904b67c731f3234c45b  
100644 blob ce013625030ba8dba906f756967f9e9ca394464a hello.txt  
100644 blob cc628cccd10742baea8241c5924df992b5c019f71 world.txt
```

This is Git's representation of a tree. It creates one tree for every directory in your project, including the root, and that tree lists the contents of the directory. Each entry in a tree is either another tree — i.e. a subdirectory — or a *blob*, which is a regular file. Our project just contains two regular files right now, so we see two blobs here.

Each entry in a tree lists its mode, its type (either `blob` or `tree`), its ID, and its filename. The mode of a file is a numeric representation of its type (whether it's a regular file, a directory, etc.) and its permissions (whether it can be read, written or executed and by whom). We'll discuss this numeric representation further when we look at storing directories and executable files in Chapter 5, *Growing trees*, but for now all you need to know is that `100644` means the entry is a regular file and is readable but not executable.

Just as commits refer to trees, trees refer to their entries by a hexadecimal ID. The file `hello.txt` has ID `ce013625030ba8dba906f756967f9e9ca394464a`, so let's use `cat-file` again to have a look at it:

¹³https://en.wikipedia.org/wiki/Unix_time

```
$ git cat-file -p ce013625030ba8dba906f756967f9e9ca394464a
hello
```

For blobs, Git just stores their literal contents. This is the content we added when we ran `echo hello > hello.txt`. Likewise, `world.txt` has ID `cc628ccd10742baea8241c5924df992b5c019f71` in the tree, and using `cat-file` with this ID prints its file contents:

```
$ git cat-file -p cc628ccd10742baea8241c5924df992b5c019f71
world
```

So, we can use `cat-file` to ask Git to show us what information it has stored for several kinds of objects: commits, trees and blobs. Commits point to trees, and trees point to blobs. But how are those things actually represented on disk? Let's head back up this chain of objects and take a look at the files they're stored in.

2.3.2. Blobs on disk

Looking again at the files stored in `.git/objects`, we can see the IDs of all the objects we just looked at, with their first two characters forming a directory name and the rest of the characters forming the filename within the directory. Git does this because this set of files will become very large over time, and some operating systems wouldn't cope with all these objects being in the same directory. It also makes it faster for Git to find objects using an abbreviated ID¹⁴.

Files in the `.git/objects` directory

```
.git
└── objects
    ├── 2f
    │   └── b7e6b97a594fa7f9ccb927849e95c7c70e39f5
    ├── 88
    │   └── e38705fdbd3608cddbe904b67c731f3234c45b
    ├── cc
    │   └── 628ccd10742baea8241c5924df992b5c019f71
    └── ce
        └── 013625030ba8dba906f756967f9e9ca394464a
```

To find out how Git stores objects in the filesystem, we can again use the `cat` command. Unfortunately, printing any file in `.git/objects` using `cat` will probably give you cryptic-looking output. Here's what happens when I try to cat the file containing our `hello.txt` blob:

```
$ cat .git/objects/ce/013625030ba8dba906f756967f9e9ca394464a
xKOR0C
```

It's pretty inscrutable. It turns out that this is because the file is compressed — Git tries to save disk space by compressing every object it stores. It does this using the DEFLATE¹⁵ compression algorithm, which is implemented by a widely-used library called zlib¹⁶.

¹⁴Section 13.3.3, “Revisions and object IDs”

¹⁵DEFLATE is described in RFC 1951 (<https://tools.ietf.org/html/rfc1951>), and in the zlib documentation (<https://zlib.net/feldspar.html>).

¹⁶<https://zlib.net/>

There isn't a straightforward way to run zlib on the command-line to decompress a file, and people typically resort to using the bindings included with their favourite programming language. Here's a Ruby program to decompress and print data written to standard input:

```
require "zlib"  
puts Zlib::Inflate.inflate(STDIN.read)
```

Ruby allows this same program to be written as a single command-line; the `-r` flag is its command-line equivalent for the `require` function above, and `-e` specifies an inline script to run instead of looking for a Ruby file to execute.

```
$ ruby -r zlib -e "puts Zlib::Inflate.inflate(STDIN.read)"
```

We can package that up as a shell alias. In Bash, `alias` allows you to package a string of command-line code up and refer to it with a single word, and then include it in other commands. Add the following line at the end of `~/.profile`, which is a file your shell runs every time it starts up:

Figure 2.4. Creating a shell alias

```
alias inflate='ruby -r zlib -e "STDOUT.write Zlib::Inflate.inflate(STDIN.read)"'
```

Reload your shell profile by running the following command:

```
$ source ~/.profile
```

Now, you should be able to `cat` the file as before, and pipe the result into our `inflate` alias, which uses Ruby's `zlib` wrapper to decompress the file's contents.

```
$ cat .git/objects/ce/013625030ba8dba9967f9e9ca394464a | inflate  
blob 6hello
```

We can see that this looks very much like the output of `git cat-file` when applied to a blob; it looks as though `cat-file` just echoed the file out verbatim. The only exception to this is a little bit of content at the beginning that we've not seen before:

```
blob 6
```

The word `blob` makes sense — this is a blob object. But what about the `6`, what does that refer to? Maybe it's a length header — the file's original contents were `hello\n`¹⁷, which is six bytes long. Let's use the `wc`¹⁸ command to count the bytes in the decompressed content:

```
$ cat .git/objects/ce/013625030ba8dba906f756967f9e9ca394464a | inflate | wc -c  
13
```

`blob 6` is six bytes, and `hello\n` is six bytes. Where's the extra byte to get us to thirteen? We can find it by taking a look at the file's bytes, using a program called `hexdump`¹⁹. `hexdump` has a number of options to control how it displays a file; we'll be using the `-c` flag, which makes it print a hexadecimal representation of the bytes in a file alongside a textual representation.

```
$ cat .git/objects/ce/013625030ba8dba906f756967f9e9ca394464a | inflate | hexdump -c
```

¹⁷\n is how most programming languages represent the *line feed* character, which appears at the end of every line in most text files. In ASCII, it is represented by the byte `0a`.

¹⁸<https://manpages.ubuntu.com/manpages/bionic/en/man1/wc.1posix.html>

¹⁹<https://manpages.ubuntu.com/manpages/bionic/en/man1/hexdump.1.html>

```
00000000 62 6c 6f 62 20 36 00 68 65 6c 6c 6f 0a |blob 6.hello.|  
0000000d
```

There's a lot to take in here if you've never used `hexdump` before. All files are just a list of bytes, and `hexdump` is showing us the numeric values of all those bytes, written in hexadecimal. The values in the central columns — `62 6c 6f` and so on — are the bytes of the file. The bytes are displayed in rows of sixteen values each. The leftmost column shows the total offset into the file that each row begins at, again in hexadecimal. The value at the bottom of this column — `0000000d` here — is the total size of the file; `d` is thirteen in hexadecimal.

The column on the right, between the `|` characters, displays the corresponding ASCII character that each byte represents. If the byte does not represent a printable character in ASCII²⁰ then `hexdump` prints a `.` in its place to make it easier to count by eye — do not confuse this with an actual `.` character, which is byte value `2E`.

The bytes `62 6c 6f 62` are the ASCII representation of the word `blob`, and `hexdump` displays as much in its right hand column. Byte `20` is the space character, and the byte `36` is the ASCII representation of the textual digit `6`. The byte following these is what we've been looking for: `00`, a null byte. `cat` won't print these, so it looks like there's nothing between `6` and `he11o` in its output, but there is something! This null byte separates the length header from the actual content of the blob object, and that accounts for the thirteenth byte.

So, Git stores blobs by prepending them with the word `blob`, a space, the length of the blob, and a null byte, and then compressing the result using `zlib`.

2.3.3. Trees on disk

Whereas blobs are stored pretty much as `cat -file` shows them to us, trees are different, and we're going to need to spend a little more time with `hexdump` to understand them.

Here's what we see if we decompress the file that's stored on disk:

```
$ cat .git/objects/88/e38705fdbd3608cddbe904b67c731f3234c45b | inflate  
tree 74100644 hello.txt6%  
#VFJ100644 world.txtt+$Y$#+\q
```

This doesn't look like anything we can easily understand; it contains a lot of non-text characters that look like garbage when displayed like this. Let's run it through `hexdump` and take a closer look.

```
$ cat .git/objects/88/e38705fdbd3608cddbe904b67c731f3234c45b | inflate | hexdump -C  
00000000 74 72 65 65 20 37 34 00 31 30 30 36 34 34 20 68 |tree 74.100644 h|  
00000010 65 6c 6c 6f 2e 74 78 74 00 ce 01 36 25 03 0b a8 |ello.txt...6%...|  
00000020 db a9 06 f7 56 96 7f 9e 9c a3 94 46 4a 31 30 30 |....V.....FJ100|  
00000030 36 34 34 20 77 6f 72 6c 64 2e 74 78 74 00 cc 62 |644 world.txt..b|  
00000040 8c cd 10 74 2b ae a8 24 1c 59 24 df 99 2b 5c 01 |...t+..$.Y$..+\..|  
00000050 9f 71 |.q|  
00000052
```

²⁰A byte is 8 bits, and can take any value from 0 to 255_{10} , or FF_{16} . ASCII only defines meanings for byte values up to 127_{10} or $7F_{16}$, and values below 32_{10} or 20_{16} represent *non-printing* characters.

Now we can see a few recognisable things: the file begins with the type of the object and its length in bytes, followed by a null byte: `tree 74` followed by the byte `00`. We can see the filenames `hello.txt` and `world.txt` in there. And, each filename is preceded by the string `100644`, the file mode that we saw printed in the `cat-file` output. Let's highlight the bits I'm referring to for clarity:

```
00000000 74 72 65 65 20 37 34 00 31 30 30 36 34 34 20 68 |tree 74.100644 h|
00000010 65 6c 6c 6f 2e 74 78 74 |ello.txt |
00000020 31 30 30 | 100 |
00000030 36 34 34 20 77 6f 72 6c 64 2e 74 78 74 |644 world.txt |
00000040 |
00000050 |
00000052 | |
```

Now, let's highlight the bytes that aren't accounted for by this information:

```
00000000 |
00000010 00 ce 01 36 25 03 0b a8 | ...6%... |
00000020 db a9 06 f7 56 96 7f 9e 9c a3 94 46 4a |....v.....FJ |
00000030 00 cc 62 | ..b |
00000040 8c cd 10 74 2b ae a8 24 1c 59 24 df 99 2b 5c 01 |...t+..$.Y$..+\.. |
00000050 9f 71 | .q |
00000052
```

The text characters like `6%`, `v` and `FJ` displayed in the right-hand column don't immediately look like anything meaningful; it's probably the case that the remaining bytes are a binary encoding of something, and some of the bytes just happen to correspond to ASCII characters, and that's what `hexdump` is displaying.

As a reminder, here's how `cat-file` displays this tree object:

```
$ git cat-file -p 88e38705fdbd3608cddbe904b67c731f3234c45b
100644 blob ce013625030ba8dba906f756967f9e9ca394464a hello.txt
100644 blob cc628cccd10742baea8241c5924df992b5c019f71 world.txt
```

What do you notice that's similar? Take a few moments to compare the hexdump with the `cat-file` representation.

Look at the first string of unaccounted-for bytes. We know `00` is a null byte, typically used to separate bits of information in a binary format. Here it's used to indicate the end of the filename, since there's no length information to tell us how long that is. Then we see `ce 01 36 25 03` — this looks a lot like the ID printed next to `hello.txt`, the ID of that blob object in the database. Likewise, the bytes `cc 62 8c cd 10` and so on look just like the ID of the `world.txt` blob.

Git is storing the ID of each entry in a packed format, using twenty bytes for each one. This might seem odd at first, because object IDs are usually displayed as forty characters — how do you fit forty characters into twenty bytes?

The trick is that object IDs are hexadecimal representations of numbers. Each hexadecimal digit represents a number from zero to fifteen, where ten is represented by `a`, eleven by `b`, and so on up to `f` for fifteen. Each hexadecimal digit represents four bits in binary; here's a table of decimal values and their hexadecimal and binary representations:

Figure 2.5. Decimal, hexadecimal and binary representations of numbers

Dec	Hex	Bin	Dec	Hex	Bin
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	a	1010
3	3	0011	11	b	1011
4	4	0100	12	c	1100
5	5	0101	13	d	1101
6	6	0110	14	e	1110
7	7	0111	15	f	1111

In a forty-digit object ID, each digit stands for four bits of a 160-bit number. Instead of splitting those bits into forty chunks of four bits each, we can split it into twenty blocks of eight bits — and eight bits is one byte. So all that’s happening here is that the 160-bit object ID is being stored in binary as twenty bytes, rather than as forty characters standing for hexadecimal digits. `hexdump` is then displaying each of those bytes back to us in hexadecimal, and since a byte is eight bits, it requires two hexadecimal digits to represent each one.

In fact, this is precisely why we usually represent bytes in hexadecimal. When dealing with binary data, we often want to operate on the individual bits within each byte. Since each hex digit represents exactly four bits, they provide a compact notation for the byte’s value that makes doing bitwise arithmetic much easier. When you see the byte `ce`, you just combine the bits for `c` and the bits for `e` and get `11001110`. If we wrote the byte in decimal it would be 206 and it would be harder to do this conversion by eye²¹.

Now we know how trees are stored. Git creates a string for each entry consisting of its mode in text (e.g. `100644`), a space, its filename, a null byte, and then its ID packed into a binary representation. It concatenates all these entries into a single string, then prepends the word `tree`, a space, and the length of the rest of the content.

You might notice that `cat -file` printed the word `blob` next to each item, but that information is not expressed directly in the file on disk. That’s because it’s implied by the entry’s mode: the `10` at the beginning of the number `100644` indicates the entry is a regular file, rather than a directory. A more extensive explanation can be found in Section 5.1.1, “File modes”.

2.3.4. Commits on disk

Fortunately, commits are a bit easier to interpret than trees. Printing the file containing commit `2fb7e6b` and inflating it prints something pretty familiar:

```
$ cat .git/objects/2f/b7e6b97a594fa7f9ccb927849e95c7c70e39f5 | inflate
commit 178tree 88e38705fdbd3608cddbe904b67c731f3234c45b
author James Coglan <james@jcoglan.com> 1511204319 +0000
committer James Coglan <james@jcoglan.com> 1511204319 +0000
```

First commit.

This looks exactly like the output of `cat -file`, with the usual pattern of prepending the content with its type (`commit`), its length and a null byte. You should use `hexdump` to verify there’s a null byte between `commit 178` and the word `tree`.

²¹If you’re unfamiliar with binary numbers and bitwise arithmetic, a short introduction is available in Appendix B, *Bitwise arithmetic*.

Commits are stored as a series of headers, followed by the message. This one contains the following headers:

- **tree**: All commits refer to a single tree that represents the state of your files at that point in the history. Git stores commits as pointers to a complete snapshot of the state of your project, rather than storing diffs between versions. It uses some tricks to make these snapshots more space-efficient, as we'll see later.
- **author**: This field is metadata that doesn't affect any computations Git might like to do on the commit, like calculating its diff or merging it with another. It contains the name and email address of the person who authored this commit, and the Unix timestamp for when it was authored.
- **committer**: This is also metadata and it will often be the same as author. The fields differ in cases where somebody writes some changes and then someone else amends the commit²², or cherry-picks it onto another branch²³. Its time reflects the time the commit was actually written, while author retains the time the content was authored. These distinctions originate from the workflow used by the Linux Kernel²⁴, which Git was originally developed to support.

These headers are followed by a blank line, and then the commit message follows. Just like the other object types, the commit data is prepended with its type and length and then compressed using zlib before being written to disk.

2.3.5. Computing object IDs

The only remaining question is how Git calculates the filename for each object. You might have heard that it names objects by taking the SHA-1 hash²⁵ of their content, but what exactly is it taking the hash of? We can find out by trying a few things ourselves.

This Ruby program calculates and prints the SHA-1 hash of three different strings: the literal string `hello\n`, that same string prepended with `blob 6` and a null byte, and this prefixed string after compression.

```
require "digest/sha1"
require "zlib"

string = "hello\n"
puts "raw:"
puts Digest::SHA1.hexdigest(string)

blob = "blob #{ string.bytesize }\0#{ string }"
puts "blob:"
puts Digest::SHA1.hexdigest(blob)

zipped = Zlib::Deflate.deflate(blob)
puts "zipped:"
puts Digest::SHA1.hexdigest(zipped)
```

²²Section 22.3.1, “Amending the HEAD”

²³Chapter 23, *Cherry-picking*

²⁴<https://www.kernel.org/>

²⁵<https://en.wikipedia.org/wiki/SHA-1>

Running this program prints the following:

```
raw:
f572d396fae9206628714fb2ce00f72e94f2258f
blob:
ce013625030ba8dba906f756967f9e9ca394464a
zipped:
3a3cca74450ee8a0245e7c564ac9e68f8233b1e8
```

We can see that the second hash printed matches the ID we saw Git assign to the blob, so now we know that it takes the SHA-1 hash of the file's content after adding the type and length prefix but before compressing it.

Git exploits a property of SHA-1 to make many of its operations safer and faster. SHA-1 is what's known as a *hash function*²⁶, which means that it takes an input string of arbitrary size and processes all of its bytes to produce a fixed-size number as output. In the case of SHA-1 this number is 160 bits, or 20 bytes, in size, and is usually expressed as a 40-digit hexadecimal representation. SHA-1 is a particular kind of hash function called a *cryptographic hash function*²⁷, which are designed with the aim that it's effectively impossible in practice to find two inputs that hash to the same output.

Of course, because hash functions accept input of any size and produce a fixed-size output, there is in theory an infinite set of inputs that produce the same result. However, cryptographic hash functions are designed so that the only practical way to find any two such inputs is by brute-force search over a very large input space, and no two inputs of relatively small size will hash to the same result. Since SHA-1 produces 160-bit outputs, there are 2^{160} , or 1,461,501,637,330,902,918,203,684,832,716,283,019,655,932,542,976 (or 1.46×10^{48} for short) possible object IDs Git could generate.

This means that if you wanted to generate every possible SHA-1 output within the age of the observable universe, you would need to hash ten thousand billion billion inputs every second for thirteen billion years. If you wanted to search for inputs so that any pair hashed to the same value — called a *collision* — you'd ‘only’ need to try 1.21×10^{24} inputs, or ten million inputs per second if you have the entire age of the universe to work with. In other words, it's *computationally infeasible* to find a collision, assuming SHA-1 provides the guarantees that it was designed to.

The upshot of this is that it's almost guaranteed that no two distinct Git objects, in any repository, will be given the same ID, and that makes Git easier to run as a distributed system. If Git used sequential integers for its objects, as some older version control systems do, then both Alice and Bob, working in their own repositories, could both generate commit number 43, attempt to push it to their shared repository, and a conflict would arise. Either you must throw away one of the commits numbered 43, or change one of their numbers and propagate that change to the other parties. This is not impossible, but it is complicated.

The problem is that Alice and Bob cannot both say ‘this is the forty-third commit’ without some form of co-ordination. The easiest solution is to centralise the system, having a central server that accepts and numbers commits, but that goes against Git's design goals of working offline and everyone having a full working independent repository on their machine. If the objects

²⁶https://en.wikipedia.org/wiki/Hash_function

²⁷https://en.wikipedia.org/wiki/Cryptographic_hash_function

generated by Alice and Bob are identified by their contents and always have different IDs, then no co-ordination is necessary when ‘numbering’ the commits, and they can both push to a shared repository without overwriting each other’s contributions.

It also helps speed up many of the comparisons Git has to do. If two objects in Git have the same ID, then to the extent guaranteed by SHA-1 they must have the same content. That means that if you see two blobs with the same ID while calculating a diff between two commits²⁸, then you know they have the same content and you don’t need to actually compare them. Likewise if two trees have the same ID then you don’t need to compare their entries; since the contents of a tree includes the hashes of all its entries, any change anywhere inside a tree results in a new ID. This radically reduces the amount of work required when comparing commits on large projects.

When synchronising two repositories over a network, you can arrange things so that if an object ID appears in one repository, then it and everything it refers to — the whole chain of parent commits, trees and blobs — does not need to be re-downloaded from the remote²⁹. If you see an object in the remote whose ID you already have, you can just skip it, and this makes it much easier to only download the object’s you’re missing without a lot of redundant data transfer.

This also explains why Git hashes objects before compressing them. Deflate compression has a number of settings that control how aggressively the data is compressed, whether it should optimise for small size or fast decompression, and how much memory is needed to decompress messages. So, the same input can lead to many different compression outputs. We wouldn’t want to be tricked into thinking that the same object compressed with different settings was actually two different objects, and that’s what would happen if hashing took place using the compressed output. This would negate all the benefits of being able to infer that two objects are the same just from their IDs, and so when we assign IDs we only want to take the object’s actual content into account, not the compression settings.

2.3.6. Problems with SHA-1

Although SHA-1, like all cryptographic hashes, was designed with the aim that finding collisions is computationally infeasible, no algorithm is perfect. Using brute force, if you hashed $2^{n/2}$ inputs, where n is the size of the hash output in bits, then there is a 50% chance that among all the inputs you’ve tried there will be a collision — that is, a pair of inputs that are different but hash to the same value. A hash is considered broken if there exists an attack requiring fewer steps to get the same probability of success. For SHA-1, this means finding an attack requiring fewer than 2^{80} or 1.21×10^{24} operations.

In February 2005, two months before the Git project was started³⁰, an attack was reported³¹ that was able to find collisions in 2^{69} steps, two thousand times faster than brute force. This was considered to be exploitable if you had a large budget at your disposal — Bruce Schneier made this comparison at the time:

In 1999, a group of cryptographers built a DES cracker. It was able to perform 2^{56} DES operations in 56 hours. The machine cost \$250K to build, although duplicates could be made in the \$50K–\$75K range. Extrapolating that machine

²⁸Section 14.1, “Telling trees apart”

²⁹Section 28.1, “Pack negotiation”

³⁰<https://github.com/git/git/commit/e83c5163316f89bfbde7d9ab23ca2e25604af290>

³¹https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html

using Moore's Law, a similar machine built today could perform 2^{60} calculations in 56 hours, and 2^{69} calculations in three and a quarter years. Or, a machine that cost \$25M–\$38M could do 2^{69} calculations in the same 56 hours.

— Bruce Schneier

In August 2006, the possibility of migrating to SHA-256 was discussed on the mailing list³². The thread contains much interesting discussion of the pros and cons of migrating, and chief among the cons is that the intention of Git's maintainers is that it does not rely on SHA-1 as proof of trust. It uses SHA-1 as an integrity check against accidental damage, not as part of a digital signature to prove that code received from a third party is trustworthy.

The possibility that an object's contents will change by *accident*, for example because a disk was corrupted or a network transfer went wrong, and it will still hash to the same value, is absolutely remote. For all practical purposes, SHA-1 works fine as an integrity check; that is, it's a security measure against infrastructure accidentally damaging your data, not against malicious attackers attempting to use the hashes as proof of trust.

For an object to be *deliberately* replaced by an attacker, there are several barriers in place. First, this requires finding not just any collision you can, but a collision with a specific blob in the repository being attacked, which would contain a minuscule subset of all possible object IDs. Finding a collision against a single specific input is called a *second pre-image attack* and is considerably harder than searching for any pair of colliding inputs. Second, the malicious input would need to be a valid Git blob, meaning that when prefixed with the word `blob` and its length, it will still hash to collide with the intended target. Again, this is very unlikely. Third, because of the type of content typically stored in Git, it's unlikely that maintainers would not notice the code being manipulated. They'll either see the diff when they review what they've pulled, or they'll notice when their project refuses to compile or runs with surprising behaviour. It's not *impossible* to hide code manipulation from project maintainers, but it does make it harder to pull off this sort of attack. Constructing a collision against a specific file, that's a valid Git blob, and a valid source file, is still prohibitively hard.

Nevertheless, Git puts some measures in place to protect against malicious input. When pulling from another repository, it will not overwrite objects you already have locally, so each ID always points to what you originally stored for it. After pulling, it prints a diff stat so you're alerted to what's changed and you can inspect the changed files. However, Git's main argument has always been that the hash isn't that important, and security is a social problem of deciding who you trust to pull patches from, and checking the code they're sending you. The argument goes that at scale, an attacker might be able to distribute their bad objects to a few members of a network, but hopefully enough project collaborators would have the correct version that those who'd received the bad version would soon find out.

It has also been argued that there are much easier ways to attack a repo, for example it's always going to be easier for an attacker to convince you to merge malicious code that you don't notice, than to try to forge a SHA-1 hash, and no version control system can protect you against that.

In February 2017, Google announced an attack on SHA-1 known as SHAttered³³. This attack requires only 9.22×10^{18} SHA-1 computations, but rather than searching for arbitrary collisions,

³²<https://marc.info/?l=git&m=115670138630031&w=2>

³³<https://shattered.io/>

it reveals a method for manufacturing them. Google deliberately constructed the two PDF files that form the collision so that they contained structures that would exploit aspects of SHA-1's design and lead to a collision. This pre-generation phase is still expensive, requiring 6,500 CPU-years and 110 GPU-years to complete³⁴. While this still does not represent a pre-image attack and is very expensive to pull off, it does usher in the possibility of other attacks, and prompted renewed calls for SHA-1 to be replaced within Git and elsewhere.

For example, a thread on the cryptography mailing list³⁵ opens with the possibility that an attacker could conduct a *birthday attack*³⁶:

Concretely, I could prepare a pair of files with the same SHA1 hash, taking into account the header that Git prepends when hashing files. I'd then submit that pull-req to a project with the "clean" version of that file. Once the maintainer merges my pull-req, possibly PGP signing the git commit, I then take that signature and distribute the same repo, but with the "clean" version replaced by the malicious version of the file.

Thinking about this a bit more, the most concerning avenue of attack is likely to be tree objects, as I'll bet you you can construct tree objs with garbage at the end that many review tools don't pick up on.

— Peter Todd

The discussion following this message suggests that SHAttered isn't an immediate threat to Git's security model, partly because of the reasons covered above, but also because of the details of the attack. For example:

So what is actually affected?

Situations where you're creating signatures that need to be valid for a long time, and where the enormous latency between legitimate signature creation and forgery isn't an issue (this is why email won't be affected, having to wait a year between email being sent and the forgery being received will probably raise at least some suspicions of foul play).

— Peter Gutmann

The argument here is that because manufacturing collisions takes a long time, by the time an attacker got around to trying to distribute their collision for a target object, all the developers in the network would already have the correct version and would keep that. However, at this point, discussion around this problem mostly takes the line that SHA-1 does need to be replaced, we just don't need to panic about it yet³⁷.

In response to SHAttered, Linus Torvalds put out a statement to this effect³⁸ essentially recapping the arguments above but also stating that work to migrate Git to another hash function

³⁴<https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html>

³⁵<http://www.metzdowd.com/pipermail/cryptography/2017-February/031595.html>

³⁶https://en.wikipedia.org/wiki/Birthday_attack

³⁷A good summary and critique of some of Git's assumptions in this space can be found in the final message in this thread: <http://www.metzdowd.com/pipermail/cryptography/2017-February/031623.html>

³⁸This was originally posted at <https://plus.google.com/+LinusTorvalds/posts/7tp2gYWQugL>. This page may disappear as Google is now in the process of shutting down Google+, so you may need to consult the Internet Archive: https://web.archive.org/web/*/https://plus.google.com/+LinusTorvalds/posts/7tp2gYWQugL.

is in progress. The method of constructing colliding inputs can also be easily detected, and Git has included patches to check for these types of collisions and reject them.

2.4. The bare essentials

In creating a new repository and making one commit to it, we've already discovered the files that sit behind a lot of Git's features. We've seen the object database where all versioned content lives, the refs that tell us where each branch ends, the HEAD file that tells us what we have checked out right now, the index where the current commit is cached, the logs that track each ref, and the configuration and hook scripts that control how this repository works.

To make a start on our own implementation, we'd like to begin with as little as possible, the bare minimum we need to make one valid commit. By a process of trial and error, we discover that we can delete most of the contents of .git other than the objects we've stored, and the chain of files that causes HEAD to point at the current commit. We can leave only these files in place, and git show will still display the commit we just made:

Figure 2.6. Minimum viable Git repository

```
.git
├── HEAD
├── objects
│   ├── 2f
│   │   └── b7e6b97a594fa7f9ccb927849e95c7c70e39f5
│   ├── 88
│   │   └── e38705fdbd3608cddbe904b67c731f3234c45b
│   ├── cc
│   │   └── 628ccd10742baea8241c5924df992b5c019f71
│   └── ce
│       └── 013625030ba8dba906f756967f9e9ca394464a
└── refs
    └── heads
        └── master
```

The file HEAD contains a symref to refs/heads/master, and that file contains 2fb7e6b97a594fa7f9ccb927849e95c7c70e39f5. If we change HEAD to contain that commit ID directly, then we can also remove refs/heads/master. However, the refs directory itself must be present for Git to consider the repository valid.

This gives us a good target to aim at to get our implementation off the ground. If we can write some code that stores itself as a commit, a tree and some blobs, and writes the ID of that commit to HEAD, then we're up and running.

Part I. Storing changes

3. The first commit

In the previous chapter, we explored all the files that are created when we run `git init` and `git commit` to create a simple commit containing a couple of files. In this chapter, our aim will be to get our own implementation off the ground by writing just enough code that can store itself as a valid Git commit. We'd like to get to a system we can use to keep the history of the project as soon as possible, and in this initial period where nothing is yet under version control, we'll implement as few features as we can get away with in order to store the code we need.

In particular, this first pass will omit a few things you'd expect this workflow to support:

- We will not have any subdirectories or executable files or symlinks in the source code, only regular files. This will keep our implementation of trees simple.
- There will be no `add` command and no index. The `commit` command will commit all the files in the working tree as they appear at that moment.
- We won't have any command-line argument processing in the `commit` command. The author details will be read from environment variables and the commit message will be read from standard input.

To distinguish our implementation from the standard Git software, I'll be calling this implementation Jit. For compatibility with Git it will use the `.git` directory and all the same file structures, but its name on the command line will be `jit` so that it's clear in each example which implementation is being used.

3.1. Initialising a repository

The very first step in the life of any Git repository comes when we run `git init`. This sets up the file structure of the repository ready for us to begin adding commits. As we saw in Section 2.4, “The bare essentials”, only three items in the `.git` directory are essential for Git to treat a directory as a valid repository: the directories `objects` and `refs`, and a `HEAD` file that's a symref to a file in `refs`. We will implement an `init` command to create these directories, with one caveat.

We're aiming for the smallest possible set of features here. While Git usually creates `HEAD` as a symref — it's a pointer to `refs/heads/master` rather than a direct reference to a commit — it's also perfectly valid for `HEAD` to just contain a commit ID, and we can make a chain of commits from there without having a named branch checked out. To begin with, we're going to omit named branches and stick to a single line of development, storing the current commit ID in `HEAD`. That means that at first there will be nothing to store in `HEAD`, because we've not created any commits. So initially, our `init` will not create a `HEAD` file.

One final thing to mention before diving into the code is how our implementation will be run. We won't implement support for executable files in the data model in this first commit, but fortunately Ruby's method of running files is very simple. You just write some Ruby code into a file, say `my_program.rb`, and you can then run it by typing `ruby my_program.rb` into the terminal. Any command-line arguments you enter after `my_program.rb` are made available by Ruby in a global array of strings called `ARGV`, short for ‘argument values’.

Therefore, what we want to be able to do is run our program with `init` and a directory name:

```
$ ruby jit.rb init path/to/repository
```

If the path to the repository is not provided, then Jit should use the current directory as the place to create the repository.

3.1.1. A basic `init` implementation

Here is a brief program that does what we want. It begins by loading two modules from the Ruby standard library: `fileutils`¹ and `pathname`², which are useful for filesystem operations. Then it checks which command it's been asked to run: `ARGV.shift` removes and returns the first item from the command-line arguments. When this command is equal to "init" then we run the initialisation code, otherwise we print an error message. We'll examine this program piece-by-piece next.

```
# jit.rb

require "fileutils"
require "pathname"

command = ARGV.shift

case command
when "init"
    path = ARGV.fetch(0, Dir.getwd)

    root_path = Pathname.new(File.expand_path(path))
    git_path  = root_path.join(".git")

    ["objects", "refs"].each do |dir|
        begin
            FileUtils.mkdir_p(git_path.join(dir))
        rescue Errno::EACCES => error
            $stderr.puts "fatal: #{error.message}"
            exit 1
        end
    end

    puts "Initialized empty Jit repository in #{git_path}"
    exit 0
else
    $stderr.puts "jit: '#{command}' is not a jit command."
    exit 1
end
```

Within the `init` block, we need to figure out where to put the new repository. `ARGV.fetch(0, Dir.getwd)` retrieves the first element of `ARGV` (after `command` has been removed from the front) if it exists, otherwise it returns a default value, in this case `Dir.getwd`³. On Unix-like systems like macOS and Linux, every process has a *working directory*, which among other things is the directory relative to which paths used by the process will be resolved. When you run a program

¹<https://docs.ruby-lang.org/en/2.3.0/FileUtils.html>

²<https://docs.ruby-lang.org/en/2.3.0/Pathname.html>

³<https://docs.ruby-lang.org/en/2.3.0/Dir.html#method-c-getwd>

from the terminal, the directory you have navigated to using `cd` becomes the initial working directory of the new process.

So, if my terminal is currently in the directory `/Users/jcoglan` and I run a program that calls `File.read("README.txt")`, then the operating system will open `/Users/jcoglan/README.txt`. In the case of `git init`, the working directory is where the new `.git` directory will be placed if no other location is specified.

We then do two important things with the path we've found.

First, we pass it through `File.expand_path`⁴ to convert any relative paths given on the command-line to absolute ones by resolving them relative to the current working directory. It is generally good practice when you receive any user input that can have a relative meaning, like a path or a date or time, to infer its absolute value as soon as possible and use that as the basis for further work. In the case of paths supplied on the command-line, the user will expect them to be relative to the current directory, but because a process's working directory can be changed, we should capture this assumption as soon as we read the input from the user. Making the path absolute will also prevent some errors that could occur when the path is combined with others.

Second, we wrap the resulting string in a `Pathname` object, which provides a set of methods for manipulating paths. When working with data that can be represented as strings — paths, URLs, HTML documents, SQL queries, etc. — wrapping the data in a typed object can provide a more structured interface to interpreting and changing the value, and helps prevent many errors that would be caused by the string being directly manipulated, especially if the user does not know some of the rules about how the strings in question are formatted. Again, these sorts of conversions should happen as early as possible at the edge of a system so all the internal components work with structured data. I try not to use basic string manipulation methods on such values and instead use functions specific to the type of data where possible.

Finally we get to the payload of this command. Having found out where to put the repository, we construct the path for the `.git` directory — `Pathname#join`⁵ adds new components to the path using the pathname separator appropriate for your system — and then create two directories, `objects` and `refs` inside that, using `FileUtils.mkdir_p`⁶. This imitates the command-line `mkdir -p` command⁷, which creates a directory along with any parent directories that might not yet exist.

3.1.2. Handling errors

Whenever we talk to the filesystem, a variety of errors can occur; a file we need might be missing, or one we're trying to create might already exist. With `mkdir_p`, the most likely error is that we're not allowed to write a directory that's a parent of the one we're trying to create. In this event, Ruby will raise an error called `Errno::EACCES`. The `Errno`⁸ error namespace is used to represent errors that are thrown by calls to the operating system, which have standard error numbers. Such *system calls*⁹ are usually done in C, where errors are represented by magic

⁴https://docs.ruby-lang.org/en/2.3.0/File.html#method-c-expand_path

⁵<https://docs.ruby-lang.org/en/2.3.0/Pathname.html#method-i-join>

⁶https://docs.ruby-lang.org/en/2.3.0/FileUtils.html#method-i-mkdir_p

⁷<https://manpages.ubuntu.com/manpages/bionic/en/man1/mkdir.1posix.html>

⁸<https://docs.ruby-lang.org/en/2.3.0/Errno.html>

⁹https://en.wikipedia.org/wiki/System_call

numbers, and the operating system includes a set of named constants for referring to these numbers¹⁰. These constants are defined in the C header file `errno.h` and Ruby just mirrors the standard names.

When `git init` tries to create a directory in a place you don't have permission to create files, it will exit with status code `1`. You can see this using the special shell variable `$?`, which holds the exit status of the last command.

```
$ mkdir private
$ sudo chown root private

$ git init private
/Users/jcoglan/private/.git: Permission denied

$ echo $?
1
```

All programs set an exit status when they end, and this status number tells the caller whether the program was successful or not. A status of `0` is considered successful and all other values are interpreted as failures. We could use many different numbers to indicate different types of error, but to keep things simple at this stage we'll just exit with status `1` whenever we encounter an error.

The other thing to notice is that when we want to report an error to the user, we print to `$stderr`. This is Ruby's representation of the *standard error* stream. All processes have two output streams, known as standard output and standard error, or `stdout` and `stderr` for short. Printing to either one will make the text show up in your terminal, but there are ways of separating the two streams, for example by routing them into different log files. Programs typically emit error messages and warnings on `stderr` and reserve `stdout` for successful output. In Ruby, calling `puts` by itself sends the text to `stdout`.

3.1.3. Running Jit for the first time

Having completed the work of the `init` command, we print a message to tell the user where we placed the new repository. Let's run our program and see that it works:

```
$ ruby jit.rb init
Initialized empty Jit repository in /Users/jcoglan/jit/.git
```

We can check the resulting files using `tree`, this time passing the `-a` flag to get it to print the contents of the `.git` directory¹¹

```
$ tree -a
.
├── .git
│   ├── objects
│   └── refs
└── jit.rb
```

¹⁰For example, <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git/tree/include/uapi/asm-generic/errno-base.h?h=v4.13.12>

¹¹On Unix-like systems, files whose names begin with a period are considered *hidden* by most tools that list directory contents. This is a historical accident. In order to prevent the `ls` program from printing the entries representing the current folder `(.)` and its parent `(..)`, the creators of Unix checked whether each filename began with a period, because this required less code than checking if the whole name was equal to either `"."` or `".."`. Because of this, any file whose name begins with a `"."` will not be printed by `ls`, and the shortcut became a cultural convention. Source: https://web.archive.org/web/*/https://plus.google.com/+RobPikeTheHuman/posts/R58WgWwN9jp

In this short bit of code, we've met many of the standard elements of a Unix program. We've learned that every process has:

- A *working directory*, the directory relative to which all paths are resolved. Ruby calls this `Dir.getwd`.
- Some command-line arguments, which are the list of strings you type after the program name in the terminal. Ruby calls this `ARGV`.
- Two output streams, called standard output and standard error. Ruby calls these `$stdout` and `$stderr`, but you don't have to say `$stdout` explicitly.
- An exit status, a numeric value returned by the program when it ends.

We've also met most of the modules that contain Ruby's filesystem interfaces: `File`, `Dir`, `Pathname` and `FileUtils`. Their design is a little haphazard and there is not a clear separation between methods for manipulating paths and methods for reading and writing files, but usually if you need to do something with the filesystem you'll want one of those modules.

3.2. The commit command

With a basic implementation of `init` working, we can now work towards writing the first commit. We want to keep things simple and write just enough code to store whatever we've written so far as a valid commit so we can begin recording the project's history as soon as possible. With that in mind, I'm going to begin introducing some abstractions, but they will be very bare-bones and are just an initial attempt to organise the code around the concepts it represents, so we don't end up with one huge script we need to tease apart later.

To begin with, let's add another branch to the `case` statement in `jit.rb` that picks some code to run based on the given command. With the `commit` command, we don't take the path to the repository as a command-line argument; we just assume the current working directory is the location of the repo. After setting up a few paths to things we'll need, we create a new `Workspace` object and ask it to list the files in the repository.

```
# jit.rb

when "commit"
  root_path = Pathname.new(Dir.getwd)
  git_path  = root_path.join(".git")
  db_path   = git_path.join("objects")

  workspace = Workspace.new(root_path)
  puts workspace.list_files
```

The `Workspace` class is responsible for the files in the working tree — all the files you edit directly, rather than those stored in `.git`. We'll place this class in its own file; add this at the top of `jit.rb` to load it:

```
require_relative "./workspace"
```

And then, we can put just enough code in `workspace.rb` to handle the `list_files` call by returning all the files in the project. To begin with, our project doesn't have any subdirectories,

only files at the top level, so it suffices to call `Dir.entries`¹² to fetch the items in the root directory. We need to remove things we want to ignore from this list; Ruby returns the strings `".."` and `".."` in this list to represent the directory itself and its parent, and we also want to ignore the `.git` directory.

```
# workspace.rb

class Workspace
  IGNORE = [".", "..", ".git"]

  def initialize(pathname)
    @pathname = pathname
  end

  def list_files
    Dir.entries(@pathname) - IGNORE
  end
end
```

Let's run what we have so far to check we get what we expect:

```
$ ruby jit.rb commit
jit.rb
workspace.rb
```

This is just what we expected: a list of each source file in the project. Now, we can begin storing those files off as blobs in the Git object database.

3.2.1. Storing blobs

The next step after listing the files from the workspace is to store their contents in the Git database by creating some blobs. Let's extend our `commit` command handler by creating a `Database` object, which will be responsible for managing the files in `.git/objects`. For each file in the workspace, we'll create a `Blob` object that contains the file's contents, and then we'll tell the database to store that blob object.

```
# jit.rb

when "commit"
  root_path = Pathname.new(Dir.getwd)
  git_path = root_path.join(".git")
  db_path = git_path.join("objects")

  workspace = Workspace.new(root_path)
  database = Database.new(db_path)

  workspace.list_files.each do |path|
    data = workspace.read_file(path)
    blob = Blob.new(data)

    database.store(blob)
  end
```

The `Database` class will be loaded from `database.rb`.

¹²<https://docs.ruby-lang.org/en/2.3.0/Dir.html#method-c-entries>

```
# jit.rb

require_relative "./database"
```

Within the `Workspace` class, we'll add a method to read the contents of a file, via a simple call to Ruby's `File.read`, after joining the requested path onto the workspace's root pathname.

```
# workspace.rb

def read_file(path)
  File.read(@pathname.join(path))
end
```

The result of this method will be passed into a `Blob` object. The value of representing the contents of the file as an object will become clearer later, when we implement trees and commits, but for now all we need to know is that `Blob` is a very basic class that wraps a string that we got by reading a file. It also has `type` method that returns "blob"; in Section 2.3, "Storing objects" we saw that each object on disk begins with its type and this method will allow the database to request the type of the object so it can be written into the object file. Finally, it has a property called `oid`, which will store the blob's object ID once the database has calculated the SHA-1 hash of the object.

`to_s` is the usual name in Ruby for the method that turns an object into a string, and we will use this to serialise each object we store. Blobs are stored by dumping the contents of the original file back out into the Git database, so all this `to_s` method needs to do is return the original file contents.

```
# blob.rb

class Blob
  attr_accessor :oid

  def initialize(data)
    @data = data
  end

  def type
    "blob"
  end

  def to_s
    @data
  end
end
```

Now, we come to the `Database` class, which stores all our content away in `.git/objects`. Its job for the time being is to take this `Blob` object we've constructed and store it on disk according to the Git database format.

Recall that all objects are stored in the database by serialising them, and then prefixing the resulting string with the object's type, a space, the length of the string, and a null byte. This content is then hashed using SHA-1 to compute the object's ID, after which the entire string is compressed and written to disk. Here's the `Database` class up to the `store` method, which we're calling from `jit.rb`.

```
# database.rb

require "digest/sha1"
require "zlib"

require_relative "./blob"

class Database
  def initialize(pathname)
    @pathname = pathname
  end

  def store(object)
    string = object.to_s.force_encoding(Encoding::ASCII_8BIT)
    content = "#{object.type} #{string.bytesize}\n#{string}"

    object.oid = Digest::SHA1.hexdigest(content)
    write_object(object.oid, content)
  end

  #
end
```

Note that we set the string's encoding¹³ to ASCII_8BIT, which is Ruby's way of saying that the string represents arbitrary binary data rather than text per se. Although the blobs we'll be storing will all be ASCII-compatible source code, Git does allow blobs to be any kind of file, and certainly other kinds of objects — especially trees — will contain non-textual data. Setting the encoding this way means we don't get surprising errors when the string is concatenated with others; Ruby sees that it's binary data and just concatenates the bytes and won't try to perform any character conversions.

The method ends by using `Digest::SHA1.hexdigest` to calculate the object's ID from its contents, as a hexadecimal string. It then calls a private method called `write_object` that does the work of writing the object to disk. Below is the rest of the `Database` class, including this method.

```
# database.rb

private

def write_object(oid, content)
  object_path = @pathname.join(oid[0..1], oid[2...-1])
  dirname     = object_path.dirname
  temp_path   = dirname.join(generate_temp_name)

  begin
    flags = File::RDWR | File::CREAT | File::EXCL
    file = File.open(temp_path, flags)
    rescue Errno::ENOENT
      Dir.mkdir(dirname)
      file = File.open(temp_path, flags)
    end

    compressed = Zlib::Deflate.deflate(content, Zlib::BEST_SPEED)
```

¹³https://docs.ruby-lang.org/en/2.3.0/String.html#method-i-force_encoding

```
    file.write(compressed)
    file.close

    File.rename(temp_path, object_path)
end

def generate_temp_name
  "tmp_obj_#{(1..6).map { TEMP_CHARS.sample }.join("")}"
end
```

The first thing this method does is compute a few more pathnames. First, it builds `object_path`, the final destination path that the blob will be written to. This is done by combining the path to the `.git/objects` directory, the first two characters of the object ID, and the remaining characters. Here's an example object path:

```
/Users/jcoglan/jit/.git/objects/90/3a71ad300d5aa1ba0c0495ce9341f42e3fc7c
```

Having constructed this path, we need to write the blob to it, but we must be careful. When we write strings to a file, the string might not be written to disk all at once. That means that if some other process attempts to read the file, it might see a partially-written blob, not the complete thing. So programs often write a file out to a temporary location, and then atomically move it to its final destination. In Git, this is done as follows.

First, it generates a random filename in the same directory as the file it's planning to write, for example:

```
/Users/jcoglan/jit/.git/objects/90/tmp_obj_gNLJvt
```

Here we see some more uses of the `Pathname` interface, for example `Pathname#dirname`¹⁴. Rather than using a string manipulation to remove the filename part of the path, for example:

```
object_path.gsub(%r{/[^\/]+$}, "")
```

We can just call `object_path.dirname`, which does the right thing. The string manipulation is hard to read, it contains an assumption that path segments are separated with a forward slash, and it doesn't deal with various other edge cases. It will also be easily broken by accident, precisely because it's hard to understand and therefore hard to maintain. This is a great reason to resort to libraries for managing structured data rather than manipulating strings yourself.

Having computed the right path, it tries to open the file with the `RDWR`, `CREAT` and `EXCL` flags. These flags mirror the names from the underlying C system libraries, and they have the following effect:

- `RDWR` means the file is opened for reading and writing; the file handle we get from `File.open` can be both read from and written to.
- `CREAT` means the operating system will attempt to create the file if it does not already exist.
- `EXCL` is used in conjunction with `CREAT` and it means an error will be thrown if the file already exists; this ensures our random filename won't clobber another one if their names happen to coincide.

¹⁴<https://docs.ruby-lang.org/en/2.3.0/Pathname.html#method-i dirname>

You can find out more about the flags that `File.open` takes by reading the manual page for the system call it wraps; run `man 2 open` in your terminal.

This attempt to open the temporary file might fail, in particular it will fail if its parent directory — in this case, `.git/objects/90` — does not exist. In this event an `Errno::ENOENT` error will be raised, and we can handle that by creating the directory¹⁵ and then trying to open the file again.

Having got the file open, we compress the object's contents using `Zlib::Deflate.deflate`¹⁶, write the result to the file, and then close it. Once this is done, we can move the file to its final destination using `File.rename`¹⁷. This is a wrapper around the `rename` system call, which again you can find out more about by running `man 2 rename` from your terminal.

The characters used for constructing temporary filenames are just the lower and uppercase latin letters and the digits 0 to 9, which I've stored off in a constant within `Database`. Ruby's `Array#sample`¹⁸ method, used in our `generate_temp_name` method, selects elements from an array at random.

```
# database.rb

TEMP_CHARS = ("a".."z").to_a + ("A".."Z").to_a + ("0".."9").to_a
```

Let's run our code as it stands and check that's it's doing what we expect. We'll remove the original `.git` directory, run `ruby jit.rb init` again, then run `ruby jit.rb commit`, and it should take the four files we have so far and store them as objects.

```
$ rm -rf .git ; ruby jit.rb init ; ruby jit.rb commit
$ tree .git
.git
├── objects
│   ├── 55
│   │   └── 631245d1f70bdcdc83b07ed6c62b6d974cda62
│   ├── 90
│   │   └── 3a71ad300d5aa1ba0c0495ce9341f42e3fc7c
│   ├── e2
│   │   └── b10be08b0d00d34a1660b062d5259c240fde32
│   └── ed
│       └── b2500bfe74c64ee5983c261458305dd54542cf
└── refs
```

We can check the contents of any of these objects by piping them through our `inflate` alias¹⁹:

```
$ cat .git/objects/90/3a71ad300d5aa1ba0c0495ce9341f42e3fc7c | inflate
blob 235class Workspace
IGNORE = [".", "..", ".git"]

def initialize(pathname)
  @pathname = pathname
end
```

¹⁵<https://docs.ruby-lang.org/en/2.3.0/Dir.html#method-c-mkdir>

¹⁶<https://docs.ruby-lang.org/en/2.3.0/Zlib/Deflate.html#method-c-deflate>

¹⁷<https://docs.ruby-lang.org/en/2.3.0/File.html#method-c-rename>

¹⁸<https://docs.ruby-lang.org/en/2.3.0/Array.html#method-i-sample>

¹⁹Figure 2.4, “Creating a shell alias”

```
def list_files
  Dir.entries(@pathname) - IGNORE
end

def read_file(path)
  File.read(@pathname.join(path))
end
end
```

Indeed, checking all the files in `.git/objects` confirms that they do contain compressed representations of the files from the workspace.

3.2.2. Storing trees

Now that we've written a few blobs to the database, we can make a tree out of them. In Section 2.3.3, "Trees on disk" we saw that a tree object contains a sorted list of the entries in the tree, and each entry contains the file's mode, its name, and the object ID of the blob that holds its current contents.

Let's extend our `commit` command handler still further. Whereas before we were simply iterating over `workspace.list_files` and calling `database.store` for each one, we will now map the list of file paths to a list of `Entry` objects that contain the file path and the object ID we got by writing the blob to the database. We'll then construct a `Tree` from those entries, store it in the database, and print its ID.

```
# jit.rb

when "commit"
  root_path = Pathname.new(Dir.getwd)
  git_path  = root_path.join(".git")
  db_path   = git_path.join("objects")

  workspace = Workspace.new(root_path)
  database  = Database.new(db_path)

  entries = workspace.list_files.map do |path|
    data = workspace.read_file(path)
    blob = Blob.new(data)

    database.store(blob)

    Entry.new(path, blob.oid)
  end

  tree = Tree.new(entries)
  database.store(tree)

  puts "tree: #{tree.oid}"
```

An `Entry` is a simple structure that exists to package up the information that `Tree` needs to know about its contents: the filename, and the object ID. `Tree` will also need to know the mode of each file, but for now all our source code is in non-executable regular files, so we will hard-code the `100644` mode string that appears in the tree file.

```
# entry.rb
```

```
class Entry
  attr_reader :name, :oid

  def initialize(name, oid)
    @name = name
    @oid = oid
  end
end
```

Now we get to the core logic for storing trees. In the last section we created `Blob` objects and passed them to `Database#store` to be saved. But if you go back and look at the `store` method, it doesn't care about the `Blob` class per se. All it requires is that the object passed into it responds to the `type` and `to_s` methods, and has an `oid` property that can be set. This is why representing blobs as objects was valuable: we can implement other classes that respond to the same interface, and thereby get the database to store other kinds of data.

Here is the `Tree` class. Some parts of this probably look very cryptic at first, but we'll walk through it next.

```
# tree.rb

class Tree
  ENTRY_FORMAT = "Z*H40"
  MODE = "100644"

  attr_accessor :oid

  def initialize(entries)
    @entries = entries
  end

  def type
    "tree"
  end

  def to_s
    entries = @entries.sort_by(&:name).map do |entry|
      ["#{MODE} #{entry.name}", entry.oid].pack(ENTRY_FORMAT)
    end

    entries.join("\n")
  end
end
```

Some of this is boilerplate that you're already familiar with from the `Blob` class: we have an `oid` property, and a `type` method that returns the type tag for `Database` to write to the file. The `to_s` method is what we really care about here, as it's where the tree's on-disk representation is implemented.

This method sorts the entries by name, and then converts each one to a string using this expression:

```
["#{MODE} #{entry.name}", entry.oid].pack(ENTRY_FORMAT)
```

`MODE` is straightforward enough; it's the string `100644` that each tree entry should begin with. This is put in a string along with the entry's name, separated by a space, and this is placed in

an array along with the object ID — the other elements of each tree entry on disk. Then we call `pack` on this array, and pass in a weird-looking string constant whose value is `Z*H40`. This probably looks like gibberish, but it's actually a special-purpose language baked into Ruby.

The `Array#pack`²⁰ method takes an array of various kinds of values and returns a string that represents those values. Exactly how each value gets represented in the string is determined by the format string we pass to `pack`. Our usage here consists of two separate encoding instructions:

- `Z*`: this encodes the first string, `"#{ MODE } #{ entry.name }"`, as an arbitrary-length null-padded string, that is, it represents the string as-is with a null byte appended to the end
- `H40`: this encodes a string of forty hexadecimal digits, `entry.oid`, by packing each pair of digits into a single byte as we saw in Section 2.3.3, “Trees on disk”

Putting everything together, this generates a string for each entry consisting of the mode `100644`, a space, the filename, a null byte, and then twenty bytes for the object ID. Ruby’s `Array#pack` supports many more data encodings and is very useful for generating binary representations of values. If you wanted to, you could implement all the maths for reading pairs of digits from the object ID and turning each pair into a single byte, but `Array#pack` is so convenient that I usually reach for that first.

Now, let’s run our program with the new features we’ve added and see what it outputs.

```
$ rm -rf .git ; ruby jit.rb init ; ruby jit.rb commit
tree: fdfedfe2aeb4c3f13c273aa125bcb49cb0f5f33d
```

It’s given us the ID of a tree, which indeed exists in the database along with a slightly different set of blob objects than we saw before. We’ve added two new files — `entry.rb` and `tree.rb` — and we’ve modified a couple of existing ones too.

```
.git
└── objects
    ├── 00
    │   └── c35422185bf1dca594f699084525e8d0b8569f
    ├── 42
    │   └── f6c3e4a9eca4d4947df77e90d2a25f760913c6
    ├── 5e
    │   └── 4fb7a0afe4f2ec9768a9ddd2c476dab7fd449b
    ├── 67
    │   └── 4745262e0085d821d600abac881ac464bbf6b3
    ├── 90
    │   └── 3a71ad300d5aa1ba0c0495ce9341f42e3fc7c
    ├── e2
    │   └── b10be08b0d00d34a1660b062d5259c240fde32
    └── fd
        └── fdfedfe2aeb4c3f13c273aa125bcb49cb0f5f33d
└── refs
```

If you delete the `.git` directory again and generate a real Git commit by using Git’s `init`, `add` and `commit` commands, you’ll see all the above object IDs appearing in `.git/objects`, plus one more for the commit itself. This tells us we’re hashing all the content correctly, and serialising the tree correctly since it hashes to the same value, `7e8d277...`, using either implementation.

²⁰<https://docs.ruby-lang.org/en/2.3.0/Array.html#method-i-pack>

Finally, let's take a look inside the tree file we've created to double-check its contents.

```
$ cat .git/objects/fd/fedfe2aeb4c3f13c273aa125bcb49cb0f5f33d | inflate | hexdump -C

00000000 74 72 65 65 20 32 31 39 00 31 30 30 36 34 34 20 |tree 219.100644 |
00000010 62 6c 6f 62 2e 72 62 00 e2 b1 0b e0 8b 0d 00 d3 |blob.rb....|
00000020 4a 16 60 b0 62 d5 25 9c 24 0f de 32 31 30 30 36 |J.`.b.%.$..21006|
00000030 34 34 20 64 61 74 61 62 61 73 65 2e 72 62 00 00 |44 database.rb..|
00000040 c3 54 22 18 5b f1 dc a5 94 f6 99 08 45 25 e8 d0 |.T".[.....E%..|
00000050 b8 56 9f 31 30 30 36 34 34 20 65 6e 74 72 79 2e |.V.100644 entry.|
00000060 72 62 00 67 47 45 26 2e 00 85 d8 21 d6 00 ab ac |rb.gGE&....!....|
00000070 88 1a c4 64 bb f6 b3 31 30 30 36 34 34 20 6a 69 |...d...100644 jil|
00000080 74 2e 72 62 00 5e 4f b7 a0 af e4 f2 ec 97 68 a9 |t.rb.^0.....h.|
00000090 dd d2 c4 76 da b7 fd 44 9b 31 30 30 36 34 34 20 |...v...D.100644 |
000000a0 74 72 65 65 2e 72 62 00 42 f6 c3 e4 a9 ec a4 d4 |tree.rb.B.....|
000000b0 94 7d f7 7e 90 d2 a2 5f 76 09 13 c6 31 30 30 36 |.}~..._v...1006|
000000c0 34 34 20 77 6f 72 6b 73 70 61 63 65 2e 72 62 00 |44 workspace.rb.|
000000d0 90 3a 71 ad 30 0d 5a a1 ba 0c 04 95 ce 93 41 f4 |.:q.0.Z.....A.|
000000e0 2e 3f cd 7c |.?|||
000000e4
```

We see the header section, tree 219 followed by a null byte as expected, and then a series of entries in name order. Here's the first one, blob.rb, with object ID e2b10be...:

```
00000000 31 30 30 36 34 34 20 | 100644 |
00000010 62 6c 6f 62 2e 72 62 00 e2 b1 0b e0 8b 0d 00 d3 |blob.rb....|
00000020 4a 16 60 b0 62 d5 25 9c 24 0f de 32 |J.`.b.%.$..2|
```

It's followed immediately by database.rb, with ID 00c3542...:

```
00000020 31 30 30 36 | 1006|
00000030 34 34 20 64 61 74 61 62 61 73 65 2e 72 62 00 00 |44 database.rb..|
00000040 c3 54 22 18 5b f1 dc a5 94 f6 99 08 45 25 e8 d0 |.T".[.....E%..|
00000050 b8 56 9f |.V.||
```

And then entry.rb, with ID 6747452..., and so on.

```
00000050 31 30 30 36 34 34 20 65 6e 74 72 79 2e | 100644 entry.|
00000060 72 62 00 67 47 45 26 2e 00 85 d8 21 d6 00 ab ac |rb.gGE&....!....|
00000070 88 1a c4 64 bb f6 b3 |...d...|
```

We can check that the referenced blobs do indeed contain the contents of the named file, for example:

```
$ cat .git/objects/67/4745262e0085d821d600abac881ac464bbf6b3 | inflate

blob 110class Entry
attr_reader :name, :oid

def initialize(name, oid)
  @name = name
  @oid = oid
end
end
```

3.2.3. Storing commits

Now that we have a tree in the database, we have something we can base a commit on. Recall from Section 2.3.4, “Commits on disk” that a commit contains a pointer to a tree, some

information about the author and committer, and the commit message. This data will be much easier to serialise than it was for trees, since it's just a few bits of plain text glued together. The trickier thing this time is where to get some of the other information from, in particular the author and the message.

Git allows the author and committer for a commit to be set in a number of different ways. The most commonly used method is to store your name and email in your `.gitconfig` file, and Git will automatically use those when you run the `commit` command.

```
# ~/.gitconfig

[user]
    name = James Coglan
    email = james@jcoglan.com
```

Unfortunately, we're just starting out and we don't have a configuration system yet, so this is out of the question.

Another method of setting the author is to use the `--author` command-line option when running `git commit`. This is much simpler than parsing Git config files, but command-line argument parsing is still more complexity than I'd like to introduce at this stage. I'd like something even simpler.

Fortunately, Git does support one very easy-to-implement method of getting the author details: *environment variables*. In Section 3.1.3, “Running Jit for the first time” we met a few of the basic ingredients that make up a Unix process. In addition to these, all processes have a set of environment variables. Whereas command-line arguments are accessed via an array inside the program (`ARGV` in Ruby) and are passed explicitly as part of the command to start a process, environment variables are accessed by name and do not have to be set explicitly as part of a command. Instead, every process inherits a copy of all the environment variables of its parent process.

You can see all the environment variables set in your shell by running the `env` command²¹. You'll probably have a few that are used to control common mechanisms in the shell, like `HOME` (the path to your home directory), `LOGNAME` (your username), `PATH` (a set of directories to search for executables in), and so on. When you run a program from the shell, the new process gets a copy of all these variables, and this provides a way to implicitly pass state and configuration from one process to another.

Git uses the environment variables `GIT_AUTHOR_NAME` and `GIT_AUTHOR_EMAIL` to set the author details. Modern releases also support a distinct pair of variables called `GIT_COMMITTER_NAME` and `GIT_COMMITTER_EMAIL`, but at first there was only one pair of variables that was used to set the author, and we'll take a similar shortcut here. In Ruby, all environment variables are accessed via a global constant called `ENV`, so we can use `ENV["GIT_AUTHOR_NAME"]` to fetch the author name. All we need to do then is arrange for these variables to be set in our shell, and we can do that by adding the following to the `~/.profile` script that runs when you start a new shell:

```
# ~/.profile
```

²¹<https://manpages.ubuntu.com/manpages/bionic/en/man1/env.1posix.html>

```
export GIT_AUTHOR_NAME="James Coglan"
export GIT_AUTHOR_EMAIL="james@jcoglan.com"
```

The other piece of information we need is the commit message, and Git provides two main methods of setting this. The `commit` command has a `--message` option, allowing the message to be passed as part of the `commit` command. Again, this will involve implementing argument parsing, and it's also quite limiting, for example it makes it harder to compose long multi-line messages.

The second option is to use your text editor; Git opens the file `.git/COMMIT_EDITMSG` for you to edit, and when you close the editor Git reads the message from this file. Implementing this feature in full involves a fair bit of work, but there's something much simpler we can implement that removes a slight bit of convenience while keeping the ability for us to use files to edit the message. The very first version of Git ever committed reads the message from standard input²², meaning the message could be passed by using the shell to pipe²³ the output of one process into Git. In particular, it lets us use `cat` to dump the contents of a file and pipe that into Git:

```
$ cat ./path/to/COMMIT_EDITMSG | git commit
```

The above is called a pipeline, and we've already seen several uses of it in this book. We used it to decompress file contents using `zlib`, and to run the result through `hexdump`. When you separate two commands with the pipe character `|`, the shell runs both commands in parallel, creating two processes. Anything the first process (`cat`) writes to its standard output stream is fed into the other process's standard input stream, letting the second process ingest the output of the first.

When the first process exits, its `stdout` stream is closed, and the second process will see an end-of-file²⁴ (EOF) signal from its `stdin` stream telling it no more data is forthcoming. In Ruby, we can call `$stdin.read` to get everything written to standard input until this EOF signal is reached.

Now we know how to get all the information we need, let's translate this into code. In `jit.rb`, we'll pick up where we left off after storing the tree in the database, and gather up the other inputs we need to build the commit: the author's name and email from the environment, the current time to append to the author and committer lines, and the message from `stdin`. Then we'll build a `Commit` out of these inputs and store it.

```
# jit.rb

tree = Tree.new(entries)
database.store(tree)

name    = ENV.fetch("GIT_AUTHOR_NAME")
email   = ENV.fetch("GIT_AUTHOR_EMAIL")
author  = Author.new(name, email, Time.now)
message = $stdin.read

commit = Commit.new(tree.oid, author, message)
database.store(commit)
```

The `Author` object used here is a simple struct that packages up the name, email and time values that form the contents of the `author` and `committer` headers and implements the `to_s` method

²²<https://github.com/git/git/blob/e83c5163316f89bfbd7d9ab23ca2e25604af290/commit-tree.c#L164-L166>

²³[https://en.wikipedia.org/wiki/Pipeline_\(Unix\)](https://en.wikipedia.org/wiki/Pipeline_(Unix))

²⁴<https://en.wikipedia.org/wiki/End-of-file>

that encodes how these values should be turned into text. It uses the `Time#strftime`²⁵ method to turn the time object we got from `Time.now` into a string; `%s` generates the Unix timestamp in seconds and `%z` generates the timezone offset. This method is a wrapper for the standard C `strftime` function²⁶.

```
# author.rb

Author = Struct.new(:name, :email, :time) do
  def to_s
    timestamp = time.strftime("%s %z")
    "#{ name } <#{ email }> #{ timestamp }"
  end
end
```

The `Commit` class is another implementation of the pattern established by `Blob` and `Tree`. It creates objects that can be passed to `Database#store` where they are then serialised and written to disk. Its `to_s` method simply generates all the lines we've seen that commit objects should contain, separated by line feed characters. Passing the `Author` object into the string interpolation syntax "`author #{ @author }`" automatically invokes its `to_s` method.

```
# commit.rb

class Commit
  attr_accessor :oid

  def initialize(tree, author, message)
    @tree     = tree
    @author   = author
    @message  = message
  end

  def type
    "commit"
  end

  def to_s
    lines = [
      "tree #{ @tree }",
      "author #{ @author }",
      "committer #{ @author }",
      "",
      @message
    ]

    lines.join("\n")
  end
end
```

This code is now ready to write a commit object to the `.git/objects` database. There is one final detail we need to put in place for this to be a valid Git repository, and to allow us to write further commits that follow on from this one. Git needs a way to know what the current commit is, so that it can determine what's changed since then, and so that it can store this as the parent of the next commit. This is what the `.git/HEAD` file is for; usually when you use Git this file

²⁵<https://docs.ruby-lang.org/en/2.3.0/Time.html#method-i-strftime>

²⁶<https://manpages.ubuntu.com/manpages/bionic/en/man3/strftime.3posix.html>

contains a pointer to something in the `.git/refs` directory, but it can also contain a commit ID directly.

In `jit.rb`, after we've stored our first commit, we should store its ID off in `.git/HEAD`. Then we can print a message to the terminal to display the new commit's ID and the first line of its message.

```
# jit.rb

commit = Commit.new(tree.oid, author, message)
database.store(commit)

File.open(git_path.join("HEAD"), File::WRONLY | File::CREAT) do |file|
  file.puts(commit.oid)
end

puts "[(root-commit) #{commit.oid}] #{message.lines.first}"
exit 0
```

To mint the first proper commit of the Jit project, let's delete the `.git` directory we've built up so far, reinitialise it, and then run `ruby jit.rb commit` with a commit message sent in via a pipe.

```
$ rm -rf .git
$ ruby jit.rb init
$ cat ../COMMIT_EDITMSG | ruby jit.rb commit

[(root-commit) 9517b3c1434bfbe34a8f43415f6e707393fddd02] \
  Initial revision of "jit", the information manager from London.
```

Inspecting the `.git` directory, we can see that commit `9517b3c...` is indeed stored in there:

```
.git
├── HEAD
└── objects
    ├── 0d
    │   └── c4e0df683cf07c9c6ce644725ff06babf4c833
    ├── 29
    │   └── 4decbbc3583283df8b7364f5d2cd3b1a4a1c1b
    ├── 42
    │   └── f6c3e4a9eca4d4947df77e90d2a25f760913c6
    ├── 61
    │   └── 1e38398f238b776d30398fadcc18b74cc738a1e
    ├── 67
    │   └── 4745262e0085d821d600abac881ac464bbf6b3
    ├── 80
    │   └── 76cdc6322f4fe4abc767e3490ff2cc0e207201
    ├── 90
    │   └── 3a71ad300d5aa1ba0c0495ce9341f42e3fc7c
    ├── 95
    │   └── 17b3c1434bfbe34a8f43415f6e707393fddd02
    ├── e2
    │   └── b10be08b0d00d34a1660b062d5259c240fde32
    ├── f2
    │   └── 88702d2fa16d3cdf0035b15a9fcfc552cd88e7
    └── ff
        └── f1eeb9a60df7667516ede342300f67362be8a6
└── refs
```

And, `.git/HEAD` contains the commit's ID as required.

```
$ cat .git/HEAD  
9517b3c1434bfbe34a8f43415f6e707393fddd02
```

Let's also check that the commit file on disk contains everything it should do:

```
$ cat .git/objects/95/17b3c1434bfbe34a8f43415f6e707393fddd02 | inflate  
  
commit 764tree 294decbbc3583283df8b7364f5d2cd3b1a4a1c1b  
author James Coglan <jcoglan.com> 1512325222 +0000  
committer James Coglan <jcoglan.com> 1512325222 +0000  
  
Initial revision of "jit", the information manager from London
```

This commit records a minimal set of functionality necessary for the code to store itself as a valid Git commit. This includes writing the following object types to the database:

- Blobs of ASCII text
- Trees containing a flat list of regular files
- Commits that contain a tree pointer, author info and message

These objects are written to `., compressed using zlib.

At this stage, there is no index and no `add` command; the `commit` command simply writes everything in the working tree to the database and commits it.

This contains the correct author information and message, and points to tree 294decbb..., which we can see is another object stored in the database.

At this point, the Jit directory should act as a valid Git repository. If we run `git show`, we see the commit we just created, including its author, date, message, and the diff showing all the source code being created. Running `git log` likewise displays the one commit we've written. However, other commands don't yet work because the data model is incomplete, for example, `git status` and `git diff` rely on the index, and so if you change the source code right now you won't be able to get Git to tell you what's changed.

Our next steps will be aimed at bootstrapping the functionality as quickly as possible. We need to add support for storing commits with parents, storing nested directories and executable files, and adding the index. Once these pieces of the data model are in place, more advanced features can be added on piece by piece.

4. Making history

In the previous chapter, we built a small program capable of storing itself as a Git commit. On running `ruby jit.rb commit`, the current contents of all the source files are written to the database. We can make some changes to the code, and run `ruby jit.rb commit` again, and that will store another commit, pointing to a tree containing the new state of the source code.

We are not yet storing any relationship between the commits themselves, and so far that has not been necessary, since we only have a single commit. However, as we make further commits, having a set of snapshots of our source code in no particular order isn't tremendously useful. To be able to see the history of the project, view what changed from one commit to the next, undo sets of changes, and to form branches and merge them back together, we need to be able to put these commits in order somehow.

4.1. The parent field

In Section 2.3, “Storing objects”, we looked at the information Git stores for a single commit. Let's revisit that repository and add a second commit to it, to see what Git stores when there is already a commit in the repository.

```
$ cd git-simple-commit  
$ echo "second" > hello.txt  
$ git add .  
  
$ git commit --message "Second commit."  
[master a134034] Second commit.  
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Notice that this time, `git commit` did not print (`root-commit`) in its output, because the new commit follows on from the last one we made, rather than being the very first commit to a repository.

After making this commit, its ID is stored in `.git/refs/heads/master` just as we've seen before.

```
$ cat .git/refs/heads/master  
a1340342e31de64d169afca54d420ce7537a0614
```

Now, if we examine the object stored under this commit ID, we see a new line of information that wasn't in the first commit: a field called `parent`, whose value is an object ID.

```
$ git cat-file -p a1340342e31de64d169afca54d420ce7537a0614  
  
tree 040c6f3e807f0d433870584bc91e06b6046b955d  
parent 2fb7e6b97a594fa7f9ccb927849e95c7c70e39f5  
author James Coglan <james@jcoglan.com> 1512724583 +0000  
committer James Coglan <james@jcoglan.com> 1512724583 +0000  
  
Second commit.
```

If we check what this object ID points at, we discover it refers to the previous commit:

```
$ git cat-file -p 2fb7e6b97a594fa7f9ccb927849e95c7c70e39f5
```

```
tree 88e38705fdbd3608cddbe904b67c731f3234c45b
author James Coglan <jcoglan.com> 1511204319 +0000
committer James Coglan <jcoglan.com> 1511204319 +0000
```

First commit.

Now we know how Git represents the order that commits were made in. Each new commit made after the initial one has a field labelled `parent`, that contains the ID of the previous commit. In this manner, each commit has a link to the one that preceded it.

4.1.1. A link to the past

You may wonder, why not use the timestamps of the author or committer field to put the commits in order? To put it briefly, using these timestamps would create fundamental problems for several commands. Imagine you wanted to list the latest five commits in a project with a thousand commits in it. With these parent links, you just need to look up the latest commit from `HEAD`, then follow its parent link, then follow the parent link on that commit, and so on until you've loaded five commits. If Git used timestamps, you would need to load all the commits from the database, and then sort them by time, which is clearly a lot more work, especially since the Git database doesn't even maintain an index of which objects are of which type. Of course, we could add indexes to make this faster, but there are deeper problems with using timestamps than just this performance issue.

A similar problem would affect the `fetch` and `push` commands that synchronise databases between Git repositories. It's easy to find all the commits you need to download from a remote repository by looking up the remote's `HEAD` and then following parent links until you see a commit ID you already have. It would be more expensive to load all the commits on the remote and select those with timestamps later than your `HEAD`.

This points at a deeper problem with using time to order things. Git is a distributed system where many people can create commits locally and then synchronise with one another over the network. Those people's clocks won't necessarily be correct, so using the times they put on their commits when merging everything together would create problems. But more fundamentally, merging branches correctly requires knowing how commits are *causally connected*, that is we need to know what previous version of the project each commit was based on, not just where it occurred in time relative to other commits.

It is more robust to put an explicit representation of the idea that commit B was derived from commit A into the data model; it makes calculating what's changed on different people's branches, and merging those changes together, much more reliable. We'll explore these ideas in more detail when we implement branching and merging¹. For now, learn to think of the timestamps as metadata, just like the author name and commit message, rather than as a structural element of the data model.

4.1.2. Differences between trees

Let's compare the trees of these two commits in our example repo. The tree for the first commit, `88e3870...`, contains two entries that refer to the blobs for filenames `hello.txt` and `world.txt`.

¹Chapter 17, *Basic merging*

```
$ git cat-file -p 88e38705fdbd3608cddbe904b67c731f3234c45b  
100644 blob ce013625030ba8dba906f756967f9e9ca394464a      hello.txt  
100644 blob cc628cccd10742baea8241c5924df992b5c019f71      world.txt
```

Here's the tree for the second commit, `040c6f3`.... Its blob ID for `world.txt` is the same as in the first commit, but it has a new ID for `hello.txt`.

```
$ git cat-file -p 040c6f3e807f0d433870584bc91e06b6046b955d  
100644 blob e019be006cf33489e2d0177a3837a2384eddebc5      hello.txt  
100644 blob cc628cccd10742baea8241c5924df992b5c019f71      world.txt
```

Just to be sure, let's check the object ID for `hello.txt` and see that it points to the blob containing the new content we wrote to the file for the second commit.

```
$ git cat-file -p e019be006cf33489e2d0177a3837a2384eddebc5  
second
```

The entry for `world.txt` has not changed between the two commits, because in Git object IDs are derived from their content. We didn't change the file `world.txt` in our second commit, so it still points to the same blob as it did before. We can now see one of the efficiencies created by Git's storage model: rather than storing a complete copy of your project for every commit, it only creates new blobs when files have their content changed. Files with the same content will point to the same blob, even when those files appear in many trees across many commits. Since most commits only change a small proportion of the files in a project, this means most of the contents of your tree can be reused between commits, without wasting a lot of disk space and creating a lot more data that would need to be sent over the network.

4.2. Implementing the parent chain

The above examination of the tree structures across two commits shows that we don't need to modify our code for writing the blobs and trees to the database. We can continue to store the current state of the files (some of which will be unchanged) off as blobs, form a tree out of those and store it. Similarly, we can continue to write the new commit's ID to `.git/HEAD` as before.

The only thing that we need to change is that the `Commit` object we create must be given the current value of `.git/HEAD` and include a `parent` field in its `to_s` method if necessary. Since the logic for managing the `.git/HEAD` file is about to get more complicated, let's introduce a new abstraction and move some code from `jit.rb` into it. I'll call this new class `Refs` — it will eventually expand in scope to manage files in `.git/refs` — and it exposes a method called `update_head`, which takes an object ID and contains the logic from `jit.rb` for writing that ID to `.git/HEAD`.

```
# refs.rb  
  
class Refs  
  def initialize(pathname)  
    @pathname = pathname  
  end
```

```

def update_head(oid)
  flags = File::WRONLY | File::CREAT
  File.open(head_path, flags) { |file| file.puts(oid) }
end

private

def head_path
  @pathname.join("HEAD")
end
end

```

To this class let's add an additional method for reading the .git/HEAD file, returning nothing if the file does not exist. Calling `File.read` with a non-existent file would normally raise `Errno::ENOENT`. If the conditional on `File.exist?` fails, this method implicitly returns `nil` rather than raising an exception.

```

# refs.rb

def read_head
  if File.exist?(head_path)
    File.read(head_path).strip
  end
end

```

Now, we can begin to use this `Refs` class from `jit.rb` as part of the `commit` command logic. We'll create an instance of it along with the objects that manage the workspace and the database:

```

# jit.rb

workspace = Workspace.new(root_path)
database = Database.new(db_path)
refs = Refs.new(git_path)

```

And then we'll gather the parent object ID from `refs` as part of the information we collect when building the new commit. We pass this parent value — which may be `nil` — into the `Commit` class. At the end of the command, we include `(root-commit)` in the output if there was no parent ID to be found.

```

# jit.rb

parent = refs.read_head
name = ENV.fetch("GIT_AUTHOR_NAME")
email = ENV.fetch("GIT_AUTHOR_EMAIL")
author = Author.new(name, email, Time.now)
message = $stdin.read

commit = Commit.new(parent, tree.oid, author, message)
database.store(commit)
refs.update_head(commit.oid)

is_root = parent.nil? ? "(root-commit)" : ""
puts "[#{is_root}#{commit.oid}] #{message.lines.first}"
exit 0

```

In the `Commit` class, we need to adjust the constructor to accept this extra parameter:

```
# commit.rb

def initialize(parent, tree, author, message)
  @parent = parent
  @tree = tree
  @author = author
  @message = message
end
```

Then, we include a parent field in the output of `to_s` if the `@parent` variable has a value. Whereas before we used an array literal including all the fields when setting lines below, we now initialise it as an empty array and then push a sequence of fields into it. This makes it a bit easier to conditionally include the parent field.

```
# commit.rb

def to_s
  lines = []

  lines.push("tree #{ @tree }")
  lines.push("parent #{ @parent }") if @parent
  lines.push("author #{ @author }")
  lines.push("committer #{ @author }")
  lines.push("")
  lines.push(@message)

  lines.join("\n")
end
```

With these changes in place, we can again run `ruby jit.rb commit` with a message piped in on standard input, and a new commit will be written to the repository, with the first commit as its parent. We can verify this has been done correctly using the `git log` command:

```
$ git log --oneline
f297d6c Use HEAD to set the parent of the new commit
9517b3c Initial revision of "jit", the information manager from London
```

4.2.1. Safely updating .git/HEAD

In Section 3.2.1, “Storing blobs”, we discussed the need to avoid writing files in place. Writes to the filesystem do not necessarily happen all at once, and so if a process reads a file while another process is writing to it, the reader might see partially updated content. In the case of `.git/HEAD`, this might mean getting an empty string or half of a commit ID. But for `.git/HEAD` and other references that will appear in `.git/refs` later on, there is an additional problem, which is that the file’s content changes over time.

Files in `.git/objects` never change; because the names of object files are determined by their content, under the assumption that SHA-1 is doing its job it should never be the case that two processes attempt to write different content to the same object file. If two processes write the file at the same time, it shouldn’t matter which one wins, because they’ll both be writing the same data. All we care about with object files is that the writes appear to be atomic, that is, all existing database files are complete whenever they are read. For this purpose it’s sufficient to pick a random filename to write the data out to, and then rename this file.

This is not the case for `.git/HEAD` and other references — their whole purpose is to have stable, predictable names and change their content over time, enabling us to find the latest commit without needing to know its ID. Writes to them must still appear to be atomic, but we can no longer assume that two processes trying to write to the same file will be writing the same data. In fact, we should assume that two processes trying to change a reference at the same time is an error, because unless those processes are explicitly co-ordinating with each other, they will probably disagree about the state of the system and the final value of the reference will depend on whichever process happens to finish last.

Such *race conditions*² become even more important when data must be read from a file, then modified or transformed in some way before being written back to the file. Or, when we need to check the file's current value before deciding whether to overwrite it — we need to know that the value won't be changed by another process while we're making this decision. These problems will raise their heads when we introduce the index³, when we delete branches⁴, and when implementing the push command⁵, but for now all we want is to ensure that changes to `.git/HEAD` appear to be atomic and that two instances of Jit don't both try to change it at the same time.

We can do this by introducing a new abstraction called a `Lockfile`. This will be initialised with the path of the file we want to change, and it will attempt to open a file to write to by appending `.lock` to the original pathname. We need to pick a well-known name here rather than generating a random pathname, because the whole point is to prevent two processes from getting access to the same resource at once, and this is easier if they're both trying to access the same file.

Let's begin writing this new class in `lockfile.rb`. It defines a few custom error types and an initialiser that stores the desired pathname and calculates the path for the `.lock` file that we'll use for writing.

```
# lockfile.rb

class Lockfile
  MissingParent = Class.new(StandardError)
  NoPermission = Class.new(StandardError)
  StaleLock    = Class.new(StandardError)

  def initialize(path)
    @file_path = path
    @lock_path = path.sub_ext(".lock")

    @lock = nil
  end

# ...
end
```

Now we define this class's critical method, called `hold_for_update`. The purpose of this method is to let the caller attempt to acquire a lock for writing to the file, and to be told whether they were successful. This is done by attempting to open the `.lock` file with the `CREAT` and `EXCL` flags, which means the file will be created if it does not exist, and an error will result if

²https://en.wikipedia.org/wiki/Race_condition

³Chapter 6, *The index*

⁴Section 15.5.3, “Deleting branches”

⁵Section 29.2.2, “Updating remote refs”

it already exists. The first process to call this method will create the `.lock` file, and any other process that tries to acquire this lock while that file still exists will fail to do so.

If this `File.open` call succeeds, then we store the file handle in `@lock` so we can write to it, and we return `true`. If the file already exists, we catch the `EEXIST` error and return `false`. If the directory containing the file does not exist, we catch an `ENOENT` error and convert that into our own `MissingParent` error — this means that rather than the low-level `ENOENT` error that could have any number of causes bubbling up to the top of the program, we get an error that gives us a better idea of where the problem originated. Similarly, we convert `EACCES` errors into our own `NoPermission` error.

```
# lockfile.rb

def hold_for_update
  unless @lock
    flags = File::RDWR | File::CREAT | File::EXCL
    @lock = File.open(@lock_path, flags)
  end
  true
rescue Errno::EEXIST
  false
rescue Errno::ENOENT => error
  raise MissingParent, error.message
rescue Errno::EACCES => error
  raise NoPermission, error.message
end
```

We need two further methods for the caller to use after acquiring the lock: `write` and `commit`. The `write` method builds up data to be written to the original filename, by storing it in the `.lock` file, and `commit` sends all the accumulated data to the final destination by closing the `.lock` file and renaming it to the original pathname. It then discards the `@lock` handle so no more data can be written. Both these methods should raise an error if they are called when `@lock` does not exist, since that indicates either that the lock has been released, or that the caller never acquired it in the first place.

```
# lockfile.rb

def write(string)
  raise_on_stale_lock
  @lock.write(string)
end

def commit
  raise_on_stale_lock

  @lock.close
  File.rename(@lock_path, @file_path)
  @lock = nil
end

private

def raise_on_stale_lock
  unless @lock
    raise StaleLock, "Not holding lock on file: #{ @lock_path }"
  end
end
```

```
end
```

This class fulfills our two needs for changes to `.git/HEAD`. Changes appear to be atomic, since new data is written to a temporary file which is then renamed, and two processes cannot change the file at the same time, since the temporary file is opened such that the caller fails if it already exists.

We can now use this in the `Refs#update_head` method to prevent two instances of `Jit` both moving `.git/HEAD` at the same time. We open a `Lockfile` on `.git/HEAD`, then check whether we were successful, throwing a `LockDenied` error to prevent further execution if not. If we acquired the lock, then we write the new commit ID to the file, plus a line break character, and then commit the changes.

```
# refs.rb

LockDenied = Class.new(StandardError)

def update_head(oid)
  lockfile = Lockfile.new(head_path)

  unless lockfile.hold_for_update
    raise LockDenied, "Could not acquire lock on file: #{head_path}"
  end

  lockfile.write(oid)
  lockfile.write("\n")
  lockfile.commit
end
```

This modification to `Refs` ensures that the chain of commits is properly recorded, since it prevents `.git/HEAD` from being inconsistently updated. The `Lockfile` abstraction will find further use later, as we introduce more types of record-keeping into the repository.

4.2.2. Concurrency and the filesystem

It's worth pausing here to examine why this technique works. The ability to prevent two processes from opening a file at the same time is provided by the flags passed to `File.open`:

```
# lockfile.rb

flags = File::RDWR | File::CREAT | File::EXCL
@lock = File.open(@lock_path, flags)
```

With the above call, we are asking the operating system to open the file whose pathname is given by `@lock_path`, and we're using the `File::CREAT` and `File::EXCL` flags. This is not a Ruby-specific interface, in fact it mirrors the interface provided by C for the same task:

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>

int main()
{
  int f = open("foo.txt", O_RDWR | O_CREAT | O_EXCL);
```

```
if (f < 0) {
    printf("something went wrong: %s\n", strerror(errno));
} else {
    printf("open file descriptor: %d\n", f);
}

return 0;
}
```

Running this program twice, it succeeds in opening the file on the first run, but fails the second time because the file already exists:

```
$ ./open_file
open file descriptor: 3

$ ./open_file
something went wrong: File exists
```

Both our Ruby code and the above C program are invoking the `open` system call⁶, which supports a set of flags controlling how the file should be opened. The `O_CREAT` flag means the file will be created if absent, and in combination with that the `O_EXCL` flag means the call fails if the file already exists. The only difference here is in naming: in Ruby, we can use modules and classes as namespaces for constants, like `File::CREAT`, whereas in C all constants are defined globally and so are namespaced by prefixing them with something, in this case `O_` for ‘open’.

This mutual exclusion technique works because we are telling the operating system, in a single system call: please open this file, and create it if does not exist, but fail if it already exists. Receiving this instruction all at once means the operating system can check whether the file exists, and then either raise an error or create it, as a single atomic action from the point of view of user-space software. That means, there’s no way for the computer to behave as though one program executed an `open()` call while another program was also part-way through its own `open()` call, and thereby cause a conflict.

Consider what would happen if we performed the check for existence ourselves:

```
if File.exist?(@lock_path)
    return false
else
    @lock = File.open(@lock_path)
    return true
end
```

If we implemented `Lockfile` using this code, then it would be possible for two concurrently-running instances of this program to both run the `if File.exist?` line, see that the file does not exist, and so both go ahead and create it, possibly writing different content to it afterward. This is a race condition, and it is only by delegating the call to create the file if it’s absent to the operating system that we can ensure multiple programs don’t cause this kind of error.

4.3. Don’t overwrite objects

Before we move on to the next feature, there’s a small improvement we can make. When we first wrote the `Database` class, we made it write every object it was given — every blob, tree,

⁶<https://manpages.ubuntu.com/manpages/bionic/en/man2/open.2.html>

and commit — to disk. This was reasonable, because we were writing just enough code to make the first commit, and the database was empty leading up to that point. However, on subsequent commits, many of the objects that make up the commit will already exist, particularly blobs for files that haven’t changed since the previous commit. It’s wasteful to spend time writing these already-existing objects to disk, so we’d like to avoid it if possible.

When we first created the `Database` class in Section 3.2.1, “Storing blobs”, we implemented the `write_object` method to take an object ID and the content of the object, and it began by using the object ID to generate the pathname for the object on disk.

```
# database.rb

def write_object(oid, content)
  object_path = @pathname.join(oid[0..1], oid[2...-1])
  dirname     = object_path.dirname
  temp_path   = dirname.join(generate_temp_name)

  #
  end
```

A small addition here can save us writing the object if it already exists. All we need to do is check if the object already exists proceeding.

```
# database.rb

def write_object(oid, content)
  object_path = @pathname.join(oid[0..1], oid[2...-1])
  return if File.exist?(object_path)

  dirname     = object_path.dirname
  temp_path   = dirname.join(generate_temp_name)

  #
  . . .
```

We now have everything in place to create a chain of commits, recording the history of the project, without overwriting old data. Next, we will need to flesh out our representation of trees.

5. Growing trees

So far we've been able to make good progress, adding new files to the codebase, making changes to them and committing those changes. That's a really useful set of functionality in a very small amount of code! However, Jit is still very limited when it comes to real-world projects: it's only able to store a flat list of files in a single directory, and it has no way of storing executable files.

We didn't need these features initially in order to get the project off the ground, but adding them will get us much closer to a full implementation of Git's tree model. It will also give us more freedom over how to organise the project itself, which will certainly help as it grows.

5.1. Executable files

Right now, we're invoking Jit by running `ruby jit.rb`. This runs the executable program `ruby` that's installed on the system, passing it the filename `jit.rb`. Ruby reads this file and runs it for us. Ultimately, we'd like to do away with the need for the end user to have to know what `ruby` is or type it into their shell. Instead, it would be good if `jit.rb` could be executed all by itself, just by typing its name.

In Section 2.3, “Storing objects”, we saw that Git stores the items in a directory using a tree object, which contains an entry for each item recording its name, object ID and *mode*, a magic number with value `100644`. Let's now examine what changes when we store an executable file in a repository.

```
$ git init git-executable-file
$ cd git-executable-file

$ touch hello.txt world.txt
$ chmod +x world.txt

$ git add .
$ git commit --message "First commit."
[master (root-commit) aa004c9] First commit.
 2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 hello.txt
create mode 100755 world.txt
```

The command `chmod +x world.txt` makes the `world.txt` executable. The `chmod`¹ command changes the *mode* of a file, and the argument `+x` can be read as ‘add the *executable* permission’. Running this command means that `world.txt` can be run as a program just by typing its name. But what is this *mode* property, and why has it been hard-coded to `100644` in our trees so far?

5.1.1. File modes

We get a hint of the answer above when Git prints out what was changed in the commit:

```
create mode 100644 hello.txt
create mode 100755 world.txt
```

¹<https://manpages.ubuntu.com/manpages/bionic/en/man1/chmod.1posix.html>

The file `world.txt`, which is executable, has a different mode: `100755`. This mode is what's stored in the tree object for the commit. The argument given to `cat-file` below, `HEAD^{tree}`, is a way of asking Git for the tree of the commit referenced by `.git/HEAD`, rather than having to specify its object ID. The blobs have the same object ID because they have the same content: they're both empty.

```
$ git cat-file -p HEAD^{tree}

100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    hello.txt
100755 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    world.txt
```

A file's mode is one of the pieces of metadata that's returned by the `stat()`² system call. Let's use Ruby's `File` module to inspect our two files and check their modes:

```
>> File.stat("hello.txt").mode
=> 33188

>> File.stat("world.txt").mode
=> 33261
```

Those numbers don't look immediately interesting, but here's what happens when we display them in octal, or base 8:

```
>> File.stat("hello.txt").mode.to_s(8)
=> "100644"

>> File.stat("world.txt").mode.to_s(8)
=> "100755"
```

The numbers stored by Git are the octal representation of the file's mode. Many things that store or set file modes use octal numbers, because it makes it easier to read the information the mode represents in a compact format. These numbers actually represent a set of bits that store information about the file; the mode of `hello.txt` is `1000000110100100` in binary, which breaks down into the following groups:

Figure 5.1. Interpreting the bits of a file mode

1000	000	110	100	100
-----	-----	---	---	---
file	special	user	group	other

permissions				

The mode is sixteen bits — two bytes — in total, and it packs a lot of information into those two bytes. The first group of four bits tells us what kind of file it is; regular files have the value 1000_2 , or 8_{10} , or 10_8 .

All the other groups of three bits are sets of permission-related data; in fact the mode is mostly used to represent permissions. The first set are the *special* permissions, consisting of the *setuid*³, *setgid* and sticky bits⁴. If the *setuid* bit is set on an executable file, then any user that runs the file gains the permissions of the file's owner. Similarly, setting *setgid* means the user running the program gains the permissions of the file's group. The sticky bit controls who is allowed

²<https://manpages.ubuntu.com/manpages/bionic/en/man2/stat.2.html>

³<https://en.wikipedia.org/wiki/Setuid>

⁴https://en.wikipedia.org/wiki/Sticky_bit

to rename or delete a file. These special permissions are not that frequently used, and so these bits are usually all zero.

The next set are the *user* permissions, which in this example are set to 110_2 . These bits represent the *read*, *write* and *execute* permissions. The read and write bits are 1 while the execute bit is 0, so the file's owner is allowed to read and write the file, but not execute it.

The final two sets of bits are also read/write/execute bits, but for the *group* and *other* respectively. These describe the permissions for anybody that is not the file's owner, and they work as follows. All files belong to a group, for example on macOS if you run `ls -l` you'll see a couple of columns showing each file's owner and group — for me they are `jcoqlan staff` for most of my files. In order to read a file, either:

- the user must be the file's owner, and the file's user-read bit must be set
- the user must be a member of the file's group, and the file's group-read bit must be set
- the file's other-read bit must be set

The same pattern applies for the write and execute operations. So the *group* bits determine the permissions for members of the file's group, and the *other* (sometimes called *world*) bits determine the permissions for everyone else.

They're usually written in octal because octal digits represent three bits of information. Base-8 digits range from 0 to 7 (111_2). Each group of three bits can be read as an octal digit and versa:

Figure 5.2. Octal digit file permissions

Octal	Binary	Permissions
0	000	none
1	001	execute
2	010	write
3	011	write and execute
4	100	read
5	101	read and execute
6	110	read and write
7	111	all

Now we know that in an octal mode number like 100644_8 , the first two digits 10_8 denote a regular file, the third digit 0 means no special permissions are set, the 6 means the file's owner can read and write the file, and the two 4 digits means members of the file's group and everyone else can read it. 100755_8 is almost the same, except all users are allowed to execute the file as well.

However, useful as it is to know what these numbers mean, Git only uses a very restrictive set of them. It does not actually store arbitrary file modes, in fact it only stores whether the file is executable by the owner. Suppose we create another file and set its permissions to 655_8 before storing it in Git. These permissions mean it's readable and writable by the owner, and readable and executable by all other users.

```
$ cd git-executable-file
```

```
$ touch other.txt
```

```
$ chmod 655 other.txt  
  
$ git add .  
$ git commit --message "Second commit."  
[master 794f801] Second commit.  
 1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 other.txt
```

Git reports the file's mode as 100644_8 , and if we check what's stored in the database we see the same thing:

```
$ git cat-file -p HEAD^{tree}  
  
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 hello.txt  
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 other.txt  
100755 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 world.txt
```

So Git only really has two regular file modes: 100755_8 for executables and 100644_8 for everything else.

5.1.2. Storing executables in trees

Our aim in the next commit is to arrange things so that we can execute `jit.rb` directly without having to type `ruby` at the prompt. Right now, if we attempt to do this we get an error:

```
$ ./jit.rb  
-bash: ./jit.rb: Permission denied
```

Fixing this requires two steps. First, we need to tell the operating system how to run the file when somebody executes it. Because `jit.rb` is not a native binary program as one would get by compiling a C file, we need to tell the OS the name of another program to use to interpret the file. In our case, that means `ruby`. To tell the OS to use `ruby` to run our file, we put the line `#!/usr/bin/env ruby` at the top of it.

```
#!/usr/bin/env ruby  
  
require "fileutils"  
require "pathname"  
# ...
```

This line is called a *shebang*⁵, and it requires an absolute path to an executable on the system. We can't just type `#!ruby`, we need to say exactly where `ruby` is in the filesystem, for example `#!/usr/bin/ruby`. But, Ruby can be installed in many different places on different machines. For example, I have a lot of different versions installed on my laptop and the one I'm currently using is at `/opt/rubies/ruby-2.4.3/bin/ruby`. That's what the `/usr/bin/env` part is for: `env` is a program that just runs whatever command it's followed by in the current environment, and it has a predictable location on most systems. That's why you'll often see shebangs beginning with `/usr/bin/env`: it means they can put the name of some other program after it and the operating system will run that program wherever the current user has it installed.

The second step is to add the executable permission to the file for all users, using `chmod`:

```
$ chmod +x jit.rb
```

⁵[https://en.wikipedia.org/wiki/Shebang_\(Unix\)](https://en.wikipedia.org/wiki/Shebang_(Unix))

We can now execute the file `jit.rb` directly. The operating system will run it, and we'll now see an error coming from Jit itself because we're running it without naming a command:

```
$ ./jit.rb
jit: ' ' is not a jit command.
```

Now we can begin adding code to store `jit.rb` as an executable entry in the tree of the next commit. When we're adding the files as blobs to the database, we need to detect whether they're executable, and store them with mode `100755` instead of `100644`.

We'll begin by grabbing the file stat of each file returned by the `workspace`, and passing that stat value into the `Entry` objects we build.

```
# jit.rb

when "commit"
# ...

entries = workspace.list_files.map do |path|
  data = workspace.read_file(path)
  blob = Blob.new(data)

  database.store(blob)

  stat = workspace.stat_file(path)
  Entry.new(path, blob.oid, stat)
end
```

The `stat_file` method lives in `Workspace`, and it just calls Ruby's `File.stat` method with the given pathname.

```
# workspace.rb

def stat_file(path)
  File.stat(@pathname.join(path))
end
```

We're passing this `stat` value into the `Entry` class, so we need to adjust its `initialize` method to accept that value and store it. Let's also add a `mode` method, which will return either `100644` or `100755` based on whether the `stat` says the entry is for an executable file.

```
# entry.rb

class Entry
  attr_reader :name, :oid

  REGULAR_MODE    = "100644"
  EXECUTABLE_MODE = "100755"

  def initialize(name, oid, stat)
    @name = name
    @oid  = oid
    @stat = stat
  end

  def mode
    @stat.executable? ? EXECUTABLE_MODE : REGULAR_MODE
  end
end
```

```
    end  
end
```

The last change we need to make is to the `Tree` class. Previously, its `to_s` method was using the hard-coded value `100644` as the mode for all entries. Now, given the `Entry#mode` method we just added, we can ask the entries for their mode instead of hard-coding it.

```
# tree.rb  
  
def to_s  
  entries = @entries.sort_by(&:name).map do |entry|  
    ["#{entry.mode} #{entry.name}", entry.oid].pack(ENTRY_FORMAT)  
  end  
  
  entries.join("")  
end
```

That's it — a few small adjustments and we're now able to set modes dynamically based on the file's stat, rather than hard-coding everything to `100644`, and we've learned what that magic number means in the process.

5.2. Nested trees

Having executable files is nice, but we're still really constrained in how we organise the project because we're only able to store a single flat directory of files. Real-world projects use directories to group related files together, and directories can be nested inside one another.

The Jit source tree currently looks like this:

Figure 5.3. Jit project file layout

```
.  
├── LICENSE.txt  
├── author.rb  
├── blob.rb  
├── commit.rb  
├── database.rb  
├── entry.rb  
├── jit.rb  
├── lockfile.rb  
├── refs.rb  
└── tree.rb  
└── workspace.rb
```

Let's try to make one small change to this layout. Ruby projects typically store their executables in a subdirectory called `bin`⁶, so let's rename the file `jit.rb` to `bin/jit`. When we turn Jit into a package and install it in future, this will mean users can just type `jit` in their terminal to activate it.

```
$ mkdir bin  
$ mv jit.rb bin/jit
```

The source code tree now looks like this:

⁶Although `bin` is short for *binary*, Ruby executables are not truly binary files. True binaries are files containing machine code that can directly executed by the operating system and hardware, and don't require interpretation by some other program. However, for historical reasons the terms *executable* and *binary* are often used interchangeably, and this is just the convention Ruby has adopted.

Figure 5.4. Project layout with jit in a directory

```
.  
├── LICENSE.txt  
├── author.rb  
├── bin  
│   └── jit  
├── blob.rb  
├── commit.rb  
├── database.rb  
├── entry.rb  
├── lockfile.rb  
├── refs.rb  
└── tree.rb  
└── workspace.rb
```

Jit can now be executed by typing `./bin/jit` in the project directory. To run it by just typing `jit` from any directory, we can add the `bin` directory to `PATH`, the environment variable that tells the shell where to find executable programs:

```
$ export PATH="$PWD/bin:$PATH"
```

Putting `$PWD/bin`, i.e. `bin` within the current working directory, at the front of `PATH` means the shell will look there first when you type commands in. You can now just type `jit` to run Jit. It's possible to make it start up a little faster by also setting the following variable:

```
$ export RUBYOPT="--disable gems"
```

This turns off Ruby's ability to load third-party packages, which Jit doesn't need, and this speeds up the `require` function. Only set this in shells where you want to run Jit; if you set it in all your shells then any other Ruby projects you have may stop working.

The task we have now is to replace our code that builds a single `Tree` from a flat list of `Entry` objects with a routine for building a recursively nested structure composed of `Tree` objects that can contain both further `Tree` values as well as `Entry` objects. First, let's examine how Git stores these values.

5.2.1. Recursive trees in Git

As we've done before, let's run a small example and take a look at what Git has stored as a result. In the following snippet, we create a project containing a single file, nested inside a couple of directories: `a/b/c.txt`. We then store it using `git add` and `commit` the result.

```
$ git init git-nested-tree  
$ cd git-nested-tree  
  
$ mkdir -p a/b  
$ touch a/b/c.txt  
  
$ git add .  
$ git commit --message "First commit."
```

Now, let's use `cat-file` to observe what Git has stored as the tree for this commit.

```
$ git cat-file -p HEAD^{tree}
```

```
040000 tree c4a644afb090a8303bdb28306a2f803017551f25      a
```

Where before we've seen entries beginning with `100644 blob`, we now see `040000 tree`. The object ID and the name of the entry (`a`) are still present. Let's check what's literally stored on disk for this object:

```
$ cat .git/objects/48/92032aa62c84b74cc28b74b70f4c6d973ea9f1 | inflate | hexdump -C
00000000  74 72 65 65 20 32 38 00  34 30 30 30 30 20 61 00  |tree 28.40000 a.| 
00000010  c4 a6 44 af b0 90 a8 30  3b db 28 30 6a 2f 80 30  |..D....0;.(0j/.0|
00000020  17 55 1f 25 0a                               | .U.%.| 
00000025
```

As we've seen before, the one entry in this tree is the file mode (`40000`), followed by the name (`a`) and the object ID (`c4 a6 44 ...`). Note that although `cat-file` prints a leading zero in `040000`, that zero is not present in the tree as stored on disk.

This file mode is again an incomplete picture of what's in the filesystem. If we check the mode of the directory `a` ourselves, we see it's `40755`.

```
>> File.stat("bin").mode.to_s(8)
=> "40755"
```

Directories do indeed have the file type bits set to 4 (100_2 , compared to 1000_2 for regular files), and their permissions are set to 755_8 , which means they're executable⁷. However, Git ignores all the permission bits and just stores 40000_8 to indicate it's a directory, and nothing more.

If we inspect the object the `a` directory entry points at, we see another tree containing the `b` directory.

```
$ git cat-file -p c4a644afb090a8303bdb28306a2f803017551f25
040000 tree 1721a7a91e87f5413c842a9c5ce73f674459e92b      b
```

Inspecting the `b` object in turn yields a final tree containing a regular file, `c.txt`.

```
$ git cat-file -p 1721a7a91e87f5413c842a9c5ce73f674459e92b
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391      c.txt
```

So we have seen that Git represents trees of files in quite a direct way. Just as regular files are listed using an object ID that points to a blob, subdirectories are listed using an object ID that refers to another tree. The structure of the objects in the database mirrors how we think of them in the filesystem hierarchy.

A very useful property arises from storing trees this way. When we want to compare the trees of two commits, for example to generate a diff, we typically need to recursively examine each tree object we find, right down to the individual blobs. But remember that every object in the database has an ID that's the SHA-1 hash of its contents, and the contents of a tree is just a list of the hashes and names of its entries. This means that if we see two tree entries with the same ID, we know their contents are exactly the same, and we can skip over them entirely. On big projects with thousands of files and millions of lines of code, this is a huge performance win; it

⁷To read a file, you must have the *read* permission on the file itself and you must also have the *execute* permission on all directories that lead to the file. ‘Executing’ a directory means traversing it to access the files inside. ‘Reading’ a directory means listing its contents.

also helps reduce transfer of redundant data over the network, and it makes easy to detect that a tree has been moved without changing the files inside it.

This method of storing a tree of information where each tree is labelled with the hash of its children is called a Merkle tree⁸, and this same data structure sits behind other distributed consensus protocols such as Bitcoin.

5.2.2. Building a Merkle tree

Let's return to the code in the `commit` command that reads the files from disk, stores them as blobs, creates entries from these blobs, and stores a tree containing said entries. The last change we made to this was to add the `stat` information to the `Entry` objects.

```
# bin/jit

entries = workspace.list_files.map do |path|
  data = workspace.read_file(path)
  blob = Blob.new(data)

  database.store(blob)

  stat = workspace.stat_file(path)
  Entry.new(path, blob.oid, stat)
end

tree = Tree.new(entries)
database.store(tree)
```

Two related changes must be made here. First, the `Workspace#list_files` method only reads the top-level directory to get the project's filenames; it does not recurse into subdirectories. It will need to do so in order to find everything in the project. Second, the `Tree` we're building is flat — it does not contain further trees inside. It must be converted into a recursive structure that we can save to the database in the required way.

Let's deal with `Workspace` first. Its `list_files` method is currently very simple; it calls `Dir.entries` to list the contents of the root directory.

```
# workspace.rb

def list_files
  Dir.entries(@pathname) - IGNORE
end
```

We need to make this method read the filesystem recursively. It will now take an argument called `dir` that defaults to the root pathname of the `workspace`. It will get the list of filenames from that directory as before, but now it will map over them and check whether each filename points to a directory. If it does, then we recurse into `list_files` again with this filename, otherwise we just return the path to the file, relative to the root of the `workspace`.

```
# workspace.rb

def list_files(dir = @pathname)
```

⁸https://en.wikipedia.org/wiki/Merkle_tree

```

filenames = Dir.entries(dir) - IGNORE

filenames.flat_map do |name|
  path = dir.join(name)

  if File.directory?(path)
    list_files(path)
  else
    path.relative_path_from(@pathname)
  end
end
end

```

`flat_map` is like `map` in that it converts one array to another by invoking the block with each element. The difference is that if the block returns an array, these inner arrays are flattened, for example:

```

>> ["hello", "world"].map { |word| word.chars }
=> [["h", "e", "l", "l", "o"], ["w", "o", "r", "l", "d"]]

>> ["hello", "world"].flat_map { |word| word.chars }
=> ["h", "e", "l", "l", "o", "w", "o", "r", "l", "d"]

```

This means that the arrays returned by recursive calls to `list_files` are joined together, so that the end result is one flat list of all the filenames in the project.

Having made this change to `Workspace`, Jit will still run without throwing any errors. The *structure* of the data returned by `workspace#list_files` has not changed, it is still a flat list of filenames:

```

>> workspace = Workspace.new(Pathname.new(Dir.getwd))

>> workspace.list_files
=> [Pathname("author.rb"), Pathname("bin/jit"), Pathname("blob.rb"),
  Pathname("commit.rb"), Pathname("database.rb"), Pathname("entry.rb"),
  Pathname("lockfile.rb"), Pathname("refs.rb"), Pathname("tree.rb"),
  Pathname("workspace.rb")]

```

The `commit` command could quite happily take this list of paths, read and stat the files they point to, save them as blobs, construct a list of tree entries and save the resulting tree. However, that tree would contain an entry for a blob labelled `bin/jit`, which is not right. It should contain a *tree* labelled `bin`, which tree then contains a blob labelled `jit`. We need to change how the tree is built.

In the `commit` command code, we can make a minor adjustment. We continue to process the list of paths to create blobs and `Entry` objects as before. But after doing that, rather than calling `Tree.new` and storing this single tree to the database, we'll build a nested `Tree` from the entries array, and then traverse the tree, finding every subtree inside it and storing each one in the database.

```

# bin/jit

entries = workspace.list_files.map do |path|
  #
end

```

```
root = Tree.build(entries)
root.traverse { |tree| database.store(tree) }
```

These new `Tree.build` and `Tree#traverse` methods are where the real work happens. We need to take the flat array of `Entry` values and build a nested set of `Tree` objects that reflect the structure we want to see in the database. The code as it stands just calls `Tree.new(entries)`, producing a tree containing this list:

Figure 5.5. Tree containing a flat list of entries

```
Tree.new([
  Entry.new("LICENSE.txt", "f288702..."),
  Entry.new("author.rb", "611e383..."),
  Entry.new("bin/jit", "7f6a2e0..."),
  Entry.new("blob.rb", "e2b10be..."),
  Entry.new("commit.rb", "ea1fc70..."),
  Entry.new("database.rb", "e37167c..."),
  Entry.new("entry.rb", "b099b1f..."),
  Entry.new("lockfile.rb", "72ea461..."),
  Entry.new("refs.rb", "c383169..."),
  Entry.new("tree.rb", "c94f775..."),
  Entry.new("workspace.rb", "b7a4c17...")
])
```

What we'd like instead is to call `Tree.build(entries)` and end up with a nested structure, where each filename in the tree is mapped to either an `Entry` or another `Tree`:

Figure 5.6. Tree containing a nested tree

```
Tree.new({
  "LICENSE.txt" => Entry.new("LICENSE.txt", "f288702..."),
  "author.rb"    => Entry.new("author.rb", "611e383..."),
  "bin"          => Tree.new({
    "jit"        => Entry.new("bin/jit", "7f6a2e0...")
  }),
  "blob.rb"      => Entry.new("blob.rb", "e2b10be..."),
  "commit.rb"    => Entry.new("commit.rb", "ea1fc70..."),
  "database.rb"  => Entry.new("database.rb", "e37167c..."),
  "entry.rb"     => Entry.new("entry.rb", "b099b1f..."),
  "lockfile.rb"  => Entry.new("lockfile.rb", "72ea461..."),
  "refs.rb"      => Entry.new("refs.rb", "c383169..."),
  "tree.rb"      => Entry.new("tree.rb", "c94f775..."),
  "workspace.rb" => Entry.new("workspace.rb", "b7a4c17...")
})
```

To accomplish this, we begin by defining the `Tree.build` method. Writing `def self.build` rather than `def build` means this is a *class method*; rather than being called on instances of the `Tree` class, it's called on the `Tree` class itself, and its job is to construct a set of `Tree` objects that reflect the `entries` array.

It does this by creating a new empty `Tree`, then iterating over the `entries`, adding each one to the tree in turn.

```
# tree.rb

def self.build(entries)
  entries.sort_by! { |entry| entry.name.to_s }
```

```
root = Tree.new

entries.each do |entry|
  root.add_entry(entry.parent_directories, entry)
end

root
end
```

The `Entry#parent_directories` method returns all the parent directories of the entry's name in descending order. Since `Workspace#list_files` now returns `Pathname` values, we can use `Pathname#descend`⁹ to generate the list of parent paths.

```
# entry.rb

def parentDirectories
  @name.descend.to_a[0...-2]
end
```

`Pathname#descend` yields the names of the directories leading to the given path; the subscript `[0...-2]` means all but the last item, the entry's path itself, are returned. For example:

```
>> Pathname.new("bin/nested/jit").descend.to_a[0...-2]
=> [Pathname("bin"), Pathname("bin/nested")]

>> Pathname.new("entry.rb").descend.to_a[0...-2]
=> []
```

The loop in `Tree.build` therefore results in the following sequence of calls to `Tree#add_entry`:

```
root.add_entry([], Entry(name="LICENSE.txt"))
root.add_entry([], Entry(name="author.rb"))
root.add_entry(["bin"], Entry(name="bin/jit"))
root.add_entry([], Entry(name="blob.rb"))
# etc.
```

Note that the full list of entries is sorted before building the tree from them:

```
entries.sort_by! { |entry| entry.name.to_s }
```

Git requires that tree objects have their entries sorted in a certain way. Although in the filesystem the items in a directory are not necessarily returned in alphabetical order, it's important for Git to sort them so it can tell if two trees are the same. Given two trees containing the same entries, we would like them to be given the same object ID, but this can only happen if they serialise to exactly the same string. Forcing a sort order on the entries means that a tree will only change its ID if something meaningful changes in its entries.

We're currently sorting the entries by name in `Tree#to_s`, but that is no longer suitable. To see why, let's observe what happens when we store a directory whose name is a prefix of the name of a file in the same tree.

```
$ git init git-tree-sorting
$ cd git-tree-sorting

$ mkdir foo
```

⁹<https://docs.ruby-lang.org/en/2.3.0/Pathname.html#method-i-descend>

```
$ touch foo/bar.txt foo.txt  
$ git add .  
$ git commit --message "First commit."
```

If we print the tree of this commit, we see that the tree `foo` comes after the blob `foo.txt`.

```
$ git cat-file -p HEAD^{tree}  
  
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391      foo.txt  
040000 tree 91c9988a7a7250917d0bdea6b89e07a5cf0b3a81      foo
```

That's odd: most languages would sort these names the other way round:

```
>> ["foo.txt", "foo"].sort  
=> ["foo", "foo.txt"]
```

The reason this happens is because Git actually sorts the file list for the entire project before building the tree, rather than sorting entries within trees themselves. That is, it's starting with a list containing `foo/bar.txt` and `foo.txt`, and here `foo.txt` sorts first:

```
>> ["foo/bar.txt", "foo.txt"].sort  
=> ["foo.txt", "foo/bar.txt"]
```

Iterating over this list of files in sorted order will result in these calls to `Tree#add_entry`:

```
root.add_entry([], Entry(name="foo.txt"))  
root.add_entry(["foo"], Entry(name="foo/bar.txt"))
```

`foo.txt` is added to the root tree first, and then the subtree `foo` is created, and they remain in that order when serialised. The only quirk here is that each entry's name field is a `Pathname` object, and Ruby sorts those differently to plain strings:

```
>> ["foo/bar.txt", "foo.txt"].map { |s| Pathname.new(s) }.sort  
=> [Pathname("foo/bar.txt"), Pathname("foo.txt")]
```

That means we need to turn the pathnames back into strings to get the sorting we want, hence sorting by `entry.name.to_s`.

`Tree#add_entry` is what will insert the entry at the right point in the tree. To make it work, we need to change how `Tree` stores its internal state. Rather than just storing the list of entries it's given, `Tree.new` will now take no arguments, and will set up an empty hash to store entries that are added to it. Ruby hashes iterate over their values in the order in which the keys were first inserted, so as long as we call `add_entry` in the order of the sorted list of entries, the tree will serialise correctly.

```
# tree.rb  
  
def initialize  
  @entries = {}  
end
```

Now we come to the `add_entry` method itself. This takes an array of `Pathname` values called `parents`, and an `Entry` object, and inserts the `Entry` at the right point in the tree. If `parents` is empty, that means the `Entry` belongs in the current `Tree`, and it is simply added to the `@entries` hash. But, if `parents` is not empty, then we need to create a subtree. We take the first element

of parents and create a new Tree with that name, and then call `add_entry` on this new Tree with the same arguments, excluding the first element of `parents`.

```
# tree.rb

def add_entry(parents, entry)
  if parents.empty?
    @entries[entry.basename] = entry
  else
    tree = @entries[parents.first.basename] ||= Tree.new
    tree.add_entry(parents.drop(1), entry)
  end
end
```

This method uses `Pathname#basename`¹⁰, to access the last component of each parent directory path, and to access the filenames of the `Entry` objects themselves. The `Entry` class simply delegates this method to its `@name` property:

```
# entry.rb

def basename
  @name.basename
end
```

For example, the call `tree.add_entry([], Entry(name="author.rb"))` results in this happening:

```
@entries["author.rb"] = entry
```

But, when we add an entry like `bin/jit`, we call `tree.add_entry(["bin"], Entry(name="bin/jit"))`, the result of which is:

```
tree = @entries["bin"] ||= Tree.new
tree.add_entry([], entry)
```

The guarded assignment `||=` means that nothing happens if `@entries["bin"]` already exists, so we don't overwrite any existing partially-completed subtrees.

This takes care of *building* the tree; the call to `Tree.build(entries)` will now return a nested structure of `Tree` and `Entry` objects. Now we need to deal with *storing* the tree, which is triggered by the line

```
root.traverse { |tree| database.store(tree) }
```

The `Tree#traverse` method must iterate over the entire structure and yield every `Tree` object it finds, so that those trees can be saved to the database. The order in which it yields these trees is important: in order to save one tree, we need to know the object IDs of all its entries. That means any subtrees must be saved before their parent, so that they gain an ID that becomes part of the parent tree's data. Therefore, `Tree#traverse` must yield the deepest subtrees first, and the root tree last of all.

Here's the implementation of `Tree#traverse`. We iterate over the `@entries` in the tree, and if the entry is a `Tree` then we recursively call `traverse` on it, passing the block along. After this

¹⁰<https://docs.ruby-lang.org/en/2.3.0/Pathname.html#method-i basename>

iteration is complete, we yield the Tree that traverse was called on, that is we only visit a tree after all its subtrees have been visited.

```
# tree.rb

def traverse(&block)
  @entries.each do |name, entry|
    entry.traverse(&block) if entry.is_a?(Tree)
  end
  block.call(self)
end
```

A small adjustment is also needed to Tree#to_s. We previously stored @entries as an array and used Entry#name to populate the serialised entry. But now, @entries is a hash whose keys are the filenames of each Entry, while Entry#name contains the entry's complete path. So, we should use the hash keys to fill out the names in the serialised entries instead.

```
# tree.rb

def to_s
  entries = @entries.map do |name, entry|
    ["#{entry.mode} #{name}", entry.oid].pack(ENTRY_FORMAT)
  end

  entries.join("\n")
end
```

This method reminds us that anything that can appear in a Tree requires a mode method. Let's add one to tree that returns the requisite 40000 string.

```
# tree.rb

def mode
  Entry::DIRECTORY_MODE
end
```

I've put the exact mode value next to the other ones used for blobs in entry.rb.

```
# entry.rb

REGULAR_MODE     = "100644"
EXECUTABLE_MODE = "100755"
DIRECTORY_MODE   = "40000"
```

Now when we run jit commit, the tree is written out as expected:

```
$ git cat-file -p HEAD^{tree}

100644 blob f288702d2fa16d3cdf0035b15a9fcbe552cd88e7      LICENSE.txt
100644 blob 611e38398f238b776d30398fadcb18b74cc738a1e      author.rb
040000 tree d1c53ec61d337b9ee32e2025d8757ca387dae0e7      bin
100644 blob e2b10be08b0d00d34a1660b062d5259c240fde32      blob.rb
100644 blob ea1fc7038f704606989f8b69fdbbb0675fe6d9a8      commit.rb
100644 blob e37167cd77a8bcd5339b3d93b3113afa01d19dfd      database.rb
100644 blob 14899940c8aa6a748f7712071aa3af796d1ffbd2      entry.rb
100644 blob 72ea461f8bbb8f9033840d53bf782a8331f193a1      lockfile.rb
100644 blob c383169b22f51388fa08baeea493839d6d5f4950      refs.rb
100644 blob 8e7fef3b1e3210727d7f11be77605706b4c71ce3      tree.rb
```

```
100644 blob 2c1d863dc421e6b524c9d2e3ceee123a8e4c886f    workspace.rb
$ git cat-file -p d1c53ec61d337b9ee32e2025d8757ca387dae0e7
100755 blob 5bc41f7b9a07188b71dfbb124e5433422821750e    jit
```

5.2.3. Flat or nested?

The implementation presented above may strike some readers as odd. Why do we read from the nested directories of the filesystem, turn the result into a flat list of entries, then turn that list back into a tree structure to store it? Would it not be simpler and more direct to translate the filesystem structure directly into tree objects in the database?

As it turns out, it's hard to do this without coupling the `Tree`, `Blob` and `Workspace` classes closer together. For example, we could have `Workspace#list_files` return a nested representation of the files on disk, something like:

```
{
  "author.rb" => nil,
  "bin" => {
    "jit" => nil
  },
  "blob.rb" => nil,
  "commit.rb" => nil,
  # etc.
}
```

One can imagine iterating over this structure, saving a blob for each regular file, and saving a tree for each item with children once all its children have themselves been saved. But now the processes of saving blobs and saving trees have been intermingled, whereas they are currently separate: first we save off all the blobs, then we build a tree out of the results. This doesn't seem too bad, but it would make the code slightly harder to follow and maintain, and would also require the caller to know how to reconstruct system pathnames from the filenames in this nested structure.

We'd also have to pick somewhere to put the recursive function necessary to traverse the structure. Should it live in `Tree`, so that `Tree` now has to read data from `Workspace` and create `Blob` objects? Should it live in `Workspace` so that class now needs to create `Tree` and `Blob` objects and save them using `Database`? Maybe we need to turn the `commit` command code into a class to glue all these currently-separate concepts together. It's not clear at this stage, but I do know I'd like to keep all these classes separate since they will need to be used independently of each other later, and it's generally good practice to keep the number of connections between system components to a minimum.

Then there's the question of what this structure should actually contain. Above I've represented directories as nested hashes, and regular files as nothing (`nil`). However, checking whether a file is a directory in order to generate this structure requires calling `stat()`, and storing trees also requires file stat information, so rather than statting every file twice, we could return the `stat` objects as part of the structure:

```
{
  "author.rb" => File::Stat(...),
  "bin" => {
```

```
"jit" => File::Stat(...)  
},  
"blob.rb" => File::Stat(...),  
"commit.rb" => File::Stat(...),  
# etc.  
}
```

But now we're creating new coupling between `Workspace` and database objects: `Workspace` is returning data that's tailored to what `Tree` wants the information for. Maybe that's fine, but creating functionality like this usually means one ends up with many functions that do similar things but return their results in slightly different ways.

Maybe, instead of returning a single complete structure from `Workspace#list_files`, we could give it an interface for exploring the filesystem and saving blobs and trees as we go. Again, this makes things more complicated; either `Workspace` itself would need to know to yield the deepest directories first so that tree storage works correctly, or its interface would need to become more complicated so that `Tree` or some other caller could drive the iteration, effectively reproducing the work that `Workspace#list_files` and `Tree#traverse` already do.

None of these options seem particularly appealing, but I would encourage you to try to implement them yourself to see what trade-offs you need to make. The design I've gone with keeps `Workspace`, `Database`, `Tree` and `Blob` relatively separate from each other, so although things could be written to run slightly more efficiently, this would probably make the code more complex. Even if the project grows to thousands of files, I'm not worried about holding and processing that volume of entries in memory.

But, I'm also using a little knowledge of what's coming next. After we're done with trees, we'll be working on adding the `add` command and the `index`. Whereas our current `commit` command stores all the blobs and trees and creates a commit in the database, in Git the `add` command reads the working tree, stores blobs in the database and adds them to the index, and the `commit` command reads the index to create trees. The index is stored as a flat list of paths, so to minimise the amount of change necessary I've deliberately designed the interface between `Workspace` and `Tree` to be close to what will happen when the index is introduced. This is also the reason I don't want `Tree` and `Workspace` coupled together: we will shortly change things so that creating a commit does not involve reading the working tree at all.

5.3. Reorganising the project

To round out this chapter, let's take advantage of our new capability to store arbitrarily nested trees. Having all the source files at the root of the project will not scale very well and it's not how Ruby projects are typically organised. Usually, source code is stored in the `lib` directory, and each class is stored in a file of the same name under that. So, `Entry` is kept in `lib/entry.rb`. Ruby does not require classes to be layed out on disk like this, it's just a convention that makes things easier to find.

It's also common practice to use classes and modules and namespaces to group related functionality together. For example, the `Blob`, `Tree`, `Commit` and `Author` classes are all related to representing information for storage in the `.git/objects` database, and so I'd like to namespace them under the `Database` class. Therefore, `Blob` becomes `Database::Blob`, it's stored in `lib/database/blob.rb` and its source code now looks like this:

```
class Database
  class Blob

    attr_accessor :oid

    def initialize(data)
      @data = data
    end

    #
    #

  end
end
```

Classes in the same namespace don't have to use the full namespace to refer to each other, for example, code inside Database::Tree can just use the expression Tree to refer to itself. Code outside the namespace needs adjusting, for example bin/jit now needs to call Database::Tree.build(entries).

After making these changes, the project tree looks like this and we can store it as another commit.

Figure 5.7. Project organised into directories

```
.
├── LICENSE.txt
├── bin
│   └── jit
└── lib
    ├── database
    │   ├── author.rb
    │   ├── blob.rb
    │   ├── commit.rb
    │   └── tree.rb
    ├── database.rb
    ├── entry.rb
    ├── lockfile.rb
    ├── refs.rb
    └── workspace.rb
```

This completes our work on storing Git trees and sets us up with a good organisation to flesh out the project further.

6. The index

Our database implementation is now sophisticated enough to support storing arbitrary trees and executable files, and we could quite happily keep writing commits using this infrastructure. However, as the project grows, some problems will arise.

For one, performance will tend to get worse over time. Our `commit` command currently reads and hashes every file in the project in order to store blobs, and as we add more code, that process will take longer. It's especially wasteful because in most commits on large projects, most of the files in the project will not have changed, so no new blobs will be stored as a result of reading them. We only need to know the existing object IDs of the blobs in order to build the trees for the new commit.

Performance is also a concern for the `status` and `diff` commands. While working we often want to know what we've changed since the last commit. Directly comparing the trees and blobs in the database with the files in the working tree again requires reading every file in the project, unless we have some other data source that we can use to avoid doing some of this work.

Finally, and probably most importantly, it becomes cumbersome to use. You'll notice that we do not currently have an `add` command; the `commit` command stores the current state of all the files. It would be nice to have finer control over what goes into the next commit, so that we can decide to store a chunk of work as multiple small commits, rather than one big one containing many logical changes.

The `add` command and the `index` solve these problems by providing a cache of all the blobs representing the current state of the project. It contains the object IDs of the blobs corresponding to the current files, which is used to build the next commit. It also caches a lot of information from the filesystem, such as the size of files and the time at which they were last changed, which speeds up operations that need to detect changes in the working tree.

6.1. The add command

In Git, the `commit` command does not commit whatever we have changed automatically. To tell it which changes we want to include in the next commit, we use the `add` command, which adds files to the *index* or *staging area*. We can see this by using the `status` command to inspect the current state.

Let's create a new repository and create an empty file in it, and see what `status` reports after that.

```
$ git init git-add-status
$ cd git-add-status

$ touch file.txt
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

```
file.txt  
nothing added to commit but untracked files present (use "git add" to track)
```

Git reports that `file.txt` is an *untracked file*. That means it's not in the latest commit, and it's not in the index, and therefore won't be included in the next commit. It helpfully tells us that to get `file.txt` into the repository, we need to run `git add`. Let's do that and see what difference it makes:

```
$ git add file.txt  
$ git status  
On branch master  
  
Initial commit  
  
Changes to be committed:  
(use "git rm --cached <file>..." to unstage)  
  
new file:   file.txt
```

The file is now listed under *changes to be committed*. That means it's in the index, and the indexed content differs from `file.txt` in the latest commit¹.

`file.txt` is currently empty. Let's add some content to it, and see how that changes the status:

```
$ echo "new content" > file.txt  
$ git status  
On branch master  
  
Initial commit  
  
Changes to be committed:  
(use "git rm --cached <file>..." to unstage)  
  
new file:   file.txt  
  
Changes not staged for commit:  
(use "git add <file>..." to update what will be committed)  
(use "git checkout -- <file>..." to discard changes in working directory)  
  
modified:   file.txt
```

The file is now listed in two places: *changes to be committed*, and *changes not staged for commit*. The version to be committed is the one that's just an empty file: the version of the file as it was when we ran `git add file.txt`. The changes not staged are everything we've added since running `git add`. When a file in the working tree differs from what's in the index, it is listed here until we add the file to update it in the index.

6.2. Inspecting .git/index

Let's have a look at what's stored in the index so that we understand what the `status` command is telling us. What does it mean for changes to be *staged*? We can run our old friend `hexdump` to have a look at what's been stored in `.git/index`:

¹This is a new repository, so there aren't any commits, but if we were working in a repository where `.git/HEAD` existed it would mean the file either doesn't exist in that commit, or the indexed content differs from that commit.

```
$ hexdump -C .git/index

00000000  44 49 52 43 00 00 00 02  00 00 00 01 5a 4f 7e c1  |DIRC.....Z0~.|
00000010  00 00 00 00 5a 4f 7e c0  00 00 00 00 00 00 29  |....Z0~.....)|
00000020  00 00 00 52 00 00 81 a4  00 00 03 e8 00 00 03 e8  |...R.....|
00000030  00 00 00 00 e6 9d e2 9b  b2 d1 d6 43 4b 8b 29 ae  |.....CK.)..|
00000040  77 5a d8 c2 e4 8c 53 91  00 08 66 69 6c 65 2e 74  |wZ....S...file.t|
00000050  78 74 00 00 4d 1e 21 ff  5f ef 09 29 52 d2 7d f4  |xt..M.!_...)R.)..|
00000060  83 ac d5 a7 20 49 d8 0f                           |.... I..|
00000068
```

Spend a few moments trying to guess what any of this data might mean. Do any of the values look familiar to you? For example, we can see the filename `file.txt` appears in there. Does anything else jump out?

Fortunately, we don't need to guess our way through this, because Git includes documentation for the index file format². It tells us that the file begins with:

- A 12-byte header consisting of
 - 4-byte signature:
The signature is { 'D', 'I', 'R', 'C' } (stands for "dircache")
 - 4-byte version number:
The current supported versions are 2, 3 and 4.
 - 32-bit number of index entries.

32 bits is 4 bytes, so this header is 12 bytes in total. Let's snip the first 12 bytes out of our hexdump:

```
00000000  44 49 52 43 00 00 00 02  00 00 00 01          |DIRC.....|
```

As expected, we see the string `DIRC`, and then two four-byte numbers: `00 00 00 02` and `00 00 00 01`. This tells us the index is using version 2 of the file format, and there is 1 entry in it.

The documentation tells us that the header is followed by the entries themselves, and each entry begins with a few values that we get by calling `stat()` on the file, comprising ten 4-byte numbers in all:

```
32-bit ctime seconds, the last time a file's metadata changed
32-bit ctime nanosecond fractions
32-bit mtime seconds, the last time a file's data changed
32-bit mtime nanosecond fractions
32-bit dev
32-bit ino
32-bit mode
32-bit uid
32-bit gid
32-bit file size
```

The *ctime* is the file's *change* time, which is the last time either its contents or any of its metadata (like its permissions) was changed. The *mtime* is the *modify* time, which is the last time the file's contents was changed, ignoring metadata changes. *dev* is the ID of the hardware device the file

²<https://git-scm.com/docs/index-format>

resides on, and *ino* is the number of the *inode*³ storing its attributes. The *mode* is the number we studied in Section 5.1.1, “File modes”, and *uid* and *gid* are the IDs of the file’s user and group respectively. All these numbers are part of the data structure returned by the `stat()` system call.

Looking at our hexdump again, we can see the first four time-related values in the next block of 16 bytes. The nanosecond fields are zero because this system doesn’t report the ctime and mtime at that level of granularity.

```
00000000          5a 4f 7e c1 |          zo~.|
00000010  00 00 00 00 5a 4f 7e c0  00 00 00 00 |....zo~.....|
```

Following that, we see the six numbers storing the device and inode IDs, the mode, user and group IDs, and the file size. The final four bytes here (00 00 00 00) are the size, which is zero as we didn’t write any data to the file.

```
00000010          00 00 00 29 |          ...|
00000020  00 00 00 52 00 00 81 a4  00 00 03 e8 00 00 03 e8 |...R.....|
00000030  00 00 00 00 |....|
```

The documentation then says the 160-bit (20-byte) SHA-1 of the object appears. This is the ID of the blob created by writing the file’s contents to the `.git/objects` database. Since it’s 20 bytes, we know the object ID is being stored in packed binary form, not as text. Here it is in the hexdump:

```
00000030          e6 9d e2 9b  b2 d1 d6 43 4b 8b 29 ae |      .....CK.)|
00000040  77 5a d8 c2 e4 8c 53 91 |wZ....S. |
```

If we look up that object in the database, `cat-file` prints no output: this object is the empty blob.

```
$ git cat-file -p e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
```

The final elements of an entry are a two-byte set of flags — just like file modes, these two bytes pack several pieces of information into a small space — and the file’s name. The whole entry is padded with zeroes such that its total length is a multiple of 8.

```
00000040          00 08 66 69 6c 65 2e 74 |      ..file.t|
00000050  78 74 00 00 |xt.. |
```

For now, the only thing we need to know about the flags bytes (00 08 above) are that they contain the length of the filename: the string `file.txt` is 8 bytes long.

The index is ended by another 20-byte SHA-1 hash, only this time it’s the hash of the index contents itself. This is just used to check the integrity of the index, and doesn’t refer to anything in the object database.

```
00000050          4d 1e 21 ff  5f ef 09 29 52 d2 7d f4 |      M.!._...)R.)|
00000060  83 ac d5 a7 20 49 d8 0f |.... I..|
```

When the `status` command lists a file under *changes to be committed*, it means that the file is listed in the index with a blob ID that differs from the blob ID in the latest commit. When it lists that file under *changes not staged for commit*, that means the file is in the index, but the file in the working tree differs from its information — metadata and blob ID — that’s stored in

³<https://en.wikipedia.org/wiki/Inode>

the index. To include the changes in the next commit, we must use the add command to write the latest version as a blob and store that blob's ID in the index.

If we run `git add file.txt` again after updating its contents, we can see its size and object ID in the index change:

```
$ git add file.txt  
$ hexdump -C .git/index
```

Here's the section of the index listing the file's size (00 00 00 0c) and its object ID (b6 6b a0 ...):

```
00000030 00 00 00 0c b6 6b a0 6d 31 5d 46 28 0b b0 9d 54 |....k.m1]F(...T|  
00000040 61 4c c5 2d 16 77 80 9f |aL.-.w..|
```

And again, if we look up this blob in the database it does contain the file's new contents:

```
$ git cat-file -p b66ba06d315d46280bb09d54614cc52d1677809f  
new content
```

Let's also check what's been stored in the database. Even though we've not run `commit` on this repository yet, the index contains object IDs, so these objects must exist. Indeed, `.git/objects` does contain the two blobs we've seen in the index:

```
$ tree .git/objects/  
.git/objects/  
└── b6  
    └── 6ba06d315d46280bb09d54614cc52d1677809f  
└── e6  
    └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391  
└── info  
└── pack
```

When the index is updated, the old blob isn't removed. We only add new blobs to the database and change what the index points to. From our observations, we can deduce that the `add` command does two things: it stores the given file as a blob in `.git/objects`, and it adds a reference to that blob along with a cache of the file's current metadata to `.git/index`.

Finally, it's worth knowing the `ls-files` command. This lists all the files in the index, and if you pass the `--stage` flag it also lists the file's mode and object ID.

```
$ git ls-files --stage  
100644 b66ba06d315d46280bb09d54614cc52d1677809f 0 file.txt
```

It also lists the file's *stage*, which is the single 0 above, but we don't need to concern ourselves with that for now. We'll come back to this when we look at merge conflicts⁴.

6.3. Basic add implementation

There are quite a few details to get right for a properly working `add` command, but we can tackle them in small incremental steps. At first, `add` will just add things to the index and the `commit` command will remain unchanged; once we're happy with how the index is working we'll change `commit` to read from there rather than from the working tree.

⁴Section 18.3, "Conflicted index entries"

Below is a first pass at the top-level code for the add command in `bin/jit`. Just as in the previous commands, we begin by establishing the paths based on the current working directory, and create various pieces of Git infrastructure we'll need. Here, we need a `Workspace` and `Database`, and we'll also create a new kind of object: the `Index`.

Initially, add will take a single file path, and use `Workspace` to read that file and its metadata. We'll save the file contents as a blob, and then store the file path, the blob's ID and the `stat()` metadata in the index.

```
# bin/jit

command = ARGV.shift

case command

# ...

when "add"
  root_path = Pathname.new(Dir.getwd)
  git_path  = root_path.join(".git")

  workspace = Workspace.new(root_path)
  database  = Database.new(git_path.join("objects"))
  index     = Index.new(git_path.join("index"))

  path = Pathname.new(ARGV.first)
  data = workspace.read_file(path)
  stat = workspace.stat_file(path)

  blob = Database::Blob.new(data)
  database.store(blob)
  index.add(path, blob.oid, stat)

  index.write_updates
  exit 0
```

The `Workspace` and `Database` classes work just as before; everything up to storing the blob in the database is unchanged from what we've been doing for the `commit` command. The first new method we need is `Index#add`.

The job of `Index` is to manage the list of cache entries stored in the file `.git/index`. As such, we'll instantiate it with a hash called `@entries` to store its data, and a `Lockfile` to handle saving this data to disk.

```
# lib/index.rb

require "digest/sha1"

require_relative "./index/entry"
require_relative "./lockfile"

class Index
  def initialize(pathname)
    @entries = {}
    @lockfile = Lockfile.new(pathname)
  end
```

```
# ...
end
```

The add method takes a Pathname, an object ID, and a File::Stat and creates a cache entry out of them. It doesn't write this entry to disk immediately, it just stores it in memory — since we'll soon want to add lots of files to the index in one go, it makes sense to batch them up into a single write to the filesystem.

```
# lib/index.rb

def add(pathname, oid, stat)
  entry = Entry.create(pathname, oid, stat)
  @entries[pathname.to_s] = entry
end
```

This Entry class is not the one we have been using in the commit command. This actually refers to Index::Entry, a new class that will serve a similar purpose of packaging up a file path, blob ID and stat data, but with the addition of all the other fields contained in .git/index. It's defined as a struct containing all the fields that appear in a .git/index entry, and the Entry.create method takes the pathname, object ID and stat value and extracts all the information the entry needs.

```
# lib/index/entry.rb

class Index
  REGULAR_MODE      = 0100644
  EXECUTABLE_MODE  = 0100755
  MAX_PATH_SIZE    = 0xffff

  entry_fields = [
    :ctime, :ctime_nsec,
    :mtime, :mtime_nsec,
    :dev, :ino, :mode, :uid, :gid, :size,
    :oid, :flags, :path
  ]

  Entry = Struct.new(*entry_fields) do
    def self.create(pathname, oid, stat)
      path  = pathname.to_s
      mode  = stat.executable? ? EXECUTABLE_MODE : REGULAR_MODE
      flags = [path.bytesize, MAX_PATH_SIZE].min

      Entry.new(
        stat.ctime.to_i, stat.ctime_nsec,
        stat.mtime.to_i, stat.mtime_nsec,
        stat.dev, stat.ino, mode, stat.uid, stat.gid, stat.size,
        oid, flags, path)
    end
  end
end
```

The leading `0` in the values of REGULAR_MODE and EXECUTABLE_MODE means they're octal numbers; unlike trees in .git/objects, the index stores file modes as numbers rather than text. The leading `0x` in MAX_PATH_SIZE indicates a hexadecimal number. The path length is stored to make parsing easier: rather than scanning forward byte-by-byte until it finds a null byte, the

parser can read the length and jump forward that many bytes in one go. If the path is longer than FFF_{16} (4,095₁₀) bytes, Git stores FFF_{16} here and falls back to incremental scanning. To make this more efficient, entries are padded so their size is a multiple of 8 bytes, so the parser can scan by 8 bytes at a time rather than 1 byte.

That takes care of the `Index#add` method. To complete our basic version of the add command, we need the index to write itself to disk when we call `Index#write_updates`. We'll do this using `Lockfile`; if we fail to acquire the lock that means `.git/index.lock` already exists so some other process is currently making changes to the index. Once we've acquired the lock, we write the header containing the file format version and number of entries, and then each of the entries in turn.

```
# lib/index.rb

HEADER_FORMAT = "a4N2"

def write_updates
  return false unless @lockfile.hold_for_update

  begin_write
  header = ["DIRC", 2, @entries.size].pack(HEADER_FORMAT)
  write(header)
  @entries.each { |key, entry| write(entry.to_s) }
  finish_write

  true
end
```

Here we're again using `Array#pack` to serialise the header data; `a4N2` means a 4-byte string followed by two 32-bit big-endian⁵ numbers.

The `begin_write`, `write` and `finish_write` methods are defined as below. We need to end the file with a SHA-1 hash of its contents, so as we write content to the `Lockfile`, we also add it to a `Digest::SHA1` object as we go. `finish_write` writes the resulting hash to the `Lockfile` and then commits it. SHA-1 hashing can be performed incrementally on a stream of data, so implementing the writing process like this means we don't have to build up one big string of all the index data in memory and then get the SHA-1 of the whole thing.

```
# lib/index.rb

def begin_write
  @digest = Digest::SHA1.new
end

def write(data)
  @lockfile.write(data)
  @digest.update(data)
end

def finish_write
  @lockfile.write(@digest.digest)
  @lockfile.commit
end
```

⁵<https://en.wikipedia.org/wiki/Endianness>

Serialising the entries is handled by the `Entry#to_s` method:

```
# lib/index/entry.rb

ENTRY_FORMAT = "N10H40nZ*"
ENTRY_BLOCK  = 8

def to_s
  string = to_a.pack(ENTRY_FORMAT)
  string.concat("\0") until string.bytesize % ENTRY_BLOCK == 0
  string
end
```

Calling `to_a` on a struct returns an array of the values of all its fields, in the order they're defined in `Struct.new`. We're using `Array#pack` again, where the format string represents the following fields that correspond to how Git lays data out in the index:

- `N10` means ten 32-bit unsigned big-endian numbers
- `H40` means a 40-character hex string, which will pack down to 20 bytes
- `n` means a 16-bit unsigned big-endian number
- `Z*` means a null-terminated string

Finally, the string is padded with null bytes until its size is a multiple of the block size, 8 bytes. There must be at least one null byte to terminate the entry's path, so if this string's length is already a multiple of 8, then 8 null bytes are appended.

We can check our implementation by running `git add` and inspecting the `.git/index` file afterward, and then running `jit add` with the same file and checking the index looks the same as what Git wrote. Here's the index state after using Git to add a file:

```
$ rm -f .git/index* ; git add lib/index.rb ; hexdump -C .git/index

00000000  44 49 52 43 00 00 00 02  00 00 00 01 5a 50 d1 c4  |DIRC.....ZP..|
00000010  00 00 00 00 5a 50 d1 c4  00 00 00 00 00 00 2a  |....ZP.....*|
00000020  00 00 07 e1 00 00 81 a4  00 00 03 e8 00 00 03 e8  |.....|
00000030  00 00 03 32 fc 27 b7 5e  0d 12 9d 00 d8 40 dc 12  |...2.'^.@..|
00000040  aa 92 df ac e7 b1 44 e1  00 0c 6c 69 62 2f 69 6e  |.....D...lib/in|
00000050  64 65 78 2e 72 62 00 00  00 00 00 00 0d 5f e8 be  |dex.rb.....|
00000060  24 1c bf 31 3d cc 97 88  67 32 95 07 ac 1d 76 10  |$..1=...g2....v.|
00000070
```

And now the state after adding the same file using Jit:

```
$ rm -f .git/index* ; jit add lib/index.rb ; hexdump -C .git/index

00000000  44 49 52 43 00 00 00 02  00 00 00 01 5a 50 d1 c4  |DIRC.....ZP..|
00000010  00 00 00 00 5a 50 d1 c4  00 00 00 00 00 00 2a  |....ZP.....*|
00000020  00 00 07 e1 00 00 81 a4  00 00 03 e8 00 00 03 e8  |.....|
00000030  00 00 03 32 fc 27 b7 5e  0d 12 9d 00 d8 40 dc 12  |...2.'^.@..|
00000040  aa 92 df ac e7 b1 44 e1  00 0c 6c 69 62 2f 69 6e  |.....D...lib/in|
00000050  64 65 78 2e 72 62 00 00  00 00 00 00 0d 5f e8 be  |dex.rb.....|
00000060  24 1c bf 31 3d cc 97 88  67 32 95 07 ac 1d 76 10  |$..1=...g2....v.|
00000070
```

The index is byte-for-byte identical, so we know Jit is generating the same blob ID as Git, and it's writing all the same information to the index as Git does.

6.4. Storing multiple entries

Our basic add command works for the limited set of functionality we've covered so far: we can add a single file and Jit will write the same data to `.git/index` as Git does. But writing a single file to the index is not very useful — we'd like to have every file in the project indexed.

Though we will eventually want to implement incremental updates to the index rather than overwriting it each time, doing so is quite a bit of work and won't actually give us the feature we want right now in an easy-to-use way. It would be quite tedious to have to run `jit add <file>` by hand for every file in the project. It would be better if `jit add` could take multiple filenames at once, and we could combine this with other command-line programs to add all the files.

For example, it would be good to be able to run this:

```
$ jit add $(find bin lib -type f)

# or,

$ find bin lib -type f | xargs jit add
```

`find`⁶ is a program that searches for files, and `xargs`⁷ takes the output of one program and turns it into command-line arguments for another.

Accepting multiple inputs requires only a small change to the `add` command: rather than taking `ARGV.first` as the only input, we can iterate over everything in `ARGV`. In our original implementation, we wrote the following to store the file as a blob and add it to the index:

```
# bin/jit

path = Pathname.new(ARGV.first)
data = workspace.read_file(path)
stat = workspace.stat_file(path)

blob = Database::Blob.new(data)
database.store(blob)
index.add(path, blob.oid, stat)
```

All we need to do is put a loop around this that uses all the values in `ARGV`:

```
# bin/jit

ARGV.each do |path|
  path = Pathname.new(path)
  data = workspace.read_file(path)
  stat = workspace.stat_file(path)

  blob = Database::Blob.new(data)
  database.store(blob)
  index.add(path, blob.oid, stat)
end
```

⁶<https://manpages.ubuntu.com/manpages/bionic/en/man1/find.1posix.html>

⁷<https://manpages.ubuntu.com/manpages/bionic/en/man1/xargs.1posix.html>

We need to make a small adjustment to the `Index` class to make this work. Git requires the contents of `.git/index` to be sorted by filename, and that's easy to achieve when there's only one entry. Now that we have multiple entries, and inputs on the command-line won't necessarily be given in name order, we need an explicit way to keep the entries in order.

Ruby's `Hash` data structure doesn't provide a way to keep its keys sorted, but there is another structure we can use for this: `SortedSet`⁸. Like `Set`⁹, it's a collection that only allows unique elements, and the rules for checking if two members are equal are the same ones used for keys in a hash. The difference is that the `SortedSet#each` method ensures that iteration happens over the set's elements in sorted order.

We'll create a new `SortedSet` to hold the keys when we instantiate an `Index`:

```
# lib/index.rb

def initialize(pathname)
  @entries = {}
  @keys = SortedSet.new
  @lockfile = Lockfile.new(pathname)
end
```

When an entry is added to the `Index`, we'll add its key to this set, as well as to the original `@entries` hash.

```
# lib/index.rb

def add(pathname, oid, stat)
  entry = Entry.create(pathname, oid, stat)
  @keys.add(entry.key)
  @entries[entry.key] = entry
end
```

The `Index::Entry#key` method exists to encapsulate the data that's used to sort the entries; it just returns the entry's path.

```
# lib/index/entry.rb

def key
  path
end
```

With the `@keys` set in place, we can define a method for iterating over the index's entries, called `each_entry`. It iterates over the `@keys` set and yields each value in `@entries` in this order.

```
# lib/index.rb

def each_entry
  @keys.each { |key| yield @entries[key] }
end
```

Finally, we use this new method to iterate over the entries when writing the index out to disk. Rather than our current implementation which iterates over `@entries` directly:

```
# lib/index.rb
```

⁸<https://docs.ruby-lang.org/en/2.3.0/SortedSet.html>
⁹<https://docs.ruby-lang.org/en/2.3.0/Set.html>

```
def write_updates
# ...
@entries.each { |key, entry| write(entry.to_s) }
```

We'll use the new each_entry method to do the iteration, which makes sure the entries are written in order.

```
# lib/index.rb

def write_updates
# ...
each_entry { |entry| write(entry.to_s) }
```

With these changes in place, we can pass a list of all the project files to the add command, and git ls-files will confirm that they're all listed in the index.

```
$ rm -f .git/index* ; jit add $(find bin lib -type f) ; git ls-files

LICENSE.txt
bin/jit
lib/database.rb
lib/database/author.rb
lib/database/blob.rb
lib/database/commit.rb
lib/database/tree.rb
lib/entry.rb
lib/index.rb
lib/index/entry.rb
lib/lockfile.rb
lib/refs.rb
lib/workspace.rb
```

With all the project files in the index, it just became easier to check the state of our work in progress. Having an up-to-date index means the git status and git diff commands will now work, so we can easily see what we've changed while continuing work on the project.

6.5. Adding files from directories

Though we're now able to store all the project files in the index, it's still not as convenient as it could be. It's certainly nice that we can use other Unix tools like find to get all the project files, but it would be even better if Jit did this work for us.

Git allows you to pass directory names to the add command, and it will add everything inside that directory. For example, we'd like to be able to run

```
$ jit add lib/database
```

to add just the database-related files, or

```
$ jit add .
```

to add everything in the project. The shell can do a certain amount of work for us, for example running this command will make the shell expand the * into a list of all the files in the directory before running jit:

```
$ jit add lib/database/*
```

However, there's a limit to what the shell can do for us without the user interface becoming cumbersome. We'll need to add direct support for adding directories.

Currently, the add command code assumes that every argument given to it is a regular file, and so attempts to read it and store it as a blob.

```
# bin/jit

ARGV.each do |path|
  path = Pathname.new(path)
  data = workspace.read_file(path)
  stat = workspace.stat_file(path)

  blob = Database::Blob.new(data)
  database.store(blob)
  index.add(path, blob.oid, stat)
end
```

To support adding directories, we need to expand each argument into a list of files. If an argument names a single file, then it should expand to a list containing just that file, but if it's a directory, then it should expand to a list of all the files in that directory.

We can work towards this by making a change to `Workspace#list_files`. This method is currently designed to be used without arguments by the caller, in order to list every file in the project. It calls itself recursively with directory names in order to search the entire tree. Therefore, it currently assumes that any input given to it is a directory name, and so it calls `Dir.entries` on it.

If we improve this method so that it can also take names of regular files, then we can use it to expand any path into a list of files. Here's an updated implementation:

```
# lib/workspace.rb

def list_files(path = @pathname)
  if File.directory?(path)
    filenames = Dir.entries(path) - IGNORE
    filenames.flat_map { |name| list_files(path.join(name)) }
  else
    [path.relative_path_from(@pathname)]
  end
end
```

The input still defaults to the root of the `Workspace` so we can call it without arguments and get all the files in the project — this keeps our `commit` command working. If the input is indeed a directory, then we get all the entries inside the directory and recursively call `list_files` with each one. But, if the input is anything else, we return an array whose sole element is that input, relative to the root of the `Workspace`.

We've essentially turned the method inside-out. Rather than getting a directory's entries and checking if each of those is itself a directory, we check if the *input* is a directory and then list its contents if so.

Our improved `list_files` method means we can come back to the `add` command and change it to accept directory names. With each path from the `ARGV` array, we first expand it using

`File.expand_path` — this makes all paths absolute, and converts things like `.` into the path to the current directory. Then, we pass the result to `Workspace#list_files`, which gives us a list of paths relative to the root of the project. With each of these relative paths, we do the same thing we were doing before: we store a blob and add the result to the index.

```
# bin/jit

ARGV.each do |path|
  path = Pathname.new(File.expand_path(path))

  workspace.list_files(path).each do |pathname|
    data = workspace.read_file(pathname)
    stat = workspace.stat_file(pathname)

    blob = Database::Blob.new(data)
    database.store(blob)
    index.add(pathname, blob.oid, stat)
  end
end
```

The user interface of the `add` command is now complete. It can take any number of arguments, which can name files or directories, and it adds everything you would expect. Because of the use of `OrderedSet` to store the filenames, the index will only include each file once even if we call `add` with the same file multiple times.

```
$ jit add lib/database
$ git ls-files

lib/database/author.rb
lib/database/blob.rb
lib/database/commit.rb
lib/database/tree.rb
```

7. Incremental change

The work of the previous chapter has got us to a place where we can store all the project files in the index. In order to make the index truly useful, we need to be able to use it to compose commits, and that means we have two further changes to make. First, it should be possible to update the index incrementally, so we can add a single file to it and not lose all the other contents; this means we can be selective about which changes should go into the next commit. And second, we must change the `commit` command to read from the index, rather than the working tree.

7.1. Modifying the index

Our current `Index` implementation begins in an empty state in memory, and then has the inputs of the `add` command added to it before being dumped to disk. That means it overwrites whatever is currently stored there, so any files not mentioned in the call to `add` are forgotten.

```
$ jit add lib/database
$ git ls-files
lib/database/author.rb
lib/database/blob.rb
lib/database/commit.rb
lib/database/tree.rb

$ jit add lib/database/author.rb
$ git ls-files
lib/database/author.rb
```

The second call to `add` in the above example should not cause all the other files to be forgotten; it should just update `lib/database/author.rb` and leave everything else alone. To make this happen, we need to read the existing index from disk before making changes to it.

In `bin/jit`, let's add an extra call before the loop where we iterate over the `add` arguments to update the index. We'll call a new method, `Index#load_for_update`, to load the existing index into memory.

```
# bin/jit

workspace = Workspace.new(root_path)
database  = Database.new(git_path.join("objects"))
index     = Index.new(git_path.join("index"))

index.load_for_update

ARGV.each do |path|
  #
end
```

7.1.1. Parsing .git/index

The `Index#load_for_update` method takes care of all the steps needed to read the index data from `.git/index`, to rebuild the data that was in memory before the index was last written. We'll take a look at how it works below.

```
# lib/index.rb

def load_for_update
  if @lockfile.hold_for_update
    load
    true
  else
    false
  end
end
```

`load_for_update` begins by trying to acquire a lock using `Lockfile#hold_for_update`. Previously we've only used this method when we're about to change a file such as `.git/HEAD` — why is it necessary for reading?

The point of a lock is to stop two processes from overwriting each other's changes to the same file. If we didn't take a lock before reading the file, then the following sequence of events is possible:

- Alice runs `add`, which begins by reading the current index from disk
- Bob runs `add`, which also reads the index, getting the same data as Alice
- Alice adds `alice.rb` to her in-memory copy of the index
- Alice acquires a write lock and writes her index to `.git/index`
- Bob adds `bob.py` to his in-memory index
- Bob acquires a write lock and writes his index to `.git/index`

We now have a *lost update*: Alice's addition of `alice.rb` has been lost because Bob, not knowing about the existence of `alice.rb`, overwrote `.git/index`.

This situation can be prevented by requiring everyone to acquire the lock on `.git/index` before reading it, stopping anyone else from changing the file between the time you read it and the time you write it, releasing your lock. This strategy is a form of *pessimistic locking*, a type of concurrency control¹.

Having successfully opened the `Lockfile`, we call `load`, which does the real work of reading the index from disk.

```
# lib/index.rb

def load
  clear
  file = open_index_file

  if file
    reader = Checksum.new(file)
    count = read_header(reader)
    read_entries(reader, count)
    reader.verify_checksum
```

¹https://en.wikipedia.org/wiki/Concurrency_control

```
    end
  ensure
    file&.<close>
  end
```

The first thing this method does is called `Index#clear`, which resets the in-memory state of the `Index`. It does the work we were previously doing in `Index#initialize`, with the addition of setting a flag called `@changed` that will signal whether the index has been modified since it was loaded.

```
# lib/index.rb

def initialize(pathname)
  @pathname = pathname
  @lockfile = Lockfile.new(pathname)
  clear
end

def clear
  @entries = {}
  @keys    = SortedSet.new
  @changed = false
end
```

Next, we use `Index#open_index_file` to open a file handle for the locked file so we can read from it. This opens the real `.git/index`, rather than the lock file `.git/index.lock` which is currently empty awaiting new data to be written. If the file does not exist, this returns `nil`: this should not be treated as an error, as it's perfectly normal for a file we're changing not to exist yet.

```
# lib/index.rb

def open_index_file
  File.open(@pathname, File::RDONLY)
rescue Errno::ENOENT
  nil
end
```

After opening the file, we instantiate a new kind of object called a `Checksum`. Just as we calculated a SHA-1 hash of the contents while writing the index, we must now calculate the same thing while reading it, and verify the stored value at the end. As this logic is similar to that used for writing, I'm creating an abstraction for it rather than adding more state to the `Index` class itself.

The `Index::Checksum` class takes a file handle and creates a new `Digest::SHA1` object. Its `read` method reads the requested number of bytes from the file, raising an error if there is less data left in the file than we were expecting². It updates the checksum with this data and then returns it. It also has a `verify_checksum` method, which reads the final 20 bytes of the file and compares them to the checksum computed from the rest of the data.

```
# lib/index/checksum.rb

require "digest/sha1"
```

²Ruby's `File#read` method returns empty strings rather than raising an error once the end of the file is reached.

```
class Index
  class Checksum

    EndOfFile = Class.new(StandardError)

    CHECKSUM_SIZE = 20

    def initialize(file)
      @file = file
      @digest = Digest::SHA1.new
    end

    def read(size)
      data = @file.read(size)

      unless data.bytesize == size
        raise EndOfFile, "Unexpected end-of-file while reading index"
      end

      @digest.update(data)
      data
    end

    def verify_checksum
      sum = @file.read(CHECKSUM_SIZE)

      unless sum == @digest.digest
        raise Invalid, "Checksum does not match value stored on disk"
      end
    end
  end
end
```

The first step in processing the file data called from `Index#load` is the `read_header` method. Remember that `.git/index` begins with 12 bytes containing the string `DIRC`, a 4-byte number representing the file format version, and another 4-byte number containing the number of entries. In `read_header`, we read 12 bytes from the file, then use `String#unpack`³ to parse them. `String#unpack` is the inverse of `Array#pack`: just as we can use a format string to serialise an array of data, we can use the same format to parse a string back into an array. This will save us a lot of work while parsing.

```
# lib/index.rb

HEADER_SIZE = 12
HEADER_FORMAT = "a4N2"
SIGNATURE = "DIRC"
VERSION = 2

def read_header(reader)
  data = reader.read(HEADER_SIZE)
  signature, version, count = data.unpack(HEADER_FORMAT)

  unless signature == SIGNATURE
```

³<https://docs.ruby-lang.org/en/2.3.0/String.html#method-i-unpack>

```
    raise Invalid, "Signature: expected '#{ SIGNATURE }' but found '#{ signature }'"
end
unless version == VERSION
  raise Invalid, "Version: expected '#{ VERSION }' but found '#{ version }'"
end

count
end
```

If either the signature or version is not what we expect, we raise an error, otherwise we return the entry count so it can be used by the next step of the process.

Next, `load` calls `read_entries` with the `Checksum` instance and the entry count. This method reads the entries from disk into the in-memory `@entries` structure. It runs the following procedure either for `count` iterations, or until it runs out of data.

First, we read 64 bytes from the file. This is the smallest size that an entry can be; ten 4-byte numbers, a 20-byte object ID, two bytes of flags and then a null-terminated string, which must be at least one character and a null byte, all of which adds up to 64 bytes. An entry must end with a null byte and its size must be a multiple of 8 bytes, so until the last byte of entry is `\0` we read 8-byte blocks from the file and add them to `entry`. We could use the path length that's stored in the flags bytes to make this run faster, but it would also make the code more complicated since we need to fall back to scanning for paths longer than 4,095 bytes. Scanning also works for short paths, and I'd rather keep the code as simple as possible for now.

Once a complete entry has been read, we parse it by calling `Entry.parse` and store the result using `Index#store_entry`.

```
# lib/index.rb

ENTRY_FORMAT = "N10H40nZ*"
ENTRY_BLOCK = 8
ENTRY_MIN_SIZE = 64

def read_entries(reader, count)
  count.times do
    entry = reader.read(ENTRY_MIN_SIZE)

    until entry.byteslice(-1) == "\0"
      entry.concat(reader.read(ENTRY_BLOCK))
    end

    store_entry(Entry.parse(entry))
  end
end
```

Like `Index#read_header`, `Entry.parse` uses `String#unpack` with `ENTRY_FORMAT` to read the string into an `Entry` object. This lets us process quite a long list of different kinds of data with almost no code.

```
# lib/index/entry.rb

def self.parse(data)
  Entry.new(*data.unpack(ENTRY_FORMAT))
end
```

`Index#store_entry` does the work previously done by the `Index#add` method, by adding the `Entry` to the `@entries` hash and its key to the `@keys` set.

```
# lib/index.rb

def store_entry(entry)
  @keys.add(entry.key)
  @entries[entry.key] = entry
end
```

We can now change `Index#add` to call `store_entry` with the `Entry` it creates, and we'll also make it set the `@changed` flag to indicate something was added to the index since it was read from disk.

```
# lib/index.rb

def add(pathname, oid, stat)
  entry = Entry.create(pathname, oid, stat)
  store_entry(entry)
  @changed = true
end
```

Finally, `load` calls `Index::Checksum#verify_checksum` to check the data we've read is not corrupt. If all goes smoothly, the data has been successfully read from disk and `Index#load_for_update` returns `true`.

7.1.2. Storing updates

Once the data from `.git/index` has been loaded into memory, the `add` command uses the `Index#add` method to add new files to the index or overwrite existing ones. `Index#add` just modifies the data that's already in place, and then sets the `@changed` flag so we know the index is different from what's stored on disk.

To complete the story of incrementally updating the index, some changes are needed to `Index#write_updates`. Whereas before it was responsible for acquiring the write lock, it no longer needs to do that; we acquired this lock before reading the file. Instead, we modify its functionality so that it sets `@changed` to `false` at the end, to indicate all changes are written to disk. If `@changed` is `false` when this method is called, we just call `@lockfile.rollback` and return, skipping all the work of serialising the index and writing it to the filesystem.

I've also extracted the checksum functionality into the `Index::Checksum` class.

```
# lib/index.rb

def write_updates
  return @lockfile.rollback unless @changed

  writer = Checksum.new(@lockfile)

  header = [SIGNATURE, VERSION, @entries.size].pack(HEADER_FORMAT)
  writer.write(header)
  each_entry { |entry| writer.write(entry.to_s) }

  writer.write_checksum
```

```
    @lockfile.commit

    @changed = false
end
```

`Lockfile#rollback` is similar to `Lockfile#commit`, except that rather than renaming the `.lock` file to the real path, it just deletes the `.lock` file so any changes are discarded and other processes are now free to acquire the lock.

```
# lib/lockfile.rb

def rollback
  raise_on_stale_lock

  @lock.close
  File.unlink(@lock_path)
  @lock = nil
end
```

The `write` and `write_checksum` methods in `Index::Checksum` are just the old `write` and `finish_write` methods from `Index`, extracted into this new class. Since `Index::Checksum` is instantiated with a file handle, it can be used for writing as well as reading, as long as we pass all data flowing through it into the `Digest::SHA1` object.

```
# lib/index/checksum.rb

def write(data)
  @file.write(data)
  @digest.update(data)
end

def write_checksum
  @file.write(@digest.digest)
end
```

We've now changed the `add` command so that it reads the index data from `.git/index` on disk, modifies it somehow, then writes the updated index back to the filesystem. This means we can update the index gradually as we work, and be selective about which changes we want to include in the next commit. The next step is to have the `commit` command use the index, rather than the working tree, to build the next commit tree.

7.2. Committing from the index

The index now works well enough that we can actually begin to use it. In Git, committed trees are prepared by reading from the index, rather than from the working tree. This means the `commit` command doesn't need to hash the entire project every time — it just uses the blob IDs that are cached in `.git/index`.

Here's the state of the `commit` command up to the point where we store the trees, as we left it back in Section 5.2.2, “Building a Merkle tree”. It scans the `Workspace` for all the files in the project, stores blobs for all of them, and generates a list of `Entry` objects to feed into `Database::Tree.build`.

```
# bin/jit
```

```
workspace = Workspace.new(root_path)
database = Database.new(db_path)
refs     = Refs.new(git_path)

entries = workspace.list_files.map do |path|
  data = workspace.read_file(path)
  blob = Database::Blob.new(data)

  database.store(blob)

  stat = workspace.stat_file(path)
  Entry.new(path, blob.oid, stat)
end

root = Database::Tree.build(entries)
root.traverse { |tree| database.store(tree) }
```

Much of this code can now be removed: the `add` command is now responsible for storing blobs, and the `Index` keeps a cache of everything needed to build a tree, including each entry's pathname, blob ID and file mode. Rather than asking the `Workspace` for all the files, let's instead put in a call to `Index#load` to get the index into memory, and then pass `Index#each_entry` to `Database::Tree.build`.

```
# bin/jit

database = Database.new(git_path.join("objects"))
index    = Index.new(git_path.join("index"))
refs     = Refs.new(git_path)

index.load

root = Database::Tree.build(index.each_entry)
root.traverse { |tree| database.store(tree) }
```

We only need to call `Index#load`, not `Index#load_for_update`, because this command is not going to mutate the index. It can just read it once, and then it has no further interaction with the `.git/index` file.

`Index#each_entry` is a method we'd previously used in `Index#write_updates` to save the entries to the file:

```
# lib/index.rb

def write_updates
  #
  # ...
  each_entry { |entry| writer.write(entry.to_s) }
```

In this use it takes a block, but we can change it slightly so that it's also callable without a block, and it will then return a collection called an Enumerator⁴ containing the entries. `enum_for(:each_entry)` returns an object whose `each` method will iterate over the entries.

```
# lib/index.rb

def each_entry
```

⁴<https://docs.ruby-lang.org/en/2.3.0/Enumerator.html>

```
if block_given?
  @keys.each { |key| yield @entries[key] }
else
  enum_for(:each_entry)
end
end
```

This means that `Database::Tree.build` can simply call `each` on the value it receives. Rather than using `each_entry`, which would couple it to the `Index` interface, calling `each` allows `Database::Tree.build` to take any kind of Ruby collection and not care what type it is. Also, because `Index` keeps its entries in sorted order, `Tree.build` no longer needs to sort the collection it receives itself.

To keep `Tree.build` and `Tree#add_entry` working, `Index::Entry` needs to support the same interface as the original `Entry` class. In particular, we need to implement the `parent_directories` and `basename` methods, this time using the entry's path field.

```
# lib/index/entry.rb

def parentDirectories
  Pathname.new(path).descend.to_a[0...-2]
end

def basename
  Pathname.new(path).basename
end
```

One other small adjustment must be made to `Database::Tree#to_s`. The original `Entry` class had a `mode` method that returned the mode as a string, since that's what `Tree` wanted to store it as. However, the `Index::Entry` class stores and returns `mode` as a number, and we must convert it to an octal string representation here.

```
# lib/database/tree.rb

def to_s
  entries = @entries.map do |name, entry|
    mode = entry.mode.to_s(8)
    ["#{ mode } #{ name }", entry.oid].pack(ENTRY_FORMAT)
  end

  entries.join("\n")
end
```

We'll change the `Tree#mode` method to match; let's store the numerical mode for trees in a constant because other code will need to refer to it later.

```
# lib/database/tree.rb

TREE_MODE = 040000

def mode
  TREE_MODE
end
```

We've not had to change a lot here to migrate from the workspace to the index. All we've done is remove the code for storing blobs in favour of reading from the index, and made some small

additions to make `Index::Entry` compatible with `Database::Tree`. The overall structure of the `Database::Tree` logic has remained largely unchanged.

7.3. Stop making sense

At this point, we need to take care of some edge cases in how the index works. To see what these are, let's use Jit as we've built it so far to make a repository with two files in it.

```
$ jit init tmp  
$ cd tmp  
$ touch alice.txt bob.txt  
$ jit add .  
$ echo "first commit" | jit commit
```

If we use `git ls-files` to examine the state of the index, we see the files that we just added.

```
$ git ls-files  
alice.txt  
bob.txt
```

Now, suppose we remove the file `alice.txt` and replace it with a directory of the same name, placing another file inside it.

```
$ rm alice.txt  
$ mkdir alice.txt  
$ touch alice.txt/nested.txt  
$ jit add .
```

Let's run `ls-files` again to check the index state:

```
$ git ls-files  
alice.txt  
alice.txt/nested.txt  
bob.txt
```

This list does not make sense: it is not possible to have both a file and a directory with the same name in a normal filesystem. Though our working tree only contains the regular files `alice.txt/nested.txt` and `bob.txt`, the index maintains that the file `alice.txt` still exists. If we turned this index into a commit, we'd never be able to check it out, since the files `alice.txt` and `alice.txt/nested.txt` cannot both exist at the same time.

This is the problem with caches, especially those that are updated piecemeal. It's easy to implement them naively such that the state they contain cannot possibly represent a state of affairs in the real world. When the index was overwritten on every invocation of `add`, this was not a problem: the contents would definitely reflect the state of the filesystem at a single point in time. But now that we can change it bit-by-bit, it might contain entries that are not compatible with one another, as the files on disk can change between `add` calls. We'd better fix the `Index#add` method so that it discards any existing entries that don't fit with the new one.

7.3.1. Starting a test suite

All the code we've written so far is there to directly support the commands we wanted to run while developing the project. It would be easy for us to notice that some element of the `add`

or `commit` command was broken, because either Jit would crash and show a stack trace, or it would write bad data to the database or index and we'd be alerted to this when checking our work with Git's `status`, `log` and `show` commands. In short it would be difficult for us not to notice bugs in this code in the normal course of our work on it.

Now we're about to start adding code to address edge cases; situations that don't necessarily crop up in our project but will cause some repositories to become invalid if left unchecked. This is a good time to begin writing some automated tests so that we know we've not accidentally broken things without noticing.

Let's add a single test and the infrastructure needed to run it. We'll check that adding a single item to `Index` causes that item to show up in `Index#each_entry`. The test below is written using minitest⁵, a gem⁶ that's bundled with recent Ruby versions. Loading `minitest/autorun` means minitest will execute any tests that have been defined when this file is run. The lines that look like

```
let(:name) { expression }
```

define values that are available within the tests. The first time `name` is mentioned in each test, `expression` will be evaluated to return its value.

We set up `index` to refer to a new `Index` object, instantiated using a path inside the test directory. This path doesn't particularly matter since we won't write the index to disk during this test — it will only exist in memory. We're only going to be testing how file paths are handled, but the `Index#add` method requires an object ID and a `File::Stat` value. It doesn't matter what these are as long as they're valid; we'll use `SecureRandom`⁷ to generate an object ID, and use the `File.stat` of the test file itself for these values.

The test proper begins with the line `it "adds a single file"`; minitest runs the code in this block and if the assertion fails, it reports an error. All we're checking here is that the file paths in the index match the names we've passed to `Index#add`.

```
# test/index_test.rb

require "minitest/autorun"

require "pathname"
require "securerandom"
require "index"

describe Index do
  let(:tmp_path) { File.expand_path("../tmp", __FILE__ ) }
  let(:index_path) { Pathname.new(tmp_path).join("index") }
  let(:index) { Index.new(index_path) }

  let(:stat) { File.stat(__FILE__ ) }
  let(:oid) { SecureRandom.hex(20) }

  it "adds a single file" do
    index.add("alice.txt", oid, stat)
```

⁵<http://docs.seattlerb.org/minitest/>

⁶Gems are libraries of third-party code that's not part of Ruby's standard library, although Ruby does include a few widely-used gems out of the box. Find out more at <https://rubygems.org/>.

⁷<https://docs.ruby-lang.org/en/2.3.0/SecureRandom.html>

```
    assert_equal ["alice.txt"], index.each_entry.map(&:path)
end
end
```

This file can be run directly from the command-line by running `ruby test/index_test.rb`. However, as more test files are added to the project it's more common to define a task for running them all. Rake⁸, a build tool included with Ruby, includes a task for doing just that.

```
# Rakefile

require "rake/testtask"

Rake::TestTask.new do |task|
  task.pattern = "test/*_test.rb"
end

task :default => :test
```

Now we can run `rake` and see the results of the single test we've defined. Rake and minitest are third-party packages bundled with Ruby, so if you have set `RUBYOPT` to disable gems⁹, you'll need to unset that variable; run `unset RUBYOPT` before running the following command.

```
$ rake

# Running:

.

Finished in 0.002064s, 484.4961 runs/s, 484.4961 assertions/s.

1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

Everything seems to be working, so we'll commit this and begin adding more tests.

7.3.2. Replacing a file with a directory

In Section 7.3, “Stop making sense”, we saw that adding a file whose parent directory had the same name as an existing file, left the existing file in the index.

```
$ git ls-files
alice.txt
alice.txt/nested.txt
bob.txt
```

When we add `alice.txt/nested.txt`, we would like the entry for `alice.txt` to be removed, since that name is now bound to a directory and so cannot be a regular file. Let's write a test to express this requirement.

```
# test/index_test.rb

it "replaces a file with a directory" do
  index.add("alice.txt", oid, stat)
  index.add("bob.txt", oid, stat)
```

⁸<https://ruby.github.io/rake/>

⁹Section 5.2, “Nested trees”

```
    index.add("alice.txt/nested.txt", oid, stat)

    assert_equal ["alice.txt/nested.txt", "bob.txt"],
                 index.each_entry.map(&:path)
end
```

This test follows the *Arrange, Act, Assert*¹⁰ pattern that's common to a lot of unit testing. First we arrange the state of the system to some starting point; we add the files alice.txt and bob.txt to the index. Then we act, that is we perform the action we're trying to test, which in this case means adding a new entry for alice.txt/nested.txt. Finally, we assert, to check the result of the action. Here, we'd like to see that when we list all the paths in the index, it contains alice.txt/nested.txt, but not alice.txt.

Running the tests now shows us that this functionality does not work as required: alice.txt appears in the output.

```
$ rake

# Running:

.F

Finished in 0.020562s, 97.2668 runs/s, 97.2668 assertions/s.

1) Failure:
Index#test_0002_replaces a file with a directory
--- expected
+++ actual
@@ -1 +1 @@
-["alice.txt/nested.txt", "bob.txt"]
+["alice.txt", "alice.txt/nested.txt", "bob.txt"]

2 runs, 2 assertions, 1 failures, 0 errors, 0 skips
rake aborted!
Command failed with status (1)

Tasks: TOP => test
(See full trace by running task with --trace)
```

Now we know that the test accurately expresses our requirement and identifies a shortcoming in the code, we can go about fixing it. In Index#add, we'll add a step before storing the new Index::Entry that discards any entries that it might conflict with.

```
# lib/index.rb

def add(pathname, oid, stat)
  entry = Entry.create(pathname, oid, stat)
  discard_conflicts(entry)
  store_entry(entry)
  @changed = true
end
```

The `discard_conflicts` method will remove any entries whose name matches the name of one of the new entry's parent directories. For example, if the new entry is lib/index/

¹⁰<http://wiki.c2.com/?ArrangeActAssert>

entry.rb, then the entries for lib and lib/index must be removed if they exist. We can use the Index::Entry#parent_directories method for this.

```
# lib/index.rb

def discard_conflicts(entry)
  entry.parent_directories.each do |dirname|
    @keys.delete(dirname.to_s)
    @entries.delete(dirname.to_s)
  end
end
```

Strictly speaking, removing entries from @keys is all that's needed to make the test pass, since that means they won't be yielded by Index#each_entry. However, having data in @entries that's not mirrored in @keys is more likely to cause confusion and further errors, so I'd rather keep them consistent.

We can go back to our original example and check that adding all the files again causes the unwanted entry to be removed:

```
$ git ls-files
alice.txt
alice.txt/nested.txt
bob.txt

$ jit add .

$ git ls-files
alice.txt/nested.txt
bob.txt
```

That's one problem fixed! The inverse of this problem is when a file is added whose name matches existing directories in the index, and this requires a little more work to remedy.

7.3.3. Replacing a directory with a file

Imagine you have an index containing two files: alice.txt, and nested/bob.txt. Now we try to add a file called simply nested. This means the existing entry nested/bob.txt is now invalid, because nested is now a regular file and so can no longer be a directory. All files inside this directory must be removed from the index.

Below are a couple of tests to express this requirement. The first is a simple case where the conflicted directory name only has direct children. The second is more complicated in that the directory itself has subdirectories that must also be removed.

```
# test/index_test.rb

it "replaces a directory with a file" do
  index.add("alice.txt", oid, stat)
  index.add("nested/bob.txt", oid, stat)

  index.add("nested", oid, stat)

  assert_equal ["alice.txt", "nested"],
```

```
    index.each_entry.map(&:path)
end

it "recursively replaces a directory with a file" do
  index.add("alice.txt", oid, stat)
  index.add("nested/bob.txt", oid, stat)
  index.add("nested/inner/claire.txt", oid, stat)

  index.add("nested", oid, stat)

  assert_equal ["alice.txt", "nested"],
    index.each_entry.map(&:path)
end
```

Running the tests again, both these new examples fail.

```
1) Failure:
Index#test_0003_replaces a directory with a file
--- expected
+++ actual
@@ -1 +1 @@
-["alice.txt", "nested"]
+["alice.txt", "nested", "nested/bob.txt"]

2) Failure:
Index#test_0004_recursively replaces a directory with a file
--- expected
+++ actual
@@ -1 +1 @@
-["alice.txt", "nested"]
+["alice.txt", "nested", "nested/bob.txt", "nested/inner/claire.txt"]
```

When the entry `nested` is added, the entries `nested/bob.txt` and `nested/inner/claire.txt` must be removed. Here's the state of the `Index` before this new entry is added:

```
@keys = SortedSet.new([
  "alice.txt",
  "nested/bob.txt",
  "nested/inner/claire.txt"
])

@entries = {
  "alice.txt"          => Entry.new(...),
  "nested/bob.txt"     => Entry.new(...),
  "nested/inner/claire.txt" => Entry.new(...)
}
```

How should we detect which entries to remove when `nested` is added? We could use the data we already have and iterate over all the entries to find those that have this new file as one of their parent directories:

```
entries_to_remove = @entries.values.select do |entry|
  entry.parent_directories.include?("nested")
end
```

This would probably work, but I'd rather not iterate over the entire index every time we add a new entry. It would be nice to index the data in a way that makes deleting things easier.

Maybe we could restructure `@entries` so that, rather than being a flat mapping from full paths to `Entry` objects, it's a nested structure that mirrors the filesystem:

```
  @entries = {
    "alice.txt" => Entry.new(...),
    "nested" => {
      "bob.txt" => Entry.new(...),
      "inner" => {
        "claire.txt" => Entry.new(...)
      }
    }
  }
```

This makes removing entire directories much easier:

```
# delete "nested"
@entries.delete("nested")

# delete "nested/inner"
@entries["nested"].delete("inner")
```

However, we still need the `@keys` set, because the entries must be sorted by their full pathname rather than filenames in each subtree being sorted. And we just made inserting and looking up a single entry more complicated; instead of

```
@entries["lib/index/entry.rb"]
```

we now have to split the path up and do three hash lookups:

```
path = Pathname.new("lib/index/entry.rb")

entry = path.each_filename.reduce(@entries) do |table, filename|
  table[filename]
end
```

Attempting to restructure `@entries` to make some operations more convenient has made other operations harder. In such situations, you can always add another data structure, as long as you keep it in sync with the others. If you run a web application, you likely have a primary database, a search index, and a cache, and it's fine to have copies of the same information in all of these as long as they're consistent enough for your needs. Each data store exists to make certain kinds of queries easier or faster.

In this case, I'm going to add another structure to `Index` that stores a set of all the entry keys that live inside each directory. Here's our index state again:

```
@keys = SortedSet.new([
  "alice.txt",
  "nested/bob.txt",
  "nested/inner/claire.txt"
])

@entries = {
  "alice.txt" => Entry.new(...),
  "nested/bob.txt" => Entry.new(...),
  "nested/inner/claire.txt" => Entry.new(...)
}
```

To these I'm going to add the following structure:

```
#parents = {
    "nested"      => Set.new(["nested/bob.txt", "nested/inner/claire.txt"]),
    "nested/inner" => Set.new(["nested/inner/claire.txt"])
}
```

The `@parents` variable is a hash that maps directory names to a set of all the entry keys that fall under that directory. Having this structure means that when a file is added, we can look up its name in `@parents`, and if it exists, remove all the entries named in the corresponding set. Let's set it up in the `Index#clear` method; the syntax below creates a Hash that creates a new Set whenever we ask it for a key that doesn't exist yet¹¹.

```
# lib/index.rb

def clear
  @entries = {}
  @keys    = SortedSet.new
  @parents = Hash.new { |hash, key| hash[key] = Set.new }
  @changed = false
end
```

Now when we store an entry in the index, we iterate over its parent directories and add the entry's path to each directory's entry set in `@parents`.

```
# lib/index.rb

def store_entry(entry)
  @keys.add(entry.key)
  @entries[entry.key] = entry

  entry.parent_directories.each do |dirname|
    @parents[dirname.to_s].add(entry.path)
  end
end
```

With the `@parents` structure populated, we can extend `Index#discard_conflicts` and use it to remove directories that conflict with the new entry. If `@parents` contains the given path, then we clone the children for that path and pass each one to `remove_entry`. We need to clone `@parents[path]` before iterating it, because `remove_entry` will modify `@parents` as each child path is removed, and it's an error to iterate over a Set while it's being mutated.

```
# lib/index.rb

def discard_conflicts(entry)
  entry.parent_directories.each { |parent| remove_entry(parent) }
  remove_children(entry.path)
end

def remove_children(path)
  return unless @parents.has_key?(path)

  children = @parents[path].clone
  children.each { |child| remove_entry(child) }
```

¹¹<https://docs.ruby-lang.org/en/2.3.0/Hash.html#method-c-new>

```
end
```

`Index#remove_entry` consists of the logic that previously lived in `discard_conflicts`, plus logic for removing the entry from the `@parents` structure. This is the inverse of what we do in `store_entry`: we iterate over the entry's parent directories and remove the entry's path from each parent's set in `@parents`. If this leaves any of these sets empty, the set is removed from `@parents` entirely.

```
# lib/index.rb

def remove_entry(pathname)
  entry = @entries[pathname.to_s]
  return unless entry

  @keys.delete(entry.key)
  @entries.delete(entry.key)

  entry.parent_directories.each do |dirname|
    dir = dirname.to_s
    @parents[dir].delete(entry.path)
    @parents.delete(dir) if @parents[dir].empty?
  end
end
```

Now, we rerun the tests and see that everything is passing.

```
$ rake

# Running:

....
```



```
Finished in 0.004465s, 895.8567 runs/s, 895.8567 assertions/s.

4 runs, 4 assertions, 0 failures, 0 errors, 0 skips
```

A little extra work when adding and removing an entry has made it easy to detect file/directory name conflicts and remove entries that are not compatible with the new addition. We don't have to find one magic data structure that solves all our problems, we can use multiple structures that solve different problems, as long as we keep them in sync.

7.4. Handling bad inputs

The index functionality is now fleshed out enough to handle a lot of common use cases, and so we probably won't want to radically change it for a while. Now is a good time to consider what might go wrong in the `add` command, and add code to handle any errors.

The main work of this command is done between the call to load the index, `index.load_for_update`, and the one to commit the changes to disk, `index.write_updates`. That code currently looks like this:

```
# bin/jit

index.load_for_update
```

```
ARGV.each do |path|
  path = Pathname.new(File.expand_path(path))

  workspace.list_files(path).each do |pathname|
    data = workspace.read_file(pathname)
    stat = workspace.stat_file(pathname)

    blob = Database::Blob.new(data)
    database.store(blob)
    index.add(pathname, blob.oid, stat)
  end
end

index.write_updates
```

The command gets all its inputs from the user, and there are various ways the user's input might be wrong. Before we had incremental updating, the only viable use of add was to call it with the root directory of the project, but now, it'll be more common to specify individual files to add. The arguments the user provides might name files that don't exist, or they might be files we are not allowed to read. The add command should handle these situations by printing an appropriate error, exiting with a non-zero status and it should leave the index unchanged and unlocked.

7.4.1. Non-existent files

Let's begin by creating a single file in an otherwise-empty directory:

```
$ echo "hello" > hello.txt
```

When we try to add this file using Git, along with the name of another file that does not exist, we get an error, and nothing is written to the database:

```
$ git add hello.txt no-such-file
fatal: pathspec 'no-such-file' did not match any files

$ echo $?
128

$ tree .git/objects
.git/objects
├── info
└── pack
```

We can handle this in Jit by changing how `Workspace#list_files` works. If the path passed into it does not correspond to a file, then we'll raise an error. We could also choose to handle this by returning an empty array, on the basis that the given name does not match any files, but then we would not be able to differentiate between calling `add` with the name of an empty directory (which is not an error) and calling it with a non-existent file (which is).

Previously, `Workspace#list_files` was not checking if the filename it was given existed in the filesystem. Let's add a check for that and continue to return an array containing that name if so. If the file does not exist, then we'll raise an exception.

```
# lib/workspace.rb

MissingFile = Class.new(StandardError)
```

```
def list_files(path = @pathname)
  relative = path.relative_path_from(@pathname)

  if File.directory?(path)
    filenames = Dir.entries(path) - IGNORE
    filenames.flat_map { |name| list_files(path.join(name)) }
  elsif File.exist?(path)
    [relative]
  else
    raise MissingFile, "pathspec '#{ relative }' did not match any files"
  end
end
```

Next, we'll separate the add code into two pieces. It currently loops over the arguments in ARGV once, and for each argument, it passes the value to `Workspace#list_files` and then iterates over the result of that call, adding all the files to the index. In the above example, that would result in `hello.txt` being written to the database before we discovered that no-such-file does not exist. Let's break it up so that first we discover all the paths to be added, and then we add each of them to the index once we have the complete list.

```
# bin/jit

paths = ARGV.flat_map do |path|
  path = Pathname.new(File.expand_path(path))
  workspace.list_files(path)
end

paths.each do |path|
  data = workspace.read_file(path)
  stat = workspace.stat_file(path)

  blob = Database::Blob.new(data)
  database.store(blob)
  index.add(path, blob.oid, stat)
end
```

Then, we can put an error-catching `begin/rescue` block around the first loop so that if any of the inputs are found not to exist, we print an error and exit, before we even begin adding anything to the database.

```
# bin/jit

begin
  paths = ARGV.flat_map do |path|
    path = Pathname.new(File.expand_path(path))
    workspace.list_files(path)
  end
rescue Workspace::MissingFile => error
  $stderr.puts "fatal: #{ error.message }"
  index.release_lock
  exit 128
end
```

We've also added a call to `Index#release_lock` just before exiting. Recall that the `add` command begins by acquiring a lock on `.git/index` by creating the file `.git/index.lock`.

We don't want to leave that file in place after `jit` exits, because then further calls to the `add` command will always fail. The `Index#release_lock` method simply calls `Lockfile#rollback` to delete the `.lock` file along with any changes written to it.

```
# lib/index.rb

def release_lock
  @lockfile.rollback
end
```

7.4.2. Unreadable files

When Git's `add` command is given the names of files that exist but are unreadable, its behaviour is slightly different. It still avoids updating the index entirely; if any of the inputs to `add` cannot be read then none of the inputs gets added. Let's create another file and make it unreadable using `chmod -r`:

```
$ echo "world" > world.txt
$ chmod -r world.txt
```

Now, if we add our readable file `hello.txt` and the unreadable file `world.txt`, we get a different error, but the database contains a single object after the call:

```
$ git add hello.txt world.txt
error: open("world.txt"): Permission denied
error: unable to index file world.txt
fatal: adding files failed

$ echo $?
128

$ tree .git/objects
.git/objects
├── ce
│   └── 013625030ba8dba906f756967f9e9ca394464a
└── info
└── pack
```

The object in `.git/objects/ce` has an object ID that you might recognise as the blob `hello\n`; it's the contents of `hello.txt`, so that file did get added to the database. However, this blob is not in the index, indeed in this new repository there is no `.git/index` file and `git ls-files` does not print anything. I'm not entirely sure why Git adds this object to the database with no references in `.git/index` or `.git/refs` pointing to it — with nothing pointing to it, the object may very well get lost.

Nevertheless, we should handle this kind of error, and we'll just follow Git's lead for compatibility purposes. We'll begin by catching `Errno::EACCES` in `Workspace#read_file` and `Workspace#stat_file`, the methods for reading from the filesystem, and raising our own `NoPermission` error with a brief error message.

```
# lib/workspace.rb

NoPermission = Class.new(StandardError)

def read_file(path)
```

```
File.read(@pathname.join(path))
rescue Errno::EACCES
  raise NoPermission, "open('#{ path }'): Permission denied"
end

def stat_file(path)
  File.stat(@pathname.join(path))
rescue Errno::EACCES
  raise NoPermission, "stat('#{ path }'): Permission denied"
end
```

Next, we'll put a handler for this new error around the loop that adds blobs to the database and the index, so that any paths up to the one that's unreadable are successfully saved.

```
# bin/jit

begin
  paths.each do |path|
    data = workspace.read_file(path)
    stat = workspace.stat_file(path)

    blob = Database::Blob.new(data)
    database.store(blob)
    index.add(path, blob.oid, stat)
  end
rescue Workspace::NoPermission => error
  $stderr.puts "error: #{ error.message }"
  $stderr.puts "fatal: adding files failed"
  index.release_lock
  exit 128
end
```

The add command now handles the most common types of user input errors around reading files, but in solving them we've highlighted one further error condition it should handle: what if the .git/index.lock file exists when it's invoked?

7.4.3. Locked index file

We can easily trigger Git's error handler for a locked index file by touching the file .git/index.lock before running the add command.

```
$ touch .git/index.lock
$ git add hello.txt
fatal: Unable to create '/Users/jcoglan/git-add-errors/.git/index.lock': File exists.
```

Another git process seems to be running in this repository, e.g.
an editor opened by 'git commit'. Please make sure all processes
are terminated **then** try again. If it still fails, a git process
may have crashed in this repository earlier:
remove the file manually to **continue**.

```
$ echo $?
128
```

Git prints more than just a one-line error message in this case — it tries to tell you why this might happen and what you should do about it. Curiously, the example it gives is misleading;

while the `commit` command has your editor open to compose a commit message, the index is not actually locked¹².

To handle this error we're going to need some minor changes to existing code. `Lockfile#hold_for_update` currently returns `true` or `false` to indicate whether the lock was acquired or already held. In order to generate an error including the path of the `.lock` file, let's instead make it raise an exception on failure, which is the `rescue` branch for `Errno::EEXIST`.

```
# lib/lockfile.rb

LockDenied = Class.new(StandardError)

def hold_for_update
  unless @lock
    flags = File::RDWR | File::CREAT | File::EXCL
    @lock = File.open(@lock_path, flags)
  end
rescue Errno::EEXIST
  raise LockDenied, "Unable to create '#{ @lock_path }': File exists."
rescue Errno::ENOENT => error
  raise MissingParent, error.message
rescue Errno::EACCES => error
  raise NoPermission, error.message
end
```

This change means that `Index#load_for_update` can be simplified to simply call `Lockfile#hold_for_update` without checking its return value; if an exception is raised then `load` will not be called.

```
# lib/index.rb

def load_for_update
  @lockfile.hold_for_update
  load
end
```

In the `add` command code, we must catch this exception from our call to `Index#load_for_update` and print the necessary error messages. The symbol `<<~` begins a *heredoc*¹³, where the tilde indicates that leading indentation should be stripped from the string's lines.

```
# bin/jit

begin
  index.load_for_update
rescue Lockfile::LockDenied => error
  $stderr.puts <<~ERROR
  fatal: #{ error.message }

  Another jit process seems to be running in this repository.
  Please make sure all processes are terminated then try again.
```

¹²I assume this message was once true in some version of Git, but at some point the behaviour was changed and this message was not updated. Unlike our implementation so far, Git's `commit` command does modify the index by caching additional information about trees in it, but it commits those changes and releases the lock before launching your editor. The current branch pointer in `.git/refs/heads` is not locked until the commit is written to the database

¹³https://docs.ruby-lang.org/en/2.3.0/syntax/literals_rdoc.html#label-Here+Documents

```
If it still fails, a jit process may have crashed in this
repository earlier: remove the file manually to continue.

ERROR
exit 128
end
```

This change to `Lockfile#hold_for_update` also allows the other caller of it to be simplified. In `Refs#update_head` we're currently checking its return value and raising an exception on failure.

```
# lib/refs.rb

def update_head(oid)
  lockfile = Lockfile.new(head_path)

  unless lockfile.hold_for_update
    raise LockDenied, "Could not acquire lock on file: #{ head_path }"
  end

  lockfile.write(oid)
  lockfile.write("\n")
  lockfile.commit
end
```

`Lockfile` is now raising an exception for us, and so this check can be removed.

```
# lib/refs.rb

def update_head(oid)
  lockfile = Lockfile.new(head_path)

  lockfile.hold_for_update
  lockfile.write(oid)
  lockfile.write("\n")
  lockfile.commit
end
```

The add command is now much more robust against the common errors it will encounter, but these changes have left its code in a rather messy state. In the next chapter we will alter the command code to make it easier to work with.

8. First-class commands

Jit now supports three commands in its command-line interface: `init`, `add` and `commit`. While we've created abstractions to manage the data in `.git`, the commands that use these abstractions are still one long case expression in `bin/jit`. This design is beginning to hurt: the code is accruing a fair amount of logic, and it's expressed as one big script rather than being broken down into smaller functions. For example, after the work of the last chapter, the `add` command looks like this:

```
# bin/jit

when "add"
  root_path = Pathname.new(Dir.getwd)
  git_path  = root_path.join(".git")

  workspace = Workspace.new(root_path)
  database  = Database.new(git_path.join("objects"))
  index     = Index.new(git_path.join("index"))

  begin
    index.load_for_update
    rescue Lockfile::LockDenied => error
      $stderr.puts <<~ERROR
        fatal: #{error.message}

        Another jit process seems to be running in this repository.
        Please make sure all processes are terminated then try again.
        If it still fails, a jit process may have crashed in this
        repository earlier: remove the file manually to continue.
      ERROR
      exit 128
  end

  begin
    paths = ARGV.flat_map do |path|
      path = Pathname.new(File.expand_path(path))
      workspace.list_files(path)
    end
    rescue Workspace::MissingFile => error
      $stderr.puts "fatal: #{error.message}"
      index.release_lock
      exit 128
  end

  begin
    paths.each do |path|
      data = workspace.read_file(path)
      stat = workspace.stat_file(path)

      blob = Database::Blob.new(data)
      database.store(blob)
      index.add(path, blob.oid, stat)
    end
    rescue Workspace::NoPermission => error
      $stderr.puts "error: #{error.message}"
      $stderr.puts "fatal: adding files failed"
```

```
    index.release_lock
    exit 128
end

index.write_updates
exit 0
```

This is quite a lot of logic to have sitting around in one big undifferentiated lump. It will be much easier to maintain this code if we start to discover some new abstractions in it and break it into smaller pieces. It should also become easier to test; the code is currently coupled to global objects like ENV, ARGV and the standard I/O streams, and it invokes exit which will end any Ruby process that invokes it. Removing these couplings will mean we can control the code from a test suite and get some automated checking in place for what is becoming a complex piece of logic.

8.1. Abstracting the repository

The `init` command is unusual among Git commands in that it's run to create a repository rather than to work with an existing one. But the others, `add` and `commit`, have some similar-looking boilerplate code to set up the objects they need to work with.

Here's the start of the `add` command:

```
# bin/jit

when "add"
  root_path = Pathname.new(Dir.getwd)
  git_path  = root_path.join(".git")

  workspace = Workspace.new(root_path)
  database  = Database.new(git_path.join("objects"))
  index     = Index.new(git_path.join("index"))
```

And here's `commit`:

```
# bin/jit

when "commit"
  root_path = Pathname.new(Dir.getwd)
  git_path  = root_path.join(".git")

  database = Database.new(git_path.join("objects"))
  index   = Index.new(git_path.join("index"))
  refs    = Refs.new(git_path)
```

As we add more commands, it would be good not to keep copy-pasting this boilerplate into each one, duplicating the information about where various pieces of the infrastructure are located in the filesystem. Let's create an abstraction to tie all the pieces together and encapsulate this knowledge: the `Repository` class.

Instead of the above boilerplate, what's the minimum amount of code we could replace it with? In order to figure out the paths to all the Git components, the only thing we need to know is the path to the `.git` directory, which is inside the directory that `jit` is invoked from. We can instantiate a `Repository` object with that path and use it to build all the other pieces.

```
# bin/jit
```

```
root_path = Pathname.new(Dir.getwd)
repo = Repository.new(root_path.join(".git"))
```

The commands will now use this `repo` object to get all other objects they want to talk to. For example, here's a section of the `commit` command that loads the index and writes its contents as a tree to the database.

```
# bin/jit

repo.index.load

root = Database::Tree.build(repo.index.each_entry)
root.traverse { |tree| repo.database.store(tree) }
```

The `Repository` class does not contain any logic per se; it just provides an interface for getting a `Database`, `Index`, `Refs` and `Workspace`, and it knows the filesystem locations each of these should be bound to.

Each method uses *memoization*¹, not for performance but for consistency. Because some of these objects are stateful, like `Index`, we want to make sure that each time we call `repo.index` we are getting the same `Index` instance. In Ruby this is done by caching the result of the first call to `Repository#index` with the guarded assignment `@index ||=`.

```
# lib/repository.rb

require_relative "./database"
require_relative "./index"
require_relative "./refs"
require_relative "./workspace"

class Repository
  def initialize(git_path)
    @git_path = git_path
  end

  def database
    @database ||= Database.new(@git_path.join("objects"))
  end

  def index
    @index ||= Index.new(@git_path.join("index"))
  end

  def refs
    @refs ||= Refs.new(@git_path)
  end

  def workspace
    @workspace ||= Workspace.new(@git_path.dirname)
  end
end
```

This may well look like overkill: in order to save about 3 lines of code in each command, we've written a 25-line class. However, the important thing is not the physical volume of code, it's

¹<https://en.wikipedia.org/wiki/Memoization>

how many times the same fact is stated in the codebase. If every single command repeated the code that said, ‘the database is stored in `.git/objects`’, then we’d have to change a lot of code if we ever change our mind or want to change how the database works. The introduction of the `Repository` class means this bit of knowledge only exists once in the codebase, and so it makes the program easier to change. Every time multiple pieces of your program have an implicit agreement that’s not enforced, that’s an opportunity for a maintainer to change one piece of code without changing others that depend on it. This, rather than eliminating literal duplication of lines of code, is the essence of the DRY principle².

8.2. Commands as classes

The other thing that’s noticeable about `bin/jit` is how long it’s getting. The prospect of it continuing to be one big case expression that contains the code of every single command that `jit` responds to is not an appealing one; it will make the code harder to maintain over time, it will make diffs less useful and merge conflicts more likely, and it also makes it harder to test all the code that’s in that file. A good step would be to make the `jit` executable as small as possible, so that it just invokes a class containing the code for the command it wants to run. This should result in all the logic being moved to places where it can be tested.

As a first step towards this, let’s just move all the command code as-is into three new classes: `Command::Init`, `Command::Add` and `Command::Commit`. The `bin/jit` executable will be reduced to calling `Command.execute` with the first element of `ARGV` to invoke one of these classes, plus a little error-handling code.

```
# bin/jit

#!/usr/bin/env ruby

require_relative "../lib/command"

begin
  name = ARGV.shift
  Command.execute(name)

rescue Command::Unknown => error
  $stderr.puts "jit: #{error.message}"
  exit 1

rescue => error
  $stderr.puts "fatal: #{error.message}"
  if ENV["DEBUG"]
    error.backtrace.each do |line|
      $stderr.puts "        from #{line}"
    end
  end
  exit 1
end
```

Here we’re handling one specific kind of error that we expect: what happens when the given command name does not exist. I’ve also put in a catch-all rescue for any other errors so that they’re handled nicely, including printing the full stack trace³ only if the environment variable

²https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

³https://en.wikipedia.org/wiki/Stack_trace

DEBUG is set. That means end users won't see this information but a maintainer can expose it if desired.

Next we need to define this new Command.execute method. This will be a small wrapper that selects one of the classes from the Command namespace to invoke based on the name it's given. If the name is not recognised it raises an error, otherwise it instantiates the selected class and calls its run method.

```
# lib/command.rb

require_relative "./command/add"
require_relative "./command/commit"
require_relative "./command/init"

module Command
  Unknown = Class.new(StandardError)

  COMMANDS = {
    "init"  => Init,
    "add"   => Add,
    "commit" => Commit
  }

  def self.execute(name)
    unless COMMANDS.has_key?(name)
      raise Unknown, "'#{name}' is not a jit command."
    end

    command_class = COMMANDS[name]
    command_class.new.run
  end
end
```

The command classes themselves each contain nothing but a run method that houses the code extracted verbatim from bin/jit. I have not made any changes to the body of the command itself. For example, here is the Command::Commit class:

```
# lib/command/commit.rb

require "pathname"
require_relative "../repository"

module Command
  class Commit

    def run
      root_path = Pathname.new(Dir.getwd)
      repo = Repository.new(root_path.join(".git"))

      repo.index.load

      root = Database::Tree.build(repo.index.each_entry)
      root.traverse { |tree| repo.database.store(tree) }

      parent = repo.refs.read_head
      name   = ENV.fetch("GIT_AUTHOR_NAME")
      email  = ENV.fetch("GIT_AUTHOR_EMAIL")
```

```
author = Database::Author.new(name, email, Time.now)
message = $stdin.read

commit = Database::Commit.new(parent, root.oid, author, message)
repo.database.store(commit)
repo.refs.update_head(commit.oid)

is_root = parent.nil? ? "(root-commit) " : ""
puts "[#{ is_root }#{ commit.oid }] #{ message.lines.first }"
exit 0
end

end
end
```

The `Command::Init#run` and `Command::Add#run` methods contain the bodies of the `init` and `add` commands from `bin/jit` respectively. This a marginal improvement in that each command is now stored in its own file and so will be a bit easier to locate and maintain. However, the code is still hard to test and we must fix that before making further changes.

8.2.1. Injecting dependencies

Having the command code in classes that we can call lets us write scripts to test their functionality, but it's still rife with problems. To demonstrate some of them, here's a script that tries to initialise a new repository, create a file in it, add that file to the index, and then list the contents of the index. If you're thinking this is hard to follow, that's normal — we'll examine why this is below.

```
# test_add.rb

require "fileutils"
require "pathname"

require_relative "./lib/command/init"
require_relative "./lib/command/add"
require_relative "./lib/index"

def exit(*)
end

repo_dir = File.expand_path("../tmp-repo", __FILE__)

ARGV.clear
ARGV << repo_dir
Command::Init.new.run

path = File.join(repo_dir, "hello.txt")
flags = File::RDWR | File::CREAT
File.open(path, flags) { |file| file.puts("hello") }

ARGV.clear
ARGV << "hello.txt"
 FileUtils.cd(repo_dir) { Command::Add.new.run }

index_path = Pathname.new(repo_dir).join(".git", "index")
index = Index.new(index_path)
```

```
index.load
index.each_entry do |entry|
  puts [entry.mode.to_s(8), entry.oid, entry.path].join(" ")
end

 FileUtils.rm_rf(repo_dir)
```

Let's run this file and see what it prints out.

```
$ ruby test_add.rb
Initialized empty Jit repository in /Users/jcoglan/jit/tmp-repo/.git
100644 ce013625030ba8dba906f756967f9e9ca394464a hello.txt
```

This looks fine; the index contains the file we added with the right mode and object ID. However, the test script itself is full of quirks that would be good to get rid of.

The first of these is that we're overwriting the `exit` function with the line `def exit(*)` near the top of the file. That's because each command internally calls `exit` with the status it should exit with, but because that code is in the same Ruby process as this test script, the entire process exits. If we didn't override this method, then nothing after the call to `Command::Init.new.run` would execute.

The next problem is that we're manipulating the global argument array `ARGV`. This contains the arguments to the current invocation of `ruby`, but to make our commands work, we need to empty it out and fill it with the arguments we want the command to see. Not only is this awkward, it might break other code. If we're running this code inside a test suite that uses command-line arguments, changing `ARGV` might break its behaviour. In general, it's not a good idea to mutate process globals like this from deep inside application code.

Another global setting we're changing is the Ruby process's working directory. The `add` command finds the root of the current repository using `Dir.getwd`⁴, so we need to change what this returns so the `add` command is executed in the temporary test repository, not the main one containing the Jit source code. This is done using `FileUtils.cd`⁵, which changes the current working directory for the duration of the block passed to it. Again, changing global process state can affect other code running within the process. For example, something that's writing to a log file while this code runs might begin writing to the wrong path. It's better to explicitly use absolute paths as inputs to components that need to use the filesystem, rather than changing global state.

Then we have the problem of handling output from these commands. Running the `Command::Init` class has written `Initialized empty Jit repository...` to the terminal, before our own output that displays the contents of the index. In general, we don't want runtime output from the commands intermingled with output from the test suite — it makes the test results harder to read, and it prevents us from using code in the tests to inspect what each command prints.

A similar problem presents itself when we need to control the standard input stream to each command. Here's an extra snippet that runs the `commit` command:

⁴<https://docs.ruby-lang.org/en/2.3.0/Dir.html#method-c-getwd>
⁵<https://docs.ruby-lang.org/en/2.3.0/FileUtils.html#method-c-cd>

```
ARGV.clear
ENV["GIT_AUTHOR_NAME"] = "A. U. Thor"
ENV["GIT_AUTHOR_EMAIL"] = "author@example.com"
$stdin = StringIO.new("Commit message.")
FileUtils.cd(repo_dir) { Command::Commit.new.run }
$stdin = STDIN
```

We're mutating the `ENV` global because `Command::Commit` gets author information from it, and it's a problem for similar reasons that mutating `ARGV` and `Dir.getwd` is. We're also messing with the process's input stream, temporarily reassigning `$stdin` to a `StringIO`⁶ containing the data we want the command to read, before setting it back to the real input stream `STDIN`. Ruby provides global variables — `$stdin`, `$stdout`, `$stderr` — rather than constants for its I/O streams, but replacing them still carries the risk that you're affecting the entire Ruby process, not just the component we're testing.

All these problems can be solved by decoupling the `Command` classes from all the global references they're using. Rather than reaching out into the global namespace for `ARGV`, for example, a command must have the command-line arguments explicitly passed into it. Let's change `bin/jit` so that it passes in all the process components the commands will need. It also expects to receive an object back that contains the exit status.

```
# bin/jit

cmd = Command.execute(Dir.getwd, ENV, ARGV, $stdin, $stdout, $stderr)
exit cmd.status
```

The `Command.execute` method now uses the first element of the argument list to select a command, and reserves the rest of the list for inputs to that command, without mutating `ARGV`. It instantiates the chosen command by passing all the process components into it, and it also returns the command object so that `bin/jit` can read its status.

```
# lib/command.rb

def self.execute(dir, env, argv, stdin, stdout, stderr)
  name = argv.first
  args = argv.drop(1)

  unless COMMANDS.has_key?(name)
    raise Unknown, "'#{name}' is not a jit command."
  end

  command_class = COMMANDS[name]
  command = command_class.new(dir, env, args, stdin, stdout, stderr)

  command.execute
  command
end
```

To complete the change, we'll introduce a new class called `Command::Base`, which all the other commands will inherit from. Its constructor takes all the process components and stores them as instance variables.

```
# lib/command/base.rb
```

⁶<https://docs.ruby-lang.org/en/2.3.0/StringIO.html>

```
module Command
  class Base

    attr_reader :status

    def initialize(dir, env, args, stdin, stdout, stderr)
      @dir = dir
      @env = env
      @args = args
      @stdin = stdin
      @stdout = stdout
      @stderr = stderr
    end

  end
end
```

The commands can now be changed to refer to these instance variables instead of the globals they were using. `Dir.getwd` becomes `@dir`, `ENV` becomes `@env`, `ARGV` becomes `@args`, and so on for the I/O streams. For example, here's the new state of the `Commit` command:

```
# lib/command/commit.rb

require "pathname"

require_relative "../base"
require_relative "../repository"

module Command
  class Commit < Base

    def run
      root_path = Pathname.new(@dir)
      repo = Repository.new(root_path.join(".git"))

      repo.index.load

      root = Database::Tree.build(repo.index.each_entry)
      root.traverse { |tree| repo.database.store(tree) }

      parent = repo.refs.read_head
      name = @env.fetch("GIT_AUTHOR_NAME")
      email = @env.fetch("GIT_AUTHOR_EMAIL")
      author = Database::Author.new(name, email, Time.now)
      message = @stdin.read

      commit = Database::Commit.new(parent, root.oid, author, message)
      repo.database.store(commit)
      repo.refs.update_head(commit.oid)

      is_root = parent.nil? ? "(root-commit) " : ""
      puts "[#{is_root}#{commit.oid}] #{message.lines.first}"
      exit 0
    end
  end
end
```

There are still a few minor wrinkles to smooth out. The `File.expand_path` method expands paths relative to the Ruby process's current working directory, unless a different one is specified. We must expand arguments relative to the command's own directory instead:

```
# lib/command/init.rb

root_path = Pathname.new(File.expand_path(path, @dir))
```

In fact, this is a common operation needed in several commands, so let's add it to the `Base` class.

```
# lib/command/base.rb

def expanded_pathname(path)
  Pathname.new(File.expand_path(path, @dir))
end
```

Next, the existing `puts` calls with no receiver (as opposed to calls like `@stderr.puts`) in these classes will write to the Ruby process's real standard output if left unchanged. We don't need to change the calls themselves; instead, we can redefine `puts` in `Command::Base`, then any call to it from one of the command classes will use this new definition. Our version will write to the `@stdout` object stored by the command.

```
# lib/command/base.rb

def puts(string)
  @stdout.puts(string)
end
```

Finally, there's the problem of calling `exit` inside these `Command` classes. It would be nice if calling `exit` within a command stopped the command executing any further, but did not exit the Ruby process. We can use Ruby's throw/catch mechanism for this. In `Command::Base` we'll define `exit` to store the exit code for the command, and then call `throw :exit` — this will halt further execution of the call stack that lead to the `throw`.

```
# lib/command/base.rb

attr_reader :status

def exit(status = 0)
  @status = status
  throw :exit
end
```

To tie things together, we need a method that will catch the `:exit` signal. The `Command::Base#execute` method does just that: it calls the `run` method in a `catch` block, so that the effect is to execute the code in `run` up to the point that `exit` is called.

```
# lib/command/base.rb

def execute
  catch(:exit) { run }
end
```

If you look back at the `Command.execute` method, you'll see it calls `execute` rather than `run` on the command object it creates, so the command's code runs until it calls `exit`. By returning the command object to the caller, we can get the command's status code.

The technique we've used above where `bin/jit` passes in all the things that `Command` will need, and `Command` then uses these values rather than global references, is known as *dependency injection*⁷ and it's a simple yet powerful technique for decoupling program components, making them easier to reuse and test. We can now change our test script to use the new `Command.execute` interface — notice how we no longer need to redefine `exit`, mutate `ARGV`, or use `FileUtils.cd`.

```
require "fileutils"
require "pathname"

require_relative "./lib/command"
require_relative "./lib/index"

repo_dir = File.expand_path("../tmp-repo", __FILE__)
io = StringIO.new

cmd = Command.execute(Dir.getwd, {}, ["init", repo_dir], io, io, io)
puts "init: #{cmd.status}"

path = File.join(repo_dir, "hello.txt")
flags = File::RDWR | File::CREAT
File.open(path, flags) { |file| file.puts("hello") }

cmd = Command.execute(repo_dir, {}, ["add", "hello.txt"], io, io, io)
puts "add: #{cmd.status}"

index_path = Pathname.new(repo_dir).join(".git", "index")
index = Index.new(index_path)
index.load
index.each_entry do |entry|
  puts [entry.mode.to_s(8), entry.oid, entry.path].join(" ")
end

 FileUtils.rm_rf(repo_dir)
```

This now expresses the syntax of each command more directly, lets us inspect the exit status of each one easily, and doesn't produce any extraneous output such as the message printed by `init`.

```
$ ruby test_add.rb
init: 0
add: 0
100644 ce013625030ba8dba906f756967f9e9ca394464a hello.txt
```

Testing the `commit` command still requires about the same amount of setup as before, but everything it uses is a local variable rather than a global object:

```
env = {
  "GIT_AUTHOR_NAME" => "A. U. Thor",
  "GIT_AUTHOR_EMAIL" => "author@example.com"
}
input = StringIO.new("Commit message.")
cmd = Command.execute(repo_dir, env, ["commit"], input, io, io)
puts "commit: #{cmd.status}"
```

⁷https://en.wikipedia.org/wiki/Dependency_injection

The `Command::Commit` class now only has this small `env` hash to work with, rather than the `ENV` global that contains all the variables in your current environment. That means you can be much more explicit about what the command depends on, and make sure it's not accidentally depending on a variable that happens to be set in your environment but isn't stated in the test. We also didn't need to rebind the standard input stream for the whole Ruby process.

Dependency injection doesn't necessarily make things simpler in the sense of there being less code. What it does is remove implicit couplings between system components in a way that makes it easier to isolate each component from the global environment. It means you can be more confident that you're really testing the right thing, because there are fewer hidden assumptions, and it also means your components are less likely to leak side effects into the rest of the system or the process running your tests.

8.3. Testing the commands

The changes in the previous section have made it much easier to write automated tests for the `Command` classes. But rather than ad-hoc scripts as we saw above, let's add tests to our automated minitest suite so they're run every time we run `rake`.

Let's begin with a single test to help us work out a bunch of support code. I'd like to write a test that says, when I call `add` with a single file, that file gets added to the index. I'd like to be able to say this as tersely as possible, because the easier it is to write tests, the more I'm likely to write, and the more robust the codebase will be as a result.

Here's my first attempt at writing this test:

```
# test/command/add_test.rb

require "minitest/autorun"
require "command_helper"

describe Command::Add do
  include CommandHelper

  def assert_index(expected)
    repo.index.load
    actual = repo.index.each_entry.map { |entry| [entry.mode, entry.path] }
    assert_equal expected, actual
  end

  it "adds a regular file to the index" do
    write_file "hello.txt", "hello"

    jit_cmd "add", "hello.txt"

    assert_index [[0100644, "hello.txt"]]
  end
end
```

This test uses several high-level functions to drive the activity, and many of them are not defined directly in this file. Instead, I'm using this example to drive out a set of support code that I'm likely to need for testing other commands. `repo` should return me a `Repository` object tied to some temporary directory used for testing, `write_file` should write some content to a file, and `jit_cmd` runs a Jit command with the given arguments, capturing its output for later inspection.

These methods are defined in the `CommandHelper` module that's included at the top of the test. Below is an initial definition of that module that contains the required methods. `repo_path` returns a path inside the test directory, where we'll initialise a temporary repository for the duration of each test. Just like the methods in `Repository` itself, `repo` is memoised in case we need to use it to access stateful objects. `jit_cmd` creates three new I/O stream objects and passes them into the `Command.execute` method, so that anything written to them can be inspected after we call the command.

At the top of the module we define the `included` method, which Ruby executes whenever the `CommandHelper` is included into another class. It adds a before hook to the including test suite to initialise a temporary repository using the `init` command, and an after hook to delete this directory. These hooks mean that every test begins with a new empty repository and we get a clean slate on top of which to build each test scenario.

```
# test/command_helper.rb

require "fileutils"
require "pathname"

require "command"
require "repository"

module CommandHelper
  def self.included(suite)
    suite.before { jit_cmd "init", repo_path.to_s }
    suite.after { FileUtils.rm_rf(repo_path) }
  end

  def repo_path
    Pathname.new(File.expand_path("../test-repo", __FILE__))
  end

  def repo
    @repository ||= Repository.new(repo_path.join(".git"))
  end

  def write_file(name, contents)
    path = repo_path.join(name)
    FileUtils.mkdir_p(path.dirname)

    flags = File::RDWR | File::CREAT | File::TRUNC
    File.open(path, flags) { |file| file.write(contents) }
  end

  def jit_cmd(*argv)
    @stdin = StringIO.new
    @stdout = StringIO.new
    @stderr = StringIO.new

    @cmd = Command.execute(repo_path.to_s, {}, argv, @stdin, @stdout, @stderr)
  end
end
```

With this infrastructure in place we can think about the other use cases for the `add` command. We've already tested some of its behaviour in Section 7.3, “Stop making sense”, so we know what the in-memory behaviour of the `Index` class is when different filenames are added to it.

What we want to know now is how everything works together: does the command-line interface work, and does it integrate with the filesystem properly?

For example, we should check that it stores the fact that some files are executable:

```
# test/command/add_test.rb

it "adds an executable file to the index" do
  write_file "hello.txt", "hello"
  make_executable "hello.txt"

  jit_cmd "add", "hello.txt"

  assert_index [[0100755, "hello.txt"]]
end
```

This necessitates another helper method in `CommandHelper`:

```
# test/command_helper.rb

def make_executable(name)
  File.chmod(0755, repo_path.join(name))
end
```

We can check that `add` accepts multiple filenames and adds them to the index, and that if we call `add` twice with a different file each time, then the files are added incrementally. The second test implicitly asserts that the entries are kept ordered by filename, and that the `.git/index.lock` file is removed at the end — if it were not, the second invocation of `add` should fail.

```
# test/command/add_test.rb

it "adds multiple files to the index" do
  write_file "hello.txt", "hello"
  write_file "world.txt", "world"

  jit_cmd "add", "hello.txt", "world.txt"

  assert_index [[0100644, "hello.txt"], [0100644, "world.txt"]]
end

it "incrementally adds files to the index" do
  write_file "hello.txt", "hello"
  write_file "world.txt", "world"

  jit_cmd "add", "world.txt"

  assert_index [[0100644, "world.txt"]]

  jit_cmd "add", "hello.txt"

  assert_index [[0100644, "hello.txt"], [0100644, "world.txt"]]
end
```

Then we have the ability to pass directory names to `add`. These tests make sure that we can pass a directory name to add the files inside it, and that the special directory name `.` is allowed to specify adding everything inside the project root.

```
# test/command/add_test.rb
```

```
it "adds a directory to the index" do
  write_file "a-dir/nested.txt", "content"

  jit_cmd "add", "a-dir"

  assert_index [[0100644, "a-dir/nested.txt"]]
end

it "adds the repository root to the index" do
  write_file "a/b/c/file.txt", "content"

  jit_cmd "add", "."

  assert_index [[0100644, "a/b/c/file.txt"]]
end
```

We've tested the effect of the add command — that it modifies the index — but not what its outputs are. We ought to check that when successful, the command exits with status 0 and does not print anything to stdout or stderr.

```
# test/command/add_test.rb

it "is silent on success" do
  write_file "hello.txt", "hello"

  jit_cmd "add", "hello.txt"

  assert_status 0
  assert_stdout ""
  assert_stderr ""
end
```

Again, this requires some new helpers. We can use the information stored away by the jit_cmd helper to inspect its exit status and what it prints to the output streams.

```
# test/command_helper.rb

def assert_status(status)
  assert_equal(status, @cmd.status)
end

def assert_stdout(message)
  assert_output(@stdout, message)
end

def assert_stderr(message)
  assert_output(@stderr, message)
end

def assert_output(stream, message)
  stream.rewind
  assert_equal(message, stream.read)
end
```

It's also important to check what happens in error scenarios, especially because these are situations we don't expect to happen often in our day-to-day work and so it's easy to overlook

them. Code that's not regularly exercised can easily accumulate bugs, and in error cases this can result in disastrous things happening to the user's data.

Let's add some tests for the three error cases we've considered: missing files, unreadable files, and a locked index. In each case we check what's written to stderr, we check the exit status, and we make sure nothing is written to the index itself.

```
# test/command/add_test.rb

it "fails for non-existent files" do
  jit_cmd "add", "no-such-file"

  assert_stderr <<~ERROR
    fatal: pathspec 'no-such-file' did not match any files
  ERROR
  assert_status 128
  assert_index []
end

it "fails for unreadable files" do
  write_file "secret.txt", ""
  make_unreadable "secret.txt"

  jit_cmd "add", "secret.txt"

  assert_stderr <<~ERROR
    error: open('secret.txt'): Permission denied
    fatal: adding files failed
  ERROR
  assert_status 128
  assert_index []
end

it "fails if the index is locked" do
  write_file "file.txt", ""
  write_file ".git/index.lock", ""

  jit_cmd "add", "file.txt"

  assert_status 128
  assert_index []
end
```

There are now enough tests here for me to feel confident in changing the code, knowing I won't break any of this functionality. That doesn't mean they're *complete* — there are certainly other things we could add tests for, and you should think through what other assumptions the code makes that aren't expressed here. Testing is all about being able to change code with confidence, and how much test coverage you need to feel confident will depend on the person or team working with the code, and what the risks of making mistakes are. Personally, I tend to be a little forgetful, and so I like to have all the important requirements captured as tests so I don't need to remember to manually check everything on each change.

Testing is also limited by what you're able to automate. For example, one thing I've not tested above is that the added files are written to the database. That's because we've not yet written any code for reading the `.git/objects` database. We will need such code as we build up

more commands, and we can come back and expand these tests when we have more tools for inspecting the repository.

8.4. Refactoring the commands

Now that we have some more tests in place we can *refactor*⁸ the code, changing its structure without changing its behaviour. The tests make sure that we don't change its behaviour in any significant way, and within the constraints imposed by the tests we can make any change we like. There are a couple of things I'd like to improve right now: there's some duplication between commands, and the `Command::Add#run` is very long and hard to follow.

8.4.1. Extracting common code

This is a very minor thing, but both the `Command::Add#run` and `Command::Commit#run` methods both begin with the same code:

```
root_path = Pathname.new(@dir)
repo = Repository.new(root_path.join(".git"))
```

This is much less boilerplate than we would have had before the changes we made in Section 8.1, “Abstracting the repository”, but it's still wasteful and error-prone to have every command class repeat the instructions for finding the right directory and creating a `Repository` around it.

Now that the commands inherit from `Command::Base`, we can simply replace the local variable `repo` with a method call by defining an appropriate method in the base class.

```
# lib/command/base.rb

def repo
  @repo ||= Repository.new(Pathname.new(@dir).join(".git"))
end
```

This is one nice thing about working in Ruby — because parentheses on method calls are optional, it's easy to replace a local variable with a method call of the same name without needing to change any of the subsequent code. It makes it easy to extract parts of functions and break them into smaller pieces.

8.4.2. Reorganising the add command

The main reason I wanted to add all the tests for the `add` command in Section 8.3, “Testing the commands” is that the `Command::Add#run` method has become a sprawling mess. In the process of adding error handling in Section 7.4, “Handling bad inputs”, we turned the `add` command code from a short easy-to-follow script into a lengthy page of code that doesn't tell a very clear story. We saw the current state of the `add` command at the beginning of this chapter, and the only difference is that it's now been moved into a class.

There are far too many things going on all in one function here. There's the main functionality of the command — reading files, storing them as blobs, adding them to the index — but it's mixed in with lots of error-handling in various places, including copy for each kind of error that really gets in the way when trying to understand the main purpose of the method. It becomes

⁸https://en.wikipedia.org/wiki/Code_refactoring

impossible to see important details about it, such as the fact that the whole process is bookended by claiming a lock on the index and writing any updates back to disk.

Let's break the method into smaller pieces so that it becomes easier to follow. We can begin by breaking the error-handling code out into separate methods, and moving all the rescue clauses to the end of the `run` method rather than having multiple `begin/rescue` blocks throughout the method. This makes it easier to see and change how each error is handled without cluttering the main method body.

```
# lib/command/add.rb

LOCKED_INDEX_MESSAGE = <<~MSG
  Another jit process seems to be running in this repository.
  Please make sure all processes are terminated then try again.
  If it still fails, a jit process may have crashed in this
  repository earlier: remove the file manually to continue.
~MSG

def run
  # [main method body elided]

rescue Lockfile::LockDenied => error
  handle_locked_index(error)
rescue Workspace::MissingFile => error
  handle_missing_file(error)
rescue Workspace::NoPermission => error
  handle_unreadable_file(error)
end

private

def handle_locked_index(error)
  @stderr.puts "fatal: #{error.message}"
  @stderr.puts LOCKED_INDEX_MESSAGE
  exit 128
end

def handle_missing_file(error)
  @stderr.puts "fatal: #{error.message}"
  repo.index.release_lock
  exit 128
end

def handle_unreadable_file(error)
  @stderr.puts "error: #{error.message}"
  @stderr.puts "fatal: adding files failed"
  repo.index.release_lock
  exit 128
end
```

The `run` method's main body is now reduced to the following:

```
# lib/command/add.rb

repo.index.load_for_update
```

```
paths = @args.flat_map do |path|
  path = expanded_pathname(path)
  repo.workspace.list_files(path)
end

paths.each do |path|
  data = repo.workspace.read_file(path)
  stat = repo.workspace.stat_file(path)

  blob = Database::Blob.new(data)
  repo.database.store(blob)
  repo.index.add(path, blob.oid, stat)
end

repo.index.write_updates
exit 0
```

This is considerably clearer than it was before, but we can break it up even further so that all `run` does is tell a high-level story and all the detail happens elsewhere:

```
# lib/command/add.rb

def run
  repo.index.load_for_update
  expanded_paths.each { |path| add_to_index(path) }
  repo.index.write_updates
  exit 0
rescue Lockfile::LockDenied => error
  handle_locked_index(error)
rescue Workspace::MissingFile => error
  handle_missing_file(error)
rescue Workspace::NoPermission => error
  handle_unreadable_file(error)
end

private

def expanded_paths
  @args.flat_map do |path|
    repo.workspace.list_files(expanded_pathname(path))
  end
end

def add_to_index(path)
  data = repo.workspace.read_file(path)
  stat = repo.workspace.stat_file(path)

  blob = Database::Blob.new(data)
  repo.database.store(blob)
  repo.index.add(path, blob.oid, stat)
end
```

This is as terse as I'd like this method to be: `run` loads the index, expands all the paths it's given, adds each one to the index, and writes the index.

To check we've not made any mistakes and introduced bugs by moving the code around, we can run the tests that we wrote in Section 8.3, "Testing the commands".

```
$ rake
Run options: --seed 48441

# Running:
.....
Finished in 0.057807s, 242.1852 runs/s, 380.5767 assertions/s.

14 runs, 22 assertions, 0 failures, 0 errors, 0 skips
```

Everything seems to be working, so we'll commit this and move on to adding more features. It's worth stopping at regular intervals while working on a project to check that the code hasn't got into a state that's going to make it harder for you to keep making progress. Make sure the code is covered by tests that capture its external behaviour, and then you can restructure it to suit your needs without worrying about accidentally adding bugs.

9. Status report

Over the course of the last few chapters, we've focussed on the `add` command that builds `.git/index`, a cache of blobs added to the database that is used to build the next commit. Although we've written automated tests that use the `Index` interface to inspect the data stored on disk, the `jit` user interface doesn't yet offer any way to do this. Without any way to see how our work differs from the latest commit, we can't use the `add` command to its full potential and selectively add files to the next commit. `add` is reduced to being an extra step we're obliged to do just before each commit, rather than something we can use regularly during our workflow.

```
$ jit add .
$ echo "commit message" | jit commit
```

To give us visibility into the state of the repository, Git has the `status` command. It shows a summary of the differences between the tree of the `HEAD` commit, the entries in the index, and the contents of the workspace, as well as listing conflicting files during a merge. It can display these changes in multiple formats; the default is a human-readable format with differences highlighted in red and green, but there's also a machine-readable format that's activated by the `--porcelain` option. This format is designed to be nicer for automated scripts to consume, and that's what we'll be implementing first since it will make it easier to write tests.

We'll be using *test-driven development*¹ (TDD) to implement this feature. The `add` and `commit` commands, while having a few special-case behaviours, are fairly straightforward tools whose code we execute frequently while developing the codebase. We wrote tests for the edge cases but by-and-large if these commands were not working we would quickly notice. In contrast, a command like `status` needs to deal with lots of possible combinations of states, not all of which we'll just bump into in the normal course of our work. That means it's much more important to drive the requirements by writing tests for each different scenario we expect it to handle, so we can always be sure we've not broken some rarely-encountered state.

One element of practicing TDD is that one only writes as much code as is needed to pass the currently-failing test(s), never any more. If you find yourself anticipating a situation that should be handled, you stop and write a test for it first. In the early stages this often leads to code that seems obviously wrong, but the incremental progress it offers can be valuable in situations where fitting every permutation of a problem in one's head at once is too difficult. It provides focus by targeting one case at a time while providing reassurance by building up a set of tests to make sure you don't break existing code.

9.1. Untracked files

Let's begin by adding support for listing *untracked* files, that is, files that are not listed in the index. In an empty Git repository, creating a couple of files and then checking the status shows those two files in name order with the status `??`, which means they are untracked.

```
$ touch file.txt another.txt
$ git status --porcelain
?? another.txt
?? file.txt
```

¹https://en.wikipedia.org/wiki/Test-driven_development

We can simulate this using an automated test. In a new test suite, `test/command/status_test.rb`, we'll add a single test case that writes two files and then checks the status is as above.

```
# test/command/status_test.rb

require "minitest/autorun"
require "command_helper"

describe Command::Status do
  include CommandHelper

  def assert_status(output)
    jit_cmd "status"
    assert_stdout(output)
  end

  it "lists untracked files in name order" do
    write_file "file.txt", ""
    write_file "another.txt", ""

    assert_status <<~STATUS
      ?? another.txt
      ?? file.txt
    STATUS
  end
end
```

This initially fails because there is no command named `status`. So, we'll create one and add it to the list in the `Command` module:

```
# lib/command.rb

COMMANDS = {
  "init"  => Init,
  "add"   => Add,
  "commit" => Commit,
  "status" => Status
}
```

The `Command::Status` class will at first do very little. Remember, we're approaching the code test-first, and we should focus only on making the currently failing test pass, not on implementing a complete solution all at once. The simplest thing that could make the current test pass is simply to list the contents of the workspace with `??` in front of each name.

```
# lib/command/status.rb

require_relative "../base"

module Command
  class Status < Base

    def run
      repo.workspace.list_files.sort.each do |path|
        puts "?? #{ path }"
      end
    end

    exit 0
  end
end
```

```
    end  
  
end
```

That makes the test suite pass, but it's obviously wrong. If we run `git status` on the project right now, it lists everything as untracked! It also sorts the files slightly incorrectly, because `Workspace#list_files` returns `Pathname` objects, which sort the `.` and `/` characters differently from regular strings.

```
$ git status  
?? LICENSE.txt  
?? Rakefile  
?? bin/jit  
?? lib/command/add.rb  
?? lib/command/base.rb  
?? lib/command/commit.rb  
?? lib/command/init.rb  
?? lib/command/status.rb  
?? lib/command.rb  
?? lib/database/author.rb  
?? lib/database/blob.rb  
?? lib/database/commit.rb  
?? lib/database/tree.rb  
?? lib/database.rb  
?? lib/entry.rb  
?? lib/index/checksum.rb  
?? lib/index/entry.rb  
?? lib/index.rb  
?? lib/lockfile.rb  
?? lib/refspec.rb  
?? lib/repository.rb  
?? lib/workspace.rb  
?? test/command/add_test.rb  
?? test/command/status_test.rb  
?? test/command_helper.rb  
?? test/index_test.rb
```

This might be *wrong*, but it's still a step forward from not having a `status` command at all. That's the essence of TDD: to introduce new requirements one-by-one to guide the implementation towards a complete solution in small steps. Some steps will require very little work, as above, but they might help us realise what the next test we need is. In this case, we can see we should write a test for untracked files in a repository that has some files in the index.

9.1.1. Untracked files not in the index

A minimal test that will force us to keep indexed files out of the `status` output appears below. We write a single file called `committed.txt`, add it to the index, and then commit. We make a commit here rather than just adding the file to the index, because when we've finished the `status` implementation, any difference between the index and the last commit will cause additional output, so we want the index and `HEAD` to be the same.

Then, we add one more untracked file and check what `status` prints.

```
# test/command/status_test.rb
```

```
it "lists files as untracked if they are not in the index" do
  write_file "committed.txt", ""
  jit_cmd "add", "."
  commit "commit message"

  write_file "file.txt", ""

  assert_status <<~STATUS
    ?? file.txt
  STATUS
end
```

The `commit` helper function is a new addition to `CommandHelper`. It requires some extra methods for setting environment variables and preparing some content in `@stdin`. The `jit_cmd` method must also be modified to use the `@env` and `@stdin` variables that may have been set up by these new methods, providing some default values if not.

```
# test/command_helper.rb

def set_env(key, value)
  @env ||= {}
  @env[key] = value
end

def set_stdin(string)
  @stdin = StringIO.new(string)
end

def jit_cmd(*argv)
  @env ||= {}
  @stdin ||= StringIO.new
  @stdout = StringIO.new
  @stderr = StringIO.new

  @cmd = Command.execute(repo_path.to_s, @env, argv, @stdin, @stdout, @stderr)
end

def commit(message)
  set_env("GIT_AUTHOR_NAME", "A. U. Thor")
  set_env("GIT_AUTHOR_EMAIL", "author@example.com")
  set_stdin(message)
  jit_cmd("commit")
end
```

Now, this test fails with the following output. The diffs in minitest tend to be a little difficult to read, but this tells us that instead of a single line of output `?? file`, we're seeing two lines: `?? committed.txt` and `?? file.txt`. The file `committed.txt` must be excluded from the output.

```
1) Failure:
Command::Status#test_0001_lists files as untracked if they are not in the index
--- expected
+++ actual
@@ -1,2 +1,3 @@
-"??" file.txt
+"??" committed.txt
+??" file.txt
"
```

A minor amendment to `Command::Status` will fix this. To find the untracked files, we list all the files in `Workspace` and filter them based on whether the path is tracked in the index. We must load the index from disk before doing this, and because we're not going to modify the index, we call `Index#load` rather than `Index#load_for_update`.

```
# lib/command/status.rb

def run
  repo.index.load

  untracked = repo.workspace.list_files.reject do |path|
    repo.index.tracked?(path)
  end

  untracked.sort.each do |path|
    puts "??" #{ path }"
  end

  exit 0
end
```

The `Index#tracked?` method returns true if and only if the index contains an entry by the given name, that is if `@entries` contains the given pathname as a key.

```
# lib/index.rb

def tracked?(path)
  @entries.has_key?(path.to_s)
end
```

This simple check keeps `committed.txt` out of the status output, and the test passes.

9.1.2. Untracked directories

The functionality for listing untracked files is almost complete, there's just one optimisation that Git makes here. If a directory does not contain any files that appear in the index, then that directory is itself listed as untracked, rather than its contents. We can see this in the following example:

```
$ mkdir dir
$ touch file.txt dir/another.txt

$ git status --porcelain
?? dir/
?? file.txt
```

Let's reproduce this example as a test; the `write_file` helper takes care of creating any necessary directories for us. In order to exclude the obvious naive implementation of simply not listing anything inside directories, we also include a test to check that untracked files inside directories are listed, if that directory contains other tracked files, possibly inside other subdirectories. The second test below reproduces the following situation:

```
$ mkdir -p a/b
$ touch a/b/inner.txt
$ git add .
$ git commit --message "commit message"
```

```
$ mkdir a/b/c
$ touch a/outer.txt a/b/c/file.txt
$ tree
.
└── a
    ├── b
    │   └── c
    │       └── file.txt
    └── outer.txt

$ git status --porcelain
?? a/b/c/
?? a/outer.txt
```

The file `a/outer.txt` is listed as untracked, even though directory `a` does not itself directly contain any tracked files. The file `a/b/inner.txt` is tracked, and that means that directories `a` and `b` are considered tracked too. Directory `c` remains untracked because it does not, directly or indirectly, contain any tracked files.

Here are the above examples expressed as tests in Ruby.

```
# test/command/status_test.rb

it "lists untracked directories, not their contents" do
  write_file "file.txt", ""
  write_file "dir/another.txt", ""

  assert_status <<-STATUS
  ?? dir/
  ?? file.txt
  STATUS
end

it "lists untracked files inside tracked directories" do
  write_file "a/b/inner.txt", ""
  jit_cmd "add", "."
  commit "commit message"

  write_file "a/outer.txt", ""
  write_file "a/b/c/file.txt", ""

  assert_status <<-STATUS
  ?? a/b/c/
  ?? a/outer.txt
  STATUS
end
```

At the moment, they both fail. We must come up with a solution for the simple case of a top-level untracked directory without causing `status` to ignore tracked nested directories.

```
1) Failure:
Command::Status#test_0003_lists untracked directories, not their contents
--- expected
+++ actual
@@ -1,3 +1,3 @@
-"??" dir/
```

```
+"? dir/another.txt
 ?? file.txt
 ""

2) Failure:
Command::Status#test_0004_lists untracked files inside tracked directories
--- expected
+++ actual
@@ -1,3 +1,3 @@
-"? a/b/c/
+"? a/b/c/file.txt
 ?? a/outer.txt
"
```

This is going to require a little more work than our previous change, but some data structures we added to `Index` previously will help. First, we'll rewrite the `Command::Status#run` method. We'll still load the index, but the workflow after that is less direct. We create a `SortedSet` to hold the list of untracked files we find, then call a new method called `scan_workspace`, then we list out any untracked files that were added in the process.

```
# lib/command/status.rb

def run
  repo.index.load

  @untracked = SortedSet.new

  scan_workspace

  @untracked.each { |path| puts "? #{path}" }

  exit 0
end
```

The `Command::Status#scan_workspace` method is a recursive procedure. Rather than calling `Workspace#list_files` to recursively list everything in the project in a single call, we need to take control of the recursion ourselves. If we find a directory that is not tracked, then we can just add it to the `@untracked` set and not bother exploring it any further.

We'll do this by calling a new method `workspace#list_dir`, which as we'll see shortly will return the contents of a single directory, without recursing, as a set of filenames and associated `File::Stat` objects that describe each file. For each item in this set, if the path is tracked in the index then we call `scan_workspace` recursively, if the item is a directory. If the path is not tracked, then we add it to the `@untracked` set.

```
# lib/command/status.rb

def scan_workspace(prefix = nil)
  repo.workspace.list_dir(prefix).each do |path, stat|
    if repo.index.tracked?(path)
      scan_workspace(path) if stat.directory?
    else
      path += File::SEPARATOR if stat.directory?
      @untracked.add(path)
    end
  end
end
```

```
end
```

Our new method `Workspace#list_dir` works as follows. Given some directory name that's relative to the root of the project, we resolve it and list out its entries. For each entry we find, we calculate its relative path from the root of the project and call `File.stat` on it. We return the results of doing this as a hash.

```
# lib/workspace.rb

def list_dir(dirname)
  path    = @pathname.join(dirname || "")
  entries = Dir.entries(path) - IGNORE
  stats   = {}

  entries.each do |name|
    relative = path.join(name).relative_path_from(@pathname)
    stats[relative.to_s] = File.stat(path.join(name))
  end

  stats
end
```

For example, if we call it with `lib` then we get the following description of the contents of that directory, which will not be in any particular order.

```
>> workspace = Workspace.new(Pathname.new(Dir.getwd))

>> workspace.list_dir("lib")
=> {
  "lib/database"      => File::Stat(...),
  "lib/workspace.rb" => File::Stat(...),
  "lib/repository.rb" => File::Stat(...),
  "lib/entry.rb"       => File::Stat(...),
  "lib/refs.rb"        => File::Stat(...),
  "lib/database.rb"   => File::Stat(...),
  "lib/index"          => File::Stat(...),
  "lib/command"        => File::Stat(...),
  "lib/command.rb"     => File::Stat(...),
  "lib/lockfile.rb"    => File::Stat(...),
  "lib/index.rb"       => File::Stat(...)
}
```

The `File::Stat` objects returned by this call let us tell whether each item is a directory, what its mode and file size are, when it was last modified, and other useful information. It might seem expensive to call `stat()` on every file in the tree, but it cannot really be avoided. To recursively enumerate a directory, we need to check if each file is a directory, and that requires calling `stat()`. We may as well return the result of that `stat()` to the caller so they can make further use of it. In fact, the index allows us to do just that, as we'll see shortly. Even for untracked items, we need to know if it's a directory because we need to add a trailing slash to its name in the status output. At least skipping over untracked directories means we can skip calling `stat()` for all its children.

Finally, we need to adjust the `Index#tracked?` method so that it works with directories rather than just files. Fortunately, in Section 7.3.3, “Replacing a directory with a file”, we added a structure called `@parents` to the `Index` class that contains the name of every directory that

contains an indexed file, whether directly or indirectly. So, amending the `tracked?` method merely requires checking whether the name appears as a key in either the `@entries` hash or in `@parents`.

```
# lib/index.rb

def tracked?(path)
  @entries.has_key?(path.to_s) or @parents.has_key?(path.to_s)
end
```

With this change in place, all our tests are once again passing.

9.1.3. Empty untracked directories

There is one more edge case in how untracked directories are listed that unfortunately means it's not enough to simply skip any directory that's not in the index. To appear in the status output, a directory must contain, either directly or indirectly, a file that would be added to the index if you passed the directory to add. In the absence of support for `.gitignore` or `.git/info/exclude`, this just means the directory has to contain at least one file.

Here's an example to show this effect in action:

```
$ mkdir outer
$ git status --porcelain

$ mkdir outer/inner
$ git status --porcelain

$ touch outer/inner/file.txt
$ git status --porcelain
?? outer/
```

Note that it's `outer/`, not `outer/inner/` that's listed, so it's still the outermost untracked directory that should be printed. Let's reproduce these examples as tests:

```
# test/command/status_test.rb

it "does not list empty untracked directories" do
  mkdir "outer"

  assert_status ""
end

it "lists untracked directories that indirectly contain files" do
  write_file "outer/inner/file.txt", ""

  assert_status <<~STATUS
  ?? outer/
  STATUS
end
```

So far we've been using our `write_file` helper to create a file and implicitly create all its containing directories. We now need a helper to create directories themselves:

```
# test/command_helper.rb

def mkdir(name)
```

```
 FileUtils.mkdir_p(repo_path.join(name))
end
```

To make these test pass we can add an extra condition in `Command::Status#scan_workspace`, that uses a new method called `trackable_file?` to decide whether to add the untracked object to the results.

```
# lib/command/status.rb

def scan_workspace(prefix = nil)
  repo.workspace.list_dir(prefix).each do |path, stat|
    if repo.index.tracked?(path)
      scan_workspace(path) if stat.directory?
    elsif trackable_file?(path, stat)
      path += File::SEPARATOR if stat.directory?
      @untracked.add(path)
    end
  end
end
```

The `trackable_file?` method takes a file path and a `File::Stat` object, and returns true if the input represents either an untracked file, or a directory containing untracked files. If the input is a file, then we immediately return with the result of asking the `Index` if the file is not tracked. Otherwise, we use `Workspace#list_dir` to get the directory contents, and split them into two lists: one list of files and one of directories. We then check each of these lists to see if any of them contains a tracked file, using a recursive call to `trackable_file?`. Processing the files first means we can stop as soon as we find any untracked files in the tree, without recursing into its sibling directories any further.

```
# lib/command/status.rb

def trackable_file?(path, stat)
  return false unless stat

  return !repo.index.tracked?(path) if stat.file?
  return false unless stat.directory?

  items = repo.workspace.list_dir(path)
  files = items.select { |_, item_stat| item_stat.file? }
  dirs = items.select { |_, item_stat| item_stat.directory? }

  [files, dirs].any? do |list|
    list.any? { |item_path, item_stat| trackable_file?(item_path, item_stat) }
  end
end
```

This incurs a little extra inspection of the contents of untracked directories, but it still stops scanning the tree as soon as it finds any files, so we're still saving work by skipping nested untracked directories that won't add anything to the output.

It is worth noting that this routine will do a *depth-first search*²: the tree under each item in `dirs` will be fully explored before moving onto the next. However, Ruby's `Enumerable#any?`³

²https://en.wikipedia.org/wiki/Depth-first_search

³<https://docs.ruby-lang.org/en/2.3.0/Enumerable.html#method-i-any-3F>

returns true as soon as it finds a single element of the array for which the block returns true, so one directory containing a file will block its sibling directories from being explored. For tracked directories, every directory and file must be visited, so whether we do this depth- or *breadth-first*⁴ doesn't make much of a difference. But for a routine that stops as soon as it finds a particular thing, the search order can make a big difference. I've used depth-first here simply because it requires less code to implement.

9.2. Index/workspace differences

The next step is to implement what you might be used to seeing as the list of 'red' files when you run `git status`: paths where the workspace content differs from what's in the index. These are changes you have made since the last commit, but which have not been staged using the `add` command.

All the tests in this section will need to have something in the index to begin with, and that index will need to have been committed so that these tests don't produce any differences between `HEAD` and the index later. The starting point for all these tests will be equivalent to the state after running the following commands:

```
$ mkdir -p a/b
$ echo "one" > 1.txt
$ echo "two" > a/2.txt
$ echo "three" > a/b/3.txt
$ git add .
$ git commit --message "commit message"

$ tree
.
└── a
    ├── 1.txt
    └── b
        └── 3.txt

$ git status
On branch master
nothing to commit, working directory clean
```

In minitest, `describe` blocks can be nested. Each test inside a `describe` block runs the `before` blocks from all its containing `describe` blocks, from the outermost to the innermost. This provides both a way of organising a suite into sections, and incrementally building up a shared starting point that all the tests in a section share. To kick off this set of tests, we'll start a new `describe` block for index/workspace changes and reproduce the above sequence of commands in its `before` block. All the tests in this section will begin with this state set up.

```
# test/command/status_test.rb

describe "index/workspace changes" do
  before do
    write_file "1.txt", "one"
    write_file "a/2.txt", "two"
    write_file "a/b/3.txt", "three"
```

⁴https://en.wikipedia.org/wiki/Breadth-first_search

```
    jit_cmd "add", "."
    commit "commit message"
end
```

9.2.1. Changed contents

The most common kind of change we want to detect is that a file that's in the index has its contents changed in the workspace. Let's add a couple of tests, one that asserts that running `status` in a clean state produces no output, and one that changes two of the files and then checks they are listed with the status `M` in the output, reproducing the following example:

```
$ echo "changed" > 1.txt
$ echo "modified" > a/2.txt

$ git status --porcelain
M 1.txt
M a/2.txt
```

Notice that there's a leading space before the `M` in this example. The porcelain `status` output begins each row with two characters; the first indicates differences between HEAD and the index, the second between the index and the workspace. All the examples in this section will therefore have leading spaces, and in order to encode this in a `<<- heredoc`, the leading space must be escaped lest Ruby should mistake it for removable indentation. This actually serves to highlight the space, which could otherwise be overlooked by the reader.

Here are the two tests we'll start off with:

```
# test/command/status_test.rb

it "prints nothing when no files are changed" do
  assert_status ""
end

it "reports files with modified contents" do
  write_file "1.txt", "changed"
  write_file "a/2.txt", "modified"

  assert_status <<-STATUS
    \ M 1.txt
    \ M a/2.txt
  STATUS
end
```

The first test passes, but the second fails; we've not yet written any code that would produce such output.

```
1) Failure:
Command::Status::index/workspace changes#test_0002_reports files with modified contents
--- expected
+++ actual
@@ -1,3 +1,2 @@
- " M 1.txt
- M a/2.txt
-
+## encoding: ASCII-8BIT
```

```
+""
```

To make this work, we're going to need a couple of extra pieces of state. `@stats` will store a cache of `File::Stat` values for each file in the workspace, because we'll need them to compare files against the index. `@changed` is another `SortedSet` like `@untracked` that will store the names of files that differ from the index.

After calling the existing `scan_workspace` method, we'll call `detect_workspace_changes`, which will iterate over the index and compare its contents to data gathered during the `scan_workspace` phase. Once we've found all the changes, we print them out, using the prefix `M` for changed files and `??` for untracked ones as before.

```
# lib/command/status.rb

def run
  repo.index.load

  @stats      = {}
  @changed    = SortedSet.new
  @untracked = SortedSet.new

  scan_workspace
  detect_workspace_changes

  @changed.each { |path| puts " M #{ path }" }
  @untracked.each { |path| puts "?? #{ path }" }

  exit 0
end
```

The `scan_workspace` method needs a minor alteration: for files that are tracked, we add their `File::Stat` value to the `@stats` cache. This saves us calling `stat()` repeatedly on the same file; since the `status` command requires a complete comparison of the `HEAD` commit, index and workspace, we'd like to take every opportunity to reduce the number of system calls, especially those that read from disk. Accessing disk is slow compared to reading data out of memory, and it will improve our program's performance if we avoid doing so unnecessarily.

```
# lib/command/status.rb

def scan_workspace(prefix = nil)
  repo.workspace.list_dir(prefix).each do |path, stat|
    if repo.index.tracked?(path)
      @stats[path] = stat if stat.file?
      scan_workspace(path) if stat.directory?
    else
      path += File::SEPARATOR if stat.directory?
      @untracked.add(path)
    end
  end
end
```

`Command::Status#detect_workspace_changes` simply iterates over all the entries in the index and calls `check_index_entry` with each one. `check_index_entry` in turn compares the entry in the index to the `File::Stat` value from the workspace for the corresponding path. Recall that we're trying to detect files whose contents have changed. One cheap way of detecting that

is, if the index entry and the current file on disk have different sizes, the file contents must have changed.

```
# lib/command/status.rb

def detect_workspace_changes
  repo.index.each_entry { |entry| check_index_entry(entry) }
end

def check_index_entry(entry)
  stat = @stats[entry.path]
  @changed.add(entry.path) unless entry.stat_match?(stat)
end

# lib/index/entry.rb

def stat_match?(stat)
  size == 0 or size == stat.size
end
```

This is not a complete check for whether a file has changed, but we will flesh it out as we add more tests. This example shows one advantage of caching stat information in the index: it allows us to check whether files have changed without incurring the expense of reading all their contents.

With these additions in place, our tests pass, and jit can now faithfully show us its own status.

```
$ jit status
M lib/command/status.rb
M lib/index/entry.rb
M test/command/status_test.rb
```

Running the add command over all the changed files syncs up the index with the workspace, and afterward status does not print any differences.

```
$ jit add lib/test
$ jit status
```

9.2.2. Changed mode

What other kinds of changes should Git care about? Well, in its database the only information it stores about each file is its contents, which we've partially dealt with above, and its mode. The mode is another value that's captured by the `stat()` call, and so adding a check for it should be straightforward.

Here's a test that changes the mode of one of the indexed files, making it executable. The `status` command should report such files as modified.

```
# test/command/status_test.rb

it "reports files with changed modes" do
  make_executable "a/2.txt"

  assert_status <<~STATUS
  \ M a/2.txt
```

```
  STATUS  
end
```

As expected, this test fails because `status` produces no output — this change is not currently detected.

```
1) Failure:  
Command::Status::index/workspace changes#test_0003_reports files with changed modes  
--- expected  
+++ actual  
@@ -1,2 +1,2 @@  
- " M a/2.txt  
- "  
+# encoding: ASCII-8BIT  
+" "
```

A small change to `Index::Entry#stat_match?` should suffice, comparing the indexed and stored modes as well as the file sizes. This requires a little extra complexity though: Git doesn't store arbitrary file modes, it only stores 100644_8 and 100755_8 . So, rather than compare the indexed mode to the literal one read from disk, we need to calculate what mode the file would have if it were added to the index.

```
# lib/index/entry.rb  
  
def stat_match?(stat)  
  mode == Entry.mode_for_stat(stat) and (size == 0 or size == stat.size)  
end
```

To implement the mode check, we'll need to extract some logic currently contained in the `Index::Entry.create` method. This method takes a `File::Stat` and uses it to pick which mode to store for the file.

```
# lib/index/entry.rb  
  
class Index  
  REGULAR_MODE      = 0100644  
  EXECUTABLE_MODE  = 0100755  
  
  Entry = Struct.new(*entry_fields) do  
    def self.create(pathname, oid, stat)  
      mode = stat.executable? ? EXECUTABLE_MODE : REGULAR_MODE  
      # ...  
    end  
  end  
end
```

We'll extract this decision into `Index::Entry.mode_for_stat`:

```
# lib/index/entry.rb  
  
class Index  
  REGULAR_MODE      = 0100644  
  EXECUTABLE_MODE  = 0100755  
  
  Entry = Struct.new(*entry_fields) do  
    def self.create(pathname, oid, stat)  
      mode = Entry.mode_for_stat(stat)
```

```
# ...
end

def self.mode_for_stat(stat)
  stat.executable? ? EXECUTABLE_MODE : REGULAR_MODE
end
end
end
```

This small addition to `Index::Entry#stat_match?` has again got the tests passing, so we'll commit our work so far and move on to some more complicated cases.

9.2.3. Size-preserving changes

In Section 9.2.1, “Changed contents” we implemented a simple check to detect a file’s contents changing: if its size has changed, it must be different. But it’s possible to change a file’s contents without changing its size, and our code currently will not detect that. Here’s a test to force us to act:

```
# test/command/status_test.rb

it "reports modified files with unchanged size" do
  write_file "a/b/3.txt", "hello"

  assert_status <<-STATUS
  \ M a/b/3.txt
  STATUS
end
```

As expected, the test fails as `status` does not report the change — it only reports files as changed if their size or mode has changed.

```
1) Failure:
Command::Status::index/workspace changes#test_0003_reports modified files with unchanged size
--- expected
+++ actual
@@ -1,2 +1,2 @@
-"
-"
+# encoding: ASCII-8BIT
+""
```

It seems as though we can’t rely on the `stat()` results alone to tell us whether a file has changed. They’re a useful shortcut, but in some cases we’re just going to have to look at the file contents and compare them to what’s in the index.

Let’s make some further changes to `check_index_entry`. If the size or mode differs, we’ll record the change and return. Otherwise, we fall through to a routine that reads the file, constructs a `Database::Blob` from it, and asks the database to hash the blob by calling a new method `Database#hash_object`. If this hash differs from the object ID recorded in the index, then the file has changed.

```
# lib/command/status.rb

def check_index_entry(entry)
```

```
stat = @stats[entry.path]
return @changed.add(entry.path) unless entry.stat_match?(stat)

data = repo.workspace.read_file(entry.path)
blob = Database::Blob.new(data)
oid = repo.database.hash_object(blob)

@changed.add(entry.path) unless entry.oid == oid
end
```

The `Database#hash_object` method is a piece of functionality extracted from the `Database#store` method. As a reminder, `store` takes a `Blob`, `Tree` or `Commit` object, serialises it, computes the hash of the result, and then stores the object in `.git/objects` in a file name derived from the hash.

```
# lib/database.rb

def store(object)
  string = object.to_s.force_encoding(Encoding::ASCII_8BIT)
  content = "#{$object.type} #{string.bytesize}\x0#{string}"

  object.oid = Digest::SHA1.hexdigest(content)
  write_object(object.oid, content)
end
```

For the purposes of checking the status, we'd like to be able to hash a blob *as though* it were going to be written to the database, so we can compare it to the object ID stored in the index. However, we don't actually want to write this object to disk — it would waste time creating objects that are often not going to end up being part of a commit. And remember, we're doing this for every file whose `stat()` information is unchanged from what's in the index; every opportunity to save time is worth considering, and writing to disk is expensive.

We can break up the functionality of `Database#store` into a few separate methods. Hashing the object and storing it on disk both require the object to be serialised, so we'll break that out as a distinct method. Then, the `hash_object` and `store` methods can build on this in different ways; `hash_object` just returns the result of hashing the serialised object, while `store` calculates the hash, assigns it to the object's `oid` property, and uses the hash to write the serialised object to a file.

```
# lib/database.rb

def store(object)
  content = serialize_object(object)
  object.oid = hash_content(content)

  write_object(object.oid, content)
end

def hash_object(object)
  hash_content(serialize_object(object))
end

private

def serialize_object(object)
  string = object.to_s.force_encoding(Encoding::ASCII_8BIT)
```

```
"#{ object.type } #{ string.bytesize }\0#{ string }"
end

def hash_content(string)
  Digest::SHA1.hexdigest(string)
end
```

This direct check of the file contents makes the test pass, and jit once again accurately reports its own status.

```
$ jit status
M lib/command/status.rb
M lib/database.rb
M test/command/status_test.rb
```

However, the command is now really expensive to run. For all entries where the size and mode are unchanged (i.e. most files most of the time), we read the file and hash it. This is just as bad as the commit command was before we introduced the index, and it will take an awful long time to run on large codebases.

We can observe what this process is doing by using a tool that's installed on most Linux systems called strace⁵, which logs all the system calls a process makes. We'll run jit status under strace, filtered to look for open() calls, and pipe the results into the file ~/status.log. Looking at that file afterward, we can see every file in the project being opened:

```
$ strace -e open jit status 2> ~/status.log
$ cat ~/status.log

# ...
open("/home/ubuntu/jit/LICENSE.txt", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/Rakefile", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/bin/jit", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/command.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/command/add.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/command/base.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/command/commit.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/command/init.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/command/status.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/database.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/database/author.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/database/blob.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/database/commit.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/database/tree.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/entry.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/index.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/index/checksum.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/index/entry.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/lockfile.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/refs.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/repository.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/lib/workspace.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/test/command/add_test.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/test/command/status_test.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/test/command_helper.rb", O_RDONLY|O_CLOEXEC) = 8
open("/home/ubuntu/jit/test/index_test.rb", O_RDONLY|O_CLOEXEC) = 8
```

⁵<https://manpages.ubuntu.com/manpages/bionic/en/man1/strace.1.html>

Each one of these `open()` calls represents the operating system reading the named file from disk, and Jit will then calculate the SHA-1 hash of its contents. Such computations are expensive, and it would be ideal if we could make `status` do its job without needing to read the entire contents of the workspace — that's where the timestamps stored in the index come in. After the following optimisation, we should see fewer calls to `open()` in this trace, meaning our program is spending less time waiting for the operating system to interact with the disk hardware.

9.2.4. Timestamp optimisation

We saw in Section 6.2, “Inspecting `.git/index`” that two of the values the index records for each file are its *ctime* and *mtime*, accurate to the nanosecond. A file’s *ctime*, or change time, is the most recent time at which its attributes — its owner, group, permissions, and so on — were changed. Its *mtime*, or modify time, is the last time at which its contents were changed. If these times are the same in the workspace and the index, then it’s a very good bet that the file has not been changed.

Let’s add a test that does nothing but change a file’s *mtime* using the `touch`⁶ command, which is available in Ruby as `FileUtils.touch`⁷. This won’t change the behaviour of the existing code, but it will trigger it into executing an optimisation code path we’re about to add.

```
# test/command/status_test.rb

  it "prints nothing if a file is touched" do
    touch "1.txt"

    assert_status ""
  end
```

The `touch` helper is another addition to the `CommandHelper` module.

```
# test/command_helper.rb

  def touch(name)
    FileUtils.touch(repo_path.join(name))
  end
```

The optimisation we want to make is to avoid reading and hashing a file if its timestamps match those stored in the index. We’ll change the `Command::Status#check_index_entry` method to add a check that returns if `entry.times_match?(stat)` is true. If we call through to reading the file and find that it has not in fact changed, then we call `repo.index.update_entry_stat(entry, stat)` to update the index so its timestamps match those on disk. This deals with the common pattern of changing a file, saving it, and then undoing your changes — the file’s timestamps will have changed but not its contents, so putting those new timestamps in the index will stop us reading the file again next time.

```
# lib/command/status.rb

  def check_index_entry(entry)
    stat = @stats[entry.path]

    return @changed.add(entry.path) unless entry.stat_match?(stat)
```

⁶<https://manpages.ubuntu.com/manpages/bionic/en/man1/touch.1posix.html>

⁷<https://docs.ruby-lang.org/en/2.3.0/FileUtils.html#method-c-touch>

```
    return if entry.times_match?(stat)

    data = repo.workspace.read_file(entry.path)
    blob = Database::Blob.new(data)
    oid = repo.database.hash_object(blob)

    if entry.oid == oid
      repo.index.update_entry_stat(entry, stat)
    else
      @changed.add(entry.path)
    end
  end
```

We've introduced a couple of new methods here. `Index::Entry#times_match?` takes a `File::Stat` and checks whether the times in the index entry are equal to the times read from the filesystem.

```
# lib/index/entry.rb

def times_match?(stat)
  ctime == stat.ctime.to_i and ctime_nsec == stat.ctime.nsec and
  mtime == stat.mtime.to_i and mtime_nsec == stat.mtime.nsec
end
```

`Index#update_entry_stat` takes an `Index::Entry` and a `File::Stat` and delegates the call to `Index::Entry#update_stat`. The reason this method goes through `Index` is so we can set the `@changed` flag that signals that the updated index should be written to disk.

```
# lib/index.rb

def update_entry_stat(entry, stat)
  entry.update_stat(stat)
  @changed = true
end
```

`Index::Entry#update_stat` takes the `File::Stat` and updates all its properties from the filesystem information, as though the file had been passed to `Index#add`.

```
# lib/index/entry.rb

def update_stat(stat)
  self.ctime      = stat.ctime.to_i
  self.ctime_nsec = stat.ctime.nsec
  self.mtime     = stat.mtime.to_i
  self.mtime_nsec = stat.mtime.nsec
  self.dev       = stat.dev
  self.ino       = stat.ino
  self.mode      = Entry.mode_for_stat(stat)
  self.uid       = stat.uid
  self.gid       = stat.gid
  self.size      = stat.size
end
```

Finally, since we're now updating the index, we need to load the index with `Index#load_for_update` and save it with `Index#write_updates`, and these calls should book-end the method calls that go and gather all the status information.

```
# lib/command/status.rb

def run
  @stats      = {}
  @changed    = SortedSet.new
  @untracked = SortedSet.new

  repo.index.load_for_update

  scan_workspace
  detect_workspace_changes

  repo.index.write_updates

  @changed.each { |path| puts " M #{ path }" }
  @untracked.each { |path| puts "?? #{ path }" }

  exit 0
end
```

If you run `git status` under `strace` again and look at the logs, you may see that unchanged files inside the project are still being opened, because the index doesn't have up-to-date timestamps for those entries. On running it a second time, however, all these `open()` calls should be gone. If you check the index using `hexdump` before and after running `status` each time, you'll also notice that the first call changes it — check the SHA-1 hash at the very end of the file.

9.2.5. Deleted files

In the absence of merge conflicts⁸, the other status that can be reported for files in the workspace is `D`, which means the file is in the index but does not exist in the workspace. This status is not particularly useful yet because we don't have any command for removing files from the index⁹, but in the interests of completeness we'll add support for this status.

Here are a couple of simple examples: when a file is deleted, it should be listed with `D` in the column for index/workspace changes. If a directory is deleted, then all the files inside that directory in the index should be listed as deleted. Unlike the reporting of untracked files, all other statuses list only files, not directories. In this case, we don't list a directory as being removed from the index, we list every file inside it.

```
# test/command/status_test.rb

it "reports deleted files" do
  delete "a/2.txt"

  assert_status <<~STATUS
  \ D a/2.txt
  STATUS
end

it "reports files in deleted directories" do
  delete "a"
```

⁸Chapter 19, *Conflict resolution*

⁹Section 21.1, “Removing files from the index”

```
assert_status <<STATUS
  \ D a/2.txt
  \ D a/b/3.txt
STATUS
end
```

We need one new helper method to make these tests work: `CommandHelper#delete` removes the file or directory passed to it.

```
# test/command_helper.rb

def delete(name)
  FileUtils.rm_rf(repo_path.join(name))
end
```

When we run these tests, we get an error, because `Command::Status#check_index_entry` is getting `nil` when it tries to fetch a `File::Stat` for the given path, so the first method it tries to call on this value generates an exception.

```
1) Error:
Command::Status::index/workspace changes#test_0006_reports deleted files:
NoMethodError: undefined method `size' for nil:NilClass
  /Users/jcoglan/building-git/jit/lib/command/status.rb:46:in `check_index_entry'
  # ...

2) Error:
Command::Status::index/workspace changes#test_0007_reports files in deleted directories:
NoMethodError: undefined method `size' for nil:NilClass
  /Users/jcoglan/building-git/jit/lib/command/status.rb:46:in `check_index_entry'
  # ...
```

Until now, when we found a file that differed from the index, we just added its path to the `@changed` set. But now, we don't want to print `M` for everything in this set, so we're going to need to record what type of change was observed for each path. A path can actually have multiple types of change against it, for example, it may be added to the index but modified in the workspace. To support this, we'll add a new hash called `@changes` whose values default to sets. This structure will map paths to a set of types of changes observed for that path. Untracked files will still be recorded separately.

```
# lib/command/status.rb

def run
  @stats      = {}
  @changed    = SortedSet.new
  @changes    = Hash.new { |hash, key| hash[key] = Set.new }
  @untracked = SortedSet.new

  # ...
```

We've been recording changes so far by calling `@changed.add(entry.path)` to add paths to the set of changed files. But now, we want to record the path and the type of change observed, so we'll add a method for doing that. `record_change` adds the path to the `@changed` set, and it records the type of the change in the `@changes` hash.

```
# lib/command/status.rb
```

```
def record_change(path, type)
  @changed.add(path)
  @changes[path].add(type)
end
```

The fix for the actual problem highlighted by the tests is to change `check_index_entry` so that, if we fail to find a `File::Stat` object for the entry's path, we record a change type of `:workspace_deleted` and bail out.

```
# lib/command/status.rb

def check_index_entry(entry)
  stat = @stats[entry.path]

  unless stat
    return record_change(entry.path, :workspace_deleted)
  end
```

The other places where we were logging changes also need to be updated to use `record_change`; we'll use the name `:workspace_modified` for all changes that lead to an `M` being displayed. Here's the check for size and mode:

```
# lib/command/status.rb

unless entry.stat_match?(stat)
  return record_change(entry.path, :workspace_modified)
end
```

And here's the check for hashing the file contents:

```
# lib/command/status.rb

if entry.oid == oid
  repo.index.update_entry_stat(entry, stat)
else
  record_change(entry.path, :workspace_modified)
end
```

Running the tests again, we see that the code is no longer crashing, but it is printing the wrong thing. The deleted files are being added to `@changes`, but so far everything in that set is printed with an `M`, whereas we need them to be printed with a `D`.

```
1) Failure:
Command::Status::index/workspace changes#test_0006_reports deleted files
--- expected
+++ actual
@@ -1,2 +1,2 @@
- " D a/2.txt
+" M a/2.txt
"

2) Failure:
Command::Status::index/workspace changes#test_0007_reports files in deleted directories
--- expected
+++ actual
@@ -1,3 +1,3 @@
- " D a/2.txt
- D a/b/3.txt
```

```
+" M a/2.txt  
+ M a/b/3.txt  
\"
```

This can be solved by amending the code at the end of `Command::Status#run` that prints the results. Since it's about to get a bit more complicated, I'll extract it into its own method, to be called after updating the index and before exiting.

```
# lib/command/status.rb  
  
def run  
  # ...  
  
  repo.index.write_updates  
  
  print_results  
  exit 0  
end
```

`print_results` iterates over the `@changed` set, but now invokes `status_for` to determine what to print next to each filename. Untracked files are printed with `??` after everything else, as before.

```
# lib/command/status.rb  
  
def print_results  
  @changed.each do |path|  
    status = status_for(path)  
    puts "#{status} #{path}"  
  end  
  
  @untracked.each do |path|  
    puts "?? #{path}"  
  end  
end
```

The `status_for` method takes a path, looks up its set of changes in the `@changes` hash, and returns a two-character string that's appropriate for each path. The default value is two spaces; if the file was found to be deleted then we change its status to `D`; if it was found to be modified we change it to `M`.

```
# lib/command/status.rb  
  
def status_for(path)  
  changes = @changes[path]  
  
  status = " "  
  status = " D" if changes.include?(:workspace_deleted)  
  status = " M" if changes.include?(:workspace_modified)  
  
  status  
end
```

This completes the workspace-related functionality of the `status` command. We're still not able to see the differences between the latest commit and the index and so know what will go into the next commit, but we are able to see what has changed relative to the index, and that at least makes the workflow of running `add .` and then committing more tolerable.

For a full picture, we need to compare the index to the last commit, and this requires developing some infrastructure for reading objects back out of the `.git/objects` database. We'll be looking at that in the next chapter.

10. The next commit

In the last chapter, we partially implemented the status command so that it can show us what has changed between the index and the workspace. It shows us which index entries will be added or updated if we run add ., but to make incremental additions truly useful, we also need to see what has been updated in the index and will be part of the next commit; we need to see the difference between the latest commit and the current index.

In order to achieve this, we'll need to be able to read the complete tree associated with the commit that .git/HEAD is pointing at. We'll often refer to the latest commit as the HEAD commit, or simply HEAD. If we can get the blob ID for every item in the commit's tree, we can compare those to the entries in the index and detect any differences.

To make a start on this, here's a script that attempts to print all the files in HEAD. Many of the methods I'm calling here do not yet exist; the point of writing such an example is to imagine the code one would like to write to achieve some task, and then add all the missing functionality required to make it work.

```
# show_head.rb

require "pathname"
require_relative "./lib/repository"

repo = Repository.new(Pathname.new(Dir.getwd).join(".git"))

head_oid = repo.refs.read_head
commit = repo.database.load(head_oid)

def show_tree(repo, oid, prefix = Pathname.new(""))
  tree = repo.database.load(oid)

  tree.entries.each do |name, entry|
    path = prefix.join(name)
    if entry.tree?
      show_tree(repo, entry.oid, path)
    else
      mode = entry.mode.to_s(8)
      puts "#{mode} #{entry.oid} #{path}"
    end
  end
end

show_tree(repo, commit.tree)
```

If we can get this to run, then we'll have implemented all the methods we need to complete the status command.

10.1. Reading from the database

The first method in our show_head.rb example script is a call to Database#load, which is given the object ID returned by Refs#read_head. We want it to return a Database::Commit object that's reconstructed by reading the object from .git/objects.

Let's start implementing the `Database#load` method. The method itself will just be a public wrapper around two building blocks: reading an object from disk, and caching the result in a hash called `@objects`. Git objects should not be modified after being read from disk; if anything about them is changed then we should get a new object with a new ID. So, once an object is read from disk we can hold onto it in memory and avoid reading it again for the rest of the process.

```
# lib/database.rb

def load(oid)
  @objects[oid] ||= read_object(oid)
end
```

The cache will need to be set up in the constructor for `Database`.

```
# lib/database.rb

def initialize(pathname)
  @pathname = pathname
  @objects = {}
end
```

The `Database#read_object` method is where the actual work happens. The first thing it needs to do is locate the object in the filesystem based on its object ID. Recall that this path is calculated in the `write_object` method:

```
# lib/database.rb

def write_object(oid, content)
  object_path = @pathname.join(oid[0..1], oid[2..-1])
  #
end
```

Let's extract this into a method so that `read_object` can use it.

```
# lib/database.rb

def write_object(oid, content)
  path = object_path(oid)
  #
end

def object_path(oid)
  @pathname.join(oid[0..1], oid[2..-1])
end
```

Now, `read_object` begins by reading the object from disk and using `Zlib` to decompress it. It then creates a `StringScanner`¹ with the resulting data. `StringScanner` is part of Ruby's standard library and is very useful for parsing data, letting us consume data by matching against a *regular expression*², or looking for the next occurrence of an expression.

All objects begin with the object's type, then a space, then a null byte. We use `StringScanner` to `scan_until` we find a space, giving us the object's type, then again until we find a null byte, giving us the size. Once we know the type, we look up a class in a hash called `TYPES` by this

¹<https://docs.ruby-lang.org/en/2.3.0/StringScanner.html>

²https://en.wikipedia.org/wiki/Regular_expression

name, and call `parse` on it, handing it the `StringScanner` so it can process the rest of the data and return the resulting object. Finally, we attach the given object ID to the object and return it.

```
# lib/database.rb

require "strscan"

def read_object(oid)
  data    = Zlib::Inflate.inflate(File.read(object_path(oid)))
  scanner = StringScanner.new(data)

  type  = scanner.scan_until(/\ /).strip
  _size = scanner.scan_until(/\0/)[0...2]

  object = TYPES[type].parse(scanner)
  object.oid = oid

  object
end
```

`TYPES` is a constant in the `Database` class that maps type names back to the classes that represent them.

```
# lib/database.rb

TYPES = {
  "blob"  => Blob,
  "tree"  => Tree,
  "commit" => Commit
}
```

This `Database#read_object` method gives us a new interface we need to implement: the `Blob`, `Tree` and `Commit` classes all need a `parse` method that takes a `StringScanner` and returns an object of the right type. To get our main script working we don't actually need to read blobs themselves, but parsing them is trivial so we may as well deal with it now.

10.1.1. Parsing blobs

A `Blob` is just a wrapper around a string we got by reading a file, so it doesn't really require any parsing per se other than reading all the data out of the object file. We just need to expose an attribute reader for the `@data` property so the caller can get the blob's contents.

```
# lib/database/blob.rb

attr_reader :data

def self.parse(scanner)
  Blob.new(scanner.rest)
end
```

10.1.2. Parsing commits

Now we can deal with something more complicated: commits. As we saw in Section 2.3.4, “Commits on disk”, a commit is stored as a series of line-delimited header fields, followed by a blank line, followed by the message. To parse it, we'll set up a loop that uses `scan_until` to

read up to the next line break. If the line is blank, we break the loop. Otherwise, we split the line on its first space and assign the resulting key-value pair to a hash.

Once the loop is done, we extract the parent, tree and author fields from the hash, we get the message from scanner.rest, and we return a new Commit with these values. We only need to read the tree property for now, so we'll expose an attribute reader for that.

```
# lib/database/commit.rb

attr_reader :tree

def self.parse(scanner)
  headers = {}

  loop do
    line = scanner.scan_until(/\n/).strip
    break if line == ""

    key, value = line.split(/ +/, 2)
    headers[key] = value
  end

  Commit.new(
    headers["parent"],
    headers["tree"],
    headers["author"],
    scanner.rest)
end
```

This implementation makes some simplifications. For example, it ignores the committer field, because our implementation always writes the same data to the author and committer fields. To parse real-world Git commits, we'll need to add support for a distinct committer. It also leaves the author field as a string rather than parsing it further into an Author object, as it is when the commit is written to the database. Again, we don't actually need to read structured author information yet, so we'll leave that until later.

Now that we can parse commits, the call to database.load(head_oid) will return a Database::Commit object with a tree property that we can use to get the ID of the commit's tree.

10.1.3. Parsing trees

The final type of database object we need to parse is trees, the objects that record which version of each file was present in any given commit. In Section 2.3.3, “Trees on disk” we learned that a tree is composed of a series of entries, each of which contains a file’s mode and name, and the object ID of the blob containing the file’s contents.

To parse this, we set up a loop that runs until the end of the input is reached, which we detect using scanner.eos?, short for *end of string*. In each iteration, we scan until the next space to get the file’s mode, and interpret the resulting string as an integer in base 8. Then we scan until the next null byte to get the file’s name. Finally we use scanner.peek to read the next 20 bytes, which are the object ID, and unpack it using the pattern H40, which means a 40-digit hexadecimal string. Having gathered these three items, we make a new Entry out of them — not an Index::Entry, but a Database::Entry, a new class that represents entries read from the

database. This class has fewer fields than `Index::Entry`, since it does not contain all the stat information that the index contains.

Once we've created all the entries and collected them in a hash, we return a new `Database::Tree` containing the entries.

```
# lib/database/tree.rb

def self.parse(scanner)
    entries = {}

    until scanner.eos?
        mode = scanner.scan_until(/\ /).strip.to_i(8)
        name = scanner.scan_until(/\0/)[0...-2]

        oid = scanner.peek(20).unpack("H40").first
        scanner.pos += 20

        entries[name] = Entry.new(oid, mode)
    end

    Tree.new(entries)
end
```

We'll change the `Database::Tree` class's constructor so that it can take an optional set of entries as input, rather than taking no arguments and always starting with an empty hash as it did in Section 5.2.2, "Building a Merkle tree". We'll also expose the `entries` instance variable so that other methods can iterate over a tree's contents.

```
# lib/database/tree.rb

attr_reader :entries

def initialize(entries = {})
    @entries = entries
end
```

The `Database::Entry` class is a simple structure that holds the object ID and file mode, and has a single method that our `show_tree.rb` test script relies upon. `Database::Entry#tree?` returns true if the entry points to another tree, rather than a blob, which we detect by checking if its mode is the `400008` mode that's assigned to trees.

```
# lib/database/entry.rb

class Database
    Entry = Struct.new(:oid, :mode) do
        def tree?
            mode == Tree::TREE_MODE
        end
    end
end
```

Using a `Struct` for this data will make it easier to compare tree entries for equality when we come to implement tree diffs in Section 14.1, "Telling trees apart".

It might seem odd that when we were writing trees to the database, we filled `@entries` with `Index::Entry` objects, but we now fill them with objects of a different

type — `Database::Entry`. In a dynamically typed language like Ruby, the classes that objects belong to is not as important as which methods they respond to. For example, look at the `Database::Tree#to_s` method:

```
# lib/database/tree.rb

def to_s
  entries = @entries.map do |name, entry|
    mode = entry.mode.to_s(8)
    ["#{ mode } #{ name }", entry.oid].pack(ENTRY_FORMAT)
  end

  entries.join("\n")
end
```

What assumptions does this method make about the value of `entry`? The only methods it calls on it are `mode`, which should return an integer, and `oid`, which should return a 40-character hexadecimal string. The same is true of our `Database::Entry` class, so we can use instances of it where previously we might have used an `Index::Entry`. This ability to substitute values of one type for values of another is called the Liskov substitution principle³.

There are other methods in `Database::Tree` that make additional requirements about entry values; `Database::Tree#add_entry` and `Database::Tree#traverse` require entries to respond to `basename` and `traverse` respectively. However, these methods are only used when creating trees to write to the database, not when reading them; trees read from the database should not be modified.

But, there's a more interesting distinction to be made between trees when they are written and when they are read. When they are created during the `commit` command, a `Tree` is a recursive structure made of `Index::Entry` and other `Tree` objects; each `Tree` contains the whole structure of all the nodes inside it. But when they are read, a `Tree` is a flat structure containing only its immediate children, represented as `Database::Entry` objects. Many operations that read trees from the database will not need the entire tree to be read, so we'd like to be able to load the tree one node at a time, rather than having the database spend time recursively fetching an entire tree we might not need.

This structural difference means that a `Tree` being created is fundamentally different from a `Tree` being read — maybe they should be different classes? When grouping data and behaviour into classes, it's often useful to look at which sets of data and methods are used together, rather than grouping around an abstract concept like a ‘tree’ that they are all related to. In this case, all the previously-defined `Tree` methods — `add_entry`, `traverse`, `mode`, `type` and `to_s` — are used when creating trees, and `entries` is used when reading them. It's a perfectly reasonable idea to split these two groups of methods into two classes, say, `WritableDatabase` and `ReadableTree`. However, that would separate the code that serialises a `Tree` from that that parses one, and in the other `Database` object classes and in `Index` these routines are kept together in the same class.

There's not a right or wrong answer here, it's a trade-off about which data and behaviour we'd like to be kept together, in light of what will make the code easier to work with. For now, I'd

³https://en.wikipedia.org/wiki/Liskov_substitution_principle

prefer to make Tree look like like the other database objects, and keep its reading and writing methods together.

10.1.4. Listing the files in a commit

With all the above machinery in place, our `show_head.rb` script is complete, and when run, will print out the mode and object ID of every blob in the commit.

```
$ ruby show_head.rb

100644 f288702d2fa16d3cdf0035b15a9fcbe552cd88e7 LICENSE.txt
100644 c8c218e0d599831338af0748af283a149b9ff8d2 Rakefile
100755 0148dff020d1ce12464239c792e23820183c0829 bin/jit
100644 182dce271bb629839f16a8c654a0bcba85aaabdb lib/command.rb
100644 5a4cc34fa29a4a46702fbe701a9410bb6fc5b5be lib/command/add.rb
100644 e7eade573e8be1dfc6cb5e7af949f1222db757d3 lib/command/base.rb
100644 b066585fc6c16bd97020e5b2dc12d6b357fdd546 lib/command/commit.rb
100644 846e2d1d9d5f8f67efe6cd0aa1f23fe3fc276588 lib/command/init.rb
100644 0b9112d352746a3f53f0a2c323d8cc4e45e06bf5 lib/command/status.rb
100644 a34d3a1b228173757e8eebac9ce36a70530bd9e lib/database.rb
100644 974cb4219bc52d52a51c5dd603a825f98b67d838 lib/database/author.rb
100644 06a23b3af65371201e3bad0c719a9f1eec62a89 lib/database/blob.rb
100644 bf864ea11e09a6b1826c6c85bbc1f0c281cf952 lib/database/commit.rb
100644 4f7f284bee588d099cff219625752721da08d017 lib/database/tree.rb
100644 14899940c8aa6a748f7712071aa3af796d1ffbd2 lib/entry.rb
100644 b2402c884235cf458a56f5e39b81cb33772d37f0 lib/index.rb
100644 6bfbb89a5da1ad37f54a53e04060cc11bc81d3d6 lib/index/checksum.rb
100644 2090ffffd61b81e0c047bdeecc86c347ea595d246 lib/index/entry.rb
100644 e3fcecae505cf59ac0c84ded377838e68ab316bf lib/lockfile.rb
100644 9a7fcc04af2d7dc6e9080964bf5135162e9518c6 lib/refs.rb
100644 0e511fe66af08ba50e856a3321f9a808b8cab5a0 lib/repository.rb
100644 b3171c6c9c95672f650da8e11f1bbe0753ebfc0b lib/workspace.rb
100644 8e93a73bd9f36144a1299c995e600dc7e756c847 test/command/add_test.rb
100644 08b71c3782e1b198fcc0e8762e1398792f68178a test/command/status_test.rb
100644 21f313e6188b0917d5cb7cf17f77ca9d26405d92 test/command_helper.rb
100644 fac3b1462e80a9bff019fb51cee9b502589addcd test/index_test.rb
```

A final check on the project's status will tell us what's changed, so we know what will be committed when we run the `add .` and `commit` commands.

```
$ jit status
M lib/database.rb
M lib/database/blob.rb
M lib/database/commit.rb
M lib/database/tree.rb
?? lib/database/entry.rb
```

10.2. HEAD/index differences

Now that we have the ability to read trees from the database, we can compare the index against the HEAD commit to see what changes will be included in the next commit. These changes appear in the first column of the `--porcelain` status format and, ignoring merge conflicts and rename detection for the time being, this column can display three different states. A means a file was added, that is, it appears in the index but not in HEAD. D means it was deleted, that is it appears in HEAD but not in the index. And finally, M means a file is modified; it appears in both HEAD and the index but with a different mode or object ID.

All the tests in this section will begin with a single commit in the repository, just so that HEAD exists and points at a commit. The index will begin in sync with HEAD and no changes will be made in the workspace. The starting state will be equivalent to running these commands:

```
$ mkdir -p a/b  
$ echo "one" > 1.txt  
$ echo "two" > a/2.txt  
$ echo "three" > a/b/3.txt  
$ git add .  
$ git commit --message "first commit"
```

Here is the same state of affairs expresses as minitest setup code, using our CommandHelper module.

```
# test/command/status_test.rb  
  
describe "head/index changes" do  
  before do  
    write_file "1.txt", "one"  
    write_file "a/2.txt", "two"  
    write_file "a/b/3.txt", "three"  
  
    jit_cmd "add", ".  
    commit "first commit"  
  end
```

10.2.1. Added files

To check how Git represents added files, let's try adding a couple of new files to the index. We'll add one inside an already-tracked directory:

```
$ echo "four" > a/4.txt  
$ git status --porcelain  
?? a/4.txt  
  
$ git add .  
$ git status --porcelain  
A a/4.txt
```

Notice Git displays the specific file as untracked at first, before listing it as added after add has been run. Next we'll git reset --hard to remove any changes since the HEAD commit, add a file in a directory that's not part of the HEAD tree:

```
$ mkdir -p d/e  
$ echo "five" > d/e/5.txt  
$ git status --porcelain  
?? d/  
  
$ git add .  
$ git status --porcelain  
A d/e/5.txt
```

At first, the directory d is listed as untracked — Git lists untracked directories, not all the files inside them. However once the file is added, the file d/e/5.txt is listed as added. So, when a directory is in the index but is not part of the HEAD tree, Git still lists each new file inside it individually, rather than listing the directory itself as added.

Below are two tests that reproduce the above examples.

```
# test/command/status_test.rb

it "reports a file added to a tracked directory" do
  write_file "a/4.txt", "four"
  jit_cmd "add", "."

  assert_status <<STATUS
    A a/4.txt
  STATUS
end

it "reports a file added to an untracked directory" do
  write_file "d/e/5.txt", "five"
  jit_cmd "add", "."

  assert_status <<STATUS
    A d/e/5.txt
  STATUS
end
end
```

These tests currently fail because `status` produces no output; it currently only detects differences between the index and workspace, of which none exist after running `add ..`. We need to build some infrastructure for detecting and reporting HEAD/index differences.

Currently, the code of the `Command::Status#run` method looks like this:

```
# lib/command/status.rb

def run
  #
  # ...

  repo.index.load_for_update

  scan_workspace
  detect_workspace_changes

  repo.index.write_updates

  #
end
```

Let's add a step to this recipe that loads the whole of the HEAD commit's tree. We'll also replace `detect_workspace_changes` with `check_index_entries` since we'll need to do more when scanning the index than just comparing it to the workspace.

```
# lib/command/status.rb

scan_workspace
load_head_tree
check_index_entries
```

Many commands won't need to load an entire tree recursively, which is why we implemented `Database#load` to return a single `Tree` at a time. However, loading the whole tree will make implementing `status` a bit more convenient, and as we'll see in later examples, some `status` output requires displaying the whole contents of a sub-tree even if it's no longer in the index;

there is not the same opportunity to skip unindexed directories as there was when detecting untracked files.

The `load_head_tree` method is split into two parts. The first reads the object ID from `Refs#read_head`, and loads the commit with that ID using `Database#load`. Then we enter a recursive procedure called `read_tree`, which takes the ID of a tree, loads it from the database, and then finds any entries that themselves refer to trees and loads them too. The results of this process are stored in the `@head_tree` hash, which maps file paths to `Database::Entry` objects that hold the mode and object ID of each file. You'll recognise the structure of this method from the `show_head.rb` script at the beginning of this chapter.

```
# lib/command/status.rb

def load_head_tree
  @head_tree = {}

  head_oid = repo.refs.read_head
  return unless head_oid

  commit = repo.database.load(head_oid)
  read_tree(commit.tree)
end

def read_tree(tree_oid, pathname = Pathname.new(""))
  tree = repo.database.load(tree_oid)

  tree.entries.each do |name, entry|
    path = pathname.join(name)
    if entry.tree?
      read_tree(entry.oid, path)
    else
      @head_tree[path.to_s] = entry
    end
  end
end
```

Once `load_head_tree` has completed, we call `check_index_entries`. This does the same thing that `detect_workspace_changes` was doing — iterating over the index and checking each entry. However now, as well as checking each entry against the workspace, we also check it against the HEAD tree.

```
# lib/command/status.rb

def check_index_entries
  repo.index.each_entry do |entry|
    check_index_against_workspace(entry)
    check_index_against_head_tree(entry)
  end
end
```

`check_index_against_workspace` is just the `check_index_entry` method with its name altered to reflect its specific responsibility. `check_index_against_head_tree` is a new method that compares each index entry against the entries in `@head_tree` to detect differences. The first difference we've written tests for is files being added in the index that don't appear in HEAD, and this is easy enough to check for.

```
# lib/command/status.rb

def check_index_against_head_tree(entry)
  item = @head_tree[entry.path]

  unless item
    record_change(entry.path, :index_added)
  end
end
```

Finally, we need to adjust `status_for` to return statuses for HEAD/index changes as well as index/workspace differences. Rather than returning a single string, we'll now generate two strings, one for each column of the status output, and concatenate them together.

```
# lib/command/status.rb

def status_for(path)
  changes = @changes[path]

  left = " "
  left = "A" if changes.include?(:index_added)

  right = " "
  right = "D" if changes.include?(:workspace_deleted)
  right = "M" if changes.include?(:workspace_modified)

  left + right
end
```

This is where the importance of storing sets rather than single statuses in `@changes` becomes apparent. A single file could have the status `AM`, meaning it exists in the index but not in HEAD, and its copy in the workspace differs from the one in the index.

10.2.2. Modified files

To check that the `M` status is reported correctly, we need two test cases. In one, we'll make a file executable and update the index, and we expect to see that change reflected in the output. In the other, we'll change the contents of the file and update the index, changing the file's object ID relative to HEAD.

```
# test/command/status_test.rb

it "reports modified modes" do
  make_executable "1.txt"
  jit_cmd "add", "."

  assert_status <<~STATUS
    M 1.txt
    STATUS
  end

  it "reports modified contents" do
    write_file "a/b/3.txt", "changed"
    jit_cmd "add", "."

    assert_status <<~STATUS
      M 3.txt
      C 3.txt
      STATUS
    end
end
```

```
M a/b/3.txt
STATUS
end
```

Again, both these tests fail because the status command produces no output. This is easily remedied by extending the `check_index_against_head_tree` method so that, if it finds there is a corresponding item in `@head_tree`, we check the index entry's mode and oid against it and record a change if there are any differences.

```
# lib/command/status.rb

def check_index_against_head_tree(entry)
  item = @head_tree[entry.path]

  if item
    unless entry.mode == item.mode and entry.oid == item.oid
      record_change(entry.path, :index_modified)
    end
  else
    record_change(entry.path, :index_added)
  end
end
```

Then we add one more clause to `status_for` to generate an `M` for `:index_modified` changes.

```
# lib/command/status.rb

left = ""
left = "A" if changes.include?(:index_added)
left = "M" if changes.include?(:index_modified)
```

Now we can see jit reporting its own status accurately again. If we run `git status` once, it shows an `M` in the second column; the files are modified relative to the index. If we call `add` and check the status again, there is an `M` in the first column instead; the files in the index are now modified relative to HEAD.

```
$ git status
M lib/command/status.rb
M test/command/status_test.rb

$ git add .

$ git status
M lib/command/status.rb
M test/command/status_test.rb
```

10.2.3. Deleted files

Our final status check for now will be reporting files that are in the HEAD tree but not in the index, which should be reported with a `D` status in the first column. One thing is worth pointing out here: if a whole directory is removed from the index, Git's `status` command still lists all the files inside that directory as deleted.

```
$ git rm -rf a
$ git status --porcelain
D a/2.txt
```

```
D a/b/3.txt
```

As was mentioned above, this means that `read_tree` cannot skip directories that are not tracked in the index, because we need the names of all the files inside that directory to print an accurate status.

Below are two further Ruby tests, one for deleting a single file and one for deleting a directory. Because we don't yet have an `rm` command, I'm simulating it by deleting the `.git/index` file and then adding the whole workspace to the index again, which will result in an index missing the deleted files.

```
# test/command/status_test.rb

it "reports deleted files" do
  delete "1.txt"
  delete ".git/index"
  jit_cmd "add", "."

  assert_status <<-STATUS
  D 1.txt
  STATUS
end

it "reports all deleted files inside directories" do
  delete "a"
  delete ".git/index"
  jit_cmd "add", "."

  assert_status <<-STATUS
  D a/2.txt
  D a/b/3.txt
  STATUS
end
```

To detect deleted HEAD items, we need to find any items in the HEAD tree that aren't in the index. We will have missed these while calling `check_index_entries`, so we need to make a pass over `@head_tree` and collect any items in there that are missing from the index. Let's add another step to the recipe in `run` to record the remaining `@head_tree` items:

```
# lib/command/status.rb

scan_workspace
load_head_tree
check_index_entries
collect_deleted_head_files
```

The `collect_deleted_head_files` method simply iterates any items left in `@head_tree` and marks those paths that are untracked with the `:index_deleted` status.

```
# lib/command/status.rb

def collect_deleted_head_files
  @head_tree.each_key do |path|
    unless repo.index.tracked_file?(path)
      record_change(path, :index_deleted)
    end
  end
end
```

```
end
```

Notice we're not using the existing `Index#tracked?` method for this, because that returns `true` for tracked directories as well as files. If a file that exists in the `HEAD` tree is now a directory in the index, we would like to list it as deleted rather than ignoring. Therefore the `Index` interface requires a slight modification.

```
# lib/index.rb

def tracked_file?(path)
  @entries.has_key?(path.to_s)
end

def tracked?(path)
  tracked_file?(path) or @parents.has_key?(path.to_s)
end
```

One extra clause in `status_for` will take care of displaying these items correctly.

```
# lib/command/status.rb

left = ""
left = "A" if changes.include?(:index_added)
left = "M" if changes.include?(:index_modified)
left = "D" if changes.include?(:index_deleted)
```

That takes care of all the common statuses that Git reports. When we cover merge conflicts⁴ there will be more statuses to consider, but for now this is enough to accurately report any state the project could be in. As we continue to build new features, I will be adding tests to the codebase in the `test` directory, but will not be mentioning them further in the text. If you want to see how each feature is tested, you can consult the history in the Jit repository.

10.3. The long format

The `status` command we've implemented so far emits the so-called porcelain format, which you get if you run `git status --porcelain`. This is really easy for other programs to parse, but not particularly intuitive to a human reader. That's why Git's default status format is the one you're probably more familiar with:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   1.txt
    deleted:   a/2.txt
    new file:  z.txt

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    deleted:   1.txt
    modified:  a/b/3.txt
```

⁴Chapter 19, *Conflict resolution*

```
Untracked files:  
(use "git add <file>..." to include in what will be committed)  
  
 5.txt
```

This is called the long format; you can get it using the `--long` flag but it's the default output if you don't use any other flags. It highlights the files listed in green and red, it gives you helpful instructions for making changes, and it clearly separates the changes that will go into the next commit from those that are still not in the index. While the porcelain format is easier to write tests for, the long format is nicer for day-to-day work.

Before launching into implementing this format, let's look at the way we're representing the repository status in memory. The above status looks like this when requested in the porcelain format:

```
$ git status --porcelain  
MD 1.txt  
D a/2.txt  
M a/b/3.txt  
A z.txt  
?? 5.txt
```

Our `Command::Status` class uses an internal representation that's very close to this representation — this is not surprising, since it was directly motivated by the requirements of this format. When generating this status, the in-memory values held by the command will be:

```
@changed = SortedSet.new(["1.txt", "a/2.txt", "a/b/3.txt", "z.txt"])  
  
 @changes = {  
   "1.txt"    => Set.new([:index_modified, :workspace_deleted]),  
   "a/2.txt"  => Set.new([:index_deleted]),  
   "a/b/3.txt" => Set.new([:workspace_modified]),  
   "z.txt"    => Set.new([:index_added])  
 }  
  
 @untracked = SortedSet.new(["5.txt"])
```

How would we generate the long format using this information? We need to generate two separate lists of index changes and workspace changes, rather than listing them together. To find index changes, we'd need to filter the `@changed` set for paths for which `@changes` contains any index-related changes. Then when printing each path, we'd need to find out exactly which index-related status to use. Ruby's `Set` class makes this a little easier, but we'd need something like:

```
INDEX_STATUSES = [:index_added, :index_deleted, :index_modified]  
  
index_changes = @changed.map do |path|  
  status = @changes[path].intersection(INDEX_STATUSES).first  
  [path, status]  
end  
  
index_changes.reject! { |_, status| status == nil }
```

It's not the end of the world, but it does require a lot more iteration over the structures after we've generated them — `map`, `intersection` and `reject!` are very convenient but they hide a

lot of work. And while this code may be very pithy, it's not particularly easy to follow — why are we taking the first member of the intersection of two sets here? Adding a new feature has got us using our existing data structures in strange ways that's going to make the feature harder to maintain in future. Maybe it's time for a little refactoring.

10.3.1. Making the change easy

There's an oft-quoted saying by Kent Beck⁵, which goes:

for each desired change, make the change easy (warning: this may be hard), then
make the easy change

— Kent Beck

In our current situation, the thing we want to make easy is we'd like to write a method of printing the status output that lists index and workspace changes separately. Take another look at the current structure we're using to record changes:

```
@changes = {
  "1.txt"    => Set.new([:index_modified, :workspace_deleted]),
  "a/2.txt"   => Set.new([:index_deleted]),
  "a/b/3.txt" => Set.new([:workspace_modified]),
  "z.txt"     => Set.new([:index_added])
}
```

And, look closely at this line in the above code:

```
status = @changes[path].intersection(INDEX_STATUSES).first
```

This has something in common with the code we're using to generate the single-letter statuses for the porcelain format:

```
# lib/command/status.rb

def status_for(path)
  changes = @changes[path]

  left = " "
  left = "A" if changes.include?(:index_added)

  right = " "
  right = "D" if changes.include?(:workspace_deleted)
  right = "M" if changes.include?(:workspace_modified)

  left + right
end
```

The clue to what's wrong with this code is the word `first`. There's an assumption there that when you check `@changes` for index-related statuses, you will get a set containing a single value (or nothing). That assumption is also encoded in the `status_for` method: we select a single letter for each column depending on which of the `:index_` or `:workspace_` statuses exist for each path. This code reveals that each path has only one index-related and one workspace-related status, at most.

⁵<https://twitter.com/kentbeck/status/250733358307500032>

Given this revelation, why are we using a Set for the statuses each path has? It seemed like a reasonable-enough choice at the time; each path can have multiple, unique statuses. However, a Set is an open-ended collection. That's fine for collecting a set of paths that will differ from project to project, but the set of statuses each path can take is fixed: each path can be changed in the index and the workspace, and can be added, modified or deleted in each case. There's no need for an open-ended collection to represent this, and using one actually adds complexity to the program!

So what would be a better data representation that would reduce the amount of iteration needed? Here's one idea:

```
@changes = {
    "1.txt"    => { :index => :modified, :workspace => :deleted },
    "a/2.txt"   => { :index => :deleted },
    "a/b/3.txt" => { :workspace => :modified },
    "z.txt"     => { :index => :added }
}
```

This would reduce the need to iterate over the statuses for each path, but it would not remove the need to iterate over the paths themselves. The search for index changes would now look like this:

```
index_changes = @changed.map do |path|
    [path, @changes[path][:index]]
end

index_changes.reject! { |_, status| status == nil }
```

This is certainly an improvement, but we can go one better by turning the structure inside out:

```
@index_changes = {
    "1.txt"    => :modified,
    "a/2.txt"   => :deleted,
    "z.txt"     => :added
}

@workspace_changes = {
    "1.txt"    => :deleted,
    "a/b/3.txt" => :modified
}
```

Now, finding the index and workspace changes doesn't require any work: the data is structured to represent them directly. Getting the merged list for the porcelain output is still simple, provided we preserve the @changed set that lists the names of all changed files in tandem with the above structures. Getting the status for each file in the porcelain output is also simple: we just look up the file's name in the above two hashes.

Let's apply this design change to the `Command::Status` class. The state initialised at the top of the `run` method will now look like this, where in place of the `@changes` hash we now have a hash for the index and one for the workspace.

```
# lib/command/status.rb

def run
```

```
    @stats          = {}
    @changed        = SortedSet.new
    @index_changes = {}
    @workspace_changes = {}
    @untracked_files = SortedSet.new

    # ...
end
```

We change each of the `record_change` calls in the class to restructure the arguments: rather than a single status name, we now pass a reference to which hash we want to insert the status into.

```
record_change(entry.path, :workspace_deleted)
```

becomes:

```
record_change(entry.path, @workspace_changes, :deleted)
```

The `record_change` method must be changed to accept the modified arguments. It still adds the path to the `@changed` set, but it now assigns the status to a hash that's also given in the arguments.

```
# lib/command/status.rb

def record_change(path, set, type)
  @changed.add(path)
  set[path] = type
end
```

Finally, the `status_for` method can be simplified using these new structures to dynamically look up the status letter for each column.

```
# lib/command/status.rb

SHORT_STATUS = {
  :added    => "A",
  :deleted  => "D",
  :modified => "M"
}

def status_for(path)
  left  = SHORT_STATUS.fetch(@index_changes[path], " ")
  right = SHORT_STATUS.fetch(@workspace_changes[path], " ")

  left + right
end
```

Having made these changes, all the tests still pass so we can commit here and go on to add the new functionality.

10.3.2. Making the easy change

Having the data organised into the distinct collections `@index_changes`, `@workspace_changes` and `@untracked_files` should make it straightforward to write the long format implementation — all it has to do is list out these structures.

Before we make those changes though, we need to add an extra argument when we invoke the status commands in the tests. The long format will now be the default, and the porcelain format requires passing the `--porcelain` option.

```
# test/command/status_test.rb

def assert_status(output)
  jit_cmd "status", "--porcelain"
  assert_stdout(output)
end
```

To make use of this option, we rename the existing `print_results` method to `print_porcelain_format`, and reimplement `print_results` to make a choice of which format to print, based on the input.

```
# lib/command/status.rb

def print_results
  if @args.first == "--porcelain"
    print_porcelain_format
  else
    print_long_format
  end
end
```

`print_long_format` defines the overall structure of the long status message. If there are any changes in the index, then we print those out, and ditto for workspace changes and untracked files. Finally we print a trailing message that summarises what is staged for the next commit.

```
# lib/command/status.rb

def print_long_format
  print_changes("Changes to be committed", @index_changes)
  print_changes("Changes not staged for commit", @workspace_changes)
  print_changes("Untracked files", @untracked_files)

  print_commit_status
end
```

The `print_changes` method takes care of printing `@index_changes` and `@workspace_changes`, which are hashes with a similar structure; they map strings representing filenames to one of the symbols `:added`, `:modified` or `:deleted`. The method looks up the long-form status message for each of these states and prints it out, left-justified so that the filenames are aligned in a column. It also prints `@untracked_files`, which is an array of paths without associated status symbols.

```
# lib/command/status.rb

LABEL_WIDTH = 12

LONG_STATUS = {
  :added    => "new file:",
  :deleted  => "deleted:",
  :modified => "modified:"
}
```

```
def print_changes(message, changeset)
  return if changeset.empty?

  puts "#{ message }:"
  puts ""
  changeset.each do |path, type|
    status = type ? LONG_STATUS[type].ljust(LABEL_WIDTH, " ") : ""
    puts "\t#{ status }#{ path }"
  end
  puts "
```

Finally, `print_commit_status` prints a trailing line at the end of the output. If there are changes to the index, then nothing is printed, otherwise a message is chosen based on the state of the workspace changes and untracked files.

```
# lib/command/status.rb

def print_commit_status
  return if @index_changes.any?

  if @workspace_changes.any?
    puts "no changes added to commit"
  elsif @untracked_files.any?
    puts "nothing added to commit but untracked files present"
  else
    puts "nothing to commit, working tree clean"
  end
end
```

With these changes in place, the `status` command should now tell us the state of the repository, consistently for each format:

```
$ jit status --porcelain
M lib/command/status.rb
M test/command/status_test.rb

$ jit status
Changes not staged for commit:

      modified: lib/command/status.rb
      modified: test/command/status_test.rb

no changes added to commit
```

If we add one of the changed files, then that file should now be listed as staged for the next commit.

```
$ jit add lib/command

$ jit status --porcelain
M lib/command/status.rb
M test/command/status_test.rb

$ jit status
Changes to be committed:

      modified: lib/command/status.rb
```

```
Changes not staged for commit:
```

```
    modified: test/command/status_test.rb
```

Everything seems to be working, so we can make another commit here. It's worth noting that the commit including this change only adds new lines to the codebase; nothing is deleted. This often the aim with 'making the change easy': we first rearrange the existing code so that the new feature is a simple addition, rather than adding new functionality by directly changing a lot of existing code.

10.3.3. Orderly change

There is one small problem with the above implementation: it prints index changes in the wrong order sometimes. Here is a section of the output of the status commands using Git and Jit for the repository we looked at at the beginning of Section 10.3, "The long format":

```
$ git status  
  
Changes to be committed:  
  
    modified: 1.txt  
    deleted: a/2.txt  
    new file: z.txt  
  
$ jit status  
  
Changes to be committed:  
  
    modified: 1.txt  
    new file: z.txt  
    deleted: a/2.txt
```

In Jit, the deleted file is listed last, even though its name should sort before the name of the file that's been newly added. This is because of how the `@index_changes` hash is constructed. First we find the added and modified files while iterating over the index, and because the index entries are sorted, these files remain in the correct order in the hash. However, deleted files are added in the final step where we iterate over the remaining tree items, in `collect_deleted_head_files`. That means deleted files will always be listed last.

Does that mean we need to redesign our data structures yet again? Do we need more analogs of the `@changed` set which stores a sorted list of all the changed paths, but for each of the separate sets of changes? Far from it — the general structure is just what we need to print the results, it's just the order that's not right. We can fix this either by sorting the keys in `@index_changes` before iterating over it, or, we can introduce a variant of a hash that keeps its keys sorted transparently.

We'll create a subclass of Ruby's `Hash` class whose assignment method adds the given key to a `SortedSet`. The `each` method is then overridden to iterate the hash in the order of this set.

```
# lib/sorted_hash.rb  
  
require "set"
```

```
class SortedHash < Hash
  def initialize
    super
    @keys = SortedSet.new
  end

  def []=(key, value)
    @keys.add(key)
    super
  end

  def each
    @keys.each { |key| yield [key, self[key]] }
  end
end
```

All that remains is to replace the normal hashes in which we store the changes:

```
# lib/command/status.rb

@index_changes      = {}
@workspace_changes = {}
```

with instances of this `SortedHash` class.

```
# lib/command/status.rb

@index_changes      = SortedHash.new
@workspace_changes = SortedHash.new
```

And hey presto, the index changes come out in the right order.

```
$ jit status
Changes to be committed:

modified:  1.txt
deleted:   a/2.txt
new file:  z.txt
```

10.4. Printing in colour

Our `status` command now prints all the right information, but it's lacking one thing: a splash of colour. Git lists changes in the long format with the index changes in green, and the workspace changes and untracked files in red. It does this by embedding *escape codes*⁶ in the text that it writes to standard output, and the terminal interprets these codes as instructions to change the formatting, rather than as literal text. For example, writing the characters `\e[31m` to the terminal will make it print the text that follows it in red. `\e` represents the *escape character*, a non-printing control character with byte value `1b` in ASCII.

Let's add a few annotations to `Command::Status` to indicate which bits of output we want in which colours. The `print_changes` method handles both index and workspace changes, so it will need to take an argument for which colour to use.

⁶https://en.wikipedia.org/wiki/ANSI_escape_code

```
# lib/command/status.rb

def print_long_format
  print_changes("Changes to be committed", @index_changes, :green)
  print_changes("Changes not staged for commit", @workspace_changes, :red)
  print_changes("Untracked files", @untracked_files, :red)

  print_commit_status
end
```

Then, each call to `puts` will need to pass some of its input through a function to wrap it with colour-changing escape codes, for example:

```
# lib/command/status.rb

def print_changes(message, changeset, style)
  # ...
  status = type ? LONG_STATUS[type].ljust(LABEL_WIDTH, " ") : ""
  puts "\t" + fmt(style, status + path)
  # ...
end
```

We'll define the `fmt` method in `Command::Base` so that all commands can use it. All it does is pass the requested style and string off to a new module called `Color` that handles terminal colour formatting.

```
# lib/command/base.rb

def fmt(style, string)
  Color.format(style, string)
end
```

`Color.format` will take the name of a style — currently either `:red` or `:green`, look up the code for that style in a table⁷, and return the given string wrapped in escape codes that will cause the string to be printed in that colour. The characters `\e[m` appended at the end reset the terminal's formatting to its default state.

```
# lib/color.rb

module Color
  SGR_CODES = {
    "red"  => 31,
    "green" => 32
  }

  def self.format(style, string)
    code = SGR_CODES.fetch(style.to_s)
    "\e[#{code}m#{string}\e[m"
  end
end
```

With these simple changes made, the `status` command now prints the project's state in colour. Notice that the final line of output is in the terminal's default colour, because of the `\e[m` appended to the strings shown in red.

⁷The abbreviation SGR stands for *select graphic rendition*

```
$ jit status
Changes not staged for commit:

    modified:   lib/command/base.rb
    modified:   lib/command/status.rb

Untracked files:

    lib/color.rb

no changes added to commit
$
```

After adding one of the files to the index, it moves to the upper list, rendered in green.

```
$ jit add lib/command/base.rb
$
$ jit status
Changes to be committed:

    modified:   lib/command/base.rb

Changes not staged for commit:

    modified:   lib/command/status.rb

Untracked files:

    lib/color.rb

$
```

There's only one slight problem with this new feature. Sometimes the user will want to pipe Jit's commands into another process, for example, to write their output to a file or view it in a pager like `less`⁸:

```
$ jit status | less
```

If we run this, the escape characters are shown on screen:

```
Changes to be committed:  
  
  ESC[32mmodified: lib/command/base.rbESC[m  
  
Changes not staged for commit:  
  
  ESC[31mmodified: lib/command/status.rbESC[m
```

With `less`, we can use the `-R` option to make it interpret escape codes rather than printing them, but in general, users piping a command into another process will expect escape codes to be stripped out.

This can be accomplished by checking if the output stream we're writing to is a TTY⁹, short for *teletype*. Loosely speaking, this means the stream is connected to a terminal, or something that acts like a terminal, rather than being connected to a file or another process. The term comes from historical computer systems where the terminal would literally be a typewriter hooked up to a remote machine, which would print output on paper as it arrived. This was later superseded by video screens, and the terminal program on your computer emulates this kind of output device — you'll sometimes hear it referred to as a *terminal emulator*¹⁰.

In Ruby, any I/O object can tell you if it's a TTY using the `isatty` method. We can put a check into our `fmt` method to check if `@stdout` is a TTY, and return the string unmodified if not.

```
# lib/command/base.rb  
  
def fmt(style, string)  
  @stdout.isatty ? Color.format(style, string) : string  
end
```

This modification means the `status` command will still print in colour when it's writing to a terminal, but not when being piped into another program. That's why the `fmt` method needs to be defined in a command class, and not as a global helper function: we need to inspect what the command is writing to in order to determine how to change the string.

One final note on terminal formatting: these escape codes are state-changing instructions, not a nested markup language. The `\e[m` added at the end by `fmt` resets the formatting state, so any colours being used before this string was written will be forgotten.

For example, the Ruby code

```
fmt(:red, "my #{ fmt :green, "first" } program")
```

will generate the string

⁸<https://manpages.ubuntu.com/manpages/bionic/en/man1/less.1.html>

⁹<https://en.wikipedia.org/wiki/TTY>

¹⁰https://en.wikipedia.org/wiki/Terminal_emulator

```
\e[31mmy \e[32mfirst\e[m program\e[m
```

The word `program` will not be printed in red, but in the terminal's default colour, because the `\e[m` after the word `first` resets everything. This means these control codes cannot represent nested formatting instructions. That's not a problem right now, but there are cases where you might want to redesign the `fmt` interface to allow for nested styling.

11. The Myers diff algorithm

With the `status`, `add` and `commit` commands, we have a nice set of tools for working on the codebase and building up a version history. But we're still missing a lot of things that make Git truly useful: we can't see exactly which lines we've changed from the last commit, we can't see a log of what changed in past commits, and we can't create branches and merge them back together.

Before we can make progress with any of these things, we need to take a digression into studying diff algorithms. Remember that Git stores snapshots of our content, not sets of changes, as its primary data structure. In order to have a `diff` command, it needs to be able to calculate what's changed between versions of a file on the fly so it can display those changes to us. And diffs are not only useful for displaying changes to the user; they are also an important building block inside the `merge` command¹, whose internal workings are based on calculating changes done to a file on parallel development branches.

In this chapter, we'll walk through the default diff algorithm used by Git. It was developed by Eugene W. Myers, and the original paper is available online². While the paper is quite short, it is also mathematically dense and is focussed on proving properties of the algorithm. The explanations here will be less rigorous, but will hopefully be more intuitive, giving a detailed example of what the algorithm actually does and how it works.

11.1. What's in a diff?

Before we launch into investigating the algorithm, it's worth spending some time thinking about what a diff is and why we care about how they are generated. This will illuminate some of the properties that diff algorithms are designed to exhibit.

Say we want to calculate the difference between two strings of letters:

- $a = \text{ABCABBA}$
- $b = \text{CBABAC}$

By *difference*, we mean a sequence of edits that will convert string a into string b . One possible such sequence is to simply delete each character in a , and then insert each character in b , or to use common diff notation:

```
- A  
- B  
- C  
- A  
- B  
- B  
- A  
+ C  
+ B
```

¹Chapter 20, *Merging inside files*

²<http://www.xmailserver.org/diff2.pdf>

```
+ A
+ B
+ A
+ C
```

However, we wouldn't consider this a good-quality diff since it doesn't tell us anything interesting. Changes to source code typically leave much of a file unmodified and we really want to see specific sections of code that were inserted or deleted. A diff that shows the entire file being removed and replaced with the new version isn't much use to us. A diff that doesn't attempt to isolate which regions of a file were changed will also lead to more merge conflicts.

A better diff of these two strings would be:

```
- A
- B
C
+ B
A
B
- B
A
+ C
```

This makes the smallest possible number of changes to *a* in order to produce *b*, so it's a better visualisation of what changed. It's not the only possible solution, for example these are also valid edit scripts:

Figure 11.1. Three equivalent diffs

1. - A - B C - A B + A B A + C	2. - A + C B - C A B - B A + C	3. + C - A B - C A B - B A + C
--	--	--

However, the property they have in common is that they are all *minimal*: they make the smallest number of edits possible, which in this case is five. In general, trying to minimise the number of edits displayed when comparing code will result in diffs that make more sense to a human reader, and will also tend to make conflicts less likely when the diff algorithm is used to merge files.

What's interesting about these examples is that they differ in which sections they consider to be the same between the strings, and in which order they perform edits. From looking at diffs, you probably have an intuitive idea that diffs only show the things that changed, but these examples show that there are many possible interpretations of the difference between two files, particularly when the files contain many lines with the same content.

So, the purpose of a diff algorithm is to provide a strategy for generating diffs that have certain desirable properties. We usually want diffs to be as small as possible, but there are

other considerations. For example, when you change something, you're probably used to seeing deletions followed by insertions, not the other way round. That is, you'd rather see solution 2 than solution 3 above. And, when you change a whole block of code, you'd like to see the whole chunk being deleted followed by the new code being inserted, rather than many deletions and insertions interleaved with each other.

Figure 11.2. Interleaving deletions and insertions

Good:	- one	Bad:	- one
	- two		+ four
	- three		- two
	+ four		+ five
	+ five		+ six
	+ six		- three

You also probably want to see deleted or inserted code that aligns with your idea of the code's structure. For example, if you insert a method, you'd like that method's end to be considered new, rather than the end of the preceding method:

Figure 11.3. Aligning diff changes with logical code blocks

Good:	class Foo	Bad:	class Foo
	def initialize(name)		def initialize(name)
	@name = name		@name = name
	end		+ end
+		+	
+	def inspect	+	def inspect
+	@name	+	@name
+	end		end
	end		end

Myers' algorithm is just one such strategy, but it's fast and it produces diffs that tend to be of good quality most of the time. It does this by being *greedy*, that is trying to consume as many lines that are the same before making a change (thereby avoiding the 'wrong end' problem), and also by preferring deletions over insertions when given a choice, so that deletions appear first.

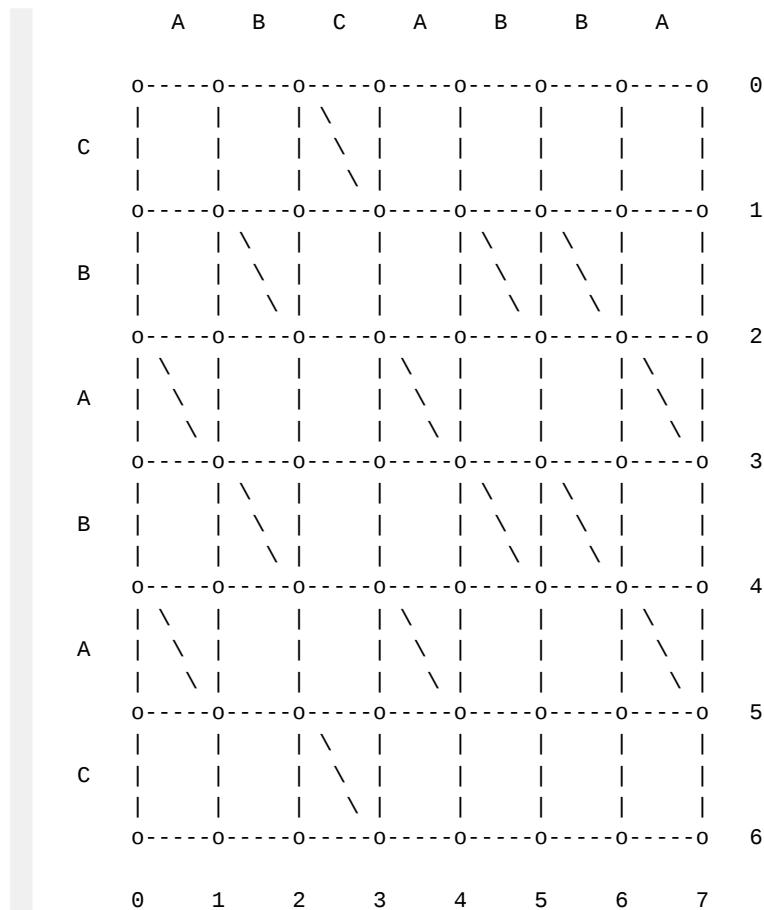
11.2. Time for some graph theory

Now that we've thought about what a diff algorithm is designed to achieve, we can begin to look specifically at the Myers algorithm. It is based on the idea that finding the *shortest edit script* (SES) can be modelled as a graph search. Let's take our two strings, $a = \text{ABCABBA}$ and $b = \text{CBABAC}$, and build a graph of all the ways we can get from a to b .

The (x, y) co-ordinates in the grid shown below correspond to steps in the editing process; at $(0,0)$ we have string a , that is, we have not started editing. Moving rightward (increasing x) corresponds to deleting a character from a , for example moving to $(1,0)$ means we've deleted the first A from a , so our edited string is now BCABBA. Moving downward (increasing y) corresponds to inserting a character from b , for example if we now move from $(1,0)$ down to $(1,1)$, we insert the first C from b , and our edited string is thus CBCABBA. At position $(4,3)$, we have converted ABCA into CBA, but we still need to convert the trailing BBA of a into BAC. The bottom-right position $(7,6)$ corresponds to having converted string a fully into string b .

As well as moving rightward and downward, in some positions we can also move diagonally. This occurs when the two strings have the same character at the position's indexes, for example the third character in a and the first character in b are both c , and so position $(2,0)$ has a diagonal leading to $(3,1)$. This corresponds to consuming an equal character from both strings, neither deleting nor inserting anything.

Figure 11.4. Edit graph for converting ABCABBA into CBABAC



The idea behind the Myers algorithm is quite simple: we want to get from $(0,0)$ to $(7,6)$ (the bottom-right) in as few moves as possible. There is a general-purpose method for finding the shortest path between two points in any graph: Dijkstra's algorithm³. However, certain structural properties of these edit graphs mean that the shortest path can be found using a method that takes less time and space than Dijkstra's general-purpose method. A modification to the base algorithm allows it to also consume a linear (rather than quadratic) amount of memory as it runs.

In Myers' method, a *move* is a single step rightward (a deletion from a) or downward (an insertion from b). The most number of moves we could take to get from a to b is 13: the total length of the two strings. This can be done by moving rightwards all the way to $(7,0)$ and then downwards to $(7,6)$.

However, diagonal paths correspond to consuming a matching character from both strings without performing any edits, and so walking them is free. We want to maximise the number of

³https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

diagonal steps we take and minimise the number of rightward/downward moves. Another way of putting this is that we want to find the *longest common subsequence*⁴ (LCS) of a and b , that is the longest sequence of characters that appear in both strings and in the same order. As the Myers paper mentions, finding the SES and the LCS are equivalent problems. The examples above show that we can actually get from a to b making only five edits, and Myers provides a strategy for finding that pathway.

11.2.1. Walking the graph

To develop an intuition for how the algorithm works, let's start exploring the graph. To try to find the shortest path to the bottom-right position, we'll explore every possible path from (0,0) in tandem until we find a path that reaches the end. I recommend keeping the above grid handy while you follow this.

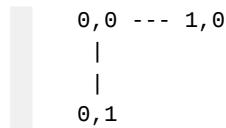
Let's start by recording our initial position at (0,0).

Figure 11.5. Initial graph exploration state



We have two options from this position: we can move downward and reach (0,1) or move rightward and reach (1,0). We'll record this pictorially using lines to represent the moves we've taken and which direction they were in. The numbers at each point represent the grid position we reach after taking the moves that led to it.

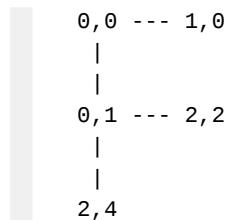
Figure 11.6. Exploration state after a single move



Now let's consider (0,1). If we move downward from here we reach (0,2), but there is a diagonal from there to (1,3), and from (1,3) to (2,4), and since diagonal moves are free we can say that moving downward from (0,1) gets us to (2,4) at the cost of only one move. Therefore we'll mark the move from (0,1) to (2,4) as a single step in our walk.

Moving rightward from (0,1) takes us to (1,1) and again there is a diagonal from there to (2,2). Let's mark both these moves on our walk.

Figure 11.7. Partial exploration state for two moves

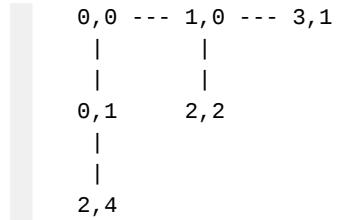


⁴https://en.wikipedia.org/wiki/Longest_common_subsequence_problem

Now let's consider the other branch we took from (0,0), moving rightward to (1,0). Moving downward from (1,0) takes us to (1,1), which as we just found out gets us to (2,2). Moving rightward from (1,0) takes us to (2,0), which has a diagonal to (3,1). Again, we'll record both these steps.

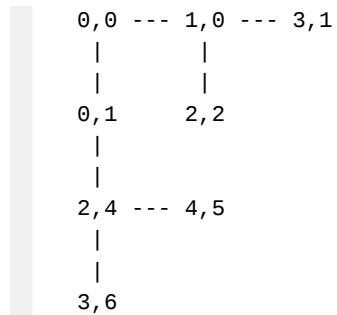
I'm recording (2,2) as being visited via (1,0) rather than (0,1) for reasons that will become clear a little later. For intuition, consider that making a rightward move first means performing a deletion first, and we generally want deletions to appear before insertions.

Figure 11.8. Complete exploration state after two moves



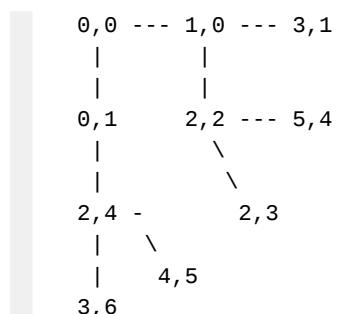
We've now fully explored the graph to two moves deep and we can begin our third move. Moving downward from (2,4) gets us to (2,5), and from there is a diagonal to (3,6). Moving rightward from (2,4) takes us to (3,4), where again a diagonal takes us to (4,5).

Figure 11.9. First explorations for the third move



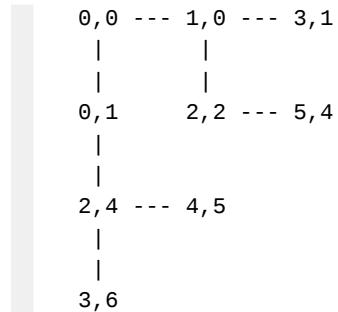
Next, we consider (2,2). Moving rightward from there is as we've seen before: we move to (3,2), and follow the diagonals from there to (5,4). Moving downward introduces a new situation, however: this move gets us to (2,3) and there is no diagonal from there. Now, if we were doing a general-purpose graph search, we'd want to record both the result of moving rightward from (2,4) and the result of moving downward from (2,2), that is:

Figure 11.10. Second explorations for the third move



However, the structure of the particular graphs we're examining means that it's sufficient to just store the *best* position you can reach after a certain set of edits. The above record shows us that making two insertions then a deletion (down twice, and then right) gets us to (4,5), whereas making the deletion first, and then the two insertions, gets us to (2,3). So, we'll just keep the (4,5) result and throw the (2,3) away, indicating (4,5) is the best position reachable after one deletion and two insertions *in any order*.

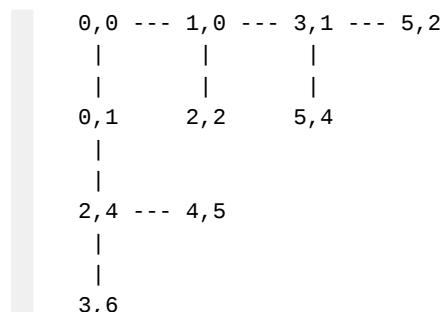
Figure 11.11. Discarding the worse result



Finally in our depth-2 scan, we visit (3,1). Moving downward from there goes to (3,2), which leads diagonally to (5,4), and so we'll again record this as a move downward from (3,1) rather than rightward from (2,2). Moving rightward from (3,1) gives (4,1), which has a diagonal to (5,2).

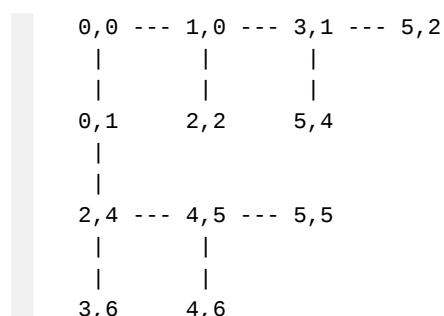
Here's the completed record after three moves:

Figure 11.12. Complete exploration state after three moves



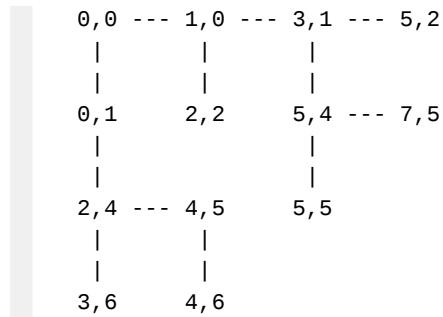
You're probably getting the hang of this by now so let's rattle through the remaining moves. We can't move downward from (3,6), and moving rightward from there gives (4,6), which is also reachable downward from (4,5), so we'll mark it as such. Rightward of (4,5) is (5,5).

Figure 11.13. First explorations for the fourth move



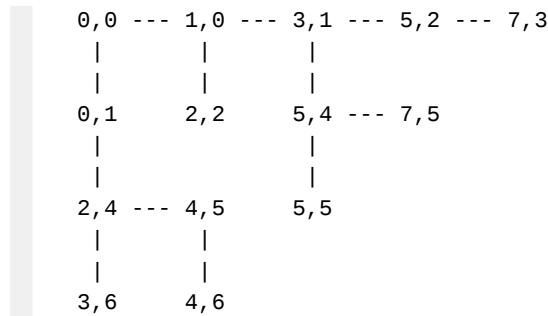
(5,5) is also downward of (5,4) so we'll mark that, and moving rightward from (5,4) gives (6,4), with a diagonal leading to (7,5).

Figure 11.14. Second explorations for the fourth move



Downward from (5,2) also leads to (7,5), and moving rightward from (5,2) leads to (7,3), thus completing the fourth row of the scan.

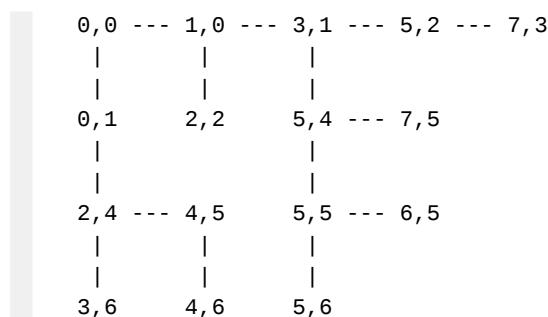
Figure 11.15. Complete exploration state after four moves



Now we begin the fifth row. Since we know there are edits from a to b requiring only five edits, we expect this row of the scan to find the bottom-right position, (7,6).

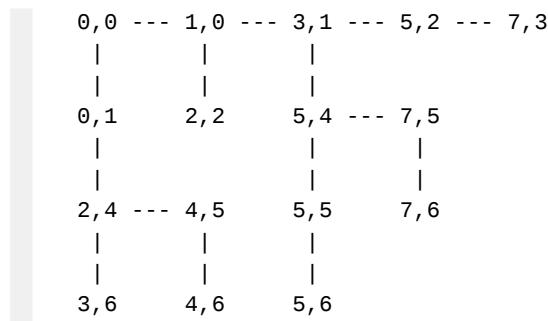
There is nothing downward from (4,6), and rightward of that is (5,6), which is also downward from (5,5). Rightward of (5,5) is (6,5).

Figure 11.16. Partial exploration state for five moves



Finally, moving downward from (7,5) gives (7,6) — the final position! This is certainly better than (6,5), which we reached by going right, right, down, down, right, and so we replace it in our trace of the moves.

Figure 11.17. Reaching the bottom-right of the graph

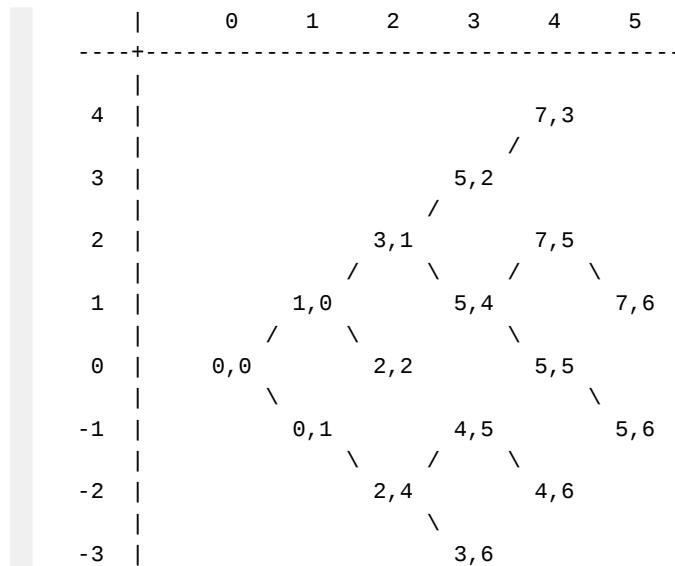


So that's the basic idea the algorithm is based on: given two strings, find the shortest path through a graph that represents the edit space between the two. We explore every possible route through the graph breadth-first, and stop as soon as we reach the final position. To turn this into working code, we first need to look at a space-saving trick the algorithm uses to store the above search in memory.

11.2.2. A change of perspective

Having seen how the graph search works, we're going to change the representation slightly to get us toward how the Myers algorithm actually works. Imagine that we take the above graph walk and render it rotated by 45 degrees. The number along the horizontal axis, d , is the depth we've reached in the graph, i.e. how many moves we've made so far, remembering that diagonal moves are free. The number along the vertical axis we call k , and notice that for every move in the graph, $k = x - y$ for each move on that row.

Figure 11.18. Exploration state in d - k space



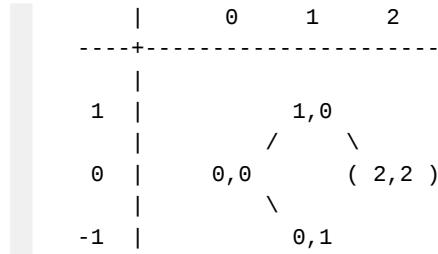
Moving rightward increases x , and so increases k by 1. Moving downward increases y and so decreases k by 1. Moving diagonally increases both x and y , and so it keeps k the same. So, each time we make a rightward or downward move followed by a chain of diagonals, we either increment or decrement k by 1. What we are recording is the furthest through the edit graph we can reach for each value of k , at each step.

Here's how the algorithm proceeds. For each d beginning with 0, we fill in each move for k from $-d$ to $+d$ in steps of 2. Our aim at each (d, k) position is to determine the best move we can make from the previous position. The best move is the one that gives us the highest x value; maximising x rather than y means we prioritise deletion over insertion.

To discover the best move, we need to decide whether we should pick a downward move from $(d - 1, k + 1)$, or a rightward move from $(d - 1, k - 1)$. If k is $-d$ then the move must be downward, likewise if k is $+d$ then we must move rightward. For all other values of k , we pick the position with the highest x from the two adjacent k slots in the previous column, and determine where that move leads us.

For example, consider the move at $(d, k) = (2, 0)$. We can either move rightward from $(d, k) = (1, -1)$ where $(x, y) = (0, 1)$, or downward from $(d, k) = (1, 1)$ where $(x, y) = (1, 0)$.

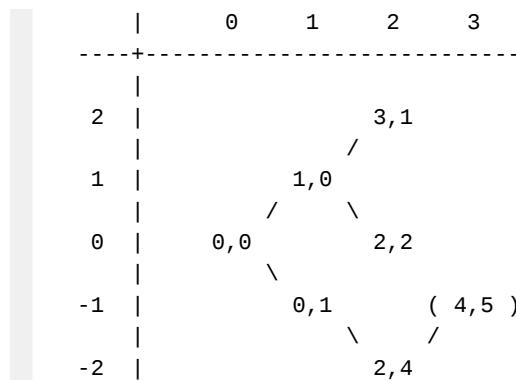
Figure 11.19. Picking a move from the higher x value



$(1, 0)$ has a higher x value than $(0, 1)$, so we pick a move downward from $(1, 0)$ to $(1, 1)$, which leads us to $(2, 2)$ diagonally. Therefore we record $(x, y) = (2, 2)$ for $(d, k) = (2, 0)$. This explains why we recorded the move via this path when $(2, 0)$ is also reachable by going rightward from $(0, 1)$; picking the previous position with the highest x value means we try to maximise the number of deletions we make before trying insertions.

In some situations, the two previous positions will have the same x value. For example, consider the move at $(d, k) = (3, -1)$, where we can move downward from $(x, y) = (2, 2)$ or rightward from $(x, y) = (2, 4)$. Moving rightward will increase x , so we move from $(2, 4)$ to $(3, 4)$ and then diagonally to $(4, 5)$.

Figure 11.20. Picking a rightward move when both predecessors have equal x



There are a few final simplifications that get us to the algorithm as presented in the paper. The first is that, since we're storing each (x, y) position indexed against k , and $k = x - y$, we don't

need to store y since it can be calculated from the values of k and x . The second is that we don't need to store the direction of the move taken at each step, we just store the best x value we can achieve at each point. The path will be derived *after* we've completed this process to find the smallest d that gets us to the bottom-right position; once we know where the final position shows up we can backtrack to find which single path out of the many we've explored will lead us there.

Removing those details leaves us with this information:

Figure 11.21. Exploration state with redundant data removed

	0	1	2	3	4	5
4					7	
3				5		
2			3		7	
1		1		5		7
0	0		2		5	
-1		0		4		5
-2			2		4	
-3				3		

The final simplification is that the x values in round d depend only on those in round $d - 1$, and because each round alternately modifies either the odd or the even k positions, each round does not modify the values it depends on from the previous round. Therefore the x values can be stored in a single flat array, indexed by k . In our example, this array would evolve as follows with each value of d :

Figure 11.22. State array after each move

k	-3	-2	-1	0	1	2	3	4
$d = 0$				0				
$d = 1$				0	0	1		
$d = 2$		2	0	2	1	3		
$d = 3$	3	2	4	2	5	3	5	
$d = 4$	3	4	4	5	5	7	5	7
$d = 5$	3	4	5	5	7	7	5	7

The iteration stops when we discover we can reach $(x, y) = (7, 6)$ at $(d, k) = (5, 1)$.

We've now arrived at the representation of the problem used in the algorithm, and we can translate this into working code.

11.2.3. Implementing the shortest-edit search

We'll create a method called `Diff.diff(a, b)` that takes two lists of strings, `a` and `b`. We usually want to compare files line-by-line, so we'll include a convenience method for coercing values into arrays, which splits the input into lines if it's a single string.

```
# lib/diff.rb

module Diff
  def self.lines(document)
    document.is_a?(String) ? document.lines : document
  end

  def self.diff(a, b)
    Myers.diff(Diff.lines(a), Diff.lines(b))
  end
end
```

Now we'll begin writing the `Myers` class that will implement the algorithm discussed above. To start with, we'll just make some boilerplate for storing the two lists as instance variables on an object that implements the `diff` method, which we'll define later once all the building blocks are in place.

```
# lib/diff/myers.rb

module Diff
  class Myers

    def self.diff(a, b)
      Myers.new(a, b).diff
    end

    def initialize(a, b)
      @a, @b = a, b
    end

    def diff
      # ...
    end
  end
end
```

To return a diff, we need to find the shortest edit script. We begin by storing `n` as the size of `@a` and `m` as the size of `@b`, and `max` as the sum of those; that's the most number of moves we might need to make. Here's the beginning of the `Diff::Myers#shortest_edit` method, which will continue in the snippets below.

```
# lib/diff/myers.rb

def shortest_edit
  n, m = @a.size, @b.size
```

```

max = n + m

# ...
end

```

Then we set up an array to store the latest value of x for each k . k can take values from $-max$ to max , and in Ruby a negative array index is interpreted as reading from the end of the array. The actual order of the elements doesn't matter, we just need the array to be big enough so that there's space for the positive and negative k values.

We set $v[1] = 0$ so that the iteration for $d = 0$ picks $x = 0$. We need to treat the $d = 0$ iteration just the same as the later iterations since we might be allowed to move diagonally immediately. Setting $v[1] = 0$ makes the algorithm behave as though it begins with a virtual move downwards from $(x, y) = (0, -1)$.

```

# lib/diff/myers.rb

v = Array.new(2 * max + 1)
v[1] = 0

```

Next, we begin a nested loop: we iterate d from 0 to max in the outer loop, and k from $-d$ to d in steps of 2 in the inner loop.

```

# lib/diff/myers.rb

(0 .. max).step do |d|
  (-d .. d).step(2) do |k|

```

Within the loop, we begin by choosing whether to move downward or rightward from the previous round. If k equals $-d$, or if it's not equal to d and the $k+1$ x -value is greater than the $k-1$ x -value, then we move downward, i.e. we take the x value as being equal to the $k+1$ x -value in the previous round. Otherwise we move rightward and take x as one greater than the previous $k-1$ x -value. We calculate y from this chosen x value and the current k .

```

# lib/diff/myers.rb

if k == -d || (k != d &amp; v[k + 1] < v[k - 1])
  x = v[k + 1]
else
  x = v[k - 1] + 1
end

y = x - k

```

Having taken a single step rightward or downward, we see if we can take any diagonal steps. As long as we've not deleted the entire $@a$ string or added the entire $@b$ string, and the elements of each string at the current position are the same, we can increment both x and y . Once we finish moving, we store off the value of x we reached for the current k .

```

# lib/diff/myers.rb

while x < n &amp; y < m &amp; @a[x] == @b[y]
  x, y = x + 1, y + 1
end

```

```
v[k] = x
```

Finally, we return the current value of d if we've reached the bottom-right position, telling the caller the minimum number of edits required to convert from a to b .

```
# lib/diff/myers.rb

return d if x >= n and y >= m
```

This minimal version of the algorithm calculates the smallest number of edits we need to make, but not what those edits are. To do that, we need to backtrack through the history of x values generated by the algorithm.

11.3. Retracing our steps

The graph search encoded in the `shortest_edit` method lets us explore all possible edit paths until we find the shortest path necessary to reach the end. Once we've got there, we can look at the data we've recorded in reverse to figure out which single path led to the result.

Let's look at our example again. Recall that the `shortest_edit` method records the best value of x we can reach at each (d, k) position:

Figure 11.23. Completed exploration state for $\text{diff}(\text{ABCABBA}, \text{ CBABAC})$

	0	1	2	3	4	5
4					7	
3				5		
2			3		7	
1		1		5		7
0	0		2		5	
-1		0		4		5
-2			2		4	
-3				3		

We know the final position is at $(x, y) = (7, 6)$, so $k = 1$, and we found it after $d = 5$ steps. From $(d, k) = (5, 1)$, we can track backward to either $(4, 0)$ or $(4, 2)$:

Figure 11.24. Backtracking choices for $(d, k) = (5, 1)$

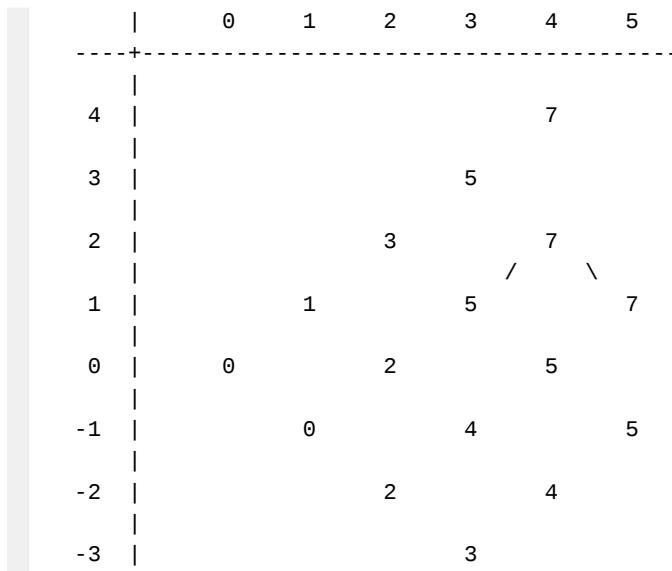
	0	1	2	3	4	5
4					7	
3				5		
2			3		(7)	
1		1		5		[7]
0	0		2		(5)	
-1		0		4		5
-2			2		4	
-3				3		

We see that $(4,2)$ contains the higher x value, and so according to the logic of `shortest_edit` we must have reached $(5,1)$ via a downward move from there. This tells us that the (x, y) position before $(7,6)$ was $(7,5)$.

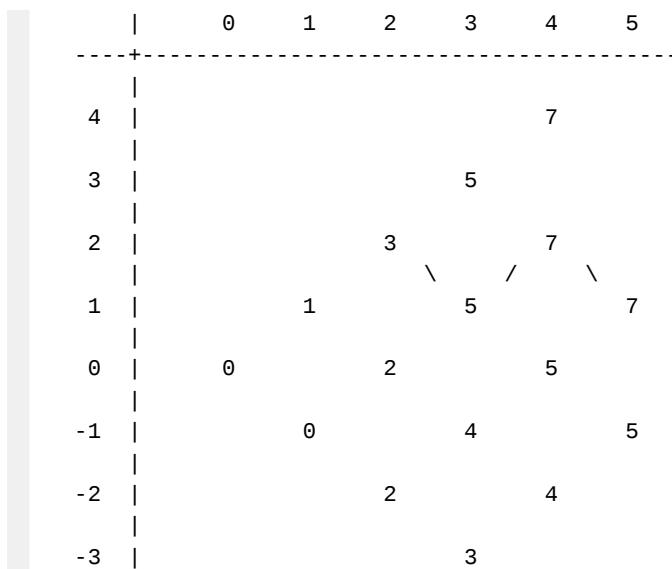
 Figure 11.25. Choosing the previous highest x value

	0	1	2	3	4	5
4					7	
3				5		
2			3		7	\
1		1		5		7
0	0		2		5	
-1		0		4		5
-2			2		4	
-3				3		

Via a similar argument, to move back from $(d, k) = (4,2)$ we must have come from $(3,1)$ or $(3,3)$. These positions have an equal x value, so we would have chosen to move rightward from $(3,1)$, or from $(x, y) = (5,4)$ to $(7,5)$.

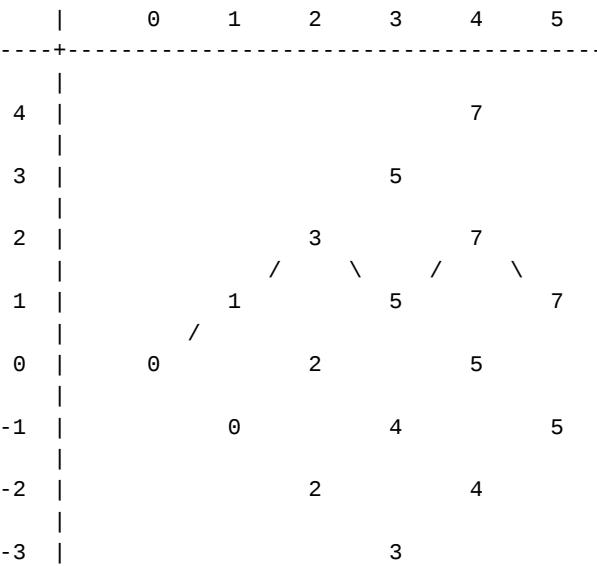
Figure 11.26. Backtracking from $(d, k) = (4,2)$


Before that, we have a choice between $(d, k) = (2,0)$ and $(2,2)$, and $(2,2)$ has the higher x value.

 Figure 11.27. Backtracking from $(d, k) = (3,1)$


After this, we're on the edge of the grid and we can only have made rightward moves.

Figure 11.28. Backtracking to the starting position (0,0)



We've walked all the way back to the start, and now we know that the (x, y) positions of each move are as follows; remember that $y = x - k$.

$(0, 0) \rightarrow (1, 0) \rightarrow (3, 1) \rightarrow (5, 4) \rightarrow (7, 5) \rightarrow (7, 6)$

These positions are enough to figure out the diagonal moves between each position: we decrement both x and y until one of them is equal to the values in the previous position, and then we end up a single downward or rightward move away from that position. For example, to get from (5,4) back to (3,1), we observe that $5 > 3$ and $4 > 1$, and so we can step diagonally back one step to (4,3). This still has neither co-ordinate equal to (3,1) so we take another step back to (3,2). Now, we've reached the same x value as (3,1), so we must make an upward step from (3,2) to (3,1) to complete the move.

11.3.1. Recording the search

To perform this backtracking, we need to make a small adjustment to `shortest_edit`. Rather than just recording the latest x value for each k , we need to keep a trace of the array before each turn of the outer loop.

We add the variable `trace` to store these snapshots of `v`, and push a copy of `v` into it on each loop. Rather than just returning the number of moves required, we now return this trace to the caller.

```
# lib/diff/myers.rb

v      = Array.new(2 * max + 1)
v[1]  = 0
trace = []

(0 .. max).step do |d|
  trace.push(v.clone)

  (-d .. d).step(2) do |k|
    # calculate the next move as before
  end
end

return trace if x >= n and y >= m
```

```
    end
end
```

At the end of the method, `trace` will contain enough information to let us reconstruct the path that leads to the final position. It essentially contains the best x value that we found at each (d, k) , as we saw earlier. Each copy of `v` that we stored corresponds to a column from our search history.

11.3.2. And you may ask yourself, how did I get here?

Now we can implement the backtracking in code. We can write a method that takes a trace produced by the `shortest_edit` method, and the target final (x, y) position, and works out the backtrace. For each move, it yields the x and y values before and after the move.

The method iterates over the trace in reverse order. `each_with_index` generates a list that pairs each copy of `v` in the trace up with its corresponding `d` value, and then `reverse_each` iterates over that list backwards.

```
# lib/diff/myers.rb

def backtrack
  x, y = @a.size, @b.size

  shortest_edit.each_with_index.reverse_each do |v, d|
    # ...
  end
end
```

At each step of the trace, we calculate the k value, and then determine what the previous k would have been, using the same logic as in the `shortest_edit` method:

```
# lib/diff/myers.rb

k = x - y

if k == -d or (k != d and v[k - 1] < v[k + 1])
  prev_k = k + 1
else
  prev_k = k - 1
end
```

From that previous k value, we can retrieve the previous value of x from the trace, and use these k and x values to calculate the previous y .

```
# lib/diff/myers.rb

prev_x = v[prev_k]
prev_y = prev_x - prev_k
```

Then we begin yielding moves back to the caller. If the current x and y values are both greater than the previous ones, we know we can make a diagonal move, so we yield a move from $x-1, y-1$ to x, y and then decrement x and y as long as this condition holds.

```
# lib/diff/myers.rb

while x > prev_x and y > prev_y
```

```

    yield x - 1, y - 1, x, y
    x, y = x - 1, y - 1
end

```

Finally, we yield a move from the previous x and y from the trace, to the position we reached after following diagonals. This should be a single downward or rightward step. If d is zero, there is no previous position to move back to, so we skip this step in that case. After yielding this move, we set x and y to their values in the previous round, and continue the loop.

```

# lib/diff/myers.rb

yield prev_x, prev_y, x, y if d > 0

x, y = prev_x, prev_y

```

Running this on our example inputs, this method yields the following pairs of positions, which you should verify correspond to the full path through the edit graph, in reverse:

```

(7, 5) -> (7, 6)
(6, 4) -> (7, 5)
(5, 4) -> (6, 4)
(4, 3) -> (5, 4)
(3, 2) -> (4, 3)
(3, 1) -> (3, 2)
(2, 0) -> (3, 1)
(1, 0) -> (2, 0)
(0, 0) -> (1, 0)

```

To glue everything together, we need a method that takes two texts to diff, calls `shortest_edit` to get a trace, and then runs that trace through `backtrack` to generate a diff: a sequence of lines marked as either deletions, insertions, or equal lines.

We'll need a structure to represent lines of a diff, which we'll call *edits*. Let's include a convenient method for turning an edit into a string, which puts the right symbol at the start of the line of text for when it's printed as part of a patch.

```

# lib/diff.rb

SYMBOLS = {
  :eq => " ",
  :ins => "+",
  :del => "-"
}

Edit = Struct.new(:type, :text) do
  def to_s
    SYMBOLS.fetch(type) + text
  end
end

```

As `backtrack` yields each pair of positions in reverse, this method pushes a diff edit onto an array. If the x values are the same between positions, we know it's a downward move, or an insertion; if the y values are the same then it's a deletion, otherwise the lines are equal. To build the diff, we select lines from each file as appropriate: a line from `a` for deletions, from `b` for insertions, and (arbitrarily) from `a` for equal lines. We return the reversed array at the end to give the edits in the right order.

```
# lib/diff/myers.rb

def diff
  diff = []

  backtrack do |prev_x, prev_y, x, y|
    a_line, b_line = @a[prev_x], @b[prev_y]

    if x == prev_x
      diff.push(Edit.new(:ins, b_line))
    elsif y == prev_y
      diff.push(Edit.new(:del, a_line))
    else
      diff.push(Edit.new(:eql, a_line))
    end
  end

  diff.reverse
end
```

This completes the algorithm; we can now take our example inputs, pass them to `Diff.diff`, and get a list of edits that we can print:

```
require "./lib/diff"

a = "ABCABBA".chars
b = "CBABAC".chars

edits = Diff.diff(a, b)
edits.each { |edit| puts edit }
```

Running this program prints:

```
-A
-B
C
+B
A
B
-B
A
+C
```

Just as we expected. We can use this new `Diff` module to build the patch output for the `diff` command (and other commands that can be made to display patches, like `log`), and it will also be an important ingredient when we come to implement the `merge` command.

Although this is certainly one of the more theory-intensive elements of building a version control system, the code that results from it is actually pretty short and simple — around 50 lines of Ruby if you ignore class and method definition boilerplate. As we'll see as we progress, this one bit of dense maths-heavy code buys us a lot of power in displaying and manipulating code changes.

12. Spot the difference

In the last chapter we developed an important building block that underlies many of Git's commands: a diff algorithm for computing the shortest edit script between two files. With that in place, let's add a command that improves upon status by showing us a line-by-line account of exactly what's changed within each file: the `diff` command.

Before we can start building `diff`, though, we need a little refactoring.

12.1. Reusing status

The `diff` command, when called with no arguments, displays the files in the *changes not staged for commit* section of the status output, i.e. the files that have changed in the workspace and haven't been updated in the index. If you run `diff --cached`, it instead shows you the differences between HEAD and the index — the *changes to be committed*.

Since it needs to know about which files have changed between HEAD, the index and the workspace, it would be good if `diff` could reuse the status logic for discovering which files have changed. Let's see if we can extract this information from the `status` command and reuse it.

The `Command::Status` class contains a `run` method that sets up some variables to hold state, loads the index, collects all the change data, saves the index, then prints the results. I've included this method in full below, followed by just the first lines of all the other methods in the class.

```
# lib/command/status.rb

module Command
  class Status < Base

    def run
      @stats           = {}
      @changed         = SortedSet.new
      @index_changes   = SortedHash.new
      @workspace_changes = SortedHash.new
      @untracked_files = SortedSet.new

      repo.index.load_for_update

      scan_workspace
      load_head_tree
      check_index_entries
      collect_deleted_head_files

      repo.index.write_updates

      print_results
      exit 0
    end

    private

    def print_results
    def print_long_format
```

```
def print_changes(changeset, style)
def print_commit_status
def print_porcelain_format
def status_for(path)
def record_change(path, set, type)
def scan_workspace(prefix = nil)
def trackable_file?(path, stat)
def load_head_tree
def read_tree(tree_oid, pathname = Pathname.new(""))
def check_index_entries
def check_index_against_workspace(entry)
def check_index_against_head_tree(entry)
def collect_deleted_head_files

end
end
```

It's easy to miss as we were building this up incrementally, but this class is quite large. Looking through the list of methods, they broadly fall into two groups: those from `record_change` on down are, for want of a better term, *business logic*: they go and gather all the information for the user interface to display, and they contain the rules for deciding what has changed. The methods from `print_results` to `status_for` are display logic: they convert the data gathered by the business logic into some format that appears on your screen.

If we extract all the business methods into their own class, then we can use them inside other commands, like `diff`. This will leave the `Command::Status` class containing only logic about how the status information should be displayed, as shown below.

```
# lib/command/status.rb

module Command
  class Status < Base

    def run
      repo.index.load_for_update
      @status = repo.status
      repo.index.write_updates

      print_results
      exit 0
    end

    private

    def print_results
    def print_long_format
    def print_changes(changeset, style)
    def print_commit_status
    def print_porcelain_format
    def status_for(path)

  end
end
```

I've left the calls to load and write the index here, because putting them inside the status logic would make it harder to use; if the status logic called `Index#load_for_update` but it was being

called from a command that's already loaded the index for some other reason, that will cause bugs. The index is effectively a piece of global state, and its loading should be controlled by the top-level command. This also gives commands a choice over whether they update the index; although it uses status information, the `diff` command does not update the index like the `status` command does.

So, `Command::Status` now gets the information it needs by calling `Repository#status`, which is defined simply as instantiating another class, `Repository::Status`:

```
# lib/repository.rb
```

```
def status
  Status.new(self)
end
```

The `Repository::Status` class is where we've now put all the business logic. It takes a `Repository` as input and performs all the same steps that were previously being done in `Command::Status`. It exposes attributes for all the information the `status` command wants to display.

```
# lib/repository/status.rb
```

```
class Repository
  class Status

    attr_reader :changed,
                :index_changes,
                :workspace_changes,
                :untracked_files

    def initialize(repository)
      @repo = repository
      @stats = {}

      @changed = SortedSet.new
      @index_changes = SortedHash.new
      @workspace_changes = SortedHash.new
      @untracked_files = SortedSet.new

      scan_workspace
      load_head_tree
      check_index_entries
      collect_deleted_head_files
    end

    private

    def record_change(path, set, type)
    def scan_workspace(prefix = nil)
      # etc.
    end
  end
end
```

We can now reuse the `Repository::Status` class to power our `diff` implementation.

12.2. Just the headlines

As we're adding a new command, I'll go into `lib/command.rb` and add the boilerplate to load the new command class and store it in the list of command names.

```
# lib/command.rb

require_relative "./command/add"
require_relative "./command/commit"
require_relative "./command/diff"
require_relative "./command/init"
require_relative "./command/status"

module Command
  Unknown = Class.new(StandardError)

  COMMANDS = {
    "init"  => Init,
    "add"   => Add,
    "commit" => Commit,
    "status" => Status,
    "diff"   => Diff
  }

  #
end
```

Now, running `git diff` shows me the following output:

```
diff --git a/lib/command.rb b/lib/command.rb
index 182dce2..5b09b2b 100644
--- a/lib/command.rb
+++ b/lib/command.rb
@@ -1,5 +1,6 @@
  require_relative "./command/add"
  require_relative "./command/commit"
+require_relative "./command/diff"
  require_relative "./command/init"
  require_relative "./command/status"

@@ -10,7 +11,8 @@ module Command
  "init"  => Init,
  "add"   => Add,
  "commit" => Commit,
- "status" => Status
+ "status" => Status,
+ "diff"   => Diff
}

def self.execute(dir, env, argv, stdin, stdout, stderr)
```

There are several distinct pieces of information here. First is the `diff` line that tells us the names of the two things being compared. Git's `diff` command can be used to compare arbitrary pairs of files, but this line is telling us it's comparing the same filename from two different sources, which it calls `a` and `b`.

Next is the `index` line, which shows two short hexadecimal numbers and what looks like the file's mode, `100644`. These two hexadecimal numbers are the object ID stored in the index and the hash of the current file respectively. Indeed, we can see this by using our `Index` class:

```
>> index = Index.new(Pathname.new(".git/index"))
>> index.load
>> entry = index.each_entry.find { |e| e.path == "lib/command.rb" }
>> entry.oid
=> "182dce271bb629839f16a8c654a0bcba85aaabdb"
```

So the first number in the `index` line is `182dce2`, the first seven digits of the indexed blob ID. The second number, `5b09b2b`, is the first seven digits of the result of hashing the file as a blob, as `Database#hash_object` does.

```
>> db = Database.new(Pathname.new(".git/objects"))
>> blob = Database::Blob.new(File.read("lib/command.rb"))
>> db.hash_object(blob)
=> "5b09b2b204e775aa5c263bbe3e5ffe099eb70767"
```

So we know that to print this line, we need the blob ID from the index and the object hash of the current file, plus the file's mode.

The following two lines beginning `---` and `+++` are the start of the diff, and they repeat the filenames we saw on the `diff` line. Everything following these lines is patch information showing which lines have been changed. To begin with, we'll focus on getting these header lines right, before using the `Diff` module from Chapter 11, *The Myers diff algorithm* to generate patch information.

12.2.1. Unstaged changes

We'll begin building the `Command::Diff` class by loading the index, getting the repository status, and then just printing information for the files that are modified in the workspace.

```
# lib/command/diff.rb

def run
  repo.index.load
  @status = repo.status

  @status.workspace_changes.each do |path, state|
    case state
    when :modified then diff_file_modified(path)
    end
  end

  exit 0
end
```

`diff_file_modified` will need to generate the information we saw above, for which it will need to get the object IDs from the index and the workspace. This requires only a small addition to the `Index` class to allow us to look up a specific entry by its path.

```
# lib/index.rb

def entry_for_path(path)
  @entries[path.to_s]
```

```
end
```

The rest of the information we need can be found using existing methods.

```
# lib/command/diff.rb

def diff_file_modified(path)
    entry = repo.index.entry_for_path(path)
    a_oid = entry.oid
    a_mode = entry.mode.to_s(8)
    a_path = Pathname.new("a").join(path)

    blob = Database::Blob.new(repo.workspace.read_file(path))
    b_oid = repo.database.hash_object(blob)
    b_path = Pathname.new("b").join(path)

    puts "diff --git #{a_path} #{b_path}"
    puts "index #{short a_oid}..#{short b_oid} #{a_mode}"
    puts "--- #{a_path}"
    puts "+++ #{b_path}"
end

def short(oid)
    repo.database.short_oid(oid)
end
```

For printing the output, we've introduced a new method called `Database#short_oid`, which produces a shortened form of an object ID. Initially, this method will just truncate the ID to seven characters, but I've put the method in the `Database` class because it would be worth improving this operation so that it checks that the shortened ID identifies a unique object. If there is more than one object whose ID begins with these seven characters, then more characters should be added. For now, though, it's extremely unlikely that two objects will collide in this way¹, and this implementation will suffice.

```
# lib/database.rb

def short_oid(oid)
    oid[0..6]
end
```

Having made these changes, Jit is now able to partially report a diff of the files we've changed.

```
$ jit diff

diff --git a/lib/command.rb b/lib/command.rb
index 182dce2..5b09b2b 100644
--- a/lib/command.rb
+++ b/lib/command.rb
diff --git a/lib/database.rb b/lib/database.rb
index 8fb8b1b..63b2180 100644
--- a/lib/database.rb
+++ b/lib/database.rb
diff --git a/lib/index.rb b/lib/index.rb
index a088ac7..b15ff4f 100644
--- a/lib/index.rb
```

¹At the time of writing, the repository contains 327 objects. There are thirteen pairs sharing the first three digits of their ID, one pair sharing the first four digits, and no pairs sharing five or more digits.

```
+++ b/lib/index.rb
```

The other way in which a file can be reported as modified is if its mode has been changed. Let's see what Git reports if we change the mode of a file whose content is the same:

```
$ chmod +x Rakefile
$ git diff

diff --git a/Rakefile b/Rakefile
old mode 100644
new mode 100755
```

The `index` line and the `---` and `+++` lines that introduce the patch no longer appear, because there is no change in the file's contents. Instead, we have two new lines that report the change in the file's mode. What happens if we also change the file's contents?

```
diff --git a/Rakefile b/Rakefile
old mode 100644
new mode 100755
index c8c218e..b18c019
--- a/Rakefile
+++ b/Rakefile
@@ -1,3 +1,4 @@
+# a new line
require "rake/testtask"

Rake::TestTask.new do |task|
```

The `old mode` and `new mode` lines still appear, but now the `index` and patch lines are back. The only difference is the mode no longer appears at the end of the `index` line.

Let's adjust the `diff_file_modified` method to accommodate changed file modes. We need to add some code to get the mode of the changed file, which we get from the file's `stat()` information. Since the `Repository::Status` has already gathered this information for all the file's we're looking at, we could get the `File::Stat` object from there instead of asking `Workspace` for it a second time. We'll add an attribute to `Repository::Status` to expose this information.

```
# lib/repository/status.rb

attr_reader :stats
```

Now we can make the required changes to `Command::Diff`. If the modes differ then we print the `old mode` and `new mode` lines, and if the object IDs are the same then we return before printing any patch information.

```
# lib/command/diff.rb

def diff_file_modified(path)
  entry = repo.index.entry_for_path(path)
  a_oid = entry.oid
  a_mode = entry.mode.to_s(8)
  a_path = Pathname.new("a").join(path)

  blob = Database::Blob.new(repo.workspace.read_file(path))
  b_oid = repo.database.hash_object(blob)
  b_mode = Index::Entry.mode_for_stat(@status.stats[path]).to_s(8)
  b_path = Pathname.new("b").join(path)
```

```

    puts "diff --git #{ a_path } #{ b_path }"

unless a_mode == b_mode
  puts "old mode #{ a_mode }"
  puts "new mode #{ b_mode }"
end

return if a_oid == b_oid

oid_range = "index #{ short a_oid }..#{ short b_oid }"
oid_range.concat(" #{ a_mode }") if a_mode == b_mode

puts oid_range
puts "--- #{ a_path }"
puts "+++ #{ b_path }"
end

```

The final kind of change we can have in the workspace is that a file has been deleted. Let's move a file out of the repository and see how Git reports it.

```

diff --git a/Rakefile b/Rakefile
deleted file mode 100644
index c8c218e..0000000
--- a/Rakefile
+++ /dev/null
@@ -1,8 +0,0 @@
-require "rake/testtask"
-
-Rake::TestTask.new do |task|
-  task.libs << "test"
-  task.pattern = "test/**/*_test.rb"
-end
-
-task :default => :test

```

It shows the mode from the index next to `deleted file mode`, it shows `0000000` as the object ID for the missing file, and the `+++` line shows the path `/dev/null` rather than `b/Rakefile` as the `diff` line does. The patch shows the entire file being deleted.

We'll add another clause to the code in `Command::Diff#run` to handle deleted files:

```

# lib/command/diff.rb

@status.workspace_changes.each do |path, state|
  case state
  when :modified then diff_file_modified(path)
  when :deleted then diff_file_deleted(path)
  end
end

```

The `diff_file_deleted` method will have to do some of the same tasks as `diff_file_modified`. We still need the data from the index, but we can no longer read the file from the workspace. Instead, we use null values for all the file's data. We can also remove some of the code paths for printing the output since they don't apply in this case.

```
# lib/command/diff.rb
```

```
NULL_OID = "0" * 40
NULL_PATH = "/dev/null"

def diff_file_deleted(path)
  entry = repo.index.entry_for_path(path)
  a_oid = entry.oid
  a_mode = entry.mode.to_s(8)
  a_path = Pathname.new("a").join(path)

  b_oid = NULL_OID
  b_path = Pathname.new("b").join(path)

  puts "diff --git #{ a_path } #{ b_path }"
  puts "deleted file mode #{ a_mode }"
  puts "index #{ short a_oid }..#{ short b_oid }"
  puts "--- #{ a_path }"
  puts "+++ #{ NULL_PATH }"
end
```

With the addition of this method, Jit can display the correct header information for deleted files.

```
$ jit diff

diff --git a/Rakefile b/Rakefile
deleted file mode 100644
index c8c218e..0000000
--- a/Rakefile
+++ /dev/null
diff --git a/lib/command/diff.rb b/lib/command/diff.rb
index bb1996c..c983883 100644
--- a/lib/command/diff.rb
+++ b/lib/command/diff.rb
```

12.2.2. A common pattern

Before we move on and deal with displaying staged changes, we can restructure the code we have so far to make that easier. Although they are superficially different, the `diff_file_modified` and `diff_file_deleted` methods have some structural similarities. Both begin with exactly the same code to load an entry from the index, and when we work on displaying staged changes we'll also need to do this — the only difference is the index data will become the b rather than the a in the diff output. Then, the methods differ in how they load the current state of a file; one reads from the workspace while the other puts in some null values. Finally, they both print some output, which again has some superficial differences but could be unified into a single routine.

Let's separate concerns by reimaging the printing logic as a method `print_diff` that takes two inputs a and b and shows a comparison between them. It doesn't care how a and b were constructed, but it does require them both to respond to the methods `path`, `mode`, `oid` and `diff_path`. The code below is enough to cover both the modified and deleted file cases.

```
# lib/command/diff.rb

def print_diff(a, b)
  return if a.oid == b.oid and a.mode == b.mode

  a.path = Pathname.new("a").join(a.path)
```

```

b.path = Pathname.new("b").join(b.path)

puts "diff --git #{ a.path } #{ b.path }"
print_diff_mode(a, b)
print_diff_content(a, b)
end

def print_diff_mode(a, b)
  if b.mode == nil
    puts "deleted file mode #{ a.mode }"
  elsif a.mode != b.mode
    puts "old mode #{ a.mode }"
    puts "new mode #{ b.mode }"
  end
end

def print_diff_content(a, b)
  return if a.oid == b.oid

  oid_range = "index #{ short a.oid }..#{ short b.oid }"
  oid_range.concat(" #{ a.mode }") if a.mode == b.mode

  puts oid_range
  puts "--- #{ a.diff_path }"
  puts "+++ #{ b.diff_path }"
end

```

We'll invent a new data structure that fulfills the requirements of a and b. It will take path, oid and mode as inputs, and have a method diff_path that returns the path, or /dev/null if mode is not set.

```

# lib/command/diff.rb

Target = Struct.new(:path, :oid, :mode) do
  def diff_path
    mode ? path : NULL_PATH
  end
end

```

Now, we can extract the printing logic from diff_file_modified and diff_file_deleted and replace it with code that constructs two of these Target objects, and then passes them to print_diff.

```

# lib/command/diff.rb

def diff_file_modified(path)
  entry = repo.index.entry_for_path(path)
  a_oid = entry.oid
  a_mode = entry.mode

  blob = Database::Blob.new(repo.workspace.read_file(path))
  b_oid = repo.database.hash_object(blob)
  b_mode = Index::Entry.mode_for_stat(@status.stats[path])

  a = Target.new(path, a_oid, a_mode.to_s(8))
  b = Target.new(path, b_oid, b_mode.to_s(8))

  print_diff(a, b)

```

```

end

def diff_file_deleted(path)
  entry = repo.index.entry_for_path(path)
  a_oid = entry.oid
  a_mode = entry.mode

  a = Target.new(path, a_oid, a_mode.to_s(8))
  b = Target.new(path, NULL_OID, nil)

  print_diff(a, b)
end

```

Now we can clearly see that the methods differ only in the way they construct the `Target` objects. A `Target` can be created either from data in the index, or from a file in the workspace, or from nothing. Let's turn each of these constructions into its own method.

```

# lib/command/diff.rb

def from_index(path)
  entry = repo.index.entry_for_path(path)
  Target.new(path, entry.oid, entry.mode.to_s(8))
end

def from_file(path)
  blob = Database::Blob.new(repo.workspace.read_file(path))
  oid = repo.database.hash_object(blob)
  mode = Index::Entry.mode_for_stat(@status.stats[path])

  Target.new(path, oid, mode.to_s(8))
end

def from_nothing(path)
  Target.new(path, NULL_OID, nil)
end

```

Now, the `diff_file_modified` and `diff_file_deleted` methods are reduced to compositions of `print_diff` with pairs of these `from_` methods, and we can easily inline those into the main loop.

```

# lib/command/diff.rb

@status.workspace_changes.each do |path, state|
  case state
  when :modified then print_diff(from_index(path), from_file(path))
  when :deleted  then print_diff(from_index(path), from_nothing(path))
  end
end

```

12.2.3. Staged changes

Whereas running `git diff` with no arguments prints the differences between the index and the workspace, running `git diff --cached` (or its alias, `git diff --staged`) prints the *changes staged for commit*, the differences between the HEAD tree and the index.

To handle this, let's add a choice in `Command::Diff#run` to run `diff_head_index` if the `--cached` argument is given, or `diff_index_workspace` otherwise. `diff_head_index` prints

changes based on `Repository::Status#index_changes` rather than `workspace_changes`, and for now it only deals with modified files. `diff_index_workspace` contains the logic we previously had in the `run` method.

```
# lib/command/diff.rb

def run
  repo.index.load
  @status = repo.status

  if @args.first == "--cached"
    diff_head_index
  else
    diff_index_workspace
  end

  exit 0
end

private

def diff_head_index
  @status.index_changes.each do |path, state|
    case state
    when :modified then print_diff(from_head(path), from_index(path))
    end
  end
end

def diff_index_workspace
  @status.workspace_changes.each do |path, state|
    # ...
  end
end
```

This addition requires a new way of constructing a `Target` from information in the `HEAD` tree. Recall from Section 10.2, “`HEAD/index differences`” that `Repository::Status` loads the `HEAD` tree into an instance variable — rather than rebuild that information ourselves, let’s load it from there.

```
# lib/repository/status.rb

attr_reader :head_tree
```

Then we can use this to implement a `from_head` method in `Command::Diff`. Since both `Index::Entry` and `Database::Entry` respond to the methods `oid` and `mode`, we can extract the logic for building a `Target` from `from_index` and reuse it in `from_head`.

```
# lib/command/diff.rb

def from_head(path)
  entry = @status.head_tree.fetch(path)
  from_entry(path, entry)
end

def from_index(path)
  entry = repo.index.entry_for_path(path)
```

```
    from_entry(path, entry)
end

def from_entry(path, entry)
  Target.new(path, entry.oid, entry.mode.to_s(8))
end
```

With this small addition, Jit can now display staged file modifications. If we run `status` and `diff` before staging anything, we see our changes as unstaged:

```
$ jit status

Changes not staged for commit:

modified: lib/command/diff.rb
modified: lib/repository/status.rb
modified: test/command/diff_test.rb

no changes added to commit

$ jit diff

diff --git a/lib/command/diff.rb b/lib/command/diff.rb
index d04b00f..8edb023 100644
--- a/lib/command/diff.rb
+++ b/lib/command/diff.rb
diff --git a/lib/repository/status.rb b/lib/repository/status.rb
index c1ea864..ca6ef59 100644
--- a/lib/repository/status.rb
+++ b/lib/repository/status.rb
diff --git a/test/command/diff_test.rb b/test/command/diff_test.rb
index 3784ac3..d692609 100644
--- a/test/command/diff_test.rb
+++ b/test/command/diff_test.rb
```

If we then add the changed files, `diff` by itself prints nothing but `diff --cached` prints the same changes we just saw.

```
$ jit add lib test

$ jit status

Changes to be committed:

modified: lib/command/diff.rb
modified: lib/repository/status.rb
modified: test/command/diff_test.rb

$ jit diff

$ jit diff --cached

diff --git a/lib/command/diff.rb b/lib/command/diff.rb
index d04b00f..8edb023 100644
--- a/lib/command/diff.rb
+++ b/lib/command/diff.rb
diff --git a/lib/repository/status.rb b/lib/repository/status.rb
index c1ea864..ca6ef59 100644
--- a/lib/repository/status.rb
```

```
+++ b/lib/repository/status.rb
diff --git a/test/command/diff_test.rb b/test/command/diff_test.rb
index 3784ac3..d692609 100644
--- a/test/command/diff_test.rb
+++ b/test/command/diff_test.rb
```

Adding support for added and deleted files is just a question of combining methods we already have:

```
# lib/command/diff.rb

def diff_head_index
  @status.index_changes.each do |path, state|
    case state
    when :added  then print_diff(from_nothing(path), from_index(path))
    when :modified then print_diff(from_head(path), from_index(path))
    when :deleted then print_diff(from_head(path), from_nothing(path))
    end
  end
end
```

Printing added files requires one small change: it's now possible for `a.mode` to be `nil`, and we need to handle that in `print_diff_mode`.

```
# lib/command/diff.rb

def print_diff_mode(a, b)
  if a.mode == nil
    puts "new file mode #{b.mode}"
  elsif b.mode == nil
    puts "deleted file mode #{a.mode}"
  elsif a.mode != b.mode
    puts "old mode #{a.mode}"
    puts "new mode #{b.mode}"
  end
end
```

The remaining step that's necessary to complete the behaviour of `print_diff` is to make it display the line-by-line patch between one version and another.

12.3. Displaying edits

Using the `Diff` module that we developed in Chapter 11, *The Myers diff algorithm*, it's straightforward to generate and print the difference between two versions of a file. We just need to modify `print_diff_content` to generate and display the diff edits for the two versions after it's finished generating the header lines.

```
# lib/command/diff.rb

def print_diff_content(a, b)
  return if a.oid == b.oid

  oid_range = "index #{short a.oid}..#{short b.oid}"
  oid_range.concat(" #{a.mode}") if a.mode == b.mode

  puts oid_range
  puts "--- #{a.diff_path}"
```

```
    puts "+++\#{ b.diff_path }"

    edits = ::Diff.diff(a.data, b.data)
    edits.each { |edit| puts edit }
end
```

This addition requires the `Target` objects to have an additional field called `data` that will contain the text of the file in addition to its other metadata.

```
# lib/command/diff.rb

Target = Struct.new(:path, :oid, :mode, :data) do
  def diff_path
    mode ? path : NULL_PATH
  end
end
```

Then, we need to augment the `from_entry`, `from_file` and `from_nothing` methods to populate the `Target#data` property. `from_entry` will do this by loading a blob by ID from the database, and `from_file` will do it by using the data it's already read from the workspace. `from_nothing` will hard-code an empty string, so that added files show up as the entire file being added and deleted files show up with all their lines being deleted.

```
# lib/command/diff.rb

def from_entry(path, entry)
  blob = repo.database.load(entry.oid)
  Target.new(path, entry.oid, entry.mode.to_s(8), blob.data)
end

def from_file(path)
  blob = Database::Blob.new(repo.workspace.read_file(path))
  oid = repo.database.hash_object(blob)
  mode = Index::Entry.mode_for_stat(@status.stats[path])

  Target.new(path, oid, mode.to_s(8), blob.data)
end

def from_nothing(path)
  Target.new(path, NULL_OID, nil, "")
end
```

Having added these changes, our `diff` command will now display them for us. I have elided some of the output below with lines like `# ...`, because the edits generated by the `Diff.diff` method include all unchanged lines, not just the changed ones. When we're using diffs as a building block for other operations that will be useful, but for displaying diffs to the end user it would be nice to filter out long unchanged sections.

```
$ jit status

Changes not staged for commit:

modified: lib/command/diff.rb

no changes added to commit

$ jit diff
```

```
diff --git a/lib/command/diff.rb b/lib/command/diff.rb
index 1450e7c..2ec0952 100644
--- a/lib/command/diff.rb
+++ b/lib/command/diff.rb
require "pathname"
+
require_relative "./base"
+require_relative "../diff"

module Command
  class Diff < Base
# ...
    NULL_OID = "0" * 40
    NULL_PATH = "/dev/null"

-    Target = Struct.new(:path, :oid, :mode) do
+    Target = Struct.new(:path, :oid, :mode, :data) do
      def diff_path
        mode ? path : NULL_PATH
      end
# ...
    end

    def from_entry(path, entry)
-      Target.new(path, entry.oid, entry.mode.to_s(8))
+      blob = repo.database.load(entry.oid)
+      Target.new(path, entry.oid, entry.mode.to_s(8), blob.data)
    end

    def from_file(path)
# ...
      oid = repo.database.hash_object(blob)
      mode = Index::Entry.mode_for_stat(@status.stats[path])

-      Target.new(path, oid, mode.to_s(8))
+      Target.new(path, oid, mode.to_s(8), blob.data)
    end

    def from_nothing(path)
-      Target.new(path, NULL_OID, nil)
+      Target.new(path, NULL_OID, nil, "")
    end

    def short(oid)
# ...
      puts oid_range
      puts "--- #{a.diff_path}"
      puts "++#{b.diff_path}"
+
+      edits = ::Diff.diff(a.data, b.data)
+      edits.each { |edit| puts edit }
    end

  end
# ...
```

Our next task is to finesse this output. The output our `diff` command is currently generating is not wrong, but it could be considerably nicer. To round out this chapter we'll implement

three improvements to its functionality. First, displaying long unchanged stretches of files for the user to scroll back through is annoying, so we'll make `diff` just display the lines that have changed, plus a little surrounding context. Second, we'll make it print its output in colour to make deletions and insertions a little more recognisable. And third, we'll have the command pipe its output into a pager program, so that the user can scroll through the output and search it from the top, rather than having to scan back through their terminal to find the start of the output.

12.3.1. Splitting edits into hunks

When Git displays a diff, it does not typically show the entire file including all the unchanged lines. Instead, it splits the changes within the file into what it calls *hunks*, groups of changes that are close together plus a few lines of unchanged content on either side. Recall the first diff we looked at in this chapter:

```
diff --git a/lib/command.rb b/lib/command.rb
index 182dce2..5b09b2b 100644
--- a/lib/command.rb
+++ b/lib/command.rb
@@ -1,5 +1,6 @@
  require_relative "./command/add"
  require_relative "./command/commit"
+require_relative "./command/diff"
  require_relative "./command/init"
  require_relative "./command/status"

@@ -10,7 +11,8 @@ module Command
    "init"  => Init,
    "add"   => Add,
    "commit" => Commit,
-  "status" => Status
+  "status" => Status,
+  "diff"   => Diff
}

def self.execute(dir, env, argv, stdin, stdout, stderr)
```

The line `@@ -1,5 +1,6 @@` before the line-by-line diff tells us that this hunk begins on line 1 of version a and includes 5 lines of it, and begins on line 1 of version b and contains 6 lines of it. Likewise the line `@@ -10,7 +11,8 @@ module Command` says this hunk begins on line 10 of a and line 11 of b, and includes 7 lines of a and 8 lines of b. The text `module Command` is an extra bit of context; Git attempts to display the enclosing function or module for the changes in each hunk.

These regions are padded with up to three lines of unchanged content to provide some context for the changes within. A hunk is defined as a set of changes whose context overlaps, i.e. if the context window is three lines, then any pair of changes separated by up to six unchanged lines are part of the same hunk.

Printing the header of each hunk requires knowing the line numbers of the displayed edits in versions a and b of the file. The result of `Diff.diff` is an array of `Diff::Edit` objects that have a type and a `text` property. We could regenerate the line numbers while iterating over this array, but it will be a little easier to implement hunk filtering if we include line number information directly in the edit objects. To set us up for implementing hunks, we'll change `Diff::Edit` so that instead of its `text` field, it instead has two fields `a_line` and `b_line`, each of which points to a `Diff::Line` object which has a number and text.

We begin by changing `Diff.lines` so that it wraps each line of the source document in a `Line` object with a 1-indexed number.

```
# lib/diff.rb

Line = Struct.new(:number, :text)

def self.lines(document)
  document = document.lines if document.is_a?(String)
  document.map.with_index { |text, i| Line.new(i + 1, text) }
end
```

This requires a couple of small amendments to the `Diff::Myers` code. When detecting diagonals, instead of directly comparing string values we are now comparing `Line` objects, so we need to look at their `text` field to determine equality.

```
# lib/diff/myers.rb

while x < n and y < m and @a[x].text == @b[y].text
  x, y = x + 1, y + 1
end
```

When we're building the list of edits, we should now pass in the lines from both versions into `Diff::Edit.new`, rather than a single string for the edit's content.

```
# lib/diff/myers.rb

def diff
  diff = []

backtrack do |prev_x, prev_y, x, y|
  a_line, b_line = @a[prev_x], @b[prev_y]

  if x == prev_x
    diff.push(Edit.new(:ins, nil, b_line))
  elsif y == prev_y
    diff.push(Edit.new(:del, a_line, nil))
  else
    diff.push(Edit.new(:eql, a_line, b_line))
  end
end

diff.reverse
end
```

And the `Diff::Edit` class can then store both lines rather than just a single `text` field.

```
# lib/diff.rb

Edit = Struct.new(:type, :a_line, :b_line) do
  def to_s
    line = a_line || b_line
    SYMBOLS.fetch(type) + line.text
  end
end
```

The `Diff::Edit` objects now have enough information to implement hunks. To begin with, let's change `Command::Diff#print_diff_content` so that instead of getting a list of edits and

printing them, it instead gets a list of hunks and prints each one. Each hunk has a header and a list of edits.

```
# lib/command/diff.rb

def print_diff_content(a, b)
  return if a.oid == b.oid

  oid_range = "index #{ short a.oid }..#{ short b.oid }"
  oid_range.concat(" #{ a.mode }") if a.mode == b.mode

  puts oid_range
  puts "--- #{ a.diff_path }"
  puts "+++ #{ b.diff_path }"

  hunks = ::Diff.diff_hunks(a.data, b.data)
  hunks.each { |hunk| print_diff_hunk(hunk) }
end

def print_diff_hunk(hunk)
  puts hunk.header
  hunk.edits.each { |edit| puts edit }
end
```

We're now calling `Diff.diff_hunks` to compare the two files. This is a simple method that combines two operations: the existing `Diff.diff` method, and a new method called `Diff::Hunk.filter`, which collects the edits from `Diff.diff` into a list of `Hunk` objects.

```
# lib/diff.rb

def self.diff_hunks(a, b)
  Hunk.filter(Diff.diff(a, b))
end
```

The job of `Diff::Hunk.filter` is to take an array of `Diff::Edit` objects and group them so that long sections of unchanged content are filtered out, and changes that appear close together are part of the same hunk. We'll define a `Hunk` as a struct with fields `a_start`, `b_start` and `edits`. `a_start` and `b_start` will record the line numbers in versions `a` and `b` at which the hunk starts, and they are necessary in case the hunk exclusively contains either deletions or insertions, so there are no lines from one of the versions present. `edits` will be an array of `Diff::Edit` objects that are part of the `Hunk`.

`Hunk.filter` scans through the input list `edits` and it works in two phases. Initialising `offset` to `0`, we first scan through the `edits` until we find one whose type is not `:eq1`. The `&.` operator in `edits[offset]&.type` handles the case where we've scanned past the end of the list and `edits[offset]` is `nil`. Once we find a change, we skip back `HUNK_CONTEXT + 1` lines so we're looking at the edit just before where we want the hunk to start. In all cases this will either be an `eq1` edit, or nothing if `offset` is negative. We record `a_start` and `b_start` from the line numbers of this edit, and then start a new `Hunk` and call another method called `Hunk.build` that will collect all the edits in it and return the new `offset`.

```
# lib/diff/hunk.rb

module Diff
```

```
HUNK_CONTEXT = 3

Hunk = Struct.new(:a_start, :b_start, :edits) do
  def self.filter(edits)
    hunks = []
    offset = 0

    loop do
      offset += 1 while edits[offset]&.type == :eql
      return hunks if offset >= edits.size

      offset -= HUNK_CONTEXT + 1

      a_start = (offset < 0) ? 0 : edits[offset].a_line.number
      b_start = (offset < 0) ? 0 : edits[offset].b_line.number

      hunks.push(Hunk.new(a_start, b_start, []))
      offset = Hunk.build(hunks.last, edits, offset)
    end
  end
end

end
```

For example, say we're trying to find the first hunk in this list of edits. The first column is the offset in the edits array of each change, the second column is the edit's a_line.number and the third its b_line.number.

Figure 12.1. List of edits in a diff

0	1	1	module Command
1	2	2	Unknown = Class.new(StandardError)
2	3	3	
3	4	4	COMMANDS = {
4	5	5	"init" => Init,
5	6	6	"add" => Add,
6	7	7	"commit" => Commit,
7	8	-	"status" => Status
8	8	+	"status" => Status,
9	9	+	"diff" => Diff
10	9	10	}
11	10	11	
12	11	12	def self.execute(dir, env, argv, stdin, stdout, stderr)
13	12	13	# ...

We try to find the first change, and we find it at offset = 7.

Figure 12.2. Scanning to find the first change

```

0      1      1      module Command
1      2      2      Unknown = Class.new(StandardError)
2      3      3
3      4      4      COMMANDS = {
4      5      5          "init"    => Init,
5      6      6          "add"     => Add,
6      7      7          "commit"  => Commit,
-> 7      8          - "status" => Status
8      8      8          + "status" => Status,
9      9      9          + "diff"   => Diff
10     9      10     }
11     10     11
12     11     12      def self.execute(dir, env, argv, stdin, stdout, stderr)
13     12     13      # ...

```

Then we skip back four lines — one more than the hunk context of three — so the offset pointer is at 3. This sets us up for the second phase of the loop, which moves forward one line at a time and decides whether to include it in the current hunk.

Figure 12.3. Setting up to begin a hunk

```

0      1      1      module Command
1      2      2      Unknown = Class.new(StandardError)
2      3      3
-> 3      4      4      COMMANDS = {
4      5      5          "init"    => Init,
5      6      6          "add"     => Add,
6      7      7          "commit"  => Commit,
7      8      8          - "status" => Status
8      8      8          + "status" => Status,
9      9      9          + "diff"   => Diff
10     9      10     }
11     10     11
12     11     12      def self.execute(dir, env, argv, stdin, stdout, stderr)
13     12     13      # ...

```

Having found the start of the hunk, we then enter the `Hunk.build` method, which is displayed below. It takes the current `Hunk` object, the `edits` array, and the starting offset we found in the previous stage. Its job is to add edits to the hunk until there are no more changes that are close to the last one we collected.

It does this by starting a counter at `-1`, and then running a loop until this counter is equal to zero. In each iteration, we first add the current edit to the hunk, as long as `offset` is not negative and `counter` is positive. These checks make sure that we're not pointing at the empty space before the start of the `edits` array, and we're not on the first turn of the loop — we don't want to collect the first edit as it's the one before where we want the hunk to start.

Then, we increment `offset` and decide whether to continue the hunk. If `offset` has gone past the end of the `edits` array then we break, otherwise we look ahead to the edit a few steps away from our current position, at the far edge of the context window, and change the counter based on its type. If the edit has type `:ins` or `:del`, we reset the counter to `2 * HUNK_CONTEXT + 1`; when we encounter a change we want to display at least that edit and `HUNK_CONTEXT` lines on

either side of it. Otherwise we let the counter tick down. We continue this loop until the counter reaches zero or we run out of edits, and finally we return the new offset to the caller.

```
# lib/diff/hunk.rb

def self.build(hunk, edits, offset)
  counter = -1

  until counter == 0
    hunk.edits.push(edits[offset]) if offset >= 0 and counter > 0

    offset += 1
    break if offset >= edits.size

    case edits[offset + HUNK_CONTEXT]&.type
    when :ins, :del
      counter = 2 * HUNK_CONTEXT + 1
    else
      counter -= 1
    end
  end

  offset
end
```

In our example, on the first turn of this loop, counter is -1 and incrementing offset moves it to 4. Three lines away from that is the edit at offset 7, which is a deletion. This sets counter to `2 * HUNK_CONTEXT + 1`, which is 7.

Figure 12.4. Capturing the first context line

```
0   1   1     module Command
1   2   2       Unknown = Class.new(StandardError)
2   3   3
3   4   4     COMMANDS = {
->  4   5   5       "init"    => Init,
5   6   6       "add"     => Add,
6   7   7       "commit"   => Commit,
->  7   8           -   "status"  => Status
8   8   8       +   "status"  => Status,
9   9   9       +   "diff"    => Diff
10  9   10      }
11  10  11
12  11  12     def self.execute(dir, env, argv, stdin, stdout, stderr)
13  12  13       # ...
```

On the next turn of the loop, offset is incremented to 5 and we check the edit at offset 8. This is an insertion, so again counter is reset to 7.

Figure 12.5. Expanding the hunk on reading a change

```

0   1   1      module Command
1   2   2      Unknown = Class.new(StandardError)
2   3   3
3   4   4      COMMANDS = {
4   5   5          "init"  => Init,
-> 5   6   6          "add"   => Add,
6   7   7          "commit" => Commit,
7   8   8          - "status" => Status
-> 8   9   9          + "status" => Status,
9   9   10         + "diff"  => Diff
10  9   10        }
11  10  11
12  11  12      def self.execute(dir, env, argv, stdin, stdout, stderr)
13  12  13        # ...

```

The same thing happens at offset 6. But when offset is incremented to 7, we check the edit at offset 10 and see that it does not represent a change. Therefore counter ticks down to 6.

Figure 12.6. Detecting the end of a group of changes

```

0   1   1      module Command
1   2   2      Unknown = Class.new(StandardError)
2   3   3
3   4   4      COMMANDS = {
4   5   5          "init"  => Init,
5   6   6          "add"   => Add,
6   7   7          "commit" => Commit,
-> 7   8   8          - "status" => Status
8   8   9   9          + "status" => Status,
9   9   10         + "diff"  => Diff
-> 10  9   10        }
11  10  11
12  11  12      def self.execute(dir, env, argv, stdin, stdout, stderr)
13  12  13        # ...

```

So far, the loop will have added edits 4, 5 and 6 to the hunk. It now adds edit 7 and the counter continues to tick down to 5. As we find no further changes in the edits array, the counter will continue to tick down as we add edits 8 and 9, which represent changes, and then 10, 11 and 12 which are unchanged lines. At this point the counter will have reached 0 and the loop will exit. This completes the work of `Diff::Hunk.filter` and it returns an array of `Diff::Hunk` objects.

All that remains is to implement the `Hunk#header` method that produces the `@@` line at the beginning of each hunk. To get the starting offset for the a version, we call `edits.map(&:a_line).compact` to get all the lines from edits that have a line in the a version; to get the starting number we get the `number` property of the first of these, or the hunk's `a_start` value if there are no lines.

```

# lib/diff/hunk.rb

def header
  a_offset = offsets_for(:a_line, a_start).join(",")
  b_offset = offsets_for(:b_line, b_start).join(",")
  "@@ -#{a_offset} +#{b_offset} @@"

```

```
end

private

def offsets_for(line_type, default)
  lines = edits.map(&line_type).compact
  start = lines.first&.number || default

  [start, lines.size]
end
```

Having made these changes, the `diff` command now displays only the lines that have changed, with three lines of context and the hunk header lines.

```
$ jit status

Changes not staged for commit:

  modified: lib/command/diff.rb
  modified: lib/diff.rb

Untracked files:

  lib/diff/hunk.rb

no changes added to commit

$ jit diff

diff --git a/lib/command/diff.rb b/lib/command/diff.rb
index 2ec0952..eb1a819 100644
--- a/lib/command/diff.rb
+++ b/lib/command/diff.rb
@@ -112,9 +112,14 @@
    puts " --- #{ a.diff_path }"
    puts " +++ #{ b.diff_path }"

-    edits = ::Diff.diff(a.data, b.data)
-    edits.each { |edit| puts edit }
+    hunks = ::Diff.diff_hunks(a.data, b.data)
+    hunks.each { |hunk| print_diff_hunk(hunk) }
    end

+    def print_diff_hunk(hunk)
+      puts hunk.header
+      hunk.edits.each { |edit| puts edit }
+    end
+
    end
  end
diff --git a/lib/diff.rb b/lib/diff.rb
index e1f804d..b3acd79 100644
--- a/lib/diff.rb
+++ b/lib/diff.rb
@@ -1,3 +1,4 @@
+require_relative "./diff/hunk"
 require_relative "./diff/myers"

module Diff
```

```
@@ -24,4 +25,8 @@
  def self.diff(a, b)
    Myers.diff(Diff.lines(a), Diff.lines(b))
  end
+
+  def self.diff_hunks(a, b)
+    Hunk.filter(Diff.diff(a, b))
+  end
end
```

Git actually displays a different diff for the file `lib/command/diff.rb` here, because on top of the basic diff algorithm it tries to ‘slide’ groups of edits up and down to combine them into fewer groups, thereby making the diff smaller. In this case, the two lines before the start of the inserted `print_diff_hunk` method are the same as the two lines at the end of this group of insertions, so these edits can be slid upwards and combined with the edits to the previous method.

```
-      edits = ::Diff.diff(a.data, b.data)
-      edits.each { |edit| puts edit }
+      hunks = ::Diff.diff_hunks(a.data, b.data)
+      hunks.each { |hunk| print_diff_hunk(hunk) }
+    end
+
+    def print_diff_hunk(hunk)
+      puts hunk.header
+      hunk.edits.each { |edit| puts edit }
+    end
end
```

Although we had to modify the `Diff` module slightly, being able to uniquely identify lines by their number rather than by their contents will become useful again later, for example when we implement merging².

12.3.2. Displaying diffs in colour

When Git prints diffs in the above format, it prints the header lines in bold, the hunk boundaries in cyan, deletions in red and insertions in green. We already have some infrastructure for displaying output in colour; we just need a couple of additions to the `SGR_CODES` list in `color`.

```
# lib/command/color.rb

SGR_CODES = {
  "bold"  => 1,
  "red"   => 31,
  "green" => 32,
  "cyan"  => 36
}
```

Then, we can modify `Command::Diff` to format its output appropriately. All the lines that print header information should have the method `puts` replaced with a call to a method that wraps the string in bold formatting, for example:

```
# lib/command/diff.rb
```

²Chapter 20, *Merging inside files*

```
def header(string)
  puts fmt(:bold, string)
end

def print_diff_mode(a, b)
  if a.mode == nil
    header("new file mode #{ b.mode }")
  elsif b.mode == nil
    header("deleted file mode #{ a.mode }")
  elsif a.mode != b.mode
    header("old mode #{ a.mode }")
    header("new mode #{ b.mode }")
  end
end
```

The `print_diff_hunk` method can be similarly expanded to colour all of its lines correctly.

```
# lib/command/diff.rb

def print_diff_hunk(hunk)
  puts fmt(:cyan, hunk.header)
  hunk.edits.each { |edit| print_diff_edit(edit) }
end

def print_diff_edit(edit)
  text = edit.to_s.rstrip

  case edit.type
  when :eq1 then puts text
  when :ins then puts fmt(:green, text)
  when :del then puts fmt(:red, text)
  end
end
```

The `fmt` method already takes care of removing colour formatting unless the program is writing to a TTY, so no further changes are necessary here; the `diff` command now displays its output in full colour.

```
$ jit diff
diff --git a/lib/color.rb b/lib/color.rb
index f710d3c..a0f73f8 100644
--- a/lib/color.rb
+++ b/lib/color.rb
@@ -1,7 +1,9 @@
 module Color
   SGR_CODES = {
+     "bold" => 1,
     "red" => 31,
-     "green" => 32
+     "green" => 32,
+     "cyan" => 36
   }

   def self.format(style, string)
```

12.3.3. Invoking the pager

Our `diff` command currently prints all its output and then returns you to the shell prompt, with the last screen-full of output visible on the screen. As the output of `diff` can often be many screens worth of text, this can be annoying to interact with; we're forced to scroll back through the terminal to find the start of the output. If we're particularly unlucky, the output will have disappeared off the end of the terminal's scroll-back buffer.

Git's `diff` command improves upon this by displaying its output in a *pager*³ like `less`. A pager is a program that displays a file or other text stream in the terminal, allowing you to scroll up and down within the file. `less` itself provides many other navigational shortcuts similar to those in Vim, and lets you search for text within the file.

We can accomplish the same thing ourselves very easily, by piping the `diff` command into `less` ourselves. You can do this with any program whose output you'd like to view in this way.

```
$ jit diff | less
```

However, this loses the colour formatting that we added in the previous section. Because `jit` no longer has a TTY for its standard output stream, it stops producing colour codes. Running `git diff | less` has the same effect. It is possible to run `less` with the `-R` option that makes it correctly display colour-formatted text, but when one program is piped into another, the first program has no idea about the capabilities, or even the name, of the program it's piping into. There's no way for `git` to notice that it's being piped into `less -R` and therefore turn colour formatting back on.

³https://en.wikipedia.org/wiki/Terminal_pager

If we want to display colour-formatted output in the pager, we need to launch the pager ourselves so that we know what our program is writing output to. This also saves the user the trouble of remembering to type `| less` themselves. In the `Command::Diff#run` method, before we begin printing output, we'll put in a call to start up the pager.

```
# lib/command/diff.rb

def run
  repo.index.load
  @status = repo.status

  setup_pager

  if @args.first == "--cached"
    diff_head_index
  else
    diff_index_workspace
  end

  exit 0
end
```

This `setup_pager` method will be defined in `Command::Base`. It returns if `@pager` is already assigned, or if `@isatty` is not set. Otherwise, we instantiate a new class called `Pager`, passing in the current environment and standard output/error streams. Finally, we reassign the `@stdout` variable to the pager's `input` attribute, so that `puts` writes to the pager instead of directly to the terminal.

```
# lib/command/base.rb

def setup_pager
  return if defined? @pager
  return unless @isatty

  @pager = Pager.new(@env, @stdout, @stderr)
  @stdout = @pager.input
end
```

The `@isatty` variable is set in `Command::Base#initialize`. Since we're reassigning `@stdout` to something that's not a TTY, but we still want to emit colour codes, we store off whether the original standard output is a TTY in the constructor, and then use that value later when deciding whether to format printed strings.

```
# lib/command/base.rb

def initialize(dir, env, args, stdin, stdout, stderr)
  #
  # ...

  @isatty = @stdout.isatty
end

def fmt(style, string)
  @isatty ? Color.format(style, string) : string
end
```

Now let's take a look at the `Pager` class itself. Its job is to launch a pager program as a *child process*⁴, that runs in parallel with the current Jit process, which is a Ruby process that's running our program. It will inherit the standard output/error streams from the Jit process, so that it appears in the terminal where Jit is running. It exposes an `input` attribute for our program to write to, and everything written there will be displayed by the pager.

Let's take a look at the class and then walk through how it works.

```
# lib/pager.rb

class Pager
  PAGER_CMD = "less"
  PAGER_ENV = { "LESS" => "FRX", "LV" => "-c" }

  attr_reader :input

  def initialize(env = {}, stdout = $stdout, stderr = $stderr)
    env = PAGER_ENV.merge(env)
    cmd = env["GIT_PAGER"] || env["PAGER"] || PAGER_CMD

    reader, writer = IO.pipe
    options = { :in => reader, :out => stdout, :err => stderr }

    @pid = Process.spawn(env, cmd, options)
    @input = writer

    reader.close
  end
end
```

The class accepts an environment and two output streams, which when the program is running for real will correspond to `ENV`, `$stdout` and `$stderr`. It creates an extended environment by merging `PAGER_ENV` into the current environment; the order of the merge means that if the current environment defines the same variables as `PAGER_ENV`, the values in the current environment take precedence. Next, we select the name of the pager program to run: we use the user's `GIT_PAGER` or `PAGER` environment variable if either of these is set, and fall back to a default value stored in `PAGER_CMD`.

The variables in `PAGER_ENV` exist to control the behaviour of the pager program. We don't know the identity of the pager we'll end up running, and we don't know what command-line arguments we're allowed to pass to it, but some pagers can be configured via the environment. Any variables the pager doesn't understand will just be ignored, whereas programs typically crash if given command-line arguments they don't understand. `less` uses the `LESS` environment variable; the switch `F` means it will exit without entering scrolling mode if the output is less than a single screen, `R` means that formatting codes will be passed straight through to the output stream without being escaped, which preserves colour formatting, and `x` stops `less` sending some special instructions to the terminal such as those to clear the screen before starting. The pager `lv`⁵ uses the `LV` variable and has an option called `-c` for enabling colour display. These environment variables are the exact values that Git sets when it runs a pager.

⁴https://en.wikipedia.org/wiki/Child_process

⁵<https://manpages.ubuntu.com/manpages/bionic/en/man1/lv.1.html>

The next set of instructions actually execute the child process. First we call `IO.pipe`⁶, a wrapper around the `pipe()` system call⁷, to create a channel for our process to send data to the child process. It returns two objects, a *reader* and a *writer*, where anything written to the writer can be read from the reader. Then we set up options, which sets the standard I/O streams for the new process; it will inherit `stdout` and `stderr` from our process, but its `stdin` will be the reader created by `pipe()`, which will let it receive any data we want to pass into it. Finally, having built the environment and options and selected a program to run, we call `Process.spawn`⁸ to start the pager process. It runs asynchronously, so `Process.spawn` doesn't block until the pager program exits — the pager runs in tandem with our program. `Process.spawn` returns the *process ID*⁹ or *PID* of the pager process.

We assign the writer end of the pipe to `@input`, so now anything can write to the pager's `input` attribute and the data will show up in the pager. We close the reader because our main process has no further use for it; it will remain open inside the pager process.

The fact that commands can now write to a pipe means we also need to add an error handler to the `Command::Base#puts` method. It's possible for the user to exit the pager program while our Jit command is still running, in which case the call to `@stdout.puts` will start raising an `Errno::EPIPE` exception as the pipe has been closed. In this event, there's no point in continuing to run the command as its output will never be seen by the user, so we exit. This assumes that the command is not going to make any further changes to the files on disk, and so exiting early is perfectly safe. Indeed, most Git commands that produce long paged output do not make any changes to the database, index or workspace.

```
# lib/command/base.rb

def puts(string)
  @stdout.puts(string)
rescue Errno::EPIPE
  exit 0
end
```

That takes care of redirecting the command's output into the pager, but we need one more change to `Command::Base` to make everything work. The `Pager` class starts the pager program as a child process, in parallel with the current Jit process. The two processes continue to execute concurrently, Jit feeding input into the pager via the pipe. When Jit's main process has finished everything it's doing — in this case, generating all the diff output — it will exit, and that poses a problem for the pager.

When a process exits, any child processes it has spawned will keep running; these are referred to as *orphan processes*¹⁰. This in itself isn't a problem in this case: we want the pager to keep running until the user explicitly closes it. However, it needs to keep receiving input from the user on standard input, and it needs to be able to update what's displayed in the terminal through its standard output. But, once Jit exits, control returns to the shell: it displays your prompt again, and anything you type becomes text for the next command to run, rather than being fed into the pager. The shell has reclaimed the standard I/O streams, cutting the pager off from interacting

⁶<https://docs.ruby-lang.org/en/2.3.0/IO.html#method-c-pipe>

⁷<https://manpages.ubuntu.com/manpages/bionic/en/man2/pipe.2.html>

⁸<https://docs.ruby-lang.org/en/2.3.0/Process.html#method-c-spawn>

⁹https://en.wikipedia.org/wiki/Process_identifier

¹⁰https://en.wikipedia.org/wiki/Orphan_process

with the user. Some pagers will notice this and exit of their own accord, but some will keep running in the background where you cannot meaningfully interact with them.

Fortunately, solving this problem is relatively easy. Until Jit exits, it and the pager are running in parallel. To prevent control returning to the shell, we can block Jit from exiting until the pager has also been closed by the user. All we need to do is wait for the pager process to exit. The method `Process.waitpid`¹¹ takes a PID and blocks until the corresponding process has finished. We'll wrap this in a method on our `Pager` class so the caller can wait for the pager to exit.

```
# lib/pager.rb

def wait
  Process.waitpid(@pid) if @pid
  @pid = nil
end
```

Now it's just a case of inserting a hook into `Command::Base` so that when the current command calls `exit`, it waits for the pager to exit if there's a pager active. Inside the `execute` method, we continue to call `run` until it throws `:exit`, and then we wait for the pager to exit.

```
# lib/command/base.rb

def execute
  catch(:exit) { run }

  if defined? @pager
    @stdout.close_write
    @pager.wait
  end
end
```

We also use this moment to tell the pager there is no more input to display by closing `@stdout` — this lets `less` exit without scrolling if its input does not fill a full screen, and it enables the `G` keyboard shortcut for scrolling to the end of the output.

That's all we need to allow our program to run its output through a pager. Any further commands we build can reuse the `setup_pager` method to opt in to this functionality.

¹¹<https://docs.ruby-lang.org/en/2.3.0/Process.html#method-c-waitpid>

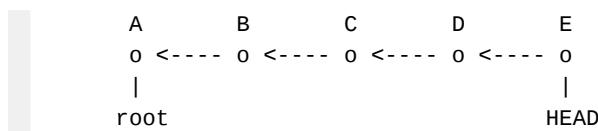
Part II. Branching and merging

13. Branching out

We now have basic versions of the commands we tend to use most for composing each commit: `status` and `diff` to see what we've changed, and `add` and `commit` for updating the index and writing commits. That's enough to support a single author tracking their changes to a project, but to enable collaborative workflows, we need to be able to create branches — threads of history that diverge before being merged back together.

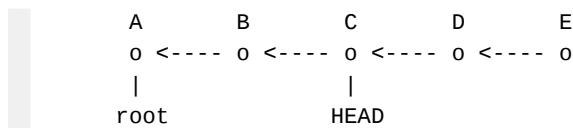
So far, we've been building a linear sequence of commits by storing the latest commit ID in `.git/HEAD`. Each commit contains a pointer to its parent, forming a linked list from `HEAD` all the way back to the root commit.

Figure 13.1. Commits forming a linked list



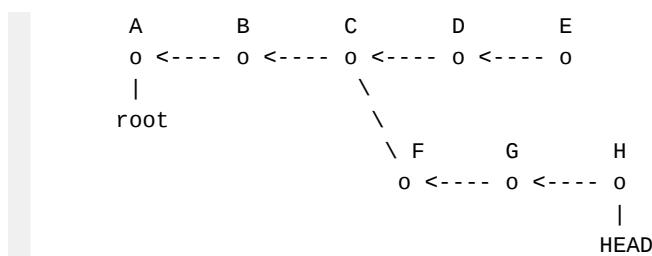
Branches let us build several diverging histories from the same starting point, without losing track of where the end of each history is. Right now, we could quite freely replace the contents of `.git/HEAD` so that it contains the ID of the latest commit's grandparent:

Figure 13.2. Moving HEAD to an older commit



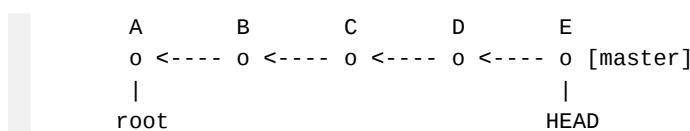
And then we could start building commits from there to produce a parallel history:

Figure 13.3. Commits diverging from the original history



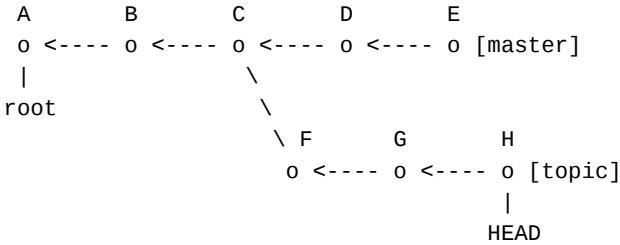
However, the reference to commit E has now been lost, so unless we know its object ID we can't retrieve it any more. Branches provide a way to store named references to multiple commits, rather than only having a single `HEAD` with which to build histories. If our original branch was called, say, `master`:

Figure 13.4. master branch referring to the original history



Then we can develop on a parallel branch and still use the name `master` to refer to the original line of development.

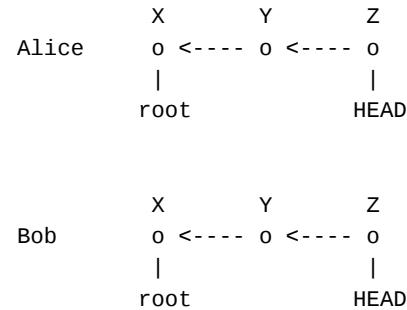
Figure 13.5. Named branches with diverging histories



Retaining named references to commits allows us to switch between different lines of development, and merge divergent histories back together. But why is this even useful? Why can't we make do with the single `HEAD` that's been serving us well so far?

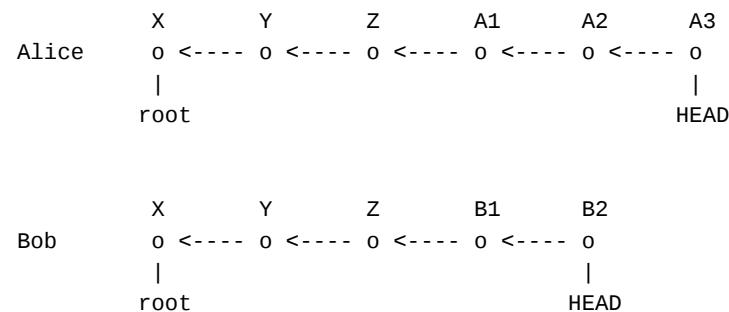
The answer is in Git's distributed nature. It's designed so that multiple people can create chains of commits locally and then reconcile their changes into a unified history. Let's say Alice and Bob both have a copy of a repository on their personal laptops, and that repository contains three commits:

Figure 13.6. Two copies of the same history



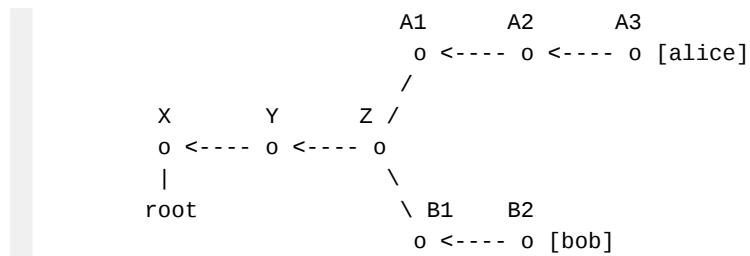
They both make commits on their local copy, and these commits will both begin with `Z` as their parent, but they'll have different content and therefore different IDs.

Figure 13.7. Chains of commits built on a common history



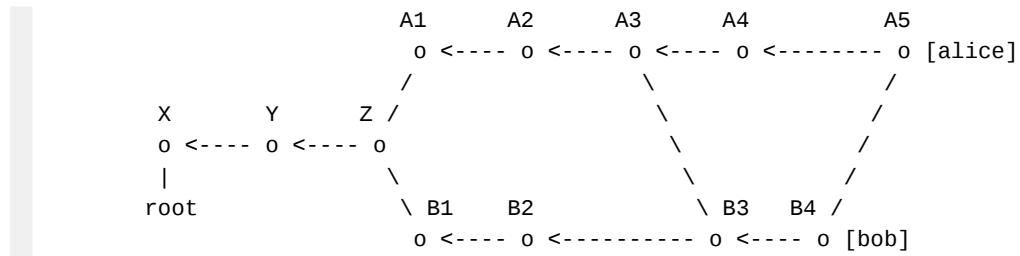
These local histories can be looked at as branches of a single shared history; Alice and Bob have `X`, `Y` and `Z` in common but then their histories diverge.

Figure 13.8. Branches from a shared history



To collaborate on the codebase, Alice and Bob need to be able to send their changes to each other and merge them together periodically, so that their changes are integrated into a single resulting tree.

Figure 13.9. Alice and Bob merging each other's changes



The above graph represents the fact that after committing B_2 , Bob fetches Alice's history ending at A_3 , and merges the histories together to produce B_3 . Then Alice commits A_4 , and Bob creates B_4 on top of his history including Alice's changes up to A_3 . Finally, Alice fetches B_4 and merges it with her commit A_4 to produce A_5 . Alice now has every change that's been made to the code, while Bob has everything except Alice's commit A_4 .

The workings of this merging process will be the subject of later chapters. What matters for now is that branching and merging are fundamental to Git's ability to operate as a distributed system. Without these features, some centralised co-ordination would be needed to get a team of contributors to produce a single shared history. Some systems assign each commit a sequential numeric ID, and you can only create a new commit if you have pulled down the latest one stored on the central server. Others require contributors to lock files they plan to work on in advance, and release the locks when they're done so someone else can modify them.

The thing that sets Git apart from centralised systems is that its method of co-ordination does not require contributors to always communicate with a central server. Each team member creates their own history locally on their computer, and Git provides tools for merging histories together when each team member chooses to do so.

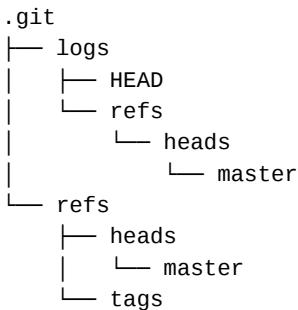
In order to support the creation of multiple divergent lines of history, we need two new commands: `branch`, which creates and deletes branch pointers, and `checkout`, which lets us switch between branches. We'll deal with the `branch` command in this chapter and tackle `checkout` in the next.

13.1. Examining the `branch` command

We'll be building a simple version of the `branch` command that will let us create, list and delete branches. To begin with, we should observe what creating a branch actually means by inspecting the contents of the `.git` directory.

Most Git repositories start out with a single branch called `master`. After making some commits on this branch, we observe that the `.git/logs` and `.git/refs` directories look as follows:

Figure 13.10. .git/logs and .git/refs for a single master branch



The `.git/HEAD` file, rather than containing a literal commit ID, contains a reference to the file `refs/heads/master`. It's this file that actually contains a commit ID.

```
$ cat .git/HEAD  
ref: refs/heads/master  
  
$ cat .git/refs/heads/master  
eca8c1dfbc55147d42bddb3dee8aa1dc458005c7
```

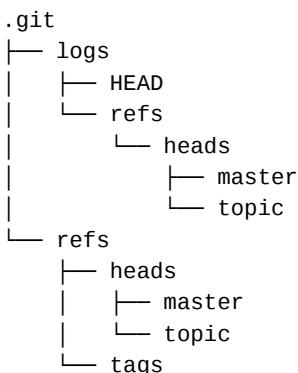
`.git/refs/heads/master` is a branch reference; it contains the ID of the last commit on the `master` branch. It works just like we've been using `.git/HEAD`, but now `.git/HEAD` refers to another file rather than acting as a commit pointer itself. When we create a new commit, the current contents of `refs/heads/master` becomes the commit's parent ID, and the new commit's ID replaces the contents of `refs/heads/master`.

Now let's create a new branch using the `branch` command:

```
$ git branch topic
```

Looking at the contents of `.git` again, we see a new entry in `.git/logs/refs/heads` and in `.git/refs/heads`.

Figure 13.11. .git/logs and .git/refs after creating the topic branch



The files `HEAD` and `refs/heads/master` are still the same as they were before:

```
$ cat .git/HEAD  
ref: refs/heads/master  
  
$ cat .git/refs/heads/master  
eca8c1dfbc55147d42bddb3dee8aa1dc458005c7
```

HEAD still points at the master branch and the latest commit on master has not changed. If we look at the new topic ref, we see it contains the same data as master:

```
$ cat .git/refs/heads/topic  
eca8c1dfbc55147d42bddb3dee8aa1dc458005c7
```

So, running `git branch topic` creates a new reference at `.git/refs/heads/topic` whose value is the ID of the commit that HEAD currently points at. Essentially, it creates a new named reference to wherever we currently are in the history. Note that it is not a *symbolic* reference to the current branch like HEAD is. Instead, it's a *copy* of the current position of HEAD and this new branch pointer can be moved independently of other branches.

Since it takes user input for the name of the branch, it's also worth noting how this command can fail. If the given branch already exists, or its name is not a valid branch name, we get an error.

```
$ git branch topic  
fatal: A branch named 'topic' already exists.  
$ echo $?  
128  
  
$ git branch ^  
fatal: '^' is not a valid branch name.  
$ echo $?  
128
```

branch can be run with a second argument that specifies where to start the branch. For example, `git branch topic @^` starts the topic branch, not at the position of HEAD, but pointing at the parent of HEAD. @ is an alias for HEAD and ^ denotes the parent of whatever precedes it. That's why ^ cannot be used as character in branch names.

Running branch without a name or starting point makes it list the existing branches, with the one currently referred to by HEAD printed with a * next to it.

```
$ git branch  
* master  
second  
topic
```

Adding the `--verbose` option (or simply `-v`) makes it print a little more information: the abbreviated commit ID the branch currently points at, and the first line of that commit's message.

```
$ git branch --verbose  
* master eca8c1d third commit  
second f77b04b second commit  
topic eca8c1d third commit
```

Finally, the `-D` option (short for `--delete --force`) makes the branch command delete any branches named in the command, printing the ID they were pointing at. Trying to delete a branch that does not exist is an error.

```
$ git branch -D topic  
Deleted branch topic (was eca8c1d).  
  
$ git branch -D topic  
error: branch 'topic' not found.  
$ echo $?
```

1

The functionality of the `branch` command itself is not especially complicated, and mostly consists of creating, listing and deleting files in the `.git/refs/heads` directory. However, in implementing this command we'll encounter Git's *revisions* notation, and write an interpreter for a subset of it.

13.2. Creating a branch

As we saw above, the basic form of the `branch` command takes a branch name and creates it as a file in `.git/refs/heads`, containing the ID of the latest commit that `HEAD` points at. Since `branch` is concerned with managing files in `.git/refs`, we'll delegate most of its functionality to the `Refs` class. We'll meet other commands that use `.git/refs` later, and `Refs` will provide a common interface for all of them to use.

To begin with, let's create a very simple version of the `branch` command that just supports calling `branch` with a single argument, the name of the new branch, and handling errors that result from trying to create it.

```
# lib/command/branch.rb

module Command
  class Branch < Base

    def run
      create_branch

      exit 0
    end

    private

    def create_branch
      branch_name = @args[0]
      repo.refs.create_branch(branch_name)
      rescue Refs::InvalidBranch => error
        $stderr.puts "fatal: #{error.message}"
        exit 128
    end
  end
end
```

Until now, the `Refs` class has been very small and simple and just deals with reading and writing `.git/HEAD`. It uses our `Lockfile` class to prevent multiple processes from concurrently changing the same reference file. Now, we need to expand this class's responsibilities to deal with managing references other than `HEAD`, in particular, those stored in `.git/refs/heads`. Let's start by implementing the `Refs#create_branch` method to see what new functionality it requires. This method takes a branch name, checks it is syntactically valid and that the branch does not already exist on disk, and then creates it with the object ID currently stored in `.git/HEAD`.

```
# lib/refs.rb

InvalidBranch = Class.new(StandardError)
```

```
INVALID_NAME = /  
  ^\.  
  | \/\.  
  | \.\.  
  | ^/  
  | \/$  
  | \.lock$  
  | @\{  
  | [\x00-\x20*:?\[\\^~\x7f]  
/x  
  
def create_branch(branch_name)  
  path = @heads_path.join(branch_name)  
  
  if INVALID_NAME =~ branch_name  
    raise InvalidBranch, "'#{ branch_name }' is not a valid branch name."  
  end  
  
  if File.file?(path)  
    raise InvalidBranch, "A branch named '#{ branch_name }' already exists."  
  end  
  
  update_ref_file(path, read_head)  
end
```

The `INVALID_NAME` regular expression defines a set of strings that cannot be included in a branch name. The Git source file `refs.c`¹ lists the following conditions for a *refname*² being valid:

```
* Try to read one refname component from the front of refname.  
* Return the length of the component found, or -1 if the component is  
* not legal. It is legal if it is something reasonable to have under  
* ".git/refs/"; We do not like it if:  
*  
* - any path component of it begins with ".", or  
* - it has double dots "..", or  
* - it has ASCII control characters, or  
* - it has ":", "?", "[", "\", "^", "~", SP, or TAB anywhere, or  
* - it has "*" anywhere unless REFNAME_REFSPEC_PATTERN is set, or  
* - it ends with a "/", or  
* - it ends with ".lock", or  
* - it contains a "@{" portion
```

Our regex lists a set of patterns separated by `|` so that if the input string matches any of these, it's considered invalid:

- `^\., \/\.` — if the string contains a *path component* beginning with `.`, that means it either begins with `.` or it contains `/..`. This prevents branches being given names that Unix systems would confuse with hidden files.
- `\.\.` — strings containing `..` anywhere are invalid, because it would be confused with Git's `..` operator for specifying commit ranges³, and could be confused for the `..` path traversal operator.

¹<https://github.com/git/git/blob/v2.15.0/refs.c#L58-L70>

²A *refname* is any name that can be used to refer to an object: a full or abbreviated object ID, a branch name, a remote name, or `HEAD`, `ORIG_HEAD` and friends.

³Section 16.2.3, “Excluding branches”

- `^\/` — if the string begins with a slash then it may be mistaken for an absolute path and cause `create_branch` and other `Refs` methods to write to files outside the repository.
- `\$/` — if the string ends with a slash that's usually a sign of user error since a trailing slash denotes a directory name.
- `\.lock$` — naming a branch something like `topic.lock` would make it impossible to move a branch called `topic`, because we would not be able to create a lockfile⁴ for it.
- `@\{` — Git supports expressions like `branch@\{upstream\}` in its revisions notation, so allowing branches to contain this string would make such expressions ambiguous.
- `[\x00-\x20*:?\[\^\~\x7f]` — this takes care of all disallowed single characters; `\x00-\x20` covers ASCII control characters, tabs and spaces, and `\x7f` handles the ASCII *DEL*⁵ character.

A couple of these rules are critical for security, particularly the `..` rule and the leading slash rule. We use `Pathname#join` to build a file path from the refname, and refnames containing these characters result in paths that point to locations outside the `.git/refs` directory:

```
>> Pathname.new(".git/refs/heads").join("topic")
=> Pathname(".git/refs/heads/topic")

>> Pathname.new(".git/refs/heads").join("../.../.../topic")
=> Pathname("../topic")

>> Pathname.new(".git/refs/heads").join("/topic")
=> Pathname("/topic")
```

It's important that our program is not tricked into overwriting files outside those it's responsible for, especially when a remote client is pushing changes⁶. These path handling issues are a very common cause of vulnerabilities in web applications, allowing clients to read and write files the server operator never intended to expose⁷.

The `create_branch` method relies on a new instance variable called `@heads_path`, which we define in the constructor along with other paths relevant to this class's responsibilities.

```
# lib/refs.rb

def initialize(pathname)
  @pathname = pathname
  @refs_path = @pathname.join("refs")
  @heads_path = @refs_path.join("heads")
end
```

And, it relies on a generalisation of the `update_head` method, which takes a pathname as a parameter rather than hard-coding `head_path` as the file to update.

```
# lib/refs.rb
```

⁴Section 4.2.1, “Safely updating `.git/HEAD`”

⁵https://en.wikipedia.org/wiki/Delete_character

⁶Section 29.2.3, “Validating update requests”

⁷https://en.wikipedia.org/wiki/Directory_traversal_attack

```
def update_ref_file(path, oid)
  lockfile = Lockfile.new(path)

  lockfile.hold_for_update
  lockfile.write(oid)
  lockfile.write("\n")
  lockfile.commit

rescue Lockfile::MissingParent
  FileUtils.mkdir_p(path.dirname)
  retry
end
```

In fact, this method allows us to replace `update_head` with an implementation that delegates to this more general function.

```
# lib/refs.rb

HEAD = "HEAD"

def update_head(oid)
  update_ref_file(@pathname.join(HEAD), oid)
end
```

This implementation is sufficient to let us create new branches pointing at the same commit as `HEAD`, as we can verify by running it and comparing the files on disk before and after:

```
$ tree .git/refs
.git/refs

$ jit branch master

$ tree .git/refs
.git/refs
└── heads
    └── master

$ cat .git/HEAD
1bcf703685b6f70e273b6668cc50dd3ad9e0756a

$ cat .git/refs/heads/master
1bcf703685b6f70e273b6668cc50dd3ad9e0756a
```

13.3. Setting the start point

When `branch` is run with two arguments, for example `git branch topic @~3`, the second argument specifies where to point the branch at instead of using the commit referenced by `HEAD`. It does this using a special shorthand language that Git uses to refer to commits and other database objects. It calls expressions in this language *revisions* and you can read about them by running `git help revisions`⁸.

For example, the revision `@~3` means ‘the third ancestor of the commit that `HEAD` refers to’; `@` is an alias for `HEAD`, and the rest of the expression denotes ‘third ancestor’, or the parent of the parent of the parent of the preceding expression.

⁸Or, read the same documentation online at <https://git-scm.com/docs/gitrevisions>

13.3.1. Parsing revisions

Git's revision notation is extensive and includes many operators for traversing the object database, but to begin with we will only implement a small subset of it that's commonly used for addressing commits. In our subset, a revision `<rev>` can be either:

- a `<refname>`, i.e. the name of a branch, a remote, an object ID, or `HEAD`
- `<rev>^`, i.e. a revision followed by `^`
- `<rev>~<n>`, i.e. a revision followed by `~` and a number

Notice that this definition of `<rev>` is recursive. Strictly speaking, the syntax of revisions is a regular language⁹ and can be matched using a regex. However, for the purposes of evaluating these expressions it will be more convenient to convert them into a recursive structure. For example, `@^` would become `(parent (ref "HEAD"))`, `HEAD~42` would become `(ancestor (ref "HEAD") 42)`, `master^^` would be `(parent (parent (ref "master")))` and finally `abc123~3^` would be parsed into the expression `(parent (ancestor (ref "abc123") 3))`.

At this point we're only concerned with the syntax of these expressions, rather than evaluating them to turn them into commit IDs. We want to analyse the revision string to determine whether it's valid, and if so what its internal structure is. We also don't care what kind of reference a `<refname>` is; it could be a branch name, an object ID, or `HEAD`, but the differences between these references only become relevant when we try to evaluate the expression.

The following `Revision` class defines a function called `Revision.parse` that takes a revision string; we'll examine how it works below.

```
# lib/revision.rb

class Revision
  Ref      = Struct.new(:name)
  Parent   = Struct.new(:rev)
  Ancestor = Struct.new(:rev, :n)

  INVALID_NAME = /
    ^\.
    | \/\.
    | \.\.
    | ^\/
    | \/$
    | \.lock$"
    | @\{
    | [\x00-\x20*:?\[\\"~\x7f]
  /x

  PARENT   = /^(.+)\^$/
  ANCESTOR = /^(.+)-(\d+)$/

  REF_ALIASES = {
    "@" => "HEAD"
  }
```

⁹https://en.wikipedia.org/wiki/Regular_language

```
def self.parse(revision)
  if match = PARENT.match(revision)
    rev = Revision.parse(match[1])
    rev ? Parent.new(rev) : nil
  elsif match = ANCESTOR.match(revision)
    rev = Revision.parse(match[1])
    rev ? Ancestor.new(rev, match[2].to_i) : nil
  elsif Revision.valid_ref?(revision)
    name = REF_ALIASES[revision] || revision
    Ref.new(name)
  end
end

def self.valid_ref?(revision)
  INVALID_NAME =~ revision ? false : true
end
```

This short `Revision.parse` function is a recursive procedure for matching valid revision expressions and determining their structure. It works by checking the input string for various conditions and breaking it down into smaller pieces.

For example, take the input "master^". We begin by calling

```
match = PARENT.match("master^")
```

where `PARENT` refers to the regex `/^(.+)\^$/`, which matches any string that ends with `^` and captures everything up to that final character. This produces a match, with `match[1] == "master"`. We call `Revision.parse("master")` to process the rest of the string, and if that returns a value then build a new `Parent` node containing the result. The function has stripped the trailing `^` off the string and wrapped the preceding expression with a structure that represents the `parent` operator.

The second branch works in the same way, with the only modification being that the `ANCESTOR` pattern contains a second capture group (`\d+`) that matches the number following the `~`. The preceding expression is parsed recursively and is wrapped in an `Ancestor` node containing the result and the matched number.

The final clause checks that whatever is left if none of the trailing operators match is a valid refname, returning a `Ref` node if so. The `REF_ALIASES` lookup takes care of converting the `@` expression into `HEAD`. If none of the conditions apply to the input string, then `Revision.parse` returns `nil`.

Let's try our parser out on a few of the example expressions we looked at above:

```
>> Revision.parse "@^"
=> Revision::Parent(rev=Revision::Ref(name="HEAD"))

>> Revision.parse "HEAD~42"
=> Revision::Ancestor(rev=Revision::Ref(name="HEAD"), n=42)

>> Revision.parse "master^^"
=> Revision::Parent(
  rev=Revision::Parent(
```

```
rev=Revision::Ref(name="master"))

>> Revision.parse "abc123~3^"
=> Revision::Parent(
  rev=Revision::Ancestor(
    rev=Revision::Ref(name="abc123"),
    n=3))
```

We see that the parse correctly produces nested structures representing the meaning of the expression. These structures are called *abstract syntax trees*¹⁰ or ASTs, and they provide a structured representation of a query, program or data format. Most software that interprets commands from blobs of text — compilers, databases, API servers, and so on — begins by converting its input into an AST for further analysis.

13.3.2. Interpreting the AST

The next step is to *interpret* the AST against the repository, to turn it into an object ID. To accomplish this, we'll give each of the node types Ref, Parent and Ancestor a method called `resolve`, which will take a context object that provides methods for querying the repository. The aim of each `resolve` implementation is to return an object ID, or `nil` if it cannot be resolved to an ID. Let's begin with `Ref`:

```
# lib/revision.rb

Ref = Struct.new(:name) do
  def resolve(context)
    context.read_ref(name)
  end
end
```

The purpose of a `Ref` node is to try and convert a refname like `HEAD` or `master` into a commit ID, and we'll achieve this by calling `read_ref(name)` on the context object. We'll look at the context methods after we've defined all the node `resolve` methods.

The next node type we have is `Parent`. This is slightly more complex than `Ref` because `Parent` nodes contain another revision node inside themselves — see the parse tree for `@^` above. The `Parent#resolve` method calls `resolve` on its inner node, and passes the result of that to another context helper method called `commit_parent`, which will load the given object ID to get a commit and then return that commit's parent ID.

```
# lib/revision.rb

Parent = Struct.new(:rev) do
  def resolve(context)
    context.commit_parent(rev.resolve(context))
  end
end
```

`Ancestor` is just a variation on this theme that calls `commit_parent` a given number of times, like nesting a number of `Parent` nodes together.

```
# lib/revision.rb
```

¹⁰https://en.wikipedia.org/wiki/Abstract_syntax_tree

```
Ancestor = Struct.new(:rev, :n) do
  def resolve(context)
    oid = rev.resolve(context)
    n.times { oid = context.commit_parent(oid) }
    oid
  end
end
```

Now, we must define this context object that's passed around the parse tree. These objects will be instances of the `Revision` class itself, and that's where we'll define the helper methods that the nodes make use of. First, the `commit_parent` method, which loads an object ID to get a commit, and then returns that commit's parent.

```
# lib/revision.rb

def commit_parent(oid)
  return nil unless oid

  commit = @repo.database.load(oid)
  commit.parent
end
```

This method requires us to expose the `parent` property on the `Database::Commit` class, which we've not needed until now.

```
# lib/database/commit.rb

attr_reader :parent, :tree
```

Note that, although `Database#load` can in general return trees and blobs, we're assuming it returns a commit here. For now, this is a reasonable assumption since the only possible results of revision expressions are the contents of `HEAD` and other references that only point to commits, and values of a commit's `parent` field which always names another commit.

The other context method we need is `Revision#read_ref`, which delegates to the `Refs` object attached to the repository.

```
# lib/revision.rb

def read_ref(name)
  @repo.refs.read_ref(name)
end
```

`Refs#read_ref` needs to take a refname like `HEAD` or `master` and return its value. To do this, it attempts to find the named reference in the `.git`, `.git/refs` and `.git/refs/heads` directories; if a matching file is found then its contents are returned, otherwise it returns `nil`.

```
# lib/refs.rb

def read_ref(name)
  path = path_for_name(name)
  path ? read_ref_file(path) : nil
end

def path_for_name(name)
  prefixes = [@pathname, @refs_path, @heads_path]
  prefix = prefixes.find { |path| File.file? path.join(name) }
```

```
prefix ? prefix.join(name) : nil
end

def read_ref_file(path)
  File.read(path).strip
rescue Errno::ENOENT
  nil
end
```

We now have a parser for revision expressions, a means of recursively interpreting them given some context, and enough support methods to make the function of each node type work. Now we need to glue these elements together to make something that can execute revision queries against a repository. Let's add a constructor to the `Revision` class that takes a `Repository` and a string containing an expression. It stores off these inputs as the `@repo` and `@expr` variables, and parses the expression.

Let's also add a method for executing the query: the `resolve` method calls `@query.resolve(self)` if `@query` is not `nil`. This passes the `Revision` instance as the context argument into the root node of the AST to begin the interpretation. If this does not result in an object ID being returned, then we raise an error.

```
# lib/revision.rb

InvalidObject = Class.new(StandardError)

def initialize(repo, expression)
  @repo  = repo
  @expr  = expression
  @query = Revision.parse(@expr)
end

def resolve
  oid = @query&.resolve(self)
  return oid if oid

  raise InvalidObject, "Not a valid object name: '#{ @expr }'."
end
```

The `Command::Branch` class can now use the `Revision` class to resolve its second argument to a commit ID, if one was given:

```
# lib/command/branch.rb

def create_branch
  branch_name = @args[0]
  start_point = @args[1]

  if start_point
    revision = Revision.new(repo, start_point)
    start_oid = revision.resolve
  else
    start_oid = repo.refs.read_head
  end

  repo.refs.create_branch(branch_name, start_oid)
```

```
rescue Refs::InvalidBranch, Revision::InvalidObject => error
  @stderr.puts "fatal: #{error.message}"
  exit 128
end
```

The Refs#create_branch method just needs to be amended to take the branch starting point as an argument, rather than using the value of HEAD, and now we can use a revision expression on the command-line to set where we want the new branch to point at.

```
# lib/refs.rb

def create_branch(branch_name, start_oid)
  path = @heads_path.join(branch_name)
  #
  update_ref_file(path, start_oid)
end
```

The Revision class we've just built is a little *interpreter*¹¹, in fact it follows the *interpreter design pattern*¹². This method of recursive evaluation is a popular technique for building simple languages; although most production interpreters don't directly evaluate program ASTs like this, the technique is still used to perform static analysis and compile the program to another form. If you've not built a programming language before and would like to try, this method is a good place to start.

In Git, the parsing and evaluation of these expressions is actually combined into a single process. However, for most applications of this technique, it's advisable to fully parse an expression before evaluating it, to make sure it's syntactically valid. This is especially true if the expressions in your language can cause side effects.

13.3.3. Revisions and object IDs

The Revision class can currently resolve references that use the name HEAD or the name of a branch to identify a commit. Git also allows revisions to contain abbreviated object IDs, for example the expression 6140951^{^^} could mean the grandparent of commit 6140951cc2a0f62e968e251f0e4ec42858b523dd.

To make this work we need to expand our implementation of Revision#read_ref, which is called by Revision::Ref#resolve. Currently this asks Refs if it can find any references by the given name, and returns nothing if not. Now, let's add a fallback that tries to match the given name against IDs in the database, and if there's a unique match then the full matching ID is returned.

```
# lib/revision.rb

def read_ref(name)
  oid = @repo.refs.read_ref(name)
  return oid if oid

  candidates = @repo.database.prefix_match(name)
  return candidates.first if candidates.size == 1
```

¹¹[https://en.wikipedia.org/wiki/Interpreter_\(computing\)](https://en.wikipedia.org/wiki/Interpreter_(computing))

¹²https://en.wikipedia.org/wiki/Interpreter_pattern

```
    nil  
end
```

This relies on a new method, `Database#prefix_match`, which attempts to find IDs that begin with the given string. Recall that Git stores objects in `.git/objects` with the first two characters of the ID forming a subdirectory and the rest of the characters forming the filename. We can use this fact to narrow the search: we list out all the files in the directory for the given string, and reconstruct the full ID for each of these filenames. Then we iterate to filter the list down to only those IDs that begin with the given name. If the directory does not exist that simply means there are no objects matching the given name, so we return an empty array.

```
# lib/database.rb  
  
def prefix_match(name)  
  dirname = object_path(name).dirname  
  
  oids = Dir.entries(dirname).map do |filename|  
    "#{dirname.basename}#{filename}"  
  end  
  
  oids.select { |oid| oid.start_with?(name) }  
rescue Errno::ENOENT  
  []  
end
```

This works fine when there's only one object matching the given ID prefix. If you use a long enough name, then most of the time you'll get a unique object. But sometimes, multiple objects will have IDs matching the given name, and that makes the revision ambiguous. When this happens, Git will print a message like this:

```
$ git branch topic 5ccf  
  
error: short object ID 5ccf is ambiguous  
hint: The candidates are:  
hint: 5ccfb75 commit 2005-08-08 - Update rev-parse flags list.  
hint: 5ccf27b tree  
hint: 5ccf0ba blob  
fatal: Not a valid object name: '5ccf'.
```

As well as the usual `Not a valid object` message that we've implementing using exceptions, here we see another error logged before the program exits, followed by some *hint text* that tells us more about the problem. We can see the short IDs of all the objects that match our revision, along with their type and, for commits, the date and the title line.

We'll need to expand the `Database::Commit` and `Database::Author` classes to return this information. We last saw the `Commit` class back in Section 10.1.2, “Parsing commits”, where we reconstructed a `Commit` object we'd read from disk. First, let's add the `title_line` method to return the first line of the commit's message:

```
# lib/database/commit.rb  
  
def title_line  
  @message.lines.first  
end
```

Then we can deal with the more involved changes required for the author information. `Commit.parse` currently constructs a `Commit` without further parsing the author header, since we've not needed to access that information before:

```
# lib/database/commit.rb

def self.parse(scanner)
# ...
Commit.new(
  headers["parent"],
  headers["tree"],
  headers["author"],
  scanner.rest)
end
```

Let's now change things so that the author header is converted from a string back into the `Author` object that existed when the commit was created¹³, by introducing a new method called `Author.parse`:

```
# lib/database/commit.rb

Commit.new(
  headers["parent"],
  headers["tree"],
  Author.parse(headers["author"]),
  scanner.rest)
```

Below is the `Author` class expanded with a new `parse` method that reverses the serialisation performed by `Author#to_s`. We've also added the `short_date` method which returns the author's date in the format used in the error message above.

```
# lib/database/author.rb

require "time"

class Database
  TIME_FORMAT = "%s %z"

  Author = Struct.new(:name, :email, :time) do
    def self.parse(string)
      name, email, time = string.split(/<|>/).map(&:strip)
      time = Time.strptime(time, TIME_FORMAT)

      Author.new(name, email, time)
    end

    def short_date
      time.strftime("%Y-%m-%d")
    end

    def to_s
      timestamp = time.strftime(TIME_FORMAT)
      "#{name} <#{email}> #{timestamp}"
    end
  end
end
```

¹³Section 3.2.3, “Storing commits”

```
end
```

`Time.strptime`¹⁴ is the inverse of `Time#strftime`: it takes a string and a format and converts the string back into a `Time` value by matching it against the format.

Now we can turn our attention back to the `Revision` class and make it detect and report cases where an abbreviated ID is ambiguous. `Revision#resolve` will continue to raise an error for `Command::Branch` to handle, but it will also log any non-crashing failures that happen in the lead-up to this error.

Let's create a new structure called `HintedError` that contains a message and a hint — this will correspond to the `error:` and `hint:` lines in the Git output. We'll also add a readable `@errors` variable to the `Revision` class to store up any errors that happen while a revision is being evaluated.

```
# lib/revision.rb

HintedError = Struct.new(:message, :hint)

attr_reader :errors

def initialize(repo, expression)
  @repo  = repo
  @expr  = expression
  @query = Revision.parse(@expr)
  @errors = []
end
```

We'd like to log a failure when the `Revision#read_ref` method finds more than one candidate object ID from calling `Database#prefix_match`.

```
# lib/revision.rb

def read_ref(name)
  oid = @repo.refs.read_ref(name)
  return oid if oid

  candidates = @repo.database.prefix_match(name)
  return candidates.first if candidates.size == 1

  if candidates.size > 1
    log_ambiguous_sha1(name, candidates)
  end

  nil
end
```

`log_ambiguous_sha1` constructs the error message by reading all the candidate IDs from the database and listing out their short ID and type. If the object is a commit, then it also appends the `Author#short_date` and the `Command#title_line` outputs that we implemented above.

```
# lib/revision.rb

def log_ambiguous_sha1(name, candidates)
```

¹⁴<https://docs.ruby-lang.org/en/2.3.0/Time.html#method-c-strptime>

```
objects = candidates.sort.map do |oid|
  object = @repo.database.load(oid)
  short  = @repo.database.short_oid(object.oid)
  info   = "#{short} #{object.type}"

  if object.type == "commit"
    "#{info} #{object.author.short_date} - #{object.title_line}"
  else
    info
  end
end

message = "short SHA1 #{name} is ambiguous"
hint = ["The candidates are:"]
@errors.push(HintedError.new(message, hint))
end
```

The `Command::Branch` class can now use this extra error information collected by the `Revision` object when it handles the `Revision::InvalidObject` error that's raised by `Revision#resolve`.

```
# lib/command/branch.rb

def create_branch
  # ...
rescue Revision::InvalidObject => error
  revision.errors.each do |err|
    @stderr.puts "error: #{err.message}"
    err.hint.each { |line| @stderr.puts "hint: #{line}" }
  end
  @stderr.puts "fatal: #{error.message}"
  exit 128
end
```

The other kind of error that allowing object IDs in revisions exposes us to is that we might load the wrong type of object. When all we could do was read references and use the `^` and `-` operators, it was only possible to end up with commit IDs. But now we can pick any ID ourselves, we might also get an ID that refers to a tree or a blob when we evaluate a revision.

To guard against this, let's pass in an argument to `Revision#resolve` that identifies the kind of object we want. For the `branch` command, we only want to get commit IDs as the value of `branch` references.

```
# lib/command/branch.rb

revision = Revision.new(repo, start_point)
start_oid = revision.resolve(Revision::COMMIT)
```

The `resolve` method can take this type and use it to check that the ID that results from evaluating the expression does indeed yield an object of the right type.

```
# lib/revision.rb

COMMIT = "commit"

def resolve(type = nil)
  oid = @query&.resolve(self)
```

```
    oid = nil if type and not load_typed_object(oid, type)

    return oid if oid

    raise InvalidObject, "Not a valid object name: '#{ @expr }'."
end
```

Similarly, the `commit_parent` method can use `load_typed_object` to generate an error if the ID it's been asked to get the parent of is not a commit object.

```
# lib/revision.rb

def commit_parent(oid)
  return nil unless oid

  commit = load_typed_object(oid, COMMIT)
  commit.parent
end
```

The `load_typed_object` method takes an object ID and a desired type, loads the object from the Database and checks its type. If the types match then the object is returned, otherwise we add an error to the `Revision#errors` list to alert the user to the source of the failure, and we return `nil`.

```
# lib/revision.rb

def load_typed_object(oid, type)
  return nil unless oid

  object = @repo.database.load(oid)

  if object.type == type
    object
  else
    message = "object #{ oid } is a #{ object.type }, not a #{ type }"
    @errors.push(HintedError.new(message, []))
    nil
  end
end
```

With these error checks in place, the `branch` command will now refuse to create a branch using something that is not a commit ID.

```
$ jit branch topic 1cded5e
error: object 1cded5e81def122d9227c2c5db6c258ad4f40de7 is a tree, not a commit
fatal: Not a valid object name: '1cded5e'.
```

In Git, object types are sometimes used to resolve ambiguous matches. For example, if you ask for a branch to start at `bc24^` and the name `bc24` matches multiple object IDs, we can use the fact that the `^` operator only works on commits to resolve the ambiguity. If only one of the candidate IDs refers to a commit, then Git will use the ID and discard the other candidates. This effect can be achieved by recombining the ingredients we've seen above, but I'll leave this an exercise for the reader.

It still remains for us to implement the branch behaviours for listing and deleting branches. However, these features are best left until after we've implemented checkout. For now, we can

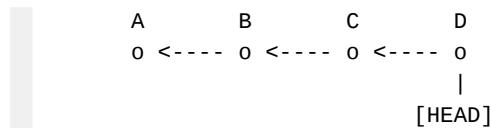
get by with running `ls .git/refs/heads` to list branches, and `rm .git/refs/heads/<branch>` to remove them. After all, branches are just files that store pointers, and we can use regular Unix command-line tools to inspect them.

14. Migrating between trees

In the previous chapter we developed the `branch` command, which creates files under `.git/refs/heads` that contain pointers to specific commits. On its own, this is not particularly useful; it allows us to give memorable names to commits in a linear history, but it doesn't allow us to retrieve the contents of those commits and load them back into our workspace.

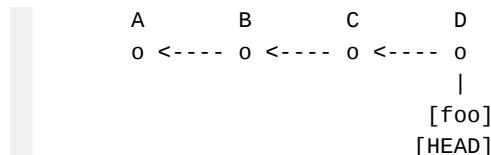
For example, if we have a chain of commits where `.git/HEAD` points at the latest one, and every commit points to its parent:

Figure 14.1. Chain of commits



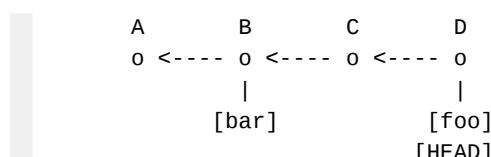
We can use `branch` to create another pointer to the latest commit, or to any previous one. For example, running `branch foo` will create another pointer to commit `D` stored in `.git/refs/heads/foo`. In the following diagrams, `HEAD` is a shorthand for the pointer stored in `.git/HEAD`, while other names are shorthands for pointers in `.git/refs/heads/<name>`.

Figure 14.2. New branch pointing to the last commit



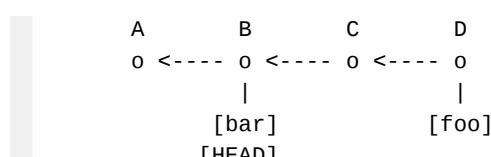
We can create pointers to older commits using the revisions notation¹. Running `branch bar @~2` will create a pointer to commit `B`:

Figure 14.3. New branch pointing to an older commit



To make branches truly useful, we need to be able to switch between different versions of the codebase so that we can begin to build a new chain of commits off that version. For example, if we're in the above state, then we'd like to be able to go back to commit `B`, loading its tree back into our workspace so we can make changes to it. This would also move `HEAD` to reflect what the currently checked-out commit is.

Figure 14.4. Moving `HEAD` to a different branch



¹Section 13.3, “Setting the start point”

The checkout command's job is to apply changes to the repository so that when we run `checkout` bar, the workspace, index and `HEAD` are restored to the state they were in after creating the commit bar points at. In this chapter we'll examine how these changes are performed, before tying the branch, checkout and commit commands together in the next chapter to produce fully working branches.

This is the first time that Jit will be making changes to the workspace and index that are not explicitly specified by the user. The user might have made changes to the workspace and index such that they differ from `HEAD`, putting the repository in an unclean state. We'd like to avoid overwriting such changes if possible, otherwise they'll be lost. If we can't perform the checkout without discarding the user's changes, we should bail out and display an error to the user. To make these decisions, we're going to need a way to tell which files need changing in order to migrate from the current `HEAD` to whichever commit `<rev>` resolves to. Then, we need to apply these changes to the workspace and index, and finally adjust `HEAD` to reflect our new position in the commit graph.

When we run `checkout`, the first thing we need to do is amend the workspace so that it reflects the tree of the commit we're moving to. This task begins with the question: which files do we need to change in order that the workspace reflects the new tree?

14.1. Telling trees apart

When the user has not made any uncommitted changes, the workspace and index reflect the tree that belongs to the commit that `HEAD` points at. Executing a checkout means that we change the contents of the workspace and index to reflect the tree of a different commit. The simplest way to implement this would be to delete every tracked file in the workspace and delete the index, and then rebuild both of them from the new tree. This is a straightforward way to ensure all the repository components are in a consistent state.

However this suffers from two major drawbacks. First, any uncommitted changes made by the user will be lost, which severely restricts the utility of the checkout command. If the only files you've changed are ones that do not differ between `HEAD` and `<rev>`, there's no reason you shouldn't be able to keep those changes while switching from `HEAD` to `<rev>`, and quickly test your changes against different versions of the codebase without having to commit and merge your changes into multiple branches. Second, it's wasteful. Typically, the number of files that differ between `HEAD` and `<rev>` is small relative to the total number of files in the tree, so there's a performance win to be had in only changing those files that differ, rather than deleting everything and then recreating all the files as they were before.

Therefore, it would be useful to figure out which files actually differ between `HEAD` and `<rev>`, so that we can update only those files, and leave any unchanged ones alone. This will also let us preserve non-conflicting uncommitted changes, and exit with an error if there are any conflicts.

Imagine we have a project where the penultimate commit, `@^`, has the tree shown on the left, and the latest commit, `@`, has the tree on the right. Remember that a tree in the database is a list of entries that have a name, mode, and object ID, and those with mode `400008` are pointers to other trees. The listing below shows the fully expanded tree; the entries named `config`, `database`, `lib` and `models` point to sub-trees, and all the other entries point to blobs. Each entry is listed with its abbreviated object ID.

Figure 14.5. Trees of two adjacent commits

```

@^:
├── README.md [45a4fb7]
├── TODO.md [ec63514]
└── config [9bc815e]
    └── database [39995f4]
        ├── development.json [d00491f]
        └── production.json [0cfbf08]
└── lib [bfe70e0]
    └── app.rb [00750ed]

@:
├── README.md [45a4fb7]
├── config [9bc815e]
    └── database [39995f4]
        ├── development.json [d00491f]
        └── production.json [0cfbf08]
└── lib [8be4dc3]
    ├── app.rb [64bb6b7]
    └── models [0e0262f]
        └── repository.rb [7ed6ff8]
        └── user.rb [b8626c4]

```

We'd like to compile a list of files that differ between these two trees. Let's start by examining just the top-level tree for each commit, ignoring any sub-trees that might be present. The following listing reproduces the above trees with all the sub-trees removed.

Figure 14.6. Top level of trees from two commits

```

@^:
├── README.md [45a4fb7]
├── TODO.md [ec63514]
└── config [9bc815e]
    └── lib [bfe70e0]

@:
├── README.md [45a4fb7]
└── config [9bc815e]
    └── lib [8be4dc3]

```

Let's check the entries on the left one-by-one. `README.md` has object ID `45a4fb7` in both trees, and so this file has not changed. `TODO.md` is present on the left but not on the right, which tells us this file has been deleted.

Next, the `config` entry has ID `9bc815e` in both trees, and so it too is unchanged. Even though `config` is a directory, we don't need to check inside it, because the `config` entries in each tree point at exactly the same tree object: `9bc815e`. This lets us skip all the work of examining the files inside this sub-tree as we know they'll be identical. This is one of the big advantages of the Merkle tree construction²: not only does it save space by causing trees to share references to a unique copy of an unchanged object, it also makes detecting changes across trees much faster.

Finally, `lib` is also a tree entry, but its version differs: `bfe70e0` on the left, `8be4dc3` on the right. Since it's a tree entry, we don't record that as a change in itself. Instead, we recurse into the `lib` subtree on each side by following the object pointers and comparing the results.

The `lib` subtrees in our two commits look like this, again showing only their immediate children and no sub-trees:

Figure 14.7. Trees for the `lib` sub-tree of two commits

```

@^:lib
└── app.rb [00750ed]

@:lib
└── app.rb [64bb6b7]
    └── models [0e0262f]

```

Here we can see that the blob entry `app.rb` has changed from `00750ed` to `64bb6b7`, so we can record that as a change to `lib/app.rb`. Meanwhile, the tree entry `models` appears on the right but not on the left: it's a new directory. Again, rather than marking that as a change in itself we usually want to know about the individual files inside the directory, so we follow the tree one level deeper into `lib/models`.

²Section 5.2.2, “Building a Merkle tree”

There is no sub-tree `lib/models` in the left-hand commit, so we'll treat that as an empty tree, while the right-hand commit contains two blob entries for this tree:

Figure 14.8. Trees for the `lib/models` sub-tree of two commits

```

@^:lib/models
          @:lib/models
          └── repository.rb [7ed6ff8]
          └── user.rb [b8626c4]

```

Since the left-hand tree is empty, we just mark all the files on the right-hand side as newly created. Putting all the changes we noticed together, we end up with this set of changes:

- `TODO.md`, which had the contents of blob `ec63514`, has been deleted
- `lib/app.rb` has been edited, changing from blob `00750ed` to `64bb6b7`
- `lib/models/repository.rb` has been created with the contents of blob `7ed6ff8`
- `lib/models/user.rb` has been created with the contents of blob `b8626c4`

This information could be encoded as the following Ruby data structure, which maps pathnames to a pair of values representing the file's content in each of the compared trees.

```

{
  "TODO.md"           => ["ec63514", nil],
  "lib/app.rb"         => ["00750ed", "64bb6b7"],
  "lib/models/repository.rb" => [nil, "7ed6ff8"],
  "lib/models/user.rb"    => [nil, "b8626c4"]
}

```

In practice, the structure will be slightly more complicated than this. We'll have `Pathname` objects rather than bare strings as the keys, and `Database::Entry` objects (which have an `oid` and `mode`) rather than simple object IDs in the pairs on the right. We'll also check the modes of entries as we compare them, not just the object IDs of their content.

To translate this process into working code, we'll create a new class called `Database::TreeDiff` that will implement the change detection routine. It will take a `Database` so that it can read trees from `.git/objects`, and exposes a hash called `@changes` that will accumulate a record of file differences.

```

# lib/database/tree_diff.rb

class Database
  class TreeDiff

    attr_reader :changes

    def initialize(database)
      @database = database
      @changes = {}
    end

    # ...

  end
end

```

Comparing two trees is done by passing their object IDs to the method `compare_oids`, along with an optional path `prefix` whose default value is empty. This prefix will be used to construct the full path to each changed file as we recurse down the trees.

If the two given object IDs are the same (either the same string value, or both `nil`) then this method exits immediately since there cannot be any differences between two references to the same object. Otherwise, we use `oid_to_tree` with each ID to load the tree from the database and extract the entries from it, substituting the empty hash `{}` if either ID is `nil`. `oid_to_tree` can cope with being given either a commit or a tree ID as input; if it loads the object from the Database and gets a `Commit` object, then it just loads the commit's tree pointer³. Once we have two sets of tree entries, we invoke the `detect_deletions` and `detect_additions` methods to compare their contents.

```
# lib/database/tree_diff.rb

def compare_oids(a, b, prefix = Pathname.new(""))
  return if a == b

  a_entries = a ? oid_to_tree(a).entries : {}
  b_entries = b ? oid_to_tree(b).entries : {}

  detect_deletions(a_entries, b_entries, prefix)
  detect_additions(a_entries, b_entries, prefix)
end

def oid_to_tree(oid)
  object = @database.load(oid)

  case object
  when Commit then @database.load(object.tree)
  when Tree   then object
  end
end
```

Recall that a `Tree` object's `entries` structure is a hash mapping names of entries to `Entry` objects. For example, the root tree in the left-hand commit above looks like this:

```
{
  "README.md" => Database::Entry(oid="45a4fb7", mode=0100644),
  "TODO.md"   => Database::Entry(oid="ec63514", mode=0100644),
  "config"     => Database::Entry(oid="9bc815e", mode=040000),
  "lib"        => Database::Entry(oid="bfe70e0", mode=040000)
}
```

The `detect_deletions` method shown below does most of the work of detecting changes between the two trees. It iterates over the entries in the first tree, and fetches the corresponding entry by name from the second tree — `entry` is the left-hand `Entry` object, `other` the right-hand one. If these entries are equal, that is, their modes and object IDs are the same, we skip to the next entry. Otherwise, we take two further processing steps, one for trees and one for blobs.

The values `tree_a` and `tree_b` are set to the object IDs of `entry` and `other` respectively, if `entry` or `other` is not `nil` and points to a tree. If `entry` does not point to a tree then `tree_a` is

³Git refers to an object that can be coerced into a tree object as a *tree-ish*. Commits are tree-ish because they all carry a pointer to a tree as part of their content.

`nil` and similarly for `other` and `tree_b`. We then pass the resulting IDs to `compare_oids`, to recurse into the sub-trees if any are present at this point in the scan.

Finally we compare blob entries, if any are present. `blobs` is set to the pair of `Entry` objects `[entry, other]`, with either member set to `nil` if it refers to a sub-tree. That is, we only retain differences between *files*; if a file has been replaced with a directory, then this change is recorded as `[entry, nil]`. When we process these changes, directories will be implicitly created as needed in the workspace to contain any new files inside the new sub-tree. If this pair contains any non-`nil` values, then it's added to `@changes` against the current path.

```
# lib/database/tree_diff.rb

def detect_deletions(a, b, prefix)
  a.each do |name, entry|
    path = prefix.join(name)
    other = b[name]

    next if entry == other

    tree_a, tree_b = [entry, other].map { |e| e.&.tree? ? e.oid : nil }
    compare_oids(tree_a, tree_b, path)

    blobs = [entry, other].map { |e| e.&.tree? ? nil : e }
    @changes[path] = blobs if blobs.any?
  end
end
```

`detect_additions` takes care of any entries that exist in the second tree but not in the first. Iterating over `b` rather than `a`, it skips any entries that exist in `a`. For the remaining missing entries, if the current entry in `b` points to a tree, then we recursively call `compare_oids` with its ID, passing `nil` for the first argument to represent the missing sub-tree on the left. Otherwise, the entry points to a blob and represents a newly added file, which we record in `@changes`.

```
# lib/database/tree_diff.rb

def detect_additions(a, b, prefix)
  b.each do |name, entry|
    path = prefix.join(name)
    other = a[name]

    next if other

    if entry.tree?
      compare_oids(nil, entry.oid, path)
    else
      @changes[path] = [nil, entry]
    end
  end
end
```

As a convenience, we'll add an interface in the `Database` class for calculating and returning the diff between two commits or trees, by invoking the `TreeDiff` class and returning the resulting changes hash.

```
# lib/database.rb
```

```
def tree_diff(a, b)
  diff = TreeDiff.new(self)
  diff.compare_oids(a, b)
  diff.changes
end
```

We now have a method for determining all the tree-level differences between commits. This will also be useful for commands like `log`⁴ and `merge`⁵, but for now we'll focus on the next task checkout must perform: applying this diff to the workspace and index.

14.2. Planning the changes

Now that we have a method for detecting the differences between two trees, we can begin to build the checkout command around that. This command is given a revision as an argument, and it will begin by using the `Revision` class⁶ to resolve this to a commit ID. It will also fetch the current commit ID by reading `.git/HEAD` using `Refs#read_head`. Given these two IDs, we'll create a *migration* by calling a new method, `Repository#migration`, and tell it to apply its changes to the repository state. The error handling for invalid revisions is similar to that from `Command::Branch`⁷.

```
# lib/command/checkout.rb

module Command
  class Checkout < Base

    def run
      @target = @args[0]

      @current_oid = repo.refs.read_head

      revision = Revision.new(repo, @target)
      @target_oid = revision.resolve(Revision::COMMIT)

      tree_diff = repo.database.tree_diff(@current_oid, @target_oid)
      migration = repo.migration(tree_diff)
      migration.apply_changes

      exit 0

    rescue Revision::InvalidObject => error
      handle_invalid_object(revision, error)
    end

    private

    def handle_invalid_object(revision, error)
      revision.errors.each do |err|
        @stderr.puts "error: #{err.message}"
        err.hint.each { |line| @stderr.puts "hint: #{line}" }
      end
      @stderr.puts "error: #{error.message}"
    end
  end
end
```

⁴Section 16.1.5, “Displaying patches”

⁵Section 17.4, “Performing a merge”

⁶Section 13.3, “Setting the start point”

⁷Section 13.3.3, “Revisions and object IDs”

```
    exit 1
  end

end
end
```

The `Repository#migration` method constructs and returns a `Repository::Migration` object, taking the current repository and the two given commit IDs as input.

```
# lib/repository.rb

def migration(tree_diff)
  Migration.new(self, tree_diff)
end
```

The `Migration` class's job is to plan the changes that need to be made to the workspace, check they do not conflict with any uncommitted changes made by the user, and then execute those changes. We'll skip conflict detection for now, and focus on planning the changes. The class has a structure called `@changes` that holds lists of files that need either creating, updating or deleting, and then two sets of directory names: `@mkdirs` collects directories we need to create, and `@rmdir`s collects directories we need to delete.

```
# lib/repository/migration.rb

class Repository
  class Migration

    attr_reader :changes, :mkdirs, :rmdir

    def initialize(repository, tree_diff)
      @repo = repository
      @diff = tree_diff

      @changes = { :create => [], :update => [], :delete => [] }
      @mkdirs = Set.new
      @rmdir = Set.new
    end

    #
    #

    end
  end
end
```

The `apply_changes` method that's called by `Command::Checkout` is a high-level plan for the phases of the migration. For now, it just plans the changes, and then updates the workspace.

```
# lib/repository/migration.rb

def apply_changes
  plan_changes
  update_workspace
end
```

The purpose of the `plan_changes` method is to figure out what changes we'd need to make in the filesystem so that the workspace is made to reflect the tree of the commit we're checking out. The output of `Database#tree_diff` is not quite in the right shape for doing this; it's enough

information for, say, displaying a diff between two commits, but *executing* those changes requires a little more planning.

Every tree in the database represents a valid arrangement of files that can exist on disk; the code we saw in Section 7.3, “Stop making sense” makes sure the index does not contain files whose names clash with directories required to hold other files. So it must be possible to apply a tree diff to a set of files on disk and get another valid arrangement, the only question is how can we guarantee that.

In particular, the entries in a tree diff are not in any specific order. But, when we execute those changes in the filesystem, order becomes important. If the current HEAD contains a file called `lib/app`, but the new tree contains one called `lib/app/user.rb`, then the file `lib/app` must be deleted from the workspace before we try to create `lib/app/user.rb`. That is, this sequence of commands will work:

```
rm lib/app
mkdir lib/app
echo "<file contents>" > lib/app/user.rb
```

Whereas the following will not — the existence of a file at `lib/app` means a directory cannot be created there, and `mkdir` will fail.

```
mkdir lib/app
echo "<file contents>" > lib/app/user.rb
rm lib/app
```

So, splitting the changes in a tree diff into multiple categories means we can order things so that all deletions are performed first, clearing the path for any new files that should be created. Updates to files do not conflict with anything else; if a tree diff says a file has been updated, this implies all the directories above it still exist, and it has not been replaced with a directory. Nevertheless, it is useful for the purpose of clarity to separate out updates into their own list.

The `plan_changes` method scans over the tree diff entries, dividing them into creations (those entries where the file does not exist in the old commit), updates (where the file exists in both commits) and deletions (where the file does not exist in the new commit). For deletions, all the file’s parent directories are added to the `@rmdir`s list as candidates for deletion, while for creations and updates we add the parent directories to `@mkdir`s to ensure these directories exist prior to writing the file.

```
# lib/repository/migration.rb

def plan_changes
  @diff.each do |path, (old_item, new_item)|
    record_change(path, old_item, new_item)
  end
end

def record_change(path, old_item, new_item)
  if old_item == nil
    @mkdirs.merge(path.dirname.descend)
    action = :create
  elsif new_item == nil
    @rmdir.merge(path.dirname.descend)
    action = :delete
  else
    @mkdirs.merge(path.dirname.descend)
    @rmdir.merge(path.dirname.descend)
    action = :update
  end
end
```

```
    else
      @mkdirs.merge(path.dirname.descend)
      action = :update
    end
    @changes[action].push([path, new_item])
  end
```

We only need to record the new item in the change plan, because we don't need to know the file's old contents in order to delete or overwrite it. This data structure now encodes enough information to be directly executed against the filesystem.

14.3. Updating the workspace

Since applying these changes requires updating files in the workspace, we'll delegate responsibility for performing them to the `Workspace` class rather than having the `Migration` class talk directly to the filesystem. The `Migration#update_workspace` method simply passes the migration object to the `Workspace` for execution.

```
# lib/repository/migration.rb

def update_workspace
  @repo.workspace.apply_migration(self)
end
```

`Workspace#apply_migration` takes a `Migration` and executes its change plan. It first applies the deletions, then it attempts to delete any directories that might have been left empty as a result of deleting these files. We iterate over `rmdir`s in reverse sort order because then we'll delete each directory before attempting to delete its parent. You cannot delete a directory that is not empty, so it's important that each directory's empty subdirectories are deleted before trying to delete the directory itself.

Next, it creates any directories that are necessary to hold the created and updated files. This is done in forward sort order on the directory name, because a directory must exist before sub-directories can be added to it. After creating these directories, we apply the updates and creations.

```
# lib/workspace.rb

def apply_migration(migration)
  apply_change_list(migration, :delete)
  migration.rmdir.sort.reverse_each { |dir| remove_directory(dir) }

  migration.mkdirs.sort.each { |dir| make_directory(dir) }
  apply_change_list(migration, :update)
  apply_change_list(migration, :create)
end
```

`apply_change_list` does the work of putting the files in the workspace in the correct state. It fetches a list of changes from the `Migration` using the `action` key passed in by `apply_migration`, and iterates over them. For each file, we delete the file if it exists, including deleting any directory that might currently be at that path, by calling `FileUtils.rm_rf`⁸. If the action is `:delete` we skip to the next change at this point.

⁸https://docs.ruby-lang.org/en/2.3.0/FileUtils.html#method-i-rm_rf

For creations and updates, we open the file and write the new data to it. We use the flags `WRONLY` as we only want to write to the file, `CREAT` so the file is created if it does not exist (which it should not as we just deleted it), and `EXCL` so that opening the file fails if it already exists. This prevents this writing bad data because another process was modifying the filesystem at the same time. Finally we use `File.chmod`⁹ to set the file's mode to that stored in the tree.

```
# lib/workspace.rb

def apply_change_list(migration, action)
  migration.changes[action].each do |filename, entry|
    path = @pathname.join(filename)

    FileUtils.rm_rf(path)
    next if action == :delete

    flags = File::WRONLY | File::CREAT | File::EXCL
    data = migration.blob_data(entry.oid)

    File.open(path, flags) { |file| file.write(data) }
    File.chmod(entry.mode, path)
  end
end
```

This method is not given blob data immediately; we don't want to read the contents of all updated files into memory all at once in case that gets particularly large. We're given a `Database::Entry` from which we fetch the blob ID, and we can use this to ask the `Migration` to load the blob's data for us.

```
# lib/repository/migration.rb

def blob_data(oid)
  @repo.database.load(oid).data
end
```

This method exists because `Workspace` and `Database` are low-level components with direct access to the filesystem and they're not coupled to each other, which means `Workspace` cannot ask `Database` directly for information. `Migration` is a higher-level workflow tool that exists to integrate changes to multiple low-level subsystems, so it has access to all the repository's components.

The other elements of the `Workspace#apply_migration` method are calls to `remove_directory` and `make_directory`. These are wrappers around the `Dir.rmdir`¹⁰ and `Dir.mkdir`¹¹ methods that reflect the purposes we're using them for here. When we call `remove_directory` it's because we'd like to remove any directories that are empty after deleting files. If the directory is not actually empty, or does not exist, or is not a directory, we don't mind, so we rescue any of these errors if they happen. We use `make_directory` to create any directories that *must* exist in order for subsequent file writes to succeed, so this does a bit more work to make sure this happens. It calls `stat()` on the filename, deletes it if it's a regular file, and then creates a directory there unless the file is already a directory.

```
# lib/workspace.rb
```

⁹<https://docs.ruby-lang.org/en/2.3.0/File.html#method-c-chmod>

¹⁰<https://docs.ruby-lang.org/en/2.3.0/Dir.html#method-c-rmdir>

¹¹<https://docs.ruby-lang.org/en/2.3.0/Dir.html#method-c-mkdir>

```
def remove_directory(dirname)
  Dir.rmdir(@pathname.join(dirname))
rescue Errno::ENOENT, Errno::ENOTDIR, Errno::ENOTEMPTY
end

def make_directory(dirname)
  path = @pathname.join(dirname)
  stat = stat_file(dirname)

  File.unlink(path) if stat&.file?
  Dir.mkdir(path) unless stat&.directory?
end
```

You might notice that these methods do more work than is strictly necessary if the repository is in a clean state with no uncommitted changes. If there's a file somewhere that we expected a directory, or vice versa, that means that file is untracked, and we just deleted it. As it turns out, Git does not preserve all untracked changes; there are some situations where if you have a file where a directory was expected or vice versa, Git will delete the file and all its contents will be lost.

14.4. Updating the index

After running `checkout`, the `status` output should be clean — if your repository had no uncommitted changes, then nothing should be printed when you run `status`, and if there *were* uncommitted changes, those are the only things `status` should print after `checkout`. This means, the index should reflect what's now in the workspace, and it should reflect the contents of the commit we're moving to. We can make all these things the case by applying the migration changes to the index as well as the workspace, and by updating `.git/HEAD` to point at the commit we're migrating to.

In `Command::Checkout`, let's open the `Index` for updates before applying the migration, and then save any updates after the migration succeeds. We'll end by using `Refs#update_head` to put the new commit ID in `.git/HEAD`.

```
# lib/command/checkout.rb

repo.index.load_for_update

tree_diff = repo.database.tree_diff(@current_oid, @target_oid)
migration = repo.migration(tree_diff)
migration.apply_changes

repo.index.write_updates
repo.refs.update_head(@target_oid)
```

We add a new step to `Repository::Migration#apply_changes` that will use `Index#add` and `Index#remove` to reflect the changes made to the workspace. Deletions can be performed simply by calling `Index#remove`, but other changes need to collect `stat()` information from the workspace so we can cache it in the updated index entry. This is why the workspace should be up to date before we execute these index changes.

```
# lib/repository/migration.rb
```

```
def apply_changes
  plan_changes
  update_workspace
  update_index
end

def update_index
  @changes[:delete].each do |path, _|
    @repo.index.remove(path)
  end

  [:+create, :update].each do |action|
    @changes[action].each do |path, entry|
      stat = @repo.workspace.stat_file(path)
      @repo.index.add(path, entry.oid, stat)
    end
  end
end
```

`Index#remove` is actually a new method we've not used before, but it can be implemented fairly simply on top of methods already introduced in Section 7.3, "Stop making sense" for ensuring consistency. Given a pathname, it deletes the entry at exactly the given pathname if one exists, and deletes any index entries that are nested under that name if it's a directory using the `remove_children` that we defined earlier.

```
# lib/index.rb

def remove(pathname)
  remove_entry(pathname)
  remove_children(pathname.to_s)
  @changed = true
end
```

Like the workspace changes, this is more destructive than we'd expect if the repository did not have any uncommitted changes, like a file being replaced with a directory containing other tracked files. However, again it turns out that Git will actually evict such entries from the index if you perform a checkout, and those changes will be lost.

14.5. Preventing conflicts

The methods we've been discussing for migrating the workspace and index are completely general; they should put the workspace and index in the correct state no matter what state it starts in. However, there are some states where we'd rather not execute any of these changes. A simple example is when you've made an unstaged change to a file, and that same file differs between the current `HEAD` and the target commit. Applying the tree diff in this situation would result in your changed file being overwritten and your changes — which are not persisted in the index or database — being discarded.

In this situation and many other conflict scenarios, Git will refuse to execute the checkout. To decide whether a checkout can go ahead, it needs to check whether any files listed in the tree diff coincide with untracked, unstaged or uncommitted changes in your workspace and index. We already have a class that detects such things: the `Repository::Status` class. The problem is that we want to check specific individual files, whereas the `Status` class is a monolithic full-scan process of the entire workspace and index.

14.5.1. Single-file status checks

We can extract some elements of `Repository::Status` that would be useful to us here. The `trackable_file?` method¹² can be extracted wholesale, and that will let us detect that a path corresponds to an untracked file or a directory containing untracked files. The `check_index_against_workspace` method¹³ compares an index entry to the file in the workspace and either updates the cached index entry or records that the file contains unstaged changes; we don't want these side effects when we're checking a `Migration` but we could reuse the comparison logic. Similarly, `check_index_against_head_tree`¹⁴ contains the logic for comparing an index entry to an item from the `HEAD` tree, and logging an uncommitted change if they differ; the comparison performed by this method can also be extracted.

Suppose that in the constructor of the `Repository::Status` class we create an instance of a new class called `Repository::Inspector`:

```
# lib/repository/status.rb

def initialize(repository)
  @inspector = Inspector.new(repository)
  #
end
```

Then, in the `Status` methods for checking index entries, we replace the comparison logic with a call to a new method on the `Inspector` class, and leave the side effects in the `Status` class. Note how the `Inspector` class's methods take `File::Stat`, `Index::Entry` and `Database::Entry` objects as input; the class isn't responsible for statting files or fetching things from the index or database, it assumes the caller knows how to fetch the data it wants to compare. The `Inspector` methods return a symbol, one of `:untracked`, `:modified`, `:added` or `:deleted` to represent the state of the file, and `Status` uses this symbol to record the change.

```
# lib/repository/status.rb

def check_index_against_workspace(entry)
  stat = @stats[entry.path]
  status = @inspector.compare_index_to_workspace(entry, stat)

  if status
    record_change(entry.path, @workspace_changes, status)
  else
    @repo.index.update_entry_stat(entry, stat)
  end
end

def check_index_against_head_tree(entry)
  item = @head_tree[entry.path]
  status = @inspector.compare_tree_to_index(item, entry)

  if status
    record_change(entry.path, @index_changes, status)
  end
end
```

¹²Section 9.1.3, “Empty untracked directories”

¹³Section 9.2.4, “Timestamp optimisation”

¹⁴Section 10.2.2, “Modified files”

The `trackable_file?` method and the comparison logic is then extracted into the new `Repository::Inspector` class. The `trackable_file?` method is a verbatim copy of the method from `Status`, while the other methods are a slightly modified version of the `Status` comparison logic that allows us to pass in any objects we want to compare and get a symbol back telling us if anything has changed. If the objects being compared are equal, then these comparison methods return `nil`.

```
# lib/repository/inspector.rb

class Repository
  class Inspector

    def initialize(repository)
      @repo = repository
    end

    def trackable_file?(path, stat)
      # as before
    end

    def compare_index_to_workspace(entry, stat)
      return :untracked unless entry
      return :deleted unless stat
      return :modified unless entry.stat_match?(stat)
      return nil if entry.times_match?(stat)

      data = @repo.workspace.read_file(entry.path)
      blob = Database::Blob.new(data)
      oid = @repo.database.hash_object(blob)

      unless entry.oid == oid
        :modified
      end
    end

    def compare_tree_to_index(item, entry)
      return nil unless item or entry
      return :added unless item
      return :deleted unless entry

      unless entry.mode == item.mode and entry.oid == item.oid
        :modified
      end
    end

  end
end
```

We can now use the `Inspector` class inside the `Migration` class to check every entry in the tree diff for conflicts with uncommitted changes.

14.5.2. Checking the migration for conflicts

In the `Migration` constructor, let's create an instance of `Inspector`, an `@errors` array to collect error messages to display to the user, and then a structure called `@conflicts` that will store the names of all conflicted files, organised by which type of conflict they're exhibiting.

```
# lib/repository/migration.rb

Conflict = Class.new(StandardError)

attr_reader :changes, :mkdirs, :rmdir, :errors

def initialize(repository, tree_diff)
# ...

@inspector = Inspector.new(repository)
@errors = []

@conflicts = {
  :stale_file => SortedSet.new,
  :stale_directory => SortedSet.new,
  :untracked_overwritten => SortedSet.new,
  :untracked_removed => SortedSet.new
}
end
```

Git has four different error messages it can display for conflicting changes, and the four keys in the @conflicts hash will correspond to these messages.

We need to populate the @conflicts record as we prepare the change plan, and once we've finished planning we should fill the @errors array with a list of messages derived from the conflicts we found. Let's add a couple of hooks into the Migration#plan_changes method for performing these tasks:

```
# lib/repository/migration.rb

def plan_changes
@diff.each do |path, (old_item, new_item)|
  check_for_conflict(path, old_item, new_item)
  record_change(path, old_item, new_item)
end

collect_errors
end
```

For each item in the tree diff, we call `check_for_conflict` with the item's pathname, and its state in the old tree and in the new tree. `path` is a `Pathname` object, and `old_item` and `new_item` are both either a `Database::Entry` or `nil`. The `check_for_conflict` method runs through a series of checks for situations where executing the checkout would result in uncommitted changes being lost, and records any conflicts found in the `@conflicts` record.

```
# lib/repository/migration.rb

def check_for_conflict(path, old_item, new_item)
entry = @repo.index.entry_for_path(path)

if index_differs_from_trees(entry, old_item, new_item)
  @conflicts[:stale_file].add(path.to_s)
  return
end

stat = @repo.workspace.stat_file(path)
type = get_error_type(stat, entry, new_item)
```

```

if stat == nil
  parent = untracked_parent(path)
  @conflicts[type].add(entry ? path.to_s : parent.to_s) if parent

elsif stat.file?
  changed = @inspector.compare_index_to_workspace(entry, stat)
  @conflicts[type].add(path.to_s) if changed

elsif stat.directory?
  trackable = @inspector.trackable_file?(path, stat)
  @conflicts[type].add(path.to_s) if trackable
end
end

```

The method begins by loading the `Index::Entry` for the given path. The first check it performs is a call to `index_differs_from_trees`, which returns true if the index entry differs from both the states given in the tree diff. If the index entry's mode and object ID are equal to those of `old_item`, that means the index is unchanged from HEAD for this file, so there is no uncommitted change. If the index entry is equal to `new_item`, then applying the tree diff will have no effect, so it's still safe. Only if the index entry differs from `old_item` and `new_item` do we have uncommitted content that will be lost if we migrate.

```

# lib/repository/migration.rb

def index_differs_from_trees(entry, old_item, new_item)
  @inspector.compare_tree_to_index(old_item, entry) and
  @inspector.compare_tree_to_index(new_item, entry)
end

```

Recall that `Inspector#compare_tree_to_index` returns a truthy value (a symbol) if its inputs differ. If it returns a symbol for both `old_item` and `new_item`, then `check_for_conflict` records a `:stale_file` error and exits. If the index entry matches either `old_item` or `new_item`, then we move on to check for various conditions that could exist in the workspace. It grabs the `File::Stat` of the path in question, and performs a number of tests depending on the result.

If `stat` is `nil`, i.e. there is no file in the workspace at the given path, that's not a problem on its own. Applying a tree diff will either delete a file, which is fine if it doesn't exist, or create or update it, which are also fine because there's no data at the given path that would be lost. However, it's possible that an untracked file exists at a parent path. For example, if we're checking `lib/app.rb`, which is updated in our example tree diff, and that file no longer exists but a file exists at `lib`, then applying the tree diff would require deleting the file at `lib` so that `lib/app.rb` could be written. This would result in lost data and so it blocks the checkout from proceeding. This check is done using the `Migration#untracked_parent` method, which stats every path that contains the one in question to see if an untracked file is present there.

```

# lib/repository/migration.rb

def untracked_parent(path)
  path.dirname.ascend.find do |parent|
    next if parent.to_s == ".."

    parent_stat = @repo.workspace.stat_file(parent)
    next unless parent_stat&.file?
  end
end

```

```
    @inspector.trackable_file?(parent, parent_stat)
  end
end
```

If `stat` represents a regular file, then we want to know if the file in the workspace contains changes that have not been added to the index. We call `Inspector#compare_index_to_workspace` to perform this check, and record an error if it returns anything, which would happen if the file is not in the index (entry is `nil`), or the index entry and the file have different contents. This check prevents the checkout from overwriting changes that exist only in the workspace and have not been committed to the index or database.

Finally, if `stat` represents a directory, then it may be in conflict with the tree diff, because tree diffs only contain paths of regular files. We want to know if the directory contains any untracked files that would be deleted if we removed the directory to make way for a file present in the new tree. This check can be performed by a call to `Inspector#tracked_file?`.

The above checks on the workspace file add conflicted pathnames to `@conflicts[type]`, where `type` is determined by calling this method with the `File::Stat`, `Index::Entry` and the new `Database::Entry` for the path we're checking.

```
# lib/repository/migration.rb

def get_error_type(stat, entry, item)
  if entry
    :stale_file
  elsif stat.directory?
    :stale_directory
  elsif item
    :untracked_overwritten
  else
    :untracked_removed
  end
end
```

If the path is in the index, then we use `:stale_file` as the error type; if the path refers to a directory then it's `:stale_directory`; if the path exists in the new tree then we use `:untracked_overwritten` and if none of the above apply it's `:untracked_removed`. This choice of type simply determines which error message Git shows for each conflicting path.

The `check_for_conflict` method is quite a lot to take in, but it's a good example of the fact that validation and error detection logic can often account for much more code than the success-path behaviour of a component. By incorporating the above logic, `Migration` acts like a database transaction by making sure that a checkout can either be performed completely or not at all. We don't want to begin executing a checkout and find out part-way through that one of its changes would cause data loss, because quitting at that point would leave the user's repository in a very confusing state. It's important in situations like this to plan a set of changes and verify that they are free of conflicts before performing any of them.

14.5.3. Reporting conflicts

The final stage of preventing conflicts is to report any problems you found to the user, so they know why we refused to run the checkout and can amend any problems. The

Migration#collect_errors method is called at the end of plan_changes and gathers all the conflicts up into a list of error messages. If @conflicts contains any paths for a given type, we build a list of these paths and top-and-tail it with some copy contained in a constant called MESSAGES. If @errors is not empty at the end of this, we raise an exception to block the migration from proceeding.

```
# lib/repository/migration.rb

def collect_errors
  @conflicts.each do |type, paths|
    next if paths.empty?

    lines = paths.map { |name| "\t#{ name }" }
    header, footer = MESSAGES.fetch(type)

    @errors.push([header, *lines, footer].join("\n"))
  end

  raise Conflict unless @errors.empty?
end
```

The MESSAGES constant contains all the conflict error messages that Git uses to explain what's wrong with the repository:

```
# lib/repository/migration.rb

MESSAGES = {
  :stale_file => [
    "Your local changes to the following files would be overwritten by checkout:",
    "Please commit your changes or stash them before you switch branches."
  ],
  :stale_directory => [
    "Updating the following directories would lose untracked files in them:",
    "\n"
  ],
  :untracked_overwritten => [
    "The following untracked working tree files would be overwritten by checkout:",
    "Please move or remove them before you switch branches."
  ],
  :untracked_removed => [
    "The following untracked working tree files would be removed by checkout:",
    "Please move or remove them before you switch branches."
  ]
}
```

All we need to do now is catch this exception in Command::Checkout. We handle it by unlocking the index, discarding any changes that might have been made, and then printing all the messages we gathered during collect_errors.

```
# lib/command/checkout.rb

def run
  # ...
rescue Repository::Migration::Conflict
  handle_migration_conflict(migration)
end
```

```
def handle_migration_conflict(migration)
  repo.index.release_lock

  migration.errors.each do |message|
    @stderr.puts "error: #{ message }"
  end
  @stderr.puts "Aborting"
  exit 1
end
```

14.6. The perils of self-hosting

We now have a checkout command that can recall the tree from any commit in the repository, and load it into the workspace and index. Let's test it out by checking which files exist in the lib/repository directory and then switching to a commit before inspector.rb was introduced. Here is the directory as of our latest commit:

```
$ ls lib/repository
inspector.rb migration.rb status.rb
```

Let's create a branch here, called master, so that we have a saved reference to the latest commit and we can retrieve it after we've checked out an older version.

```
$ jit branch master
```

Now, if we checkout master~2, we see that inspector.rb has disappeared! We now have the codebase in the state it was in before we added conflict detection.

```
$ jit checkout master~2

$ ls lib/repository
migration.rb status.rb
```

And, we can get back to the latest version of the code by checking out master again.

```
$ jit checkout master

$ ls lib/repository
inspector.rb migration.rb status.rb
```

However, we must use our new-found power carefully. We can't go back to arbitrary points in history, because the checkout command is running code that's in this project's workspace. If we go back too far, to a time before lib/command/checkout.rb existed:

```
$ jit checkout master~5

$ ls lib/command
add.rb    base.rb   branch.rb commit.rb diff.rb   init.rb   status.rb
```

Then it becomes impossible to reach any other version, as the code for the checkout command has vanished.

```
$ jit checkout master
jit: 'checkout' is not a jit command.
```

This sort of problem is endemic to projects that are *self-hosting*, which means that they somehow operate on their own source code. Much software that deals with source code, such

as version control systems, compilers, linters and so on, can often process its own source code, and this sometimes means putting the project in a state where it will no longer run, and the code required to fix it no longer exists!

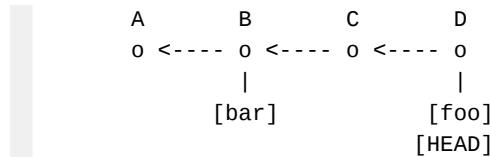
This is why, if you're ever uncertain about whether an action is safe, it's best to save off a backup of the project when it's in a working state. In Jit's case, the code for checkout does still exist somewhere inside `.git/objects`, and it might be a fun exercise to try and retrieve it without using the command itself. But, I'll leave that for you to try while we move on to finishing up branch management in the next chapter.

15. Switching branches

Our checkout command allows us to reload the contents of older commits into our workspace, and go back to the latest commit by checking out the name of a branch. To complete its functionality, we need to make a few changes to how `.git/HEAD` is managed.

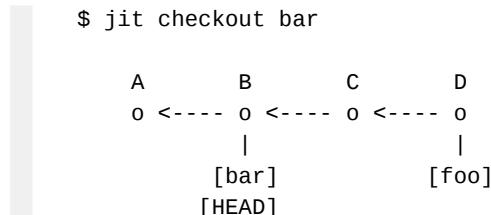
In the example at the start of the previous chapter, we had a chain of commits with two branch pointers, `foo` and `bar`, where `foo` points at the last commit in the chain, and `bar` points to that commit's grandparent. `HEAD` also points at the final commit, and that's what's loaded into the workspace.

Figure 15.1. Two branch pointers into a commit history



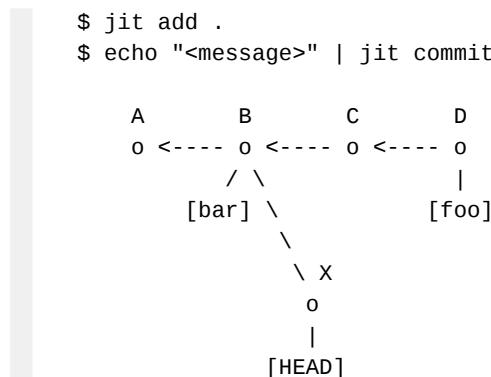
Right now, running `checkout bar` will move `HEAD` so that it points at the same commit as `bar`.

Figure 15.2. Moving HEAD to point at an older commit



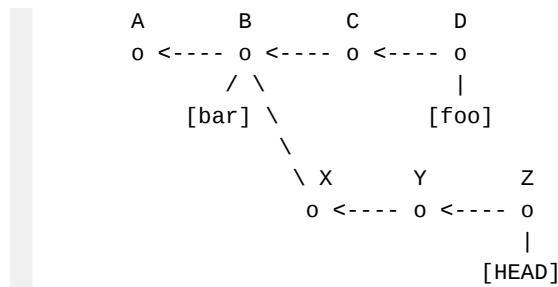
With the repository in this state, we could make some changes in the workspace and commit them, and that would create another commit with commit `B` as its parent, and move `HEAD` to point at it. However, the branch pointer `bar` remains where it is.

Figure 15.3. Committing moves the HEAD pointer



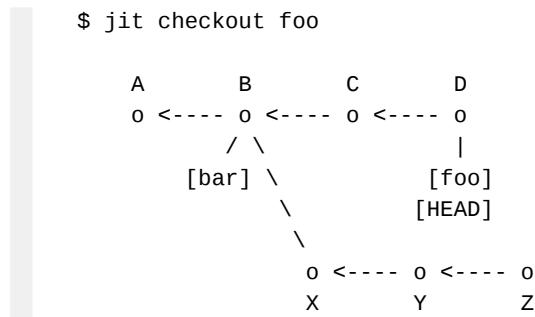
Making further commits continues the line of history from `X` onwards.

Figure 15.4. Committing a chain of commits



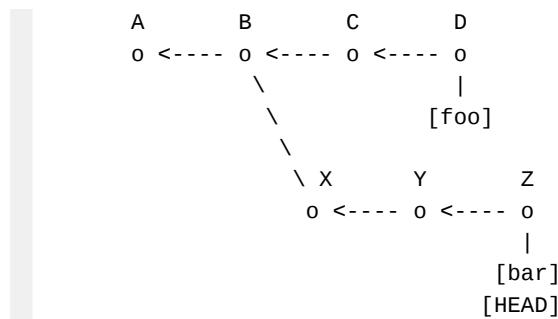
As long as we stay on this line of history, everything works fine. But what if we switch back to the `foo` branch by running `checkout foo`?

Figure 15.5. Moving `HEAD` to another branch



There is now no pointer that refers to commit `Z`, so it's effectively lost, unless we write its commit ID down somewhere. This reveals the final requirement for branch pointers: they should follow along with `HEAD` when we make commits, so that they track the latest commit added to their history. If this were the case, then we'd end up with this state after making commit `Z`:

Figure 15.6. Branch pointer following a commit chain



We can now switch back over to `foo` and return to our branch by running `checkout bar`. Allowing the branch pointers to move along with `HEAD` as we make commits means we can create multiple lines of history to explore different options and allow a group of people to work concurrently on separate changes to the project. The branch pointers give us memorable names to refer to the ends of different chains in the commit graph, so we don't need to remember long commit IDs when we want to switch between versions and manipulate the history.

Note how in the above history, `foo` and `bar` refer to different commits. But, `foo~2` and `bar~3` both refer to the same commit: commit `B`. The branches have a *common ancestor*, the latest

commit that appears in both of their histories, and this means commit *B* can be referred to by multiple *names*: its object ID (or an abbreviation), or any expression that resolves to *B* by following the chain of commit parents.

15.1. Symbolic references

In our implementation so far, `.git/HEAD` and `.git/refs/heads/<name>` always contain a commit ID. When we run `checkout bar`, we copy the contents of `.git/refs/heads/bar` into `.git/HEAD`, and we don't store any representation of `bar` being the *current branch*. When we make a commit, we have no way of telling that the `bar` pointer should move along with `HEAD`, other than that they happen to contain the same ID. That on its own is not a strong enough signal that `bar` should be moved; we may have created multiple branch pointers at the same commit and we don't necessarily want all of them to track every commit from that point.

Git solves this problem by using *symbolic references*, or *symlinks* for short. As we saw in Section 13.1, “Examining the branch command”, `.git/HEAD` in a brand new repository contains this text:

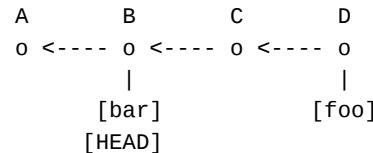
```
ref: refs/heads/master
```

We say that `.git/HEAD` contains a symref to `refs/heads/master`, which is the file containing the `master` branch pointer. That file may not exist yet if we've not made any commits, but when we do, the commit's ID will be written to `.git/refs/heads/master` and `.git/HEAD` will be unchanged.

When we want to get the ID of the current commit, or update it, the symref says we should consult the file `.git/refs/heads/master` rather than `.git/HEAD`. If the file `.git/refs/heads/master` itself contains a string beginning with `ref:` rather than an object ID, that will lead us to another file, and so on. We continue to recursively follow these references until we find a file that contains a commit ID.

Recall this example state from earlier:

Figure 15.7. Two branch pointers into a commit history



We can recreate this situation by running the following commands:

```
$ for msg in A B C D ; do
  echo "$msg" > file.txt
  git add .
  git commit --message "$msg"
done

$ git branch foo @
$ git branch bar @~2
```

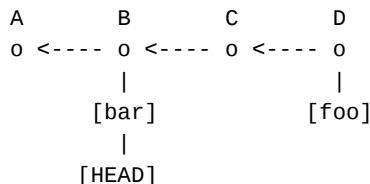
```
$ git checkout bar  
$ git branch -D master
```

We can check the state of the history by using `git log`, and then look at the contents of various references in `.git`.

```
$ git log --graph --oneline foo bar  
* 37896d3 (foo) D  
* 0bb7df5 C  
* 59741f0 (HEAD -> bar) B  
* c957c02 A  
  
$ cat .git/HEAD  
ref: refs/heads/bar  
  
$ cat .git/refs/heads/foo  
37896d38c419f6ed1ff88b32d04189fb8b9cf722  
  
$ cat .git/refs/heads/bar  
59741f0d8365b799026d18975921a201264d052e
```

The log output shows a single linear history in which `foo` points at commit `D`, and `HEAD` points to `bar`, which points at commit `B`. Checking the files in `.git`, we see that `.git/HEAD` contains `ref: refs/heads/bar`, and `.git/refs/heads/bar` contains the ID of commit `B`. So `HEAD` is strictly a reference to the branch pointer `bar`, rather than a reference directly to commit `B`. We'll indicate this by a distinct line connecting `HEAD` to `bar`.

Figure 15.8. `HEAD` referring to a branch pointer



15.1.1. Tracking branch pointers

Now when we make a new commit, the symref in `HEAD` tells us to move `bar`, and since `HEAD` is a reference to `bar`, it follows along automatically. Let's confirm this by making a few commits on the `bar` branch and then checking the state again.

```
$ for msg in X Y Z ; do  
    echo "$msg" > file.txt  
    git commit --all --message "$msg"  
done  
  
$ git log --graph --oneline bar foo  
* c1cd8d9 (HEAD -> bar) Z  
* 6d86240 Y  
* 743019f X  
| * 37896d3 (foo) D  
| * 0bb7df5 C  
|/  
* 59741f0 B
```

```
* c957c02 A

$ cat .git/HEAD
ref: refs/heads/bar

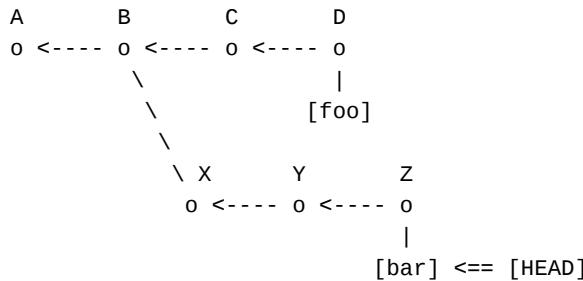
$ cat .git/refs/heads/foo
37896d38c419f6ed1ff88b32d04189fb8b9cf722

$ cat .git/refs/heads/bar
c1cd8d9d26a7827bbdc930c47f59a3cdcac1e88e
```

The history now contains a fork: commits *C* and *X* both have commit *B* as their parent, and a distinct chain of commits follows each one. We can see that `foo` still points at commit *D*, while `bar` (and therefore `HEAD`) now points to commit *Z*. The two branches have a common ancestor in commit *B*.

In the following diagrams, a double arrow `<==` represents a symbolic reference, a vertical line `|` represents a branch pointer to a commit ID, and a single arrow `<--` or diagonal \ represents a commit's parent pointer. The state of the history after the above commits is now:

Figure 15.9. Branch tracking newly created commits



If we checkout the branch `foo`, that means we change `.git/HEAD` to `ref: refs/heads/foo`, and `bar` remains where it is.

Figure 15.10. Switching `HEAD` to point at a different branch

```
$ git checkout foo

    graph TD
        A((A)) --> B((B))
        B --> C((C))
        B --> D((D))
        B --> X((X))
        B --> Y((Y))
        B --> Z((Z))
        C --> D
        X --> Z
        D --> Z
        foo["[foo]"] <==> HEAD["[HEAD]"]
        bar["[bar]"]
```

15.1.2. Detached HEAD

It is still possible for `HEAD` to contain a commit ID rather than a symref, so that it points directly at a commit rather than at another branch pointer. Git refers to such a state as *detached HEAD*, and in this situation making a commit will move `HEAD` but not any branch pointer. For example,

if we checkout `foo^`, then `HEAD` points directly at the parent of `foo`, and `foo` and `bar` remain where they were.

```
$ git checkout foo^

$ git log --graph --oneline bar foo
* c1cd8d9 (bar) Z
* 6d86240 Y
* 743019f X
| * 37896d3 (foo) D
| * 0bb7df5 (HEAD) C
|/
* 59741f0 B
* c957c02 A

$ cat .git/HEAD
0bb7df555072bdb8fbf3f3b209fee91b92f210d
```

Or, displayed using our usual representation:

Figure 15.11. Detached HEAD, pointing directly to a commit

```
$ git checkout foo^

A      B      C      D
o <---- o <---- o <---- o
     \    |    |
     \  [HEAD]  [foo]
      \
     \ X      Y      Z
      o <---- o <---- o
                       |
                     [bar]
```

We can still make commits while in detached `HEAD` state; `HEAD` will track the history but no other branch pointer will move.

```
$ for msg in P Q ; do
    echo "$msg" > file.txt
    git commit --all --message "$msg"
done

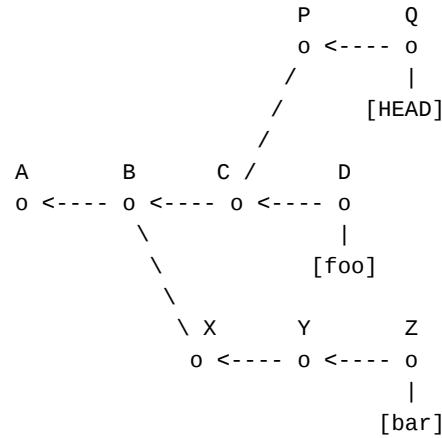
$ git log --graph --oneline @ foo
* ae70c8a (HEAD) Q
* 8f0068f P
| * 37896d3 (foo) D
|/
* 0bb7df5 C
* 59741f0 B
* c957c02 A

$ cat .git/HEAD
ae70c8aaa3f80aa0104123d940e14c91ad5ced01

$ cat .git/refs/heads/foo
37896d38c419f6ed1ff88b32d04189fb8b9cf722
```

This log tells us that `HEAD` points directly at commit `Q`, `foo` still points at commit `D`, and these commits have a common ancestor at commit `C`.

Figure 15.12. Branch formed while in detached HEAD mode



15.1.3. Retaining detached histories

If we were to checkout branch `foo` now, there would be no references to commit `Q` remaining, and it would effectively be lost. If we want to keep this chain of commits, we can use the `branch` command to create a new pointer to our current `HEAD` position. Running `branch qux` creates a new pointer to the commit `HEAD` points at, but `HEAD` still points directly at a commit, rather than at the new branch pointer.

```
$ git branch qux

$ git log --graph --oneline qux foo
* ae70c8a (HEAD, qux) Q
* 8f0068f P
| * 37896d3 (foo) D
|/
* 0bb7df5 C
* 59741f0 B
* c957c02 A

$ cat .git/HEAD
ae70c8aaa3f80aa0104123d940e14c91ad5ced01

$ cat .git/refs/heads/qux
ae70c8aaa3f80aa0104123d940e14c91ad5ced01
```

The above listing shows that `HEAD` and `qux` both independently point at commit `Q`. If we want to track this branch and have the pointer `qux` move along with our commits, we need to check that branch out so that `HEAD` becomes a symref to it.

```
$ git checkout qux

$ git log --graph --oneline qux foo
* ae70c8a (HEAD -> qux) Q
* 8f0068f P
| * 37896d3 (foo) D
|/
* 0bb7df5 C
* 59741f0 B
* c957c02 A

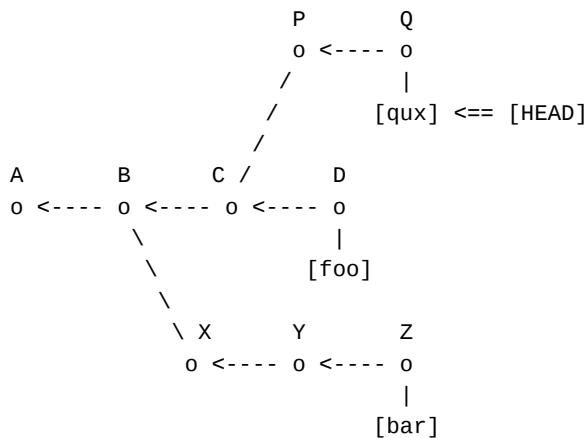
$ cat .git/HEAD
```

```
ref: refs/heads/qux

$ cat .git/refs/heads/qux
ae70c8aaa3f80aa0104123d940e14c91ad5ced01
```

We are now in the following state, where `HEAD` is attached to `qux`, so if we continue to make commits on top of `Q`, the `qux` pointer will follow along. We're then free to switch to any other branch without losing a reference to the tip of the `qux` branch.

Figure 15.13. Reattaching `HEAD` to a new branch pointer



15.2. Linking `HEAD` on checkout

The first step in supporting symrefs in our implementation is to make the `checkout` command write a symref to `.git/HEAD` if the revision we've asked for is the name of a branch. The `Command::Checkout` class currently makes this call to change the contents of `HEAD`.

```
# lib/command/checkout.rb

repo.refs.update_head(@target_oid)
```

We'll replace this call with a new method, `Refs#set_head`, which takes two arguments: the requested target, and an object ID. For example, if we ran `checkout foo`, and the `Revision` class resolved the expression `foo` to the ID `c957c02`, then we'd call `set_head("foo", "c957c02")`.

```
# lib/command/checkout.rb

repo.refs.set_head(@target, @target_oid)
```

The `Refs#set_head` method inspects its first argument to see if it's a branch name, that is, whether a file exists with the name `.git/refs/heads/<name>`. If it does, we write a symref to `.git/HEAD` containing a pointer to this branch file. Otherwise, we store the object ID we were given as a normal reference.

```
# lib/refs.rb

def set_head(revision, oid)
  head = @pathname.join(HEAD)
  path = @heads_path.join(revision)

  if File.file?(path)
    relative = path.relative_path_from(@pathname)
```

```
    update_ref_file(head, "ref: #{ relative }")
else
  update_ref_file(head, oid)
end
end
```

15.2.1. Reading symbolic references

To keep all our existing code working, we now need to add functionality for reading symrefs. The `Refs#read_head` and `Refs#read_ref` methods should return an object ID, but their current implementation just returns the contents of the requested file, so they might return a string like `"ref: refs/heads/master"`. We need to modify these methods so that instead of returning the text of a symref, they follow that symref and return an object ID.

These methods currently rely on a method called `read_ref_file`, which takes a pathname and returns the contents of that file as a string. To work towards replacing this method, let's invent one that returns an object of a different type depending on whether the file contains a symref or an object ID.

```
# lib/refs.rb

SymRef = Struct.new(:path)
Ref     = Struct.new(:oid)

SYMREF = /^ref: (.+)/

def read_oid_or_symref(path)
  data  = File.read(path).strip
  match = SYMREF.match(data)

  match ? SymRef.new(match[1]) : Ref.new(data)
rescue Errno::ENOENT
  nil
end
```

`Refs#read_oid_or_symref` uses a regular expression to detect a symref. If the contents of the read file begins with `ref:`, we return a `SymRef` value holding the path the symref points to. Otherwise, we return a `Ref` whose object ID is the contents of the file. If the file does not exist, we return `nil`.

We can use this method to build one that takes a path and always returns an object ID, by recursively following symrefs until it finds a normal reference. The `read_symref` method calls `read_oid_or_symref`, and if the result is a `SymRef` it recurses to read the path the symref points at. Otherwise, if the result is a `Ref`, it just returns the ref's object ID.

```
# lib/refs.rb

def read_symref(path)
  ref = read_oid_or_symref(path)

  case ref
  when SymRef then read_symref(@pathname.join(ref.path))
  when Ref      then ref.oid
  end
end
```

We can now replace the calls to `read_ref_file` with calls to `read_symref`, and the public `Refs` methods for reading references will now return an object ID if `.git/HEAD` or any other ref file contains a symref.

```
# lib/refs.rb

def read_head
  read_symref(@pathname.join(HEAD))
end

def read_ref(name)
  path = path_for_name(name)
  path ? read_symref(path) : nil
end
```

15.3. Printing checkout results

The checkout command is almost complete: it makes all the required changes to the workspace, index and refs so that the repository is in the correct state. The only thing left to do is to print some output about what it did, so the user knows what's happened.

We're already printing output in the case of errors, particularly errors concerning an invalid revision or local changes conflicting with the tree diff. All the output we're going to produce now relates to the checkout command having succeeded, there's just a few different outcomes to deal with. The printed output is determined by the current object ID before and after the checkout, and whether the repository had a detached `HEAD` in each state.

If the repository has a detached `HEAD` before the migration is executed, that means `HEAD` is not a symref to a branch pointer but a direct reference to a commit. If we move `HEAD`, we might be destroying the only reference to that commit, so it would be a good idea to tell the user what its ID is in case they need to find it again later. In this situation, Git will print the commit's abbreviated ID and the title line of its message.

```
Previous HEAD position was e83c516 Initial revision of "git"
```

Next, if `HEAD` was a symref but the checkout replaces it with a direct commit reference, Git prints the name of the revision you've requested, followed by some information about detached `HEAD` state.

```
Note: checking out 'v2.15.0'.
```

```
You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.
```

```
If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:
```

```
git checkout -b <new-branch-name>
```

Finally, if `HEAD` is a symref to a branch after the checkout, Git will print the name of the branch we've switched to.

```
Switched to branch 'master'
```

But, if HEAD is detached and points directly at a commit, Git prints the abbreviated ID and title line of the commit that HEAD now points at.

```
HEAD is now at 3443546 Use a *real* built-in diff generator
```

Printing these messages requires gathering a little extra information while the checkout command runs, and then adding some extra methods to the `Command::Checkout` class to display the output. Here's the full content of `Command::Checkout#run` incorporating these changes; we'll go through the other new methods required below.

```
# lib/command/checkout.rb

def run
  @target = @args[0]

  @current_ref = repo.refs.current_ref
  @current_oid = @current_ref.read_oid

  revision    = Revision.new(repo, @target)
  @target_oid = revision.resolve(Revision::COMMIT)

  repo.index.load_for_update

  tree_diff = repo.database.tree_diff(@current_oid, @target_oid)
  migration = repo.migration(tree_diff)
  migration.apply_changes

  repo.index.write_updates
  repo.refs.set_head(@target, @target_oid)
  @new_ref = repo.refs.current_ref

  print_previous_head
  print_detachment_notice
  print_new_head

  exit 0

rescue Repository::Migration::Conflict
  handle_migration_conflict(migration)

rescue Revision::InvalidObject => error
  handle_invalid_object(revision, error)
end
```

The first adjustment we've made is to how the `@current_oid` value is computed:

```
# lib/command/checkout.rb

@current_ref = repo.refs.current_ref
@current_oid = @current_ref.read_oid
```

This uses a new method `Refs#current_ref` to determine which ref file contains the current commit ID, or to put it another way, which file would be updated when we make a commit. For example, if `.git/HEAD` contains a commit ID, then the current ref would be `HEAD`, but if it instead contains a symref to the `master` branch then the current ref is `refs/heads/master`. `Refs#current_ref` is similar to `Refs#read_symref`, only instead of recursing until it finds an object ID, it stops one step short and returns a `SymRef` containing the path that contains the ID.

```
# lib/refs.rb

def current_ref(source = HEAD)
  ref = read_oid_or_symref(@pathname.join(source))

  case ref
  when SymRef then current_ref(ref.path)
  when Ref, nil then SymRef.new(self, source)
  end
end
```

Having a reference to the current ref lets us tell whether HEAD was detached before the checkout; in a detached HEAD state, the current ref is HEAD rather than a branch pointer in refs/heads.

We get the current object ID by following the current ref one step further to read its object ID, using a new method called `read_oid`. This is implemented on the `Refs::SymRef` and `Refs::Ref` classes; a `Ref` can just return the object ID it holds, but a `SymRef` needs to ask the `Refs` class to read the path it contains.

```
# lib/refs.rb

SymRef = Struct.new(:refs, :path) do
  def read_oid
    refs.read_ref(path)
  end
end

Ref = Struct.new(:oid) do
  def read_oid
    oid
  end
end
```

Notice that the `SymRef` struct has gained another parameter: `refs`. Because it needs to call back to the `Refs#read_ref` method, it needs a reference to the `Refs` object that created it. The `Refs#current_ref` method above passes `self` into `SymRef.new`. We have one other existing place where we're calling `SymRef.new`: the `Refs#read_oid_or_symref` method. Let's amend that to pass the `Refs` object into the `SymRef` object.

```
# lib/refs.rb

def read_oid_or_symref(path)
  # ...
  match ? SymRef.new(self, match[1]) : Ref.new(data)
  # ...
end
```

The only other change to the `checkout` command before the addition of the new printing methods is that after calling `Refs#set_head` we call `Refs#current_ref` again to find out what the current ref is after HEAD has been updated, and we store this in `@new_ref` so we can compare it to `@current_ref` while printing the results.

The first step in printing the output is to display the previous HEAD position, if we're in detached HEAD state before the checkout, and the checkout changes the current commit ID.

```
# lib/command/checkout.rb
```

```
def print_previous_head
  if @current_ref.head? and @current_oid != @target_oid
    print_head_position("Previous HEAD position was", @current_oid)
  end
end
```

Calling `SymRef#head?` tells us whether the current ref represents a detached HEAD state; in this state the current ref is HEAD itself.

```
# lib/ref.rb

SymRef = Struct.new(:refs, :path) do
  # ...

  def head?
    path == HEAD
  end
end
```

Command `:Checkout#print_head_position` is a helper that prints some details about a given commit, prefixed with the given message. It displays the message, followed by the abbreviated commit ID, and the first line of its message. All output from the checkout command is printed to standard error.

```
# lib/command/checkout.rb

def print_head_position(message, oid)
  commit = repo.database.load(oid)
  short = repo.database.short_oid(commit.oid)

  @stderr.puts "#{message} #{short} #{commit.title_line}"
end
```

The next step is to print a message about entering detached HEAD state, if HEAD has become detached. This means that the current ref was not HEAD before the checkout, but it is HEAD afterward.

```
# lib/command/checkout.rb

DETACHED_HEAD_MESSAGE = <<~MSG
  You are in 'detached HEAD' state. You can look around, make experimental
  changes and commit them, and you can discard any commits you make in this
  state without impacting any branches by performing another checkout.

  If you want to create a new branch to retain commits you create, you may
  do so (now or later) by using the branch command. Example:

    jit branch <new-branch-name>
MSG

def print_detachment_notice
  return unless @new_ref.head? and not @current_ref.head?

  @stderr.puts "Note: checking out '#{@target}'."
  @stderr.puts ""
  @stderr.puts DETACHED_HEAD_MESSAGE
```

```
    @stderr.puts ""
end
```

Finally, we print some information about the new state. If `@new_ref` (the current ref after the checkout is executed) is `HEAD`, then we're in detached `HEAD` state and we use `print_head_position` to display the new `HEAD` information. Otherwise, we've checked out a branch — either the new ref is the same as the old one, and we print `Already on <branch>`, or we've switched branches so we display `Switched to branch <branch>`.

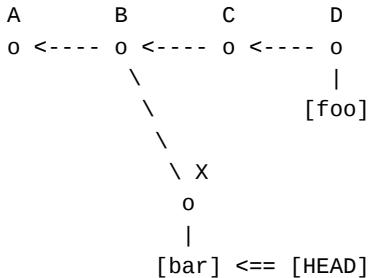
```
# lib/command/checkout.rb

def print_new_head
  if @new_ref.head?
    print_head_position("HEAD is now at", @target_oid)
  elsif @new_ref == @current_ref
    @stderr.puts "Already on '#{ @target }'"
  else
    @stderr.puts "Switched to branch '#{ @target }'"
  end
end
```

15.4. Updating HEAD on commit

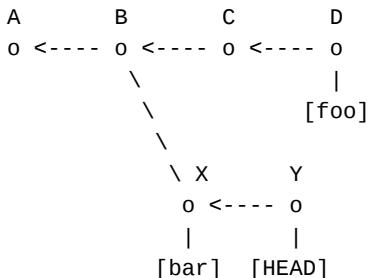
We're now very close to having branches that work properly. The only missing feature is that when `.git/HEAD` is a symref to a branch pointer, the `commit` command should update the target of the symref, rather than writing to `.git/HEAD` itself. Currently, if we're in this state, where `HEAD` is a symref to `refs/heads/bar`:

Figure 15.14. `HEAD` attached to a branch



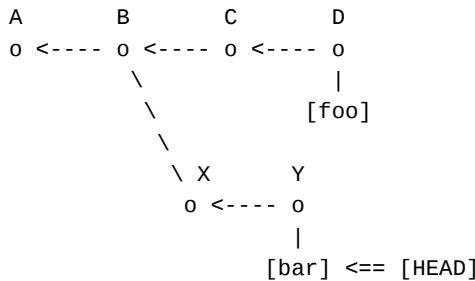
Making another commit will write the ID of that commit directly into `.git/HEAD`, leaving us in this state where `HEAD` points at the new commit but `bar` remains unmoved:

Figure 15.15. Detaching `HEAD` by making a commit



Instead, we want to update the `refs/heads/bar` pointer, so that the branch pointer tracks the history as we built it:

Figure 15.16. Branch pointer following a new commit



The `Command::Commit` class calls `Refs#update_head` to store the new commit ID. Just as we updated `Refs#read_head` to follow symrefs, we can update `Refs#update_head` to do the same. Rather than having it call `update_ref_file` internally, let's make it call a new method `update_symref`.

```
# lib/refs.rb

def update_head(oid)
  update_symref(@pathname.join(HEAD), oid)
end
```

`Refs#update_symref` works similarly to its `read` counterpart: it recursively reads symrefs until it finds a ref file that contains an object ID (or until it finds an absent file). Once it finds the ref at the end of the chain, it writes the new commit ID to that ref.

```
# lib/refs.rb

def update_symref(path, oid)
  lockfile = Lockfile.new(path)
  lockfile.hold_for_update

  ref = read_oid_or_symref(path)
  return write_lockfile(lockfile, oid) unless ref.is_a?(SymRef)

  begin
    update_symref(@pathname.join(ref.path), oid)
  ensure
    lockfile.rollback
  end
end

def write_lockfile(lockfile, oid)
  lockfile.write(oid)
  lockfile.write("\n")
  lockfile.commit
end
```

This is where the lockfiles used to update refs¹ start to really become important. Because we're reading a file before deciding whether to write to it, we need to make sure no other process changes the files we're looking at in between the read and the write. So, before reading the file, we acquire a lock against it. If we find that the file does not contain a symref, then we immediately write and commit the lockfile and return. Otherwise, we keep a hold of the lock and recurse to update the symref's target. After returning from the recursive call, we release

¹Section 4.2.1, “Safely updating .git/HEAD”

the lock we were holding, in an ensure block so that the locks we're holding are released even if the recursive call fails, for example because some other process is holding a lock on another file we need.

For example, say that `.git/HEAD` contains a symref to `refs/heads/master`. Calling `Refs#update_head` will first create the lockfile `.git/HEAD.lock`. If this succeeds, we read `.git/HEAD` and get back the string "ref: refs/heads/master". We follow this reference and open another lockfile: `.git/refs/heads/master.lock`. We read `.git/refs/heads/master` and find that it contains an object ID, so now we stop recursing. We write the new ID to `.git/refs/heads/master.lock`, move it to `.git/refs/heads/master`, and then pop back up the stack to the original `update_symref` call. We now delete the file `.git/HEAD.lock`, and the whole procedure is complete. We safely update the reference at the end of a symref chain, locking every file we visit along the way. If any of these files are already locked by another process, we release any locks we'd acquired and exit.

Finally, note that the `write_lockfile` method is code extracted from the original implementation of `update_ref_file`, which now looks like this:

```
# lib/refs.rb

def update_ref_file(path, oid)
  lockfile = Lockfile.new(path)

  lockfile.hold_for_update
  write_lockfile(lockfile, oid)

  rescue Lockfile::MissingParent
    FileUtils.mkdir_p(path.dirname)
    retry
end
```

We've now established two different behaviours for updating `HEAD`. The `checkout` command uses `Refs#set_head` and only writes to the `.git/HEAD` file, putting either a commit ID or a symref there. It does not modify any branch pointers. `commit`, on the other hand, uses `Refs#update_head` and follows the symref chain starting at `.git/HEAD`, writing a commit ID to the file at the end of the chain.

15.4.1. The master branch

Now that we can read and write symrefs, it is safe to initialise new repositories with `HEAD` being a symref to a branch. When Git creates a new repository, it puts `ref: refs/heads/master` in `.git/HEAD`, so the first commit will be tracked by the `master` branch. `master` is not special, it's just a branch pointer like any other, it's just the name of the branch that Git initialises a repository with.

We can mirror this functionality by making a small addition to `Command#run`: we'll have it create the `.git/refs/heads` directory, and write a symref to `.git/HEAD`.

```
# lib/command/init.rb

DEFAULT_BRANCH = "master"
```

```
def run
  path = @args.fetch(0, @dir)

  root_path = expanded.pathname(path)
  git_path = root_path.join(".git")

  ["objects", "refs/heads"].each do |dir|
    begin
      FileUtils.mkdir_p(git_path.join(dir))
    rescue Errno::EACCES => error
      @stderr.puts "fatal: #{error.message}"
      exit 1
    end
  end

  refs = Refs.new(git_path)
  path = File.join("refs", "heads", DEFAULT_BRANCH)
  refs.update_head("ref: #{path}")

  puts "Initialized empty Jit repository in #{git_path}"
  exit 0
end
```

That's all there is to it. New repositories will write their first commit ID to .git/refs/heads/master rather than .git/HEAD, and from there they can create and switch branches however they choose.

15.5. Branch management

To finish up our treatment of branches for now, we'll add two more abilities to the branch command: we'll let it list all the branches in the repository, and delete branches by name.

Listing the existing branches is done by running branch with no arguments. In the repository we studied in Section 15.1.3, "Retaining detached histories", this produces:

```
$ git branch
  bar
  foo
* qux
```

The asterisk * indicates which branch is current; that is if we make a new commit right now, the qux branch pointer is the one that will move. The command can display more information if we add the --verbose flag, or -v for short.

```
$ git branch --verbose
  bar c1cd8d9 Z
  foo 37896d3 D
* qux ae70c8a Q
```

With this option, it displays the abbreviated ID and the title line of the message of the commit that each branch points at.

Deleting branches uses a couple of different flags. The --delete flag, -d for short, means that instead of creating the named branch(es), the branch command will delete them. This only

deletes the pointers in `.git/refs/heads`, not the commits that the branches point at. However, the branches will only be deleted if they have been merged, that is, if the commits they point at are ancestors of your current `HEAD`. The `--force` flag, or `-f`, causes the branch to be deleted even if it has not been merged. The flag `-D` is a shorthand for `--delete --force`.

```
$ git branch -D foo bar
Deleted branch foo (was 37896d3).
Deleted branch bar (was c1cd8d9).
```

Any number of branches can be listed to be deleted, and the output of the command is a summary of the commits the deleted branches were pointing at. The commits still exist in `.git/objects` so you can retrieve a commit even if you've deleted all the branches that were pointing at it, as long as you can remember its ID.

15.5.1. Parsing command-line options

We've used command-line options before, for example the `status --porcelain` and `diff --cached` commands. When implementing those, we just checked whether the command's first argument was equal to the flag in question. But now, our needs for recognising options given on the command-line are becoming more complicated. For example, getting verbose information when listing branches can be done with either `branch --verbose` or `branch -v`. When deleting branches, the branch names can appear before or after the flags, and the flags can appear in any order and use any combination of shorthands, so all these perform the same function:

```
branch --delete --force master
branch -f master --delete
branch master -f -d
branch -df master
branch -D master
branch master -D
```

This arbitrary ordering and combination of arguments makes the task of parsing the input quite tricky. The shell doesn't have any built-in concept of command-line option formatting, and all the above result in the program receiving a list of strings as input — `["master", "-f", "-d"]` or `["-df", "master"]`, for example. We'd like to be able to accept any of these inputs and produce a canonical representation of the options used, along with a list of any command-line arguments that aren't the names of flags. In other words, something like:

```
options = { :delete => true, :force => true }
argv    = ["master"]
```

Fortunately, Ruby's standard library includes a module for parsing conventional command-line options, called `OptionParser`². To recognise the `--delete` and `-f` options as well as their shorthands, we can write:

```
require "optparse"

options = {}
parser = OptionParser.new

parser.on("-d", "--delete") { options[:delete] = true }
parser.on("-f", "--force") { options[:force] = true }
```

²<https://docs.ruby-lang.org/en/2.3.0/OptionParser.html>

Let's try the parser out on some input.

```
argv = ["--delete", "--force", "master"]
parser.parse!(argv)

options # => { :delete => true, :force => true }
argv   # => ["master"]
```

OptionParser also takes care of recognising combined shorthands for us, and it allows options and other arguments to appear in any order.

```
argv = ["master", "-fd"]
parser.parse!(argv)

options # => { :force => true, :delete => true }
argv   # => ["master"]
```

We'll use OptionParser to support our ongoing input parsing needs, but first, we'll need to convert our existing uses of command-line flags to use it. In Command::Base#execute, we'll insert a call before run that invokes a new method, parse_options. The aim of this method call is to generate a new data structure called @options that contains any options set by the user, and to leave any unparsed inputs in the @args array.

```
# lib/command/base.rb

def execute
  parse_options
  catch(:exit) { run }

  if defined? @pager
    @stdout.close_write
    @pager.wait
  end
end
```

Command::Base#parse_options uses OptionParser to set up a new parser, and inside that parser we call define_options. The default implementation of define_options is an empty method, but command classes can override this to define any options that apply to them. After the parser is defined, we can call its parse! method to extract any options from @args.

```
# lib/command/base.rb

def parse_options
  @options = {}
  @parser = OptionParser.new

  define_options
  parser.parse!(@args)
end

def define_options
end
```

If the user passes things that look like options (i.e. that begin with one or two dashes) and that aren't recognised by the parser, OptionParser will raise an exception. So to make our existing Command::Status and Command::Diff classes keep working, we need to move them over to use this define_options hook.

Command::Status has an option called `--porcelain` that changes the output format. We can easily accommodate this by setting a default value for the `:format` option and letting this flag override it.

```
# lib/command/status.rb

def define_options
  @options[:format] = "long"
  @parser.on("--porcelain") { @options[:format] = "porcelain" }
end
```

Then in `Command::Status#print_results`, we use this option to determine which printing routine to call.

```
# lib/command/status.rb

def print_results
  case @options[:format]
  when "long"      then print_long_format
  when "porcelain" then print_porcelain_format
  end
end
```

Command::Diff also has a single option, `--cached`, but Git also has an alias for this option called `--staged`. OptionParser lets us support both with a simple declaration. In the case of the `diff` command, the option affects whether we compute the diff between HEAD and the index, or between the index and the workspace.

```
# lib/command/diff.rb

def define_options
  @parser.on "--cached", "--staged" do
    @options[:cached] = true
  end
end

def run
  # ...

  if @options[:cached]
    diff_head_index
  else
    diff_index_workspace
  end

  # ...
end
```

With this new parsing infrastructure in place we can move on to implementing the extra branch management features.

15.5.2. Listing branches

When we run the `branch` command with no arguments, Jit should list the existing branches in alphabetical order, with a `*` next to the one that `HEAD` is pointing at. If we add the `--verbose`

option, it should additionally print the abbreviated commit ID and title line of the message for the commit each branch is pointing at.

To begin to implement this, we'll need a method on the `Refs` class that manages the state stored in the `.git/refs` directory. A new method, `Refs#list_branches`, will return an array of `SymRef` objects representing all the branch pointers that exist in `.git/refs/heads`. It does this using a recursive helper method, `list_refs`, which takes the path to a directory and recursively finds all the references within it.

```
# lib/refs.rb

def list_branches
  list_refs(@heads_path)
end

def list_refs(dirname)
  names = Dir.entries(dirname) - [".", ".."]

  names.map { |name| dirname.join(name) }.flat_map do |path|
    if File.directory?(path)
      list_refs(path)
    else
      path = path.relative_path_from(@pathname)
      SymRef.new(self, path.to_s)
    end
  end
rescue Errno::ENOENT
  []
end
```

This method reads the list of files that exist in `.git/refs/heads` and maps them each to a `SymRef` containing the path of the branch pointer relative to the `.git` directory. For example, on the repository we studied earlier with `foo`, `bar` and `qux` branches, this method returns the following list:

```
[  
  SymRef(path="refs/heads/bar"),  
  SymRef(path="refs/heads/foo"),  
  SymRef(path="refs/heads/qux")  
]
```

Why return `SymRef` objects here rather than just a list of names like `["bar", "foo", "qux"]`? Well, this list is going to be used for various bits of internal plumbing, rather than just displayed immediately to the user, and for that plumbing it's worth being a bit more specific about what we're referring to. A bare string could refer to all sorts of locations in `.git/refs`, or could be an abbreviated object ID, whereas a `SymRef` makes it clear we're referring to a specific pointer, and not an object ID.

Where we need the short branch name, or the object ID the branch points at, we'll implement methods for that. The `Ref` and `SymRef` classes already have `read_oid`³ for getting the object ID a symref ultimately resolves to. We'll add a method `SymRef#short_name` for getting the branch

³Section 15.3, “Printing checkout results”

name without the `refs/heads` prefix; this will call through to `Refs#short_name`, which finds the first path out of `.git/refs/heads/` and `.git/` that's a prefix of the given path, and removes it.

```
# lib/refs.rb

SymRef = Struct.new(:refs, :path) do
  # ...

  def short_name
    refs.short_name(path)
  end
end

def short_name(path)
  path = @pathname.join(path)

  prefix = [@heads_path, @pathname].find do |dir|
    path.dirname.ascend.any? { |parent| parent == dir }
  end

  path.relative_path_from(prefix).to_s
end
```

So for example, calling `SymRef.new("refs/heads/master").short_name` returns the string "master". With these new `Refs` methods in place we can now implement the additions to the `branch` command. We'll implement `define_options` to detect the `--verbose` flag, and change `run` so that it calls `Command::Branch#list_branches` if there are no arguments given.

```
# lib/command/branch.rb

def define_options
  @parser.on("-v", "--verbose") { @options[:verbose] = true }
end

def run
  if @args.empty?
    list_branches
  else
    create_branch
  end

  exit 0
end
```

The `list_branches` method in `Command::Branch` does the main work of printing the branches in the repository. It begins by fetching the `current_ref` — the `SymRef` representing the current branch pointer — and then fetches a list of all the branches from `Refs#list_branches` and sorts them by their path. To support verbose mode, we calculate `max_width`, the maximum length of any of the branch short names, so we can add padding to the names and cause the commit IDs to be horizontally aligned. Then, we iterate over the branches, printing the information for each branch. If the branch's `SymRef` is the same as the current ref, we flag it with a * and green text, and if we're in verbose mode, we append some additional information about the branch.

```
# lib/command/branch.rb

def list_branches
```

```
current = repo.refs.current_ref
branches = repo.refs.list_branches.sort_by(&:path)
max_width = branches.map { |b| b.short_name.size }.max

setup_pager

branches.each do |ref|
  info = format_ref(ref, current)
  info.concat(extended_branch_info(ref, max_width))
  puts info
end
end

def format_ref(ref, current)
  if ref == current
    "* #{fmt :green, ref.short_name }"
  else
    " #{ref.short_name }"
  end
end
```

On our example repository, the branches structure contains the following data:

```
[  
  SymRef(path="refs/heads/bar"),  
  SymRef(path="refs/heads/foo"),  
  SymRef(path="refs/heads/qux")  
]
```

Say that `Refs#current_ref` returns `SymRef(path = "refs/heads/foo")`; the branch command will now print:

```
bar  
* foo  
qux
```

This is one reason for returning `SymRef` objects from `Refs#list_branches`: it makes it easy to compare each branch to the current ref.

If we're in verbose mode, we use `extended_branch_info` to build a string of extra information about the branches. Recall that the verbose output looks like this:

```
$ git branch --verbose  
bar c1cd8d9 Z  
* foo 37896d3 D  
  qux ae70c8a Q
```

When the branch names have different lengths, padding is added so that the commit IDs line up. The `extended_branch_info` method takes a `SymRef` and the `max_width` computed earlier, and builds this additional string of padding, commit ID and message, by reading the required information from the database.

```
# lib/command/branch.rb

def extended_branch_info(ref, max_width)
  return "" unless @options[:verbose]
```

```
commit = repo.database.load(ref.read_oid)
short  = repo.database.short_oid(commit.oid)
space   = " " * (max_width - ref.short_name.size)

 "#{ space } #{ short } #{ commit.title_line }"
end
```

Using `{Ref, SymRef}#read_oid` means that even if the given SymRef represents a file that itself contains a symref, the chain of references is followed until we resolve to an object ID.

15.5.3. Deleting branches

Finally, we need to be able to delete branches. Deletion uses the `--delete`, `--force`, and `-D` flags; we'll add these to our option definitions, and adjust the `run` method so that if `:delete` is set we call `delete_branches` instead of listing or creating anything.

```
# lib/command/branch.rb

def define_options
  @parser.on("-v", "--verbose") { @options[:verbose] = true }

  @parser.on("-d", "--delete") { @options[:delete] = true }
  @parser.on("-f", "--force") { @options[:force] = true }

  @parser.on "-D" do
    @options[:delete] = @options[:force] = true
  end
end

def run
  if @options[:delete]
    delete_branches
  elsif @args.empty?
    list_branches
  else
    create_branch
  end

  exit 0
end
```

The `delete_branches` method simply iterates over the given arguments and calls `delete_branch` for each one. That method in turn performs the actual branch deletion. Since we currently have no way of merging branches, it is impossible for any branch to have already been merged, and so we'll do nothing unless the `:force` option is set. Otherwise, we'll call a new method `Refs#delete_branch` that performs the backend work of deleting the branch pointer, and returns the object ID the branch was pointing at. On success it prints the deleted branch with its abbreviated commit ID, but if `Refs` raises an error indicating the branch does not exist, we print that error and exit with a non-zero status.

```
# lib/command/branch.rb

def delete_branches
  @args.each { |branch_name| delete_branch(branch_name) }
end
```

```
def delete_branch(branch_name)
  return unless @options[:force]

  oid = repo.refs.delete_branch(branch_name)
  short = repo.database.short_oid(oid)

  puts "Deleted branch #{branch_name} (was #{short})."

rescue Refs::InvalidBranch => error
  @stderr.puts "error: #{error}"
  exit 1
end
```

Whereas `Command::Branch#delete_branch` deals with the command's user interface, `Refs#delete_branch` contains the business logic that actually removes the branch. We want to report the branch's current commit ID before it is deleted, and we'd like to guarantee that the branch pointer is not changed between us reading its current value and removing it. Therefore, we should turn this into an atomic operation by taking out a lock against the file before reading its contents and deleting it.

If calling `read_symref` on the branch pointer returns nothing, that means the branch does not exist and we should raise an error. Otherwise, we can delete the pointer using `File.unlink`, delete any now-empty parent directories, and return the commit ID, ensuring that the lockfile is deleted afterwards.

```
# lib/refs.rb

def delete_branch(branch_name)
  path = @heads_path.join(branch_name)

  lockfile = Lockfile.new(path)
  lockfile.hold_for_update

  oid = read_symref(path)
  raise InvalidBranch, "branch '#{branch_name}' not found." unless oid

  File.unlink(path)
  delete_parent_directories(path)

  oid
ensure
  lockfile.rollback
end
```

The `delete_parent_directories` method cleans up any parent directories that may be left empty after deleting the branch. For example, if we have a branch named `fix/delete-branches`, that will be stored at `.git/refs/heads/fix/delete-branches`. The `File.unlink` call above removes the `delete-branches` file, but the directory `.git/refs/heads/fix` may now be empty, and so should be removed. This method removes any empty parent directories, but stops once it reaches `.git/refs/heads`, or once it reaches a non-empty directory; `Dir.rmdir` raises the `ENOTEMPTY` error in this case.

```
# lib/refs.rb

def delete_parent_directories(path)
```

```
path.dirname.ascend do |dir|
  break if dir == @heads_path
begin
  Dir.rmdir(dir)
rescue Errno::ENOTEMPTY
  break
end
end
```

We now have enough basic infrastructure in place to create, inspect and delete branches in our repository, and the branch pointers will track chains of commits as we make them. The relationship between the `Command::Branch` class and the `Refs` class shows a common design pattern, in which the command class contains user interface logic — recognising options, printing output and so on — and calls through to a back-end class to perform the actual operation.

The design of the `delete_branch` method in particular shows how user interface requirements can influence the underlying business logic. If we only want to delete the branch pointer, we could have made a simple call to `File.unlink` and we'd be done. However, the need to extract information from the branch before removing it introduces possible race conditions, requiring the introduction of a lock to make sure other process don't change the information as we're reading it.

16. Reviewing history

Over the course of the last three chapters, we've developed the branch and checkout commands that let us create a forking history in our codebase and navigate between different points in that history. This functionality allows a single developer to pursue multiple streams of work from a single starting point, and retrieve older versions of the project. And, given some way of transferring history over a network¹, it allows multiple developers to work on their own changes in parallel.

On its own, branching is of limited utility. The ability for multiple contributors to go off in their own directions without locking the files they want to change is indeed liberating, and represents a significant change in workflow over the previous generation of version control systems. However, we'd ultimately like to merge everyone's contributions together into a single version that we can release to end users.

We will examine the `merge` command in due course, but before we get there some extra tooling will be useful. There is currently no way to view the history of changes to the project. This isn't a huge problem when you're authoring the changes that go into each commit yourself, and you can review them using the `status`² and `diff`³ commands. But when you use the `merge` command, you're typically merging commits you wrote days, weeks or months ago, or changes written by someone else that you've never seen before. In this situation it would be good to have some way to review what will change if you perform the merge, and for that we need the `log` command.

Git's `log` command⁴ includes a lot of complex functionality and its documentation runs to over ten thousand words. Here we'll focus on a small subset of it that will be most useful for performing merges: we'd like to be able to see the commits whose changes will be applied to `HEAD` when we perform a merge, along with the diffs of those commits. We'd also like to view the commits that affect a particular file or directory so we can check for changes to specific parts of the codebase.

We'll build up to these features in small pieces. Beginning with a simple linear history, we'll develop a few options for varying how commits are displayed. Then, we'll move on to more complex histories and methods for filtering them to select which commits we want to see.

16.1. Linear history

Before we introduced the ability to create branches, it was only possible to produce a single linear history, in which `.git/HEAD` points at the latest commit, and each commit points to its parent, if it has one. Every commit contains a pointer to a *tree*, a complete snapshot of the project. Although we tend to think of commits as representing changes to the project, it's important to remember that they actually represent snapshots, and changes are inferred by comparing a commit's tree to that of its parent.

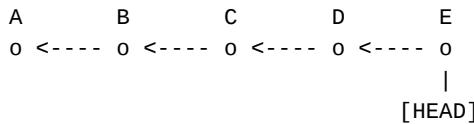
¹Chapter 28, *Fetching content*

²Chapter 9, *Status report*

³Chapter 12, *Spot the difference*

⁴<https://git-scm.com/docs/git-log>

Figure 16.1. Single chain of commits



Each dot labelled *A*, *B*, and so on represents a commit. A vertical line | represents a reference to a commit, stored in .git/HEAD or a file under .git/refs. The horizontal arrows represent the commits' parent pointers. Unless otherwise indicated, parent pointers in diagrams from here on will always point from right to left.

When we run `git log` with no arguments, we typically see a list of all the commits that lead up to the current HEAD commit, in reverse chronological order. To begin with, we'll be focussing on different ways to display commits, rather than selecting which commits to display, but it will nevertheless be useful to separate the `log` command's logic into two main pieces: iterating the selected commits, and displaying each one. Since the output could be quite long, we invoke the pager⁵ so that the user can scroll through from the beginning.

```
# lib/command/log.rb

module Command
  class Log < Base

    def run
      setup_pager
      each_commit { |commit| show_commit(commit) }
      exit 0
    end

    # ...

  end
end
```

To begin with, the `each_commit` method will only deal with the kind of history described above: a linear chain, beginning with HEAD. It will read the object ID from .git/HEAD, and then load that commit and grab its parent pointer, in a loop until it runs out of commits.

```
# lib/command/log.rb

def each_commit
  oid = repo.refs.read_head

  while oid
    commit = repo.database.load(oid)
    yield commit
    oid = commit.parent
  end
end
```

An important detail of this method is that it yields commits to the caller as it scans through the history, rather than collecting all the history into a big list. This means we print commits as soon as they're read from the database, rather than waiting until we've scanned right back to

⁵Section 12.3.3, “Invoking the pager”

the root commit before we start printing anything. On large repositories, this *streaming* effect is important in ensuring the `log` command remains usable and doesn't block for a long time while the entire history is loaded into memory.

For the remainder of this section, we'll be focussing on the `show_commit` method that displays each commit, and we'll come back to the `each_commit` logic in the following section.

16.1.1. Medium format

By default, Git displays commits in a format similar to this:

```
commit 0a065e34feca5e82a63be4b5777e8b2545e36bf1
Author: James Coglan <jcoglan.com>
Date:   Sun Feb 11 15:14:00 2018 +0000

The Myers diff algorithm

Here we implement the main algorithm described in "An O(ND) Difference
Algorithm and its Variations" [1] for finding a shortest edit script for
converting one string into another.

The particular representation of strings used here is that of lines in a
text file; each input is split into its constituent lines before being
fed into the algorithm. If the inputs are already arrays, they are left
unchanged.

Calling `Diff.diff(a, b)` returns an array of `Diff::Edit` objects
describing the changes between the two versions, which can be
immediately printed to produce a diff representation in the terminal.

[1]: http://www.xmailserver.org/diff2.pdf
```

This format is what Git refers to as `medium`. The `--pretty` or `--format` option can be used to select from many different output formats, but `medium` is the default.

Producing this output format requires only a small amount of code in `Command::Log#show_commit` to print various details of the commit. It prints a blank line before all but the first commit—the first call to `blank_line` does nothing—and then prints the commit ID in yellow (SGR code 33), the author's name and email, and the date in a readable format. Finally it prints each line of the message, indented by four spaces.

```
# lib/command/log.rb

def blank_line
  puts "" if defined? @blank_line
  @blank_line = true
end

def show_commit(commit)
  author = commit.author

  blank_line
  puts fmt(:yellow, "commit #{ commit.oid }")
  puts "Author: #{ author.name } <#{ author.email }>"
  puts "Date:  #{ author.readable_time }"
  blank_line
```

```
commit.message.each_line { |line| puts "    #{ line }" }
end
```

The `Database::Author#readable_time` time method is another use of `Time#strftime` that generates human-readable output, including English month and day names and a colon-separated 24-hour clock time.

```
# lib/database/author.rb

def readable_time
  time.strftime("%a %b %-d %H:%M:%S %Y %z")
end
```

The only other addition we need is to expose the `message` property in the `Database::Commit` class so it can be read.

```
# lib/database/commit.rb

attr_reader :message
```

16.1.2. Abbreviated commit IDs

Among the many formatting options provided by Git's `log` command, one of the simplest is the `--abbrev-commit` option. This means that, instead of printing the full 40-digit commit ID, Git will print an abbreviated version, which makes some other output options more usable.

Ruby's `OptionParser` class allows a single declaration for both the positive and negative forms of an option, for example `--[no-]abbrev-commit`. The default value of the `abbrev` option is `:auto`, which will let us set this option from other command-line flags if it's not already been explicitly set.

```
# lib/command/log.rb

def define_options
  @options[:abbrev] = :auto

  @parser.on "--[no-]abbrev-commit" do |value|
    @options[:abbrev] = value
  end
end
```

To handle this option, we'll introduce a new method `Command::Log#abbrev` which takes a commit and returns its ID in the right format.

```
# lib/command/log.rb

def show_commit(commit)
  # ...
  puts fmt(:yellow, "commit #{ abbrev(commit) }")
  # ...
end

def abbrev(commit)
  if @options[:abbrev] == true
    repo.database.short_oid(commit.oid)
```

```
    else
      commit.oid
    end
  end
```

16.1.3. One-line format

The `--abbrev-commit` option works particularly well with the `--pretty=oneline` option, in fact there's a single flag called `--oneline` that is equivalent to both of these. Let's add declarations for the `--pretty--format` and `--oneline` flags in `Command::Log#define_options`; notice how `--oneline` only sets `:abbrev` if its current value is `:auto`, so `--oneline` does not override the effect of passing `--no-abbrev-commit`.

```
# lib/command/log.rb

  @options[:abbrev] = :auto
  @options[:format] = "medium"

  @parser.on "--pretty=<format>", "--format=<format>" do |format|
    @options[:format] = format
  end

  @parser.on "--oneline" do
    @options[:abbrev] = true if @options[:abbrev] == :auto
    @options[:format] = "oneline"
  end
```

With these options defined, we can replace the existing `show_commit` method with some logic that dispatches to two different display methods based on the value of the `:format` option. The `oneline` format just displays the possibly-abbreviated commit ID and its title line, making it useful for quickly skimming through a list of changes.

```
# lib/command/log.rb

def show_commit(commit)
  case @options[:format]
  when "medium" then show_commit_medium(commit)
  when "oneline" then show_commit_oneline(commit)
  end
end

def show_commit_medium(commit)
  # existing behaviour
end

def show_commit_oneline(commit)
  puts "#{ fmt :yellow, abbrev(commit) }#{ commit.title_line }"
end
```

These small additions mean Jit can now display a brief summary of the commit history, for example:

```
$ jit log --oneline

ae6e49a One-line log format
1e781bc Abbreviate commit IDs in logs
```

```
5a2e7b8 Log a linear history in medium format
8ffdd0f Show the current branch in `status`
c10e15f Delete branches using `--delete` and/or `--force`
# etc.
```

16.1.4. Branch decoration

We now come to some features of `log` that are a bit more involved than a simple output format change, and require gathering more information from the object database. The first of these is the `--decorate` flag, which tells `log` to label each commit with the names of any branches or other refs that point at it. If the branch is also the current ref, its name is prefixed with `HEAD`. For example, if we created a branch called `topic` pointing at `master~3`, the above log would look like this:

```
ae6e49a (HEAD -> master) One-line log format
1e781bc Abbreviate commit IDs in logs
5a2e7b8 Log a linear history in medium format
8ffdd0f (topic) Show the current branch in `status`
c10e15f Delete branches using `--delete` and/or `--force`
```

The `--decorate` flag takes four values: `short` produces the above decoration style, and `full` prints the full name of each ref, for example `refs/heads/master`. `auto` preserves its default behaviour, which is to print `short` format decoration if the output is going to a terminal (i.e. a TTY), and nothing otherwise. Finally, `no` (or `--no-decorate`) turns decoration off regardless of the output destination. The value of this flag is optional; `--decorate` is equivalent to `--decorate=short`.

To support this functionality, we'll need a lookup table of the refs that point at each commit. For example, for the above log listing we'd need a structure like the following, which maps each commit ID (abbreviated here) to a list of `SymRef` objects representing the references that point at it.

```
{
  "ae6e49a..." => [SymRef(path="HEAD"), SymRef(path="refs/heads/master")],
  "8ffdd0f..." => [SymRef(path="refs/heads/topic")]
}
```

Combined with the knowledge that `SymRef(path="refs/heads/master")` is the current ref, this information is enough to generate the above output.

Let's add a method to the `Refs` class that generates this information, and call it `reverse_refs`. It begins by making a hash where the default value for missing keys is an empty array. It then constructs a `SymRef` for `HEAD`, reads the object ID it points at, and adds `HEAD` to the list of refs for that ID. Then it does the same for the branches; calling `list_all_refs` to get a list of `SymRef` objects representing all the branch pointers in `.git/refs`, we iterate over them and get the commit ID for each one. We push the current reference into the list of refs for its commit ID. The returned table has the structure illustrated above.

```
# lib/refs.rb

def reverse_refs
  table = Hash.new { |hash, key| hash[key] = [] }
```

```

list_all_refs.each do |ref|
  oid = ref.read_oid
  table[oid].push(ref) if oid
end

table
end

def list_all_refs
  [SymRef.new(self, HEAD)] + list_refs(@refs_path)
end

```

We can now begin modifying the log command's options to support the --[no-]decorate flag:

```

# lib/command/log.rb

@options[:decorate] = "auto"

@parser.on "--decorate[=<format>]" do |format|
  @options[:decorate] = format || "short"
end

@parser.on "--no-decorate" do
  @options[:decorate] = "no"
end

```

In Command::Log#run, we'll collect the ref information we need before we begin iterating over the history, by calling the Refs#reverse_refs method we just defined, as well as Refs#current_ref which returns the SymRef for the branch that HEAD currently points at.

```

# lib/command/log.rb

def run
  setup_pager

  @reverse_refs = repo.refs.reverse_refs
  @current_ref = repo.refs.current_ref

  each_commit { |commit| show_commit(commit) }

  exit 0
end

```

Next, we modify the existing show_commit_* methods to append the decorations for the current commit after the spot where they display the commit's ID.

```

# lib/command/log.rb

def show_commit_medium(commit)
  # ...
  puts fmt(:yellow, "commit #{ abbrev(commit) }") + decorate(commit)
  # ...
end

def show_commit_oneline(commit)
  id = fmt(:yellow, abbrev(commit)) + decorate(commit)
  puts "#{id} #{commit.title_line}"
end

```

And now we get to the decoration logic itself. If `:decorate` is set to "no", or it's set to "auto" and we're not printing to a TTY, the `decorate` method returns an empty string. Otherwise, it looks up the current commit ID in the reverse refs table, and again returns an empty string if no refs are found. If we did find some refs and the current ref is not HEAD, we filter HEAD out of the list of refs; if HEAD is a symref to a branch then we will print HEAD -> before that branch rather than printing HEAD as a distinct ref. Finally, it uses `decoration_name` (defined below) to generate the printable name for each ref, and combines the results in a parenthesised comma-separated list, highlighting the punctuation in yellow.

```
# lib/command/log.rb

def decorate(commit)
  case @options[:decorate]
  when "auto" then return "" unless @isatty
  when "no"   then return ""
  end

  refs = @reverse_refs[commit.oid]
  return "" if refs.empty?

  head, refs = refs.partition { |ref| ref.head? and not @current_ref.head? }
  names = refs.map { |ref| decoration_name(head.first, ref) }

  fmt(:yellow, " (") + names.join(fmt(:yellow, ", ")) + fmt(:yellow, ")")
end
```

The call to `partition`⁶ separates a SymRef for HEAD out of the main list unless HEAD is itself the current ref. For example, if the initial value of `refs` is [HEAD, master], this code gives us `head = [HEAD]` and `refs = [master]`. The HEAD ref is used to annotate the current ref with HEAD ->.

The `decoration_name` method is responsible for turning each SymRef object into a printable name: its `short_name` if `:decorate` is "short" or "auto", and its full path if `:decorate` is "full". It highlights this name in bold green, and prefixes it with HEAD -> in bold cyan if the SymRef is equal to the current ref and distinct from HEAD.

```
# lib/command/log.rb

def decoration_name(head, ref)
  case @options[:decorate]
  when "short", "auto" then name = ref.short_name
  when "full"      then name = ref.path
  end

  name = fmt(ref_color(ref), name)

  if head and ref == @current_ref
    name = fmt(ref_color(head), "#{head.path} -> #{name}")
  end

  name
end

def ref_color(ref)
  ref.head? ? [:bold, :cyan] : [:bold, :green]
```

⁶<https://docs.ruby-lang.org/en/2.3.0/Enumerable.html#method-i-partition>

```
end
```

To support highlighting text in both bold and a colour at the same time, we need to amend the `Command::Base#fmt` method⁷. This method puts an escape code at the end of the string that cancels all preceding formatting, for example, `fmt(:green, "hello")` returns "`\e[32mhello \e[m`". That `\e[m` at the end means the terminal resets all formatting. For this reason, it cannot be nested; for example:

```
>> fmt(:green, fmt(:bold, "hello") + " world")
=> "\e[32m\e[1mhello\e[m world\e[m"
```

The `\e[32m` turns on green colouring, and `\e[1m` turns on bold formatting. But then the `\e[m` after `hello` cancels all formatting, and so the word `world` will not be printed in green.

To fix this, we need to make `fmt` take a list of styles to merge into a single instruction, for example:

```
>> fmt([:bold, :green], "hello")
=> "\e[1;32mhello\e[m"
```

The semicolon-separated values within `\e[...m` activate multiple formatting modes at once. The change to `Color.format` to achieve this effect is fairly simple: we coerce the `style` input to an array if it isn't one already using `[*style]`, map each array element to an SGR code, and combine the results with `;`.

```
# lib/command/color.rb

def self.format(style, string)
  codes = [*style].map { |name| SGR_CODES.fetch(name.to_s) }
  "\e[#{ codes.join(";") }m#{ string }\e[m"
end
```

Our `log` command now displays logs with decorations so we can see what each branch points at.

```
$ jit log --oneline

9f8baa7 (HEAD -> master) Decorate log entries with the refs that point to them
ae6e49a One-line log format
1e781bc Abbreviate commit IDs in logs
5a2e7b8 Log a linear history in medium format
8ffdd0f Show the current branch in `status`
# etc.
```

16.1.5. Displaying patches

The final addition we'll make to `show_commit` before turning our attention to filtering the history, is the addition of the `--patch` flag, aliased as `-p` and `-u`, and the `--no-patch` flag, aliased as `-s`. The `log` and `diff` commands support these flags to either show or suppress diffs in the output. This addition won't be too complicated, it just requires a little refactoring first.

In Chapter 12, *Spot the difference*, we developed the `diff` command that contains all the logic needed to print the diff between two versions of a file. In particular, in Section 12.2.2, “A

⁷Section 10.4, “Printing in colour”

common pattern” we organised things into a method called `print_diff` that takes two `Target` objects representing the versions to compare, and it would then calculate and print the diff between them.

The `log` command needs exactly the same logic, so let’s extract the necessary methods out of `Command::Diff` and into a shared module called `PrintDiff`. The listing below just indicates which values and methods have been extracted; the method bodies are exactly as they were before.

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

module Command
  module PrintDiff

    NULL_OID  = "0" * 40
    NULL_PATH = "/dev/null"

    Target = Struct.new(:path, :oid, :mode, :data) do
      # ...
    end

    private

    def from_entry(path, entry)
    def from_nothing(path)
    def header(string)
    def short(oid)
    def print_diff(a, b)
    def print_diff_mode(a, b)
    def print_diff_content(a, b)
    def print_diff_hunk(hunk)
    def print_diff_edit(edit)

  end
end
```

To keep the `diff` command working, we just need to include this new module into the `Command::Diff` class so that it inherits the required methods. The methods in `PrintDiff` rely on the `fmt` and `puts` implementation in `Command::Base` in order to colourise the output correctly, and they will shortly need access to the command-line options, so it’s important they’re invoked as methods in a command class rather than being extracted into a distinct object.

```
# lib/command/diff.rb

module Command
  class Diff

    include PrintDiff

  end
end
```

With that extraction done, we can begin to define the patch options. We’ll define these in a method in `PrintDiff`:

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

```
def define_print_diff_options
  @parser.on("-p", "-u", "--patch") { @options[:patch] = true }
  @parser.on("-s", "--no-patch") { @options[:patch] = false }
end
```

In `Command::Diff`, we'll default the `:patch` option to `true`—that is the purpose of the `diff` command after all—and call `define_print_diff_options` to make `diff` recognise the required flags.

```
# lib/command/diff.rb

def define_options
  @options[:patch] = true
  define_print_diff_options

  @parser.on "--cached", "--staged" do
    @options[:cached] = true
  end
end
```

In this command, all the printing work can be skipped if `:patch` was switched off, so we'll just escape early from the methods that perform this work⁸.

```
# lib/command/diff.rb

def diff_head_index
  return unless @options[:patch]
  #
end

def diff_index_workspace
  return unless @options[:patch]
  #
end
```

We can now make a similar addition to `Command::Log`. In this case `:patch` defaults to `false`, but if it's set, a call to `show_patch` at the end of `show_commit` will cause that commit's diff to be calculated and printed.

```
# lib/command/log.rb

include PrintDiff

def define_options
  @options[:patch] = false
  define_print_diff_options
  #
end

def show_commit(commit)
  case @options[:format]
  when "medium" then show_commit_medium(commit)
  when "oneline" then show_commit_oneline(commit)
```

⁸It may seem somewhat pointless to have a `diff` command that allows diff printing to be turned off. However, in Git this command is also used as a programmatic check to detect changes; `diff --no-patch --exit-code` does not print anything but exits with a non-zero exit status only if there are any changes.

```
    end

    show_patch(commit)
end
```

`show_patch` is a short helper function that decides whether to display a diff based on the options, prints a blank line to separate the diff from the commit message, and then calls `print_commit_diff` which does the actual work.

```
# lib/command/log.rb

def show_patch(commit)
  return unless @options[:patch]

  blank_line
  print_commit_diff(commit.parent, commit.oid)
end
```

The `print_commit_diff` method is in the `PrintDiff` module, and mostly reuses existing building blocks. In Section 14.1, “Telling trees apart” we developed the method `Database#tree_diff`, which can take two commit IDs and return a report of which files changed between the two commits. To calculate the change introduced by a commit we generate the tree diff between its parent and itself. We can then sort the paths in the resulting diff, and for each one extract the old and new tree entries. We convert each of these entries to a `Target` via `from_entry`, and call our existing `print_diff` method to output the result.

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

def print_commit_diff(a, b)
  diff = repo.database.tree_diff(a, b)
  paths = diff.keys.sort_by(&:to_s)

  paths.each do |path|
    old_entry, new_entry = diff[path]
    print_diff(from_entry(path, old_entry), from_entry(path, new_entry))
  end
end
```

Since `old_entry` and `new_entry` can be `nil`, for paths with added or deleted files, we need to adjust `from_entry` so that it can return a null `Target` if it receives `nil` instead of a `Database::Entry`.

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

def from_entry(path, entry)
  return from_nothing(path) unless entry

  blob = repo.database.load(entry.oid)
  Target.new(path, entry.oid, entry.mode.to_s(8), blob.data)
end
```

This also necessitates one minor change to the `blank_line` function: in one-line mode, we don’t print blank lines before patch output, so we’ll make that method do nothing if `:format` is set to “`oneline`”.

```
# lib/command/log.rb
```

```
def blank_line
  return if @options[:format] == "oneline"
  puts "" if defined? @blank_line
  @blank_line = true
end
```

Diffs form such a large part of Git's internal workings, and it's useful to be able to compare two arbitrary commits to see what we changed, for example to see what functionality was altered between two releases. As such, we'll also add functionality to the `diff` command so that if it is invoked with exactly two arguments, it interprets them as revisions and prints the difference between them.

```
# lib/command/diff.rb

def run
  # ...

  if @options[:cached]
    diff_head_index
  elsif @args.size == 2
    diff_commits
  else
    diff_index_workspace
  end

  exit 0
end
```

The `diff_commits` method will resolve both the arguments to commit IDs and then pass them into the `print_commit_diff` method we defined above.

```
# lib/command/diff.rb

def diff_commits
  return unless @options[:patch]

  a, b = @args.map { |rev| Revision.new(repo, rev).resolve }
  print_commit_diff(a, b)
end
```

16.2. Branching histories

Having fleshed out the `show_commit` method with a few different options for displaying commit information, let's turn our attention to the `each_commit` method that generates the list of commits to show. We'll introduce new abilities step-by-step, allowing the `log` command to deal with more complicated histories and more ways to filter the history to find exactly what we're interested in seeing.

As a small first step, we could improve our initial `each_commit` method so that we can pass an argument to tell it which commit to start searching from, rather than assuming we always want to begin at `HEAD`. We'll use the `Revision`⁹ class to resolve the argument to a commit ID of our choosing, defaulting to `HEAD` if no arguments are given.

⁹Section 13.3, “Setting the start point”

```
# lib/command/log.rb

def each_commit
  start = @args.fetch(0, Revision::HEAD)
  oid   = Revision.new(repo, start).resolve(Revision::COMMIT)

  while oid
    commit = repo.database.load(oid)
    yield commit
    oid = commit.parent
  end
end
```

We can now pass revision identifiers, like the ones accepted by `branch`, to the `log` command, to set where we want the log to begin. The command will then print the linear history walking backwards from that point.

16.2.1. Revision lists

Before the history-searching logic gets any more complicated, it would be good to extract it into its own class to isolate its state from the rest of the `log` command. In Git, `log` is based on a lower-level command called `rev-list`¹⁰ that is just responsible for finding commits in the history matching certain criteria. You'll notice that `log` and `rev-list` accept a lot of similar options and their docs overlap somewhat; `log` uses the internals of `rev-list` to find the commits it's interested in, and then prints each commit as requested.

In Jit, we'll have our own `RevList` class that can be used to search the history. For now, it will just contain logic extracted from the `Command::Log#each_commit` method.

```
# lib/rev_list.rb

class RevList
  def initialize(repo, start)
    @repo  = repo
    @start = start || Revision::HEAD
  end

  def each
    oid = Revision.new(@repo, @start).resolve(Revision::COMMIT)

    while oid
      commit = @repo.database.load(oid)
      yield commit
      oid = commit.parent
    end
  end
end
```

We can replace the call to `each_commit` in `Command::Log` with a use of the `RevList` class instead.

```
# lib/command/log.rb

def run
  setup_pager
```

¹⁰<https://git-scm.com/docs/git-rev-list>

```

@reverse_refs = repo.refs.reverse_refs
@current_ref  = repo.refs.current_ref

@rev_list = RevList.new(repo, @args[0])
@rev_list.each { |commit| show_commit(commit) }

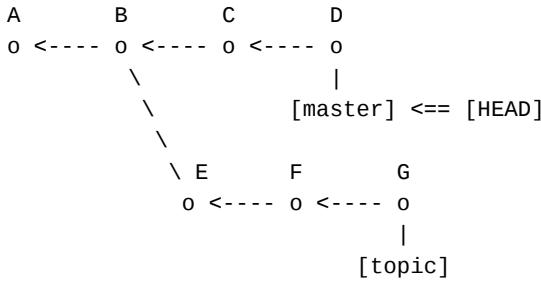
exit 0
end

```

16.2.2. Logging multiple branches

We can now look at a more complicated type of history: rather than a simple linear one, let's look at one with a couple of branches. In the history below, `HEAD` points to the `master` branch, which points at commit `D`, while the branch `topic` points at `G`. These branches have a common ancestor at `B`, and `A` is a root commit.

Figure 16.2. History with two branches



We say that commit `X` is *reachable* from commit `Y` if `X` can be found by beginning at `Y` and following any number of parent pointers from there. Another way of putting this is that `X` is an *ancestor* of `Y`. In this history, commit `B` is an ancestor of `D`, and it is also an ancestor of `C`, `E`, `F` and `G`. Commit `C` is an ancestor of `D`, but not of `G`: there is no chain of parent pointers leading from `G` to `C`.

Git's `log` command accepts multiple revision specifiers and prints out all the commits reachable from all these revisions, including the starting revisions themselves. For example, while `git log master` would print only commits `D`, `C`, `B`, `A`, and `git log topic` would display `G`, `F`, `E`, `B`, `A`, running `git log master topic` will print all the commits in the above graph.

Importantly, it only lists each commit once. That is, even though commits `A` and `B` are reachable from both `master` and `topic`, they are not listed multiple times. So `log` doesn't print the complete history of each branch in turn, but rather it displays the combined history of all the branches. This means it also avoids double-printing commits if one of the arguments is an ancestor of another. For example, `git log master topic~3` still lists `D`, `C`, `B`, `A`, even though the arguments refer to `D` and its ancestor `B`.

We know which commits should be displayed, but we also need to work out which order to show them in. The commits in a branching history are *partially ordered*, meaning that for any pair of commits it's not always possible to say which one precedes the other. It seems reasonable to suggest that, say, `E` precedes `F`; `E` is the parent of `F` and so the contents of `F` depend on `E`. Likewise, `B` should precede both `C` and `E`, being the parent of both of them. However it's not obvious which order, say, `C` and `F` should appear in. They appear on concurrent lines of history

and neither is an ancestor of the other; we could delete *C* without affecting the contents of *F* and vice versa. When we have a single branch, we can view each commit as a sequential snapshot of the project in time. Once multiple branches exist, it's like we have parallel universes, and events on one branch don't affect those on another.

Nevertheless, this gives us an intuition for some log outputs that we'd consider unacceptable, since they'd make no sense to the reader. For example, on running `log master topic`, it would not be acceptable to print the entire history leading to `master`, followed by commits leading to `topic` we'd not yet visited, i.e. *D C B A G F E*. This results in commits near the tip of `topic` being displayed after their ancestors *A* and *B*, and this effect gets worse the more commits there are before the fork point at *B*.

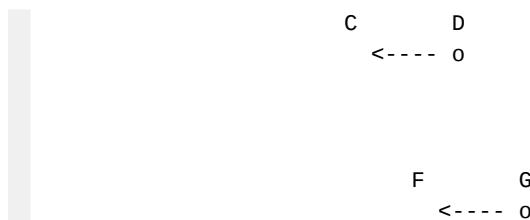
The one benefit of this approach is that it retains the efficient behaviour we highlighted in Section 16.1, “Linear history”: we don't need to read the entire history into memory before printing anything, and commits can be streamed out to the display as they're visited. But maybe there's another approach we can try. Remember that our program cannot see the entire history at once, it has to load it from disk by reading one commit at a time and following its parent pointer. In the beginning, if we're running `log master topic`, the only information we have is that commits *D* and *G* exist. We only know their IDs, we have not yet seen their contents.

Figure 16.3. Starting commits for the history search



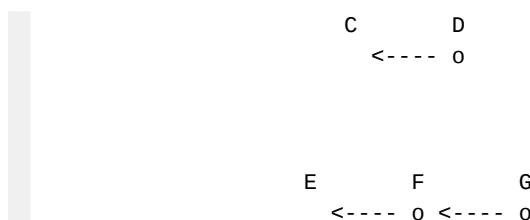
When we read each of these commits from disk, we discover the name of each of their parents: *C* is the parent of *D*, and *F* of *G*. We'll represent a loaded commit with a dot, and any commits whose names we know but which we've not yet loaded are shown without a dot marker.

Figure 16.4. Reading the start commits to find their parents



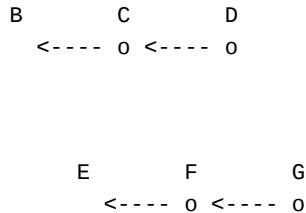
Now maybe, instead of printing the entire history from one starting point before moving onto the next, we could alternate between them. We'll print commit *G* and load its parent, *F*. This tells us that commit *E* exists.

Figure 16.5. Continuing along the topic branch



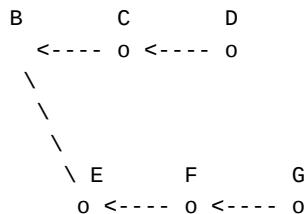
Then, we could alternate to the `master` branch, print commit `D` and load its parent, `C`. It's important to note that at this point we don't know when or even *if* these two lines will converge on a common ancestor. We know `B` and `E` exist, but as we've not loaded `E` we don't know that `B` is its parent yet. Until we find this out there's an unpredictable amount of work ahead of us — the common ancestor might be thousands of commits away and we don't want to wait that long before printing any output.

Figure 16.6. Continuing along the master branch



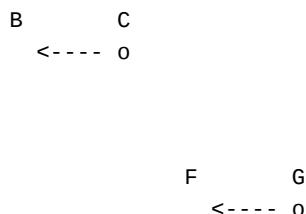
Next we would switch back to our original branch, printing commit `F` and loading `E`. At this point we discover it has the same parent as `C`, and the branches have converged. Continuing the alternation, we'd print `C`, then `E`, then `B`, then `A`. Since `B` comes after all of its descendants, that seems like a reasonable outcome.

Figure 16.7. Locating the common ancestor



That method seemed to work out fine. But what if we instead asked to log `master^ topic`? Loading the commits corresponding to these two start points would give us this information:

Figure 16.8. Starting commits for a new log search



Superficially this looks very similar to our previous start point: we have two commits, each with a different parent. We don't know when they will converge. But, if we try our round-robin approach this time, starting with `G`, we would print: `G C F B E A`. This doesn't make a lot of sense — `B`, the parent of `E`, appears before `E` in the output. How can we fix this without needing to load the whole history so that we can put everything in order before printing anything?

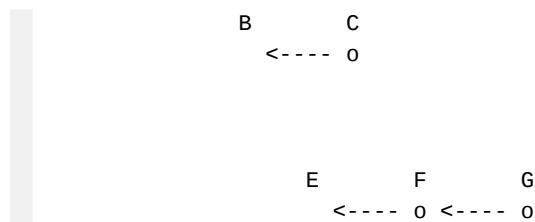
Here Git uses a neat trick. In Section 2.3.4, “Commits on disk” we learned that all commits have two headers that contain a timestamp: `author` and `committer`. In our implementation so

far, these headers always contain the same information, but we'll encounter situations where they can differ later¹¹. If everyone writing to the repository has an accurate system clock, then it will be the case that every commit has a committer timestamp that's greater than or equal to its parent: a commit must be created after its parent, since it depends on its parent's ID.

We can use this information to prioritise commits so that we always print a commit before its parent, even when we don't know when multiple branches are going to converge. In the figures above, assume that commits are arranged from left to right in order of ascending timestamp; if commit *A* is to the left of commit *B*, then it is older than commit *B*. By always picking the youngest commit to process next, we can solve our ordering problem.

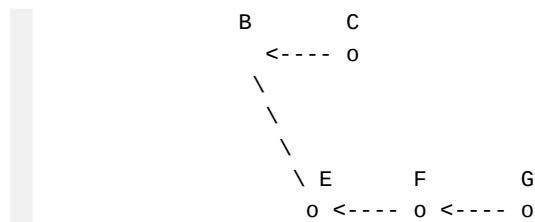
The two commits we've loaded so far are *C* and *G*. Ordering them by age, youngest first, we get a queue containing *G*, *C*. We take the first item off the queue, display it, and load its parent, *F*.

Figure 16.9. Processing the most recent commit



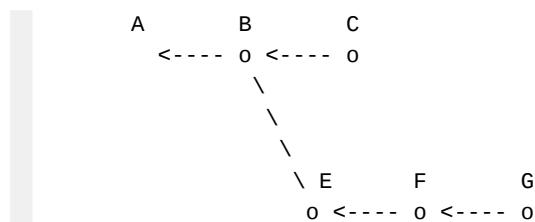
We now add *F* to the queue, preserving the order so that younger commits appear first. *F* is younger than *C*, so the queue is now *F*, *C*. We repeat the procedure, taking *F* off the front of the queue, printing it and loading its parent, *E*.

Figure 16.10. Processing the second youngest commit



E is older than *C*, so we place it behind *C* in the queue. Processing *C* next gives us:

Figure 16.11. Processing the third youngest commit



B is older than *E*, so the queue now contains *E*, *B*. The process continues, resulting in the output being *G F C E B A* — a more reasonable account of the project's history. If all commits are

¹¹Section 22.3.2, “Recording the committer”

written using accurate clocks, this method means we'll print all the commits on a set of branches before any of their common ancestors.

To implement this functionality, let's begin by passing all the command-line arguments into `RevList`, rather than only the first one.

```
# lib/command/log.rb

@rev_list = RevList.new(repo, @args)
```

Now, we're going to change the internals of `RevList` so that its each method yields the combined history of all the start points. It's going to need a few more pieces of state and some helper methods to support the graph search:

- `@commits` is a hash mapping object IDs to `Commit` objects. It caches every commit that we load via the `load_commit` method. This cache will initially prevent us re-loading data we already have, but later on it will also help us locate commits we've loaded whose information might need amending.
- `@flags` is a hash mapping commit IDs to a set of symbols. As we walk through the graph we'll associate various *flags* with each commit to mark their status, for example, to record the fact we've already visited them.
- `@queue` stores the commit-time priority queue of commits we still need to visit. We call `handle_revision`, defined below, with each input argument to add a commit to the starting queue, and queue up the `HEAD` commit if no arguments were given.

```
# lib/rev_list.rb

class RevList
  def initialize(repo, revs)
    @repo      = repo
    @commits  = {}
    @flags    = Hash.new { |hash, oid| hash[oid] = Set.new }
    @queue    = []

    revs.each { |rev| handle_revision(rev) }
    handle_revision(Revision::HEAD) if @queue.empty?
  end

  private

  def load_commit(oid)
    return nil unless oid
    @commits[oid] ||= @repo.database.load(oid)
  end

  #
end
```

To manage flags, we have two methods. `RevList#mark` adds a flag to a given commit ID, using `Set#add?`¹² which returns `true` only if the flag was not already in the commit's set.

¹²<https://docs.ruby-lang.org/en/2.3.0/Set.html#method-i-add-3F>

RevList#marked? uses Set#include?¹³ to check whether a commit has a certain flag, without changing its state.

```
# lib/rev_list.rb

def mark(oid, flag)
  @flags[oid].add?(flag)
end

def marked?(oid, flag)
  @flags[oid].include?(flag)
end
```

The handle_revision and enqueue_commit methods handle processing of the input arguments to set up the initial state of the queue. handle_revision uses the Revision class to convert the input to a commit ID, loads the commit, and adds it to the queue. enqueue_commit takes the Commit object, and marks it with the :seen flag, bailing out if that flag was already set. This means the commit has already been visited and we don't need to reprocess it. It finds the first item in the queue whose date is earlier than the new commit, and inserts the commit before that item, or at the end of the queue if no such item was found. This keeps the queue ordered in reverse date order.

```
# lib/rev_list.rb

def handle_revision(rev)
  oid = Revision.new(@repo, rev).resolve(Revision::COMMIT)

  commit = load_commit(oid)
  enqueue_commit(commit)
end

def enqueue_commit(commit)
  return unless mark(commit.oid, :seen)

  index = @queue.find_index { |c| c.date < commit.date }
  @queue.insert(index || @queue.size, commit)
end
```

That takes care of managing the queue and setting flags against each commit. The only thing left to do is implement RevList#each so that it walks the history as described above, yielding all the commits in order. It does this using an internal helper function called traverse_commits. This loops until the queue is empty, and at each iteration, it shifts the first commit off the queue, adds its parents to the queue, and yields it. add_parents again uses mark with the :added flag, which prevents the same commit being reprocessed.

```
# lib/rev_list.rb

def each
  traverse_commits { |commit| yield commit }
end

def traverse_commits
  until @queue.empty?
    commit = @queue.shift
```

¹³<https://docs.ruby-lang.org/en/2.3.0/Set.html#method-i-include-3F>

```

    add_parents(commit)
    yield commit
end
end

def add_parents(commit)
  return unless mark(commit.oid, :added)

  parent = load_commit(commit.parent)
  enqueue_commit(parent) if parent
end

```

The only change required to existing code to make all the above work is that we need to add a date method to `Database::Commit`. Git uses the committer time as the commit's date, because this is more likely to go in ascending order than the author time. However, in Jit so far, these headers are always identical, so we'll just return the author time and amend this method at such time as we add support for distinct committer data.

```

# lib/database/commit.rb

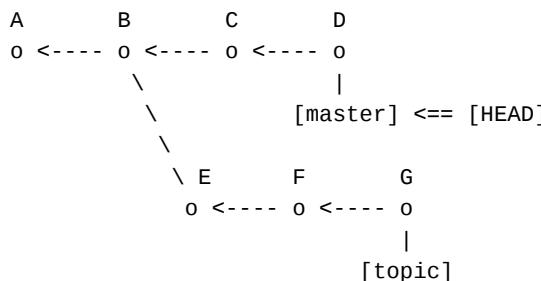
def date
  @author.time
end

```

16.2.3. Excluding branches

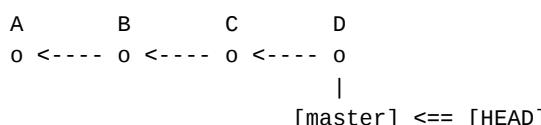
It's often useful, when working with branches, to see which changes have been applied to other branches that are not merged into your history. For example, returning to the branching history we've been examining, suppose you're working on the `topic` branch, and you'd like to know what's happened on `master` since you diverged from it.

Figure 16.12. History with two branches



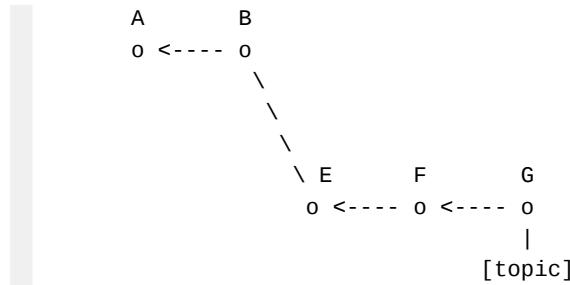
Given this history, we'd expect to be shown commits `D` and `C` in response to this request. Git has a couple of ways of phrasing this: we can either ask for `git log topic..master`, or `git log ^topic master`. These two commands are exactly equivalent. The caret (^) before a revision tells Git to omit from the logs that commit itself, and any commits that are reachable from it. You can think of it as taking the history reachable from `master`:

Figure 16.13. History reachable from master



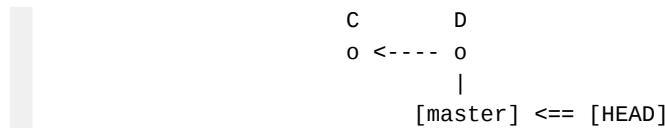
And subtracting the history reachable from topic:

Figure 16.14. History reachable from topic



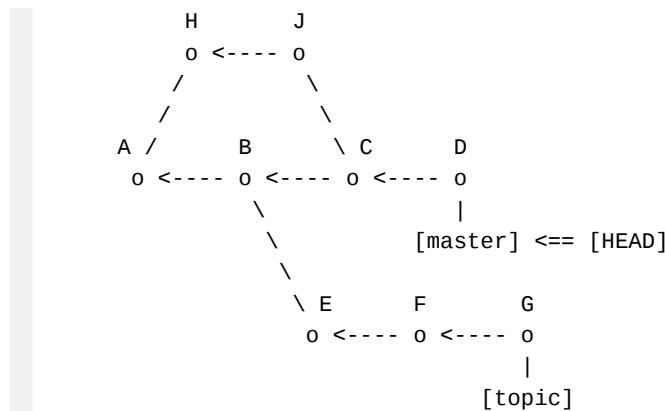
Leaving us with this portion of the history:

Figure 16.15. History reachable from master and not from topic



At first it seems tempting to work out the nearest common ancestor of the two branches¹⁴, *B*, and conduct a scan from commit *D*, stopping when we reach this common ancestor. When commits only have a single parent, this will give the correct result, but it has a performance problem: finding the common ancestor requires scanning the history once, and we then perform another scan to actually construct the history. We'd rather do everything in a single pass if possible. Worse, when we introduce merging and commits can have multiple parents, it won't even give the right answer. Consider this history, where there's a side-branch beginning at *A* and merging into *C*:

Figure 16.16. History containing merge commits



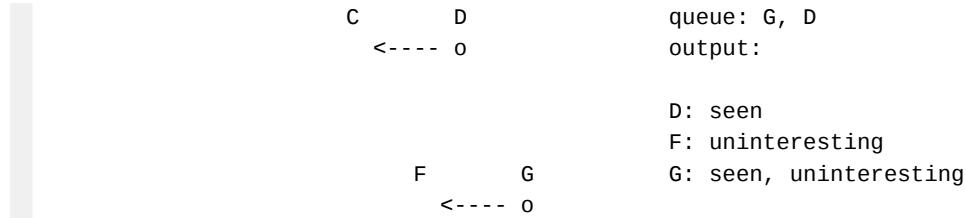
The nearest common ancestor of `master` and `topic` is still *B*, but simply excluding *B* from the results isn't enough. We should also exclude *A*, since it's reachable from `topic`, but scanning back from `master` and stopping at *B* still lets us include *A*, as it's reachable via the side-branch.

Instead, we need to extend the `RevList` algorithm to walk the history from the start points as usual, but mark anything reachable via an excluded start point so that it's filtered out of the results. Here's how Git achieves this.

¹⁴Section 17.5, “Best common ancestors with merges”

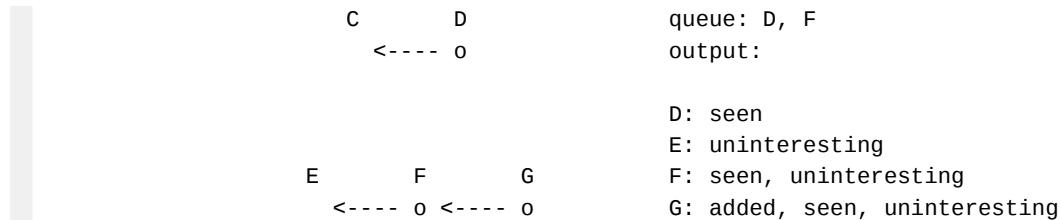
We begin as usual by loading the start-point commits D and G . We add them to the queue, ordered by date, and we mark them both as :seen. The excluded commit G and its parent F are marked :uninteresting. We can mark F even though it's not loaded because we store flags against object IDs, not Commit objects.

Figure 16.17. Initial state for a range search



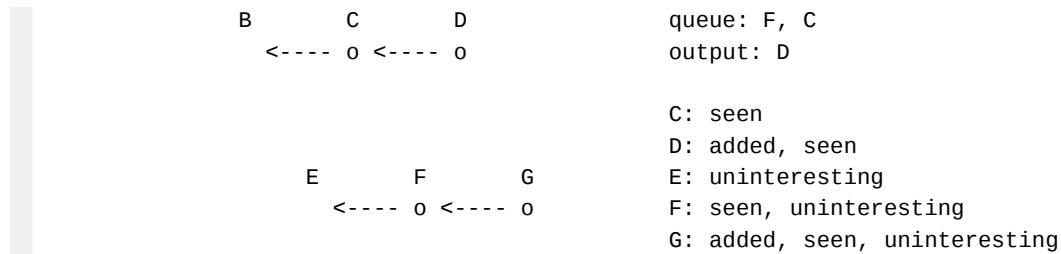
As before, we shift G off the front of the queue, mark it :added, and load its parent, F . Because G is marked :uninteresting, we mark F and its parent E in the same way. We add F to the queue and mark it :seen.

Figure 16.18. Processing an excluded commit



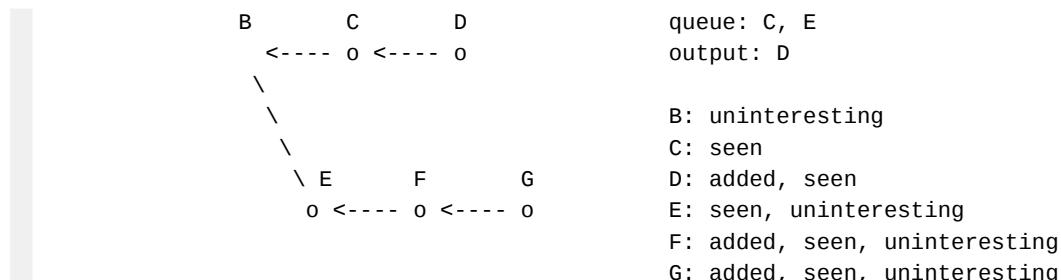
Next in the queue is D . We shift it off, mark it :added, load its parent C , and add C to the queue, marking it :seen. Because D is not marked :uninteresting, it is added to the output list, and its ancestors are not marked further.

Figure 16.19. Processing an included commit



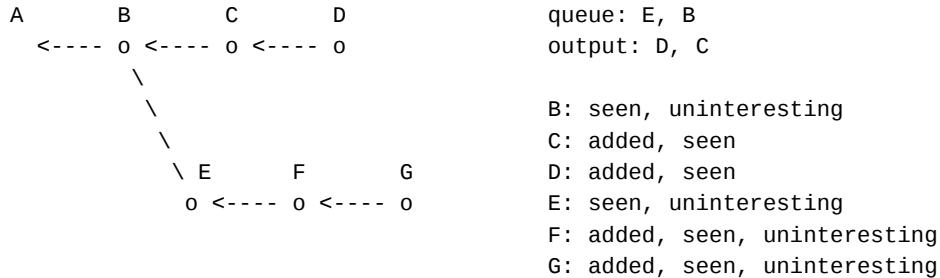
We then process F . We mark it :added, load its parent E and add that to the queue marking it :seen. Since F is uninteresting, E and B are marked as such.

Figure 16.20. Processing an uninteresting commit



Finally we process *C*, which causes *B* to be loaded and enqueued. Because *C* is not uninteresting, we add it to the output list. This leaves things in a state where all the commits in the queue are marked :uninteresting, and at this point we can stop: we will not discover any further interesting commits from this position.

Figure 16.21. Stopping the search on an all-uninteresting queue



Notice that we've stopped before loading commit *A*. When we have excluded start points, we cannot print every commit as soon as we find it, because it might turn out to be reachable from an excluded commit that we haven't seen yet. Since this history scan is blocking us from producing any output, we'd like to end it as soon as possible, without walking right back to the start of the history.

Translating this into code, the `RevList` class needs a couple new pieces of state. We say a `RevList` is *limited* if it has any excluded start points. And, we need a list in which to buffer possible output commits that's distinct from the input queue.

```
# lib/rev_list.rb

def initialize(repo, revs)
    # ...
    @limited = false
    @output = []
    # ...
end
```

The `handle_revision` method needs to be expanded to handle the two exclusion notations, `A..B` and `^A`. With the range operator `..`, either side of the range can be blank and it defaults to `HEAD`, for example `..topic` is equivalent to `HEAD..topic`. If the argument matches the range operator, we extract the two endpoints (which might be empty strings), and mark the first as uninteresting. If we see the caret operator we extract the revision following it and mark it as uninteresting. Otherwise, we treat the input as a normal interesting revision.

```
# lib/rev_list.rb

RANGE   = /^(.*)\.\.(.*)$/
EXCLUDE = /^^(.+)$/

def handle_revision(rev)
    if match = RANGE.match(rev)
        set_start_point(match[1], false)
        set_start_point(match[2], true)
    elsif match = EXCLUDE.match(rev)
        set_start_point(match[1], false)
    else
```

```
    set_start_point(rev, true)
  end
end

def set_start_point(rev, interesting)
  rev = Revision::HEAD if rev == ""
  oid = Revision.new(@repo, rev).resolve(Revision::COMMIT)

  commit = load_commit(oid)
  enqueue_commit(commit)

  unless interesting
    @limited = true
    mark(oid, :uninteresting)
    mark_parents_uninteresting(commit)
  end
end
```

When a start point is uninteresting, we set the `@limited` flag, mark the loaded commit as `:uninteresting`, and also mark its parents similarly. This is done using the `mark_parents_uninteresting` method:

```
# lib/rev_list.rb

def mark_parents_uninteresting(commit)
  while commit&.parent
    break unless mark(commit.parent, :uninteresting)
    commit = @commits[commit.parent]
  end
end
```

This method uses the `@commits` cache to mark any loaded ancestors of the given commit as uninteresting. When we encounter an uninteresting commit, we may have already loaded it and some of its ancestors via an interesting route and placed those commits in the output list. We need to go back and mark all the commits reachable from here as `:uninteresting`, but only if we've already loaded them into memory — we don't want to read the entire history off disk if we can avoid it.

Next, we need to change how the `each` method works. Previously it just called the internal method `traverse_commits` to iterate over the history, but now we need to do a bit of preparatory work if the `RevList` is limited. In this case we call a new method called `limit_list` to filter the history for uninteresting commits.

```
# lib/rev_list.rb

def each
  limit_list if @limited
  traverse_commits { |commit| yield commit }
end

def limit_list
  while still_interesting?
    commit = @queue.shift
    add_parents(commit)

    unless marked?(commit.oid, :uninteresting)
```

```

        @output.push(commit)
    end
end

@queue = @output
end

```

The `limit_list` method works a little like the original `traverse_commits` method, and it does a very similar job. Instead of walking the history and immediately yielding each commit, it walks the history until it runs out of interesting commits to look at, and buffers the interesting ones in the `@output` list. At the end, it reassigns `@queue` to point to the output list so that `traverse_commits` can consume it.

Instead of iterating until the queue is empty, `limit_list` iterates as long as the more complex condition `still_interesting` is true. In broad terms, this method tries to determine whether it's worth following any of the commits left in the queue. If the queue is empty, then there's definitely no more work to do. Conversely, if there are any commits in the queue that aren't marked as uninteresting, then we should keep searching. The date-based condition in the middle of the method deals with an edge case that we'll explore below.

```

# lib/rev_list.rb

def still_interesting?
    return false if @queue.empty?

    oldest_out = @output.last
    newest_in = @queue.first

    return true if oldest_out and oldest_out.date <= newest_in.date

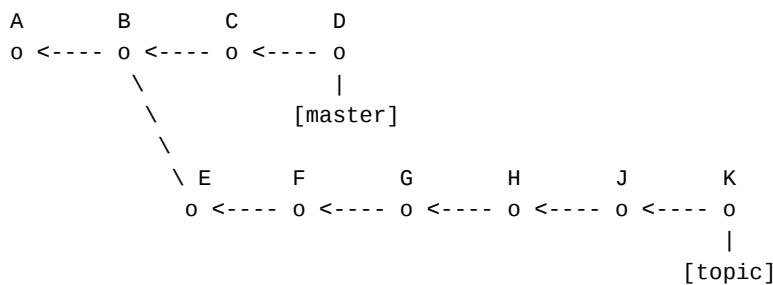
    if @queue.any? { |commit| not marked?(commit.oid, :uninteresting) }
        return true
    end

    false
end

```

The edge case happens when we run a command like `log topic..master` on the history below.

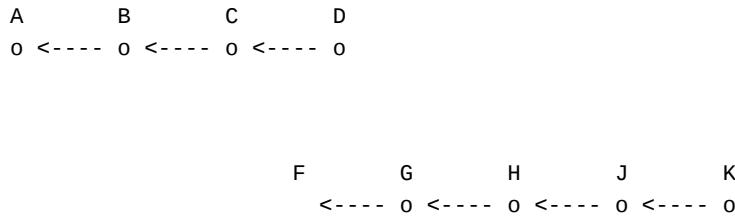
Figure 16.22. Commit graph with a long branch



Ordinarily, the commit timestamps should be such that we process everything to the right of commit `B` before `B` itself. However, if all the commits have the same timestamp, or even decreasing timestamps, we can end up in a situation where we bounce between the two branches, always putting parents at the end of the queue. This leads to us having the following commits

loaded, with the output buffer containing D , C , B , A and the input queue containing just G , which is marked uninteresting.

Figure 16.23. Loaded commits part-way through a graph search



It would be a mistake to stop here — the expected output should not include commits B and A since they are reachable from K , we just haven't discovered that yet. We'd like to know if we could reach any of the interesting commits from the uninteresting ones, and as long as the uninteresting set contains commits that are newer than any in the interesting set, that might still happen. Therefore, we pick the newest item from the queue (the first element) and the oldest commit from the output list (the last element, usually), and compare their dates.

Git does not actually guarantee that commits have ascending timestamps along a line of history, and you can in fact write commits with any timestamps on them you want. In pathological cases the above heuristic won't be enough to guarantee finding everything, and so Git actually has a tolerance built in. It allows for the conditions in `still_interesting` to fail up to five times before giving up. However, for demonstration purposes, most machine's clocks are accurate enough that the assumptions we're making in this implementation are reasonable.

Finally, a couple of tweaks are needed to existing methods to keep everything working. When we traverse the parent links in `add_parents`, we should mark parents uninteresting if the current commit is uninteresting.

```
# lib/rev_list.rb

def add_parents(commit)
  return unless mark(commit.oid, :added)

  parent = load_commit(commit.parent)
  return unless parent

  if marked?(commit.oid, :uninteresting)
    mark_parents_uninteresting(parent)
  end

  enqueue_commit(parent)
end
```

And in `traverse_commits`, we should avoid calling `add_parents` if `@limited` is set, since in that case we've already walked the history. We also need to skip commits that are marked uninteresting, as that can happen after they've been added to the output list.

```
# lib/rev_list.rb

def traverse_commits
  until @queue.empty?
    commit = @queue.shift
    add_parents(commit) unless @limited
```

```
    next if marked?(commit.oid, :uninteresting)
    yield commit
  end
end
```

With these additions in place, we can now see all the changes we'd receive when merging a given branch into the current HEAD by running:

```
$ jit log --patch ..<branch-name>
```

16.2.4. Filtering by changed paths

One final common use case for the `log` command that it won't take us much effort to add is the ability to show all the commits that changed a particular file or directory. If any of the arguments you pass to `log` correspond to the name of a file in the workspace, the history is simplified to include only the commits that affect that file.

To begin with, we'll need the ability to recognise filenames and add them to a configuration variable in `RevList`. In `handle_revision`, we add the argument as a `Pathname` to the `@prune` list if `workspace#stat_file` returns anything for that path.

```
# lib/rev_list.rb

def initialize(repo, revs)
  # ...
  @prune = []
  # ...
end

def handle_revision(rev)
  if @repo.workspace.stat_file(rev)
    @prune.push(Pathname.new(rev))
  elsif match = RANGE.match(rev)
    # ...
  end
end
```

After all the revisions have been parsed, we use the `@prune` list to build a `PathFilter`, an object that will contain the logic for matching against this list of pathnames as we traverse trees while computing diffs. We'll see how this is implemented below, after we've seen what functionality `TreeDiff` needs it to provide.

```
# lib/rev_list.rb

def initialize(repo, revs)
  # ...
  @filter = PathFilter.build(@prune)
end
```

In `RevList#add_parents`, if the commit is not marked as uninteresting, we pass it to another method called `simplify_commit` which, if any prune paths are set, attempts to eliminate parts of the history that don't change anything we're interested in. Any commit that leaves the given paths the same is marked `:treesame`.

```
# lib/rev_list.rb
```

```
def add_parents(commit)
  return unless mark(commit.oid, :added)

  parent = load_commit(commit.parent)

  if marked?(commit.oid, :uninteresting)
    mark_parents_uninteresting(parent) if parent
  else
    simplify_commit(commit)
  end

  enqueue_commit(parent) if parent
end

def simplify_commit(commit)
  return if @prune.empty?
  mark(commit.oid, :treesame) if tree_diff(commit.parent, commit.oid).empty?
end
```

RevList#tree_diff takes a pair of commit IDs and returns the TreeDiff between them, filtered for the paths we care about. We'll explore how that filtering is done below. The diff is cached so that we can reuse it later on, for example in printing the diff if the --patch flag is set.

```
# lib/rev_list.rb

def initialize(repo, revs)
  # ...
  @diffs = {}
  # ...
end

def tree_diff(old_oid, new_oid)
  key = [old_oid, new_oid]
  @diffs[key] ||= @repo.database.tree_diff(old_oid, new_oid, @filter)
end
```

The final change needed to RevList is to make traverse_commits filter commits marked :treesame out of the output.

```
# lib/rev_list.rb

def traverse_commits
  until @queue.empty?
    commit = @queue.shift
    add_parents(commit) unless @limited

    next if marked?(commit.oid, :uninteresting)
    next if marked?(commit.oid, :treesame)

    yield commit
  end
end
```

Now we need to go over and implement the path filtering functionality in the Database::TreeDiff class. First, the Database#tree_diff method needs to accept an optional PathFilter as an argument and pass it into the TreeDiff#compare_oids method.

```
# lib/database.rb
```

```
def tree_diff(a, b, filter = PathFilter.new)
  diff = TreeDiff.new(self)
  diff.compare_oids(a, b, filter)
  diff.changes
end
```

The TreeDiff#compare_oids method takes this PathFilter object and uses it to do two things: it needs to track the pathname of the tree it's currently comparing, and it needs to restrict its search to only those paths that are of interest. It does this by passing the filter argument into the detect_deletions and detect_additions methods, which will use it to guide their operation.

```
# lib/database/tree_diff.rb

def compare_oids(a, b, filter)
  return if a == b

  a_entries = a ? oid_to_tree(a).entries : {}
  b_entries = b ? oid_to_tree(b).entries : {}

  detect_deletions(a_entries, b_entries, filter)
  detect_additions(a_entries, b_entries, filter)
end
```

Within the difference-detecting methods, where we previously iterated over the a and b hashes directly, we now pass them to the PathFilter by calling filter.each_entry(a). PathFilter will restrict the iteration so that only entries that match the requested paths are visited.

```
# lib/database/tree_diff.rb

def detect_deletions(a, b, filter)
  filter.each_entry(a) do |name, entry|
    other = b[name]
    next if entry == other

    sub_filter = filter.join(name)

    tree_a, tree_b = [entry, other].map { |e| e&.tree? ? e.oid : nil }
    compare_oids(tree_a, tree_b, sub_filter)

    blobs = [entry, other].map { |e| e&.tree? ? nil : e }
    @changes[sub_filter.path] = blobs if blobs.any?
  end
end
```

A similar change is applied to detect_additions.

The call to filter.join(name) is what allows PathFilter to track the current path we're visiting. Like the call to Pathname#join that it replaces, it adds the current filename onto the current directory name to construct the complete path for the current entry; PathFilter#path exposes this complete path. It also updates the filter so that it can select the right entries in the recursive calls to compare_oids.

Let's now look at how PathFilter works. Let's say that log was run with the arguments lib/database/tree_diff.rb and test. That means we want to see commits that change the single

file `lib/database/tree_diff.rb` and that change any file in the `test` directory. `RevList` will call `PathFilter.build` with the input:

```
[  
  Pathname("lib/database/tree_diff.rb"),  
  Pathname("test")  
]
```

In the top-level directory, `TreeDiff` should consider changes to only the `lib` and `test` entries, as the others do not match the given arguments. As it recurses into each of those directories, it must be updated to only consider paths that are children of each directory. For example, when it enters the `test` directory, it should not perform any further filtering, as everything in the `test` directory is interesting. But when it enters the `lib` directory, it should consider only the `database` entry within, and only the `tree_diff.rb` entry within that.

`PathFilter` implements a data structure that makes this matching straightforward and efficient. It constructs a *trie*¹⁵, a recursive structure that represents the above list of paths, where each entry records whether the path matches one of the inputs, and what that path's children are. In the case of the above input, this trie structure will resemble the following.

```
Trie(matched = false, children = {  
  "lib" => Trie(matched = false, children = {  
    "database" => Trie(matched = false, children = {  
      "tree_diff.rb" => Trie(matched = true, children = {})  
    })  
  }),  
  "test" => Trie(matched = true, children = {})  
})
```

`PathFilter.build` constructs one of these structures from the given list of `Pathname` objects, and also initialises a `@path` variable to an empty `Pathname`; this will be used to track the current complete path as `TreeDiff` scans the tree.

```
# lib/path_filter.rb  
  
class PathFilter  
  attr_reader :path  
  
  def self.build(paths)  
    PathFilter.new(Trie.from_paths(paths))  
  end  
  
  def initialize(routes = Trie.new(true), path = Pathname.new(""))  
    @routes = routes  
    @path = path  
  end  
  
  # ...  
end
```

The `PathFilter::Trie` class implements the recursive structure that we'll use to match paths. It's a struct containing the fields `matched` and `children`; `matched` is a boolean recording whether this entry in the trie represents a path in the input list, and `children` is a hash

¹⁵<https://en.wikipedia.org/wiki/Trie>

containing the child paths of the current entry. `Trie.from_paths` constructs a trie from an array of `Pathname` objects. If there are no paths, the root node is marked as `matched`, otherwise the `matched` flag is set only on those entries that represent the inputs.

```
# lib/path_filter.rb

Trie = Struct.new(:matched, :children) do
  def self.from_paths(paths)
    root = Trie.node
    root.matched = true if paths.empty?

    paths.each do |path|
      trie = root
      path.each_filename { |name| trie = trie.children[name] }
      trie.matched = true
    end

    root
  end

  def self.node
    Trie.new(false, Hash.new { |hash, key| hash[key] = Trie.node })
  end
end
```

`Trie.node` is a helper function that constructs a new trie node whose child nodes are generated on demand. If we construct a new node, we can then set the `matched` flag on some node nested deeply within it. The child nodes are generated as needed; this simplifies the code required to build up the trie in `from_paths`.

```
>> node = Trie.node
=> Trie(matched = false, children = {})

>> node.children["a"].children["b"].children["c"].matched = true

>> node
=> Trie(matched = false, children = {
  "a" => Trie(matched = false, children = {
    "b" => Trie(matched = false, children = {
      "c" => Trie(matched = true, children = {}))}))
```

The `PathFilter` class itself has two public methods: `each_entry` and `join`. `each_entry` iterates over a hash, yielding only those entries that match the trie stored in the `@routes` variable. That is, either the current trie's `matched` flag is set, or it contains a child whose name matches the current entry.

```
# lib/path_filter.rb

def each_entry(entries)
  entries.each do |name, entry|
    yield name, entry if @routes.matched || @routes.children.has_key?(name)
  end
end
```

`PathFilter#join` is called by `TreeDiff` as it recurses down the tree structure. It returns a new `PathFilter` whose `@path` is augmented with the new path segment, and whose `@routes` has

been filtered to the requested sub-trie. If `@routes` is already a matching trie, then we continue to use that; once we've hit a matching path, all entries under that path should be considered. But otherwise, we select the relevant child from the current trie.

```
# lib/path_filter.rb

def join(name)
  next_routes = @routes.matched? ? @routes : @routes.children[name]
  PathFilter.new(next_routes, @path.join(name))
end
```

For example, if the current `@routes` trie represents the paths `lib/database/tree_diff.rb` and `test`, and we call `join` with the path segment `lib`, then we get a new `PathFilter` whose path is `lib` and whose `@routes` trie matches the path `database/tree_diff.rb`. Constructing this class as a trie makes it easy to select relevant parts of the input as we traverse a tree, and returning a new `PathFilter` rather than modifying the existing instance means there's no mutable state to manage as we drill down different branches.

Finally, we can reuse the `RevList#tree_diff` method in the `Command::Log` class. When using the `--patch` option in combination with path filtering, the `log` command should print diffs only for the selected paths. Since we've already computed those while constructing the history, we can speed things up by reusing the result while printing the output. We'll do this by passing `@rev_list` into the `print_commit_diff` method.

```
# lib/command/log.rb

def show_patch(commit)
  return unless @options[:patch]

  blank_line
  print_commit_diff(commit.parent, commit.oid, @rev_list)
end
```

`print_commit_diff` gains an optional parameter named `differ`, which defaults to `repo.database`. This allows it to call `tree_diff` either on `Database` as normal, or on a `RevList` that the caller passed in. The rest of its behaviour remains unchanged, this just lets us reuse the tree diff that the `log` command already computed.

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

def print_commit_diff(a, b, differ = nil)
  differ ||= repo.database
  diff    = differ.tree_diff(a, b)
  paths   = diff.keys.sort_by(&:to_s)

  paths.each do |path|
    old_entry, new_entry = diff[path]
    print_diff(from_entry(path, old_entry), from_entry(path, new_entry))
  end
end
```

We're now able to review the changes on a branch and inspect changes to particular files we're interested in, setting us up to move on to the `merge` command.

17. Basic merging

Having developed the log command to help us review changes on different branches, we are now in a position to implement merging. Although merging involves a lot of subtle details and we'll cover a lot of ground in implementing it, it is built on top of many of the tools and concepts we've already covered. It is also centrally important to several other Git commands, so once we have merging working, we'll be able to add a few other features in quick succession.

17.1. What is a merge?

Before diving in, we should get a basic handle on what *merging* actually entails. From your day-to-day usage of Git, you may understand that it incorporates changes from some other branch into your current branch. But as we've seen, Git *branches* don't identify a range of commits per se, but are merely pointers to a single commit in the history graph. So what is really going on?

17.1.1. Merging single commits

To begin with, let's look at the simplest possible merge situation: we have two branches that each add one commit on top of their common starting point, and then merge them. The root commit *A* contains two files, *f.txt* and *g.txt*, both containing the value 1. Commit *B* is made on the *master* branch¹, on top of commit *A*, and changes the contents of *f.txt* to 2. Commit *C*, on branch *topic*, is also derived from commit *A* and changes the value of *g.txt* to 3. We can produce this situation with the following commands:

```
$ git init git-merge-basic
$ cd git-merge-basic

$ echo "1" > f.txt
$ echo "1" > g.txt
$ git add .
$ git commit --message "A"

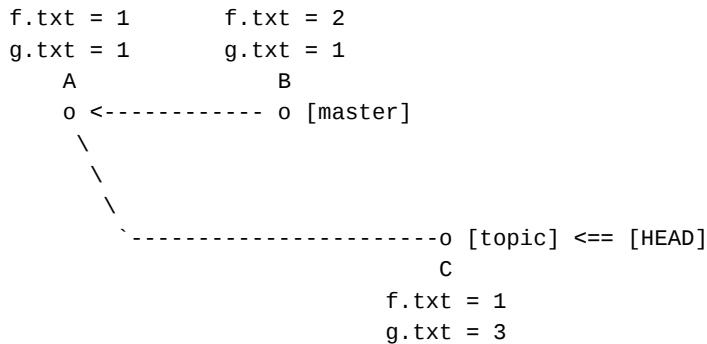
$ echo "2" > f.txt
$ git commit --all --message "B"

$ git checkout -b topic master^
$ echo "3" > g.txt
$ git commit --all --message "C"
```

The commit history looks like this, with the commits as usual arranged from left to right in ascending timestamp order. Each commit is annotated with the full contents of its tree.

¹Although Git branches are just pointers to single commits, it is common to refer to commits as being ‘on the *foo* branch’ if the commit is made with *.git/HEAD* pointing at *.git/refs/heads/foo*, or if the commit is reachable from *foo*.

Figure 17.1. Two branches changing different files



Now, let's switch back to the master branch and merge topic into it.

```
$ git checkout master
$ git merge topic --message "M"
```

Examining the resulting commit using the cat-file command shows us that, unlike all the commits we've seen previously, it has two parents. These parents are commits *B* and *C*.

```
$ git cat-file -p @

tree b0fde2e3d4ae9387eebda6eb60b193bfffec86ba3
parent 84d27ae66bd25ef58885855f1c8bcd85961b567
parent 24372dff864b5195d8f61e6fbf335d210a0c77d4
author James Coglan <james@jcoglan.com> 1536070859 +0000
committer James Coglan <james@jcoglan.com> 1536070859 +0000

M
```

What about the contents of the tree in commit *M*? Here's the tree of commit *A*; both f.txt and g.txt have the same content, blob d00491f....

```
$ git cat-file -p @~1^{tree}
100644 blob d00491fd7e5bb6fa28c517a0bb32b8b506539d4d      f.txt
100644 blob d00491fd7e5bb6fa28c517a0bb32b8b506539d4d      g.txt
```

Below are commits *B* and *C*. Commit *B* changes the contents of f.txt to 0cfbf08... relative to commit *A*, while *C* changes g.txt to 00750ed... but leaves f.txt unchanged relative to *A*.

```
$ git cat-file -p @^1^{tree}
100644 blob 0cfbf08886fc9a91cb753ec8734c84fcbe52c9f      f.txt
100644 blob d00491fd7e5bb6fa28c517a0bb32b8b506539d4d      g.txt

$ git cat-file -p @^2^{tree}
100644 blob d00491fd7e5bb6fa28c517a0bb32b8b506539d4d      f.txt
100644 blob 00750edc07d6415dcc07ae0351e9397b0222b7ba      g.txt
```

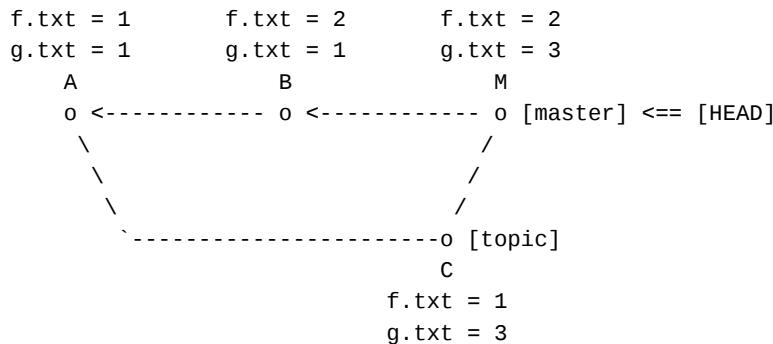
What does it mean to merge commit *C* into commit *B*, when both commits have different content for all the files in the project? How do we produce a single tree by combining them? Let's look at the tree of the merge commit *M*. It seems to contain the version of f.txt from *B*, and the *C* version of g.txt.

```
$ git cat-file -p @^{tree}
100644 blob 0cfbf08886fc9a91cb753ec8734c84fcbe52c9f      f.txt
```

```
100644 blob 00750edc07d6415dcc07ae0351e9397b0222b7ba      g.txt
```

That means the commit graph now has the following shape, including the contents of the merge commit:

Figure 17.2. History following a successful merge



What appears to have happened is that the change introduced by commit *C* has been applied to the tree of commit *B*. Commit *C* changes *g.txt* from 1 to 3, or using our tree diff data structure, $\{ \text{g.txt} \Rightarrow [1, 3] \}$. This can be applied to the tree $\{ \text{f.txt} \Rightarrow 2, \text{g.txt} \Rightarrow 1 \}$ to produce $\{ \text{f.txt} \Rightarrow 2, \text{g.txt} \Rightarrow 3 \}$, which is the tree for commit *M*.

17.1.2. Merging a chain of commits

The previous example suggests that the effect of the `merge` command is to apply the change introduced by the given commit to the current `HEAD`. But, things are a little more complicated than that. To see how, let's generate a slightly longer commit history.

```
$ git init git-merge-chain
$ cd git-merge-chain

$ echo "1" > f.txt
$ echo "1" > g.txt
$ echo "1" > h.txt
$ git add .
$ git commit --message "A"

$ echo "2" > f.txt
$ echo "3" > g.txt
$ git commit --all --message "B"

$ echo "4" > f.txt
$ git commit --all --message "C"

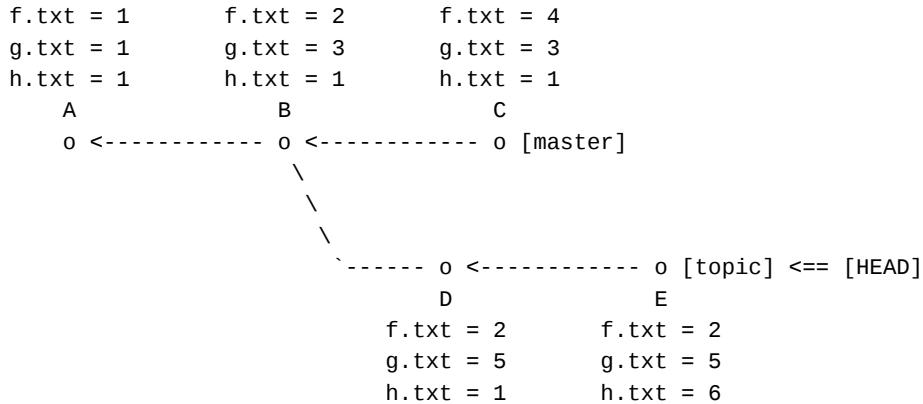
$ git checkout -b topic master^

$ echo "5" > g.txt
$ git commit --all --message "D"

$ echo "6" > h.txt
$ git commit --all --message "E"
```

These commands produce the following history:

Figure 17.3. Two branches changing multiple files



Again, we'll switch back to the master branch and merge topic.

```

$ git checkout master
$ git merge topic --message "M"

Merge made by the 'recursive' strategy.
  g.txt | 2 +-
  h.txt | 2 +-
  2 files changed, 2 insertions(+), 2 deletions(-)
  
```

The merge command tells us that files g.txt and h.txt have been updated. The trees of commits B, D, E and M are shown below. Commit B changes f.txt to 0cfbf08... and g.txt to 00750ed.... Commit D changes g.txt to 7ed6ff8.... Commit E changes h.txt to 1e8b314....

```

# commit B
$ git cat-file -p @~2^{tree}
100644 blob 0cfbf08886fca9a91cb753ec8734c84fcbe52c9f      f.txt
100644 blob 00750edc07d6415dcc07ae0351e9397b0222b7ba      g.txt
100644 blob d00491fd7e5bb6fa28c517a0bb32b8b506539d4d      h.txt

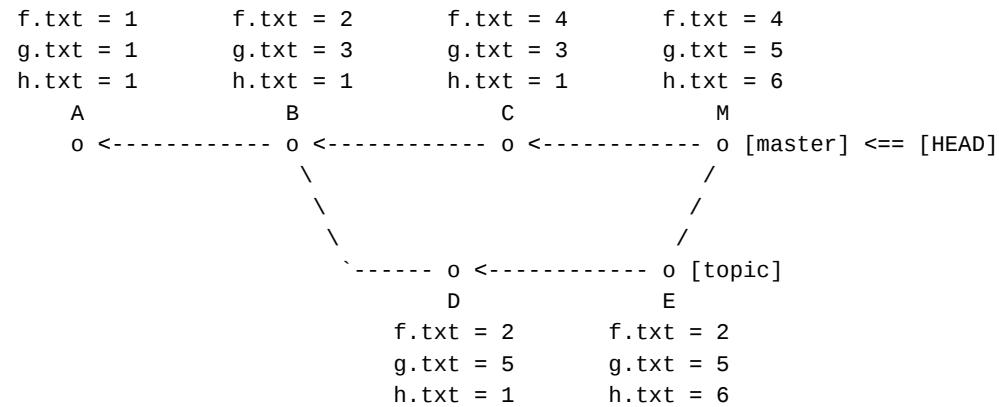
# commit D, changes g.txt
$ git cat-file -p @^2~1^{tree}
100644 blob 0cfbf08886fca9a91cb753ec8734c84fcbe52c9f      f.txt
100644 blob 7ed6ff82de6bcc2a78243fc9c54d3ef5ac14da69      g.txt
100644 blob d00491fd7e5bb6fa28c517a0bb32b8b506539d4d      h.txt

# commit E, changes h.txt
$ git cat-file -p @^2^{tree}
100644 blob 0cfbf08886fca9a91cb753ec8734c84fcbe52c9f      f.txt
100644 blob 7ed6ff82de6bcc2a78243fc9c54d3ef5ac14da69      g.txt
100644 blob 1e8b314962144c26d5e0e50fd29d2ca327864913      h.txt

# commit M
$ git cat-file -p @^{tree}
100644 blob b8626c4cff2849624fb67f87cd0ad72b163671ad      f.txt
100644 blob 7ed6ff82de6bcc2a78243fc9c54d3ef5ac14da69      g.txt
100644 blob 1e8b314962144c26d5e0e50fd29d2ca327864913      h.txt
  
```

Looking at the final tree for commit M, we see the full history looks like this:

Figure 17.4. History after merging a chain of commits



When we ran `git merge topic`, the current HEAD commit was commit *C*. If we compare commit *M* to *C*, we see that it's not just the change from commit *E* that's been applied; commit *E* only changes *h.txt* but the merge has also introduced a change to *g.txt*. That change came from commit *D*. So we can see that `merge` does not just incorporate the given commit, but also some of the history leading to that commit.

How far back in the history does `merge` go? We've seen that running `git merge topic` has included the changes from *E* and its parent *D*, but what about *B*? Commit *B* changes both *f.txt* and *g.txt*, but its contents for these files do not appear in the merge commit *M*. The merge commit uses the contents of *C* for *f.txt* and the *D* version of *g.txt*. It appears that `merge` has included changes from *E* and from *D*, but not from *B*.

What has happened is that `merge` has incorporated the changes from every commit that's reachable from `topic` but not from `HEAD`, i.e. the revision range `..topic`. It's identified that this range consists of the commits *D* and *E*, and applied the changes contained in both these commits to the tree of `HEAD` to form the merged tree.

Another way of looking at this is that `merge` identifies the nearest commit that's reachable from both `HEAD` and the merged branch, which in this case is commit *B*. This commit is what Git calls the *best common ancestor* or the *merge base* of *C* and *E*, and it represents the point at which the history of the two branches diverged. It takes all the changes committed on the merged branch since that commit, and applies them to `HEAD`.

17.1.3. Interpreting merges

Let's take a moment to consider why `merge` works the way it does. We've only identified what the `merge` command appears to do, but not why such an operation is useful or what it represents to users of Git.

In a centralised version control system, like Subversion², contributors can only make commits by pushing their work to a central repository. The repository collates the commits from all contributors into a single linear sequence, and in order to push a commit your changes must apply against the latest version. But in a distributed system like Git, there is no single 'latest version' of the code when you write a commit; two contributors might start their work from

²<https://subversion.apache.org/>

a shared copy of the same commit, but after that they are generating their own independent commit sequences. There is not a single linear history, but a network of forking paths as each contributor goes off in their own direction.

Suppose that our example represents such a situation. Two contributors, Alice and Bob, both have a copy of the project at commit B ; we can view this commit as representing the last time Alice and Bob agreed on the state of the project. Working independently on their own computers, Alice creates commit C , and Bob writes commits D and E . They now want to sync their changes back up to produce a single unified version. Given just commits C and E , it's not obvious which versions of each file we should use to build the unified version; Alice and Bob have different content for every file in the project.

Figure 17.5. Alice and Bob's trees before the merge

Tree C:	f.txt = 4 g.txt = 3 h.txt = 1	Tree E:	f.txt = 2 g.txt = 5 h.txt = 6
---------	-------------------------------------	---------	-------------------------------------

To resolve matters, Alice and Bob take the last commit upon which they both based their work — commit B — and work out what they've each changed since then.

Figure 17.6. Calculating Alice and Bob's changes since they diverged

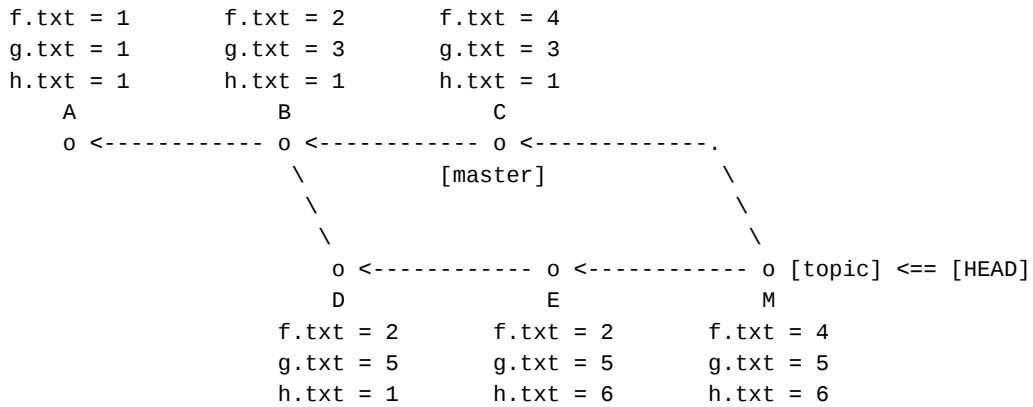
Tree B:	f.txt = 2 g.txt = 3 h.txt = 1		
Alice's changes:		Bob's changes:	
C - B = { f.txt => [2, 4] }		E - B = { g.txt => [3, 5] h.txt => [1, 6] }	

Since Alice and Bob have changed different sets of files, we can build the final tree by simply adding these differences to tree B , which produces the final version we see in tree M . This operation is called a *three-way merge*, and it's commonly used in distributed systems where multiple actors can modify a shared copy of a resource. To achieve consensus, they compare their changes to the last version they agreed on, and add those changes together.

If we adopt the notation T_X to refer to the tree of commit X , and $d_{XY} = T_Y - T_X$ to refer to the difference between the trees of commits X and Y , then we can say that: $T_M = T_B + d_{BC} + d_{BE}$. That is, the tree of M is the tree of B , plus the differences from each side of the merge. But, d_{BC} is the same as $T_C - T_B$, so $T_M = T_C + d_{BE}$. That is, when we have commit C checked out, the current tree already incorporates the difference between B and C , and we can build the tree for M by adding in the difference between B and E .

But, the above reasoning also implies that $T_M = T_E + d_{BC}$, since $d_{BE} = T_E - T_B$. That means we should get the same merge result if we have commit E checked out and apply the difference between B and C , that is we perform the following merge:

Figure 17.7. Symmetric merge result



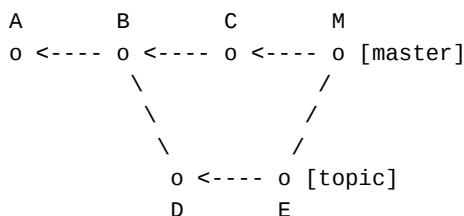
A quick inspection shows that commit *M* does indeed contain the tree from *E*, with the difference between *B* and *C* added in. The result of a merge should be symmetric with respect to Alice's and Bob's changes. In fact, the resulting commit only differs in the order of its parent fields, which will become important later.

Now, there is no guarantee that the tree generated by this procedure is *semantically valid*. Although Alice and Bob have changed different files, Bob might have changed a file that Alice's changes rely on, in such a way that Alice's changes no longer make sense. The `merge` command only represents a viable way to merge tree structures when parties have changed different parts of the tree; it automates the process of combining changes, but does not ensure that the resulting tree *works*. For that, we still need code review, automated testing, and other quality assurance measures. However, this procedure is a reasonable bet given how most software projects are organised; keeping code for independent functionality in separate files means people can work on separate parts of the software while easily merging their changes together.

Finally, it's worth remarking on why merge commits have multiple parents. So far, the parent pointer has only been used for identifying commits, as in the `<rev>^` and `<rev>~<n>` notations, and for traversing the history for the `log` command. That is, it's only used for *informational* purposes, to display information to the user and let them refer to commits without knowing their ID. However, the parent field now plays a role in determining the content of new commits, because it's used to determine where two branches diverged and therefore what changes to apply when performing a merge.

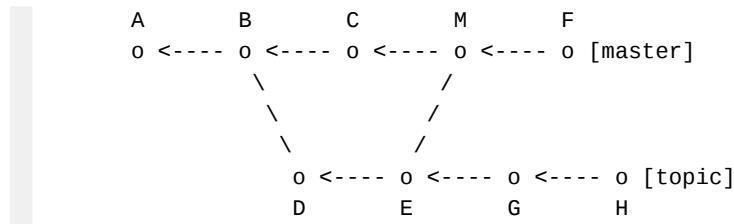
Consider our example history up to the merge commit *M*.

Figure 17.8. History leading to a merge commit



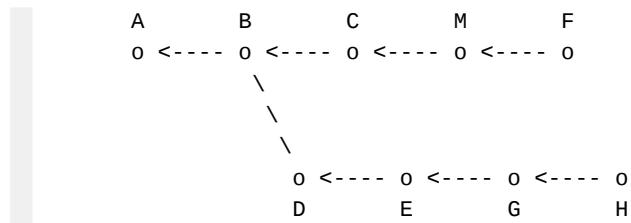
Now suppose we make some more commits on `master` and `topic`, and we want to merge these additional commits on `topic` into `master`:

Figure 17.9. Merged history with further commits



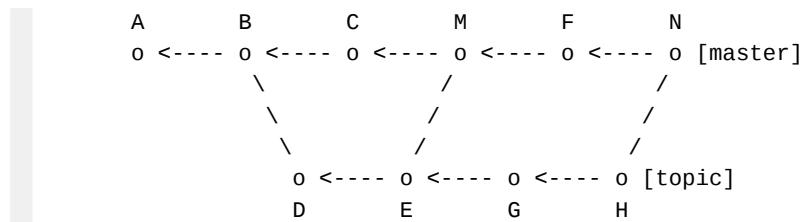
Given this history, we can identify that the latest commit that *F* and *H* have in common is commit *E* — the source of the last merge from topic. If this link between *M* and *E* did not exist, the best common ancestor of *F* and *H* would be *B*:

Figure 17.10. Merged history without merged parent links



If we pick *B* as the base of the merge, we would then attempt to apply the changes from *D*, *E*, *G* and *H* to commit *F*. But commit *F* already incorporates the changes from *D* and *E*, because it's derived from the merge commit *M*. Attempting to reapply these changes to *F* would at best be a waste of time, and at worst could result in conflicts or in overwriting more recent changes to the files affected by *D* and *E*. Keeping the parent pointer from *M* to *E* means we can use *E* as the base of the merge, and only apply the changes from *G* and *H* when building the merge commit *N*.

Figure 17.11. History following multiple merges



So having multiple parents is not just a means to show the user the full history in the `log` command. It also means that Git can support multiple merges from one branch to another, only considering new unmerged changes each time. It means Git has an accurate model of which changes have been incorporated into which commits, purely in terms of the history graph, without examining the content of those commits, and this model helps it merge only those changes that are not yet incorporated.

17.2. Finding the best common ancestor

Our first task when performing a merge is to find the best common ancestor (BCA) between the current `HEAD` commit, and the argument to the `merge` command. When the history does not contain merges and every commit has a single parent, that means there's a unique path from

a commit to any of its ancestors, and any pair of commits will have a unique BCA. To keep things simple, we'll only consider this type of history for now.

The process for finding $\text{BCA}(x, y)$ is much like the graph search techniques we saw in the last chapter while developing the `RevList` class. To find the BCA between two commits, we mark one commit with the flag `:parent1` and the other with `:parent2`. We then track back following their parent pointers, propagating the flags from each commit to its parent. As soon as we find a commit that's marked with both flags, we can return it.

Let's create a class for performing this search. Like `RevList`, the `Merge::CommonAncestors` class has a `@flags` store and a date-prioritised `@queue` of commits it needs to process. It stores the `:parent1` flag for one of its inputs and `:parent2` for the other. This class only takes commit IDs, not revision strings, and so it only needs access to a `Database`, not a full `Repository`.

```
# lib/merge/common_ancestors.rb

module Merge
  class CommonAncestors

    def initialize(database, one, two)
      @database = database
      @flags    = Hash.new { |hash, oid| hash[oid] = Set.new }
      @queue   = []
    end

    def insert_by_date(@queue, @database.load(one)) @flags[one].add(:parent1)

    insert_by_date(@queue, @database.load(two))
    @flags[two].add(:parent2)
  end

    def insert_by_date(list, commit)
      index = list.find_index { |c| c.date < commit.date }
      list.insert(index || list.size, commit)
    end

    #
  end
end
```

To find the BCA, we enter a loop. As long as there are any items left in the queue, we shift the next one off and examine its flags. If it has both the `:parent1` and `:parent2` flags, then we return its ID. Otherwise, we add the commit's flags to those of its parent. Unless the parent already has all the propagated flags, we add it to the queue for processing.

```
# lib/merge/common_ancestors.rb

BOTH_PARENTS = Set.new([:parent1, :parent2])

def find
  until @queue.empty?
    commit = @queue.shift
    flags = @flags[commit.oid]

    return commit.oid if flags == BOTH_PARENTS
```

```
    add_parents(commit, flags)
  end
end

def add_parents(commit, flags)
  return unless commit.parent

  parent = @database.load(commit.parent)
  return if @flags[parent.oid].superset?(flags)

  @flags[parent.oid].merge(flags)
  insert_by_date(@queue, parent)
end
```

Note that although this routine uses commit dates to prioritise the search, it does not rely on dates for correctness; it doesn't use dates to decide when to stop searching. Prioritising by date just means this algorithm will usually take the fewest steps to find the BCA, since it should have a date that's earlier than any of its descendants. Whereas the log command uses some heuristics for the sake of performance, we don't want the result of a merge to vary just because some commits have odd timestamps: it should depend only on the shape of the history graph.

17.3. Commits with multiple parents

The other functional change we need in order to generate merge commits is that the `Database::Commit` class currently only supports having a single parent ID, whereas we need it to have possibly many such pointers. We'll replace its `parent` attribute with `parents` instead:

```
# lib/database/commit.rb

attr_reader :parents

def initialize(parents, tree, author, message)
  @parents = parents
  #
end
```

We need to amend `Database::Commit#to_s` so that it writes out all its parent IDs, rather than writing only one if present.

```
# lib/database/commit.rb

def to_s
  lines = []

  lines.push("tree #{ @tree }")
  lines.concat(@parents.map { |oid| "parent #{ oid }" })
  #

  lines.join("\n")
end
```

Likewise, we need to adjust the `Database::Commit.parse` method so that it can read these multi-parent commits back from disk correctly, by processing multiple headers with the same name. Currently, such a commit would result in the last value for a header overwriting all the previous ones. We can fix things by collecting an array of values for each header, then using

only the first occurrence for headers that should appear once, and using the whole array for headers like parent that can have multiple values.

```
# lib/database/commit.rb

def self.parse(scanner)
  headers = Hash.new { |hash, key| hash[key] = [] }

  loop do
    line = scanner.scan_until(/\n/).strip
    break if line == ""

    key, value = line.split(/ +/, 2)
    headers[key].push(value)
  end

  Commit.new(
    headers["parent"],
    headers["tree"].first,
    Author.parse(headers["author"].first),
    scanner.rest)
end
```

There is currently a lot of code, notably the `Merge::CommonAncestors` class we just implemented, as well as `Revision`³ and `RevList`⁴, that still expects `Commit` objects to have a single `parent` method that returns an object ID or `nil`. Rather than amend all those classes immediately, we can keep them working with existing histories by implementing `Commit#parent` to return the first parent. In some cases this fully preserves correct behaviour; the `<rev>^` revision notation specifically means the first parent. Other use cases like `RevList` will need to be properly amended once we can write merge commits.

```
# lib/database/commit.rb

def parent
  @parents.first
end
```

Since the `merge` command will need to be able to generate commits, we should extract the logic for doing this from the `Command::Commit` class⁵. If we extract a method called `write_commit` that takes a list of parent IDs and a message string, we can call this from the `commit` and `merge` code to store the current state of the index as a commit and update the `HEAD` pointer.

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

module Command
  module WriteCommit

    def write_commit(parents, message)
      tree = write_tree
      name = @env.fetch("GIT_AUTHOR_NAME")
      email = @env.fetch("GIT_AUTHOR_EMAIL")
      author = Database::Author.new(name, email, Time.now)
    end
  end
end
```

³Section 13.3, “Setting the start point”

⁴Section 16.2.1, “Revision lists”

⁵Section 8.2.1, “Injecting dependencies”

```
commit = Database::Commit.new(parents, tree.oid, author, message)
repo.database.store(commit)
repo.refs.update_head(commit.oid)

commit
end

def write_tree
  root = Database::Tree.build(repo.index.each_entry)
  root.traverse { |tree| repo.database.store(tree) }
  root
end

end
end
```

The `Command::Commit` class is now reduced to reading the parent ID from `.git/HEAD`, getting the message from standard input, and then calling `write_commit`.

```
# lib/command/commit.rb

include WriteCommit

def run
  repo.index.load

  parent = repo.refs.read_head
  message = @stdin.read
  commit = write_commit([*parent], message)

  is_root = parent.nil? ? "(root-commit) " : ""
  puts "[#{is_root}#{commit.oid}] #{message.lines.first}"
  exit 0
end
```

17.4. Performing a merge

Once the code for finding the best common ancestor and writing multi-parent commits is in place, performing a merge is mostly a case of glueing existing functions together. The full code for the initial version of the `Command::Merge` class is shown below.

It begins by finding the IDs of the commits on each side of the merge: `head_oid` is the current `HEAD` commit, and `merge_oid` is the commit identified by the argument to the `merge` command. From these two IDs, we can use `Merge::CommonAncestors` to find the best common ancestor of the two commits. As Jit has not been able to generate merge commits until now, there will be a single unique BCA for the two commits; we'll call this `base_oid`.

Once the parameters of the merge are known, we can apply its effects. We load the index for updates, and then use the tools we developed for the `checkout` command⁶ to update the index and workspace. We calculate the tree diff between `base_oid` and `merge_oid`, which represents the net change on the merged branch since it diverged from the history of `HEAD`. The `Repository::Migration` class will write these changes to the project, and this will leave the

⁶Chapter 14, *Migrating between trees*

index in a state where it contains the tree of the merge commit — the tree of HEAD, plus the changes from the merged branch. As the migration affects both the index and the workspace, it will ensure the workspace reflects the contents of the index so the merge commit is effectively checked out.

At this point, we can write the updated index to disk, and write a commit from its contents, giving the head_oid and merge_oid as its parents.

```
# lib/command/merge.rb

module Command
  class Merge < Base

    include WriteCommit

    def run
      head_oid = repo.refs.read_head
      revision = Revision.new(repo, @args[0])
      merge_oid = revision.resolve(Revision::COMMIT)

      common = ::Merge::CommonAncestors.new(repo.database, head_oid, merge_oid)
      base_oid = common.find

      repo.index.load_for_update

      tree_diff = repo.database.tree_diff(base_oid, merge_oid)
      migration = repo.migration(tree_diff)
      migration.apply_changes

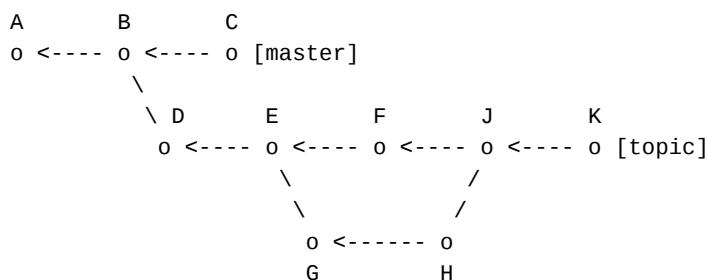
      repo.index.write_updates

      message = @stdin.read
      write_commit([head_oid, merge_oid], message)

      exit 0
    end
  end
end
```

Notice that we don't use RevList to go over the entire history from base_oid to merge_oid and apply each commit individually; we calculate the *net difference* over all the merged commits. This reduces the amount of work we need to do, and also removes a lot of potential complexity. If the merged history itself contains branching and merging paths, we'd need to decide which order to apply the commits in, and this could cause conflicts. For example, consider this history:

Figure 17.12. Branch containing fork/merge bubbles



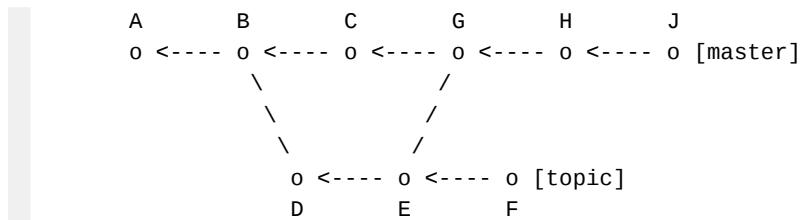
If we have `master` checked out and we want to merge `topic` in, the result could change depending on the order we apply commits F , G and H , especially if they modify the same files. It's better to assume any conflicts on these side branches were resolved in commit J than to try to reconstruct a reasonable application order for these changes. When performing a merge, we only consider the end state of the branch from B to K , and ignore how the branch got into that state.

17.5. Best common ancestors with merges

Putting a functioning `merge` command together turned out to be fairly straightforward, but its existence has introduced some new complexity. Until now, all commits that Jit could generate had at most one parent, and so the commit history of any project written with it would be a *tree*⁷. The history could diverge via the `branch` command, but divergent paths could not merge back together. One consequence of this is that there is a unique best common ancestor for any pair of commits in the history; the paths from the root commit to each commit diverge at exactly one point. We took advantage of this fact to simplify our first implementation of `Merge::CommonAncestors`.

However, in the presence of merge commits, the history forms a *directed acyclic graph*⁸ (DAG). That means there could be multiple paths from one commit to another, but you cannot have a commit being an ancestor of itself — you can't end up back where you started by following parent links. Consider the following history in which commit G has two parents, C and E . We have `master` checked out and we want to merge the `topic` branch.

Figure 17.13. Branching/merging history



Our current implementation only follows the first parent of each commit, and so it will ignore that link from G to E . It will therefore pick B as $\text{BCA}(J, F)$, rather than E , and the `merge` command will attempt to reapply the changes from commits D and E when it should only be using the changes from F .

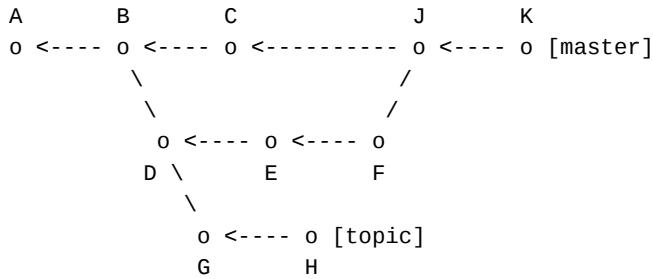
For histories with merges, we need to improve our definition of *best common ancestor*. A *common ancestor* of commits X and Y is any commit that is reachable from both X and Y . In the above history, commits A , B , D and E are all common ancestors of J and F . A *best common ancestor* of commits X and Y is any common ancestor of X and Y that is not an ancestor of any other common ancestor. In our example, commits A , B and D are all ancestors of commit E , and so E is the best common ancestor of J and F . In general, two commits may have more than one BCA, but we'll focus on cases where there's only one for now.

When searching for a BCA with merges, we can't assume the first commit we find with both flags is the BCA. For example, consider this history, in which $\text{BCA}(K, H) = D$.

⁷[https://en.wikipedia.org/wiki/Tree_\(graph_theory\)](https://en.wikipedia.org/wiki/Tree_(graph_theory))

⁸https://en.wikipedia.org/wiki/Directed_acyclic_graph

Figure 17.14. History with many candidate common ancestors



Even if our algorithm follows all the parent pointers for each commit, it may still end up identifying *B* as the BCA simply because of the order the commits are visited in — *B* is three steps away from both *K* and *H*, but *D* is four steps from *K* and so it may be visited later. Depending on the timestamps of the commits, this situation can occur even in simpler histories.

To solve this problem, two countermeasures are employed. First, rather than stopping as soon as we find a commit with both flags, we instead maintain a list of every such commit we find. When we find a candidate result, we add the flag `:result` to it, and we add the flag `:stale` to the set of flags we propagate to its parents. This means ancestors of result commits are excluded from the results, and we can stop searching once the queue contains only commits marked as `:stale`. However, in some cases this will still result in non-BCA commits being found. So the second countermeasure is that we apply `CommonAncestors` again to every pair of result commits to discover which are ancestors of the others, and remove them from the result set.

For example, suppose that `CommonAncestors` applied to commits *K* and *H* above produces two results: *B* and *D*. We'd then apply `CommonAncestors` to those commits and check the flags once a result is found. We would see that *B* has been flagged as `:parent2`, meaning it's reachable from *D*, and is thus not a BCA.

Turning these changes into code, let's begin by adding a `@results` array to the `CommonAncestors` class.

```
# lib/merge/common_ancestors.rb

def initialize(database, one, two)
    # ...
    @results = []
    # ...
end
```

We'll then modify the `find` method to process items from the queue until all the queued commits are marked `:stale`. The `process_queue` method takes a commit from the front of the queue as before, but now, if the commit has the `:parent1` and `:parent2` flags and no others, we mark it with the `:result` flag, and add `:stale` to the set of flags added to the parents. When the search is complete, `find` returns those result commit IDs that are not marked as `:stale`.

```
# lib/merge/common_ancestors.rb

def find
    process_queue until all_stale?
    @results.map(&:oid).reject { |oid| marked?(oid, :stale) }
end
```

```
private

def all_stale?
  @queue.all? { |commit| marked?(commit.oid, :stale) }
end

def marked?(oid, flag)
  @flags[oid].include?(flag)
end

def process_queue
  commit = @queue.shift
  flags = @flags[commit.oid]

  if flags == BOTH_PARENTS
    flags.add(:result)
    insert_by_date(@results, commit)
    add_parents(commit, flags + [:stale])
  else
    add_parents(commit, flags)
  end
end
```

The `add_parents` method currently follows only the first parent of the commit. This can be changed to iterate over all the commit's parents instead.

```
# lib/merge/common_ancestors.rb

def add_parents(commit, flags)
  commit.parents.each do |parent|
    next if @flags[parent].superset?(flags)

    @flags[parent].merge(flags)
    insert_by_date(@queue, @database.load(parent))
  end
end
```

Now that `CommonAncestors` can explore the complete graph, we can layer another class over it that filters out results that are ancestors of the others. The `Merge::Bases` class will take a `Database` and two commit IDs, and set up a `CommonAncestors` instance to analyse them.

```
# lib/merge/bases.rb

module Merge
  class Bases

    def initialize(database, one, two)
      @database = database
      @common   = CommonAncestors.new(@database, one, [two])
    end
  end
end
```

The `Bases#find` method will return an array of commit IDs, just like `CommonAncestors#find`. It begins by using `CommonAncestors` on its two input commits, and returning the result if at most one commit is found. Otherwise, it attempts to filter each commit out of the result set by adding it to the `@redundant` set, returning the original results with the redundant IDs removed.

```
# lib/merge/bases.rb
```

```

def find
  @commits = @common.find
  return @commits if @commits.size <= 1

  @redundant = Set.new
  @commits.each { |commit| filter_commit(commit) }
  @commits - @redundant.to_a
end

```

Bases#filter_commit works by comparing the given commit ID to all the others in the @commits set that are not already in the @redundant set. It calls CommonAncestors with the given ID as the :parent1 input and all the other commits as the :parent2 input, and runs the find method to cause a scan of the history. If we find that the given commit is marked as :parent2, or that any of the other commits are marked :parent1, then we add them to the @redundant set since one was found to be reachable from the other.

```

# lib/merge/bases.rb

def filter_commit(commit)
  return if @redundant.include?(commit)

  others = @commits - [commit, *@redundant]
  common = CommonAncestors.new(@database, commit, others)

  common.find

  @redundant.add(commit) if common.marked?(commit, :parent2)

  others.select! { |oid| common.marked?(oid, :parent1) }
  @redundant.merge(others)
end

```

Notice that we're now passing an array rather than a single ID as the :parent2 argument to CommonAncestors. This can be easily accommodated by tagging all the IDs in this list as :parent2 and then running the search as before.

```

# lib/merge/common_ancestors.rb

def initialize(database, one, twos)
  # ...

  insert_by_date(@queue, @database.load(one))
  @flags[one].add(:parent1)

  twos.each do |two|
    insert_by_date(@queue, @database.load(two))
    @flags[two].add(:parent2)
  end
end

```

Finally, switching the Command::Merge class over to use Merge::Bases instead of Merge::CommonAncestors keeps merges working when the history itself contains merges.

```

# lib/command/merge.rb

common = ::Merge::Bases.new(repo.database, head_oid, merge_oid)

```

```
base_oid = common.find.first
```

Although `Bases#find` can return multiple results, we won't deal with that case here. Git uses a *recursive* strategy to resolve such situations: if $\text{BCA}(X, Y)$ returns two commits, call them P and Q , then it starts the merge process all over again with P and Q as the inputs. If those commits have a unique BCA , then Git merges them as we've seen above, producing a *virtual commit* V . This commit V is then used as the base for a three-way merge between X and Y .

17.6. Logs in a merging history

In Section 17.3, “Commits with multiple parents” we amended the `Database::Commit` class so that its `parent` method returned its first parent ID, which was enough to keep the `log` command and underlying `RevList` class working on tree-shaped histories. However, to properly represent histories containing merges, we need to make a few amendments to the `RevList` class.

17.6.1. Following all commit parents

When we encounter a merge commit, we should follow all the parents that lead to it, rather than only displaying the history from the first parent. This can be done by changing the `RevList#add_parents` method to iterate over the commit's parent IDs.

```
# lib/rev_list.rb

def add_parents(commit)
  return unless mark(commit.oid, :added)

  parents = commit.parents.map { |oid| load_commit(oid) }

  if marked?(commit.oid, :uninteresting)
    parents.each { |parent| mark_parents_uninteresting(parent) }
  else
    simplify_commit(commit)
  end

  parents.each { |parent| enqueue_commit(parent) }
end
```

We also need to make sure multiple branches are followed when excluding a line of history using the `^<rev>` and `<rev>..<rev>` notations. This requires changing `mark_parents_uninteresting` so that it scans the entire loaded history back from the given commit. Because this history might be very large, we don't write this as a recursive function, as doing so could cause a stack overflow. Instead, we keep a queue of commits that need processing, and add commits to this queue as we walk down the history.

```
# lib/rev_list.rb

def mark_parents_uninteresting(commit)
  queue = commit.parents.clone

  until queue.empty?
    oid = queue.shift
    next unless mark(oid, :uninteresting)
```

```
    commit = @commits[oid]
    queue.concat(commit.parents) if commit
end
end
```

17.6.2. Hiding patches for merge commits

Displaying patches for merges is more complicated than displaying the diff for a single-parent commit. Our current merge implementation will only produce commits in which every file is exactly equal to a version from one of the parents, and in this situation Git produces no diff output. So for now, we will just avoid printing patches for merges entirely.

```
# lib/command/log.rb

def show_patch(commit)
  return unless @options[:patch] and commit.parents.size <= 1
  #
end
```

We'll return to this part of the codebase and complete its functionality after we're able to merge edits to the same file⁹.

17.6.3. Pruning treesame commits

When we pass in filenames as arguments to `log`, we simplify the history by excluding any commits that don't change the given files; such commits are marked `:treesame`. When we're considering a merge commit, it should only be considered interesting if its version of the files we're searching for is different from that of any of its parents. If we're looking for changes to `foo.txt`, and a merge has a parent with the same content for `foo.txt`, then either `foo.txt` was not changed on the merged branches, or the merge took its version of `foo.txt` from one of the merged commits.

Consider our example merge history from Section 17.1.2, “Merging a chain of commits”. In commit *M*, the file `g.txt` is the same as the version in commit *E* but differs from the version in commit *C*. That doesn't mean that *M* introduced the change to `g.txt` — this change is actually introduced in commit *D*, from which *M* derives its version of `g.txt`. So if a commit is treesame to any of its parents for the given files, we ignore it. If the given commit has no parents, then we compare it to the empty tree, which can be done by using `nil` as a parent ID.

```
# lib/rev_list.rb

def simplify_commit(commit)
  return if @prune.empty?

  parents = commit.parents
  parents = [nil] if parents.empty?

  parents.each do |oid|
    mark(commit.oid, :treesame) if tree_diff(oid, commit.oid).empty?
  end
end
```

⁹Section 20.2, “Logging merge commits”

17.6.4. Following only treesame parents

The final change required to `RevList` is perhaps slightly counter-intuitive. The purpose of this class when given filenames as inputs is to show the history that explains the current state of the named files. In our example history in Section 17.1.2, “Merging a chain of commits”, the state of file `g.txt` in commit M is equal to that in commit E , not commit C . That means that, whatever changes to `g.txt` might have happened on the `master` branch, they were overwritten by the changes from the topic branch, and so the current state of `g.txt` can be explained by following only commit E and ignoring commit C .

We can achieve this by having `simplify_commit` return a list of the parents that should be followed. If any treesame parent is found, then we immediately return only the ID of that parent, otherwise we return the commit’s full parent list.

```
# lib/rev_list.rb

def simplify_commit(commit)
  return commit.parents if @prune.empty?

  parents = commit.parents
  parents = [nil] if parents.empty?

  parents.each do |oid|
    next unless tree_diff(oid, commit.oid).empty?
    mark(commit.oid, :treesame)
  end

  commit.parents
end
```

In `add_parents`, we can then use the result of `simplify_commit` to decide which parents to add to the input queue.

```
# lib/rev_list.rb

def add_parents(commit)
  return unless mark(commit.oid, :added)

  if marked?(commit.oid, :uninteresting)
    parents = commit.parents.map { |oid| load_commit(oid) }
    parents.each { |parent| mark_parents_uninteresting(parent) }
  else
    parents = simplify_commit(commit).map { |oid| load_commit(oid) }
  end

  parents.each { |parent| enqueue_commit(parent) }
end
```

17.7. Revisions with multiple parents

The final change we should make to handle merge commits is to expand the revisions notation. The expressions `<rev>^` and `<rev>~<n>` still work correctly; they specifically follow the first parent pointer. But if you want to identify a different parent, there’s another expression you can use: `<rev>^<n>`. `<rev>^` is equivalent to `<rev>^1`, and `<rev>~3` is equivalent to `<rev>^1^1^1`.

To support this, the `Revision#commit_parent` method can take an optional argument that defaults to `1` to select the required parent.

```
# lib/revision.rb

def commit_parent(oid, n = 1)
  return nil unless oid

  commit = load_typed_object(oid, COMMIT)
  return nil unless commit

  commit.parents[n - 1]
end
```

The `Revision::Parent` struct needs an additional property to hold the number of the requested parent, and it should pass this into the `commit_parent` function.

```
# lib/revision.rb

Parent = Struct.new(:rev, :n) do
  def resolve(context)
    context.commit_parent(rev.resolve(context), n)
  end
end
```

And, the parser needs adjusting to capture the optional numeric argument to the parent operator, and feed it into the `Parent` struct, passing the value `1` if no parent number was given.

```
# lib/revision.rb

PARENT = /^(.+)\^(\d*)$/

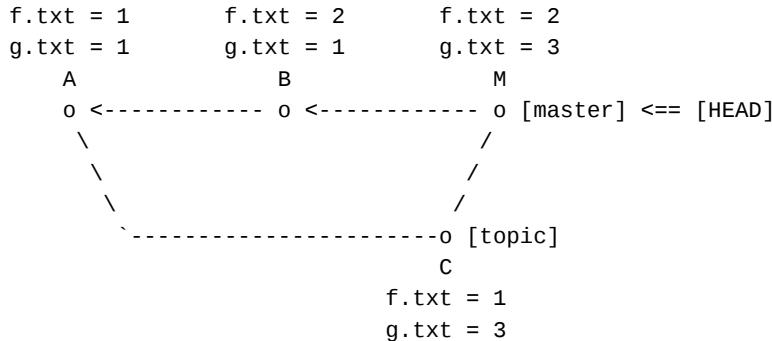
def self.parse(revision)
  if match = PARENT.match(revision)
    rev = Revision.parse(match[1])
    n = (match[2] == "") ? 1 : match[2].to_i
    rev ? Parent.new(rev, n) : nil
  elsif # ...
  end
```

The branch, checkout, log and merge commands can all now accommodate merge commits successfully, and we're ready to handle more complicated content merging situations.

18. When merges fail

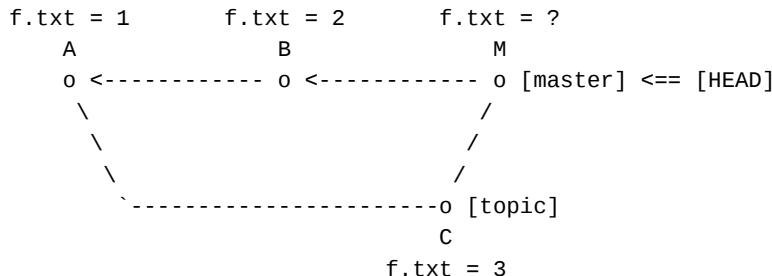
In the previous chapter we focussed on straightforward successful merges, and how they affect and are affected by the shape of the commit history graph. We studied merges where the two sides of the merge have changed different files, for example:

Figure 18.1. Merging branches changing different files



In this history, one side of the merge changes `f.txt` while the other changes `g.txt`. The changes do not overlap, and so they can be trivially added together to form the merged tree. Now, we want to look at what happens when changes overlap, for example:

Figure 18.2. Merging branches changing the same file



If both sides of the merge change `f.txt`, what should commit `M` contain? We'll get to that in due course, but in order to prepare for that, I'm going to do a little refactoring and handle a couple of edge cases in the `merge` command.

18.1. A little refactoring

In Section 17.4, “Performing a merge” we built the `Command::Merge` class with a monolithic `run` method that glues together all the machinery necessary to perform the merge. To support the work we’re about to do, it will be useful to start separating this code into distinct pieces. In particular, it will help if we separate the *inputs* of the merge — the names and IDs of the two branch heads, and the IDs of the best common ancestors (BCAs) — from its execution, that is the changes to the index and workspace resulting from the differences on the merged branch.

We’re going to restructure things so that `Command::Merge#run` looks as follows. It begins by building a `Merge::Inputs` object from the names of `HEAD` and the argument to the `merge` command; this identifies which commits are being merged and finds their BCA. It then loads the index, and spins up a `Merge::Resolve` object from the `Inputs` object; calling `execute` on this applies the necessary changes to the index and workspace. Finally, we save the index and write a commit as before, getting the parent commit IDs from the `Inputs` instance.

```
# lib/command/merge.rb

def run
  @inputs = ::Merge::Inputs.new(repo, Revision::HEAD, @args[0])
  resolve_merge
  commit_merge
  exit 0
end

private

def resolve_merge
  repo.index.load_for_update

  merge = ::Merge::Resolve.new(repo, @inputs)
  merge.execute

  repo.index.write_updates
end

def commit_merge
  parents = [@inputs.left_oid, @inputs.right_oid]
  message = @stdin.read
  write_commit(parents, message)
end
```

The `Merge::Inputs` class takes two revisions¹, which it refers to as *left* and *right*. It stores off these revision strings as the *names* of the two commits, and then resolves them to IDs. Finally it uses `Merge::Bases`² to determine the BCAs for the merge, and it exposes all these results through public attributes.

```
# lib/merge/inputs.rb

module Merge
  class Inputs

    attr_reader :left_name, :right_name,
                :left_oid, :right_oid,
                :base_oids

    def initialize(repository, left_name, right_name)
      @repo      = repository
      @left_name = left_name
      @right_name = right_name

      @left_oid  = resolve_rev(@left_name)
      @right_oid = resolve_rev(@right_name)

      common     = Bases.new(@repo.database, @left_oid, @right_oid)
      @base_oids = common.find
    end

    private

    def resolve_rev(rev)
```

¹Section 13.3, “Setting the start point”

²Section 17.5, “Best common ancestors with merges”

```
    Revision.new(@repo, rev).resolve(Revision::COMMIT)
  end

end
end
```

The job of the `Merge::Resolve` class is to take one of these `Inputs` objects and execute it, that is, reconcile the changes from both sides of the merge and apply them to the index and workspace. For now this just contains the relevant logic extracted from the `Command::Merge` class, handling only cases where a single BCA exists.

```
# lib/merge/resolve.rb

module Merge
  class Resolve

    def initialize(repository, inputs)
      @repo = repository
      @inputs = inputs
    end

    def execute
      base_oid = @inputs.base_oids.first
      tree_diff = @repo.database.tree_diff(base_oid, @inputs.right_oid)
      migration = @repo.migration(tree_diff)

      migration.apply_changes
    end
  end
end
```

This refactoring means we'll be able to execute a merge independently of how the input commits are selected, and also lets us ask interesting questions about the inputs.

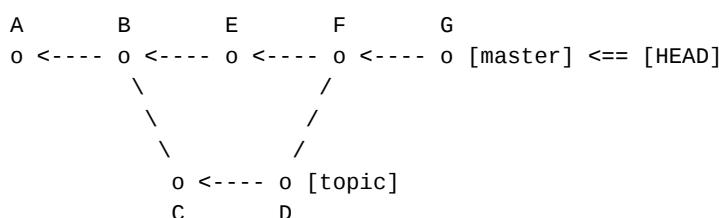
18.2. Null and fast-forward merges

There are two special cases that an `Inputs` object can be in that mean we don't actually need to perform a merge and write a new commit. First, the commit we've been asked to merge might already be an ancestor of the current `HEAD`, and second, the requested commit might be a descendant of `HEAD`.

18.2.1. Merging an existing ancestor

A *null merge* occurs when the requested merge commit is already reachable from the current `HEAD`. For example, consider this history, where the `master` branch is currently checked out.

Figure 18.3. History with topic already merged into master



In this situation, merging the topic branch means merging G and D . However, whereas in most merges the BCA is distinct from the two input commits, in this case $\text{BCA}(G, D) = D$. That is, the base of the merge is precisely the commit we've requested, and so there are no changes between the BCA and the merged commit.

Since the requested merge commit is reachable from `HEAD`, we assume that `HEAD` already incorporates its changes and so nothing needs to be done. To identify this scenario, we can add a method to `Merge::Inputs` that tells us whether the BCA is the same as the requested commit, which we call the *right* commit.

```
# lib/merge/inputs.rb

def already_merged?
  @base_oids == [@right_oid]
end
```

If this method returns true, the `Command::Merge` class can bail out before trying to make any changes.

```
# lib/command/merge.rb

def run
  @inputs = ::Merge::Inputs.new(repo, Revision::HEAD, @args[0])
  handle_merged_ancestor if @inputs.already_merged?

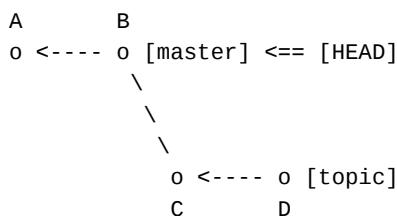
  #
end

def handle_merged_ancestor
  puts "Already up to date."
  exit 0
end
```

18.2.2. Fast-forward merge

A *fast-forward merge* represents the other BCA special case: rather than being equal to the requested commit, the BCA is equal to `HEAD`. Consider the following history, in which we have `master` checked out and ask to merge the `topic` branch.

Figure 18.4. History where `HEAD` is an ancestor of another branch



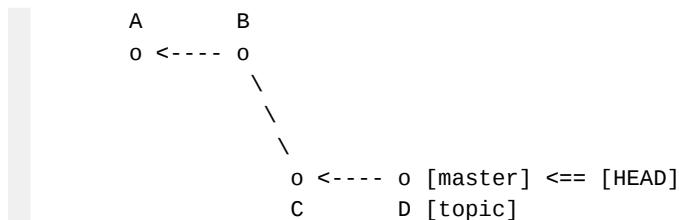
Recall our formula for computing a merge tree³: $T_M = T_B + d_{BL} + d_{BR}$, where L and R are the left and right input commits and B is their best common ancestor. But if the BCA is the same as the `HEAD` commit, that means that $B = L$ and d_{BL} is zero.

³Section 17.1.3, “Interpreting merges”

So, $T_M = T_B + 0 + d_{BR} = T_B + T_R - T_B = T_R$. That is, the tree of the merge does not have to be built by calculating a change set and applying it to some other tree. The required tree already exists: it's the tree of the merged commit.

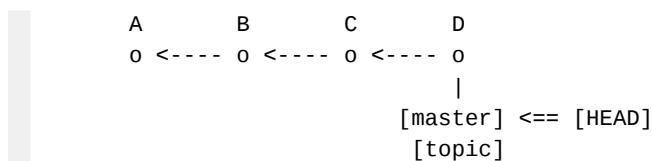
All we need to do in this situation is move the master branch pointer to refer to the merged commit, and check that commit out:

Figure 18.5. Fast-forwarding the HEAD branch



Or to draw things more simply:

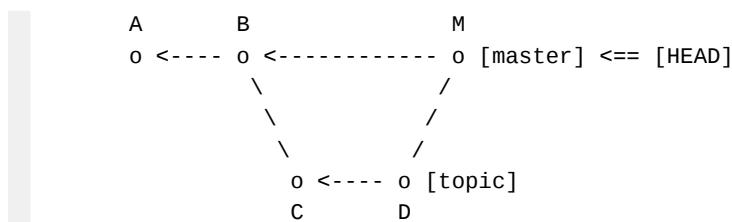
Figure 18.6. Flattened fast-forward result state



It's called a *fast-forward* because we're just shuttling the HEAD pointer along an existing history rather than performing a true merge, since there aren't any changes on HEAD that aren't already reachable from the merged commit.

If you use the `--no-ff` option, Git's `merge` command will construct a new merge commit where the left-hand side does not introduce any changes relative to the BCA, but this is not the default.

Figure 18.7. Forced merge commit created using --no-ff



Detecting this condition means checking whether the base ID for the merge is the same as the *left* ID, i.e. the current HEAD.

```
# lib/merge/inputs.rb

def fast_forward?
  @base_oids == [@left_oid]
end
```

In this event, we print a message saying we're performing a fast-forward, and then use the building blocks of tree diffs⁴ and migrations⁵ to check out the requested commit. Finally we update the current HEAD ref to point at this newly checked-out revision.

⁴Section 14.1, “Telling trees apart”

Section 14.1, Telling trees apart 5 Chapter 14, Migrating between trees

```
# lib/command/merge.rb

def run
  @inputs = ::Merge::Inputs.new(repo, Revision::HEAD, @args[0])
  handle_merged_ancestor if @inputs.already_merged?
  handle_fast_forward if @inputs.fast_forward?

  #
end

def handle_fast_forward
  a = repo.database.short_oid(@inputs.left_oid)
  b = repo.database.short_oid(@inputs.right_oid)

  puts "Updating #{a}..#{b}"
  puts "Fast-forward"

  repo.index.load_for_update

  tree_diff = repo.database.tree_diff(@inputs.left_oid, @inputs.right_oid)
  repo.migration(tree_diff).apply_changes

  repo.index.write_updates
  repo.refs.update_head(@inputs.right_oid)

  exit 0
end
```

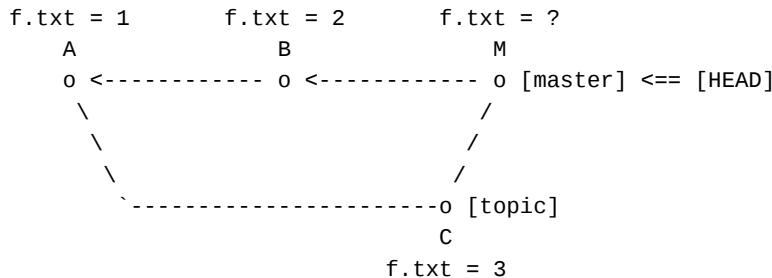
Since in this case, `@inputs.left_oid` is the same as `@inputs.base_oids[0]`, you might be wondering why we don't just use the `Merge::Resolve` class which effectively does the same tree-diff and migration work. The answer is that, in order to handle merge conflicts, that class is about to get a lot more complicated. While right now it happens to just check out the requested tree, it will shortly be doing more complex work that we don't want when faced with a fast-forward.

Just because two things happen to look the same, doesn't mean they really represent the same concept. In this case, by saying precisely that we just want to check out the merged commit and nothing more, we avoid opting in to any new behaviour the `Resolve` class might gain that we don't want. Having small building blocks is an important step in avoiding reusing code beyond its intended scope, which tends to cause more maintenance headaches than it saves.

18.3. Conflicted index entries

We can now start to talk about what happens when the changes on each side of a merge can't be easily resolved. We opened the chapter with the following example:

Figure 18.8. Merging branches changing the same file



We can recreate this situation by running the following commands. I'm using words for the contents of the files to distinguish these values from other appearances of the digits 1, 2 and 3 that will occur when we examine the index.

```
$ git init git-merge-edit-edit
$ cd git-merge-edit-edit

$ echo "one" > f.txt
$ git add .
$ git commit --message "A"

$ echo "two" > f.txt
$ git commit --all --message "B"

$ git checkout -b topic master^
$ echo "three" > f.txt
$ git commit --all --message "C"

$ git checkout master
$ git merge topic --message "M"
```

The merge command itself gives us a hint that something went wrong:

```
Auto-merging f.txt
CONFLICT (content): Merge conflict in f.txt
Automatic merge failed; fix conflicts and then commit the result.
```

18.3.1. Inspecting the conflicted repository

If we use Git's commands to inspect the state of the repository, we learn a few interesting things. First, it seems the merge commit M was not created, for `HEAD` still points at commit B .

```
$ git log --oneline  
df89aa9 (HEAD -> master) B  
9a14e0a A
```

Second, the status command is showing some new states that we haven't seen before, both in the long format and the porcelain one. It says the state of f.txt is 'both modified', denoted **UU** in the short format.

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
```

```
(use "git add <file>..." to mark resolution)

both modified: f.txt

no changes added to commit (use "git add" and/or "git commit -a")

$ git status --porcelain
UU f.txt
```

And, the file `f.txt` itself contains the text of both versions of the file that went into the merge, annotated with which commit each version came from.

```
$ cat f.txt

<<<<< HEAD
two
=====
three
>>>>> topic
```

Furthermore, trying to commit without making any changes results in an error:

```
$ git commit
U      f.txt
error: Committing is not possible because you have unmerged files.
hint: Fix them up in the work tree, and then use 'git add/rm <file>'
hint: as appropriate to mark resolution and make a commit.
fatal: Exiting because of an unresolved conflict.
```

All this suggests that Git did not know which version of the file to use, since both sides of the merge changed it. It's dumped both versions of the file into the workspace and refused to make any commits until we sort things out. But how does it represent the fact we're in a conflicted state?

18.3.2. Stages in the index

In Section 6.2, “Inspecting `.git/index`”, we examined the format of the `.git/index` file and we met the `ls-files` command, which with the `--stage` flag lists every file in the index along with its object ID and mode, as well as a *stage* number that we did not investigate further on first contact. Let's take a look at what it says now:

```
$ git ls-files --stage

100644 5626abf0f72e58d7a153368ba57db4c673c0e171 1      f.txt
100644 f719efd430d52bcfc8566a43b2eb655688d38871 2      f.txt
100644 2bdf67abb163a4ffb2d7f3f0880c9fe5068ce782 3      f.txt
```

It seems that `f.txt` is listed in the index three times, each with a different stage number. Let's inspect the blobs referenced by the object IDs of these entries:

```
$ git cat-file -p 5626abf0f72e58d7a153368ba57db4c673c0e171
one

$ git cat-file -p f719efd430d52bcfc8566a43b2eb655688d38871
two

$ git cat-file -p 2bdf67abb163a4ffb2d7f3f0880c9fe5068ce782
three
```

Each `f.txt` entry in the index points at a version of the file's contents. Stage 1 points to the version of the file from the base of the merge, stage 2 to the left-hand commit's version, and stage 3 to the right-hand's. It's the presence of these entries with a non-zero stage that means the repository is in a conflict state: there are multiple candidate versions for a file and we need to pick one before Git will let us commit.

When we were first exploring the index, we visited Git's index file format documentation⁶. Under the explanation of the fields in an index entry, one of the items is this:

```
A 16-bit 'flags' field split into (high to low bits)  
  
1-bit assume-valid flag  
1-bit extended flag (must be zero in version 2)  
2-bit stage (during merge)  
12-bit name length if the length is less than 0xFFFF; otherwise 0xFFFF  
is stored in this field.
```

Recalling our `Index::Entry` class, the `ENTRY_FORMAT` string unpacks the binary data in an entry into a Ruby struct. `N10` denotes ten 32-bit numbers, `H40` means a 40-digit hex string, `n` is a single 16-bit number, and `Z*` a null-terminated string. That `n` is the part of the entry we're interested in here, and its bits are broken down as described in the above snippet.

```
# lib/index/entry.rb  
  
ENTRY_FORMAT = "N10H40nZ*"  
  
entry_fields = [  
  :ctime, :ctime_nsec,  
  :mtime, :mtime_nsec,  
  :dev, :ino, :mode, :uid, :gid, :size,  
  :oid, :flags, :path  
]  
  
Entry = Struct.new(*entry_fields) do  
  # ...  
end
```

Here's the hexdump of the index in our example:

```
00000000  44 49 52 43 00 00 00 02  00 00 00 03 00 00 00 00 |DIRC.....|  
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |.....|  
00000020  00 00 00 00 00 81 a4 00 00 00 00 00 00 00 00 00 |.....|  
00000030  00 00 00 00 56 26 ab f0 f7 2e 58 d7 a1 53 36 8b |....V&....X..S6.|  
00000040  a5 7d b4 c6 73 c0 e1 71 10 05 66 2e 74 78 74 00 |..}..s..q..f.txt.|  
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |.....|  
00000060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 81 a4 |.....|  
00000070  00 00 00 00 00 00 00 00  00 00 00 00 f7 19 ef d4 |.....|  
00000080  30 d5 2b cf c8 56 6a 43 b2 eb 65 56 88 d3 88 71 |0.+..VjC..eV..q|  
00000090  20 05 66 2e 74 78 74 00 00 00 00 00 00 00 00 00 | .f.txt.....|  
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |.....|  
000000b0  00 00 00 00 00 81 a4 00 00 00 00 00 00 00 00 00 |.....|  
000000c0  00 00 00 00 2b df 67 ab b1 63 a4 ff b2 d7 f3 f0 |....+..g..c.....|  
000000d0  88 0c 9f e5 06 8c e7 82 30 05 66 2e 74 78 74 00 |.....0.f.txt.|  
000000e0  00 00 00 00 b6 bc 1a b6 2a 7c 9a bb c7 ab e4 e4 |.....*|.....|  
000000f0  fb 96 d7 ff 64 0b 54 57 00 00 00 00 00 00 00 00 |....d.TW|  
000000f8
```

⁶<https://git-scm.com/docs/index-format>

Notice that all the fields that come from `stat()` filesystem information are zero, because conflict candidate versions are not written to the workspace. The 16-bit *flags* fields are the two bytes appearing before the filename in each entry; here they are highlighted along with the object IDs and filenames:

```

00000030      56 26 ab f0  f7 2e 58 d7 a1 53 36 8b |   V&....X..S6.| 
00000040  a5 7d b4 c6 73 c0 e1 71  10 05 66 2e 74 78 74 | .}..s..q..f.txt | 
...
00000070                      f7 19 ef d4 |       ....| 
00000080  30 d5 2b cf c8 56 6a 43  b2 eb 65 56 88 d3 88 71 | 0.+..VjC..eV...q| 
00000090  20 05 66 2e 74 78 74 | .f.txt | 
...
000000c0      2b df 67 ab  b1 63 a4 ff b2 d7 f3 f0 |   +.g..c.....| 
000000d0  88 0c 9f e5 06 8c e7 82  30 05 66 2e 74 78 74 | .....0.f.txt | 

```

You can see the object IDs for each entry: 5626ab7..., f719efd..., 2bdf67a.... Immediately after those are the flags bytes, 10 05, 20 05, 30 05. A single hex digit represents four bits, and the filename length is the last 12 bits of this field, i.e. the last three hex digits. The leading digit contains the other flags: the *assume-valid* and *extended* flags, and the *stage*. If we write, say, 30 05 out in binary it breaks down like this:

Figure 18.9. Bitwise breakdown of the index entry flag bytes

0	0	11	000000000101
assume-valid flag	extended flag	stage	name length

When we use `String#unpack` to read this data into an object, it comes out as a single number in the `Index::Entry#flags` attribute. 3005_{16} and 001100000000101_2 are equal to $12,293_{10}$.

18.3.3. Storing entries by stage

Until now, entries in the index have been keyed by their path, such that only one entry for each filename can be present in the index. We now need them to be keyed by their path and stage, and sorted as such. In Ruby we do this by replacing `path` with `[path, stage]` as the key for an entry.

```

# lib/index/entry.rb

def key
  [path, stage]
end

```

In Ruby, arrays can be compared for sort order⁷; it works by comparing their individual elements, so, the index will still keep these entries sorted appropriately.

```

>> ["bar.txt", 3] <=> ["foo.txt", 2]
=> -1

>> ["foo.txt", 3] <=> ["foo.txt", 2]
=> 1

```

An entry's stage must be extracted from its `flags` number via a bit of bitwise arithmetic. We want to select the 13th and 14th bits from the right — the rightmost twelve digits are the

⁷<https://docs.ruby-lang.org/en/2.3.0/Array.html#method-i-3C-3D-3E>

filename length. We can do this by bit-shifting⁸ the value right by twelve places, and then selecting the two least significant digits of the result; $0x3$ is 11_2 .

```
# lib/index/entry.rb

def stage
  (flags >> 12) & 0x3
end
```

For example, 12,293 is 001100000000101_2 , left-padded to 16 digits. Bit-shifting twelve places to the right gives 0011_2 , and we select the rightmost two bits using a bitwise-and: $0011_2 \& 3_{16} = 0011_2 \& 11_2 = 11_2 = 3_{10}$ —this entry’s stage is 3. If you’re unfamiliar with operations on binary numbers, refer to Appendix B, *Bitwise arithmetic*.

To handle the change in how `Index::Entry#key` works, a few of the methods in `Index` need to be amended. The `entry_for_path` method will now take an optional `stage` argument that defaults to zero, and `tracked_file?` will check if any entries exist for the given file at any of the possible stages from 0 to 3.

```
# lib/index.rb

def entry_for_path(path, stage = 0)
  @entries[[path.to_s, stage]]
end

def tracked_file?(path)
  (0..3).any? { |stage| @entries.has_key?([path.to_s, stage]) }
end
```

The `remove_entry` method will likewise iterate over all the possible stages, removing any entries for the given pathname. It delegates to a new method called `remove_entry_with_stage`, which contains the original `remove_entry` functionality but now looks up the requested entry using a path and a stage number.

```
# lib/index.rb

def remove_entry(pathname)
  (0..3).each { |stage| remove_entry_with_stage(pathname, stage) }
end

def remove_entry_with_stage(pathname, stage)
  entry = @entries[[pathname.to_s, stage]]
  # ...
end
```

These changes are enough to keep the index working for all existing states; in normal operation all the index entries have stage 0.

18.3.4. Storing conflicts

In order to represent conflicted files in the index, we need to evict any entries with stage 0 for the given path, and then add entries with stages 1, 2 and 3 representing the tree items from the merge base and the left and right commits. The index should never contain both zero- and non-

⁸https://en.wikipedia.org/wiki/Bitwise_operation

zero-staged entries for the same path; a stage-0 entry means the file is clean while any other stage present means the file is conflicted.

Mirroring the `Index#add` method that backs the `add` command, we can add a new method `Index#add_conflict_set`. Instead of taking an object ID and a `File::Stat` for a blob that's been saved from the workspace, this method takes a trio of `Database::Entry` objects extracted from the trees of the three commits involved in the merge. Each entry is stored off with a stage from 1 to 3, after any stage-0 entries have been removed. If the file does not exist in any of the three commits, then we skip writing an entry for that stage.

```
# lib/index.rb

def add_conflict_set(pathname, items)
  remove_entry_with_stage(pathname, 0)

  items.each_with_index do |item, n|
    next unless item
    entry = Entry.create_from_db(pathname, item, n + 1)
    store_entry(entry)
  end
  @changed = true
end
```

The `Index::Entry.create_from_db` method takes the pathname, the `Database::Entry`, and the stage number, and builds a new entry from this information. All the fields usually populated from `stat()` information are zero, because the entry does not reflect a file in the workspace. The `flags` field contains the length of the pathname as usual, but the stage bits are added into its value.

```
# lib/index/entry.rb

def self.create_from_db(pathname, item, n)
  path = pathname.to_s
  flags = (n << 12) | [path.bytesize, MAX_PATH_SIZE].min

  Entry.new(0, 0, 0, 0, 0, item.mode, 0, 0, 0, item.oid, flags, path)
end
```

Recall that the `flags` field's lowest 12 bits are the pathname length, and the two bits before that represent the stage, a number from 0 to 3. To generate the number to store in the `flags` field, we therefore bit-shift the stage by twelve places, and then bitwise-or the result with the path size. Represented in binary for our example entry:

1100000000000000	= 3 << 12 = 12288
	101 = 5

11000000000101	= 12293

Finally, the `Index` needs a method to check if it's in a conflicted state, which should prevent the merge from succeeding and block us from writing any commits until the conflict is resolved. The index is conflicted if any of its entries have a non-zero stage.

```
# lib/index.rb

def conflict?
  @entries.any? { |key, entry| entry.stage > 0 }
```

```
end
```

18.4. Conflict detection

The infrastructure for storing conflicts is now in place, and we can move on to detecting them during the course of the `Merge::Resolve` class logic. Before diving in to the details, let's make the `merge` command fail if resolving the merge leaves the index in a conflicted state.

```
# lib/command/merge.rb

def resolve_merge
  repo.index.load_for_update

  merge = ::Merge::Resolve.new(repo, @inputs)
  merge.execute

  repo.index.write_updates
  exit 1 if repo.index.conflict?
end
```

Now, we need to think about why merges have been working so far, and what would cause them to fail due to a conflict. The `Merge::Resolve` class makes use of `Repository::Migration` to update the workspace and index with the changes from the merged branch, and `Repository::Migration` checks whether the applied changes would conflict with the current state of the repository⁹. In particular, it checks that the left-hand side of the tree diff matches what's currently in the index before applying it.

The tree diff data structure¹⁰ is a hash that maps `Pathname` objects to a pair of `Database::Entry` values representing the path's content before and after the change; these values are called the *pre-image* and *post-image* for a path. For example, in the merge discussed in Section 17.1.2, “Merging a chain of commits”, it looks like this (with IDs abbreviated):

```
{
  Pathname("g.txt") => [
    Database::Entry(oid="00750ed...", mode=0100644),      # pre-image for g.txt
    Database::Entry(oid="7ed6ff8...", mode=0100644)        # post-image for g.txt
  ],
  Pathname("h.txt") => [
    Database::Entry(oid="d00491f...", mode=0100644),      # pre-image for h.txt
    Database::Entry(oid="1e8b314...", mode=0100644)        # post-image for h.txt
}
```

For the migration to go ahead, the current index must match either of the images for each path — it must either be unchanged, or it must match the new value — and the workspace must match the index. If the history from the best common ancestor to the `HEAD` commit does not change any of the files changed by the incoming branch, and there are no uncommitted changes, this will be the case, and the migration succeeds.

We run into problems when the index doesn't match either image for a given entry, i.e. when the `HEAD` branch has changed one of the files that the incoming branch has also changed.

⁹Section 14.5, “Preventing conflicts”

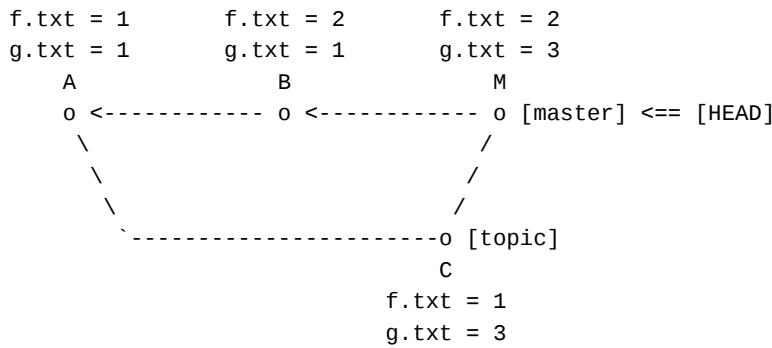
¹⁰Section 14.1, “Telling trees apart”

18.4.1. Concurrency, causality and locks

In Section 16.2.2, “Logging multiple branches” we encountered the idea that the commits in the history graph are *partially ordered*. In the `log` command this merely poses a problem for deciding which order to display the commits in so that they make sense to the user. When merging, a much more critical problem is introduced: merging changes from partially ordered commits means we need to be able to interpret the commits as though they formed a flat linear sequence. If we imagine that the changes from each commit represent a sequence of events, does the end result depend on the order in which the events take place?

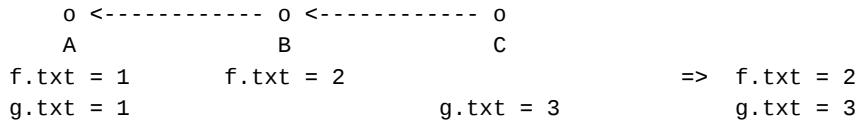
For example, consider this merge from the top of the chapter. The `master` branch changes `f.txt` while the `topic` branch changes `g.txt`, and the merge succeeds.

Figure 18.10. Merging branches changing different files



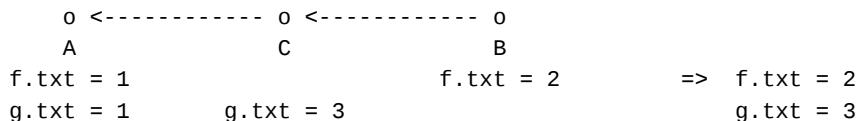
Let’s imagine that, instead of happening on parallel branches, commits `B` and `C` were instead applied one after the other with no branching. We’ll display this as follows, showing only the files each commit changes. On the right is the end state of the project files after all changes have been applied.

Figure 18.11. Sequential application of changes from parallel branches



A merge succeeds without conflicts if the commits on either side could be reordered¹¹ without changing the resulting end state. Let’s change the order of changes `B` and `C` and see what the end state is.

Figure 18.12. Reversed sequential application of changes from parallel branches

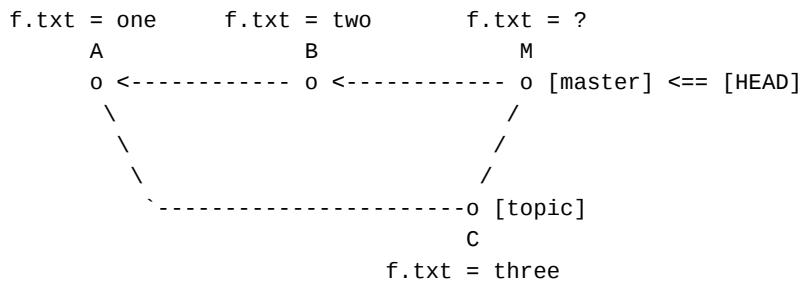


`B` and `C` change different files, so it doesn’t matter what order these changes are done in, the result is the same — commits `B` and `C` are said to be *commutative*¹², or to *commute* with one another. What if both branches change the same file?

¹¹That is, either all the commits from the left branch are applied and then those from the right branch, or vice versa. Commits within a single branch are not reordered.

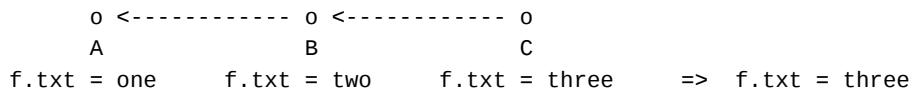
¹²https://en.wikipedia.org/wiki/Commutative_property

Figure 18.13. Merging branches changing the same file



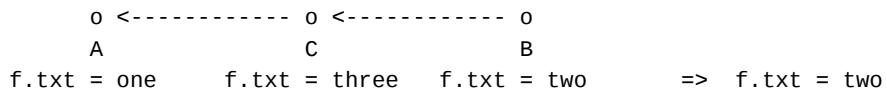
Here is a flattened history showing *B* first, then *C*:

Figure 18.14. Sequential application of changes to the same file



Now here's the history with the branches reordered:

Figure 18.15. Reversed sequential application of changes to the same file



In this case, the end result differs if we change the order the changes are applied in — *B* and *C* do not commute. When one commit directly follows another via its parent link, that means the author has already specified the order of the commits. The author of commit *B* has seen the contents of commit *A* and based their changes on it; they've decided which bits of *A* to overwrite. But when two commits are not explicitly ordered in this way, and they do not commute, Git can't guess how to combine their changes. The order of the changes matters, but the history graph does not order the changes relative to one another. Assuming one ordering or another would result in data being lost.

This problem is equivalent to the race conditions one finds in multithreaded programming or in database transactions. Databases have a concept called *serialisability*¹³, meaning that the effect of executing multiple concurrent transactions is the same as executing all the transactions one by one in some sequence — that's what we simulated above by flattening the graph and reordering commits that aren't explicitly ordered in the original history. A database will lock records so that if two concurrent transactions attempt to alter the same data item, one transaction will wait for the other to finish, or it will fail and roll back. Similar locking mechanisms can be used to prevent data races in multithreaded programming.

We can also compare this situation to the file-writing problem that we solved using `Lockfile` in Section 4.2.1, “Safely updating `.git/HEAD`”. Commits *B* and *C* are like two concurrent processes reading a file and then overwriting its contents. Both ‘read’ `f.txt` by taking its contents in commit *A*, and they replace it with their own data. Rather than locking the file, Git allows both writes to be recorded on distinct branches, but when we come to merge *B* and *C*, we need to decide whose write wins.

¹³<https://en.wikipedia.org/wiki/Serializability>

In a distributed data system, locking doesn't work especially well. That's because it requires that all the system's nodes — individual programs each storing and making changes to their copy of the data — can communicate with each other over a network, so they can agree on who holds a particular lock and is allowed to change something. If some of the nodes are disconnected from one another, this is not possible. The remaining nodes will need to wait until the failed nodes are back online so they can reach agreement, or else go ahead with their changes while some nodes are offline and hope they can resolve any discrepancies later. In effect, adding locks to Git would turn it into a centralised system, where authors need to constantly communicate to agree what the latest version of each file is.

Git uses this latter model: it does not put locks on files, or prevent anyone else from editing the same files as you. This means contributors can work on the commit history in separate branches, often modifying the same files as each other, without being connected all the time. But if their changes do not commute, they'll need to manually tell Git how to fix the resulting data conflicts.

18.4.2. Add/edit/delete conflicts

We saw above that when two branches change different files, the merge is successful. Changes to different files commute: we can reorder them and get the same result. Changes to the *same* file do not usually commute: if two branches write to the same path, the end result depends on which changes are applied last.

Git has three ways that tree items can be changed: new items can be added, and existing items can be edited or deleted. *Editing* an item can mean changing its mode or object ID. Detecting conflicts means deciding which combinations of these changes do not commute with one another.

If the merge base does not contain a particular path, it's possible that both branches add an entry at that path. If these entries differ, then the two additions do not commute; one of the entries must overwrite the other. So additions on the same path conflict with one another, and Git calls these *add/add* conflicts.

If the merge base does contain a certain path, then both sides of the merge might edit its contents. As we saw in our example above, these edits do not commute (unless both sides edit the file in the same way), and so edits conflict with one another and Git calls these *content* conflicts.

Likewise, edits conflict with deletions. If one branch edits a file and other removes it, the end result depends on how these changes are ordered. Either the file is first edited and then deleted, or it's deleted and then 'edited', a sequence of operations that doesn't even make sense. So, edits conflict with deletions, and Git calls them *modify/delete* conflicts. However, deletions do not conflict with one another; if both branches delete a file then they both create the same end state and their order is unimportant.

To detect these changes and prevent the merge from completing, we need to adjust the `Merge::Resolve#execute` method. Much like we changed `Repository::Migration` to check the incoming tree diff for conflicts with the index and workspace, we need `Merge::Resolve` to analyse the branches it's been asked to merge and flag any conflicts that result from both sides modifying the same file. Rather than directly using the tree diff from the `right` branch, the `prepare_tree_diffs` method will instead prepare a new tree diff called `@clean_diff` that

will apply against the current state of the index. A migration is used to apply this diff, and any conflicts are then added to the index.

```
# lib/merge/resolve.rb

def execute
  prepare_tree_diffs

  migration = @repo.migration(@clean_diff)
  migration.apply_changes

  add_conflicts_to_index
end
```

`prepare_tree_diffs` has the job of detecting any conflicts between the merged branches, and building a tree diff that can be applied to the index. It starts by calculating two tree diffs: one between the merge base and the left commit (the current HEAD, whose tree is in the index), and one between the base and the right commit (the commit identified on the command-line). It then iterates over the right tree diff to find out if it modifies any of the same files as the left tree diff.

```
# lib/merge/resolve.rb

def prepare_tree_diffs
  base_oid      = @inputs.base_oids.first
  @left_diff   = @repo.database.tree_diff(base_oid, @inputs.left_oid)
  @right_diff  = @repo.database.tree_diff(base_oid, @inputs.right_oid)
  @clean_diff  = {}
  @conflicts   = {}

  @right_diff.each do |path, (old_item, new_item)|
    same_path_conflict(path, old_item, new_item)
  end
end
```

`same_path_conflict`, shown below, takes a single entry from the right tree diff: its `Pathname` and the `Database::Entry` of its pre- and post-images. Its job is to check if the left diff also modifies the same file, and if so, attempt to reconcile the changes.

If the left diff does not contain the given path, then there is no conflict to report. That means we can put the right commit's version of the file into the index, which we do by storing `[base, right]` for the path in the cleaned tree diff. This will apply when handed to `Repository::Migration` because, since the left commit is the current `HEAD`, and the left branch does not modify this path, that means `base` matches the current index entry for that path.

If there is an entry in the left diff for this path, then we fetch its post-image using `@left_diff[path][1]`. If this is equal to `right`, that means both sides have changed the path but they've both changed it to have the same content; the right commit's version is the same as that in `HEAD`. Therefore, the index for this path does not need changing at all, and we can bail out.

If we get past these checks, then both sides of the merge have changed the path in different ways. In this case, we need to build some new `Database::Entry` object that somehow represents both their contents, so we can slot it into the cleaned diff and `Repository::Migration` can put this combined version into the workspace. At this point, at most one of `base`, `left` and `right` can be `nil`; if `base` is `nil` and both sides have changed the path, then neither `left` nor `right` is `nil`,

and if `base` is not `nil`, then at most one of `left` and `right` is `nil`, otherwise they'd be equal and thus would not conflict.

A `Database::Entry` has two properties, `oid` and `mode`, and we need to merge both of these values by comparing the `base`, `left` and `right` values. The `merge_blobs` and `merge_modes` methods will return the merged value of each field, and a boolean indicating whether they merged cleanly. We put the new `Database::Entry` in the cleaned diff with `left` as the pre-image so it applies against the current index, and if either value did not merge cleanly then we add the trio `[base, left, right]` to the list of conflicts to be stored in the index.

```
# lib/merge/resolve.rb

def same_path_conflict(path, base, right)
  unless @left_diff.has_key?(path)
    @clean_diff[path] = [base, right]
    return
  end

  left = @left_diff[path][1]
  return if left == right

  oid_ok, oid = merge_blobs(base&.oid, left&.oid, right&.oid)
  mode_ok, mode = merge_modes(base&.mode, left&.mode, right&.mode)

  @clean_diff[path] = [left, Database::Entry.new(oid, mode)]
  @conflicts[path] = [base, left, right] unless oid_ok and mode_ok
end
```

The `merge_blobs` and `merge_modes` methods both perform a three-way merge of their respective inputs. As a first step, both of them will attempt to use this general method for merging the three values:

```
# lib/merge/resolve.rb

def merge3(base, left, right)
  return [false, right] unless left
  return [false, left] unless right

  if left == base or left == right
    [true, right]
  elsif right == base
    [true, left]
  end
end
```

Remember only one of the inputs to this method can be `nil`, and we only invoke it if both branches have changed a certain path. If `left` is `nil`, that means the file was deleted in the left branch and modified in the right, and so this is a conflict but we can put the right commit's version into the workspace — we return `[false, right]`. If `right` is `nil` we return `[false, left]` by the same argument.

Then we deal with successful merges — although both sides have changed the file in some way, they might have changed different properties, for example one side changed the object ID while the other changed the mode. When merging each individual property, either side might be equal to the `base` value. If `left` is equal to `base`, or `left` and `right` are the same, then this value can

be successfully merged to the right commit's value. Likewise, if `right` is equal to `base`, then the right branch has not changed this value and we can just use the left's value.

If none of these checks pass, then `merge3` returns `nil`, meaning it is not possible to select either the `left` or `right` value on its own as the result of the merge: something more complicated is necessary.

In the case of `merge_blobs`, we begin by calling `merge3` and return the result if one was found. Otherwise we fall through to a routine that builds a combined blob out of the two sides, by concatenating their contents along with the `<<<<<`, `=====` and `>>>>>` markers. The blob is saved to the database and its ID is returned along with `false`, indicating a conflict.

```
# lib/merge/resolve.rb

def merge_blobs(base_oid, left_oid, right_oid)
  result = merge3(base_oid, left_oid, right_oid)
  return result if result

  blob = Database::Blob.new(merged_data(left_oid, right_oid))
  @repo.database.store(blob)
  [false, blob.oid]
end

def merged_data(left_oid, right_oid)
  left_blob = @repo.database.load(left_oid)
  right_blob = @repo.database.load(right_oid)

  [
    "<<<<< #{ @inputs.left_name }\n",
    left_blob.data,
    "=====\\n",
    right_blob.data,
    ">>>>> #{ @inputs.right_name }\n"
  ].join("")
end
```

`merge_modes` is somewhat simpler. Like `merge_blobs` it begins by calling `merge3`, but if it gets no result, it just returns the left commit's mode for the file. Recall that there are only two possible modes for blobs: `1006448` and `1007558`. The only way `merge3` does not return a value is if `left` and `right` are both non-`nil` and all three values are different from one another. The only way this can happen for modes is that both sides have added the file with different modes, e.g. `base = nil, left = 0100644` and `right = 0100755`. There's no way of combining modes — they can't be concatenated like the contents of files — so we just need to pick one of them arbitrarily but flag a conflict. Git picks the existing `HEAD` mode for the file, i.e. `left`.

```
# lib/merge/resolve.rb

def merge_modes(base_mode, left_mode, right_mode)
  merge3(base_mode, left_mode, right_mode) || [false, left_mode]
end
```

Once the whole diff has been analysed and `@clean_diff` has been built, the `execute` method uses `Repository::Migration` to apply the cleaned diff and then calls `add_conflicts_to_index`. The cleaned diff contains contents for some paths that are conflicted, just so that the migration will put that content into the workspace. But, it will also

result in stage-0 index entries being added for those paths. We need to evict those entries and replace them with conflict entries, which we do using the `Index#add_conflict_set` method¹⁴.

```
# lib/merge/resolve.rb

def add_conflicts_to_index
  @conflicts.each do |path, items|
    @repo.index.add_conflict_set(path, items)
  end
end
```

18.4.3. File/directory conflicts

There is one further kind of conflict that we've not yet touched on. Every tree diff generated by the database represents a transition from one valid tree to another, assuming the index only allows valid trees to end up forming commits¹⁵. The two diffs we generate in `Merge::Resolve` are d_{BL} and d_{BR} , the difference from the merge base to the left commit and from the base to the right commit. Whereas add/edit/delete conflicts arise because these two diffs do not commute, this final kind of conflict happens because the diffs do not *compose*: there is no way of applying both of them, in any order, and getting a valid result. That is, $T_B d_{BL} + d_{BR}$ does not have a well-defined value.

Why does this happen? Well, the two diffs might create two files that cannot coexist in the same tree. For example, if the left branch adds a file called `f.txt`, and the right branch adds one called `f.txt/g.txt`, there is no tree we can form that contains both of them; `f.txt` cannot be both a file and a directory at the same time. When this happens, Git keeps the index entry and workspace copy of `f.txt/g.txt`, and it adds the file for the conflicted path to the workspace with a modified name, e.g. `f.txt~HEAD`.

To detect this kind of conflict, we need to run a check over both diffs, not just the right one, but we can fold the check for the right diff into the existing loop. This new check will detect whether the other diff in the pair contains a post-image for any parent of the current path. For example, if `@right_diff` contains a post-image for `a/b/c.txt`, then we check whether `@left_diff` contains a post-image for `a/b` or `a`, which would be incompatible with `a/b/c.txt` existing.

If a diff affects a path but has no post-image for it, that means it deletes the given path, and so it doesn't matter if the other diff has a post-image for a parent path. As the checks proceed, they'll add files to a new hash called `@untracked`. This structure contains the renamed conflicted files that should be added to the workspace but not the index.

```
# lib/merge/resolve.rb

def prepare_tree_diffs
  # ...
  @untracked = {}
  # ...

  @right_diff.each do |path, (old_item, new_item)|
    file_dir_conflict(path, @left_diff, @inputs.left_name) if new_item
```

¹⁴Section 18.3.4, "Storing conflicts"

¹⁵Section 7.3, "Stop making sense"

```

    same_path_conflict(path, old_item, new_item)
end

@left_diff.each do |path, (_, new_item)|
  file_dir_conflict(path, @right_diff, @inputs.right_name) if new_item
end
end

```

The `file_dir_conflict` check takes a `Pathname`, one of the tree diffs, and that tree diff's 'name', i.e. the revision string used to identify it by the user. For every parent directory of the pathname, it checks whether the given diff contains a post-image for that directory. If it does, then that entry needs to be renamed. For example, if the right diff contains a post-image for `f.txt/g.txt`, and the left contains a post-image for `f.txt`, then we want to remove the `f.txt` entry from the cleaned diff and write to the workspace with the name `f.txt~HEAD` instead.

When a conflict is detected, we add it to the `@conflicts` set, choosing the order of items in the conflict based on whether we're examining the left or right diff. We delete the conflicting path from `@clean_diff`, and instead store its post-image in `@untracked` using its new name.

```

# lib/merge/resolve.rb

def file_dir_conflict(path, diff, name)
  path.dirname.ascend do |parent|
    old_item, new_item = diff[parent]
    next unless new_item

    @conflicts[parent] = case name
      when @inputs.left_name then [old_item, new_item, nil]
      when @inputs.right_name then [old_item, nil, new_item]
    end

    @clean_diff.delete(parent)
    rename = "#{parent}~#{name}"
    @untracked[rename] = new_item
  end
end

```

To add these untracked files to the workspace, `Merge::Resolve#execute` needs to perform one extra step at the end: writing all the entries in `@untracked` into the workspace.

```

# lib/merge/resolve.rb

def execute
  prepare_tree_diffs

  migration = @repo.migration(@clean_diff)
  migration.apply_changes

  add_conflicts_to_index
  write_untracked_files
end

def write_untracked_files
  @untracked.each do |path, item|
    blob = @repo.database.load(item.oid)
    @repo.workspace.write_file(path, blob.data)
  end
end

```

```
end
```

This just requires one extra method in `Workspace`: `write_file` takes a path and a string, and creates or overwrites that path so it contains the given data.

```
# lib/workspace.rb

def write_file(path, data)
  flags = File::WRONLY | File::CREAT | File::TRUNC
  File.open(@pathname.join(path), flags) { |f| f.write(data) }
end
```

Our `merge` command is now capable of detecting most common conflicts and it will fail unless the index is clean. We can now move on to the final step in merging: resolving conflicts and resuming the merge commit.

19. Conflict resolution

Over the course of the last two chapters, we've developed a `merge` command that can successfully merge non-conflicting branches, and stops if the index contains conflicts. To round out the `merge` command's behaviour, we need to allow the user to find out which files are in conflict, fix them, and commit the result when all conflicts have been resolved.

19.1. Printing conflict warnings

The first step in this process is that the `merge` command should itself print some output to tell the user which files failed to merge cleanly. For example, Git's `merge` command prints output like this when a conflict is encountered:

```
Auto-merging f.txt
CONFLICT (content): Merge conflict in f.txt
Automatic merge failed; fix conflicts and then commit the result.
```

It tells you when it's trying to perform any non-trivial merge rather than using a file version from one side of the merge, it alerts you when any file is conflicted, and it finishes off by telling you the merge failed and what to do about it.

Now, it would be a mistake to litter the `Merge::Resolve` class with `puts` calls; that class is internal machinery, not user interface code, and we should be able to run it without printing to the terminal. However, the class does need to provide some way of surfacing what it's doing, so that a user interface can display it. We'll start by adding an `on_progress` hook to `Merge::Resolve` so that `Command::Merge` can listen for any log messages generated during the merge. We'll also make it display the closing message should the index end up in a conflicted state.

```
# lib/command/merge.rb

def resolve_merge
  repo.index.load_for_update

  merge = ::Merge::Resolve.new(repo, @inputs)
  merge.on_progress { |info| puts info }
  merge.execute

  repo.index.write_updates

  if repo.index.conflict?
    puts "Automatic merge failed; fix conflicts and then commit the result."
    exit 1
  end
end
```

`Merge::Resolve#on_progress` simply stores off the given block, so that the internals of the class can invoke it and thereby pipe log information back to the caller.

```
# lib/merge/resolve.rb

def on_progress(&block)
  @on_progress = block
```

```
    end

  private

  def log(message)
    @on_progress&.call(message)
  end
```

The `same_path_conflict` method can then use this `log` method to emit the fact that it's auto-merging the current path. It also includes a call to `log_conflict`, which we'll define below, when it adds something to the `@conflicts` set.

```
# lib/merge/resolve.rb

def same_path_conflict(path, base, right)
  # ...

  log "Auto-merging #{path}" if left and right

  oid_ok, oid = merge_blobs(base&.oid, left&.oid, right&.oid)
  mode_ok, mode = merge_modes(base&.mode, left&.mode, right&.mode)

  @clean_diff[path] = [left, Database::Entry.new(oid, mode)]
  return if oid_ok and mode_ok

  @conflicts[path] = [base, left, right]
  log_conflict(path)
end
```

Likewise, `file_dir_conflict` can log the fact it's adding the file from one side of the diff, while recording a conflict on the file at the parent location that's been renamed.

```
# lib/merge/resolve.rb

def file_dir_conflict(path, diff, name)
  path.dirname.ascend do |parent|
    # ...

    log "Adding #{path}" unless diff[path]
    log_conflict(parent, rename)
  end
end
```

The `log_conflict` method is the most lengthy addition to the `Merge::Resolve` class, but it's mostly fairly uninteresting bookkeeping that encodes all Git's rules for how to report certain types of conflict. At the top level, it makes a decision about what type of message to emit, based on which versions are present in the named conflict set.

```
# lib/merge/resolve.rb

def log_conflict(path, rename = nil)
  base, left, right = @conflicts[path]

  if left and right
    log_left_right_conflict(path)
  elsif base and (left or right)
    log_modify_delete_conflict(path, rename)
```

```
    else
      log_file_directory_conflict(path, rename)
    end
  end
```

log_left_right_conflict prints CONFLICT (<type>): Merge conflict in <path>, where type is determined by whether the path exists in the merge base. If it does, then both sides have edited the given file, otherwise both sides have added it.

```
# lib/merge/resolve.rb

def log_left_right_conflict(path)
  type = @conflicts[path][0] ? "content" : "add/add"
  log "CONFLICT (#{$type}): Merge conflict in #{$path}"
end
```

log_modify_delete_conflict logs something like this when one side of the merge has changed a file while the other removed it:

```
CONFLICT (modify/delete): f.txt deleted in HEAD and modified in topic.
Version topic of f.txt left in tree.
```

If the file had to be renamed due to a file/directory conflict, that is flagged in the resulting message:

```
CONFLICT (modify/delete): f.txt deleted in HEAD and modified in topic.
Version topic of f.txt left in tree at f.txt~topic.
```

The log_modify_delete_conflict method grabs the branch names for the two merged commits and generates the above message, including the rename notification if necessary.

```
# lib/merge/resolve.rb

def log_modify_delete_conflict(path, rename)
  deleted, modified = log_branch_names(path)

  rename = rename ? " at #{rename}" : ""

  log "CONFLICT (modify/delete): #{path} " +
    "deleted in #{deleted} and modified in #{modified}. " +
    "Version #{modified} of #{path} left in tree#{rename}."
end
```

The log_branch_names helper function returns the names of the two merged branches from the Merge::Inputs object, swapping their order depending on whether the conflict set for the path contains an entry from the left or right commit.

```
# lib/merge/resolve.rb

def log_branch_names(path)
  a, b = @inputs.left_name, @inputs.right_name
  @conflicts[path][1] ? [b, a] : [a, b]
end
```

Finally, log_file_directory_conflict emits a message that alerts the user to a renamed file in the tree as a result of a file/directory conflict.

```
# lib/merge/resolve.rb

def log_file_directory_conflict(path, rename)
  type = @conflicts[path][1] ? "file/directory" : "directory/file"
  branch, _ = log_branch_names(path)

  log "CONFLICT (#{type}): There is a directory " +
    "with name #{path} in #{branch}. " +
    "Adding #{path} as #{rename}"
end
```

19.2. Conflicted status

Showing which files are conflicted when the merge first runs is a good start, but the user might have a lot of conflicts to fix in order to reach a commit state. As they work towards that goal, it would be useful to show them which files are still in conflict via the `status` command, which is currently incapable of displaying the status of conflicted index entries.

Fortunately, this is relatively straightforward to fix. In the `Repository::Status` class, we'll add and expose a new set called `@conflicts` along with all the other structures we use to store information.

```
# lib/repository/status.rb

attr_reader :conflicts

def initialize(repository)
  # ...

  @changed      = SortedSet.new
  @index_changes = SortedHash.new
  @conflicts    = SortedHash.new
  @workspace_changes = SortedHash.new
  @untracked_files = SortedSet.new

  # ...
end
```

When we check each entry in the index in `check_index_entries`, we check the entry against the workspace and HEAD tree as normal if its stage is zero. Otherwise, we add its stage to a list of stages we hold for the entry's path in the `@conflicts` set.

```
# lib/repository/status.rb

def check_index_entries
  @repo.index.each_entry do |entry|
    if entry.stage == 0
      check_index_against_workspace(entry)
      check_index_against_head_tree(entry)
    else
      @changed.add(entry.path)
      @conflicts[entry.path] ||= []
      @conflicts[entry.path].push(entry.stage)
    end
  end
end
```

```
end
```

This means that after all the checks have run, `@conflicts` will map each conflicted path to an array containing some combination of the numbers 1, 2 and 3 representing which stages exist in the index for the given path.

Now that `Repository::Status` generates the appropriate data for us, we can look at amending the two status formats to display it.

19.2.1. Long status format

To the `Command::Status#print_long_format` method we can add a clause between displaying the staged and unstaged changes, which lists the contents of the `conflicts` set.

```
# lib/command/status.rb

def print_long_format
  print_branch_status

  print_changes("Changes to be committed", @status.index_changes, :green)
  print_changes("Unmerged paths", @status.conflicts, :red, :conflict)
  print_changes("Changes not staged for commit", @status.workspace_changes, :red)
  print_changes("Untracked files", @status.untracked_files, :red)

  print_commit_status
end
```

The `print_changes` method will need some new data structures to work with that contain the labels used for each possible conflict status. The values in `Repository::Status#conflicts` can be any possible subset of [1, 2, 3], except for [1] — that state would mean both sides had deleted a file, which is not a conflict. The `CONFLICT_LONG_STATUS` structure maps these values to appropriate display strings.

```
# lib/command/status.rb

CONFLICT_LABEL_WIDTH = 17

CONFLICT_LONG_STATUS = {
  [1, 2, 3] => "both modified:",
  [1, 2]   => "deleted by them:",
  [1, 3]   => "deleted by us:",
  [2, 3]   => "both added:",
  [2]      => "added by us:",
  [3]      => "added by them:"
}
```

We also need to decide which set of labels to use for each set of changes. For all the existing change sets, we want the `LONG_STATUS` and `LABEL_WIDTH` values, whereas for conflicts we need `CONFLICT_LONG_STATUS` and `CONFLICT_LABEL_WIDTH`. We'll encode this information as follows:

```
# lib/command/status.rb

UI_LABELS = { :normal => LONG_STATUS, :conflict => CONFLICT_LONG_STATUS }
```

```
UI_WIDTHS = { :normal => LABEL_WIDTH, :conflict => CONFLICT_LABEL_WIDTH }
```

The `print_changes` method can now select which set of labels to use for the set of changes it's been given. Existing calls default to `label_set = :normal`, but the conflict set overrides this by passing `:conflict`.

```
# lib/command/status.rb

def print_changes(message, changeset, style, label_set = :normal)
  return if changeset.empty?

  labels = UI_LABELS[label_set]
  width = UI_WIDTHS[label_set]

  puts "#{ message }:"
  puts ""
  changeset.each do |path, type|
    status = type ? labels[type].ljust(width, " ") : ""
    puts "\t" + fmt(style, status + path)
  end
  puts ""
end
```

19.2.2. Porcelain status format

The porcelain format likewise needs a set of states to display. For normal non-conflicting entries, the existing implementation takes two letters from the `SHORT_STATUS` structure, one for the difference between HEAD and index, the other between the index and workspace. For conflict statuses, it will be easier to map each possible combination of stages to a two-letter code.

```
# lib/command/status.rb

CONFLICT_SHORT_STATUS = {
  [1, 2, 3] => "UU",
  [1, 2]     => "UD",
  [1, 3]     => "DU",
  [2, 3]     => "AA",
  [2]        => "AU",
  [3]        => "UA"
}
```

Then in `Command::Status#status_for`, we can select the appropriate two-letter code for conflicted entries. Because conflicted paths are included in `Repository::Status#changed`, their statuses will be included in alphabetical order with all the normal statuses we've seen previously.

```
# lib/command/status.rb

def status_for(path)
  if @status.conflicts.has_key?(path)
    CONFLICT_SHORT_STATUS[@status.conflicts[path]]
  else
    left  = SHORT_STATUS.fetch(@status.index_changes[path], " ")
    right = SHORT_STATUS.fetch(@status.workspace_changes[path], " ")
    left + right
  end
end
```

```
end
```

19.3. Conflicted diffs

The status command provides the most valuable signal about what's conflicted. Most people's workflow involves inspecting the conflicted files, finding the <<<<< markers, and selecting portions of the conflicting versions to compose a working version, leaning on their test suite to check things. However the diff command is also useful in showing you how the current state of the files differs from the versions that went into the merge.

When the diff command is run with conflicting entries in the index, you'll see something like the following:

```
diff --cc f.txt
index f719efd,2bdf67a..0000000
--- a/f.txt
+++ b/f.txt
@@@ -1,1 -1,1 +1,5 @@@
+<<<<<< HEAD
+two
+=====
+ three
++>>>>> topic
```

Notice the --cc switch instead of --git, the multiple object IDs before .. on the index line, the triple @@@ hunk markers and multiple minus-signed offsets, and the two columns of plus/minus markers on the lines of content. This is called a *combined diff*, and it shows the difference between the workspace copy of the file and the stage 2 and 3 index entries simultaneously. This diff format is also used by the log command when displaying patches for merge commits.

We won't implement this format immediately, since it's a little complex and also hard to read. For now, we'll just flag that some files are unmerged, and implement some options to diff to view the difference between specific versions of the file from before the merge.

19.3.1. Unmerged paths

The simplest change to make to the diff command is to put in a decision about how to handle conflicted files. Rather than just displaying the workspace changes from the Repository::Status object, we make a combined list of all the changed and conflicted paths before iterating over them. If the file is conflicted, we display it using a new method print_conflict_diff, otherwise we call print_workspace_diff which contains the existing display logic.

```
# lib/command/diff.rb

def diff_index_workspace
  return unless @options[:patch]

  paths = @status.conflicts.keys + @status.workspace_changes.keys

  paths.sort.each do |path|
    if @status.conflicts.has_key?(path)
      print_conflict_diff(path)
    else
```

```
    else
        print_workspace_diff(path)
    end
end
end
```

The `print_conflict_diff` method for now just prints `Unmerged path` with the name of the file, while `print_workspace_diff` prints a patch between a normal index entry and the workspace file as before.

```
# lib/command/diff.rb

def print_conflict_diff(path)
    puts "* Unmerged path #{path}"
end

def print_workspace_diff(path)
    case @status.workspace_changes[path]
    when :modified then print_diff(from_index(path), from_file(path))
    when :deleted   then print_diff(from_index(path), from_nothing(path))
    end
end
```

19.3.2. Selecting stages

Rather than displaying a combined diff, Git gives the option to see the difference between a specific version of the file before the merge, and the file in the workspace. If you pass `--base` or `-1`, it shows the difference against the file from the base of the merge — the stage 1 index entry. With `--ours` or `-2` it bases the diff on the left side of the merge, i.e. the current HEAD, and with `--theirs` or `-3` it's based on the right side, or the incoming merged commit.

Here's our example conflicted file compared to the merge base, where it contained the string `one`:

```
$ git diff --base

* Unmerged path f.txt
diff --git a/f.txt b/f.txt
index 5626abf..53e1e5a 100644
--- a/f.txt
+++ b/f.txt
@@ -1 +1,5 @@
-one
+<<<<< HEAD
+two
=====
+three
+>>>>> topic
```

Comparing to the left side of the merge, where the file had the contents `two`, we see one side of the conflicted region is the content from that version:

```
$ git diff --ours

* Unmerged path f.txt
diff --git a/f.txt b/f.txt
```

```
index f719efd..53e1e5a 100644
--- a/f.txt
+++ b/f.txt
@@ -1 +1,5 @@
+<<<<< HEAD
two
=====
+three
+>>>>> topic
```

Likewise, comparing to the right side of the merge highlights the other side of the conflicted region as unmodified.

```
$ git diff --theirs

* Unmerged path f.txt
diff --git a/f.txt b/f.txt
index 2bdf67a..53e1e5a 100644
--- a/f.txt
+++ b/f.txt
@@ -1 +1,5 @@
+<<<<< HEAD
two
=====
three
+>>>>> topic
```

These diffs are in the usual format that we implemented in Chapter 12, *Spot the difference*, we just need some code to select the right index entry to compare the workspace file to. Let's expand the `Command::Diff#define_options` method to include these stage-selection flags.

```
# lib/command/diff.rb

def define_options
  @options[:patch] = true
  define_print_diff_options

  @parser.on "--cached", "--staged" do
    @options[:cached] = true
  end

  @parser.on "-1", "--base") { @options[:stage] = 1 }
  @parser.on "-2", "--ours") { @options[:stage] = 2 }
  @parser.on("-3", "--theirs") { @options[:stage] = 3 }
end
```

We can then expand the `print_conflict_diff` method to show a patch, if the user has specified a stage. If they have not, then `@options[:stage]` will be `nil`.

```
# lib/command/diff.rb

def print_conflict_diff(path)
  puts "* Unmerged path #{path}"

  target = from_index(path, @options[:stage])
  return unless target

  print_diff(target, from_file(path))
```

```
end
```

The `from_index` method needs to gain a `stage` parameter for selection particular items from the index. If `stage` is `nil`, or the index does not contain an entry with the required stage, then we return `nil`, which causes `print_conflict_diff` to do nothing.

```
# lib/command/diff.rb

def from_index(path, stage = 0)
  entry = repo.index.entry_for_path(path, stage)
  entry ? from_entry(path, entry) : nil
end
```

The user now has enough information to accurately view the state of the repository with conflicted files, and we can provide tools for them to complete the merge once they've resolved all the conflicts.

19.4. Resuming a merge

Once we've resolved all the conflicts, we want to resume the merge — to tell Git what the merged tree should look like and have it write a merge commit as it would have done if there were no conflicts. Git lets us do this using the `commit` and `merge --continue` commands.

To complete this functionality, a few moving parts are needed. First, we need to be able to evict conflicted entries from the index, using the `add` command to replace them with a clean version. Once we've removed all the conflicts from the index, we want to run a command to write a merge commit using the original message and the merged commit ID as the second parent. This requires storing the state of the original `merge` command before it failed due to the conflicts. When we run `merge --continue` or `commit`, we want to pick up the state from the in-progress merge and use it to write a multi-parent commit, as long as the index is clean.

19.4.1. Resolving conflicts in the index

Resolving conflicts in the index is straightforward. The `add` command invokes the `Index#add` method, and just like `Index#add_conflict_set` removes any stage-0 entries for the given path, we can make `Index#add` remove any conflict entries before inserting a stage-0 entry.

```
# lib/index.rb

def add(pathname, oid, stat)
  (1..3).each { |stage| remove_entry_with_stage(pathname, stage) }

  ...
end
```

19.4.2. Retaining state across commands

When we run `merge --continue` or `commit`, with no further arguments, while a conflicted merge is in progress, it should write a merge commit using the original input message and the ID of the merged commit as the second parent. Before the `merge` command exits, it needs to write these values to disk so they can be recalled next time we run `jit`.

We can start by saving the original message and the ID of the right-hand commit when we first run the `merge` command (the ID of the left-hand commit is in `.git/HEAD`, as usual). After gathering the required information, we'll call `pending_commit.start` from the `Command::Merge` class.

```
# lib/command/merge.rb

def run
  #
  # ...

  pending_commit.start(@inputs.right_oid, @stdin.read)
  resolve_merge
  commit_merge

  exit 0
end
```

`pending_commit` is a method in the mixed-in `WriteCommit` module that forwards the call to the `Repository` class and memoises the result.

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

def pending_commit
  @pending_commit ||= repo.pending_commit
end
```

`Repository#pending_commit` in turn constructs a `Repository::PendingCommit` object using the repository's `.git` pathname.

```
# lib/repository.rb

def pending_commit
  PendingCommit.new(@git_path)
end
```

The `Repository::PendingCommit` class itself manages the storage and retrieval of these saved merge values, and lets us check whether a merge is currently in progress. It saves off the commit ID and message passed to it into the files `.git/MERGE_HEAD` and `.git/MERGE_MSG`.

```
# lib/repository/pending_commit.rb

class Repository
  class PendingCommit

    def initialize(pathname)
      @head_path = pathname.join("MERGE_HEAD")
      @message_path = pathname.join("MERGE_MSG")
    end

    def start(oid, message)
      flags = File::WRONLY | File::CREAT | File::EXCL
      File.open(@head_path, flags) { |f| f.puts(oid) }
      File.open(@message_path, flags) { |f| f.write(message) }
    end

  end
end
```

If the merge runs successfully, i.e. it does not exit due to a conflicted index, then we can immediately remove this data by calling `PendingCommit#clear`.

```
# lib/command/merge.rb

def commit_merge
  parents = [@inputs.left_oid, @inputs.right_oid]
  message = pending_commit.merge_message

  write_commit(parents, message)

  pending_commit.clear
end
```

`Repository::PendingCommit#clear` simply removes the files it was storing information in.

```
# lib/repository/pending_commit.rb

def clear
  File.unlink(@head_path)
  File.unlink(@message_path)
end
```

19.4.3. Writing a merge commit

With all the necessary state now available inside the `.git` directory, it's possible to resume a merge with the original inputs intact. Let's add the `--continue` option to the `merge` command, and dispatch to a new method `handle_continue` if it's set. We'll also handle the special case where we try to start a new merge when there's already a merge in progress.

```
# lib/command/merge.rb

def define_options
  @options[:mode] = :run
  @parser.on("--continue") { @options[:mode] = :continue }
end

def run
  handle_continue if @options[:mode] == :continue
  handle_in_progress_merge if pending_commit.in_progress?

  #
end
```

The `handle_continue` method just needs to load the index—for reading, not for updating—then write a merge commit by calling the `resume_merge` method from `Command::WriteCommit`, which we'll define below. `handle_in_progress_merge` deals with calling `merge` when one is already in progress; if `PendingCommit#in_progress?` returns true, we print an error message and exit with a non-zero status.

```
# lib/command/merge.rb

def handle_continue
  repo.index.load
  resume_merge
rescue Repository::PendingCommit::Error => error
  @stderr.puts "fatal: #{error.message}"
```

```
    exit 128
end

def handle_in_progress_merge
  message = "Merging is not possible because you have unmerged files"
  @stderr.puts "error: #{ message }."
  @stderr.puts CONFLICT_MESSAGE
  exit 128
end
```

The PendingCommit#in_progress? method just needs to return true if a merge is unfinished, which we can detect by checking whether .git/MERGE_HEAD exists.

```
# lib/repository/pending_commit.rb

def in_progress?
  File.file?(@head_path)
end
```

If we've called merge with the --continue option, then we'll try to write a merge commit using the stored state. We fetch the parents by reading .git/HEAD and .git/MERGE_HEAD, and get the message out of .git/MERGE_MSG. We then call our existing write_commit method, and clear the merge state before exiting.

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

def resume_merge
  parents = [repo.refs.read_head, pending_commit.merge_oid]
  write_commit(parents, pending_commit.merge_message)

  pending_commit.clear
  exit 0
end
```

The merge_oid and merge_message methods are defined on Repository::PendingCommit. If .git/MERGE_HEAD does not exist, we should raise an error, since we're trying to resume a merge when none is in progress. This error will be caught in Command::Merge#handle_continue, which prints the error message and exits.

```
# lib/repository/pending_commit.rb

Error = Class.new(StandardError)

def merge_oid
  File.read(@head_path).strip
rescue Errno::ENOENT
  name = @head_path.basename
  raise Error, "There is no merge in progress (#{} name } missing)."
end

def merge_message
  File.read(@message_path)
end
```

Now that we're going to be invoking merge and commit in a potential conflict situation, the resume_merge method should prevent us from writing commits when the index is conflicted; we can't generate a tree from the index unless all conflicts have been settled.

We'll add a check to the top of `resume_merge` that exits with an error if the index contains any conflict entries. Now `commit` and `merge` will complain if we try to run them when they're not able to generate a tree successfully.

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

CONFLICT_MESSAGE = <<~MSG
  hint: Fix them up in the work tree, and then use 'git add <file>'
  hint: as appropriate to mark resolution and make a commit.
  fatal: Exiting because of an unresolved conflict.
MSG

def resume_merge
  handle_conflicted_index

  # ...
end

def handle_conflicted_index
  return unless repo.index.conflict?

  message = "Committing is not possible because you have unmerged files"
  @stderr.puts "error: #{message}."
  @stderr.puts CONFLICT_MESSAGE
  exit 128
end
```

The other bit of glue we need is that, if `commit` is invoked during a merge, it should write a merge commit by reading from `PendingCommit`, rather than writing a normal commit from its own inputs.

```
# lib/command/commit.rb

def run
  repo.index.load
  resume_merge if pending_commit.in_progress?

  # ...
end
```

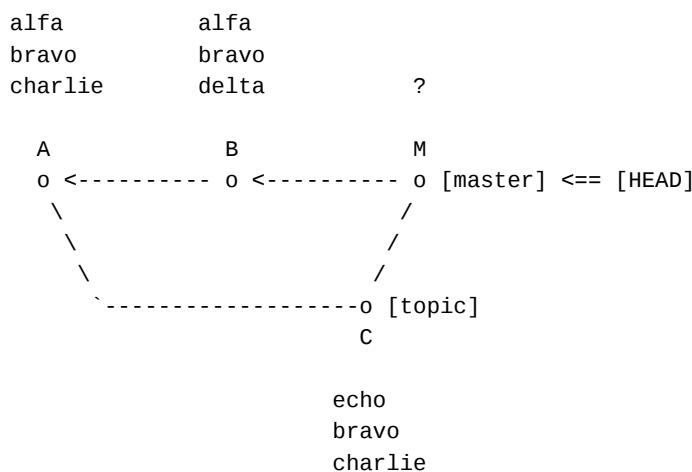
We now have a complete merge workflow: we can merge any commit we like, and all potential edge cases are handled. Null and fast-forward merges are detected and executed, and merge conflicts cause the merge to fail. The conflicts are accurately presented to the user so they can fix them, and they're able to mark files as clean and commit the result when they're finished.

20. Merging inside files

The last three chapters have set us up with a complete merging workflow. In Chapter 17, *Basic merging* we reused some of the checkout infrastructure to perform merges of branches changing different files, and amended the log and revision selection code to deal with merge commits. In Chapter 18, *When merges fail* we investigated the causes of merge conflicts, and how we should change the state of the repository when they occur. And in Chapter 19, *Conflict resolution* we updated the status and `diff` commands to report merge conflicts, and changed the `add`, `commit` and `merge` commands to let the user resolve the conflicts and finish off the merge.

Through this process we've discovered that merges are straightforward when each side of the merge has changed different files, and conflicts occur when each side's edits overlap. It would be nice if we could go one step further and merge changes even when both sides have edited the same file. For example, here's a history in which all commits contain a single file; each commit is annotated with its content for that file.

Figure 20.1. Merging branches changing the same file



It looks as though commit *B* changes line 3, while commit *C* changes line 1. Our current `merge` implementation will generate a conflict here and dump this content into the file:

```
<<<<<
alfa
bravo
delta
=====
echo
bravo
charlie
>>>>>
```

It would be preferable for the `merge` command to detect that each side has modified different regions of the file and reconcile their changes, producing:

```
echo
bravo
delta
```

As you'll have noticed when using Git, it can indeed do this. But, it's more tricky than detecting and merging changes between trees. In trees, every file has a name, which means we can unambiguously identify which parts of the tree each commit has changed. A file, however, is just a list of lines (or words, or characters, or however you decided to break it up), and so it's harder to characterise changes to it in a way that facilitates merging.

For example, suppose we describe the changes on either side as 'Alice inserted a new line at line 3' and 'Bob deleted line 5'. For example, suppose we have the following file contents:

Figure 20.2. Initial file contents

```
 alfa
 bravo
 charlie
 delta
 echo
```

If we apply the changes 'insert a new line at 3' and 'delete line 5' in that order, we get:

Figure 20.3. Inserting a new line and deleting another

<pre> alfa bravo charlie delta echo</pre>	<pre> insert @ 3 -----></pre>	<pre> alfa bravo foxtrot charlie delta echo</pre>	<pre> delete @ 5 -----></pre>	<pre> alfa bravo foxtrot charlie echo</pre>
---	-----------------------------------	--	-----------------------------------	---

If we apply these changes in the opposite order, we get:

Figure 20.4. Deleting a line and then inserting one

<pre> alfa bravo charlie delta echo</pre>	<pre> delete @ 5 -----></pre>	<pre> alfa bravo charlie delta</pre>	<pre> insert @ 3 -----></pre>	<pre> alfa bravo foxtrot charlie delta</pre>
---	-----------------------------------	---	-----------------------------------	--

We arrive at a different result, because 'line 5' does not refer to the same piece of content in each case. If you think about all the ways a file's lines can be changed — inserted, removed, modified, reordered — this line-number-based description quickly looks unsuitable.

Now, some collaborative editing applications do use a model like this, and they adjust the description of each person's edits in light of edits from others that are applied beforehand. There is a broad category of techniques in this space, known as *operational transforms*¹. However, in situations where we can't decide which order to apply edits in — like in Git's partially ordered histories — 'line *N*' is not a good identifier for a piece of content because it leads to ambiguity, as shown above.

In Git merges, we're also not typically looking at small, fine-grained edits. Instead, we're performing a three-way merge between the tips of two branches and their common ancestor,

¹https://en.wikipedia.org/wiki/Operational_transformation

and each branch could have made substantial changes. We need a way of comparing the two merged versions to their common ancestor and producing a single merge result, and fortunately, such an algorithm does exist. In studying it, we'll see how diffs can be used as a basis for further computation, rather than just as a way to display changes to the end user, as they have been until now.

20.1. The diff3 algorithm

To perform this task, Git and many other version control systems use the `diff3` algorithm. `diff3`² is a Unix utility originally created in 1988 by Randy Smith, and described formally in a paper by Sanjeev Khanna, Keshav Kunal and Benjamin C. Pierce³. The algorithm takes as input three texts: the two versions to be merged, and the original version they both derive from, and produces a single merged version, which may contain conflicts if some sections have been edited by both parties.

20.1.1. Worked example

Let's walk through an example to see how it works. This is adapted from the one given in section 2 of the Khanna, Kunal and Pierce paper. Say Alice and Bob are part of a team developing a recipe book. One of the recipes is for a fish soup and includes the following list of ingredients:

Figure 20.5. Original ingredient list

- 
1. celery
 2. garlic
 3. onions
 4. salmon
 5. tomatoes
 6. wine

Now, Alice and Bob are both going through the book and making edits. They each get their own copy of the original version and make some changes to it. Alice takes garlic and onions and moves them to appear after tomatoes in the list, while Bob moves salmon to appear second.

Figure 20.6. Alice and Bob's changes to the ingredient list



Alice	Original	Bob
1. celery	1. celery	1. celery
2. salmon	2. garlic	2. salmon
3. tomatoes	3. onions	3. garlic
4. garlic	4. salmon	4. onions
5. onions	5. tomatoes	5. tomatoes
6. wine	6. wine	6. wine

To merge the changes made by Alice and Bob into a single document, the first thing `diff3` does is calculate a diff between the original copy and Alice's version, and between the original and Bob's. We know there are multiple possible minimal diffs between two strings, but let's assume the diffs come out as follows. Here is Alice's:

²<https://manpages.ubuntu.com/manpages/bionic/en/man1/diff3.1.html>

³<https://www.cis.upenn.edu/~bcpierce/papers/diff3-short.pdf>

Figure 20.7. Alice’s diff against the original

Alice	Original
1. celery	1. celery
-	2. garlic
-	3. onions
2. salmon	4. salmon
3. tomatoes	5. tomatoes
+ 4. garlic	
+ 5. onions	
6. wine	6. wine

And here is Bob’s:

Figure 20.8. Bob’s diff against the original

Original	Bob
1. celery	1. celery
+ 2. garlic	2. salmon
3. onions	3. garlic
- 4. salmon	4. onions
5. tomatoes	5. tomatoes
6. wine	6. wine

Once we have these diffs, the next thing we do is find any lines that are unchanged in both diffs. If you look at the diffs above you’ll see that they both have matching line content for celery (line 1 in all versions), tomatoes (line 3 in Alice’s version, line 5 in the others), and wine (line 6 in all versions). We align the lines of the documents into chunks, so that these unchanged lines are matched up:

Figure 20.9. Aligning the three versions on unchanged lines

Alice	Original	Bob	
1. celery	1. celery	1. celery	A
-----	2. garlic	2. salmon	B
2. salmon	3. onions	3. garlic	
-----	4. salmon	4. onions	
-----	3. tomatoes	5. tomatoes	C
-----	5. tomatoes		
4. garlic			D
5. onions			
-----	6. wine	6. wine	E

Having done this, the merge result is generated by scanning through these chunks. For chunks where all three versions agree — chunks A, C and E above — we can just output the original text since neither Alice nor Bob changed it. For chunks like B where both Alice and Bob differ from the original, we have a conflict; the merge algorithm, having no understanding of the meaning of the text, cannot decide how to resolve this and the conflict is emitted for the user to resolve by hand. In contrast, in chunk D Bob’s version is equal to the original (in this case it is empty) while Alice’s differs. In chunks like this where only one new version differs from the original, we pick the version that changed.

To summarise, for each block we emit the following:

- *A*: all equal; emit `celery`
- *B*: Both differ; emit all versions as a conflict
- *C*: all equal; emit `tomatoes`
- *D*: Alice differs; emit `garlic`, `onions`
- *E*: all equal; emit `wine`

When Git emits these results as into the target file, conflicts are denoted by a `<<<<<` line, followed by Alice's version, then `=====` followed by Bob's version, then a `>>>>>` line:

```
celery
<<<<< Alice
salmon
=====
salmon
garlic
onions
>>>>> Bob
tomatoes
garlic
onions
wine
```

If you set `merge.conflictStyle = diff3` in your Git config, or use the standalone `diff3` program, you'll also see the original version of a conflicted chunk, denoted by the `|||||||` line:

```
celery
<<<<< Alice
salmon
||||||| original
garlic
onions
salmon
=====
salmon
garlic
onions
>>>>> Bob
tomatoes
garlic
onions
wine
```

20.1.2. Implementing diff3

Now that we understand at a high level how the algorithm works, let's look at implementing it. I'm going to create a class for doing this, since the algorithm uses several bits of state that I'd rather store as instance variables, rather than passing them between functions. We'll start by creating a method called `Merge::Diff3.merge` that takes `o`, `a` and `b`; the original version and the two versions to be merged.

After creating a new `Diff3` object we call `merge` on it, which is going to set up some initial state, then generate the output chunks, and then return a new `Result` object containing those chunks. We'll see the `Result` class in more detail later.

```
# lib/merge/diff3.rb

module Merge
  class Diff3

    def self.merge(o, a, b)
      o = o.lines if o.is_a?(String)
      a = a.lines if a.is_a?(String)
      b = b.lines if b.is_a?(String)

      Diff3.new(o, a, b).merge
    end

    def initialize(o, a, b)
      @o, @a, @b = o, a, b
    end

    def merge
      setup
      generate_chunks
      Result.new(@chunks)
    end

  end
end
```

First, let's look at the `setup` method, which initialises a few bits of state needed to run the algorithm. It creates an empty list called `@chunks` that we will append new chunks to as we discover them. Then it sets a `@line` variable for each document `o`, `a` and `b` to record how far we have scanned through all the documents; this begins at zero for all of them. Finally, we create two *match sets* that record which lines in each version are equal to the original, and identifies those matches by line number.

```
# lib/merge/diff3.rb

def setup
  @chunks = []
  @line_o = @line_a = @line_b = 0

  @match_a = match_set(@a)
  @match_b = match_set(@b)
end
```

To recap Chapter 11, *The Myers diff algorithm*, the `Diff.diff` method returns an array of `Edit` objects, each of which has a type which is one of `:del`, `:ins` or `:eq1`. Each `Edit` also has an `a_line` and a `b_line`, both of which are `Line` values representing lines from each of the input strings. For example, if we call `Diff.diff(a, b)` where `a` is the `Original` string above, and `b` is the `Bob` version:

```
>> Diff.diff(a, b)
=> [
  Edit(type=:eq1, a_line=Line(1, "celery"), b_line=Line(1, "celery")),
```

```

Edit(type=:ins, a_line=nil,           b_line=Line(2, "salmon")),
Edit(type=:eq1, a_line=Line(2, "garlic"), b_line=Line(3, "garlic")),
Edit(type=:eq1, a_line=Line(3, "onions"), b_line=Line(4, "onions")),
Edit(type=:del, a_line=Line(4, "salmon"), b_line=nil),
Edit(type=:eq1, a_line=Line(5, "tomatoes"), b_line=Line(5, "tomatoes")),
Edit(type=:eq1, a_line=Line(6, "wine"),     b_line=Line(6, "wine"))
]

```

The `a_line` values of the edits are the lines from the first `Diff.diff` input, and the `b_line` values come from the second input. The `:eq1` edits tell us which lines have been matched up between the two documents. We use this in `match_set` to build an index of line numbers in the `a` input and their matched line number in the `b` input.

```

# lib/merge/diff3.rb

def match_set(file)
  matches = {}

  Diff.diff(@o, file).each do |edit|
    next unless edit.type == :eq1
    matches[edit.a_line.number] = edit.b_line.number
  end

  matches
end

```

For our ingredient list example above, the match sets will look like this:

```

@match_a = { 1 => 1, 4 => 2, 5 => 3, 6 => 6 }
@match_b = { 1 => 1, 2 => 3, 3 => 4, 5 => 5, 6 => 6 }

```

Now we come to the main body of the algorithm. The `generate_chunks` method loops through a sequence of steps for finding and recording matching and differing chunks until it's reached the end of all the documents. Each turn of the loop performs the following steps:

- Find the start of the next non-matching chunk, returning `i` as the number of lines away from our current position that is. (If we're in a matching chunk, then the end of that chunk will be the same number of lines away in all three documents.)
- If `i` is 1, then we're already in a non-matching chunk and we need to find the start of the next matching one. We try to find the start of the next matching chunk and record `o`, `a` and `b` as the line offset in each document of the start of that chunk.
 - If we found a match and `a` and `b` are therefore set, we emit a chunk up to the offsets we just found.
 - Otherwise, there are no further matches and we emit the remainder of all documents as the final chunk.
- If `i` exists and is not 1, then we've found the start of the next non-match and we can emit a chunk up to `i` steps from our current line offsets.
- If `i` does not exist, then we've searched to the end of all three documents and we emit the remainder of all of them as the final chunk.

Here is an implementation that draws that process out, leaving the implementation of finding the next match or mismatch, and emitting chunks, to other methods.

```
# lib/merge/diff3.rb

def generate_chunks
  loop do
    i = find_next_mismatch

    if i == 1
      o, a, b = find_next_match

      if a and b
        emit_chunk(o, a, b)
      else
        emit_final_chunk
        return
      end

    elsif i
      emit_chunk(@line_o + i, @line_a + i, @line_b + i)

    else
      emit_final_chunk
      return
    end
  end
end
```

Now we just need the methods for finding the next match or mismatch, and for emitting chunks.

To find the start of the next mismatch, we start a counter *i* at 1, and step through each document line-by-line from our current position. If *i* has not counted past the end of all the documents, and if the current line in all the documents match, we increment *i* and keep looping. Once the loop stops, if *i* is still within the bounds of any of the documents then we return it, otherwise we return nothing.

```
# lib/merge/diff3.rb

def find_next_mismatch
  i = 1
  while in_bounds?(i) and
    match?(@match_a, @line_a, i) and
    match?(@match_b, @line_b, i)
    i += 1
  end
  in_bounds?(i) ? i : nil
end

def in_bounds?(i)
  @line_o + i <= @o.size or
  @line_a + i <= @a.size or
  @line_b + i <= @b.size
end

def match?(matches, offset, i)
  matches[@line_o + i] == offset + i
```

```
end
```

The `match?` method works by using the match sets we created at the start of the process. Rather than scanning through and comparing the actual text of the documents for equality, we use these match sets as an index. We know that line `(@line_a + i)` in document `@a` matches line `(@line_o + i)` in the original, if `@match_a` contains a mapping from the original line number to that in `@a`.

For instance, when we run this code at the beginning of our recipe example, `@line_o`, `@line_a` and `@line_b` are all `0`. We run the loop with `i = 1`, and checking the match sets we see that `@match_a[1] == 1` and `@match_b[1] == 1`, so we try again with `i = 2`. On this turn, the value of `@match_b[2]` is `3` and `@match_a` doesn't even have an entry for `2`, so we've found the start of a non-matching chunk. This value of `i` means the next non-matching chunk begins on line `2` relative to our current position, so we can emit one line from each document as a matching chunk. The `emit_chunk(@line_o + i, @line_a + i, @line_b + i)` call in `generate_chunks` accomplishes this.

Finding the start of the next match is a little simpler. We start a counter `o` at one more than our current `@line_o` offset, and we increment it until either it exceeds the size of `@o` or until both match sets have an entry for that line number, indicating that both diffs leave that line unchanged. We then return the final value of `o`, and the corresponding line numbers from each match set. If we didn't find any matches, these latter two values will be `nil`.

```
# lib/merge/diff3.rb

def find_next_match
  o = @line_o + 1
  until o > @o.size || (@match_a.has_key?(o) and @match_b.has_key?(o))
    o += 1
  end
  [o, @match_a[o], @match_b[o]]
end
```

Returning to our example, after emitting the first matching chunk, `@line_o`, `@line_a` and `@line_b` are all `1`, and recall that the match sets are as follows, the line numbers in `@o` appearing on the left-hand side of the arrows:

```
@match_a = { 1 => 1, 4 => 2, 5 => 3, 6 => 6 }
@match_b = { 1 => 1, 2 => 3, 3 => 4, 5 => 5, 6 => 6 }
```

We start with `o = 2`. `@match_b` has an entry for `2` but `@match_a` does not, so we try `o = 3`. Again, `@match_b` has an entry but `@match_a` does not, so we try `o = 4`. This time, `@match_a` has an entry but `@match_b` does not. Finally we try `o = 5` and see that both match sets contain an entry: `@match_a[5] = 3` and `@match_b[5] = 5`. So, we return `[5, 3, 5]` in this instance and `generate_chunks` calls `emit_chunk(5, 3, 5)`.

The methods for emitting chunks are what construct the output of the merge algorithm, and keep the line offsets up to date, moving them to the end of the chunk we just emitted in each document. `emit_chunk` creates a chunk from the current line offsets up to a given index in each document, and then sets the line offsets to the end of these ranges. We subtract one from all the offsets we're given because the diffs and the finder methods use 1-indexed line numbers, but our `@line` variables are 0-indexed offsets from the start of each document. `emit_final_chunk` just emits whatever is left from all the documents from their current line offsets onwards.

```
# lib/merge/diff3.rb

def emit_chunk(o, a, b)
  write_chunk(
    @o[@line_o ... o - 1],
    @a[@line_a ... a - 1],
    @b[@line_b ... b - 1])

  @line_o, @line_a, @line_b = o - 1, a - 1, b - 1
end

def emit_final_chunk
  write_chunk(
    @o[@line_o ... -1],
    @a[@line_a ... -1],
    @b[@line_b ... -1])
end
```

Finally, we reach the end of our chain of methods with `write_chunk`. This takes a set of lines from each document and emits the appropriate kind of chunk depending on their contents. If all three sets are equal, then we emit a `Clean` chunk object containing the original version; if Alice's chunk is equal to the original then we emit Bob's and vice versa; and if neither is equal then we emit a `Conflict` chunk containing all three versions.

```
# lib/merge/diff3.rb

def write_chunk(o, a, b)
  if a == o or a == b
    @chunks.push(Clean.new(b))
  elsif b == o
    @chunks.push(Clean.new(a))
  else
    @chunks.push(Conflict.new(o, a, b))
  end
end
```

Once `generate_chunks` finishes looping, the `merge` method returns `Result.new(@chunks)`, creating a data structure representing the result of the whole process. That data structure comes with a `to_s` method for turning the whole thing into a string that can be written back to a file, but keeping it as a structure lets us inspect it, particularly to check whether the merge was clean or not. While the `to_s` methods assume the original inputs were a series of lines of text, exposing the merge result as a data structure allows us to use this algorithm with other types of input.

We can also use these structures to generate the output text with added data that isn't necessary for the merge algorithm itself, for example the the `Result#to_s` takes two optional arguments which it uses as the names of the two versions to annotate the conflict markers.

```
# lib/merge/diff3.rb

Clean = Struct.new(:lines) do
  def to_s(*)
    lines.join("")
  end
end

Conflict = Struct.new(:o_lines, :a_lines, :b_lines) do
```

```
def to_s(a_name = nil, b_name = nil)
  text = ""
  separator(text, "<", a_name)
  a_lines.each { |line| text.concat(line) }
  separator(text, "=")
  b_lines.each { |line| text.concat(line) }
  separator(text, ">", b_name)
  text
end

def separator(text, char, name = nil)
  text.concat(char * 7)
  text.concat("#{" + name + "}") if name
  text.concat("\n")
end

Result = Struct.new(:chunks) do
  def clean?
    chunks.none? { |chunk| chunk.is_a?(Conflict) }
  end

  def to_s(a_name = nil, b_name = nil)
    chunks.map { |chunk| chunk.to_s(a_name, b_name) }.join("")
  end
end
```

We now have a complete implementation that we can use to merge data from different sources, and print out the result:

```
merged = Diff3.merge(original, alice, bob)

puts merged

# => celery
# <<<<< alice.txt
# salmon
# =====
# salmon
# garlic
# onions
# >>>>> bob.txt
# tomatoes
# garlic
# onions
# wine
```

While the `diff3` algorithm is relatively simple, it is highly sensitive to the output of the underlying diff algorithm. Depending on the lines the underlying diffs mark as equal, a merge can result in a conflict while another choice of diff algorithm could lead to a clean merge.

20.1.3. Using `diff3` during a merge

Having `Diff3.merge` lets us modify the `merge_blobs` method inside `Merge::Resolve` so that, if both sides have edited the file, we try to merge their changes rather than just concatenating both versions.

```
# lib/merge/resolve.rb

def merge_blobs(base_oid, left_oid, right_oid)
  result = merge3(base_oid, left_oid, right_oid)
  return result if result

  oids = [base_oid, left_oid, right_oid]
  blobs = oids.map { |oid| oid ? @repo.database.load(oid).data : "" }
  merge = Diff3.merge(*blobs)

  data = merge.to_s(@inputs.left_name, @inputs.right_name)
  blob = Database::Blob.new(data)
  @repo.database.store(blob)

  [merge.clean?, blob.oid]
end
```

We've replaced the old `merged_data` method with a call to `Diff3.merge`, and we serialise the result with `to_s` and store it as a blob as before. The call to `merge.clean?` returns the boolean that `same_path_conflict` uses to determine whether the path merged cleanly. The `merge` command is now able to merge changes to the same file, as long as those changes affect different regions.

20.2. Logging merge commits

When we amended the `diff` command for conflicts⁴, we came across the concept of a *combined diff*. It's now possible for a merge commit to contain a version of a file that differs from the version in both of its parents, and the `log` command likewise uses combined diffs to display such changes.

Suppose that the versions of some file in the left and right inputs to a merge, and the version in the merge commit, are as follows:

```
left = <<~EOF
  alfa
  bravo
  delta
EOF

right = <<~EOF
  echo
  bravo
  charlie
EOF

merged = <<~EOF
  echo
  bravo
  delta
  foxtrot
EOF
```

When viewing this trio of versions, either with the `diff` command during the merge, or by running `log --cc` after the merge is committed, Git will display the diff as follows:

⁴Section 19.3, “Conflicted diffs”

```
diff --cc f.txt
index d8556a9,9d24e3e..09cba68
--- a/f.txt
+++ b/f.txt
@@@ -1,3 -1,3 +1,4 @@
- alfa
+ echo
  bravo
-charlie
+delta
++foxtrot
```

This diff is characterised by the `--cc` flag on the `diff` line, the three @ symbols on hunk headers, and the multiple columns of plus/minus symbols preceding the line text. It is a combination of the diff between the left version and the merge result, and between the right version and the merge result, which are as follows:

```
>> left_diff = Diff.diff(left, merged)
=> [
    Edit(type=:del, a_line=Line(1, "alfa"), b_line=nil),
    Edit(type=:ins, a_line=nil, b_line=Line(1, "echo")),
    Edit(type=:eql, a_line=Line(2, "bravo"), b_line=Line(2, "bravo")),
    Edit(type=:eql, a_line=Line(3, "delta"), b_line=Line(3, "delta")),
    Edit(type=:ins, a_line=nil, b_line=Line(4, "foxtrot"))
]

>> left_diff.each { |edit| puts edit }

-alfa
+echo
  bravo
  delta
+foxtrot

>> right_diff = Diff.diff(right, merged)
=> [
    Edit(type=:eql, a_line=Line(1, "echo"), b_line=Line(1, "echo")),
    Edit(type=:eql, a_line=Line(2, "bravo"), b_line=Line(2, "bravo")),
    Edit(type=:del, a_line=Line(3, "charlie"), b_line=nil),
    Edit(type=:ins, a_line=nil, b_line=Line(3, "delta")),
    Edit(type=:ins, a_line=nil, b_line=Line(4, "foxtrot"))
]

>> right_diff.each { |edit| puts edit }

  echo
  bravo
-charlie
+delta
+foxtrot
```

If we compare these diffs to the combined diff above, we see that the first column of the combined diff represents the left changes and the second column the right. We also see that lines that appear in both diffs are combined into a single change. For example, `foxtrot` appears with the prefix `++` as it's added in both diffs, but also `echo` appears with a plus in the first column and nothing in the second — it's added in the left diff but an unchanged line in the

right. Similarly, delta has a plus in the second column. bravo has nothing in either column; it's unchanged in both diffs.

The above code listing illustrates how the list of `Diff::Edit` objects returned by `Diff.diff` correspond to the printed output. If we examine the combined diff, we see that its lines of text could correspond to a list of *pairs* of `Edit` objects, one from the left and one from the right. In fact, it corresponds to this recombination of the above diff structures:

```
[  
  [ Edit(:del, Line(1, "alfa"), nil),    nil           ],  
  [ Edit(:ins, nil, Line(1, "echo")),     Edit(:eql, ..., Line(1, "echo")) ],  
  [ Edit(:eql, ..., Line(2, "bravo")),   Edit(:eql, ..., Line(2, "bravo")) ],  
  [ nil,                                Edit(:del, Line(3, "charlie"), nil) ],  
  [ Edit(:eql, ..., Line(3, "delta")),   Edit(:ins, nil, Line(3, "delta")) ],  
  [ Edit(:ins, nil, Line(4, "foxtrot")), Edit(:ins, nil, Line(4, "foxtrot")) ]  
]
```

(I have omitted the `type`, `a_line` and `b_line` labels to save space, but the fields of the `Edit` objects appear in their usual order. The expression `...` indicates a `Line` object whose text is equal to the other `Line` in the edit, which happens for edits with the `:eql` type.)

The first members of each pair are the `Edit` objects from the left diff, and the second members are those from the right. Because `left_diff` and `right_diff` both result from calling `Diff.diff` with `merged` as the second input, their edits will contain the same sequence of values in their `b_line` fields: the lines from the `merged` version. They can therefore be paired up by finding edits with the same `b_line`.

Deletions do not have a `b_line` and so cannot be paired up — they sit in a row by themselves with `nil` filling the other columns. Insertions and unchanged lines can be matched up with each other: both have a `b_line` value, but only `:eql` edits also have an `a_line`. So in a combined diff, each row contains either a single deletion, or some combination of insertions and unchanged lines all having the same text.

For example, take our two diffs we saw above. We'll also make an empty list called `combined` where we'll build up our combined diff.

```
left_diff = [  
  Edit(type=:del, a_line=Line(1, "alfa"), b_line=nil),  
  Edit(type=:ins, a_line=nil, b_line=Line(1, "echo")),  
  Edit(type=:eql, a_line=Line(2, "bravo"), b_line=Line(2, "bravo")),  
  Edit(type=:eql, a_line=Line(3, "delta"), b_line=Line(3, "delta")),  
  Edit(type=:ins, a_line=nil, b_line=Line(4, "foxtrot"))]  
  
right_diff = [  
  Edit(type=:eql, a_line=Line(1, "echo"), b_line=Line(1, "echo")),  
  Edit(type=:eql, a_line=Line(2, "bravo"), b_line=Line(2, "bravo")),  
  Edit(type=:del, a_line=Line(3, "charlie"), b_line=nil),  
  Edit(type=:ins, a_line=nil, b_line=Line(3, "delta")),  
  Edit(type=:ins, a_line=nil, b_line=Line(4, "foxtrot"))]  
  
combined = []
```

`left_diff` begins with a deletion, so we'll immediately shift that off into a pair in the combined list, with `nil` in the second column.

```
left_diff = [
    Edit(type=:ins, a_line=nil, b_line=Line(1, "echo")),
    Edit(type=:eql, a_line=Line(2, "bravo"), b_line=Line(2, "bravo")),
    Edit(type=:eql, a_line=Line(3, "delta"), b_line=Line(3, "delta")),
    Edit(type=:ins, a_line=nil, b_line=Line(4, "foxtrot"))
]

right_diff = [
    Edit(type=:eql, a_line=Line(1, "echo"), b_line=Line(1, "echo")),
    Edit(type=:eql, a_line=Line(2, "bravo"), b_line=Line(2, "bravo")),
    Edit(type=:del, a_line=Line(3, "charlie"), b_line=nil),
    Edit(type=:ins, a_line=nil, b_line=Line(3, "delta")),
    Edit(type=:ins, a_line=nil, b_line=Line(4, "foxtrot"))
]

combined = [
    [Edit(:del, Line(1, "alfa")), nil], nil
]
```

Now, the next two edits in both `left_diff` and `right_diff` contain a `b_line` — a line from the merged version. We can shift these off in pairs and add them to the `combined` list.

```
left_diff = [
    Edit(type=:eql, a_line=Line(3, "delta"), b_line=Line(3, "delta")),
    Edit(type=:ins, a_line=nil, b_line=Line(4, "foxtrot"))
]

right_diff = [
    Edit(type=:del, a_line=Line(3, "charlie"), b_line=nil),
    Edit(type=:ins, a_line=nil, b_line=Line(3, "delta")),
    Edit(type=:ins, a_line=nil, b_line=Line(4, "foxtrot"))
]

combined = [
    [Edit(:del, Line(1, "alfa")), nil], nil
    [Edit(:ins, nil, Line(1, "echo")), Edit(:eql, ..., Line(1, "echo"))],
    [Edit(:eql, ..., Line(2, "bravo")), Edit(:eql, ..., Line(2, "bravo"))]
]
```

We now see a deletion at the head of `right_diff`, so we'll shift that into the `combined` list with a `nil` in the first column.

```
left_diff = [
    Edit(type=:eql, a_line=Line(3, "delta"), b_line=Line(3, "delta")),
    Edit(type=:ins, a_line=nil, b_line=Line(4, "foxtrot"))
]

right_diff = [
    Edit(type=:ins, a_line=nil, b_line=Line(3, "delta")),
    Edit(type=:ins, a_line=nil, b_line=Line(4, "foxtrot"))
]

combined = [
    [Edit(:del, Line(1, "alfa")), nil], nil
    [Edit(:ins, nil, Line(1, "echo")), Edit(:eql, ..., Line(1, "echo"))],
    [Edit(:eql, ..., Line(2, "bravo")), Edit(:eql, ..., Line(2, "bravo"))]
]
```

```
[ Edit(:eql, ..., Line(2, "bravo")), Edit(:eql, ..., Line(2, "bravo")) ],
[ nil,                                     Edit(:del, Line(3, "charlie"), nil) ]
```

And finally, `left_diff` and `right_diff` now only contain edits with a `b_line` that can be paired up, leaving us with:

```
left_diff = []
right_diff = []

combined = [
  [ Edit(:del, Line(1, "alfa"), nil),   nil           ],
  [ Edit(:ins, nil, Line(1, "echo")),   Edit(:eql, ..., Line(1, "echo")) ],
  [ Edit(:eql, ..., Line(2, "bravo")), Edit(:eql, ..., Line(2, "bravo")) ],
  [ nil,                                Edit(:del, Line(3, "charlie"), nil) ],
  [ Edit(:eql, ..., Line(3, "delta")), Edit(:ins, nil, Line(3, "delta")) ],
  [ Edit(:ins, nil, Line(4, "foxtrot")), Edit(:ins, nil, Line(4, "foxtrot")) ]
]
```

This process generalises to any number of diffs, with the rows in the combined diff having one column per input diff. At each step, we shift off any deletions from the input diffs and put them in their own rows, and then we add a combined row where all the elements are insertions or unchanged lines.

Let's write a class the performs this process. As input it will take a list of diffs, which are arrays of `Edit` objects.

```
# lib/diff/combined.rb

module Diff
  class Combined

    def initialize(diffs)
      @diffs = diffs
    end

    #
    end
  end
end
```

We can generate a list from this class by including the `Enumerable`⁵ module and implementing the `each` method to yield rows of the combined diff; calling `to_a` on this class will then return an array containing all the rows.

Rather than actually removing edits from the front of the diffs, we'll track how far we've scanned into each one. `@offsets = @diffs.map { 0 }` creates an array with one element for each diff, where initially each offset is zero. Then in a loop, we check each diff for any deletions we can consume, and yield a row for each one. We check if we've reached the end of all the diffs and return if so. The final step in the loop fetches the current edit from all the diffs (which will all be insertions or unchanged), and advances the offsets of each diff by one, before yielding a row containing these grouped edits.

⁵<https://docs.ruby-lang.org/en/2.3.0/Enumerable.html>

```
# lib/diff/combined.rb

include Enumerable

def each
  @offsets = @diffs.map { [] }

  loop do
    @diffs.each_with_index do |diff, i|
      consume_deletions(diff, i) { |row| yield row }
    end

    return if complete?

    edits = offset_diffs.map { |offset, diff| diff[offset] }
    @offsets.map! { |offset| offset + 1 }

    yield Row.new(edits)
  end
end
```

The `consume_deletions` method takes a diff and its position in the `@diffs` array. While that diff's current offset points at a deletion edit, we yield a row containing just that edit, in the appropriate column. `Array.new(@diffs.size)` makes an array the same size as `@diffs`, and we place the current edit in the correct column of that array before advancing the diff's offset and yielding the row.

```
# lib/diff/combined.rb

def consume_deletions(diff, i)
  while @offsets[i] < diff.size and diff[@offsets[i]].type == :del
    edits = Array.new(@diffs.size)
    edits[i] = diff[@offsets[i]]
    @offsets[i] += 1

    yield Row.new(edits)
  end
end
```

There are two other small helper methods in this class. `offset_diffs` returns a list of pairs of `[offset, diff]` for each diff, using `Enumerable#zip`⁶. `complete?` returns true if we've consumed all the input, that is all the offsets are equal to the corresponding diff's size.

```
# lib/diff/combined.rb

def offset_diffs
  @offsets.zip(@diffs)
end

def complete?
  offset_diffs.all? { |offset, diff| offset == diff.size }
end
```

⁶<https://docs.ruby-lang.org/en/2.3.0/Enumerable.html#method-i-zip>

The `Diff::Combined` class generates an array of `Row` objects, which are defined as follows. They just exist to hold an array of edits and turn them into a string, with the appropriate plus/minus markers and text selected from the `Edit` objects.

```
# lib/diff/combined.rb

Row = Struct.new(:edits) do
  def to_s
    symbols = edits.map { |edit| SYMBOLS.fetch(edit.type, " ") }

    del = edits.find { |edit| edit.type == :del }
    line = del ? del.a_line : edits.first.b_line

    symbols.join("") + line.text
  end
end
```

Finally, a helper method on the `Diff` module will let us pass in an array of merged file versions (as), and the merge result (b), and receive the combined diff in return.

```
# lib/diff.rb

def self.combined(as, b)
  diffs = as.map { |a| Diff.diff(a, b) }
  Combined.new(diffs).to_a
end
```

20.2.1. Unifying hunks

Just like normal diffs, combined diffs are displayed in hunks, highlighting just the changed parts and omitting long stretches of unchanged content. Here's our example again:

```
@@@ -1,3 -1,3 +1,4 @@@
- alfa
+ echo
  bravo
- charlie
+ delta
++foxtrot
```

Whereas normal diffs have hunk headers that look like `@@ -a, b +c, d @@`, these headers have three @ symbols and they have multiple offsets listed with a minus sign, but still only one offset with a plus sign. That's because all the diffs we've combined have the same `b_line` values, but different `a_line` values; they're all diffs from some prior version to the merge result. The minus-signed offsets are the offsets of those `a_line` values from the pre-merge versions. Is there a way we can make our existing `Diff::Hunk` code process a list of `Combined::Row` objects to get this output?

Well let's look at what `Hunk` relies on. It currently takes a list of `Edit` objects, where `Edit` and `Line` are defined as:

```
Edit = Struct.new(:type, :a_line, :b_line)
Line = Struct.new(:number, :text)
```

`Diff::Hunk` expects to be able to call `type` on its inputs and get one of `:eq1`, `:ins` or `:del`. It expects to be able to call `edit.a_line.number` or `edit.b_line.text` and get something

meaningful. To make `Combined::Row` objects work, we need to make them respond to this interface.

Defining type for `Combined::Row` is fairly simple. `Diff::Hunk` wants to know the type so it can check for unchanged lines, and `Command::PrintDiff` wants to use it to select the colour in which to print the edit. A row contains either one `:del` edit, or multiple `:eq1/:ins` edits. We'll make `Row#type` return `:del` or `:ins` if any of its edits are of that type, otherwise we'll return `:eq1`.

```
# lib/diff/combined.rb

Row = Struct.new(:edits) do
  # ...

  def type
    types = edits.compact.map(&:type)
    types.include?(:ins) ? :ins : types.first
  end
end
```

Defining `Row#b_line` is even simpler. All the edits in a `Row` have the same `b_line` — it's either `nil` or it's the line they were matched up on. So we can just return the `b_line` of the first edit in the row.

```
# lib/diff/combined.rb

Row = Struct.new(:edits) do
  # ...

  def b_line
    edits.first&.b_line
  end
end
```

Defining `Row#a_line` is more tricky: a row contains multiple edits and all of them have a different `a_line`. Which do we pick? The answer is, we don't pick: we use all of them, and adjust the `Hunk` code to handle this.

`Diff::Hunk` currently assumes `Edit` objects respond to `a_line` by returning a single `Line` object. But our combined hunk headers need to display multiple `a_line` offsets. This can be resolved by realising that having a single value is just a special case of having multiple values. What if we adjust `Diff::Hunk` so that everywhere it expected a single `a_line`, it now expects an array of them? The `filter` method changes like so:

```
# lib/diff/hunk.rb

Hunk = Struct.new(:a_starts, :b_start, :edits) do
  def self.filter(edits)
    # ...

    loop do
      # ...

      a_starts = (offset < 0) ? [] : edits[offset].a_lines.map(&:number)
      b_start = (offset < 0) ? nil : edits[offset].b_line.number
```

```

        hunks.push(Hunk.new(a_starts, b_start, []))
        offset = Hunk.build(hunks.last, edits, offset)
    end
end

# ...
end

```

And the `Hunk#header` method can be similarly adjusted so that it prints one @ symbol for each file version included in the diff.

```

# lib/diff/hunk.rb

def header
  a_lines = edits.map(&:a_line).transpose
  offsets = a_lines.map.with_index { |lines, i| format("-", lines, a_starts[i]) }

  offsets.push(format("+", edits.map(&:b_line), b_start))
  sep = "@" * offsets.size

  [sep, *offsets, sep].join(" ")
end

private

def format(sign, lines, start)
  lines = lines.compact
  start = lines.first&.number || start || 0

  "#{sign}#{start},#{lines.size}"
end

```

Now, `Row#a_lines` can be simply defined to return the `a_line` of all the edits:

```

# lib/diff/combined.rb

Row = Struct.new(:edits) do
  # ...

  def a_lines
    edits.map { |edit| edit.a_line }
  end
end

```

And to make `Diff::Edit` compatible with the changes to `Diff::Hunk`, we need it to respond to `a_lines` by returning its single `a_line` wrapped in an array.

```

# lib/diff.rb

Edit = Struct.new(:type, :a_line, :b_line) do
  # ...

  def a_lines
    [a_line]
  end
end

```

This unification of the `Edit` and `Row` interfaces allows us to use the same presentation code for both types of diff. All we need now is a convenient function to take a list of pre-merge versions and a merge result, and return the hunk-filtered combined diff of them all.

```
# lib/diff.rb

def self.combined_hunks(as, b)
  Hunk.filter(Diff.combined(as, b))
end
```

20.2.2. Diffs during merge conflicts

We can now use combined diffs to complete some functionality we skipped over in Section 19.3, “Conflicted diffs”: showing a combined diff for conflicted files. In `Command::Diff`, we can adjust `print_conflict_diff` so that it does a bit more than just printing `Unmerged path` if you didn’t specify a stage.

```
# lib/command/diff.rb

def print_conflict_diff(path)
  targets = (0..3).map { |stage| from_index(path, stage) }
  left, right = targets[2], targets[3]

  if @options[:stage]
    puts "* Unmerged path #{path}"
    print_diff(targets[@options[:stage]], from_file(path))
  elsif left and right
    print_combined_diff([left, right], from_file(path))
  else
    puts "* Unmerged path #{path}"
  end
end
```

`Command::PrintDiff#print_combined_diff` is very similar to the `print_diff` method, it just presents the header information a little differently to a regular diff. After that, displaying the hunks and edits works exactly the same way.

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

def print_combined_diff(as, b)
  header("diff --cc #{b.path}")

  a_oids = as.map { |a| short a.oid }
  oid_range = "index #{a_oids.join(',')}\n#{short b.oid}"
  header(oid_range)

  unless as.all? { |a| a.mode == b.mode }
    header("mode #{as.map(&:mode).join(',')}\n#{b.mode}")
  end

  header("--- a/#{b.diff_path}")
  header("+++ b/#{b.diff_path}")

  hunks = ::Diff.combined_hunks(as.map(&:data), b.data)
  hunks.each { |hunk| print_diff_hunk(hunk) }
end
```

20.2.3. Diffs for merge commits

We can also use combined diffs to accurately display merge commits, which will help people reviewing changes on a branch before merging it, should that branch itself contain merges. Let's add a method to `Database::Commit` to check if it's a merge:

```
# lib/database/commit.rb

def merge?
  @parents.size > 1
end
```

We'll define the `--cc` option to the `log` command so that it sets the `:patch` and `:combined` options.

```
# lib/command/log.rb

def define_options
  #
  # ...

@parser.on "--cc" do
  @options[:combined] = @options[:patch] = true
end
end
```

If both these options are set, then the `log` command will invoke this new method, `show_merge_patch`. Taking a commit, it works out the tree diff between each of the commit's parents and itself. It then selects the paths that appear in all the commits, by filtering `diffs.first.keys` against the keys in the other diffs, `diffs.drop(1)`. For each selected path, it selects the pre-images for the file from each tree diff and uses these to print a combined diff, using the `print_combined_diff` method we defined above.

```
# lib/command/log.rb

def show_patch(commit)
  return unless @options[:patch]
  return show_merge_patch(commit) if commit.merge?

  #
end

def show_merge_patch(commit)
  return unless @options[:combined]

  diffs = commit.parents.map { |oid| @rev_list.tree_diff(oid, commit.oid) }

  paths = diffs.first.keys.select do |path|
    diffs.drop(1).all? { |diff| diff.has_key?(path) }
  end

  blank_line

  paths.each do |path|
    parents = diffs.map { |diff| from_entry(path, diff[path][0]) }
    child   = from_entry(path, diffs.first[path][1])
  end
end
```

```
    print_combined_diff(parents, child)
  end
end
```

One other small change we can make to the log command while we're here: for merge commits, the medium format prints a line showing the IDs of the merge commit's parents before displaying all the other usual details.

```
# lib/command/log.rb

def show_commit_medium(commit)
  blank_line
  puts fmt(:yellow, "commit #{ abbrev(commit) }") + decorate(commit)

  if commit.merge?
    oids = commit.parents.map { |oid| repo.database.short_oid(oid) }
    puts "Merge: #{ oids.join(" ") }"
  end

  #
end
```

21. Correcting mistakes

One of Git’s strengths is that it tries very hard to never lose your data. Once an object is written to the database, it should never change — if it’s changed, it should have a different ID and therefore be in a different file. Git can check whether a file has been corrupted by re-hashing the object inside it, and comparing the result to the object’s ID.

This also means that once a commit is written, neither it nor anything it points to can be changed. If you change its tree, the tree would have a new ID, which would change the commit, and so change the commit’s ID. The same goes for the commit’s parents; a commit contains the IDs of its parent commits, so if those are changed, you have a different commit. So, every commit ID in Git implies a unique and unchangeable history, and its ID is a function of all the content in all the commits that lead up to it.

That resistance to change is reassuring, but sometimes we do want to change things. While writing each commit, we’d like to change what’s in the index, removing things that should no longer be there. We might decide something we’ve added to the index should be reset to its version in `HEAD`. Or, we might decide a commit is wrong after we’ve made it, and we’d like to replace it with a different commit. In this chapter we’ll look at all these problems, and in the following chapter we’ll look at ways of changing the commit history more broadly.

21.1. Removing files from the index

So far, we’ve just been adding more and more code to our repository. The tree we write into each commit has only gained new files, never lost them. There was one exception to this: in Section 5.2, “Nested trees”, we migrated the project from a flat directory structure to one with nested trees. But that was before the index existed! Since then, the index has only grown bigger, accumulating new files with every command.

What if we want to remove a file? Technically, there is a way of doing this: we can delete the file `.git/index` and then rebuild it by adding just the files we want to keep. But this is a tedious and error-prone process, especially if the index is already in a mixed state containing some staged changes, and there are other changes we don’t want staged yet. It would be much more direct to remove the unnecessary files with a single command.

As it happens, our repository does contain a redundant file: `lib/entry.rb` is a relic from the very early commits where we developed the `Tree` class, and it was later superseded by the `Index::Entry` and `Database::Entry` classes, which represent references to blobs in the index and in tree objects. The root `Entry` class is no longer used and we can delete it, but first we need the tools to do so.

Below is the beginnings of an `rm` command. In Git, this command takes a list of pathnames and removes those paths from the index and from the workspace. That’s not too complicated with the methods we already have.

```
# lib/command/rm.rb

module Command
  class Rm < Base
```

```
def run
  repo.index.load_for_update
  @args.each { |path| remove_file(path) }
  repo.index.write_updates

  exit 0
end

private

def remove_file(path)
  repo.index.remove(path)
  repo.workspace.remove(path)
  puts "rm '#{ path }'"
end

end
end
```

This just requires one new method: `workspace#remove`, which removes a single file. I'm rescuing `ENOENT` here because I don't actually mind if it turns out the file does not exist—if it's already absent, then we get the same end result.

```
# lib/workspace.rb

def remove(path)
  File.unlink(@pathname.join(path))
rescue Errno::ENOENT
end
```

What we *should* care about, however, is the possibility of accidentally losing data, or of deleting the wrong file. So next we'll look at how Git's `rm` tries to prevent such problems.

21.1.1. Preventing data loss

The regular Unix `rm`¹ command is notoriously unsafe. Typically, if you delete files using the graphical interface, your computer won't immediately delete them but move them to the *Trash* or the *Recycle Bin*. You can restore accidentally deleted files from there, or permanently remove them. But the `rm` command just deletes the file immediately, no questions asked.

Git is supposed to stop you losing data, so its `rm` is a little safer. By default, it will only remove a file if the versions of it stored in the index and the workspace match what's in the `HEAD` commit, i.e. there are no uncommitted or unstaged changes. If the index and workspace are in sync with `HEAD`, that means all their content is safely stored away in a commit and you can retrieve it if it's lost.

The `rm` command checks all the arguments you pass to make sure it's safe to remove all of them, and only if this generates no errors does it go ahead and delete anything. If you ask to remove a file that's not in the index, that's also an error.

To carry out these checks, we can use the `Repository::Inspector` class². In `Command::Rm#run`, we'll load the `HEAD` ID, create an `Inspector` instance, and create arrays to hold a couple of

¹<https://manpages.ubuntu.com/manpages/bionic/en/man1/rm.1posix.html>

²Section 14.5, “Preventing conflicts”

different types of errors. Then we'll check all the arguments for changes, and exit if any errors are found. Only if we get past this stage do we remove anything.

```
# lib/command/rm.rb

def run
  repo.index.load_for_update

  @head_oid    = repo.refs.read_head
  @inspector   = Repository::Inspector.new(repo)
  @uncommitted = []
  @unstaged   = []

  @args.each { |path| plan_removal(Pathname.new(path)) }
  exit_on_errors

  @args.each { |path| remove_file(path) }
  repo.index.write_updates

  exit 0

rescue => error
  repo.index.release_lock
  @stderr.puts "fatal: #{error.message}"
  exit 128
end
```

The `plan_removal` method is where we check each file for changes, and add to the error lists if we find anything. The `compare_*` methods on `Inspector` return a symbol like `:added`, `:modified` or `:deleted` if there's a difference detected, otherwise they return `nil`.

```
# lib/command/rm.rb

def plan_removal(path)
  unless repo.index.tracked_file?(path)
    raise "pathspec '#{path}' did not match any files"
  end

  item  = repo.database.load_tree_entry(@head_oid, path)
  entry = repo.index.entry_for_path(path)
  stat  = repo.workspace.stat_file(path)

  if @inspector.compare_tree_to_index(item, entry)
    @uncommitted.push(path)
  elsif stat and @inspector.compare_index_to_workspace(entry, stat)
    @unstaged.push(path)
  end
end
```

`Database#load_tree_entry` is a new method for fetching a single `Database::Entry` from a commit by pathname. It takes a commit ID and a path, and loads the given commit to find its tree pointer. Then it breaks the path into its constituent filenames and walks down the tree to find the correct entry. For example, say we have the tree ID `27aa5de...` and the path `lib/database/tree.rb`. If we look at the root tree object, we get:

```
$ git cat-file -p 27aa5de18a4faf281dd57e3a7b51a518b8a77b51
```

```

100644 blob f288702d2fa16d3cdf0035b15a9fcbe552cd88e7    LICENSE.txt
100644 blob c8c218e0d599831338af0748af283a149b9ff8d2    Rakefile
040000 tree aaf2979e4659d597e74b17113dc5021af47547f1    bin
040000 tree 74dc30f7343f385ce50cfdd057b1aaaf4439fb53a    lib
040000 tree c35b962bf69faff30c95df70f07839dbdac3f51    test

```

The first element of the path is `lib` and we see that this has ID `74dc30f...` in this tree. So, we go and load that tree:

```

$ git cat-file -p 74dc30f7343f385ce50cfdd057b1aaaf4439fb53a

100644 blob 5adfddd8e98f4c02ab9affdc60736ed09d89b159    color.rb
100644 blob f56d4fc1daa7cc3a6ea3981870960dd639bc8c99    command.rb
040000 tree 4868804d2876d5113bc8f99e2c85cd7f1bcf61b2    command
100644 blob ff0ef6ddeaa5990ce27f5141d8a81164d25540045    database.rb
040000 tree e199b95109e92658ffdbd6a9d881f9f46058c139    database
...

```

The next part of the path is `database`, which has ID `e199b95...` in this sub-tree. Loading that tree gives us:

```

$ git cat-file -p e199b95109e92658ffdbd6a9d881f9f46058c139

100644 blob cb18dd1522725f068376a99a9739d4b2f230d7bd    author.rb
100644 blob 38fa8cbfd484894d4d727ec5d701d6f0152d5ab0    blob.rb
100644 blob 527e1a5cb86cc0d5e43b371785ff7b146e6c2539    commit.rb
100644 blob 28024a0d9c981e40531e343f98b0f8834859b5a7    entry.rb
100644 blob a37f716437edaee8169a409bc8c2ff82345cd8ba    tree.rb
100644 blob 0b8b7fa41f6b44e720b1aca980f6841b7c0e174a    tree_diff.rb

```

Now we've found the blob we were looking for: `tree.rb` has ID `a37f716...`. So, we return the value `Database::Entry(oid="a37f716...", mode=0100644)`. The method that carries out this lookup is shown below. It begins by constructing a virtual `Database::Entry` for the given commit's tree pointer, then walks down each pathname segment, looking up each entry on the way.

```

# lib/database.rb

def load_tree_entry(oid, pathname)
  commit = load(oid)
  root   = Database::Entry.new(commit.tree, Tree::TREE_MODE)

  return root unless pathname

  pathname.each_filename.reduce(root) do |entry, name|
    entry ? load(entry.oid).entries[name] : nil
  end
end

```

If any uncommitted changes are found in the requested files, the `exit_on_errors` method will print those files that are changed and exit the program.

```

# lib/command/rm.rb

def exit_on_errors
  return if @uncommitted.empty? and @unstaged.empty?

  print_errors(@uncommitted, "changes staged in the index")

```

```
print_errors(@unstaged, "local modifications")

repo.index.release_lock
exit 1
end

def print_errors(paths, message)
  return if paths.empty?

  files_have = (paths.size == 1) ? "file has" : "files have"

  @stderr.puts "error: the following #{ files_have } #{ message }:"
  paths.each { |path| @stderr.puts "    #{ path }" }
end
```

The `rm` command is now a lot safer, and will refuse to remove anything that would result in uncommitted data being lost.

21.1.2. Refinements to the `rm` command

Sometimes, we don't want to remove a file from the workspace, just the index. Or, we want to remove things even if they have uncommitted changes. For these use cases, Git provides the `--cached` and `--force` options.

```
# lib/command/rm.rb

def define_options
  @parser.on("--cached")      { @options[:cached] = true }
  @parser.on("-f", "--force") { @options[:force] = true }
end
```

Using `--cached` means that `rm` will remove the named entry only from the index, not from the workspace. That is, it modifies the `remove_file` method like so:

```
# lib/command/rm.rb

def remove_file(path)
  repo.index.remove(path)
  repo.workspace.remove(path) unless @options[:cached]
  puts "rm '#{ path }'"
end
```

The safety checks in this case are slightly modified. `rm --cached` refuses to delete an index entry if it differs from both the HEAD commit and the workspace, but if it differs from only one of them it's safe to delete. If it matches HEAD, then its content is part of a commit, and if it matches the workspace, then its content is on disk and it can be restored to the index using the `add` command. Therefore we'll introduce a new set, `@both_changed`, to report errors specifically for this case.

```
# lib/command/rm.rb

def run
  repo.index.load_for_update

  @head_oid     = repo.refs.read_head
  @inspector   = Repository::Inspector.new(repo)
```

```
@uncommitted = []
@unstaged = []
@both_changed = []

#
# ...
end
```

The `plan_removal` method now needs to take the `--cached` option into account, and it can also accommodate `--force`. This option just means that the safety checks should be skipped; if we run `rm --force` we're asserting that we know it's dangerous, but we want to do it anyway. The updated `plan_removal` just bails out if `--force` is set, and otherwise checks for changes between HEAD, the index, and the workspace. If both staged and unstaged changes are present, then the file goes in the `@both_changed` list. Otherwise, if `--cached` is not set, the checks are applied as before.

```
# lib/command/rm.rb

def plan_removal(path)
  return if @options[:force]

  stat = repo.workspace.stat_file(path)
  raise "git rm: '#{ path }': Operation not permitted" if stat.directory?

  item = repo.database.load_tree_entry(@head_oid, path)
  entry = repo.index.entry_for_path(path)

  staged_change = @inspector.compare_tree_to_index(item, entry)
  unstaged_change = @inspector.compare_index_to_workspace(entry, stat) if stat

  if staged_change and unstaged_change
    @both_changed.push(path)
  elsif staged_change
    @uncommitted.push(path) unless @options[:cached]
  elsif unstaged_change
    @unstaged.push(path) unless @options[:cached]
  end
end
```

To complete this functionality, we just need a new clause in `exit_on_errors` to print the right copy for files with both staged and unstaged changes.

```
# lib/command/rm.rb

BOTH_CHANGED      = "staged content different from both the file and the HEAD"
INDEX_CHANGED     = "changes staged in the index"
WORKSPACE_CHANGED = "local modifications"

def exit_on_errors
  return if [@both_changed, @uncommitted, @unstaged].all?(&:empty?)

  print_errors(@both_changed, BOTH_CHANGED)
  print_errors(@uncommitted, INDEX_CHANGED)
  print_errors(@unstaged, WORKSPACE_CHANGED)

  repo.index.release_lock
  exit 1
end
```

The final refinement to `rm` that we'll implement is the `-r` flag. Sometimes it's convenient to remove a whole directory, rather than removing each file one-by-one. Running `rm -r` with a directory name accomplishes this.

```
# lib/command/rm.rb

def define_options
  @parser.on("--cached") { @options[:cached] = true }
  @parser.on("-f", "--force") { @options[:force] = true }
  @parser.on("-r") { @options[:recursive] = true }
end
```

The `-r` option doesn't affect the safety checks or the file removal process. Instead, it means that every argument passed to `rm` is expanded into a list of files — directory names are replaced with a list of their children, and file names are left as-is. We'll add a line to the `run` method to accomplish this:

```
# lib/command/rm.rb

@args = @args.flat_map { |path| expand_path(path) }
.map { |path| Pathname.new(path) }

@args.each { |path| plan_removal(path) }
exit_on_errors
```

Command `:Rm#expand_path` takes one of the arguments and expands it into a list of files. If the name is that of a tracked directory, then we return its list of child paths from the index if the `-r` flag is set. Otherwise we throw an error — since recursive removal can be really destructive, the user has to explicitly say that's what they want. If the name is that of a tracked file, it's returned as a single-item list, and otherwise we throw an error as the name does not appear in the index.

```
# lib/command/rm.rb

def expand_path(path)
  if repo.index.tracked_directory?(path)
    return repo.index.child_paths(path) if @options[:recursive]
    raise "not removing '#{path}' recursively without -r"
  end

  return [path] if repo.index.tracked_file?(path)
  raise "pathspec '#{path}' did not match any files"
end
```

This requires two new methods on the `Index` class: `Index#child_paths` returns an array of all the file names contained within a directory, and `Index#tracked_directory?` returns `true` if the given path is indeed a directory in the index. These are simply implemented on top of the `@parents` structure that caches directory names and their descendants.

```
# lib/index.rb

def child_paths(path)
  @parents[path.to_s].to_a
end

def tracked_directory?(path)
  @parents.has_key?(path.to_s)
end
```

While we're in the process of removing directories, we can amend the `Workspace#remove` method to delete any directories that become empty as a result of deleting the files inside them. The `Workspace#remove_directory` method³, attempts to remove a directory and silently fails if the directory is not empty.

```
# lib/workspace.rb

def remove(path)
  File.unlink(@pathname.join(path))
  path.dirname.ascend { |dirname| remove_directory(dirname) }
rescue Errno::ENOENT
end
```

The `rm` command is now complete and we can use it to remove our unused file:

```
$ jit rm lib/entry.rb

$ jit status
On branch master
Changes to be committed:

      deleted:    lib/entry.rb
```

21.2. Resetting the index state

While composing your next commit, you might add something to the index that you didn't actually want to commit at this point. For example, say I've just added a new method to the `Database` class and I've added this version of the file to the index.

```
$ jit add lib/database.rb
$ jit status
On branch master
Changes to be committed:

      modified:    lib/database.rb
```

Now, I decide I didn't really want that, and I want to get this change out of the index — not remove the file entirely, just make it so that the indexed version matches that from `HEAD`, not my workspace file. Git provides the `reset` command for doing just that. We'd like to be able to run `reset` with the name of a file, and reset the index state for that file back to what it was in `HEAD`, so our changes are effectively ‘unstaged’.

```
$ jit reset lib/database.rb
$ jit status
On branch master
Changes not staged for commit:

      modified:    lib/database.rb

no changes added to commit
```

The `reset` command also embodies other actions that affect the index, `HEAD`, and the workspace, but for now we'll focus on its behaviour when given a list of pathnames, which is to change the index entries for those paths to match the current `HEAD`.

³Section 14.3, “Updating the workspace”

Here's a first implementation of `Command::Reset#run`. It reads the `HEAD` ID, opens the index for updates, passes each argument to a method called `reset_path`, and saves the index.

```
# lib/command/reset.rb

module Command
  class Reset < Base

    def run
      @head_oid = repo.refs.read_head

      repo.index.load_for_update
      @args.each { |path| reset_path(Pathname.new(path)) }
      repo.index.write_updates

      exit 0
    end

    #
    end
  end
end
```

The `reset_path` method works as follows. It reads the list of tree entries for the given path in the `HEAD` commit from the database, and it removes the path from the index. It then iterates over each of the tree entries returned and inserts them into the index. This method relies on a few new methods elsewhere in the system, which we'll define below.

```
# lib/command/reset.rb

def reset_path(pathname)
  listing = repo.database.load_tree_list(@head_oid, pathname)
  repo.index.remove(pathname) if pathname

  listing.each do |path, entry|
    repo.index.add_from_db(path, entry)
  end
end
```

`Database#load_tree_list` returns all the entries under a given path in a commit, in a flattened structure. For example, given the name of a file, it returns a hash containing just that path and its corresponding `Database::Entry`.

```
>> db = Database.new(Pathname.new(".git/objects"))

>> db.load_tree_list("2a69362...", Pathname.new("lib/database.rb"))
=> { "lib/database.rb" => Database::Entry(oid="ff0ef6d...", mode=0100644) }
```

Whereas, given the name of a directory, it returns all the files within that directory, recursively, as a flattened map. In comparison to the nested tree structures stored in the database, this flattened structure is easier for working with the index, which contains a flat list of file paths.

```
>> db.load_tree_list("2a69362...", Pathname.new("lib/database"))
=> {
  "lib/database/author.rb" => Database::Entry(oid="cb18dd1...", mode=0100644),
  "lib/database/blob.rb"   => Database::Entry(oid="38fa8cb...", mode=0100644),
  "lib/database/commit.rb" => Database::Entry(oid="527e1a5...", mode=0100644),
```

```
"lib/database/entry.rb"    => Database::Entry(oid="28024a0...", mode=0100644),
"lib/database/tree.rb"     => Database::Entry(oid="a37f716...", mode=0100644),
"lib/database/tree_diff.rb" => Database::Entry(oid="0b8b7fa...", mode=0100644)
}
```

In Section 10.2, “HEAD/index differences”, we defined a method in `Command::Status` (later moved to `Repository::Status`) for doing just this, called `load_head_tree`. We’re going to replace that method with something similar and more general in the `Database` class, so we can now just call this new `Database` method from `Repository::Status`.

```
# lib/repository/status.rb

@head_tree = @repo.database.load_tree_list(@repo.refs.read_head)
```

The `Database#load_tree_list` method itself works like this. If it’s not given an object (for example if HEAD does not exist), then it returns an empty hash. Otherwise it loads the `Database::Entry` for the given path inside the commit, using the `load_tree_entry` method we defined earlier. Finally, it creates a new hash, calls `build_list` with this hash, the entry, and the pathname, and returns the hash. `build_list` takes these items and adds the entry to the hash if the entry represents a blob. If the entry represents a tree, then `build_list` calls itself recursively with each entry in the tree and its full path, so that eventually every file stored under this path is included.

```
# lib/database.rb

def load_tree_list(oid, pathname = nil)
  return {} unless oid

  entry = load_tree_entry(oid, pathname)
  list  = {}

  build_list(list, entry, pathname || Pathname.new(""))
  list
end

def build_list(list, entry, prefix)
  return unless entry
  return list[prefix.to_s] = entry unless entry.tree?

  load(entry.oid).entries.each do |name, item|
    build_list(list, item, prefix.join(name))
  end
end
```

That takes care of loading all the entries that should be put into the index. The index addition is done using this new method called `Index#add_from_db`, which uses the `Index::Entry#create_from_db` method⁴. This creates an index entry from a database entry, with no filestat information, since the index entry does not mirror a file in the workspace.

```
# lib/index.rb

def add_from_db(pathname, item)
  store_entry(Entry.create_from_db(pathname, item, 0))
  @changed = true
```

⁴Section 18.3.4, “Storing conflicts”

```
end
```

That's `reset` is its most basic form: load a subtree from the `HEAD` commit, and write it to the index, so that any changes staged for that subtree are removed.

21.2.1. Resetting to a different commit

As well as letting you reset an index entry to match the current `HEAD`, you can also reset a file to match any commit in the history. For example, `reset @~4 lib/database` will reset every file in the `lib/database` directory to match the state of the commit four steps back from `HEAD`.

We can begin to support this behaviour in `Command::Reset#run` by calling a method to select the commit ID at the beginning, rather than assuming we want the `HEAD` commit.

```
# lib/command/reset.rb

def run
  select_commit_oid

  repo.index.load_for_update
  @args.each { |path| reset_path(Pathname.new(path)) }
  repo.index.write_updates

  exit 0
end
```

Now, to how we determine which commit to use. The design of Git's command-line user interface here presents a tricky parsing problem. The first argument to `reset` can be either a path to a file, or a revision, and there's no way to tell it which type it is. Arguments after the first are definitely paths, and you *can* put arguments after `--` to indicate they're definitely paths, but if you have a file and a branch or object with the same name, the first argument is ambiguous.

For example, this command definitely means 'reset the file `readme` to match the `HEAD` commit':

```
$ git reset -- readme
```

Likewise, if there's no file named `@~12`, this means 'reset the file `readme` to match the commit twelve steps behind `HEAD`':

```
$ git reset @~12 readme
```

However, the following command is ambiguous: it could mean 'reset the file `readme` to match the `HEAD` version', or it could mean 'reset the entire index to match the branch named `readme`'.

```
$ git reset readme
```

This is a bad state of affairs: we cannot tell from the syntax of the command itself whether the word `readme` denotes the name of a file, or the name of a branch or other kind of commit identifier, and that means Git has to guess. It looks at the files in your index and the names of your branches and picks one. It would be much better if the user never had to remember this and the argument types were explicitly separated. For example, a revision argument could be called out with a switch and then all positional arguments could be assumed to be paths:

```
$ git reset readme          # set "readme" entry in index to == HEAD
$ git reset --commit readme # set entire index to match "readme" branch
```

Pretty much every Git command that accepts filenames suffers from this problem, probably for historical reasons. Plus, I believe Ruby's OptionParser has no way to distinguish arguments that come before and after the -- stopword. It prevents arguments after that word being interpreted as option names, but that's it. So I'd rather not get too bogged down in this problem and am going to resort to the simplest thing we can get away with. We'll try to resolve the first argument as a revision, and if that fails we'll assume it's a filename and leave it in the @args array. If there are no arguments given, the revision defaults to HEAD.

```
# lib/command/reset.rb

def select_commit_oid
  revision = @args.fetch(0, Revision::HEAD)
  @commit_oid = Revision.new(repo, revision).resolve
  @args.shift
rescue Revision::InvalidObject
  @commit_oid = repo.refs.read_head
end
```

With that change in place, we just need to modify reset_paths to use the selected commit ID rather than HEAD when it loads the file list for each path.

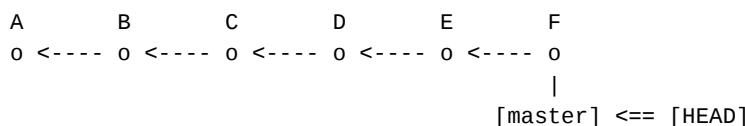
```
# lib/command/reset.rb

def reset_path(pathname)
  listing = repo.database.load_tree_list(@commit_oid, pathname)
  #
end
```

21.3. Discarding commits from your branch

What if you decide that the last few commits on your current branch aren't right, and you want to throw them away and start over? Let's say this is the current state:

Figure 21.1. Chain of commits with HEAD pointing to the branch tip



Say we decide we don't like commits *D*, *E* and *F*, and we'd like to go back to commit *C* and start committing from there. We need to move HEAD to point at *C* and start committing again, and we'd like that history to become the new master branch. We could do this by checking out commit *C*:

Figure 21.2. Checking out an earlier commit, leaving the branch pointer in place

```
$ jit checkout @~3

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

Then we could delete the existing master branch and recreate it:

Figure 21.3. Deleting the old branch pointer and restarting it at the current HEAD

```
$ jit branch -D master
$ jit branch master

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

Finally, we can check out the new `master` branch so `HEAD` starts to follow it.

Figure 21.4. Checking out the moved branch

```
$ jit checkout master

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

So technically, we can get the result we want using existing commands. However, it's fairly tedious, and it has some significant downsides. We've now lost the reference to `F`—what if we decide it was a mistake to throw away the old `master` branch? We've also lost the state of the index and workspace. It's quite common to want to keep the existing project files in your workspace, but rewind `HEAD` a few steps to redo the commits. This is the other main functionality that Git folds into the `reset` command.

When it's run without any file paths as arguments, the `reset` command runs in one of several modes. The most commonly used are:

- `soft`: This just moves `HEAD` to point at the specified commit. Note that whereas `checkout` changes the content of `.git/HEAD` itself, `reset` moves the branch that `HEAD` points at, just like the `commit` and `merge` commands.
- `mixed`: This does the same as `soft`, plus it updates the index to match the contents of the selected commit. This mode is the default.
- `hard`: This does what `mixed` does, plus it updates the workspace to match the index, effectively removing any unstaged changes.

None of these modes do any safety checks like `rm` does. They will overwrite the contents of the index and workspace without checking whether you'll lose anything; their entire purpose is to throw content away.

The `soft` and `mixed` modes are commonly used to pop the last few commits off the history while keeping the current workspace state, so you can redo the commits of the content you currently have. They're also fairly easy to build on top of our current implementation. We'll start by declaring the command-line options:

```
# lib/command/reset.rb

def define_options
  @options[:mode] = :mixed
```

```
    @parser.on("--soft") { @options[:mode] = :soft }
    @parser.on("--mixed") { @options[:mode] = :mixed }
end
```

In the `Command::Reset#run` method, we'll factor the logic for resetting paths out into its own method, `reset_files`. We'll also update `HEAD` to the selected commit if no file paths were given — this happens in all modes.

```
# lib/command/reset.rb

def run
  select_commit_oid

  repo.index.load_for_update
  reset_files
  repo.index.write_updates

  repo.refs.update_head(@commit_oid) if @args.empty?
  exit 0
end
```

`reset_files` takes care of updating the index. If we're in soft mode, it does nothing. If no file paths were given, it completely clears the index and rebuilds it from the given commit, by calling `reset_path` with `nil`. Otherwise it carries out the logic we had before, resetting the index state for each path argument.

```
# lib/command/reset.rb

def reset_files
  return if @options[:mode] == :soft

  if @args.empty?
    repo.index.clear!
    reset_path(nil)
  else
    @args.each { |path| reset_path(Pathname.new(path)) }
  end
end
```

This expanded logic just requires one extra method on `Index`. It currently has a private method called `clear` which it uses to reset its internal variables. The new method `clear!` will call that and then just set the `@changed` flag, so that the index is still written to disk if it's empty.

```
# lib/index.rb

def clear!
  clear
  @changed = true
end
```

21.3.1. Hard reset

It's tempting to implement a hard reset by running the code paths for a mixed reset, and then updating the workspace. However, this is a false economy, and actually makes it hard to implement this operation correctly. Whereas we've been updating the index by removing

the existing entries and rebuilding them from the target commit, such an approach won't work for updating the workspace. This is partly a performance concern; the index is an in-memory data structure and so making large changes to it is relatively cheap, compared to deleting and rewriting a lot of files in the workspace. Rebuilding the workspace is particularly expensive since it requires reading the blob for each file from the database, whereas the index just needs the object ID and mode from the tree items.

More importantly, the command won't behave in the right way if we implement it like this. Consider a repository in which the workspace includes two files, `f.txt` and `g.txt`. `f.txt` is in the index, while `g.txt` is not. Neither file exists in the current `HEAD` commit. So, the status of this repository will include:

```
A f.txt  
?? g.txt
```

Now, when we run `reset --hard`, any differences between `HEAD`, the index and workspace should be eliminated, with one caveat: files that are untracked before the reset should be left alone, unless those files exist in the target commit and therefore need to be overwritten. If we update the index first, then both files `f.txt` and `g.txt` will be untracked. The reset should delete `f.txt` from the workspace but leave `g.txt` alone, but we've now lost the information we need to make that distinction. Deleting everything from the workspace and rebuilding it would also result in `g.txt` being lost.

A better strategy would be to work out the differences in the index and workspace, relative to the target commit, before we make any changes to either. Any files discovered to have unstaged or uncommitted changes can then be updated in the workspace and the index. This results in the smallest amount of work necessary to reach the right state, and also means the index can be populated with updated `stat(2)` information for the corresponding workspace files.

Since it's going to differ in implementation from the existing `reset` code, I'm going to implement hard mode as a distinct method on the `Repository` class. This also means we can reuse this operation from other commands, as we'll see shortly. Let's declare the necessary option, and call `Repository#hard_reset` with the target commit if the option is set.

```
# lib/command/reset.rb

def define_options
  @options[:mode] = :mixed

  @parser.on("--soft") { @options[:mode] = :soft }
  @parser.on("--mixed") { @options[:mode] = :mixed }
  @parser.on("--hard") { @options[:mode] = :hard }
end

def reset_files
  return if @options[:mode] == :soft
  return repo.hard_reset(@commit_oid) if @options[:mode] == :hard

  #
end
```

The `Repository#hard_reset` method instantiates a class that does the actual work, calling its `execute` method.

```
# lib/repository.rb

def hard_reset(oid)
  HardReset.new(self, oid).execute
end
```

The Repository::HardReset class takes a Repository and a commit ID as input, and updates the index and workspace to match that commit ID. It does not move HEAD — this is still taken care of in the Command::Reset#run code. To make the necessary updates, it takes the status of the repository against the target commit, and gets the list of changed paths from there. For each of these paths, it calls reset_path, which we'll define below.

```
# lib/repository/hard_reset.rb

class Repository
  class HardReset

    def initialize(repo, oid)
      @repo = repo
      @oid = oid
    end

    def execute
      @status = @repo.status(@oid)
      changed = @status.changed.map { |path| Pathname.new(path) }

      changed.each { |path| reset_path(path) }
    end

    def reset_path(path)
      # ...
    end
  end
end
```

The Repository#status method until now has not taken any arguments; it returns the status relative to the current HEAD. But now we want to know how the repository differs from an arbitrary target commit, and we can make Repository::Status report that by passing in the desired commit.

```
# lib/repository.rb

def status(commit_oid = nil)
  Status.new(self, commit_oid)
end
```

Instead of using the HEAD commit, Repository::Status can now load the tree from the requested commit, if given. Its changes attribute will then contain all the index and workspace paths that differ from that commit.

```
# lib/repository/status.rb

def initialize(repository, commit_oid = nil)
  # ...

  commit_oid ||= @repo.refs.read_head
```

```
@head_tree = @repo.database.load_tree_list(commit_oid)

# ...
end
```

Back in `Repository::HardReset`, we need to fill out the `reset_path` method that updates the index and workspace for each path. It begins by removing the path from both places, and then fetches the tree entry from the listing loaded by the `Repository::Status` class. If there's no entry for that path, then we can return. Otherwise, we load the blob that the entry points at, write its contents to the workspace, take the stat of the updated file, and use that to replace the index entry.

```
# lib/repository/hard_reset.rb

def reset_path(path)
  @repo.index.remove(path)
  @repo.workspace.remove(path)

  entry = @status.head_tree[path.to_s]
  return unless entry

  blob = @repo.database.load(entry.oid)
  @repo.workspace.write_file(path, blob.data, entry.mode, true)

  stat = @repo.workspace.stat_file(path)
  @repo.index.add(path, entry.oid, stat)
end
```

When we call `Workspace#remove`, we'd like to remove whatever is there, if anything, rather than only removing files. Therefore we'll amend this method to call `FileUtils.rm_rf` rather than `File.unlink`⁵, a wrapped for `unlink(2)`⁶ that only removes files.

```
# lib/workspace.rb

def remove(path)
  FileUtils.rm_rf(@pathname.join(path))
  path.dirname.ascend { |dirname| remove_directory(dirname) }
rescue Errno::ENOENT
end
```

We also need to expand the job of `workspace#write_file`. Rather than only writing data to the given path, we also need to be able to set the new file's mode, and create its parent directory if necessary⁷.

```
# lib/workspace.rb

def write_file(path, data, mode = nil, mkdir = false)
  full_path = @pathname.join(path)
  FileUtils.mkdir_p(full_path.dirname) if mkdir

  flags = File::WRONLY | File::CREAT | File::TRUNC
  File.open(full_path, flags) { |f| f.write(data) }
```

⁵<https://docs.ruby-lang.org/en/2.3.0/File.html#method-c-unlink>

⁶<https://manpages.ubuntu.com/manpages/bionic/en/man2/unlink.2.html>

⁷Previously, creating necessary directories for files was done as part of a migration between trees; see Section 14.3, “Updating the workspace”.

```
File.chmod(mode, full_path) if mode  
end
```

This completes the logic for `Repository#hard_reset`, and allows us to get back to a known good state, regardless of what state we've got the workspace into.

21.3.2. I'm losing my HEAD

At the beginning of this section, we discussed how `reset` moves `HEAD`. This can result in a state where a chain of commits exists without any refs pointing to them.

Figure 21.5. Chain of unreachable commits

```
$ jit reset @~3  
  
A      B      C      D      E      F  
o <---- o <---- o <---- o <---- o <---- o  
|  
[master] <== [HEAD]
```

That might not be a problem — you may have decided to `reset` because those commits aren't what you wanted and you never want to see them again. However, to be on the safe side, `reset` keeps a reference to the previous `HEAD` commit around, in case you want to go and look at those 'lost' commits or retrieve content from them. It puts this reference in the file `.git/ORIG_HEAD` — when Git does anything that might result in some commits becoming unreachable, it stores the current `HEAD` commit ID there. It stores the literal commit ID, not a symref to whatever `HEAD` was pointing at.

Figure 21.6. ORIG_HEAD pointing to the previous HEAD position

```
$ jit reset @~3  
  
A      B      C      D      E      F  
o <---- o <---- o <---- o <---- o <---- o  
|           |  
[master] <== [HEAD]     [ORIG_HEAD]
```

Saving this file can be taken care of when we update `HEAD` in `Command::Reset#run`. We'll have `Refs#update_head` return the previous ID that `HEAD` was pointing at, and then write that value to the `ORIG_HEAD` ref file.

```
# lib/command/reset.rb  
  
def run  
  select_commit_oid  
  
  repo.index.load_for_update  
  reset_files  
  repo.index.write_updates  
  
  if @args.empty?  
    head_oid = repo.refs.update_head(@commit_oid)  
    repo.refs.update_ref(Refs::ORIG_HEAD, head_oid)  
  end  
  
  exit 0
```

```
end
```

Refs#update_head uses a method we introduced in Section 15.4, “Updating HEAD on commit”, called update_symref. This atomically updates the ref at the end of a chain of symbolic references, allowing commands like commit and merge to move the current branch pointer. To make sure that the value we write to ORIG_HEAD is really the value that was overwritten by update_symref, we should return the contents of the file that we read while it was locked as part of the update process.

In Refs#update_symref, we already read the ref file to determine whether it’s a symref or a literal object ID. If it’s not a symref, then we write the new value to it and return. We’ll now modify the method to return the object ID that was read before overwriting the file. The recursive call to update_symref at the end means this return value will bubble up the chain back to the original caller.

```
# lib/refs.rb

def update_symref(path, oid)
  lockfile = Lockfile.new(path)
  lockfile.hold_for_update

  ref = read_oid_or_symref(path)

  unless ref.is_a?(SymRef)
    write_lockfile(lockfile, oid)
    return ref&.oid
  end

  begin
    update_symref(@pathname.join(ref.path), oid)
  ensure
    lockfile.rollback
  end
end
```

After that, we just need a new method for updating arbitrary ref files, rather than HEAD. In this case we always want to write a literal object ID, not a symref, so we’ll use update_ref_file to implement this.

```
# lib/refs.rb

ORIG_HEAD = "ORIG_HEAD"

def update_ref(name, oid)
  update_ref_file(@pathname.join(name), oid)
end
```

The reset command is now safer, and we can get back to our pre-reset position by running reset ORIG_HEAD or checkout ORIG_HEAD.

21.4. Escaping from merges

One very common situation in which we want to throw our changes away and get back to a clean state is during a merge conflict. If the conflicts are particularly hard to fix, we might

immediately change our minds and want to cancel the merge, going back to the state of the current HEAD. Or, we might begin working on the conflicts but get bogged down, and change our mind about how our existing code should be structured before merging other changes into it, to make the merge easier.

The `merge --abort` command is for doing just this. It discards all the pending commit state and gets you back to the state you were in before the merge started — effectively performing a hard reset to the prior HEAD position.

Let's declare this option in the `merge` command, and handle it in `Command::Merge#run`. Like `reset`, `merge` also sets `ORIG_HEAD` to the value of `HEAD` before the merge starts. However, the `--abort` option does not use this to determine what state to reset to.

```
# lib/command/merge.rb

def define_options
  @options[:mode] = :run

  @parser.on("--abort") { @options[:mode] = :abort }
  @parser.on("--continue") { @options[:mode] = :continue }
end

def run
  handle_abort if @options[:mode] == :abort
  handle_continue if @options[:mode] == :continue
  handle_in_progress_merge if pending_commit.in_progress?

  @inputs = ::Merge::Inputs.new(repo, Revision::HEAD, @args[0])
  repo.refs.update_ref(Refs::ORIG_HEAD, @inputs.left_oid)

  #
end
```

The `handle_abort` method does all the required clean-up. It clears the pending commit state, and performs a hard reset to the current HEAD.

```
# lib/command/merge.rb

def handle_abort
  repo.pending_commit.clear

  repo.index.load_for_update
  repo.hard_reset(repo.refs.read_head)
  repo.index.write_updates

  exit 0
rescue Repository::PendingCommit::Error => error
  @stderr.puts "fatal: #{error.message}"
  exit 128
end
```

If there isn't currently a pending commit in progress, then `Repository::PendingCommit` should raise an error. This error will be triggered by Ruby raising an `ENOENT` error when we try to delete the state files.

```
# lib/repository/pending_commit.rb
```

```
def clear(type = :merge)
  File.unlink(@head_path)
  File.unlink(@message_path)
rescue Errno::ENOENT
  name = @head_path.basename
  raise Error, "There is no merge to abort (#{ name } missing)."
end
```

And that's it! We now have a way to bail out of a merge if we've got the repository into a bad state.

22. Editing messages

Using the `reset` command we developed in the previous chapter, it's now possible to drop a few commits from your current branch and redo them, or squash them into a single commit. For example, running `reset --soft` will move your current branch pointer but keep the existing index state, so you can use it to collapse the last few commits into a single one containing the most recent tree:

```
$ jit reset --soft @~3  
$ echo "<message>" | jit commit
```

However, this has a significant downside: although you get to keep all the tree content from the commit before you reset, the `reset` throws away the commit messages. When you run `jit commit`, you'll need to pipe in a whole new commit message. Now, maybe you have the last commit message in a file, and you can pipe that in again:

```
$ cat path/to/message.txt | jit commit
```

As we add more features for working with the history, it will become desirable to load up messages from old commits and edit them, before using them to annotate a new commit. Although reading from standard input was a good start to get us moving quickly, it's not great for handling these more complex use cases. We can improve matters by launching the user's text editor whenever they need to compose a message — let's begin by adding this capability to the `commit` command.

To remind ourselves what the `commit` command code looks like, we last saw it in Section 17.3, “Commits with multiple parents”. Below I've made a slight improvement to its output, displaying the current branch name (or detached `HEAD` if there's no current branch), and a shortened commit ID.

```
# lib/command/commit.rb  
  
def run  
  repo.index.load  
  resume_merge if pending_commit.in_progress?  
  
  parent = repo.refs.read_head  
  message = @stdin.read  
  commit = write_commit([*parent], message)  
  
  print_commit(commit)  
  
  exit 0  
end
```

The actual output logic has been moved into `Command::WriteCommit` — other commands will end up using this `print_commit` logic to display any new commits they generate.

```
# lib/shared/write_commit.rb  
  
def print_commit(commit)  
  ref = repo.refs.current_ref  
  info = ref.head? ? "detached HEAD" : ref.short_name  
  oid = repo.database.short_oid(commit.oid)
```

```
info.concat(" (root-commit)") unless commit.parent
info.concat("#{ oid }")

puts "[#{ info }] #{ commit.title_line }"
end
```

Now let's see how this code can be augmented to allow the user to set the message value that goes into the commit.

22.1. Setting the commit message

Since introducing support for command-line options¹, we can now offer an easier way to set the message when making a commit. Git supports the `--message` option, which takes the message from the option's value, and `--file`, which specifies a file from which to read the message. Both the `commit` and `merge` commands support these, so we'll add support for them to the `Command::WriteCommit` mixin, along with a `read_message` method that examines the options the user has set and returns the message string from the appropriate source.

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

def define_write_commit_options
  @parser.on "-m <message>", "--message=<message>" do |message|
    @options[:message] = message
  end

  @parser.on "-F <file>", "--file=<file>" do |file|
    @options[:file] = expanded_pathname(file)
  end
end

def read_message
  if @options.has_key?(:message)
    "#{ @options[:message] }\n"
  elsif @options.has_key?(:file)
    File.read(@options[:file])
  end
end
```

We can now load these option definitions into the `Command::Commit` class, and call `read_message` instead of `@stdin.read` to get the message value.

```
# lib/command/commit.rb

def define_options
  define_write_commit_options
end

def run
  #

  parent  = repo.refs.read_head
  message = read_message
  commit  = write_commit([*parent], message)
```

¹Section 15.5.1, “Parsing command-line options”

```
    print_commit(commit)

    exit 0
end
```

A similar change is applied in `Command::Merge#run`. These options make the `commit` and `merge` commands a bit easier to use, and we've not lost the ability to write multi-line messages — we can compose them in a file and use the `--file` option. However, it would be nicer if message composition was better integrated into the commit workflow, particularly in the case of conflicted merges. While resolving any merge conflicts, we may decide we want to change the merge message, adding more information about how we fixed things up. In this case it would be nice to present the original merge message for editing, before making the final merge commit.

22.2. Composing the commit message

Let's begin with the most common use case: when we run `commit` with no `--message` or `--file` option, or if the `--edit` option is set, Jit should launch our preferred text editor and ask us to compose a message there. We'd like to have a method that prompts the user for input via their editor, and returns the text they entered. We'd also like to be able to set the initial text they see in their editor, and add some explanatory comments that should be stripped out of the message before it's used.

So, if we run:

```
$ jit commit --message "hello world" --edit
```

Then, we should see our editor pop up with the following content:

```
hello world

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
```

When we're done editing and we close the editor, any lines that don't begin with `#` should be used as the commit message.

We will shortly introduce a class called `Editor` to implement most of this functionality, but rather than discuss that class first, I'm going to begin with the command code that calls the editor. This will give some idea of what we're aiming for and will inform the `Editor` class's design. Designing programs from the outside in like this is a common way of discovering what we need.

Here's a version of the `commit` command, where `read_message` has been passed through `compose_message` to let the user edit the message. `compose_message` in turn calls a new method, `edit_file`, which takes the path of the file we want to edit; in order to open some text in an editor, we need to write that text to a file and then ask the editor to open said file. `edit_file` also yields an `editor` object to us, which lets us write some initial content to the file, including some things that should be rendered as comments. Finally, the `editor.close` method can be used to write this content to the given file, but prevent the editor from actually launching.

```
# lib/command/commit.rb
```

```
COMMIT_NOTES = <<~MSG
Please enter the commit message for your changes. Lines starting
with '#' will be ignored, and an empty message aborts the commit.
MSG

def run
# ...

parent = repo.refs.read_head
message = compose_message(read_message)
commit = write_commit([*parent], message)

# ...
end

def compose_message(message)
edit_file(commit_message_path) do |editor|
  editor.puts(message || "")
  editor.puts("")
  editor.note(COMMIT_NOTES)

  editor.close unless @options[:edit]
end
end
```

commit_message_path will be needed in a few different places and so we define it in writeCommit: it returns the path .git/COMMIT_EDITMSG, which is where commit messages are usually composed.

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

def commit_message_path
  repo.git_path.join("COMMIT_EDITMSG")
end
```

This is an attempt at an interface that hides all the messy details of how we interact with the editor, while preserving the minimum amount of information: which file we want the text to live in, what the initial content should be, and what the final content of the file was — this should be the return value.

edit_file will be a method many commands want to use, so let's put it in Command::Base. It is essentially a thin wrapper around the Editor class, which we'll introduce shortly, with a bit of behaviour determined by the current environment. We don't want to force everyone to use the same editor, so we need to get the name of the user's editor program from an environment variable; VISUAL and EDITOR and commonly used environment variables for this, while GIT_EDITOR is a Git-specific one. Also, we don't actually want the editor to launch unless standard output is a TTY — the text editor is an interactive UI, so if the command is being run as part of another program rather than from a terminal, we shouldn't use it.

```
# lib/command/base.rb

def edit_file(path)
  Editor.edit(path, editor_command) do |editor|
    yield editor
  editor.close unless @isatty
```

```
    end
  end

  def editor_command
    @env["GIT_EDITOR"] || @env["VISUAL"] || @env["EDITOR"]
  end
```

22.2.1. Launching the editor

Now we can start to flesh out the `Editor` class itself, whose core responsibility is to let us write some initial content to a file, open the user's text editor to modify its contents, and finally return to us the completed content of the file. To make the `edit_file` method work, we need `Editor.edit` to take a pathname and a command, yield an `Editor` object for the caller to write the initial content, let the user edit the file, and then return its contents.

```
# lib/editor.rb

class Editor
  DEFAULT_EDITOR = "vi"

  def self.edit(path, command)
    editor = Editor.new(path, command)
    yield editor
    editor.edit_file
  end

  def initialize(path, command)
    @path     = path
    @command  = command || DEFAULT_EDITOR
    @closed   = false
  end

  #
end
```

Then we can flesh out the interface of the `Editor` class. As we saw in the `compose_message` method above, this object needs to respond to `puts`, `note` and `close`. `puts` writes content to the file, `note` writes content prefixing each line with `#`, and `close` just flags the editor as closed.

```
# lib/editor.rb

def puts(string)
  return if @closed
  file.puts(string)
end

def note(string)
  return if @closed
  string.each_line { |line| file.puts("# #{line}") }
end

def close
  @closed = true
end

def file
```

```
    flags = File::WRONLY | File::CREAT | File::TRUNC
    @file ||= File.open(@path, flags)
end
```

Finally, the `Editor` needs to implement `edit_file`, the final call made by `Editor.edit`. This should launch the text editor, unless the `@closed` flag has been set, and when the editor exits, it should return the contents of the file with any comment lines stripped out. If no non-comment content is present, that is all the non-comment lines match the whitespace regular expression `^\s*$`, then `remove_notes` returns `nil` indicating there was no message given.

```
# lib/editor.rb

def edit_file
  file.close
  editor_argv = Shellwords.shellsplit(@command) + [@path.to_s]

  unless @closed or system(*editor_argv)
    raise "There was a problem with the editor '#{@command}'."
  end

  remove_notes(File.read(@path))
end

def remove_notes(string)
  lines = string.lines.reject { |line| line.start_with?("#") }

  if lines.all? { |line| /^\s+$/ =~ line }
    nil
  else
    "#{lines.join("\n").strip}\n"
  end
end
```

The `Shellwords.shellsplit`² method parses a command string in the same way a shell would do, for example:

```
>> Shellwords.shellsplit("commit -m 'my message'")
=> ["commit", "-m", "my message"]
```

This lets us handle `$EDITOR` settings that might contain multiple words. For example, the Atom³ editor must be run using the command `atom --wait <file>`, otherwise it returns as soon as the file is open rather than blocking until it's closed. If we use the single string "`atom --wait`" with `system()`, the operating system will think we're trying to run a program called `atom --wait`, rather than the `atom` program with `--wait` as an argument.

The call to `system`⁴ is what runs the text editor program. This is a wrapper for the C `system()`⁵ function, which runs another program via the shell⁶. Ruby's `system` method has a safety feature: if passed multiple strings, it will run the specified program with the other strings as arguments, rather than executing the entire command as a single string via the shell. For

²<https://docs.ruby-lang.org/en/2.3.0/Shellwords.html#method-c-shellsplit>

³<https://atom.io/>

⁴<https://docs.ruby-lang.org/en/2.3.0/Kernel.html#method-i-system>

⁵<https://manpages.ubuntu.com/manpages/bionic/en/man3/system.3posix.html>

⁶Despite its name, this should not be confused with making a *system call*, which means calling a function in the kernel that mediates access to the hardware.

example, `system("rm -rf #{ path }")` will result in this shell command being run if `path` is "foo *":

```
rm -rf foo *
```

Whereas, `system("rm", "-rf", path)` will result in this command being run as a direct child process, not via the shell:

```
rm -rf "foo "*
```

That is, only the file with the literal name `foo *` will be deleted, rather than everything in the current working directory. Using `system` this way prevents you making mistakes due to how the shell breaks the command up into separate words.

`system` is the second method we've seen for launching a child process in Ruby. In Section 12.3.3, "Invoking the pager", we used `Process.spawn`, which runs the new process in a non-blocking manner: the new process runs and our Ruby program keeps going without waiting for the child process to exit. Here, we want to wait for the editor to exit before proceeding. We could use `Process.waitpid` for that, but `system` does all this for us: it blocks until the child process exits. It also makes the child process use the same standard output/error streams as the Ruby process, so if you run a terminal-based editor like Vim⁷ or Emacs⁸, the editor will appear in your terminal and Ruby will wait until you quit it.

To complete the story, we just need to define an `:edit` option to control whether the given message should be edited or not. This defaults to `:auto` and is set by the `--message` and `--file` options, unless it's been set explicitly via the `--[no-]edit` option.

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

def define_write_commit_options
  @options[:edit] = :auto
  @parser.on("-e", "--[no-]edit") { |value| @options[:edit] = value }

  @parser.on "-m <message>", "--message=<message>" do |message|
    @options[:message] = message
    @options[:edit] = false if @options[:edit] == :auto
  end

  @parser.on "-F <file>", "--file=<file>" do |file|
    @options[:file] = expanded_pathname(file)
    @options[:edit] = false if @options[:edit] == :auto
  end
end
```

22.2.2. Starting and resuming merges

In Section 19.4.2, "Retaining state across commands", we used the `PendingCommit` class to store the inputs to the `merge` command while it was suspended for the user to resolve conflicts. Its `start` method took the object ID of the merge target and the message passed on the command line. We can now use the editor to populate the message, so rather than have `PendingCommit#start` take a message as input, we'll expose the pathname where

⁷<https://www.vim.org/>

⁸<https://www.gnu.org/software/emacs/>

PendingCommit stores the message, allowing the merge command to edit that file. The start method now only writes the merged commit's ID to disk.

```
# lib/repository/pending_commit.rb

attr_reader :message_path

def start(oid)
  flags = File::WRONLY | File::CREAT | File::EXCL
  File.open(@head_path, flags) { |f| f.puts(oid) }
end
```

As we saw when introducing the PendingCommit class, the Command::Merge#run method performs the following steps when commencing a normal merge: it stores the inputs in pending_commit, it calls resolve_merge to apply the changes from the merged branch to the index and workspace, and finally commit_merge to write a merge commit of the resulting index state.

```
# lib/command/merge.rb

def run
  #
  # ...

  pending_commit.start(@inputs.right_oid)
  resolve_merge
  commit_merge

  exit 0
end
```

We can update resolve_merge so that if applying the merged changes results in a conflict, we invoke the fail_on_conflict method. This uses Editor to write some content to the PendingCommit object's message file, namely, the message given using --message or the default message of Merge commit '<rev>'. It then adds the list of conflicted files as comments, so that when we come to edit this file after fixing the conflicts, we're reminded of where the problems were and we can write up how we resolved them.

```
# lib/command/merge.rb

def resolve_merge
  repo.index.load_for_update

  #
  # ...

  repo.index.write_updates
  fail_on_conflict if repo.index.conflict?
end

def fail_on_conflict
  edit_file(pending_commit.message_path) do |editor|
    editor.puts(read_message || default_commit_message)
    editor.puts("")
    editor.note("Conflicts:")
    repo.index.conflict_paths.each { |name| editor.note("\t#{ name }") }
    editor.close
  end
end
```

```

    puts "Automatic merge failed; fix conflicts and then commit the result."
    exit 1
end

def default_commit_message
  "Merge commit '#{ @inputs.right_name }'"
end

```

The conflict_paths method is a new addition to the Index class that returns a set of those paths whose entries have a non-zero stage, indicating they're in conflict.

```

# lib/index.rb

def conflict_paths
  paths = Set.new
  each_entry { |entry| paths.add(entry.path) unless entry.stage == 0 }
  paths
end

```

If the merge was successful, then commit_merge is invoked. Whereas previously this method called pending_commit.merge_message to fetch the original message, it now invokes compose_message:

```

# lib/command/merge.rb

def commit_merge
  parents = [@inputs.left_oid, @inputs.right_oid]
  message = compose_message

  write_commit(parents, message)

  pending_commit.clear
end

```

compose_message works much like the method of the same name in the Command::Commit class, with minor variations: it uses .git/MERGE_MSG rather than .git/COMMIT_EDITMSG to store its contents, and has some different instructional comments.

```

# lib/command/merge.rb

COMMIT_NOTES = <<~MSG
Please enter a commit message to explain why this merge is necessary,
especially if it merges an updated upstream into a topic branch.

Lines starting with '#' will be ignored, and an empty message aborts
the commit.

MSG

def compose_message
  edit_file(pending_commit.message_path) do |editor|
    editor.puts(read_message || default_commit_message)
    editor.puts("")
    editor.note(COMMIT_NOTES)

    editor.close unless @options[:edit]
  end
end

```

Finally, we can deal with what happens when a merge is resumed; we need to retrieve the original merge message and let the user edit it before writing it into a commit. In this case, `WriteCommit#resume_merge` will need to call `compose_merge_message` rather than fetching the original message from `pending_commit.merge_message` and using it unmodified.

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

MERGE_NOTES = <<~MSG
It looks like you may be committing a merge.
If this is not correct, please remove the file
\t.git/MERGE_HEAD
and try again.
MSG

def resume_merge
  handle_conflicted_index

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

  pending_commit.clear
  exit 0
end
```

The `compose_merge_message` method does much the same as we've already seen in terms of using the `Editor`, and again the tweaks relate to the copy that's first written to the file. It retrieves the original message from `.git/MERGE_MSG` and moves it to `.git/COMMIT_EDITMSG` for editing. It prints an optional set of notes, seen in the `MERGE_NOTES` constant above, and finally appends the `COMMIT_NOTES` from `Command::Commit` as a comment.

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

def compose_merge_message(notes = nil)
  edit_file(commit_message_path) do |editor|
    editor.puts(pending_commit.merge_message)
    editor.note(notes) if notes
    editor.puts("")
    editor.note(Commit::COMMIT_NOTES)
  end
end
```

We've seen a few different use cases here: authoring a commit, writing a message on successful merge, storing a message on a failed merge, and editing the message when the merge is completed. All fairly straightforward additions, with slight variations in how the data is managed and what copy is shown to the user. This infrastructure will set us up well for any further use of the editor we want to make later on.

22.3. Reusing messages

In Section 21.3, “Discarding commits from your branch”, we saw that a common motivation for using the `reset` command is to move your current branch pointer back one or more commits while keeping the workspace and index state unchanged. This effectively drops the latest

commit(s) from your branch and lets you write new ones without losing the files currently in the workspace.

For example, if there is a file that should have been part of the last commit, but you forgot to add it, you might run:

```
$ jit reset --soft @^  
$ jit add <file>  
$ jit commit --message "<message>"
```

This workflow is common enough that it quickly becomes frustrating to need to retype the message from the commit you just dropped. It would be handy if we could recall the message from the dropped commit and use it as the message for its replacement. And indeed, Git has a couple of options for doing just that:

- `-C, --reuse-message`: this option takes a revision and uses the message from the named commit, rather than you needing to use the `--message` option
- `-c, --reedit-message`: this does the same thing as `--reuse-message`, but it also opens the message in the editor in case you want to change it

We can declare these options in the `Command::Commit` class; they store the given revision in the `:reuse` option, and set the `:edit` flag appropriately.

```
# lib/command/commit.rb

def define_options
  define_write_commit_options

  @parser.on "-C <commit>", "--reuse-message=<commit>" do |commit|
    @options[:reuse] = commit
    @options[:edit] = false
  end

  @parser.on "-c <commit>", "--reedit-message=<commit>" do |commit|
    @options[:reuse] = commit
    @options[:edit] = true
  end
end
```

The `run` method now needs to accommodate these options. We do that by having it call `reused_message` if `read_message` returns nothing; that is, if neither `--message` nor `--file` was used. `reused_message` uses the `:reuse` option with the `Revision` class to look up a commit and return its message.

```
# lib/command/commit.rb

def run
  # ...

  parent = repo.refs.read_head
  message = compose_message(read_message || reused_message)
  commit = write_commit([]*parent, message)

  # ...
end
```

```

def reused_message
  return nil unless @options.has_key?(:reuse)

  revision = Revision.new(repo, @options[:reuse])
  commit   = repo.database.load(revision.resolve)

  commit.message
end

```

We can now amend the tree of the latest commit while retaining its message, like so:

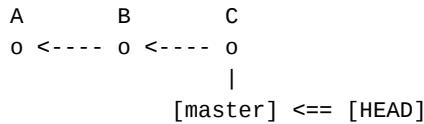
```

$ jit reset --soft @^
$ jit add <file>
$ jit commit --reuse-message ORIG_HEAD

```

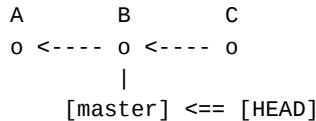
You will see much literature on Git talking about ‘changing’ or ‘amending’ commits or the history. It’s important to remember that we are not really changing any objects, since Git objects are immutable — they cannot be changed without changing their ID. What’s really happening is that we’re adding new commits and changing what the refs point to. Say we began the above operation with this chain of commits:

Figure 22.1. Commit chain before a reset



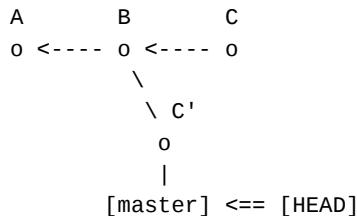
The `reset` command moves the `master` branch pointer so that it refers to *B*. Commit *C* is not deleted, it’s just no longer reachable from `master`. The `ORIG_HEAD` reference can still be used to access it.

Figure 22.2. Commit chain after a reset



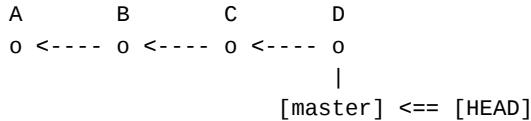
Running `commit` then creates a new commit, whose tree is that of *C* with one file added, and whose message is the same as that of *C*. Git documentation will sometimes refer to this commit as *C'*, to indicate that’s a modified version of *C*, but really there is no structural connection between these commits in the graph; *C'* is a distinct commit in its own right.

Figure 22.3. Commit graph with an ‘amended’ commit



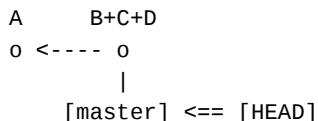
We can also use `reset` to ‘squash’ the last few commits into a single one, keeping one of their messages. For example, say we have the following commit sequence:

Figure 22.4. Chain of commits to be squashed



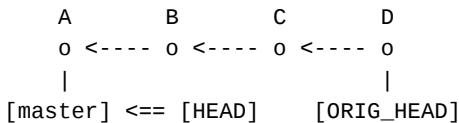
We'd like the changes from commits *C* and *D* to be folded into commit *B*; *C* and *D* effectively represent content we forgot to add to *B*. The history we'd like to have is thus these two commits, where the commit labelled *B+C+D* has the same tree as *D*:

Figure 22.5. Desired squashed history



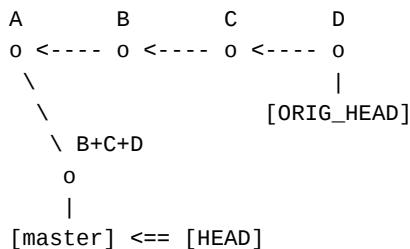
We can achieve this by running `reset --soft @~3`, to point `HEAD` at commit *A* but leave the index containing the tree from commit *D*.

Figure 22.6. HEAD reset to before the commits to be squashed



Now, we can use `commit --reuse-message ORIG_HEAD~2`, which will create a new commit whose parent is *A*, whose tree is that of *D*, and which has the message from *B*:

Figure 22.7. Squashed history



Using this technique, an arbitrary number of commits from the tip of a branch can be squashed together into a single commit.

22.3.1. Amending the HEAD

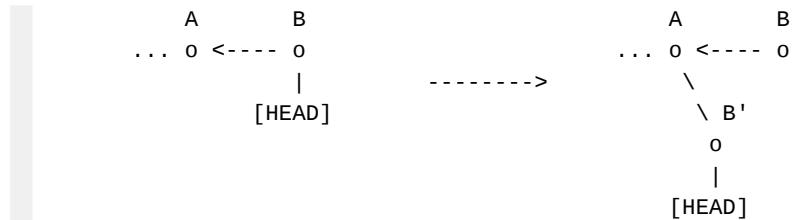
Although `reset` provides a lot of versatility, letting you point a branch at an arbitrary commit, and drop or squash the last few commits on a branch, the above workflow of amending just the latest commit is so common that it has its own command: `commit --amend`. Normally, the `commit` command writes the current index as a tree object, and uses the current `HEAD` as the parent for a new commit, which becomes the new `HEAD`.

Figure 22.8. The commit command creates a new commit



When used with the `--amend` switch, `commit` still writes the current index out as a tree, but instead of appending a new commit to the current `HEAD`, it builds a new commit with the same parents as the current `HEAD`, effectively replacing the latest commit.

Figure 22.9. The `commit --amend` command replaces an existing commit



This command retains the author and message of the replaced commit, and lets the user edit the message before committing. Because it preserves all the parents of the original `HEAD`, it's strictly more powerful than the `reset/commit` workflow described above; you can amend a merge commit and the replacement commit will itself be a merge commit, with the same parents as the old `HEAD`.

Let's declare this option and dispatch to a new method to handle it in the `Command::Commit#run` method.

```

# lib/command/commit.rb

def define_options
  define_write_commit_options

  @parser.on("-amend") { @options[:amend] = true }

  #
end

def run
  repo.index.load

  handle_amend if @options[:amend]
  resume_merge if pending_commit.in_progress?

  #
end
  
```

The `handle_amend` method looks much like the other code we've seen for writing commits. It loads the current `HEAD` ID and writes the current index out to a tree. Then, rather than reading the message from user input, it's read from the current `HEAD`. This value, and the parents and author of the old `HEAD` are used with the new tree to build a commit, which is then stored and set as the new `HEAD`.

```

# lib/command/commit.rb

def handle_amend
  old  = repo.database.load(repo.refs.read_head)
  tree = write_tree

  message = compose_message(old.message)

  new = Database::Commit.new(old.parents, tree.oid, old.author, message)
  
```

```
repo.database.store(new)
repo.refs.update_head(new.oid)

print_commit(new)
exit 0
end
```

Note that, although `commit --amend` potentially makes the original `HEAD` commit unreachable, it does not set `ORIG_HEAD`. If you have no other references to the original commit, it's effectively lost.

22.3.2. Recording the committer

In Section 2.3.4, “Commits on disk”, we learned that Git commits contain two similar headers that store a person’s name, email and a timestamp: `author` and `committer`. So far, our implementation has always set these to the same value. But the `--amend` option is one of the cases in which Git makes them differ: it reuses the `author` information from an old commit, but the `committer` field should always reflect the actual person and time at which the commit object was created.

This serves two important purposes. First, it allows end users to find out both the original author of the content, and the person who created the commit graph. This becomes especially useful when accepting patches from other people or cherry-picking commits between branches. You want to be able to find out both who wrote the code, and who wrote the commit history. Second, we saw in Section 16.2.2, “Logging multiple branches” that the `RevList` class uses commit timestamps to order its search of the history and determine when to stop. Amending and reordering commits can mean the `author` field of one commit might show an earlier time than its parent, and this would confuse the history search. The `committer` field lets us retain an always-increasing timestamp on descendant commits.

We need to make room for the `Database::Commit` class to have distinct `author` and `committer` fields, so let’s add another field to its constructor:

```
# lib/database/commit.rb

attr_reader :parents, :tree, :author, :committer, :message

def initialize(parents, tree, author, committer, message)
  @parents = parents
  @tree = tree
  @author = author
  @committer = committer
  @message = message
end
```

We’ll need to use the `@committer` variable rather than the `@author` when writing the commit out as a string:

```
# lib/database/commit.rb

def to_s
  lines = []

  lines.push("tree #{ @tree }")
```

```

    lines.concat(@parents.map { |oid| "parent #{ oid }" })
    lines.push("author #{ @author }")
    lines.push("committer #{ @committer }")
    lines.push("")
    lines.push(@message)

    lines.join("\n")
end

```

And, when reading commits back from the database, we'll need to use the `committer` header to populate this new property on the `Commit` object.

```

# lib/database/commit.rb

def self.parse(scanner)
headers = Hash.new { |hash, key| hash[key] = [] }

loop do
# ...
end

Commit.new(
headers["parent"],
headers["tree"].first,
Author.parse(headers["author"].first),
Author.parse(headers["committer"].first),
scanner.rest)
end

```

Now, we can change the `Commit#date` method to return the `committer` time, so that `RevList` continues to sort commits in the right order.

```

# lib/database/commit.rb

def date
@committer.time
end

```

Having made this change to the data model, all the places in the codebase that create new commits need to set the `committer` field. They'll need a shared way of getting the current author and timestamp, so we'll extract that into a method in `writeCommit`.

```

# lib/command/shared/write_commit.rb

def current_author
name = @env.fetch("GIT_AUTHOR_NAME")
email = @env.fetch("GIT_AUTHOR_EMAIL")

Database::Author.new(name, email, Time.now)
end

```

Then there are just two places we need to change: `writeCommit` handles the normal `commit` command, and successful and resumed merges. Here we use the current author value for both the `author` and `committer` fields.

```

# lib/command/shared/write_commit.rb

def write_commit(parents, message)

```

```
# ...

tree    = write_tree
author = current_author
commit = Database::Commit.new(parents, tree.oid, author, author, message)

# ...
end
```

The other place that writes commits is the `Command::Commit#handle_amend` method. Here, we reuse the `author` field from the old commit, and use the current author value as the committer.

```
# lib/command/commit.rb

def handle_amend
  old  = repo.database.load(repo.refs.read_head)
  tree = write_tree

  message  = amend_commit_message(old)
  committer = current_author

  new = Database::Commit.new(old.parents, tree.oid, old.author, committer, message)

  #
end
```

We now have a workflow that lets us edit commits at will, replacing commits we no longer want with corrected versions. Let's take it for a spin by creating a new repository and adding some commits to it:

```
$ jit init .. /amend
$ cd .. /amend
$ for msg in one two three ; do
>   echo "$msg" > file.txt
>   jit add .
>   jit commit --message "$msg"
> done
```

Using the `log` command, we can see the history up to the current HEAD, and inspect the diff of the latest commit:

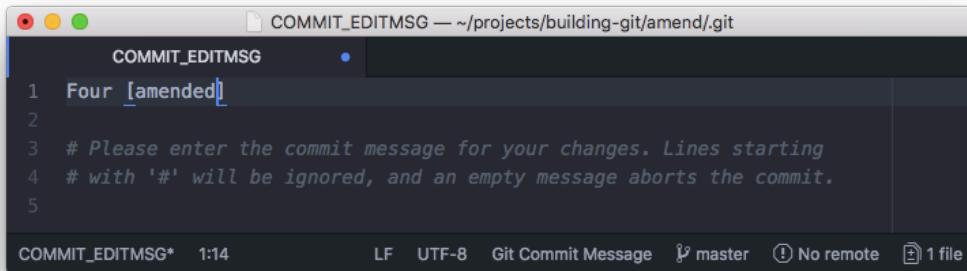
```
$ jit log --oneline
3674638 (HEAD -> master) three
b69353c two
67b69bb one

$ jit log --oneline --patch @^..
3674638 (HEAD -> master) three
diff --git a/file.txt b/file.txt
index f719efd..2bdf67a 100644
--- a/file.txt
+++ b/file.txt
@@ -1,1 +1,1 @@
-two
+three
```

Now, let's change the contents of `file.txt` and add it to the index, and then amend the latest commit:

```
$ export GIT_EDITOR="atom --wait"  
  
$ echo "four" > file.txt  
$ jit add .  
$ jit commit --amend
```

As expected, our editor pops up, and we can edit the message it presents to us.



Checking the logs after making this amendment confirms that the previous HEAD has been replaced by our amended commit.

```
$ jit log --oneline  
dc98cd4 (HEAD -> master) Four [amended]  
b69353c two  
67b69bb one  
  
$ jit log --oneline --patch @^..  
dc98cd4 (HEAD -> master) Four [amended]  
diff --git a/file.txt b/file.txt  
index f719efd..8510665 100644  
--- a/file.txt  
+++ b/file.txt  
@@ -1,1 +1,1 @@  
-two  
+four
```

The `commit --amend` command is the first tool we've seen that lets us 'modify' the project history, by replacing one commit with another. In the next chapter we'll investigate some more powerful tools for manipulating history.

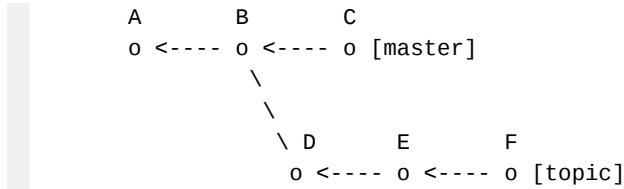
23. Cherry-picking

We've just taken our first steps into seeing how Git allows history to be 'modified': the commits themselves are not changed, but new commits are added to the graph and the refs are changed to point at these replacement commits. A combination of the `reset` and `commit` commands lets us effectively edit the latest commit on a branch, or squash the last few commits into a single one.

Git provides even more powerful ways to construct different histories from existing commits. We won't cover them in full here, but we will explore one fundamental operation from which most other history-changing commands can be built. That operation is called `cherry-pick`, and it demonstrates another application of merging while providing a powerful building block for more complicated workflows.

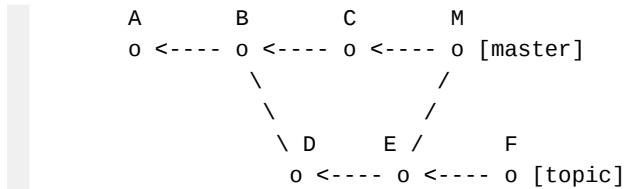
Suppose we have a history like this, in which two branches, `master` and `topic`, refer to concurrent commits:

Figure 23.1. History with two branches



It turns out that the `topic` branch contains a change that would be useful for the work being done on `master`. For example, commit `E` contains a bug fix that we'd like to get into production without deploying the entire `topic` branch. Addressing `E` with the `merge` command is simple enough — running `merge topic^` from the `master` branch will result in this history:

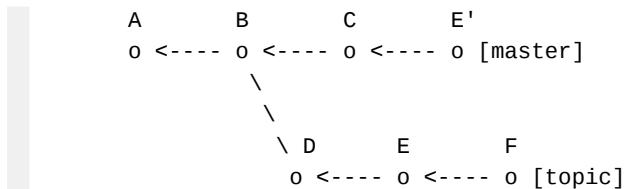
Figure 23.2. Merging part of the topic branch into master



However, commit `M` now contains more changes than we wanted: as well as the contents of `E` we've also merged the contents of `D`. Remember: merging `E` into `C` means `B` is the base of the merge, and all changes between `B` and `E` are applied to `C`.

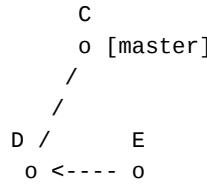
What we really want is a history like this, in which just the change introduced by `E` is applied on top of `C`, to produce `E'`:

Figure 23.3. Cherry-picking a single commit into master



The tree of commit E' , denoted $T_{E'}$, should consist of the tree of C , plus the difference between E and its parent, that is, $T_C + d_{DE}$. From Section 17.1.3, “Interpreting merges”, we know that one way to construct such a tree is to perform a merge between C and E with D as the base. That is, a merge pretending the commit graph looked like this:

Figure 23.4. Imagined history graph for a cherry-pick merge



Why should we perform a merge, rather than merely applying d_{DE} to the current `HEAD` using a migration? Because the changes on the target branch may overlap with the changes from the cherry-picked commit. In our example, C is not an ancestor of E and so E is not derived from it. If C and E both change the same file, we’d like to get a conflict, rather than having the cherry-pick overwrite our own branch’s changes.

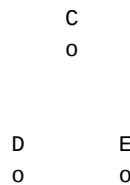
Now, the history graph does not look like this, so we should question whether this is a legit thing to do. Well, in Chapter 18, *When merges fail* we divided the merge operation into two classes: `Merge::Inputs` takes two input revisions and computes their best common ancestor in the history graph — in this case the inputs would be C and E and the base commit would be B . This `Inputs` object is then passed as an argument to `Merge::Resolve` which detects any conflicts between the two branches and updates the workspace and index accordingly. Importantly, the `Resolve` operation does not care how `Inputs` finds the base of the merge; it only needs to be given two commits and some base commit.

Since the purpose of the `merge` command is to combine all the changes on two branches since they diverged, it makes sense to take their best common ancestor as the base for the `Resolve` operation. However, in general we can use `Resolve` to combine any two commits with any third commit as the base, because this class ignores the history graph and just computes the tree diff for each side of the merge.

So, this tells us that technically we can run a ‘merge’ between C and E with D rather than B as the base. But will this give us a meaningful answer? Such a merge would produce the tree $T_D + d_{DC} + d_{DE}$ — the tree of D plus the difference between D and each side of the merge. We do want the change d_{DE} in our cherry-picked commit, but we don’t want the content from D , so we’d like to know that the term d_{DC} somehow removes the content from D that we don’t want to keep.

We’ve established that `Resolve` does not use the history graph, so from its point of view these three commits are just three unconnected objects that point to trees.

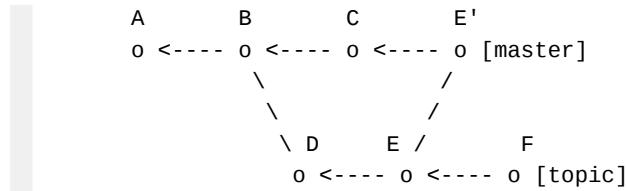
Figure 23.5. Merge input commits with history links removed



The difference d_{DC} is the set of changes required to convert tree T_D into T_C . By definition, $T_C = T_D + d_{DC}$ and so any content that exists in D but not in C should be removed by this term. If this is still a little abstract, refer back to Figure 23.1, “History with two branches”. To get from D to C , we need to walk back to B —effectively undoing the changes from D —and then forward to C , therefore incorporating the changes from C . Therefore this operation does genuinely produce the tree $T_C + d_{DE}$ as required.

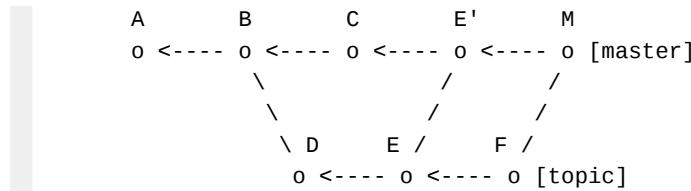
Following a normal merge, the resulting commit has two parents, which are the input commits to the merge. In this case the following history would result:

Figure 23.6. Merge commit pointing to the two input commits



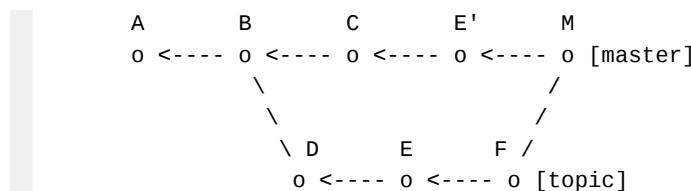
However, the commit graph needs to function such that the `merge` command incorporates all changes between two branches since they diverged. If we maintain a parent link between E and E' , and subsequently merge `topic` into `master`, the resulting history will be:

Figure 23.7. Merge commit following a previous merge



The merge M will use commit E as its base, and will therefore not incorporate commit D into T_M . To make sure the whole `topic` branch is incorporated, we need to base this merge on B , and this can be done if the link between E and E' is dropped.

Figure 23.8. Cherry-pick commit unconnected to its source



A merge of `topic` into `master` will now be based on B and will thus incorporate changes from D into the result.

Now, you may be concerned that since E and E' contain the same changes, the merge may become conflicted, or duplicate changes. Indeed, our notation for computing new trees tells us that $T_M = T_B + d_{BE'} + d_{BF}$. And, the net difference between two commits should equal the difference of each individual commit in the chain added together, so $d_{BE'} = d_{BC} + d_{CE'}$ and d_{BF}

$= d_{BD} + d_{DE} + d_{EF}$. But, if the cherry-pick operation works, $d_{CE'}$ is the same as d_{DE} , and so T_M seems to include this change twice. However, we saw in Section 18.4, “Conflict detection” and Chapter 20, *Merging inside files* that if two merge inputs contain the same change relative to the base, this does not result in a conflict and the content is not duplicated. The `+` operator in these expressions works less like arithmetic addition and more like set addition; adding two sets together does not duplicate their common elements:

```
>> Set.new([1, 2, 3]) + Set.new([3, 4, 5])
=> #<Set: {1, 2, 3, 4, 5}>
```

Indeed, a tree is a set of files, and a file is a set of lines, as far as the merge algorithms are concerned. And so, cherry-picking a commit between branches should not result in conflicts or duplication when the two branches are fully merged.

Having established that we can use the merge machinery to cherry-pick commits, we can begin implementation. First we’ll deal with cherry-picking a single commit, and build on this so we can recover if the merge results in a conflict. Then we’ll expand the command to accept ranges of commits, and again put tools in place so that we can resume the cherry-pick if any commit in the range produces a conflict.

23.1. Cherry-picking a single commit

We know we want to reuse the `Merge::Resolve` class to perform the merge, and that class takes a `Merge::Inputs` object as an argument. But, Ruby being a dynamically typed language, we don’t actually need to use a `Merge::Inputs` object—we just need something with the same interface. Rather than using `Merge::Inputs`, which calculates the base of the merge using `Merge::Bases`, we could pass in a new object that has its base commit set in some other manner.

The `Merge::Inputs` class has a few methods that `Merge::Resolve` relies on: `left_oid` and `right_oid`, which identify the input commits, `left_name` and `right_name` which provide the names of the commits to use in conflict sections in `Merge::Diff3`, and `base_oids`, which returns an array of commit IDs to use as the merge base. We can create a simple struct that has these same fields, and that gives us another kind of object that `Merge::Resolve` will accept.

```
# lib/merge/inputs.rb

module Merge
  class Inputs

    ATTRS = [:left_name, :right_name, :left_oid, :right_oid, :base_oids]
    attr_reader(*ATTRS)

    #
  end

  CherryPick = Struct.new(*Inputs::ATTRS)
end
```

We can now perform a cherry-pick by constructing a `Merge::CherryPick` object and feeding it into `Merge::Resolve`, and that’s what our `cherry-pick` command will do. Here’s the beginning of the `Command::CherryPick` class; it resolves the given argument to a commit and then passes that to another method `pick` which actually performs the operation.

```
# lib/command/cherry_pick.rb

module Command
  class CherryPick < Base

    include WriteCommit

    def run
      revision = Revision.new(repo, @args[0])
      commit   = repo.database.load(revision.resolve)

      pick(commit)

      exit 0
    end

    #
  end
end
```

The `pick` method performs the required merge and writes a new commit. It calls `pick_merge_inputs` to build a `Merge::CherryPick` object for the merge, then `resolve_merge` to execute the merge. Finally it builds a new `Commit` object whose parent is the current HEAD, and whose author and message are taken from the picked commit. The `committer` field, just like in the `commit --amend` command, reflects the current user and time, so we know who cherry-picked this commit as well as who originally wrote its content. The call to `finish_commit` saves and prints the commit.

```
# lib/command/cherry_pick.rb

def pick(commit)
  inputs = pick_merge_inputs(commit)

  resolve_merge(inputs)

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

  finish_commit(picked)
end
```

The `pick_merge_inputs` method constructs the input to the `Merge::Resolve` class that describes the required merge. Its `left` properties are taken from the current HEAD, while the `right` properties reflect the picked commit; the `right_name` is a brief human-readable description of the commit. The `base_oids` field is an array containing just the parent of the picked commit.

```
# lib/command/cherry_pick.rb

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

  left_name  = Refs::HEAD
  left_oid   = repo.refs.read_head
```

```
    right_name = "#{ short }... #{ commit.title_line.strip }"
    right_oid  = commit.oid

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

Finally, the `resolve_merge` and `finish_commit` methods are some bits of glue that we've seen before. `resolve_merge` executes the merge with the index opened for modification, and `finish_commit` saves the commit to the database, updates HEAD, and prints a brief description of the commit to the user.

```
# lib/command/cherry_pick.rb

def resolve_merge(inputs)
  repo.index.load_for_update
  ::Merge::Resolve.new(repo, inputs).execute
  repo.index.write_updates
end

def finish_commit(commit)
  repo.database.store(commit)
  repo.refs.update_head(commit.oid)
  print_commit(commit)
end
```

23.1.1. New types of pending commit

Since we're using a merge to execute our cherry-pick operation, it's possible that we'll end up with a merge conflict, and we need to exit and let the user resolve those conflicts before letting them continue the cherry-pick. The `merge` command uses the `Repository::PendingCommit` class to store the necessary state: the merged commit's ID in `.git/MERGE_HEAD` and the merge message in `.git/MERGE_MSG`. If we run `commit` while `.git/MERGE_HEAD` exists, a merge commit will be created with its parents taken from HEAD and `.git/MERGE_HEAD`.

As we saw above, when we resume a cherry-pick we don't want to create a merge commit, since it would block changes from earlier in the branch being incorporated when the branch is fully merged. So, we cannot reuse this infrastructure for resuming cherry-picks without having some indication of what *type* of merge we were in the middle of. Git handles this by storing the cherry-picked ID in `.git/CHERRY_PICK_HEAD` instead of `.git/MERGE_HEAD`, and we can tell from the existence or otherwise of these files what to do on `commit`.

Let's amend the `PendingCommit` commit class so that it can be told which type of merge is happening. Rather than storing `@head_path` immediately in its initializer, we'll just store the given directory path and figure out the right path from `HEAD_FILES` when `start` is called.

```
# lib/repository/pending_commit.rb

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

```
def initialize(pathname)
  @pathname = pathname
  @message_path = pathname.join("MERGE_MSG")
end

def start(oid, type = :merge)
  path = @pathname.join(HEAD_FILES.fetch(type))
  flags = File::WRONLY | File::CREAT | File::EXCL
  File.open(path, flags) { |f| f.puts(oid) }
end
```

Detecting whether a merge is in progress is now a question of determining which type of merge is in progress, if any.

```
# lib/repository/pending_commit.rb

def merge_type
  HEAD_FILES.each do |type, name|
    path = @pathname.join(name)
    return type if File.file?(path)
  end

  nil
end

def in_progress?
  merge_type != nil
end
```

The methods for reading the merge commit ID, and clearing the pending commit state, should now both take a type argument; running `merge --continue` should continue to read `.git/MERGE_HEAD` while `cherry-pick --continue` would read `.git/CHERRY_PICK_HEAD`. These commands should not simply resume whichever merge type happens to be in progress.

```
# lib/repository/pending_commit.rb

def merge_oid(type = :merge)
  head_path = @pathname.join(HEAD_FILES.fetch(type))
  File.read(head_path).strip
rescue Errno::ENOENT
  name = head_path.basename
  raise Error, "There is no merge in progress (#{} name } missing)."
end

def clear(type = :merge)
  head_path = @pathname.join(HEAD_FILES.fetch(type))
  File.unlink(head_path)
  File.unlink(@message_path)
rescue Errno::ENOENT
  name = head_path.basename
  raise Error, "There is no merge to abort (#{} name } missing)."
end
```

With `PendingCommit` so updated, we must change the `Command::Commit` and `Command::Merge` classes to specify which type of merge should be used. First, `merge --continue` should call `resume_merge(:merge)` so that it specifically only resumes true merges.

```
# lib/command/merge.rb

def handle_continue
  repo.index.load
  resume_merge(:merge)
rescue Repository::PendingCommit::Error => error
  @stderr.puts "fatal: #{error.message}"
  exit 128
end
```

The `commit` command is a little more dynamic: running `commit` with any type of pending commit should resume that commit. So, it first checks whether any type of merge is in progress, and resumes the merge with that type if so.

```
# lib/command/commit.rb

def run
  repo.index.load

  handle_amend if @options[:amend]

  merge_type = pending_commit.merge_type
  resume_merge(merge_type) if merge_type

  #
end
```

`resume_merge` is a method in `WriteCommit`, and for the time being we'll just move its existing logic into `write_merge_commit`, and dispatch to that method if `resume_merge(:merge)` is called. Logic for other merge types can be slotted in here later.

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

def resume_merge(type)
  case type
  when :merge then write_merge_commit
  end

  exit 0
end

def write_merge_commit
  #
  pending_commit.clear(:merge)
end
```

23.1.2. Resuming from conflicts

The above changes to `PendingCommit` mean we can now pause and resume the `cherry-pick` command. In `Command::CherryPick#pick`, we'll now call `fail_on_conflict` if the merge results in a conflicted index.

```
# lib/command/cherry_pick.rb

def pick(commit)
```

```
inputs = pick_merge_inputs(commit)
resolve_merge(inputs)
fail_on_conflict(inputs, commit.message) if repo.index.conflict?

#
end
```

The `fail_on_conflict` method takes care of saving the state of the cherry-pick operation and then exiting with an error. It first calls `PendingCommit#start` to save the picked commit's ID, with the `merge_type` set to `:cherry_pick`. Then it uses `Editor` to write into `.git/MERGE_MSG`, where it stores the picked commit's message, and a list of the files that were conflicted as comments.

```
# lib/command/cherry_pick.rb

CONFLICT_NOTES = <<-MSG
  after resolving the conflicts, mark the corrected paths
  with 'git add <paths>' or 'git rm <paths>'
  and commit the result with 'git commit'
MSG

def fail_on_conflict(inputs, message)
  pending_commit.start(inputs.right_oid, merge_type)

  edit_file(pending_commit.message_path) do |editor|
    editor.puts(message)
    editor.puts("")
    editor.note("Conflicts:")
    repo.index.conflict_paths.each { |name| editor.note("\t#{ name }") }
    editor.close
  end

  @stderr.puts "error: could not apply #{ inputs.right_name }"
  CONFLICT_NOTES.each_line { |line| @stderr.puts "hint: #{ line }" }
  exit 1
end

def merge_type
  :cherry_pick
end
```

Now we can define how `cherry-pick --continue` should work, as well as the `commit` command when `.git/CHERRY_PICK_HEAD` exists. This is going to be much like resuming a conflicted merge, and we can now slot a new branch into `WriteCommit#resume_merge` to handle resuming a cherry-pick. The `commit` command will call this when a pending cherry-pick is in progress.

```
# 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
  end

  exit 0
end
```

Like `write_merge_commit`, `write_cherry_pick_commit` begins by checking if the index is in conflict, and exits with an error if so. Otherwise, it constructs a new commit, adding the contents of `CHERRY_PICK_NOTES` to the comments in the editor when we prompt the user to edit the commit message. If everything goes smoothly, we update `HEAD` to point to the picked commit and clear the pending commit state.

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

CHERRY_PICK_NOTES = <<~MSG

  It looks like you may be committing a cherry-pick.
  If this is not correct, please remove the file
  \t.git/CHERRY_PICK_HEAD
  and try again.

MSG

def write_cherry_pick_commit
  handle_conflicted_index

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

  pick_oid = pending_commit.merge_oid(:cherry_pick)
  commit    = repo.database.load(pick_oid)

  picked = Database::Commit.new(parents, write_tree.oid,
                                commit.author, current_author,
                                message)

  repo.database.store(picked)
  repo.refs.update_head(picked.oid)
  pending_commit.clear(:cherry_pick)
end
```

All that remains is to define `cherry-pick --continue` to invoke the same logic as `commit`. As usual we define the option, call a special method if it's set, and that method just loads the index and calls `write_cherry_pick_commit` directly.

```
# lib/command/cherry_pick.rb

def define_options
  @options[:mode] = :run
  @parser.on("--continue") { @options[:mode] = :continue }
end

def run
  handle_continue if @options[:mode] == :continue

  #
end

def handle_continue
  repo.index.load
  write_cherry_pick_commit
  exit 0
rescue Repository::PendingCommit::Error => error
  @stderr.puts "fatal: #{error.message}"
end
```

```
    exit 128
end
```

We're now able to execute a single cherry-pick and recover if it results in a conflict. Next we'll add support for cherry-picking multiple commits in a single operation.

23.2. Multiple commits and ranges

As well as letting you cherry-pick a single commit, Git's `cherry-pick` command lets you select multiple commits to pick in a single command, as in `cherry-pick A B C...`. There are two main use cases for using this option rather than cherry-picking each commit individually. First, it supports ranges, as in `cherry-pick A..B` or `cherry-pick X ^Y`, and this may be much easier than finding out the ID of each commit you want. Second, it lets you treat a set of cherry-picks as an atomic transaction and roll back to the pre-cherry-pick state should any single commit fail to merge cleanly.

The command accepts all the same types of input as the `log` command for selecting ranges of commits, so we can use the `RevList` class¹ to locate all the commits we should pick. There's a couple of quirks to how `cherry-pick` uses `RevList` that we need to address before moving on.

23.2.1. Rev-list without walking

First, `cherry-pick` should apply commits in the same order they appear in the history. That is, given the history in Figure 23.1, “History with two branches”, `cherry-pick master..topic` should apply *D*, then *E*, then *F*. However, `RevList` will iterate the range `master..topic` in the opposite order, being designed primarily to serve the `log` command which begins from branch pointers. We need to iterate the commits in reverse order, which can be achieved by adding this to the `RevList` class:

```
# lib/rev_list.rb

include Enumerable
```

This will make the method `reverse_each`² available on the `RevList` class, so we don't need to implement reverse iteration ourselves.

The second quirk is more subtle. If the revisions passed to `cherry-pick` are not ranges, then two things must happen. First, we should not iterate from those commits: whereas `cherry-pick A..B` picks every commit from *A* to *B*, `cherry-pick A B` picks *only* commits *A* and *B*, not every commit reachable from them. So we need an option on `RevList` to tell it not to iterate the history unless any of the inputs are ranges. Second, the commits should be picked in the order they're given, whereas `RevList` will put its inputs into a queue sorted by the commit date. This sorting must also be switched off unless any of the inputs are ranges.

We can do this by adding a `:walk` option to `RevList`, which defaults to `true`. If the caller sets it to `false`, and none of the inputs are ranges, then we want to avoid sorting the commits and iterating their parents.

```
# lib/rev_list.rb
```

¹Section 16.2.1, “Revision lists”

²https://docs.ruby-lang.org/en/2.3.0/Enumerable.html#method-i-reverse_each

```
def initialize(repo, revs, options = {})
# ...

@walk = options.fetch(:walk, true)

revs.each { |rev| handle_revision(rev) }
handle_revision(Revision::HEAD) if @queue.empty?
end
```

If the range notation A..B or exclude notation ^A are used, then @walk is set to true; we only want to stop walking when no ranges are given.

```
# lib/rev_list.rb

def handle_revision(rev)
  if @repo.workspace.stat_file(rev)
    @prune.push(Pathname.new(rev))
  elsif match = RANGE.match(rev)
    set_start_point(match[1], false)
    set_start_point(match[2], true)
    @walk = true
  elsif match = EXCLUDE.match(rev)
    set_start_point(match[1], false)
    @walk = true
  else
    set_start_point(rev, true)
  end
end
```

In the enqueue_commit method which adds commits to the input queue, we will now only keep the queue sorted if @walk is enabled. If not, we'll push the commit on the end of the queue. This will make RevList yield the inputs in the order they're given.

```
# lib/rev_list.rb

def enqueue_commit(commit)
  return unless mark(commit.oid, :seen)

  if @walk
    index = @queue.find_index { |c| c.date < commit.date }
    @queue.insert(index || @queue.size, commit)
  else
    @queue.push(commit)
  end
end
```

Finally the add_parents method, which adds the parents of each found commit to the input queue, should do nothing unless @walk is set. This prevents RevList iterating over all the commits reachable from the inputs, and it will yield only the inputs themselves.

```
# lib/rev_list.rb

def add_parents(commit)
  return unless @walk and mark(commit.oid, :added)

# ...
end
```

Git's log and rev-list commands have --no-walk and --do-walk options that trigger exactly the above behaviour.

We can now use RevList in the cherry-pick command. One final quirk remains: since we're going to iterate the RevList in reverse order, but we want the input order to be maintained unless any ranges are used, we reverse the inputs that are passed to RevList. This is a little complication but it means all the logic about processing ranges versus other kinds of input remains in the RevList class rather than leaking into other command classes.

```
# lib/command/cherry_pick.rb

def run
  handle_continue if @options[:mode] == :continue

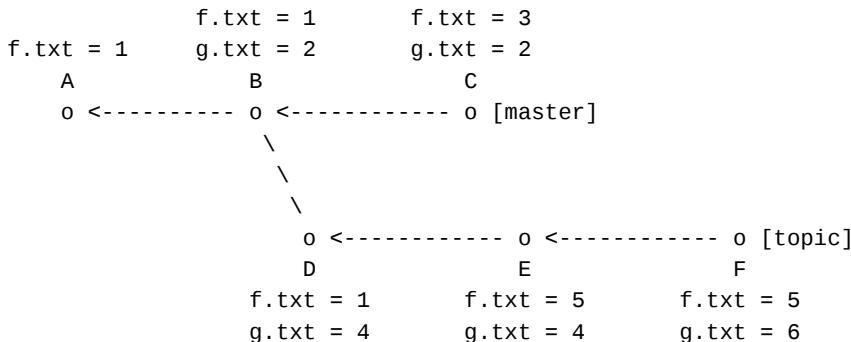
  commits = RevList.new(repo, @args.reverse, :walk => false)
  commits.reverse_each { |commit| pick(commit) }

  exit 0
end
```

23.2.2. Conflicts during ranges

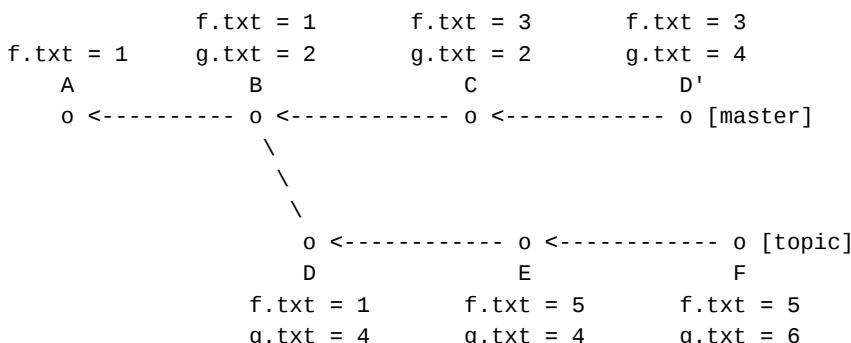
We're now able to process multiple commits with a single command, and we're able to recover if a single commit fails. However, we're still unable to recover if a series of commits fails part-way through. For example, consider this history, where each commit is annotated with its complete tree.

Figure 23.9. History with conflicting changes



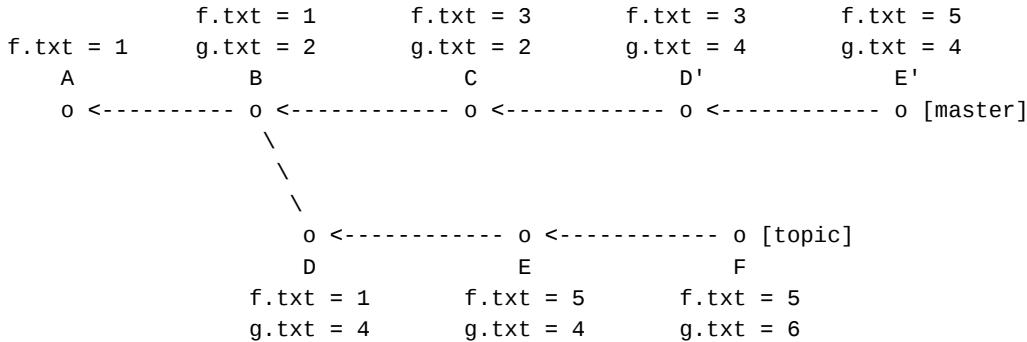
Let's say that `master` is currently checked out, and we run `cherry-pick ..topic`, which will attempt to pick commits `D`, `E` and `F`. `D` applies cleanly; the tree diff d_{BC} is $\{ f.txt \Rightarrow [1, 3] \}$ while d_{BD} is $\{ g.txt \Rightarrow [2, 4] \}$. These differences do not overlap, so `D` is picked cleanly:

Figure 23.10. Partially completed range cherry-pick



Now we try to pick E , using its parent D as the base, onto commit D' . $d_{DD'}$ is $\{ f.txt \Rightarrow [1, 3] \}$ while d_{DE} is $\{ f.txt \Rightarrow [1, 5] \}$. These changes overlap and we have a conflict on file $f.txt$. The cherry-pick command should stop here and let the user sort the conflict out. Let's say the user picks 5 — the content from E — as the merge result, so we get:

Figure 23.11. Partially completed range cherry-pick with conflicts resolved



Having done this, we should be able to resume our original cherry-pick of the range \dots topic, which means picking commit F . You should convince yourself that F will apply cleanly on top of E' ; compare $d_{EE'}$ with d_{EF} .

Our current implementation might resume commit E' correctly, but after that it will just stop. It only remembers a single pending commit, and so it forgets about commit F as soon as a merge conflict happens. We need to store some more state so that we can resume and complete the range.

If you try to recreate the above scenario using Git, you'll see that it stores this state in the directory `.git/sequencer`. Within that directory are several files. The file `todo` stores a list of the commits that are yet to be completed; it stores their abbreviated ID and title line. Here's a run-through of inspecting the `topic` branch before the cherry-pick, and then running it until the first conflict.

```

$ git log --oneline ..topic
364c819 (topic) F
50227dd E
b0f24f7 D

$ git cherry-pick ..topic
[master 5d5f9b4] D
 1 file changed, 1 insertion(+), 1 deletion(-)
error: could not apply 50227dd... E

$ ls .git/sequencer
abort-safety head      todo

$ cat .git/sequencer/todo
pick 50227dd E
pick 364c819 F

```

Commit D is successfully picked, but then commit E fails. At this point `.git/sequencer/todo` lists commits E and F as remaining to be completed. Alongside `todo` are two more files:

```

$ cat .git/sequencer/abort-safety
5d5f9b4574171693aa4dfa06c005e93a0c147f6

```

```
$ cat .git/sequencer/head  
be61f5189ae25c34bff82272d3d131af991f8a49
```

`abort-safety` contains the ID of commit D' , the copy of D that was cherry-picked onto C , which is the last commit that `cherry-pick` created. `head` contains the ID of C , which is the commit that `HEAD` was pointing at before we began the cherry-pick. These files will let us abandon and undo the whole operation, and we'll get to that shortly.

Examining these files suggests that we need a mechanism that stores the list of commits to pick when the command is first run, and the current `HEAD` commit. It picks commits from that list one by one; when a commit is successfully picked then we drop it from the list and update `abort-safety` to point to it. Otherwise we interrupt the command for the user to fix the merge conflicts. When we run `cherry-pick --continue`, if the index is not conflicted, we should create a new commit as we already do, then drop the first commit from the to-do list and continue processing it.

I'm going to introduce a class called `Repository::Sequencer` to manager this state; we'll add a method to `Command::CherryPick` for creating it.

```
# lib/command/cherry_pick.rb  
  
def sequencer  
  @sequencer ||= Repository::Sequencer.new(repo)  
end
```

The `cherry-pick` command will now begin by calling `sequencer.start`, which creates the initial sequencer state or fails if `.git/sequencer` already exists. It then uses `RevList` to grab the commit list, but instead of picking each one immediately, it stores it in the to-do list by calling `sequencer.pick`. Finally, we call `resume_sequencer`, which begins working from wherever the sequencer state currently is.

```
# lib/command/cherry_pick.rb  
  
def run  
  handle_continue if @options[:mode] == :continue  
  
  sequencer.start  
  store_commit_sequence  
  resume_sequencer  
end  
  
def store_commit_sequence  
  commits = RevList.new(repo, @args.reverse, :walk => false)  
  commits.reverse_each { |commit| sequencer.pick(commit) }  
end
```

The `resume_sequencer` method is a loop that takes the next commit from the sequencer, tries to `cherry-pick` it, and drops the commit from the to-do list if successful. Calling `pick` will exit with an error in the event of a conflict. If we run out of commits then we call `sequencer.quit` to clean up the state.

```
# lib/command/cherry_pick.rb  
  
def resume_sequencer  
  loop do
```

```
    break unless commit = sequencer.next_command
    pick(commit)
    sequencer.drop_command
  end

  sequencer.quit
  exit 0
end
```

The Sequencer class itself is instantiated with a Repository, and computes the full path for the .git/sequencer directory, along with the path for the todo file. It contains an array called @commands that will store the list of commits we want to process.

```
# lib/repository/sequencer.rb

class Repository
  class Sequencer

    def initialize(repository)
      @repo      = repository
      @pathname  = repository.git_path.join("sequencer")
      @todo_path = @pathname.join("todo")
      @todo_file = nil
      @commands  = []
    end

    #
    # ...

  end
end
```

The start method simply creates the .git/sequencer directory — this will raise an exception if the directory already exists. It also opens the to-do file, which we'll define below. pick adds a new commit to the list of commands, next_command returns the first commit in the list and drop_command removes it.

```
# lib/repository/sequencer.rb

def start
  Dir.mkdir(@pathname)
  open_todo_file
end

def pick(commit)
  @commands.push(commit)
end

def next_command
  @commands.first
end

def drop_command
  @commands.shift
end
```

If we're going to exit the cherry-pick command on conflicts and resume it later, this state needs to be written to and recalled from disk. The open_todo_file method uses Lockfile to

open the todo file for editing — we need to acquire this lock before attempting to change any sequencer state.

```
# lib/repository/sequencer.rb

def open_todo_file
  return unless File.directory?(@pathname)

  @todo_file = Lockfile.new(@todo_path)
  @todo_file.hold_for_update
end
```

So far, the sequencer state only exists in memory, and this is fine as long as commits continue to apply cleanly. Once we get a conflict, we need to dump its current state to disk; that means `fail_on_conflict` should begin by calling `sequencer.dump` before continuing with its existing logic. When we resume the command using the `--continue` option, we need to reload the sequencer state and continue processing the commits. Therefore the `handle_continue` method will write the pending commit as before, but now it will continue the range by calling `sequencer.load`, using `drop_command` to remove the first commit from the list, and then running `resume_sequencer` to continue the work.

```
# lib/command/cherry_pick.rb

def fail_on_conflict(inputs, message)
  sequencer.dump
  #
end

def handle_continue
  repo.index.load
  write_cherry_pick_commit if pending_commit.in_progress?

  sequencer.load
  sequencer.drop_command
  resume_sequencer

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

`Sequencer#dump` uses the `Lockfile` that we created in `open_todo_file` to store the `@commands` array. It writes each commit to the file using the word `pick`, the commit's abbreviated ID, and its title line. The `load` method re-opens the `todo` file, and retrieves the commit list from it, parsing each line and using `Database#prefix_match`³ to expand the ID to its full length.

```
# lib/repository/sequencer.rb

def dump
  return unless @todo_file

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

³Section 13.3.3, “Revisions and object IDs”

```
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|
    oid, _ = /^pick (\S+) (.*)$/.match(line).captures
    oids = @repo.database.prefix_match(oid)
    @repo.database.load(oids.first)
  end
end
```

Finally, we need to remove the .git/sequencer directory when the command is completed, and Sequencer#quit takes care of that.

```
# lib/repository/sequencer.rb

def quit
  FileUtils.rm_rf(@pathname)
end
```

We can now cherry-pick an arbitrary set of commits using the range operators .. and ^, and the command will correctly recover from interruptions caused by merge conflicts. The resume_sequencer method is a small example of *crash-only design*⁴: there are not separate code paths for the case where all commits apply cleanly versus when the chain is interrupted by conflicts. Instead, each starting state triggers a little bit of code that puts the system in the right state — either queueing up the commits to begin with or reloading the queue from disk — and then both use cases trigger the same business logic. Special error-handling code tends to be executed less often than *happy path* code, and one way to make a system more robust is to use the same code when you’re recovering from a crash as when you’re starting the system for the first time; the system is designed so as not to assume it’s starting from a clean state. Storing up all the work that’s left to be done, and then churning through that work until you run out of work or crash, is one technique for achieving this, and is highly applicable to background-and batch-processing with database systems.

23.2.3. When all else fails

Just as with the merge command, we sometimes get bogged down in resolving conflicts, discover a mistake we need to go back and correct, or we just plain give up if we can’t resolve conflicts easily. With cherry-pick the situation is even worse, because we’re doing an arbitrary number of merges, and the way we resolve one commit might cause conflicts later in the range. We’d better have an escape hatch so we can abandon the cherry-pick and get back to a known good state.

cherry-pick provides two options for this: --abort and --quit. Let’s declare them and add handlers for these options to the run method.

```
# lib/command/cherry_pick.rb
```

⁴https://en.wikipedia.org/wiki/Crash-only_software

```
def define_options
  @options[:mode] = :run

  @parser.on("--continue") { @options[:mode] = :continue }
  @parser.on("--abort")    { @options[:mode] = :abort }
  @parser.on("--quit")     { @options[:mode] = :quit }
end

def run
  case @options[:mode]
  when :continue then handle_continue
  when :abort    then handle_abort
  when :quit     then handle_quit
  end

  sequencer.start
  store_commit_sequence
  resume_sequencer
end
```

Now, --quit is fairly straightforward. All it does is delete all the state associated with the cherry-pick, that is the .git/CHERRY_PICK_HEAD and .git/MERGE_MSG files associated with PendingCommit, and the .git/sequencer directory used by Sequencer. In effect it abandons the cherry-pick process but leaves your HEAD, index and workspace as they are.

```
# lib/command/cherry_pick.rb

def handle_quit
  pending_commit.clear(merge_type) if pending_commit.in_progress?
  sequencer.quit
  exit 0
end
```

The --abort option is more complicated. It too erases all the cherry-pick state, but it also returns you to the state of the repository before the cherry-pick started. We'll achieve this by having Command::CherryPick#handle_abort call Sequencer#abort, which will do the actual work. Since this may affect the index, the index must be loaded for updates.

```
# lib/command/cherry_pick.rb

def handle_abort
  pending_commit.clear(merge_type) if pending_commit.in_progress?
  repo.index.load_for_update

  begin
    sequencer.abort
  rescue => error
    @stderr.puts "warning: #{error.message}"
  end

  repo.index.write_updates
  exit 0
end
```

As we saw earlier, Git stores the ID of the HEAD commit before the cherry-pick began in the file .git/sequencer/head. It should just be a case of running a hard reset to this commit to

restore the index and workspace to their original state. But, things are a little more complicated than that.

A hard reset is a dangerous, destructive operation, deleting any uncommitted work from the repository, and possibly making some commits unreachable if they're not ancestors of the commit you're resetting to. While resolving a conflict, if you run `cherry-pick --continue` then the sequencer will be resumed. However, if you use `commit` to commit the resolved state, the pending cherry-pick commit will be created, but the sequencer will not resume its work. At this point you might make other commits, switch branches, and leave the repository in an arbitrary state.

To be on the safe side, the `cherry-pick` command will not reset your `HEAD`, index and workspace unless the current state is one it recognises. Specifically, it will only abort if `HEAD` is the same as the last commit that `cherry-pick` itself created. If `HEAD` contains any other value, then we've strayed from the state that the `cherry-pick` command was tracking, and it leaves things as they are.

It achieves this by storing the ID of each commit it creates in the file `.git/sequencer/abort-safety`. Every time a commit is completed, we update that file, and when `--abort` is used, its effects are only carried out if this file matches the current `HEAD` ID. This requires a few tweaks to the `Sequencer` class. First, we need to store the paths for the `head` and `abort-safety` files, and write the current `HEAD` ID to them when the process begins.

```
# lib/repository/sequencer.rb

def initialize(repository)
  @repo      = repository
  @pathname  = repository.git_path.join("sequencer")
  @abort_path = @pathname.join("abort-safety")
  @head_path = @pathname.join("head")
  @todo_path = @pathname.join("todo")
  @todo_file = nil
  @commands = []
end

def start
  Dir.mkdir(@pathname)

  head_oid = @repo.refs.read_head
  write_file(@head_path, head_oid)
  write_file(@abort_path, head_oid)

  open_todo_file
end
```

`write_file` is a helper method inside `Sequencer` that writes to the given path using a `Lockfile` for safety.

```
# lib/repository/sequencer.rb

def write_file(path, content)
  lockfile = Lockfile.new(path)
  lockfile.hold_for_update
  lockfile.write(content)
  lockfile.write("\n")
```

```
lockfile.commit  
end
```

Then, whenever a commit is completed, we need to store the new HEAD ID in the abort-safety file. When each commit is finished, the `CherryPick` command calls `Sequencer#drop_command`, so we can use this notification to update the required file.

```
# lib/repository/sequencer.rb  
  
def drop_command  
  @commands.shift  
  write_file(@abort_path, @repo.refs.read_head)  
end
```

Finally, in `Sequencer#abort`, we can get the current values of sequencer/head, sequencer/abort-safety, and HEAD, and if the latter two do not match then we raise an error. The call to `quit` removes all the state files, and if the check succeeds then we can run a hard reset to the stored commit ID. Just as when we use the `reset --hard` command, the `cherry-pick --abort` command sets `ORIG_HEAD` to the HEAD value just before the reset takes place.

```
# lib/repository/sequencer.rb  
  
UNSAFE_MESSAGE = "You seem to have moved HEAD. Not rewinding, check your HEAD!"  
  
def abort  
  head_oid = File.read(@head_path).strip  
  expected = File.read(@abort_path).strip  
  actual   = @repo.refs.read_head  
  
  quit  
  
  raise UNSAFE_MESSAGE unless actual == expected  
  
  @repo.hard_reset(head_oid)  
  orig_head = @repo.refs.update_head(head_oid)  
  @repo.refs.update_ref(Refs::ORIG_HEAD, orig_head)  
end
```

The `--abort` option is another reason for `cherry-pick` accepting multiple arguments, rather than only accepting a single commit each time. The `--abort` feature means that the set of commits to `cherry-pick` can be treated like a database transaction, in which either all the commits apply successfully or none of them does. If any commit produces a merge conflict, `--abort` can be used to go back to the state of the repository before any of the commits were applied, even if some of them have been successfully picked.

We now have a fully functional `cherry-pick` command that can pluck a sequence of changes from anywhere in the history and apply them to the current HEAD. In the next chapter, we'll look at how many of Git's history manipulation tools can be built from this operation.

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.

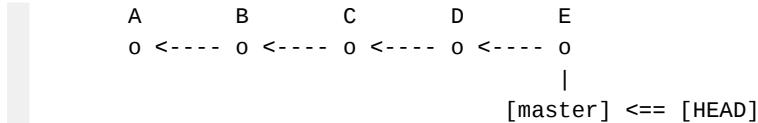
24.1. Changing old commits

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:

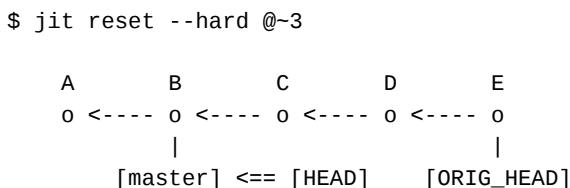
Figure 24.1. Sequence of five commits



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 $\text{ORIG_HEAD}~3..\text{ORIG_HEAD}$ 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 commute¹. 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:

¹Section 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
\           |
 \
 \
 \ 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
\           |
 \
 \
 \ 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

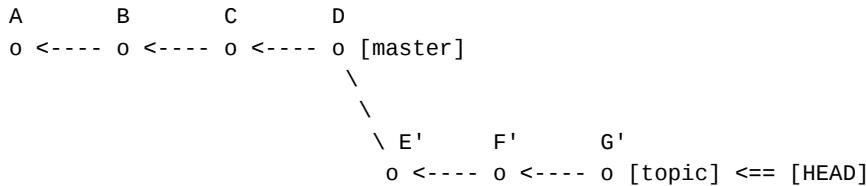
Git's rebase command is used to perform all sorts of changes to a project's history. It is highly configurable, but in its default form its job is to take a history that looks like this:

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

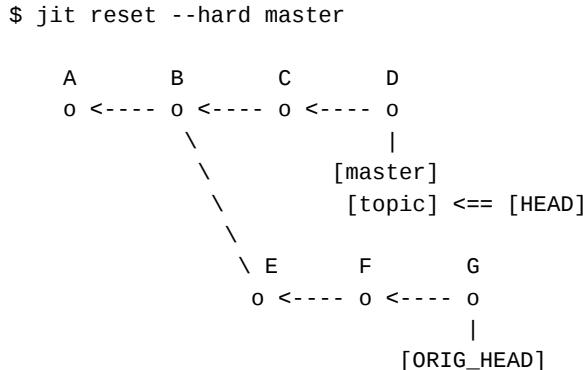


It's called *rebasing* because you are literally changing the commit your branch is based on; detaching it from its original start point and attaching it at the end of a branch so its history incorporates all the changes from that branch. This is often done to keep the history clean (avoiding merge commits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

The documentation² 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` command³ 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

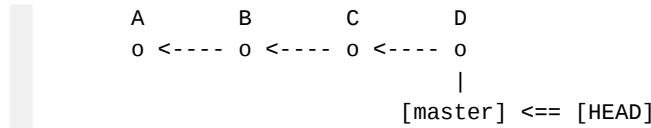
³https://git-scm.com/docs/git-rebase#_interactive_mode

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`.

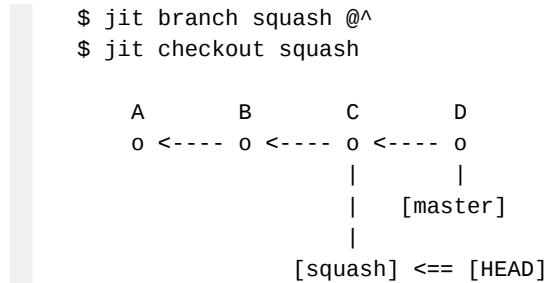
We have already seen that a combination of `reset` and `commit` can be used to squash the last few commits on a branch⁴. The `squash` command in `rebase` works very similarly, it just lets you deal with commits that are deeper in the history, so a little extra work is needed. Let's say we have the following history and want to squash commit *C* into *B*.

Figure 24.18. History before squashing



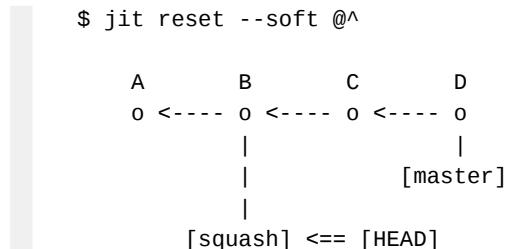
What this means is that we want commits *B* and *C* to be effectively replaced by a single commit containing the same tree as *C*, with the rest of the history following on after. So first, we need to get the index into the state of commit *C*, and we'll do that by checking out a temporary branch at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

Figure 24.19. Checking out the desired tree



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



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.

⁴Section 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]
```

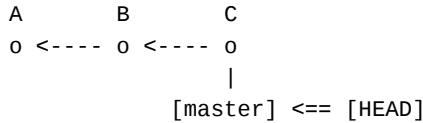
Git's squash command actually combines the messages of *B* and *C* when creating the *B+C* commit, but I'll leave that as an exercise for the reader. The above illustrates what is happening at the level of the history and the contents of commits.

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.

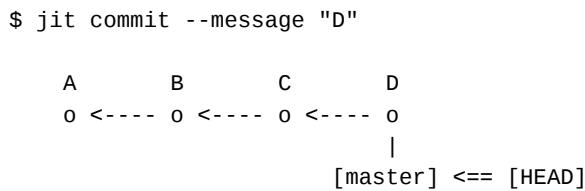
This workflow combines a reordering with the squash technique we've just seen. Let's start with our familiar linear history:

Figure 24.24. History before a fix-up



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* that contains the amendments we'd like to apply to *B*.

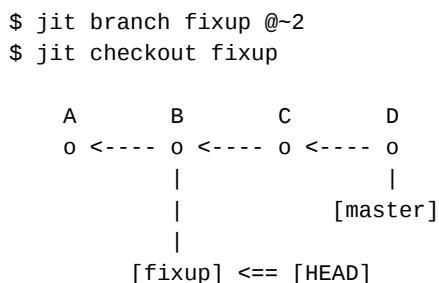
Figure 24.25. Creating a fix-up commit



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*.

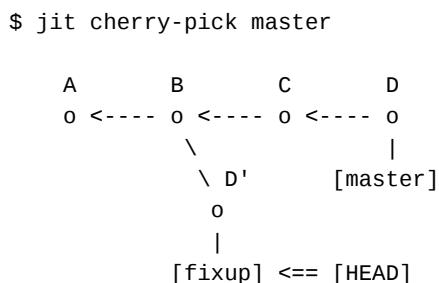
Let's start a new branch at *B*:

Figure 24.26. Starting a fix-up branch



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



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

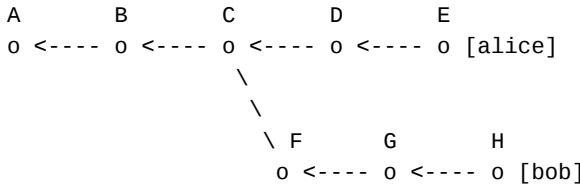
A      B      C      D
o <---- o <---- o <---- o
\     \
\     \ D'      [ORIG_HEAD]
\     o
\
\ B+D'  C'
o <---- o
|
[master] <== [HEAD]
```

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

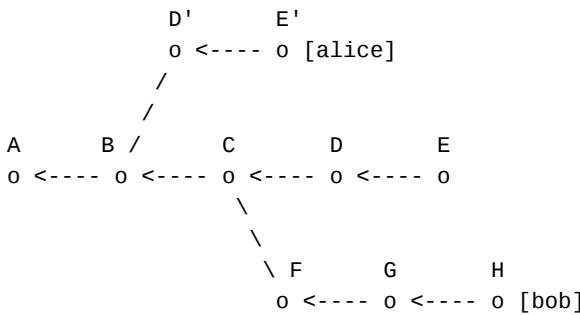
It's certainly useful to be able to edit the history of your branch, but it becomes a problem once you've shared your history with other teammates. As an example, let's imagine two coworkers, Alice and Bob, that are each working on their own branch of a project. The last commit each developer has in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

Figure 24.30. History with concurrent edits



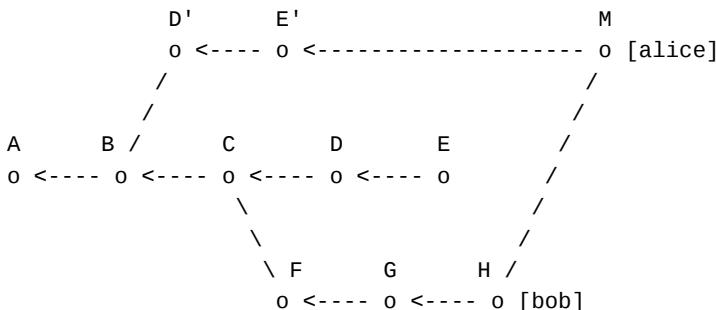
Now, suppose Alice decides *C* should not have been committed, and wants it removed from the history. She resets to *B* and then cherry-picks *D* and *E* to construct the history she actually wants.

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!

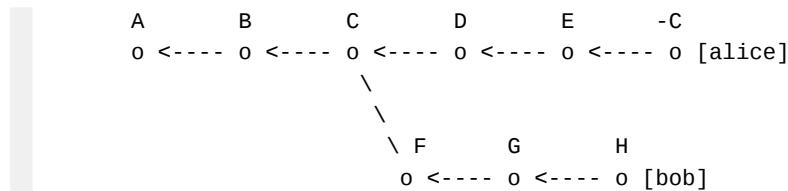
It's important to note at this point that Git is not doing anything wrong here. Git is merely a tool for tracking and reconciling concurrent edits to a set of files, and if the history says that *C* is a concurrent edit on one branch, rather than a commit shared by both branches, then that's how Git 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.

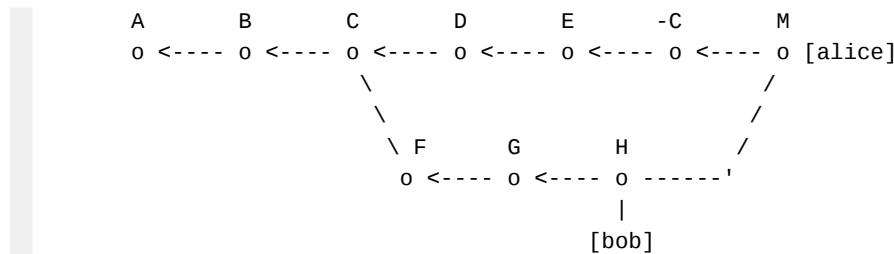
Let's now imagine that Alice does this. She writes a commit, which we'll call $-C$, that removes the content that was added in C .

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



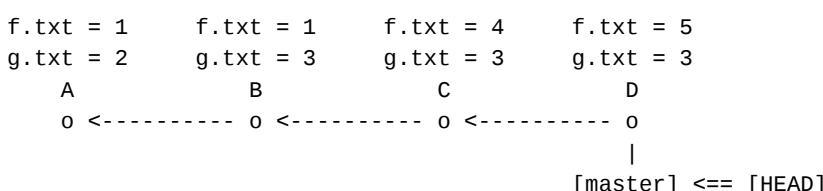
There are many reasons you'd want to make a change like this. Maybe some temporary code was added to facilitate a long refactoring, and can now be removed. Maybe a feature flag was used to incrementally roll a feature out, and is no longer necessary. Maybe you added someone's public key to your config 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.

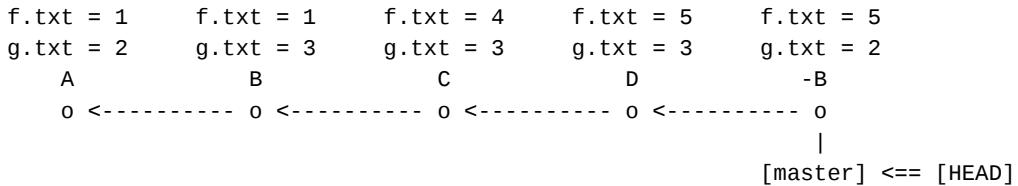
Consider the following history in which each commit modifies one of two files.

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 $d_{AB} = \{ g.txt \Rightarrow [2, 3] \}$, and inverting it to get $\{ g.txt \Rightarrow [3, 2] \}$. 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 $d_{BA} = \{ g.txt \Rightarrow [3, 2] \}$ and $d_{BD} = \{ f.txt \Rightarrow [1, 5] \}$. These do not overlap and so the end result is $T_{-B} = T_D + d_{BA} = \{ f.txt \Rightarrow 5, g.txt \Rightarrow 2 \}$.

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
  end
end

```

```
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

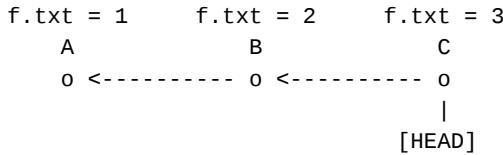
    include Sequencing
    include WriteCommit

    private

    def merge_type
      :revert
    end
  
```

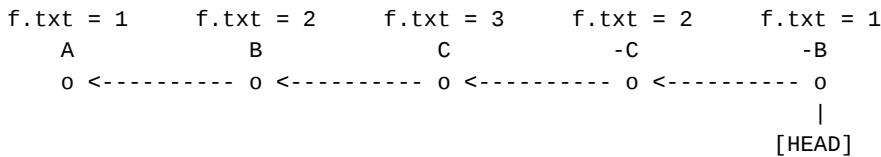
First, let's tackle storing the commit sequence. `cherry-pick` iterates over the input range from the oldest commit forwards, so the changes are applied in the same order they were originally. Since the revert commit reverses the changes, they need to be done in the reverse order from how they were originally 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 $d_{BA} = \{ f.txt \Rightarrow [2, 1] \}$ and $d_{CB} = \{ f.txt \Rightarrow [3, 2] \}$. 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
```

```
    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 repository⁵.

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

⁵Chapter 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 'git commit' to conclude merge"
    else
      puts "You have unmerged paths."
      hint "fix conflicts and run 'git commit'"
      hint "use 'git 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 'git #{op} --continue'"
  else
    hint "fix conflicts and run 'git #{op} --continue'"
  end
  hint "use 'git #{op} --abort' to cancel the #{op} operation"
  puts ""
end

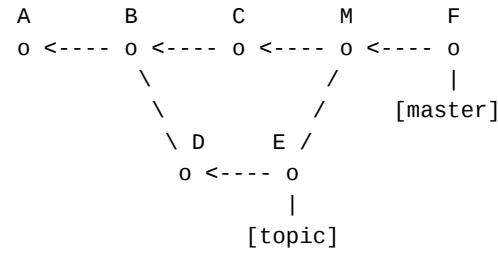
def hint(message)
  puts "  (#{message})"
end
```

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.

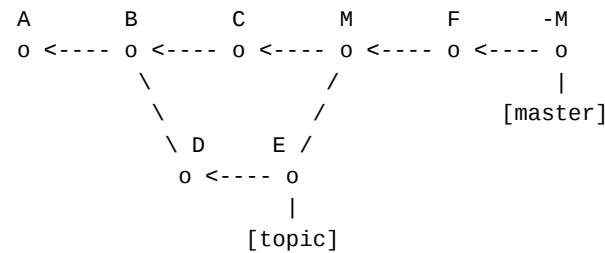
In the case of revert, this means the command can be used to effectively undo a merge. Suppose we have the following history in which *M* is a merge commit that was generated by running `git merge topic` while `master` was checked out at *C*.

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



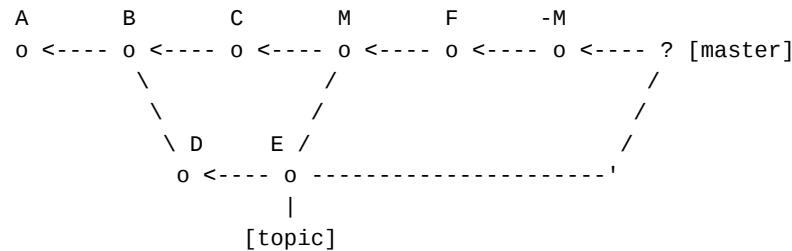
But what is the effect of *M*, and what does that mean for the tree T_{-M} ? Well, `revert` will perform a merge between the current `HEAD` (*F*) and the given commit's first parent (*C*), with the commit *M* as the base. So $T_{-M} = T_M + d_{MF} + d_{MC}$, the tree of *M* plus the difference from each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

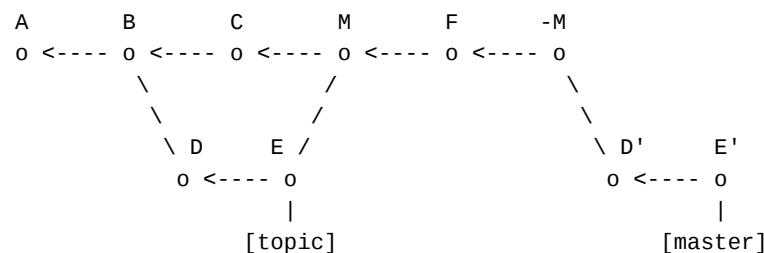
Finally, we saw in the previous chapter that the `cherry-pick` command does not create a parent link between the new commit and the one it's derived from, as that would prevent a future true merge from including the whole branch's history. A similar problem occurs if we revert a merge and then later 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



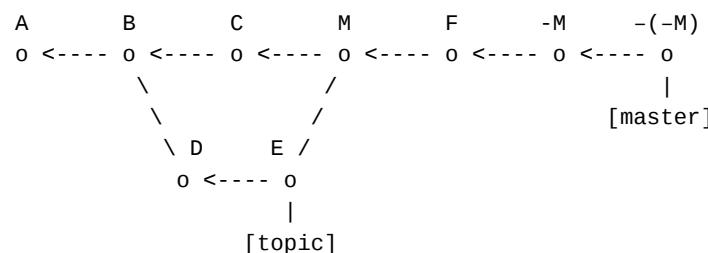
The base of this merge would be the common ancestor of $-M$ and E , which is E itself — E is an ancestor of $-M$. So this merge attempt ends up doing nothing. To get the changes from the topic branch back, we can either cherry-pick its commits using a command like `cherry-pick @~3..topic`:

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

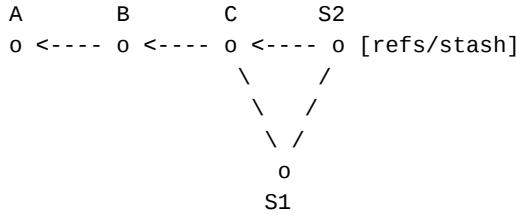
One last application of commits and cherry-picking before we move on. Git includes a command call `stash`⁶, which lets you store away uncommitted changes and retrieve them later. We won't implement this command, but it turns out it can easily be simulated with our existing tools.

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 `S1` and `S2` will be created, but `HEAD` will remain pointing at `C` and the index and workspace will match it.

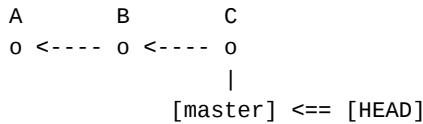
Figure 24.44. Stored pair of stash commits



`S1` has `C` as its parent and has a message like `index on <branch>: <ID> <message>`, while `S2` has both `C` and `S1` 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

graph TD
    A((A)) --> B((B))
    B --> C((C))
    C --> master["[master]"]
    C --> stash["[stash] <== [HEAD]"]
    
```

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"

graph TD
    A((A)) --> B((B))
    B --> master["[master]"]
    master --> S1["S1<br>[stash] <== [HEAD]"]
    
```

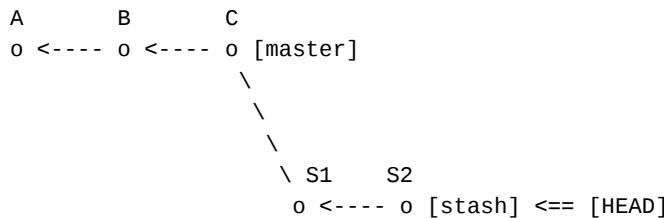
Next we need to put any unstaged changes into the index and make a second commit. We can't just use `add .` as that would include untracked files. Instead we need to select the files that have been modified and add each one. The following command does just that:

```
$ 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 matches⁷, and `cut -c 4-` selects the fourth to the last characters of each line⁸, 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"
```



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.

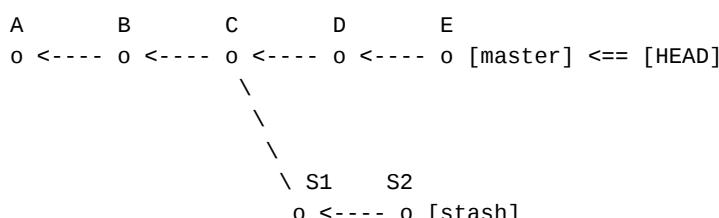
Figure 24.49. Checking out the original branch

```
$ 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

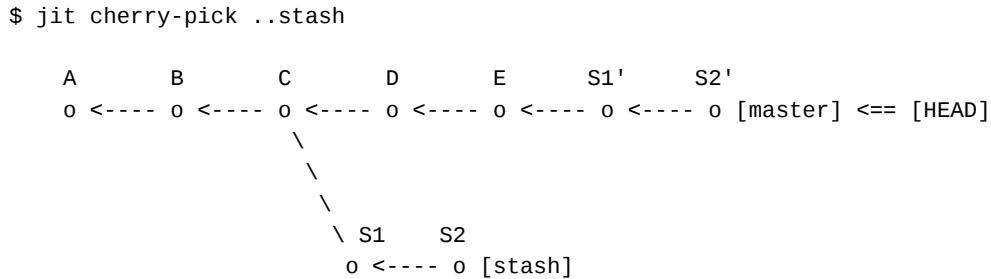


⁷<https://manpages.ubuntu.com/manpages/bionic/en/man1/grep.1.html>

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 .

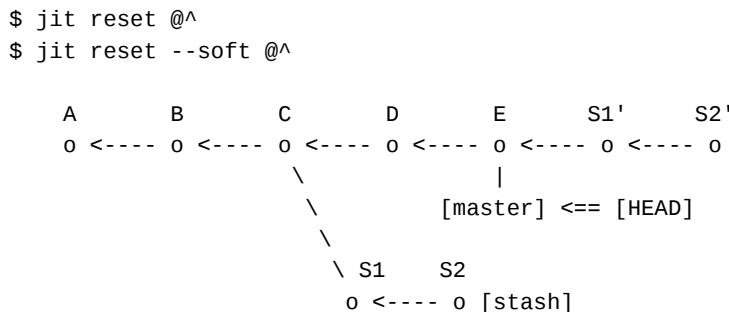
Figure 24.51. Cherry-picking the stash commits



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



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

Part III. Distribution

25. Configuration

So far, I have been aiming for a *minimal* implementation of Git, one that demonstrates the core concepts of the most common commands and avoids optional or duplicated behaviour. For example, in the last chapter we discovered that rebase can be built out of the reset, commit and cherry-pick commands, and so there's no need to implement rebase itself.

In light of that, it may seem desirable to avoid Git's configuration system; configuration usually adds inessential complexity to a program and involves many annoying parsing problems. However, as we approach the fetch and push commands for sharing our commits with our teammates, we'll discover that Git uses its config files to store information about remote repositories, and so before we can push our work to others, we'll need to add the config command.

25.1. The Git config format

Git's configuration file format is, like many such formats, designed to be read and edited both by programs and by end users, and so it is human-readable and easy to inspect. The config command sets and retrieves values from `.git/config`, and we can use it to see what types of data the format supports and how it represents them.

First, let's add a few common variables to our config. The `core.editor` setting tells Git which command to run whenever it wants to launch your editor, and `user.{name, email}` are used to populate the author and committer headers on commits. Using cat to read the config file, we can see that these variables have been organised into sections, where each section contains a key/value list of the variables within it.

```
$ git config core.editor vim
$ git config user.name "James Coglan"
$ git config user.email "james@jcoglan.com"

$ cat .git/config
[core]
    editor = vim
[user]
    name = James Coglan
    email = james@jcoglan.com
```

All the variables we've set have a name that consists of two parts: a *section* name like `core` or `user`, followed by a *variable* like `editor` or `name`. A variable name can also contain a third element, a *subsection*, between the section and variable. For example, the URL of a remote repository is stored under the variable `remote.<name>.url` where `<name>` is the name of the remote.

```
$ git config remote.origin.url "ssh://git@github.com/jcoglan/jit.git"

$ cat .git/config
[remote "origin"]
    url = ssh://git@github.com/jcoglan/jit.git
```

Subsections are represented just like regular sections, with their name between square brackets, but the subsection name appears after the section name in double quotes.

The section and variable parts of a name are case-insensitive; we can replace `remote` or `url` in the above name with its uppercase equivalent and get the same value out.

```
$ git config REMOTE.origin.URL  
ssh://git@github.com/jcoglan/jit.git
```

However, subsection names are case-sensitive; replacing `origin` with its uppercase does not match this variable.

```
$ git config remote.ORIGIN.url  
(no output)
```

If we set a variable that already exists in the file, then the new value replaces the old one.

```
$ git config core.editor atom  
  
$ cat .git/config  
[core]  
    editor = atom
```

However, it is possible to set multiple values for the same variable by using the `--add` option. If you have a variable whose value should be an array, this is how you'd represent it.

```
$ git config --add core.editor emacs  
  
$ cat .git/config  
[core]  
    editor = atom  
    editor = emacs
```

The `config` command returns the last value for a variable, unless the `--get-all` option is used, in which case it returns all the stored values.

```
$ git config core.editor  
emacs  
  
$ git config --get-all core.editor  
atom  
emacs
```

When a variable has multiple values, trying to set a new value for it will raise an error. To remove the existing values and set a single new value, we can use the `--replace-all` switch.

```
$ git config core.editor vim  
warning: core.editor has multiple values  
error: cannot overwrite multiple values with a single value  
  
$ git config --replace-all core.editor vim  
  
$ cat .git/config  
[core]  
    editor = vim
```

25.1.1. Whitespace and comments

If the `config` file could only be edited via the command-line, then our task would be much simpler: just read the file from disk, parse it into a data structure, modify that structure, and write

it back to disk. However, the config format also supports being modified directly by the end user via their text editor, and this makes it substantially more challenging to programmatically modify.

As well as the data representation generated by the `config` command, the config file may contain arbitrary whitespace and comments. Blank lines may be inserted anywhere in the file and should be ignored by the parser. The `config` command writes variables with a tab at the beginning of the line, but the parser accepts any whitespace characters in this position. Any line can be suffixed with a comment, which begins with `#` or `;` and continues until the end of the current line.

For example, I'll edit my config file so that it contains a few comments and blank lines:

```
$ cat .git/config

# This is my git config. There are many
# like it but this one is mine.

[core] # general settings
      editor = vim

[user]
      # personal information
      name = James Coglan
      email = james@jcoglan.com # do not change

[remote "origin"]
      url = ssh://git@github.com/jcoglan/jit.git
```

Now, setting the `core.editor` variable replaces the existing value, but the formatting of the rest of the file is left completely intact. Even though the parser ignores whitespace and comments, it does not throw them away when changing the file's contents.

```
$ git config core.editor emacs
$ cat .git/config

# This is my git config. There are many
# like it but this one is mine.

[core] # general settings
      editor = emacs

[user]
      # personal information
      name = James Coglan
      email = james@jcoglan.com # do not change

[remote "origin"]
      url = ssh://git@github.com/jcoglan/jit.git
```

Likewise, comments within sections are retained. For example, if we change the `user.name` setting, then the comment preceding it is kept in the file.

```
$ git config user.name "A. U. Thor"
$ cat .git/config
```

```
# This is my git config. There are many
# like it but this one is mine.

[core] # general settings
editor = emacs

[user]
    # personal information
    name = A. U. Thor
    email = james@jcoglan.com # do not change

[remote "origin"]
    url = ssh://git@github.com/jcoglan/jit.git
```

A comment on the same line as a variable is only removed if that variable is modified; the config command completely replaces the line for that variable. For example, changing user.email removes the comment that appeared after the old value.

```
$ git config user.email "me@example.com"
$ cat .git/config

# This is my git config. There are many
# like it but this one is mine.

[core] # general settings
editor = emacs

[user]
    # personal information
    name = A. U. Thor
    email = me@example.com

[remote "origin"]
    url = ssh://git@github.com/jcoglan/jit.git
```

One final complication is that the config file format allows values to be broken over multiple lines, by putting a backslash before the line break. For example, it's legal to store the user.name value in the above config as:

```
[user]
    name = A. U. \
        Thor
```

The escaped line break is removed from the value by the parser, but this formatting should be preserved if another value is edited.

25.1.2. Abstract and concrete representation

This requirement to maintain the format of a human-edited file while programmatically modifying it radically changes how we go about parsing the config format. If we had no need to support human editing, then we could parse the above file into an *abstract* structure that removes any extraneous details that don't contribute to the values of variables. For example, we might construct a nested structure that maps each section or subsection name to its constituent variables.

```
{
```

```
[ "core" ] => {
    "editor" => [ "emacs" ]
},
[ "user" ] => {
    "name"  => [ "A. U. Thor" ],
    "email" => [ "me@example.com" ]
},
[ "remote", "origin" ] => {
    "url"  => [ "ssh://git@github.com/jcoglan/jit.git" ]
}
```

However, since we've thrown away all the whitespace and comment information, when we serialise this structure back to disk we'll lose it all:

```
$ cat .git/config
[core]
    editor = emacs
[user]
    name = A. U. Thor
    email = me@example.com
[remote "origin"]
    url = ssh://git@github.com/jcoglan/jit.git
```

To preserve as much of the original text as possible, a better approach is to parse it into a *concrete* structure that directly represents lines of text, but annotates each one with the machine-readable data it contains. For example, we could represent the file as a sequence of `Line` structures, which retain the text of each line, plus a few annotations. `section` tells us the section the line belongs to, and `variable` stores any variable and value the line contains. Here's the `core` section represented this way:

```
Line(text = "[core] # general settings\n",
     section = Section(name = ["core"]),
     variable = nil)

Line(text = "\teditor = emacs\n",
     section = Section(name = ["core"]),
     variable = Variable(name = "editor", value = "emacs"))

Line(text = "\n",
     section = Section(name = ["core"]),
     variable = nil)
```

It will still be useful to index the lines by section so we can quickly locate variables and add new ones to the end of a section, so let's flesh out our structure into a map from section names to arrays of lines. In the interests of brevity I have not included every field in the `Line` structures below; all lines in the same array have the same `section` value, and any fields with no value have been elided.

```
{
  [ "core" ] => [
    Line(text = "[core] # general settings\n", ...),
    Line(text = "\teditor = emacs\n",
         variable = Variable(name = "editor", value = "emacs"),
         ...),
    Line(text = "\n", ...)
```

```

},
["user"] => [
  Line(text = "[user]\n", ...),
  Line(text = "\t# personal information\n", ...),
  Line(text = "\tname = A. U. Thor\n",
    variable = Variable(name = "name", value = "A. U. Thor"),
    ...),
  Line(text = "\temail = me@example.com\n",
    variable = Variable(name = "email", value = "me@example.com"),
    ...),
  Line(text = "\n", ...)
],
["remote", "origin"] => [
  Line(text = "[remote \"origin\"]\n", ...),
  Line(text = "\turl = ssh://git@github.com/jcoglan/jit.git\n",
    variable = Variable(name = "url", value = "ssh://git@github.com/..."),
    ...)
]
}

```

Lastly, we need to preserve the lines at the beginning of the file that do not belong to a section. These can be grouped into a list with an empty key [], where every `Line` has `variable = nil`.

```

[] => [
  Line(text = "# This is my git config. There are many\n", ...),
  Line(text = "# like it but this one is mine.\n", ...),
  Line(text = "\n", ...)
]
```

These representations of the configuration file correspond to the concepts of *abstract syntax trees*¹ and *concrete syntax trees*² found in parsing theory. An abstract syntax tree (AST) distills a text down the the minimal structure necessary to extract its meaning, whereas a concrete syntax tree (CST) retains enough information to reconstruct the original text. There are other ways we could pick to represent the CST above, but this line-oriented representation maps fairly nicely to how Git manipulates the file.

25.2. Modelling the .git/config file

To begin coding up an implementation for this format, according to the CST we designed above, let's define a few structures we'll need to represent the file's contents in memory. First, we'll define `Variable`, which has a name and a value, both of which are strings. Since variables are case-insensitive, we'll include a `Variable.normalize` method that converts a variable name to a canonical form by down-casing it. The variable's `name` field will retain the original casing seen in the file, but in order to find the right lines for a variable we'll need to compare the requested variable name using this canonical form. The `serialize` method takes care of turning the `Variable` into its form in the config file: a leading tab, followed by the name, a = sign, and the value.

```

# lib/config.rb

Variable = Struct.new( :name, :value) do
  def self.normalize(name)

```

¹https://en.wikipedia.org/wiki/Abstract_syntax_tree

²https://en.wikipedia.org/wiki/Parse_tree

```
    name&.downcase
  end

  def self.serialize(name, value)
    "\t#{ name } = #{ value }\n"
  end
end
```

Next up, we'll define `Section` as a structure with a `name`, which is an array that's either empty or contains a section name and some optional subsection names. The canonical way to represent these names is to convert the first element to lowercase, and join all remaining elements with dots.

```
# lib/config.rb

Section = Struct.new(:name) do
  def self.normalize(name)
    return [] if name.empty?
    [name.first.downcase, name.drop(1).join(".")]
  end

  def heading_line
    line = "[#{ name.first }"
    line.concat("%' "#{ name.drop(1).join(".") }%'") if name.size > 1
    line.concat("]\n")
  end
end
```

The last structure we need is `Line` which represents lines of text from the file, and annotates them with a `section` and an optional `variable`.

```
# lib/config.rb

Line = Struct.new(:text, :section, :variable) do
  def normal_variable
    Variable.normalize(variable&.name)
  end
end
```

With these structures defined, we can now take care of reading the file into memory and writing it back again.

25.2.1. Parsing the configuration

Before we can read or change any values in the config file, we need to read it into memory. Let's set up our `Config` class so that it takes a `Pathname` and sets up a `Lockfile` for that path. We'll need to lock the file when editing it so that we don't allow two processes to read the file, both change their in-memory copy, and then write the file, with one process overwriting the change the other made. The `@lines` variable will hold the structure we defined above that maps section names to arrays of `Line` values.

```
# lib/config.rb

class Config
  def initialize(path)
```

```
    @path      = path
    @lockfile = Lockfile.new(path)
    @lines    = nil
end

# ...
end
```

The Config class will support two ways to open the underlying file: `open` and `open_for_update`. The former reads the file into memory if we don't already have it loaded, while the latter acquires the lock and then reads the file. It's important to read the file after taking the lock so we can be sure no other process has modified the file before we copy it into memory.

```
# lib/config.rb

def open
  read_config_file unless @lines
end

def open_for_update
  @lockfile.hold_for_update
  read_config_file
end
```

`read_config_file` is where all the parsing logic takes place. We initialise `@lines` to an empty hash whose values are arrays. We also need to track when the current section is as we scan the file line by line, and this begins as the unnamed section, `Section.new([])`. We read each logical line from the file by calling `read_line`; this reads a single line from the file and checks if it ends with a backslash and a line break and continues reading the next line if so. We pass each line through `parse_line` to get a `Line` value, and update the `section` variable with that line's `section` value, before appending the line to the right section within `@lines`. If the file does not exist, we can silently exit.

```
# lib/config.rb

def read_config_file
  @lines = Hash.new { |hash, key| hash[key] = [] }
  section = Section.new([])

  File.open(@path, File::RDONLY) do |file|
    until file.eof?
      line = parse_line(section, read_line(file))
      section = line.section

      lines_for(section).push(line)
    end
  end
rescue Errno::ENOENT
end

def read_line(file)
  buffer = ""

  loop do
    buffer.concat(file.readline)
    return buffer unless buffer.end_with?("\\""\n")
```

```
end  
end  
  
def lines_for(section)  
  @lines[Section.normalize(section.name)]  
end
```

The `parse_line` method uses a set of regular expressions to match each line. These patterns are almost certainly unreadable at first glance so let's break each one down.

```
# lib/config.rb  
  
SECTION_LINE = /\A\s*\[((a-z0-9-]+)( "(.)")?\]\s*(\Z|#|;)/i  
VARIABLE_LINE = /\A\s*([a-z][a-z0-9-]*)\s*=\s*(.*?)\s*(\Z|#|;)/im  
BLANK_LINE = /\A\s*(\Z|#|;)/  
INTEGER = /\A-[1-9][0-9]*\Z/
```

These patterns use `\A` and `\Z` to match the start and end of the string. In Ruby regular expressions, `^` and `$` match the start and end of a *line*, so they would fail to match values that are split over multiple lines.

Several patterns end with `\s*(\Z|#|;)`, which means any number of spaces followed by either the end of the string (`\Z`), a `#` or a `;`. This pattern matches any trailing space and comments at the end of the line; anything after this can be ignored. Similarly all the line patterns begin with `\A\s*` which matches any whitespace at the start of the string.

`SECTION_LINE` then matches a section name, `[a-z0-9-]+`, meaning one or more alphanumeric characters, optionally followed by a quoted subsection name, all inside square brackets. `VARIABLE_LINE` matches a variable name, `[a-z][a-z0-9-]*`, followed by a `=` sign padded with any amount of space, followed by the value `.*?`, which non-greedily matches any characters until we find trailing whitespace or a comment. `BLANK_LINE` matches only whitespace and comments.

`parse_line` uses these patterns to classify each line as either a section heading, a variable, or a blank line. It takes the current `Section` and the next line, and if it matches `SECTION_LINE` we create a new `Section` and return a `Line` annotated with it; this becomes the `section` argument for the next call. If the line matches `VARIABLE_LINE`, then we construct a `Variable` with the name and value, and use this and the current `Section` to build the `Line`. Finally if the line matches `BLANK_LINE` then we return a `Line` with no variable. Blank lines are considered to be part of the section under whose heading they appear.

```
# lib/config.rb  
  
ParseError = Class.new(StandardError)  
  
def parse_line(section, line)  
  if match = SECTION_LINE.match(line)  
    section = Section.new([match[1], match[3]].compact)  
    Line.new(line, section)  
  elsif match = VARIABLE_LINE.match(line)  
    variable = Variable.new(match[1], parse_value(match[2]))  
    Line.new(line, section, variable)  
  elsif match = BLANK_LINE.match(line)  
    Line.new(line, section, nil)  
  end  
end
```

```
    else
      message = "bad config line #{ line_count + 1 } in file #{ @path }"
      raise ParseError, message
    end
  end

  def line_count
    @lines.each_value.reduce(0) { |n, lines| n + lines.size }
  end
```

`parse_value` does a little conversion of the string extracted by `VARIABLE_LINE`, if the string matches various patterns. The values `yes`, `on` and `true` are interpreted as the boolean value `true`, while `no`, `off` and `false` denote the opposite. If the string matches the `INTEGER` pattern then it's converted to an integer. Otherwise, the value remains a string, but we remove any escaped line breaks from it by replacing the pattern `/\\n/` with the empty string.

```
# lib/config.rb

def parse_value(value)
  case value
  when "yes", "on", "true" then true
  when "no", "off", "false" then false
  when INTEGER              then value.to_i
  else
    value.gsub(/\\\n/, "")
  end
end
```

Writing the file back to disk is much more straightforward: since we retain the text of each line, all we need to do is write all the lines back to the file and commit it.

```
# lib/config.rb

def save
  @lines.each do |section, lines|
    lines.each { |line| @lockfile.write(line.text) }
  end
  @lockfile.commit
end
```

25.2.2. Manipulating the settings

Now that we have a representation of the file in memory, we can layer an interface over it to support the operations we need. The methods in this interface will largely work by finding matching sections and lines in the file and then manipulating them in some way. To this end, we'll define two helper functions. `split_key` divides a variable name into its section part and the variable part, for example the key `["remote", "origin", "url"]` becomes `[["remote", "origin"], "url"]`. `find_lines` takes these two values and returns the section and an array of `Line` values matching the requested key. Remember that since Git allows multiple values for the same variable, more than one `Line` may be returned.

```
# lib/config.rb

def split_key(key)
  key = key.map(&:to_s)
```

```
var = key.pop

[key, var]
end

def find_lines(key, var)
  name = Section.normalize(key)
  return [nil, []] unless @lines.has_key?(name)

  lines = @lines[name]
  section = lines.first.section
  normal = Variable.normalize(var)

  lines = lines.select { |l| normal == l.normal_variable }
  [section, lines]
end
```

The simplest operations to define on top of these helpers are the *getter* methods `Config#get` and `Config#get_all`. The key argument to all these methods will be an array representing the variable name, such as `["core", "editor"]` or `["remote", "origin", "url"]`. `get_all` finds all the matching lines and returns their values, and `get` calls `get_all` and returns its last result.

```
# lib/config.rb

def get_all(key)
  key, var = split_key(key)
  _, lines = find_lines(key, var)

  lines.map { |line| line.variable.value }
end

def get(key)
  get_all(key).last
end
```

Next, we'll define a way to add a new value for a variable, that is the function of the `--add` switch to the `config` command. `Config#add` takes a key and a value, and selects the section that the given key should be added to — this will be `nil` if the section does not yet exist. It then calls a helper method called `add_variable` to perform the addition.

```
# lib/config.rb

def add(key, value)
  key, var = split_key(key)
  section, _ = find_lines(key, var)

  add_variable(section, key, var, value)
end
```

`add_variable` takes care of adding a new value to the file, and it will be used by other commands in the interface. If its `section` argument is `nil`, it calls `add_section` to create a new `Section` and a `Line` for its heading, and stores that `Line` in the `@lines` structure. The `Line` that `add_variable` creates contains a `Variable` and is appended to the existing set of lines for the given section, rather than replacing any of them.

```
# lib/config.rb
```

```

def add_variable(section, key, var, value)
    section ||= add_section(key)

    text = Variable.serialize(var, value)
    var  = Variable.new(var, value)
    line = Line.new(text, section, var)

    lines_for(section).push(line)
end

def add_section(key)
    section = Section.new(key)
    line    = Line.new(section.heading_line, section)

    lines_for(section).push(line)
    section
end

```

Next, we'll deal with the `Config#set` method that corresponds to calling `config` with two arguments to set a variable's value. This method uses `find_lines` to get both the section and the lines corresponding to the given key. If there are no existing lines, we can use `add_variable` to insert one; if there is a single line then we can replace its text to reflect the new value; otherwise we're trying to set a multi-valued variable and this should generate an error.

```

# lib/config.rb

Conflict = Class.new(StandardError)

def set(key, value)
    key, var      = split_key(key)
    section, lines = find_lines(key, var)

    case lines.size
    when 0 then add_variable(section, key, var, value)
    when 1 then update_variable(lines.first, var, value)
    else
        message = "cannot overwrite multiple values with a single value"
        raise Conflict, message
    end
end

def update_variable(line, var, value)
    line.variable.value = value
    line.text = Variable.serialize(var, value)
end

```

If we want to replace a multi-valued variable then we must use `Config#replace_all`, which corresponds to the `--replace-all` switch. This removes all the lines matching the given key, and then inserts a new one using `add_variable`.

```

# lib/config.rb

def replace_all(key, value)
    key, var      = split_key(key)
    section, lines = find_lines(key, var)

```

```
remove_all(section, lines)
add_variable(section, key, var, value)
end

def remove_all(section, lines)
  lines.each { |line| lines_for(section).delete(line) }
end
```

As well as setting new values, we can remove existing variables from the file. `Config#unset_all` corresponds to the `--unset-all` option to the `config` command. It works similarly to `replace_all`, removing all existing lines for the given key if the section exists. If a block is given, it yields to the block, which can perform additional checks on the existing lines before allowing them to be deleted. After removing the matching lines, we check whether we can remove the entire section; if there's only one line left in it, that will be the section's heading and we can remove it. The `remove_section` method matches the `--remove-section` switch to the `config` command.

```
# lib/config.rb

def unset_all(key)
  key, var      = split_key(key)
  section, lines = find_lines(key, var)

  return unless section
  yield lines if block_given?

  remove_all(section, lines)
  lines = lines_for(section)
  remove_section(key) if lines.size == 1
end

def remove_section(key)
  key = Section.normalize(key)
  @lines.delete(key) ? true : false
end
```

The `Config#unset` method, corresponding to the `--unset` switch, is a specialisation of `unset_all`; it calls `unset_all` with a block that raises an error if there is more than one line for the given key.

```
# lib/config.rb

def unset(key)
  unset_all(key) do |lines|
    raise Conflict, "#{key} has multiple values" if lines.size > 1
  end
end
```

Finally, the upcoming functionality will require a couple of methods for inspecting the sections within the configuration. These methods don't correspond to anything in the `config` command, but they provide functionality we'll need for the `remote` command. `Config#subsections` returns all the subsections that exist for a given section name. In our example, `subsections("remote")` would return `["origin"]`, since the subsection `remote.origin` exists. `Config#section?` queries whether a given section exists, which can be used to validate whether a named remote is configured.

```
# lib/config.rb

def subsections(name)
  name, _ = Section.normalize([name])
  sections = []

  @lines.each_key do |main, sub|
    sections.push(sub) if main == name and sub != ""
  end

  sections
end

def section?(key)
  key = Section.normalize(key)
  @lines.has_key?(key)
end
```

That completes the `Config` interface. The Jit repository contains an implementation of the `config` command, which is mostly boilerplate code for recognising the various options and calling into this interface, and we won't examine it in detail here.

25.2.3. The configuration stack

We've been focussing on one file here — the `.git/config` file which the `config` command uses by default. But there's more to it than that. Rather than having to copy all your common settings into every project, it would be nice define them once in a global location. For example, your `user` variables are probably the same for every project, whereas your `remote` settings are not — it makes sense to store the `remote config` within a project, but store `user` settings globally.

Therefore, Git supports not just a single configuration file, but a stack of them. The `.git/config` file within your repository has the highest precedence, but if we ask for a variable that doesn't exist in that file, Git will look in `~/.gitconfig`, that is a file stored in your home directory. Finally it will defer to the file `/etc/gitconfig`, which can define Git settings for every user on the system.

These files are referred to as the *local*, *global* and *system* config files respectively, and you can query them individually using the `--local`, `--global` and `--system` flags to the `config` command. If you don't specify a file, Git will look in all of them; `--get-all` will return all the values it finds in all the files, while a regular `get` will return the last value in the highest-precedence file. When setting a variable, only one of the files will be affected, and if no switch is used to specify which one, the local config (`.git/config`) will be changed.

Let's represent this stack of config files by creating a class that takes the path to a `.git` directory, and sets up internal references for the local, global and system config files.

```
# lib/config/stack.rb

class Config
  class Stack

    GLOBAL_CONFIG = File.expand_path("~/gitconfig")
    SYSTEM_CONFIG = "/etc/gitconfig"
```

```
def initialize(git_path)
  @configs = {
    :local => Config.new(git_path.join("config")),
    :global => Config.new(Pathname.new(GLOBAL_CONFIG)),
    :system => Config.new(Pathname.new(SYSTEM_CONFIG))
  }
end

# ...

end
end
```

For retrieving values from the stack, we can treat it as though it's a single `Config` object by implementing some of the same methods on it. We'll implement `open` to load the files into memory, and `get` and `get_all` to search for variables. `open` simply opens all the `Config` objects, while `get_all` calls `get_all` on each `Config` object and combines the results in precedence order, allowing `get` to pick the last one.

```
# lib/config/stack.rb

def open
  @configs.each_value(&:open)
end

def get(key)
  get_all(key).last
end

def get_all(key)
  [:system, :global, :local].flat_map do |name|
    @configs[name].open
    @configs[name].get_all(key)
  end
end
```

This is an instance of the *composite pattern*³ wherein a collection of objects is made to look the same as a single instance from the caller's point of view. It saves the caller the work of iterating over the collection, and of needing to know about the collection's internal precedence rules.

For setting and removing variables however, we ought to pick a particular file to modify, and so we don't expose those methods on the `Stack` class. Instead we'll expose a `file` method for selecting one of them. If the `name` argument is not a key in the `@configs` hash, it's interpreted as a pathname and used to open an arbitrary file; this corresponds to Git's `--file` option for the `config` command.

```
# lib/config/stack.rb

def file(name)
  if @configs.has_key?(name)
    @configs[name]
  else
    Config.new(Pathname.new(name))
  end
end
```

³https://en.wikipedia.org/wiki/Composite_pattern

```
end
```

Finally, we can expose this config system to the rest of the codebase by adding a method to the `Repository` class that initialises the stack for us.

```
# lib/repository.rb

def config
  @config ||= Config::Stack.new(@git_path)
end
```

25.3. Applications

We'll end this chapter by looking at a few examples of how configuration is used in Git. We'll start by improving the configuration mechanism for a couple of existing features, then something a little more complicated where we change the output of an existing command. Finally we'll see how Git uses the config format for storing other pieces of state.

25.3.1. Launching the editor

One of the simplest things to change is the way in which we detect the name of the user's editor. This is currently done using environment variables, which can be a little annoying to configure compared to storing your config in a file. Therefore, let's add support for the `core.editor` variable to the `editor_command` method:

```
# lib/command/base.rb

def editor_command
  core_editor = repo.config.get(["core", "editor"])
  @env["GIT_EDITOR"] || core_editor || @env["VISUAL"] || @env["EDITOR"]
end
```

The precedence here probably looks a little odd but when your program has multiple ways to provide the same input, it's worth considering their precedence. Command-line arguments tend to be given highest precedence, since they differ for every invocation of the program. Environment variables come next, since you can set them on every invocation, as in:

```
$ EDITOR=vim jit commit
```

Or you can use them more implicitly by exporting a variable and having it exist for every subsequent command you run:

```
$ export EDITOR=vim
$ jit commit
```

Either way, environment variables can be set differently for each shell you have open, and are a useful way to vary program behaviour between different projects. Configuration stored in files usually has lowest precedence, since a file will look the same to every program on the machine. As such, environment variables can be used to override file-stored settings on a per-shell basis, and command-line arguments override both of them on a per-command basis.

In Git, the file-stored `core.editor` setting has lower precedence than the `GIT_EDITOR` environment variable, for the above reasons. But it has higher precedence than the `VISUAL` and

EDITOR variables, since these are generic settings that many programs use and are not specific to Git. Preferring core.editor over them lets you set an editor just for Git that's different from the one you'd run by default.

25.3.2. Setting user details

One piece of configuration that's essential for making commits is to provide Git with your name and email address so it can populate the author and committer fields. We'll continue to support the GIT_AUTHOR_{NAME, EMAIL} variables, but add in support for the file-stored user.{name, email} settings. You'd typically store these in your global ~/.gitconfig file and only override them within a project if you need to use a different name and address.

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

def current_author
  config_name = repo.config.get(["user", "name"])
  config_email = repo.config.get(["user", "email"])

  name = @env.fetch("GIT_AUTHOR_NAME", config_name)
  email = @env.fetch("GIT_AUTHOR_EMAIL", config_email)

  Database::Author.new(name, email, Time.now)
end
```

Again, environment variables take precedence over file-stored settings so they can be overridden in each shell or on each invocation.

25.3.3. Changing diff formatting

Now for something a little more complicated. Git allows the formatting of some commands' output to be changed, for example it lets you change how diffs are formatted using the color.diff.* settings. If you'd prefer yellow and cyan over red and green for showing old and new content, you can set these variables:

```
$ git config color.diff.old yellow
$ git config color.diff.new cyan
```

There are a few variables that affect the content our diff command prints: context sets the style for the unchanged context lines and the ends of each hunk, meta styles the file name and version header information, frag styles the hunk headers that contain line offsets, and old and new set the style for removed and inserted content. Git colour settings support most common terminal formatting sequences, and we can expand our list of SGR codes to include them:

```
# lib/color.rb

SGR_CODES = {
  "normal"  => 0,
  "bold"    => 1,
  "dim"     => 2,
  "italic"  => 3,
  "ul"      => 4,
  "reverse" => 7,
  "strike"  => 9,
  "black"   => 30,
```

```
    "red"      => 31,
    "green"    => 32,
    "yellow"   => 33,
    "blue"     => 34,
    "magenta" => 35,
    "cyan"     => 36,
    "white"    => 37
}
```

These formats can be combined, for example you can set `color.diff.old` to "bold red". If you use two colours, the second one is used to set the background colour; whereas foreground colours begin at code 30, background codes begin at 40. So if we see a second colour in the input, we can add 10 to its code to set the background.

```
# lib/color.rb

def self.format(style, string)
  codes = [*style].map { |name| SGR_CODES.fetch(name.to_s) }
  color = false

  codes.each_with_index do |code, i|
    next unless code >= 30
    codes[i] += 10 if color
    color = true
  end

  "\e[#{ codes.join ";" }m#{ string }m"
end
```

In the `PrintDiff` module that's shared by the `diff` and `log` commands, we'll define the default styles for each type of content, and a method that takes a content type and a string, and uses `fmt` to style the string appropriately after checking the config for an override for the given style.

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

DIFF_FORMATS = {
  :context => :normal,
  :meta    => :bold,
  :frag    => :cyan,
  :old     => :red,
  :new     => :green
}

def diff_fmt(name, text)
  key   = ["color", "diff", name]
  style = repo.config.get(key)&.split(/ +/) || DIFF_FORMATS.fetch(name)

  fmt(style, text)
end
```

Having defined this hook to configure the styling, we can now replace the direct calls to `fmt` in the printing methods with calls to `diff_fmt`, and now we can style the diff output to whatever is most readable for us.

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

def header(string)
```

```
    puts diff_fmt(:meta, string)
  end

  def print_diff_hunk(hunk)
    puts diff_fmt(:frag, hunk.header)
    hunk.edits.each { |edit| print_diff_edit(edit) }
  end

  def print_diff_edit(edit)
    text = edit.to_s.rstrip

    case edit.type
    when :eql then puts diff_fmt(:context, text)
    when :ins then puts diff_fmt(:new, text)
    when :del then puts diff_fmt(:old, text)
    end
  end
```

25.3.4. Cherry-picking merge commits

Finally, let's look at an example of using the config format that's not for user-facing configuration per se. At the end of the last chapter, we examined the effect of reverting merge commits, and how our implementation always bases a cherry-pick or revert on the target commit's first parent. This may not always be what we want, and so these commands support an option to select which parent should be used when the input commit is a merge. The option is called `--mainline`, and it identifies which of the merge's parents the merge should be based on. For example, running `revert --mainline 2` bases the revert on the second parent, effectively undoing all the changes leading up to the first parent.

However, `cherry-pick` and `revert`, like `merge`, are interruptible commands. When a conflict happens, they exit to let the user fix it, and they can be resumed using the `--continue` flag. For this to work, all the state of the initial invocation must be preserved, and this includes any options that were used to control how changes are selected.

The sequencer infrastructure uses a config file stored at `.git/sequencer/opts` for this purpose. When the `--mainline` option is used, its value gets added as a variable called `options.mainline` inside that file:

```
# .git/sequencer/opts

[options]
  mainline = 2
```

This provides a way to retrieve the original options for the command when it's resumed. We can add support for this by setting up a `Config` object inside `Repository::Sequencer` when we set up the other paths.

```
# lib/repository/sequencer.rb

def initialize(repository)
  #
  # ...

  @abort_path = @pathname.join("abort-safety")
  @head_path = @pathname.join("head")
```

```

@todo_path  = @pathname.join("todo")
@config     = Config.new(@pathname.join("opts"))

#
end

```

Then, we'll make the `start` method accept an options hash and store all its fields in the `opts` config file. A new `get_option` method will let us retrieve the settings we've stored.

```

# lib/repository/sequencer.rb

def start(options)
#
# ...

@config.open_for_update
options.each { |key, value| @config.set(["options", key], value) }
@config.save

#
end

def get_option(name)
@config.open
@config.get(["options", name])
end

```

In the `Command::Sequencing` module, we can then define the `--mainline` option, and pass the options to `Sequencer#start` when we begin the cherry-pick/revert process.

```

# lib/command/shared/sequencing.rb

def define_options
#
# ...

@parser.on "-m <parent>", "--mainline=<parent>", Integer do |parent|
  @options[:mainline] = parent
end
end

def run
  case @options[:mode]
  when :continue then handle_continue
  when :abort     then handle_abort
  when :quit      then handle_quit
  end

  sequencer.start(@options)
  store_commit_sequence
  resume_sequencer
end

```

Now, having taken care of saving and retrieving the `--mainline` option, we can go ahead and use it. We'll introduce a `select_parent` method whose purpose is to pick one of the parents of a commit to use as the base of the merge (or the right-hand-side, if performing a revert). If the commit is a merge, then we require the `--mainline` option to be set, otherwise the command is ambiguous. If the commit is not a merge, then the option should not be set, as this would indicate the user expected to be picking a merge and may have selected the wrong commit.

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

def select_parent(commit)
  mainline = sequencer.get_option("mainline")

  if commit.merge?
    return commit.parents[mainline - 1] if mainline

    @stderr.puts <<~ERROR
      error: commit #{commit.oid} is a merge but no -m option was given
    ERROR
    exit 1
  else
    return commit.parent unless mainline

    @stderr.puts <<~ERROR
      error: mainline was specified but commit #{commit.oid} is not a merge
    ERROR
    exit 1
  end
end
```

All that remains is to integrate the `select_parent` method into the `pick_merge_inputs` and `revert_merge_inputs` methods for selecting the right commit to use in the merge. In `cherry-pick`, the parent is used as the base of the merge:

```
# lib/command/cherry_pick.rb

def pick_merge_inputs(commit)
  short = repo.database.short_oid(commit.oid)
  parent = select_parent(commit)

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

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

While in `revert`, it's used as the right-hand input while the base is the selected 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 = select_parent(commit)

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

These examples show how Git uses its configuration both for user-facing settings, and as an internal storage format for arbitrary key-value data. It's worth having a read through the list of Git's documented settings⁴ and picking some to try implementing yourself. For example, you could try implementing `commit.verbose`, which dumps a diff of the staged content as comments into the editor when composing a commit message. Or `merge.conflictStyle`, which if set to `diff3` makes the `Merge::Diff3` merge module include the base version in conflicted regions.

There's a lot to choose from, and they span a range from superficial presentational variations to deep behavioural changes. There's plenty of scope to test your understanding of the machinery we've developed by extending it in novel ways.

⁴https://git-scm.com/docs/git-config#_variables

26. Remote repositories

When Git was first becoming popular, one of its banner features was that it was *distributed*. Along with other *distributed version control systems* (DVCS) like Darcs¹ and Mercurial², Git led a move away from the widespread centralised model of Subversion³ and CVS⁴. Given that decentralisation is a core element of Git's design, it may seem surprising that we've not yet made any mention of it. Every command we've studied so far works on the data in your project and its `.git` directory, never touching the rest of your system, much less another computer. So where does distribution come in?

In a centralised system, there is a single copy of the repository — the equivalent of the `.git` directory — that resides on a server accessible to all team members. Each contributor has a local working copy of the project, and a little metadata, but they do not have a complete copy of the repository. If they want to commit anything, they need to push their changes to the server. If they want to do anything else, like inspect the history or perform a merge, they also need to talk to the server. The server is responsible for integrating everyone's changes and making sure conflicts are avoided or resolved. It might do this using a system of locks to ensure only one user at a time is allowed to change each file, or it might be able to detect when a submitted change is not based on the latest version and reject it.

DVCSs do not work like this. There is no special central server that everyone must submit changes to. Instead there are multiple complete copies of the repository, one on each developer's machine. The team can fetch changes from each other in any way they like, and they can work on their own local repository without a network connection, since their copy of the repository is fully independent.

Git is designed so that almost all interesting operations on the project history happen locally, in the `.git` directory of each contributor's copy. Making it distributed is reduced to the problem of copying commits between repositories, and Git's data model is designed to make it straightforward to do this efficiently and reliably. Once you've fetched commits from a collaborator, you can use all the tools we've seen so far to merge their history with your own.

Conceptually, Git's `fetch` and `push` commands essentially do two things: they copy missing objects from one repository to another, and they update pointers in the `.git/refs` directory. There is fair amount of detail in how exactly they do this, and it will take us the few remaining chapters to complete it. The important thing to remember for now is that Git achieves distribution by performing all history manipulation *locally*, and then making it easy to copy commits between repositories.

In this chapter we'll tackle a few of the building blocks necessary to build towards a working `fetch` command. We'll look at configuring remote repositories, take a look at `refsspecs`, and we'll extend the `RevList` class to find objects other than commits.

¹<http://darcs.net/>

²<https://www.mercurial-scm.org/>

³<https://subversion.apache.org/>

⁴<https://www.nongnu.org/cvs/>

26.1. Storing remote references

If you take a look inside the `.git/config` file in a repository you've cloned from GitHub, you'll probably see a section that looks like this:

```
[remote "origin"]
  url = git@github.com:jcoglan/jit.git
  fetch = +refs/heads/*:refs/remotes/origin/*
```

The `remote.origin.fetch` variable is a `refspec`. We'll learn more about those shortly, but for now it suffices to say that this variable describes how to copy references from the remote repository into our local one when we run `fetch`. It says that we'd like to copy every ref inside `refs/heads` from the remote repository into the directory `refs/remotes/origin` in our local copy. The leading plus sign means updates to our refs will be *forced*; they'll be overwritten even if the new commit IDs are not fast-forwards from the current values.

So, if the remote has a branch called `master` and we fetch from it, we should end up with a file at `.git/refs/remotes/origin/master` containing the remote's `master` branch pointer. The `fetch` command essentially makes a copy of the remote inside our own repository, retrieving any objects we don't yet have, and storing a copy of the remote branch pointers in `.git/refs/remotes`.

After this fetch, we can refer to the remote's `master` pointer by the name `origin/master` when using any command that takes a revision, such as `log` or `merge`. This is achieved by adding `.git/refs/remotes` to the list of paths searched by the `Refs` class when looking for named pointers.

```
# lib/refs.rb

REFS_DIR      = Pathname.new("refs")
HEADS_DIR     = REFS_DIR.join("heads")
REMOTES_DIR   = REFS_DIR.join("remotes")

def initialize(pathname)
  @pathname      = pathname
  @refs_path     = @pathname.join(REFS_DIR)
  @heads_path    = @pathname.join(HEADS_DIR)
  @remotes_path = @pathname.join(REMOTES_DIR)
end

def path_for_name(name)
  prefixes = [@pathname, @refs_path, @heads_path, @remotes_path]
  prefix   = prefixes.find { |path| File.file? path.join(name) }

  prefix ? prefix.join(name) : nil
end
```

We'd also like to strip this directory name off the beginning of any references inside it, for example the ref `refs/remotes/origin/master` should be displayed using the short name `origin/master`. We do this by adding `refs/remotes` to the list of paths used in `Refs#short_name`.

```
# lib/refs.rb
```

```
def short_name(path)
  path = @pathname.join(path)

  prefix = [@remotes_path, @heads_path, @pathname].find do |dir|
    path.dirname.ascend.any? { |parent| parent == dir }
  end

  path = path.relative_path_from(prefix) if prefix
  path.to_s
end
```

We've also put in a check so that if none of the prefixes match, the path is left as-is. This is because this method will be used to display the results of `fetch` and `push` commands, which let the user enter any refs of their choice, and they may enter invalid refs that don't match any of these directories.

We can now save remote references and refer to them easily without typing out their whole path.

26.2. The remote command

The snippet from `.git/config` that we saw above was generated using the `remote` command. Running the following command will cause Git to create a new remote in your configuration, with the given name and URL.

```
$ git remote add origin "git@github.com:jcoglan/jit.git"
```

That's all a remote is: a stored URL and a few other details. We use remotes because typing out the full URL and `refspec` every time we ran `fetch` would be tedious; it's much easier to type `fetch origin` and have Git look up the relevant stored parameters. Just like the `master` branch, the `origin` remote is not special, except that it's the default remote name used if you run `fetch` without any arguments.

We can list the remotes we have by running the `remote` command with no arguments, or with the `--verbose` flag.

```
$ git remote
origin

$ git remote --verbose
origin  git@github.com:jcoglan/jit.git (fetch)
origin  git@github.com:jcoglan/jit.git (push)
```

We can also remove remotes by running `remote remove <name>`, which deletes them from the configuration along with any refs in `.git/refs/remotes` as specified by their `fetch` setting.

Let's sketch out the start of a `remote` command implementation that defines the options and sub-commands, before we tackle each operation in turn.

```
# lib/command/remote.rb

module Command
  class Remote < Base

    def define_options
      @parser.on("-v", "--verbose") { @options[:verbose] = true }
```

```
    @options[:tracked] = []
    @parser.on("-t <branch>") { |branch| @options[:tracked].push(branch) }
end

def run
  case @args.shift
  when "add"    then add_remote
  when "remove" then remove_remote
  else              list_remotes
  end
end

# ...

end
end
```

The `run` method shifts the sub-command name of the front of the `@args` array, so the `@args` array will now only contain the arguments to the sub-command, not its name. These functions are mostly boilerplate for storing and retrieving data, so we'll rattle through them fairly quickly before moving on to the more conceptually interesting elements.

26.2.1. Adding a remote

The `add` sub-command takes a name and a URL. The name must not already be registered as a remote in the current repository; if it is then we'll get an exception that we must catch. The `:tracked` option is an array populated by the `-t` flag; rather than getting the default fetch setting of `+refs/heads/*:refs/remotes/<name>/*`, we can specify which branches we'd like to fetch. Running `remote add ... -t master -t topic` will result in these fetch settings being added:

```
fetch = +refs/heads/master:refs/remotes/origin/master
fetch = +refs/heads/topic:refs/remotes/origin/topic
```

`remote.<name>.fetch` is a multi-valued variable in the configuration. The `add_remote` sub-command just sends its input off to another object that does the real work.

```
# lib/command/remote.rb

def add_remote
  name, url = @args[0], @args[1]
  repo.remotes.add(name, url, @options[:tracked])
  exit 0
rescue Remotes::InvalidRemote => error
  @stderr.puts "fatal: #{error.message}"
  exit 128
end
```

`Repository#remotes` is a new piece of repository state management that wraps around the repo's local config file in order to store its data.

```
# lib/repository.rb

def remotes
  @remotes ||= Remotes.new(config.file(:local))
```

```
end
```

It returns an instance of the `Remotes` class, which is what does the real work of storing and retrieving remote configuration. We do this in a class connected to `Repository`, rather than in the `remote` command itself, because other commands like `fetch` and `push` will need access to this information.

```
# lib/remotes.rb

class Remotes
  DEFAULT_REMOTE = "origin"

  InvalidRemote = Class.new(StandardError)

  def initialize(config)
    @config = config
  end

  #
end
```

Now we get to the real logic, inside the `Remotes#add` method. It takes the remote name, URL, and possibly-empty list of tracking branches, and adds a new remote to the config.

The list of tracking branches is set to `["*"]` if the input is empty. If the given name already exists as a remote in the config, then we raise an error. Otherwise, we set the `remote.<name>.url` variable, and add a `remote.<name>.fetch` setting for each tracking branch. This is done by using the `Refspec` class, defined below, to serialise a spec whose source is `refs/heads/<branch>`, whose target is `refs/remotes/<name>/<branch>`, and whose force flag is set.

```
# lib/remotes.rb

def add(name, url, branches = [])
  branches = [ "*" ] if branches.empty?
  @config.open_for_update

  if @config.get(["remote", name, "url"])
    @config.save
    raise InvalidRemote, "remote #{ name } already exists."
  end

  @config.set(["remote", name, "url"], url)

  branches.each do |branch|
    source = Refs::HEADS_DIR.join(branch)
    target = Refs::REMOTES_DIR.join(name, branch)
    refspect = Refspec.new(source, target, true)

    @config.add(["remote", name, "fetch"], refspect.to_s)
  end

  @config.save
end
```

For now, the `Remotes::Refspec` class just takes the input source, target and forced flag, and serialises them. Later on we'll expand this class so that it can select references for us to fetch.

```
# lib/remotes/refspec.rb

class Remotes

  Refspec = Struct.new(:source, :target, :forced) do
    def to_s
      spec = forced ? "+" : ""
      spec + [source, target].join(":")
    end
  end

end
```

26.2.2. Removing a remote

Removing a remote is essentially the reverse of the above operation. In `Command::Remote`, we dispatch the `remove` sub-command to the `Remotes#remove` method:

```
# lib/command/remote.rb

def remove_remote
  repo.remotes.remove(@args[0])
  exit 0
rescue Remotes::InvalidRemote => error
  @stderr.puts "fatal: #{error.message}"
  exit 128
end
```

In `Remotes#remove`, we delete the entire section for the remote from the configuration. If this call returns `false`, then the given remote did not actually exist, and we raise an exception to notify the user.

```
# lib/remotes.rb

def remove(name)
  @config.open_for_update

  unless @config.remove_section(["remote", name])
    raise InvalidRemote, "No such remote: #{name}"
  end
ensure
  @config.save
end
```

26.2.3. Listing remotes

To list the remotes currently configured for the repository, we call the `remote` command without arguments, or with the `--verbose` option. To get the information this command needs to display, we need to wrap an object interface around the information currently stored in the `config` file.

The sub-command begins by calling `Remotes#list_remotes` to get the names of all the remotes stored in the configuration. Unless we're in verbose mode, we just print those names. In verbose mode, we fetch the remote's details using `Remotes#get`, and then the `fetch_url` and `push_url` methods of the returned object to get the relevant URLs.

```
# lib/command/remote.rb
```

```
def list_remotes
  repo.remotes.list_remotes.each { |name| list_remote(name) }
  exit 0
end

def list_remote(name)
  return puts name unless @options[:verbose]

  remote = repo.remotes.get(name)

  puts "#{name}\t#{remote.fetch_url} (fetch)"
  puts "#{name}\t#{remote.push_url} (push)"
end
```

Remotes#list_remotes just needs to load the config file, then use the Config#subsections method to get the names of all the remote section's subsections.

```
# lib/remotes.rb

def list_remotes
  @config.open
  @config.subsections("remote")
end
```

The Remotes#get method's job is to return an object that let us retrieve values necessary for fetch and push from a specific remote's subsection within the config. The Config class doesn't directly provide a way of accessing a subsection as a first-class object as in most cases this is not necessary, so we'll invent a specific class called Remotes::Remote to play this role.

```
# lib/remotes.rb

def get(name)
  @config.open
  return nil unless @config.section?("remote", name)

  Remote.new(@config, name)
end
```

The Remotes::Remote class takes the Config object and the name of the remote and provides an interface for accessing the relevant variables.

```
# lib/remotes/remote.rb

class Remotes
  class Remote

    def initialize(config, name)
      @config = config
      @name   = name

      @config.open
    end

    # ...
  end
end
```

```
end
```

To provide the `fetch_url` and `push_url` interface, the `Remote` class has to look the relevant variables up in the `Config` object, based on its name. If `remote.<name>.pushurl` is not set, then the push URL defaults to being the same as the fetch URL.

```
# lib/remotes/remote.rb

def fetch_url
  @config.get(["remote", @name, "url"])
end

def push_url
  @config.get(["remote", @name, "pushurl"]) || fetch_url
end
```

The `fetch` command will also need the `fetch` `refsspecs`, and `remote.<name>.fetch` is multi-valued so we use `Config#get_all` to retrieve it. Finally this command will need to access a setting called `uploadpack` for the given remote, which sets the name of a program that the `fetch` command will run to interact with the remote repository⁵.

```
# lib/remotes/remote.rb

def fetch_specs
  @config.get_all(["remote", @name, "fetch"])
end

def uploader
  @config.get(["remote", @name, "uploadpack"])
end
```

26.3. Refspecs

When defining the `remote add` command above, we introduced the `remote.<name>.fetch` variable and the `Refspec` class. We briefly alluded to the idea that a `refspec` is a description of the refs we want to copy from the remote, and where they should be copied to in our local repository. For example, the `refs/heads/*:refs/remotes/origin/*` instructs the `fetch` command to copy all the refs under `refs/heads/` from the remote into the directory `refs/remotes/origin` in our repo.

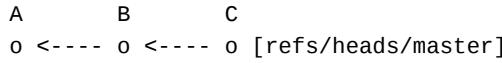
The general form of a `refspec` is `<source>:<target>` where `<source>` and `<target>` are paths inside the `.git` directory. It is interpreted as an instruction to copy the ref `<source>` from the remote into the ref `<target>` in the local repository. The source and target can also include a wildcard, as in `refs/heads/*:refs/remotes/origin/*`, which means we should copy every ref beginning with `refs/heads/` from the remote to a corresponding ref beginning with `refs/remotes/origin/` in our local repository.

If the `refspec` does not have a leading `+`, then the target will only be updated on a fast-forward, that is if the target commit ID is an ancestor of the source, as in a fast-forward merge.

For example, suppose Alice has the following three commits on her `master` branch:

⁵Section 28.2, “The `fetch` and `upload-pack` commands”

Figure 26.1. Alice's history



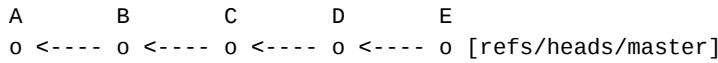
Bob sets up a remote to point at Alice's repo, with the variable `remote.alice.fetch` set to `refs/heads/master:refs/remotes/alice/master`, without a leading `+`. When he fetches, he'll receive all of Alice's commits, and his `refs/remotes/alice/master` ref will be equal to Alice's `refs/heads/master`.

Figure 26.2. Bob's repository after fetching from Alice



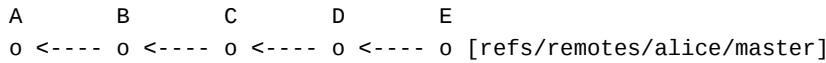
Now, Alice adds some more commits to her branch on top of commit *C*, which Bob's remote ref currently points at.

Figure 26.3. Alice's extended history



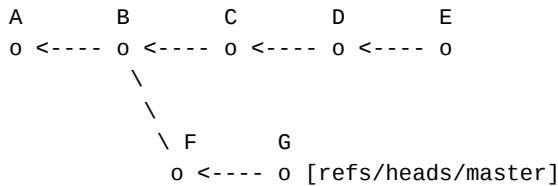
When Bob fetches a second time, he'll receive commits *D* and *E* which he does not yet have, and since *E* is a descendant of *C*, his `refs/remotes/alice/master` ref is updated to point at *E*.

Figure 26.4. Bob's updated repository



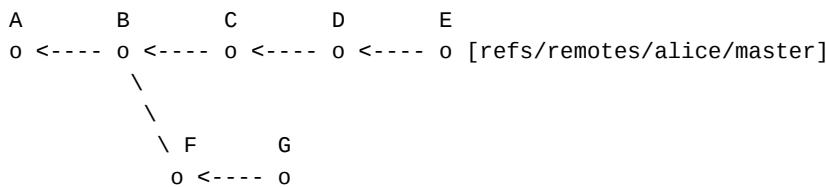
But, supposing Alice changed her mind about those last few commits and went back to *B* to start a new line of commits, she'd end up with this history:

Figure 26.5. Alice's amended history



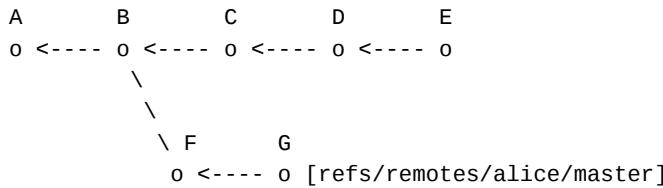
If Bob fetches from Alice now, he will still receive commits *F* and *G* which he does not have, but because *G* is not a descendant of the commit his remote ref points at, that ref will not be updated.

Figure 26.6. Bob's repository after an unforced fetch



If Bob instead prefixed his `fetch` refspec with a leading plus sign, indicating the refspec is *forced*, the ref would be updated to point at *G*.

Figure 26.7. Bob's repository after a forced fetch



When the `fetch` command is run, the first thing it does is retrieve a list of the remote repository's refs and their current values. For example if we fetched from the Git project's own repository, we might receive:

```

5d826e972970a784bd7a7bdf587512510097b8c7 HEAD
98cdfbb84ad2ed6a2eb43dafa357a70a4b0a0fad refs/heads/maint
5d826e972970a784bd7a7bdf587512510097b8c7 refs/heads/master
b2cc3488ba006e3ba171e85dffbe6f332f84bf9a refs/heads/todo
    
```

The job of a refspec is to select which refs we'd like to copy from the remote, and where to save them in our local repository. For example, given the above list and the refspec `refs/heads/*:refs/remotes/origin/*`, we'd like to produce this structure, where the boolean indicates whether this mapping is forced.

```

{
  "refs/remotes/origin/maint" => ["refs/heads/maint", false],
  "refs/remotes/origin/master" => ["refs/heads/master", false],
  "refs/remotes/origin/todo"   => ["refs/heads/todo", false]
}
    
```

If the refspec was `+refs/heads/ma*:refs/*` then we would instead get this set of mappings:

```

{
  "refs/int"  => ["refs/heads/maint", true],
  "refs/ster"  => ["refs/heads/master", true]
}
    
```

In general, since the `remote.<name>.fetch` setting is multi-valued, we'll be matching a list of refs against a list of refspecs. In this case we can match each refspec against the list to generate the above structure, and then merge all the results into a single hash.

We'll need to begin by parsing a refspec string into a `Refspec` object. The regular expression `REFSPEC_FORMAT` captures an optional leading `+`, and then two sections of characters other than `:`, separated by `:`.

```

# lib/remotes/refspec.rb

REFSPEC_FORMAT = /^(\+?)(([^\:]+):([^\:]+)$)

def self.parse(spec)
  match = REFSPEC_FORMAT.match(spec)
  Refspec.new(match[2], match[3], match[1] == "+")
end
    
```

We'll define `Refspec.expand` as a method that takes a list of refspec strings, and a list of refs, also strings, and returns a hash of the above form. To do this, it parses all the specs, calls `Refspec#match_refs` on each one, and combines the results using `Hash#merge`⁶.

⁶<https://docs.ruby-lang.org/en/2.3.0/Hash.html#method-i-merge>

```
# lib/remotes/refspec.rb

def self.expand(specs, refs)
  specs = specs.map { |spec| Refspec.parse(spec) }

  specs.reduce({}) do |mappings, spec|
    mappings.merge(spec.match_refs(refs))
  end
end
```

To perform the pattern-matching, we can convert each spec's source into a regular expression, for example `refs/heads/*` becomes `refs/heads/(.*)`. The `(.*)` matches and captures any number of characters. By matching this expression against each input ref, we can find the part of the ref that matches the `*`, and substitute it for the `*` in the target. If the source does not contain a `*`, then the refspec is treated literally and not matched against the list of refs.

```
# lib/remotes/refspec.rb

def match_refs(refs)
  return { target => [source, forced] } unless source.to_s.include?("*")

  pattern = /^#{source.sub("*", "(.*)")}$/s
  mappings = {}

  refs.each do |ref|
    next unless match = pattern.match(ref)
    dst = match[1] ? target.sub("*", match[1]) : target
    mappings[dst] = [ref, forced]
  end

  mappings
end
```

Let's try our `Refspec.expand` method out on a couple of examples and see that it does what we expect:

```
>> refs = ["HEAD", "refs/heads/master", "refs/other"]

>> specs = ["+refs/heads/*:refs/remotes/origin/*"]
>> Remotes::Refspec.expand(specs, refs)
=> { "refs/remotes/origin/master" => ["refs/heads/master", true] }

>> specs = ["refs/other:refs/remotes/origin/other"]
>> Remotes::Refspec.expand(specs, refs)
=> { "refs/remotes/origin/other" => ["refs/other", false] }
```

This is just what we wanted; the method works with wildcard and non-wildcard refspecs, and produces the right mappings with *forced* flags set correctly.

26.4. Finding objects

After using the refspecs to decide which references should be copied, the `fetch` command needs to download a bundle of objects from the remote repo. Ideally, this bundle should be as small as possible, consisting only of commits the local repository does not already have. But as well as commits, it needs to contain all the trees and blobs that make up the content of those commits, where again these should be limited to objects the receiver is lacking.

In Section 16.2.1, “Revision lists” we introduced the `RevList` class for generating lists of commits, and in Section 17.6, “Logs in a merging history” and Section 23.2.1, “Rev-list without walking” we extended it so that it could handle branching and merging histories and the demands of the `cherry-pick` and `revert` commands. We now need to extend it further to support finding trees and blobs.

We’ll do this by adding an `:objects` option to `RevList`, which defaults to `false` so the class continues to return only commits unless this option is set. We will also add a new item of state to the class: the `@pending` array will store the root tree IDs of any commits we find as we traverse the history. We’ll find all the objects that need downloading by marking commits the receiver already has as uninteresting. So, we don’t want to emit any objects we find until we’ve finished traversing the history and we know those objects aren’t part of uninteresting commits that the receiver already has. Instead, we’ll store them up in the `@pending` array and emit them at the end.

```
# lib/rev_list.rb

def initialize(repo, revs, options = {})
  # ...
  @pending = []
  # ...

  @objects = options.fetch(:objects, false)
  @walk    = options.fetch(:walk, true)

  #
end
```

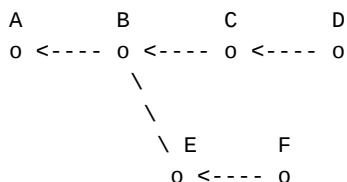
In the `RevList#each` method, we need two extra steps. First, if the `:objects` option was used, we call `mark_edges_uninteresting`, which will find the nearest commits that were excluded from the results, and mark all their content as uninteresting. Second, we add a step after `traverse_commits` to iterate over the pending objects and emit them.

```
# lib/rev_list.rb

def each
  limit_list if @limited
  mark_edges_uninteresting if @objects
  traverse_commits { |commit| yield commit }
  traverse_pending { |object| yield object }
end
```

The call to `mark_edges_uninteresting` takes care of flagging any trees and blobs that are part of commits excluded from the search. For example, imagine evaluating the revision range $D..F$ against this history:

Figure 26.8. Forking history



After running `limit_list`, the output queue will contain commits F and E , while commits D , C and B will be marked as uninteresting. The *edge* of a query is the set of uninteresting commits

immediately reachable from the interesting ones, that form the boundary of the result set. In this case, the edge consists of commit *B*. The idea is that the edge commits are the ones that all the interesting commits are derived from, so the interesting commits will contain a lot of the same trees and blobs as the edge commits. By marking the trees of edge commits as uninteresting, we can avoid sending the entire current state of the project over the network and thus make the bundle a lot smaller.

`mark_edges_uninteresting` iterates over the commits in the output queue. For each commit, we call `mark_tree_uninteresting` with the commit's tree ID if that commit was marked uninteresting. Then we check the commit's parents, to get to the potentially-uninteresting commits reachable from the interesting set, and do the same.

```
# lib/rev_list.rb

def mark_edges_uninteresting
  @queue.each do |commit|
    if marked?(commit.oid, :uninteresting)
      mark_tree_uninteresting(commit.tree)
    end

    commit.parents.each do |oid|
      next unless marked?(oid, :uninteresting)

      parent = load_commit(oid)
      mark_tree_uninteresting(parent.tree)
    end
  end
end
```

The `mark_tree_uninteresting` method recursively marks every tree and blob within a given tree as uninteresting. It does this by building a `Database::Entry` for the given tree ID, and using a helper method `traverse_tree` to recurse over its structure. This method takes the `Entry` object, and yields it to the caller. If `yield entry`, that is the given block, returns false, then we bail out. This would happen if the `mark` method returns false indicating the tree is already marked so we can stop recursing into it. If the `Entry` represents a tree, then we load it from the database and recursively call `traverse_tree` on each of its entries.

```
# lib/rev_list.rb

def mark_tree_uninteresting(tree_oid)
  entry = @repo.database.tree_entry(tree_oid)
  traverse_tree(entry) { |object| mark(object.oid, :uninteresting) }
end

def traverse_tree(entry)
  return unless yield entry
  return unless entry.tree?

  tree = @repo.database.load(entry.oid)

  tree.entries.each do |name, item|
    traverse_tree(item) { |object| yield object }
  end
end
```

`traverse_tree` takes a `Database::Entry` rather than a simple object ID because it needs to know if the entry points to a tree or not. Therefore `mark_tree_uninteresting` needs to build such an entry from a commit's tree pointer, and it uses a helper from the `Database` class to do this.

```
# lib/database.rb

def tree_entry(oid)
  Entry.new(oid, Tree::TREE_MODE)
end
```

Having marked all the uninteresting tree and blobs, we can now iterate over the commits we've found and if we decide to yield the commit, we'll add an entry for its tree to the `@pending` array.

```
# lib/rev_list.rb

def traverse_commits
  until @queue.empty?
    commit = @queue.shift
    #

    @pending.push(@repo.database.tree_entry(commit.tree))
    yield commit
  end
end
```

Finally, the `traverse_pending` method takes care of emitting all the pending trees and blobs to the caller. For every entry in the `@pending` array, we call `traverse_tree` and yield every object to the caller, as long as the object is not marked as uninteresting and we've not already seen it. The block needs to return `true` so that `traverse_tree` explores the whole tree in this case.

```
# lib/rev_list.rb

def traverse_pending
  return unless @objects

  @pending.each do |entry|
    traverse_tree(entry) do |object|
      next if marked?(object.oid, :uninteresting)
      next unless mark(object.oid, :seen)

      yield object
      true
    end
  end
end
```

We can now use `RevList` to find all kinds of content. As well as commits, it will now yield trees and blobs. Note however that whereas commits are emitted as `Database::Commit` objects, trees and blobs appear as `Database::Entry` objects that have only an object ID and a mode. This avoids loading the full content of blobs, which are not necessary to complete this search operation. If the caller needs to read blob content it must load them itself.

To finish up our changes to `RevList`, we're going to need two other small changes that help with searching for objects. First, it will be useful during `fetch` to find all the objects reachable from any of a repository's references, i.e. `HEAD` and anything under `refs`. We'll add an `:all` option to

the class, that will cause it to queue up all the commits referenced via the `Refs#list_all_refs` method.

```
# lib/rev_list.rb

def initialize(repo, revs, options = {})
  # ...

  include_refs(repo.refs.list_all_refs) if options[:all]

  revs.each { |rev| handle_revision(rev) }
  handle_revision(Revision::HEAD) if @queue.empty?
end

def include_refs(refs)
  oids = refs.map(&:read_oid).compact
  oids.each { |oid| handle_revision(oid) }
end
```

The other improvement we need relates to how missing objects are handled. Sending objects between repositories involves finding all the objects that the sender has but which the receiver is missing. For example, if the receiver has commits *A* and *B* and is requesting commits *X* and *Y*, then we can find all the commits that need sending by calling `RevList` on the remote repository with the inputs A , B , *X* and *Y*. But if the remote repository does not have commits *A* and *B* this query will fail.

We're only supplying the inputs A and B to limit the search, to try to avoid sending unnecessary objects. It's an optimisation, so it's actually fine if the sender doesn't know about these commits and chooses to ignore them. To allow this, I'm going to add an option called `:missing` to `RevList` that instructs it to suppress errors resulting from revisions that don't resolve.

```
# lib/rev_list.rb

def initialize(repo, revs, options = {})
  # ...

  @objects = options.fetch(:objects, false)
  @missing = options.fetch(:missing, false)
  @walk   = options.fetch(:walk, true)

  # ...
end
```

The effect of this option is that, if `Revision#resolve` raises a `Revision::InvalidObject` error, we'll swallow it. This makes it okay to run a query involving IDs of commits that don't exist in one of the repositories.

```
# lib/rev_list.rb

def set_start_point(rev, interesting)
  rev = Revision::HEAD if rev == ""
  oid = Revision.new(@repo, rev).resolve(Revision::COMMIT)

  # ...
```

```
rescue Revision::InvalidObject => error
  raise error unless @missing
end
```

Now that we can configure remotes and search for objects, the next step is to define a method for sending information over the network.

27. The network protocol

So far, our picture of the `fetch` command is still quite vague. We know that it grabs a list of the refs and their values from the remote repository, uses `refsspecs` to decide which ones it wants to download, and then somehow determines which objects need to be copied from the remote to the local repository to complete the history of those refs. But we've not yet seen how these data transfers actually happen — how the `fetch` command connects to a remote computer and communicates with the remote repository.

Git actually supports multiple transports for its network protocol, including SSH¹, HTTP² and Git's own custom transport. The protocol differs somewhat based on the transport being used, but we will only look at one transport here: SSH. The HTTP and Git transports require custom services running on the remote computer for Git to receive and handle requests, whereas SSH is a generic protocol that's running on many internet-connected computers out of the box. SSH is a great choice for bootstrapping a network protocol as it provides an encrypted and authenticated connection to a remote machine and allows arbitrary programs to be invoked over that connection. Git's SSH transport is full-featured and will allow us to do everything the `fetch` and `push` commands need.

27.1. Programs as ad-hoc servers

You may be familiar with the concept of a server from using the web. Your browser sends a request to a server identified by a URL, which sends back a response, and your browser displays that response as a web page. The web uses HTTP to perform this request-response cycle, and that requires having a long-running process on the host machine that's listening for TCP³ connections on port 80 or TLS⁴ connections on port 443. Apache⁵ and NGINX⁶ are popular HTTP servers⁷.

SSH works in a similar way: there is a long-running process on the remote machine that accepts incoming connections, but instead of responding to HTTP requests it allows the client to submit commands which the remote machine will then run. We will take advantage of this existing infrastructure to build our Git protocol.

Now, it is not only long-running processes accepting network connections that can act as servers. Any short-lived program can perform a similar task using its standard input and output streams. For example, here's a little program that loops, reading lines from `stdin` using `IO#gets`⁸. If the line reads `exit`, the program quits. Otherwise, it reads two numbers from the line, adds them together, and prints the result to `stdout`.

```
# adder.rb

$stdin.sync = $stdout.sync = true
```

¹https://en.wikipedia.org/wiki/Secure_Shell

²https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

³https://en.wikipedia.org/wiki/Transmission_Control_Protocol

⁴https://en.wikipedia.org/wiki/Transport_Layer_Security

⁵<https://httpd.apache.org/>

⁶<https://www.nginx.com/>

⁷You will hear the word *server* used to refer both to these long-running processes that handle incoming requests, and to the computers these processes run on. I will try to stick to the former meaning and use the term *computer*, *machine* or *host* for the latter.

⁸<https://docs.ruby-lang.org/en/2.3.0/IO.html#method-i-gets>

```
loop do
  query = $stdin.gets.strip
  exit if query == "exit"

  x, y = query.scan(/\d+/).map(&:to_i)

  $stdout.puts(x + y)
end
```

We can use this program by starting up an instance of it and then interacting with via its standard I/O. Below is a program that does this by running `ruby adder.rb` as a child process, using Ruby's `Open3`⁹ module. This wraps a convenient interface around the low-level system calls necessary to start that child process and expose its input and output streams. It also returns a thread that you can wait on if you want to wait for the process to exit. Our client program feeds three queries into `adder.rb` by writing to its input stream, and reads the results from its output stream. When it's done, it writes `exit`, causing the child process to exit.

```
# client.rb

require "open3"

queries = [ [5, 2], [8, 3], [6, 7] ]

input, output, thread = Open3.popen2("ruby adder.rb")
input.sync = output.sync = true

queries.each do |x, y|
  input.puts([x, y].join(","))
  result = output.gets

  puts "#{x} + #{y} = #{result}"
end

input.puts("exit")
```

When we run this program, we see the results computed by `adder.rb`:

```
5 + 2 = 7
8 + 3 = 11
6 + 7 = 13
```

This demonstrates how one process is able to launch another and then communicate with it, much like any other client would talk to a server. What makes SSH so useful is that we can use it to invoke a program on another machine, and communicate with it in exactly the same way. I am currently using Vagrant¹⁰ to run a development environment in a virtual machine on my laptop. I can make the `adder.rb` program available inside that machine and then invoke it over SSH by running:

```
$ ssh 127.0.0.1 -p 2222 -l ubuntu "ruby /vagrant/adder.rb"
```

This uses the `ssh`¹¹ program to connect to the virtual machine as the user `ubuntu` on port 2222, and then runs `ruby /vagrant/adder.rb` inside that machine. The `ssh` process's standard I/O

⁹<https://docs.ruby-lang.org/en/2.3.0/Open3.html>

¹⁰<https://www.vagrantup.com/>

¹¹<https://manpages.ubuntu.com/manpages/bionic/en/man1/ssh.1.html>

is connected to the process running on the remote machine, so we can interact with this ssh process and it looks just like we’re talking to the adder.rb process. All we need to do is replace the code for launching adder.rb from the client program:

```
command = "ssh 127.0.0.1 -p 2222 -l ubuntu 'ruby /vagrant/adder.rb'"  
input, output, thread = Open3.popen2(command)
```

If we run client.rb now it prints exactly the same results. The beauty of this is that we can use the same protocol to talk to a program running locally as we do to talk to a remote program over SSH. All we have to change is the initial command we run, and then everything else proceeds exactly as before.

27.2. Remote agents

In Section 26.3, “Refspecs”, we introduced the first step of the fetch process, which is to retrieve a list of the remote repository’s refs and the commits they point at. Now that we’ve seen how we can use standard I/O to communicate with a process on a remote machine, we can take a closer look at how that information is transferred.

In principle, an SSH connection would allow `fetch` to read and write arbitrary files in the remote repository, and thus use a history search to determine which objects to download. However, this has two problems. First, it would be very slow. Enumerating a `RevList` requires a potentially large number of sequential disk reads, and if each of these reads has to make a round trip over the network the search will take a long time. Second, it’s a security risk. Somebody providing Git hosting probably does not want third parties to be able to run arbitrary commands on their machines. Luckily, the SSH server config lets the administrator lock things down so that a user connected via SSH is only allowed to run programs from an approved list. The administrator can also prevent SSH connections from gaining access to the shell.

Therefore, rather than using low-level read/write commands on the remote machine, `fetch` runs a program on the remote that can perform all the relevant searches locally, and it communicates with this program over standard I/O as we’ve just seen. This program is called `git-upload-pack`, and it works as an *agent* on behalf of the `fetch` command, providing information about the repository and sending all the objects the `fetch` command wants. Note that `git-upload-pack` is the name of a standalone program, not a subcommand of `git` — this means the server owner can allow SSH connections to run the `git-upload-pack` program but not the rest of the `git` commands.

To see this in action, we can run `git-upload-pack` against a remote repository using ssh. Let’s try Git’s own repository hosted on GitHub:

```
$ ssh git@github.com "git-upload-pack '/git/git.git'"  
  
01005d826e972970a784bd7a7bdf587512510097b8c7 HEADmulti_ack thin-pack \  
    side-band side-band-64k ofs-delta shallow deepen-since deepen-not \  
    deepen-relative no-progress include-tag multi_ack_detailed  
003e98cdfbb84ad2ed6a2eb43dafa357a70a4b0a0fad refs/heads/maint  
003f5d826e972970a784bd7a7bdf587512510097b8c7 refs/heads/master  
003db2cc3488ba006e3ba171e85dffbe6f332f84bf9a refs/heads/todo
```

This output contains more than just the ref names and commit IDs, and we’ll see what those extra details mean in the next section. The important point here is that `fetch` can communicate

with the remote repo simply by running the above command. It can read from the output of `git-upload-pack` to retrieve information and download objects, and it can also write to the input of this process to request the commits it would like to receive. While it's running, `git-upload-pack` acts as a server that the client, the `fetch` command, can talk to.

27.3. The packet-line protocol

In order to communicate with the remote repository via the `git-upload-pack` program, we need a means of formatting messages sent between the two processes. In our `adder.rb` example, we used `puts` to write a message with a line feed (`\n`) at the end, and `gets` to read until the next line feed. This splits the input and output streams up into distinct messages.

Above we saw that the `git-upload-pack` program sends the following data when it's first run (a backslash represents a continuation of the current line):

```
01005d826e972970a784bd7a7bdf587512510097b8c7 HEADmulti_ack thin-pack \
    side-band side-band-64k ofs-delta shallow deepen-since deepen-not \
    deepen-relative no-progress include-tag multi_ack_detailed
003e98cdfbb84ad2ed6a2eb43dafa357a70a4b0a0fad refs/heads/maint
003f5d826e972970a784bd7a7bdf587512510097b8c7 refs/heads/master
003db2cc3488ba006e3ba171e85dffbe6f332f84bf9a refs/heads/todo
```

This represents the series of messages we saw earlier in the chapter:

```
5d826e972970a784bd7a7bdf587512510097b8c7 HEAD
98cdfbb84ad2ed6a2eb43dafa357a70a4b0a0fad refs/heads/maint
5d826e972970a784bd7a7bdf587512510097b8c7 refs/heads/master
b2cc3488ba006e3ba171e85dffbe6f332f84bf9a refs/heads/todo
```

Our `adder.rb` example used line feeds to delimit messages; each process reads from the input stream until it sees `\n`. Another common way to split a stream into messages is to prefix every message with its length, so the recipient parses the stream by reading a length header, then reading the given number of bytes. Git's protocol¹² works this way; every message is terminated with `\n` and is then prefixed with a four-digit hexadecimal number giving the length of the message and this header. For example, if you look at the line for `refs/heads/maint` above, you'll see it's prefixed with `003e`:

```
>> msg = "98cdfbb84ad2ed6a2eb43dafa357a70a4b0a0fad refs/heads/maint\n"
>> (msg.bytesize + 4).to_s(16).rjust(4, "0")
=> "003e"
```

If a message begins with `0000`, that represents an empty message and is known as a *flush packet*.

The first line — the `HEAD` reference — has some extra data appearing after the ref name. Immediately after the word `HEAD`, we see `multi_ack thin-pack side-band...`. These words denote *capabilities*¹³, optional aspects of the protocol that the peer supports. The first message sent by each party must be suffixed with a null byte followed by a space-separated list of its capabilities. The remote agent (`git-upload-pack`) always sends the first message in the conversation, and the client (`fetch`) should only send capabilities that the remote agent has

¹²<https://git-scm.com/docs/protocol-common>

¹³<https://git-scm.com/docs/protocol-capabilities>

sent; the remote agent will send some capabilities purely to advertise that it recognises them. For now we won't worry about what the various capabilities mean, but we'll make space for them in our protocol implementation.

To implement this messaging scheme, we need a class that represents the communication channel between the client and the remote agent. It will take the name of the command that's calling it, an input and output stream, and an optional list of capabilities as input. It needs the name of the calling command as this affects the serialisation of capabilities slightly. It sets the sync flag on both streams, which makes Ruby flush data to and from the underlying file descriptors immediately without buffering anything. The remote agent will instantiate this class with its standard input and output streams, while the client will use the streams it got by calling `Open3.popen2` to launch the agent.

```
# lib/remotes/protocol.rb

class Remotes
  class Protocol

    attr_reader :input, :output

    def initialize(command, input, output, capabilities = [])
      @command = command
      @input = input
      @output = output

      @input.sync = @output.sync = true

      @caps_local = capabilities
      @caps_remote = nil
      @caps_sent = false
    end

    #
    # ...
    #

    end
  end
end
```

To send a message, we want to write to the output stream. If the given input is `nil`, this denotes a flush packet and we write `0000` and nothing more. Otherwise, we call `append_caps` which appends the local capabilities to the message if they've not been sent, then calculate the size, which is the byte length of the message, plus 5 — 4 bytes for the length header and 1 for the `\n` at the end. Then we write all the data to the output.

```
# lib/remotes/protocol.rb

def send_packet(line)
  return @output.write("0000") if line == nil

  line = append_caps(line)
  size = line.bytesize + 5

  @output.write(size.to_s(16).rjust(4, "0"))
  @output.write(line)
  @output.write("\n")
end
```

append_caps takes a message and, if the capabilities have not already been sent, appends them to the message. Most commands use a null byte to separate the capabilities from the message, but fetch uses a space. If @caps_remote is set that means we've received capabilities from the other peer and should limit the ones we send to capabilities that appear in that list; a & b finds the common elements of arrays a and b.

```
# lib/remotes/protocol.rb

def append_caps(line)
  return line if @caps_sent
  @caps_sent = true

  sep = (@command == "fetch") ? " " : "\0"
  caps = @caps_local
  caps &= @caps_remote if @caps_remote

  line + sep + caps.join(" ")
end
```

To read a message, we read the next four bytes from the input stream. If this string is not a four-digit hexadecimal number, then we'll just return it; it will denote the start of a different kind of message and the caller will know what to do with it. Otherwise, we interpret this string as a length and read that many bytes from the input.

```
# lib/remotes/protocol.rb

def recv_packet
  head = @input.read(4)
  return head unless /[0-9a-f]{4}/ =~ head

  size = head.to_i(16)
  return nil if size == 0

  line = @input.read(size - 4).sub(/\n$/, "")
  detect_caps(line)
end
```

The detect_caps method deals with the first message received, which might contain a capability list at the end. The first message fetch sends will be of the form want <ID> <capabilities>, so the upload-pack command needs to split this on the second space. All other commands should split the message on the first null byte to find the capability list. This method returns the rest of the line that preceded the delimiter, after storing the capabilities in @caps_remote.

```
# lib/remotes/protocol.rb

def detect_caps(line)
  return line if @caps_remote

  if @command == "upload-pack"
    sep, n = " ", 3
  else
    sep, n = "\0", 2
  end

  parts = line.split(sep, n)
```

```

caps = (parts.size == n) ? parts.pop : ""

@caps_remote = caps.split(/ +/)
parts.join(" ")
end

```

When implementing the protocol, we'll need to be able to check what capabilities the other peer supports; the `Protocol#capable?` method returns `true` if the remote peer's capabilities include the given name.

```

# lib/remotes/protocol.rb

def capable?(ability)
  @caps_remote&.include?(ability)
end

```

Finally, it will be useful to read messages in a loop until we receive a particular message that indicates the end of a list. `Protocol#recv_until` takes a string and reads messages from the input until a message matches that string. Every message until that point is yielded to the caller.

```

# lib/remotes/protocol.rb

def recv_until(terminator)
loop do
  line = recv_packet
  break if line == terminator
  yield line
end
end

```

That completes the protocol for sending messages between the client and the remote agent. Here's another version of our `adder.rb` program that uses `Remotes::Protocol` as a wrapper around its I/O streams. It accepts an arbitrary number of inputs from the client, followed by a flush-packet, then sends the sum of all the numbers it's received.

```

# adder.rb

require_relative "./lib/remotes/protocol"

conn = Remotes::Protocol.new($stdin, $stdout)

numbers = []
conn.recv_until(nil) { |n| numbers << n.to_i }
sum = numbers.reduce(0) { |a, b| a + b }
conn.send_packet(sum.to_s)

```

A client can invoke this program via `Open3` and then use `Remotes::Protocol` to send its inputs and receive the result.

```

# client.rb

require "open3"
require_relative "./lib/remotes/protocol"

input, output, thread = Open3.popen2("ruby adder.rb")
conn = Remotes::Protocol.new(output, input)

(1..10).each { |n| conn.send_packet(n.to_s) }

```

```
conn.send_packet(nil)

result = conn.recv_packet.to_i

puts result # => 55
```

Next we need to examine the pack format, and then we'll have everything we need to send objects between repositories.

27.4. The pack format

To complete the communication protocol between `fetch` and `git-upload-pack`, we need a format for sending all the objects that `fetch` requests over the network. The wire protocol we saw in the last chapter is designed for sending short messages; the largest message size it can support is FFFF_{16} , or $65,535_{10}$, which is 64 kilobytes¹⁴. For sending object data, which is typically much larger, the protocol switches to a different format: the pack format¹⁵. This provides a way to send a stream of objects over the wire, making the total payload as small as possible to reduce transfer time.

In order to transfer a set of objects from the remote to the local repository, `git-upload-pack` bundles them up into what Git calls a *pack*. `fetch` then needs to parse this pack, and then either keep it as the primary storage for those objects, or extract it into a single file per object as we currently store them. To begin with, we will not implement delta compression in our packs. That can be added later without changing the commands that use this format.

At a high level, the pack format is fairly simple. Like the index¹⁶, it begins with a 4-byte signature (`PACK` rather than `DIRC`), a 4-byte version number (2 or 3), and a 4-byte number indicating how many objects the pack contains. This is followed by the given number of object records, and finally the SHA-1 hash of all the preceding content.

An object record consists of a header declaring the object's type and size, followed by its data in compressed form. So conceptually it's similar to a file in `.git/objects`; the difference is that the type and size are not part of the compressed data and are represented differently. The object ID is not contained in the pack since it can be computed from the object's content.

The object's type is represented numerically rather than as a string. Commits have type 1, trees type 2, and blobs are type 3. The size is represented as a little-endian variable-length number, similar to a *variable-length quantity*¹⁷. Let's say a blob has a size of $217,277_{10}$ bytes. To represent this size in an object header, imagine representing this number in binary. Just as in decimal, the rightmost digit has the lowest place value.

decimal	217277
binary	110101000010111101

Now, we'll take this binary number and split it into sections, starting with the least-significant (rightmost) bits. The first group consists of four bits while all subsequent groups contain up to seven bits. This produces three smaller binary numbers.

1101010	0001011	1101
---------	---------	------

¹⁴I will be using the binary definition of *kilobyte* as 1,024 bytes. This is also known as a *kibibyte*, abbreviated as KiB.

¹⁵<https://git-scm.com/docs/pack-format>

¹⁶See Section 6.2, “Inspecting `.git/index`”

¹⁷https://en.wikipedia.org/wiki/Variable-length_quantity

Next, we swap the order of these numbers; the encoding being little-endian means the first byte encodes the least-significant bits.

```
1101      00010111 1101010
```

Then we pad each one to eight bits, setting the most-significant bit to 0 if this number is the last one, and 1 otherwise. This leading 1 allows the format to extend to an arbitrary size; the reader keeps parsing until it sees a byte less than 80_{16} .

```
100011011 100010111 01101010
```

Finally, we store the type of the blob (3_{10} , or 11_2) in the second, third and fourth most-significant bits of the first number. That is, **10001101** becomes **10111101**.

binary	10111101	10001011	01101010
decimal	189	139	106
hexadecimal	bd	8b	6a

Putting all this together we see that this object's header can be encoded as the three bytes bd 8b 6a.

Computers don't typically provide access to the physical bits within a byte; a byte is the smallest unit of memory and we cannot speak of what order the bits are physically ordered within them. However, when performing bitwise operations, numbers work as though the bits are ordered with the most-significant on the left and the least-significant on the right.

So, we can select the least-significant four bits of a number by computing the bitwise-and $n \& 0xf$; F_{16} is 1111_2 . We can select the next seven bits by right-shifting the number four places and then using a bitwise-and against $7F_{16}$, which is 1111111_2 . That is, $(n >> 4) \& 0x7f$. To set the three type bits in the leading byte, we left-shift the type four places and bitwise-or it into the byte, i.e. $\text{byte} |= \text{type} << 4$. Further information on bitwise arithmetic can be found in Appendix B, *Bitwise arithmetic*.

27.4.1. Writing packs

We'll start with the code to generate a pack, which `git-upload-pack` will use to send data. To begin, with, we'll need a few constants that define the pack header format and the numeric type values for commits, trees and blobs.

```
# lib/pack.rb

module Pack
  HEADER_SIZE    = 12
  HEADER_FORMAT  = "a4N2"
  SIGNATURE      = "PACK"
  VERSION        = 2

  COMMIT = 1
  TREE   = 2
  BLOB   = 3
end
```

Now let's build a class whose job it is to convert a set of objects into a pack to send over the network. Our `Pack::Writer` class will be created with an output `IO` object, a database, and some options. It needs access to the database because we're only going to be giving it object

references from a `RevList` object search, and it will need to retrieve the full data for each object. When we create this object we'll also create a `Digest::SHA1` value, and everything we write to the output stream will also be added to this digest so we can append the SHA-1 hash at the very end. Finally, Git allows the compression level of packs to be configured, so we'll add support for that as an option.

```
# lib/pack/writer.rb

module Pack
  class Writer

    def initialize(output, database, options = {})
      @output = output
      @digest = Digest::SHA1.new
      @database = database

      @compression = options.fetch(:compression, Zlib::DEFAULT_COMPRESSION)
    end

    private

    def write(data)
      @output.write(data)
      @digest.update(data)
    end

    #
  end
end
```

The main method in this class will be `write_objects`. This takes a `RevList` and goes through several stages to generate the pack data. First, we prepare the *pack list*, which is the set of objects we want to send. Having found all the objects, we can then write the header with the object count, and then write all the object records. Finally, we append the SHA-1 digest of all the above content.

```
# lib/pack/writer.rb

def write_objects(rev_list)
  prepare_pack_list(rev_list)
  write_header
  write_entries
  @output.write(@digest.digest)
end
```

The `prepare_pack_list` method must build a list of all the records we want to send by enumerating the `RevList` it's given. Recall from Section 26.4, “Finding objects” that this class yields `Database::Commit` and `Database::Entry` objects to the caller. All we will do with this information for now is convert each of these into our own `Entry` type that retains the object's ID and its numeric type.

```
# lib/pack/writer.rb

Entry = Struct.new(:oid, :type)
```

```

def prepare_pack_list(rev_list)
  @pack_list = []
  rev_list.each { |object| add_to_pack_list(object) }
end

def add_to_pack_list(object)
  case object
  when Database::Commit
    @pack_list.push(Entry.new(object.oid, COMMIT))
  when Database::Entry
    type = object.tree? ? TREE : BLOB
    @pack_list.push(Entry.new(object.oid, type))
  end
end

```

Note that `Pack::Writer` is not concerned with how the objects to send are selected. It's the job of the caller to create a `RevList` with the right parameters to generate the objects it wants. The `RevList` class will however make sure that no object is emitted more than once.

Once all the entries are loaded, we can write out the header, which requires using `HEADER_FORMAT` to pack the signature, version and object count into the first 12 bytes.

```

# lib/pack/writer.rb

def write_header
  header = [SIGNATURE, VERSION, @pack_list.size].pack(HEADER_FORMAT)
  write(header)
end

```

Finally, we can write out all the object entries. The format merely concatenates every record end-to-end, compressing each object's content and prefixing it with the type and size header. This method relies on a couple of new interfaces; `Database#load_raw` loads an object without parsing its content, while `Numbers::VarIntLE` takes care of encoding the size as a variable-length integer; this array of bytes is then turned into a string by `pack("C*")`. We'll take a look at those methods below.

```

# lib/pack/writer.rb

def write_entries
  @pack_list.each { |entry| write_entry(entry) }
end

def write_entry(entry)
  object = @database.load_raw(entry.oid)

  header = Numbers::VarIntLE.write(object.size)
  header[0] |= entry.type << 4

  write(header.pack("C*"))
  write(Zlib::Deflate.deflate(object.data, @compression))
end

```

The `Database#load_raw` method is like `Database#load`, except it skips the work of parsing the object into a Commit, Tree or Blob. That's because `Pack::Writer` just wants to write the serialised object directly to the output stream and doesn't actually care about its type or internal structure — it's just a blob of data. So it would be pointless to parse the object only to re-

serialise it. This method will read the object's header and then return a `Raw` value containing the type, size, and unparsed data.

```
# lib/database.rb

Raw = Struct.new(:type, :size, :data)

def load_raw(oid)
  type, size, scanner = read_object_header(oid)
  Raw.new(type, size, scanner.rest)
end
```

`read_object_header` is a method extracted from the existing `read_object` method¹⁸, which currently looks like this:

```
# lib/database.rb

def read_object(oid)
  data    = Zlib::Inflate.inflate(File.read(object_path(oid)))
  scanner = StringScanner.new(data)

  type   = scanner.scan_until(/\ /).strip
  _size = scanner.scan_until(/\0/)[0...2]

  object = TYPES[type].parse(scanner)
  object.oid = oid

  object
end
```

We can split this method into two pieces: `read_object_header` inflates the object read from disk, and parses the type and size from the beginning of it, returning those two values and the `StringScanner`. `read_object` then calls this and further processes the object's content by parsing it with the appropriate type.

```
# lib/database.rb

def read_object(oid)
  type, _, scanner = read_object_header(oid)

  object = TYPES[type].parse(scanner)
  object.oid = oid

  object
end

def read_object_header(oid, read_bytes = nil)
  path    = object_path(oid)
  data    = Zlib::Inflate.new.inflate(File.read(path, read_bytes))
  scanner = StringScanner.new(data)

  type = scanner.scan_until(/\ /).strip
  size = scanner.scan_until(/\0/)[0...2].to_i

  [type, size, scanner]
end
```

¹⁸Section 10.1, “Reading from the database”

The final ingredient in writing out a pack is the `Numbers::VarIntLE` module that's called from `Writer#write_entry`. Short for *variable-length integer, little-endian*, it takes care of encoding an object's size value as a series of bytes. The first byte contains the least-significant four bits of the value, the next byte the next seven bits, and so on. Each byte other than the final one has the most-significant bit set, which is accomplished by a bitwise-or with 80_{16} (10000000_2). We return the bytes as an array rather than packing them into a string immediately because the caller will need to add the type value into the first byte.

```
# lib/pack/numbers.rb

module Pack
  module Numbers

    module VarIntLE
      def self.write(value)
        bytes = []
        mask = 0xf
        shift = 4

        until value <= mask
          bytes << (0x80 | value & mask)
          value >>= shift

          mask, shift = 0x7f, 7
        end

        bytes + [value]
      end
    end
  end
end
```

The initial value of `mask`, F_{16} , selects the least-significant four bits of the value, after which we right-shift it four places. The subsequent `mask` value of $7F_{16}$ selects the least-significant seven bits. We repeat this process until the value is less than or equal to the mask, meaning it will fit into a single byte.

27.4.2. Reading from packs

On the other end of the connection, `fetch` will be receiving the pack sent by `git-upload-pack` and will need to parse it, extracting each object and saving it into the `.git/objects` database. To achieve this, let's create a `Pack::Reader` class that takes an input stream. The `input` argument will be some kind of Ruby `IO`¹⁹, some object that responds to `read`, `readbyte`, `seek` and so on.

```
# lib/pack/reader.rb

module Pack
  class Reader

    def initialize(input)
      @input = input
    end
  end

```

¹⁹<https://docs.ruby-lang.org/en/2.3.0/IO.html>

```
# ...
end
end
```

The caller will instantiate `Pack::Reader` with an input stream, and then call `read_header` to find out how many objects are in the pack. Depending on how many there are, it will either want to extract the objects into individual files, or keep the pack on disk and generate an index for it²⁰.

```
# lib/pack/reader.rb

InvalidPack = Class.new(StandardError)

attr_reader :count

def read_header
  data = @input.read(HEADER_SIZE)
  signature, version, @count = data.unpack(HEADER_FORMAT)

  unless signature == SIGNATURE
    raise InvalidPack, "bad pack signature: #{signature}"
  end

  unless version == VERSION
    raise InvalidPack, "unsupported pack version: #{version}"
  end
end
```

`read_header` uses `String#unpack` to parse the first 12 bytes from the input, and it checks that the signature and version are what it's expecting.

After reading the header, the caller will want to read each object from the pack. This requires first reading its type/size header, which we can do by calling `Numbers::VarIntLE.read`, defined below. This returns the first byte of the encoding, and the size value, and we extract the type from the first byte. For now, we won't need the size value.

```
# lib/pack/reader.rb

def read_record
  type, _ = read_record_header
  Record.new(TYPE_CODES.key(type), read_zlib_stream)
end

def read_record_header
  byte, size = Numbers::VarIntLE.read(@input)
  type = (byte >> 4) & 0x7

  [type, size]
end
```

To get the object's data we need to read the zlib-compressed content following the header, but unfortunately the size header represents the record's size before it was compressed, not the length of the data stored in the pack. So we can't just read `size` bytes from the input and

²⁰<db_packs>>

then inflate it. Instead we need to read until zlib tells us the stream is finished. This presents a challenge: we don't know how long the zlib stream is, and unless we reimplement the zlib parser we've no way of detecting where it ends. We need to let zlib tell us where the object ends so we begin reading the next object from the correct position.

The simplest thing to do here would be to read a single byte at a time from `@input`, feed it into a `Zlib::Inflate` stream, and check whether the stream is finished. But drip-feeding data into zlib in a loop is extremely slow, and it would be better to pass bigger chunks of input, letting zlib's C code do more work for each iteration of our loop. Inspecting `.git/objects` tells us most of our objects are a few hundred bytes when compressed, so let's say we'll read input in 256-byte chunks.

This makes a substantial performance difference but presents two new challenges. The first is over-fetching: the last block we read for each zlib stream will contain the end of the stream and the beginning of the next object. To make sure we read the next object correctly, we need to reposition the read cursor. We can find out how much data zlib actually consumed from `Zlib::Inflate#total_in`²¹, and then we can use `IO#seek`²² to move the read cursor by the amount we've over-fetched.

The second problem is that if we're reading near the end of the pack, there may be fewer than 256 bytes left to read. The peer that's sending the pack may not explicitly close its output stream at the end of the pack—it just stops sending data but leaves the stream open. The `IO#read`²³ method blocks until the requested number of bytes are available, so if there are fewer than 256 bytes left in the pack, calling `read` will just hang indefinitely. Instead we can use `IO#read_nonblock`²⁴, which may return fewer bytes than requested and raises an exception rather than blocking if there is no data available.

Putting all this together, we can use `IO#read_nonblock` in a loop, feeding the data into `Zlib::Inflate`. We don't mind if it gives us less data than we asked for, as we don't actually know how long the zlib block is; we just want to read it in chunks rather than a byte at a time. Once `Zlib::Inflate#finished?` is true, we `IO#seek` backwards by the amount we've over-fetched, ready to read the next object.

```
# lib/pack/reader.rb

def read_zlib_stream
  stream = Zlib::Inflate.new
  string = ""
  total = 0

  until stream.finished?
    data = @input.read_nonblock(256)
    total += data.bytesize

    string.concat(stream.inflate(data))
  end
  @input.seek(stream.total_in - total, IO::SEEK_CUR)

  string
```

²¹https://docs.ruby-lang.org/en/2.3.0/Zlib/ZStream.html#method-i-total_in

²²<https://docs.ruby-lang.org/en/2.3.0/IO.html#method-i-seek>

²³<https://docs.ruby-lang.org/en/2.3.0/IO.html#method-i-read>

²⁴https://docs.ruby-lang.org/en/2.3.0/IO.html#method-i-read_nonblock

```
end
```

Record and TYPE_CODES exist to convert the pack record and its numeric type into an object we can save to the database. In Section 3.2, “The commit command”, we give the Blob, Tree and Commit classes the methods type and to_s, and Database#store would then use these methods to write the object to disk. Record plays the same role here; it exposes a type and to_s that allow the object to be sent to Database#store and get written to .git/objects. Again, Reader does not parse these objects as it doesn’t care about their internal structure; it just yields them as blob-like values back to the caller, which can process the object further if it likes.

```
# lib/pack.rb

TYPE_CODES = {
  "commit" => COMMIT,
  "tree"   => TREE,
  "blob"   => BLOB
}

Record = Struct.new(:type, :data) do
  attr_accessor :oid

  def to_s
    data
  end
end
```

Finally, we need Numbers::VarIntLE.read to parse the size from the input stream. From the first byte, we bitwise-and with F₁₆ to select the lowest four bits. On the next byte, we mask off seven bits using 7F₁₆ and shift the result left by four bits before combining it with the accumulating value. The left-shift amount increases by 7 bits for each extra byte we read, so the bytes are contributing increasingly significant bits to the result. We return the value with the first byte, so we can read the type from it.

```
# lib/pack/numbers.rb

module VarIntLE
  def self.read(input)
    first = input.readbyte
    value = first & 0xf
    shift = 4

    byte = first

    until byte < 0x80
      byte   = input.readbyte
      value |= (byte & 0x7f) << shift
      shift += 7
    end

    [first, value]
  end
end
```

For example, let’s take the example size header we generated earlier:

hexadecimal	bd	8b	6a
-------------	----	----	----

decimal	189	139	106
binary	10111101	10001011	01101010

From the first byte we select the lowest four bits; $10111101_2 \& F_{16} = 1101_2$.

1101

The most-significant bit of this byte is 1, so we continue and read the next byte. A bitwise-and with $7F_{16}$ selects the bits 0001011_2 , and we left-shift this four places and bitwise-or it with our result:

0001011 1101

Again, this byte has its most-significant bit set, so we move onto the next. This yields the bits 1101010_2 , and the left-shift is now eleven places:

1101010 0001011 1101

This gives us the bits 110101000010111101_2 , which is $217,277_{10}$. The third byte is less than 80_{16} so we can stop parsing here.

27.4.3. Reading from a stream

The `Pack::Reader` class will initially be used to read from the stream sent by the remote process — the `upload-pack` command that's generating the pack. But later we'll also use it to read from files on disk. In Unix-like systems, everything you can read data from is presented as a file²⁵, which really means you use the same set of functions to interact with any data source. In Ruby, this set of functions is the `IO` interface.

When we read a pack from a file, we'll be wanting to read specific objects from arbitrary locations, rather than read the entire pack contents sequentially. When we're reading all the objects sequentially during a `fetch` command, we'll want to perform some additional activities as the data is read: we'll need to calculate the SHA-1 hash of all the pack data so we can verify it at the end, and we'd like to track the amount of data we've read so we can index the position of each object and feed it into a progress meter²⁶. These additional services aren't needed when reading from a file, so in that case we can pass in a `File`²⁷ returned by `File.open` and that will work fine: `read` pulls data from the current position, and `seek` moves the read position. But when reading from an `upload-pack` output stream, we can wrap the `IO` stream in our own class.

A basic class that replicates the `read` and `readbyte` methods while keeping a SHA-1 hash and offset up-to-date might look like this. It takes some `IO` object as input, and wraps its `read` method so that when something calls `Stream#read`, it receives the data from the underlying `IO` and the `Stream` updates its state transparently.

```
# lib/pack/stream.rb

module Pack
  class Stream
```

²⁵https://en.wikipedia.org/wiki/Everything_is_a_file

²⁶Section 29.3, “Progress meters”

²⁷<https://docs.ruby-lang.org/en/2.3.0/File.html>

```
attr_reader :digest, :offset

def initialize(input)
  @input = input
  @digest = Digest::SHA1.new
  @offset = 0
end

def read(size)
  data = @input.read(size)
  update_state(data)
  data
end

def readbyte
  read(1).bytes.first
end

private

def update_state(data)
  @digest.update(data)
  @offset += data.bytesize
end

end
end
```

This is an example of the *decorator pattern*²⁸, wherein one object is wrapped by another that has the same interface, but supplies additional behaviour. We can pass in a `Pack::Stream` to any method that expects an `IO` to read from and it will work just the same. This pattern is an example of *polymorphism*²⁹: multiple classes implement the same interface so they can all interoperate with something that relies on said interface.

We can extend the `Stream` interface with other operations we need. To check the SHA-1 hash at the end of the pack, we need to read directly from the underlying `IO`, rather than calling the `Stream#read` method as that will update the computed SHA-1 hash.

```
# lib/pack/stream.rb

def verify_checksum
  unless @input.read(20) == @digest.digest
    raise InvalidPack, "Checksum does not match value read from pack"
  end
end
```

Our `Stream` class can accept calls to `read` and `readbyte`, but not the methods used in `Reader#read_zlib_stream`: `read_nonblock` and `seek`. If we want to use `Stream` with `Reader`, it needs to implement all the methods `Reader` depends on. These methods present a problem: when reading from a file, all the data is sitting around motionless on disk, and `IO#seek` can jump to arbitrary locations in the file so we can read the file in any order we like. When reading from a stream, like a process's standard output, `IO#seek` doesn't work. The `IO` class and the underlying kernel infrastructure don't keep hold of data you've already read — once you fetch

²⁸https://en.wikipedia.org/wiki/Decorator_pattern

²⁹[https://en.wikipedia.org/wiki/Polymorphism_\(computer_science\)](https://en.wikipedia.org/wiki/Polymorphism_(computer_science))

it with `read` it's gone, and you can't move the cursor back to an earlier point in the stream. If you want that behaviour, you need to implement it yourself.

Now, we also don't want to keep the entire stream contents in memory, as this could be very large. But as we'll see when we come to store packs on disk, we will need to parse out the chunk of the stream that corresponds to a single record. Implementing that will also let us retain data we've read recently, so that we can put it back in the queue when we detect we've read past the end of an object.

To support this requirement, we'll add two new pieces of state to `Pack::Stream`. `@buffer` will store data that's already been fetched from the underlying `IO` but was not part of the current object, so has been stored for future reads. `@capture` will store data that's read during the current record, so we can retrieve the excess and add it to `@buffer` when the end of the object is detected. `Encoding::ASCII_8BIT` is Ruby's way of denoting that a string contains binary data rather than text.

```
# lib/pack/stream.rb

def initialize(input)
  @input    = input
  @digest   = Digest::SHA1.new
  @offset   = 0
  @buffer   = new_byte_string
  @capture  = nil
end

def new_byte_string
  String.new("", :encoding => Encoding::ASCII_8BIT)
end
```

We need to implement a method of capturing all the data that was read while parsing an object, trimming off any data we over-fetched. Ideally we'd like to do this without reaching inside the `Reader` class and adding lots of instrumentation that makes it incompatible with other `IO` types. A common way of doing this in Ruby is to use blocks to cause something to monitor activity while another piece of code runs. For example, an interface for fetching a `Pack::Record` and the underlying byte data could look like this:

```
stream = Pack::Stream.new(io)
reader = Pack::Reader.new(stream)

reader.read_header

record, data = stream.capture { reader.read_record }
```

Since `Reader` will be calling methods on `Stream`, `Stream` can monitor any reads that occur while `Reader#read_record` is running, and retain the data that contributes to that object, returning it along with the record itself. To make this work, we instantiate `@capture` to a new string, then yield to execute the block, placing the result and the `@capture` buffer in the return value together. After yielding, we update the SHA-1 hash with the captured buffer, and delete the reference to it, before returning the results.

```
# lib/pack/stream.rb

def capture
```

```
    @capture = new_byte_string
    result   = [yield, @capture]

    @digest.update(@capture)
    @capture = nil

    result
end
```

Now, when we call `update_state` after reading from the `IO`, we don't necessarily want to update the SHA-1 hash immediately. If we over-fetch, we might end up reading the checksum at the end of the file, and feeding that into `@digest` would give us the wrong result. So if we're currently capturing reads, we hold off on updating `@digest` and leave that for the `capture` method to handle. If `@capture` exists, then we append the fetched data to it.

```
# lib/pack/stream.rb

def update_state(data)
  @digest.update(data) unless @capture
  @offset += data.bytesize
  @capture & .concat(data)
end
```

We now have enough information to make `seek` work. At the end of each record, `Reader#read_zlib_stream` will call `seek` to tell the `IO` object to move the cursor backwards by some amount. We can handle that by slicing off the given amount at the end of `@capture`, and storing it in `@buffer`. `@capture` will then contain exactly the data that backs the current object and nothing more. We also amend `@offset` in line with the adjustment.

```
# lib/pack/stream.rb

def seek(amount, whence = IO::SEEK_SET)
  return unless amount < 0

  data = @capture.slice!(amount .. -1)
  @buffer.prepend(data)
  @offset += amount
end
```

Next, `Stream` needs to pull data from `@buffer` before attempting to fetch more data from `@input`. A new method called `read_buffered` will wrap this operation, slicing as much as it can off the front of the `@buffer` string before fetching any remaining bytes from `@input`. If we want a blocking read for a fixed number of bytes, we call `read`, otherwise we call `read_nonblock`. This could throw an error if there's no data available (`EWOULDBLOCK`) or the stream has been closed (`EOFError`), and in those cases we'll just return what we managed to read from the buffer.

```
# lib/pack/stream.rb

def read_buffered(size, block = true)
  from_buf = @buffer.slice!(0, size)
  needed   = size - from_buf.bytesize
  from_io  = block ? @input.read(needed) : @input.read_nonblock(needed)

  from_buf.concat(from_io.to_s)

rescue EOFError, Errno::EWOULDBLOCK
```

```
    from_buf  
end
```

Stream#read and Stream#read_nonblock can both be implemented on top of read_buffered, and now we have the complete IO interface needed to support the Reader class.

```
# lib/pack/stream.rb  
  
def read(size)  
  data = read_buffered(size)  
  update_state(data)  
  data  
end  
  
def read_nonblock(size)  
  data = read_buffered(size, false)  
  update_state(data)  
  data  
end
```

This design separates concerns: Reader relies only on the IO interface and can use any source of data as input. Stream doesn't know anything about the format being parsed, and can provide these state-tracking services to anything that uses the IO interface.

For now, this pack format is sufficient to support sending commits, trees and blobs. We will later extend this to support delta compression and make the resulting packs much smaller, but first let's use this format to get a working fetch command up and running.

28. Fetching content

We have now defined all the networking machinery we need to transfer messages and objects from one repository to another: the `Remotes::Protocol` class wraps around a process's standard I/O and lets us exchange messages, while the `Pack` module sends objects from the sender (`git-upload-pack`) to the receiver (the `fetch` command). In this chapter, we'll walk through a minimal implementation of the `fetch` and `upload-pack` commands that are just enough to work together and to interoperate with the Git software.

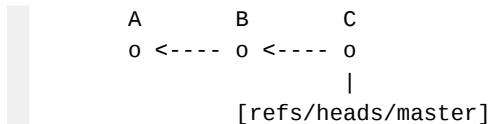
Remember throughout that the purpose of the `fetch` command is to form a copy of the remote repository's objects in our own local database, by downloading any objects we don't yet have and copying the remote's `refs/heads` pointers into our `refs/remotes` directory. We can then use these remote refs to merge our branches.

28.1. Pack negotiation

At the core of the logic of `fetch` and `upload-pack` is a conversation that should produce the smallest possible pack to send over the network. We would rather not send the entire history of the repository over the network each time, as this would be wasteful and become slower as the repository grows. We'd like to restrict the objects we send to those the receiver does not already have, and `fetch` and `upload-pack` need to have a conversation to negotiate which objects those are.

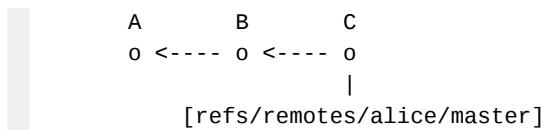
As an example, let's imagine that Alice has a repository containing three commits, where her `refs/heads/master` points at the last one in the chain.

Figure 28.1. Alice's initial repository



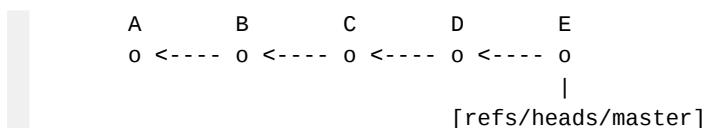
Bob has a copy of all of Alice's objects from a previous `fetch`. As Bob begins with an empty repository, let's assume this works by literally copying Alice's `.git/objects` directory into Bob's repository, so he has an exact copy of the same history. He has a remote ref, `refs/remotes/alice/master`, that mirrors Alice's `master` branch.

Figure 28.2. Bob's copy of Alice's repository



Next, suppose Alice makes a few more commits on top of her `master` branch.

Figure 28.3. Alice extends the master branch



Bob wants to fetch from Alice again. Ideally, we would like to send only commits *D* and *E* over the network, as Bob already has *C* and all its ancestors. Here's how `fetch` and `upload-pack` talk to each other to achieve this.

Suppose Bob has the `remote.alice.fetch` variable for Alice's repository set to `+refs/heads/*:refs/remotes/alice/*`, so he wants to copy all Alice's `refs/heads` references into `refs/remotes/alice`. When Bob runs `fetch`, that invokes `upload-pack` on Alice's repository. The first thing `upload-pack` does is send back a list of all Alice's refs and what they point at.

```
upload-pack:
<ID of commit E> refs/heads/master
```

Bob now knows Alice has a ref called `refs/heads/master`, and this matches his `refspec` with source `refs/heads/*`. He looks at the ID associated with this ref and sees that he does not have it, so he sends a message to Alice saying he wants it. After sending all the commit IDs he wants, he sends a flush-packet.

```
fetch:
want <ID of commit E>
(flush)
```

Bob cannot see the whole history leading to *E*, he just knows he doesn't have that commit and so wants to receive everything it points at that he does not already have. Now, if all Alice knows is that Bob wants *E*, she could send the entire history reachable from *E*, but that would be wasteful. In order to generate a minimal pack, she'd like to know which commits Bob already has. Therefore, Bob also sends a list of the commits his refs point at. His `refs/remotes/alice/master` ref points at *C*, so he sends that, followed by the message `done`.

```
fetch:
have <ID of commit C>
done
```

Now, Alice can use `RevList` to enumerate the range *C..E*, including their tree and blob objects, and she puts all those objects into a pack and sends them to Bob. Bob now has the complete history:

Figure 28.4. Bob's updated database

```
A      B      C      D      E
o <---- o <---- o <---- o <---- o
          |
[refs/remotes/alice/master]
```

Finally, he updates his remote ref. Since *C* is an ancestor of the requested commit *E*, this update is a *fast-forward*.

Figure 28.5. Bob's updates his remote reference

```
A      B      C      D      E
o <---- o <---- o <---- o <---- o
          |
[refs/remotes/alice/master]
```

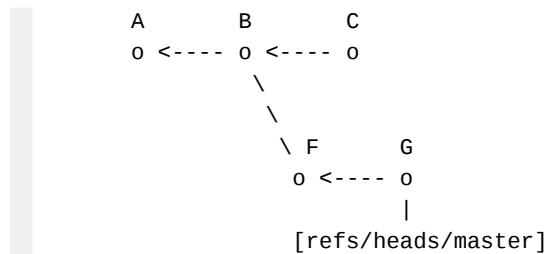
Bob's repository once again reflects the state of Alice's, and he can merge her commits with his own using the `merge` or `cherry-pick` commands.

This process works because of the consistency of object IDs across repositories. Since each object's ID is a function of its content, the same content will have the same ID in every repository on any machine, and so Bob can say he has the ID of commit *C* and Alice knows what content that refers to. The parties only need to exchange commit IDs to understand which parts of the graph they each have. We also assume that if a peer says they have a certain commit, then they have everything that commit points to — its tree, its parents, and all their history back to the root commit. For a single repository, this should be true as each commit is based on another one that's already present in the database. To maintain this assumption with distributed repositories, the protocol must always transfer all the commits necessary to complete the requested range, such that after `fetch` is finished, the receiver has the commits it sent as `want` requests, including all their content and history.

28.1.1. Non-fast-forward updates

In the above example, Alice added new commits based on a commit Bob already had. But what happens if she discards some of her history and starts developing off an older commit? Let's say she drops commit *C* and begins a new history for her `master` branch based on *B*.

Figure 28.6. Alice's rewritten history



This time `upload-pack` will send the following:

```
upload-pack:  
<ID of commit G> refs/heads/master
```

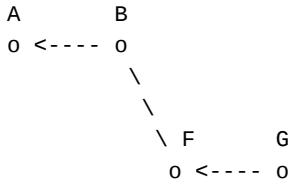
Assuming Bob has commit *C* and its history from a previous fetch, he will send the same `want` and `have` sets; he doesn't know how *G* is related to *C*, he just knows he doesn't have it.

```
fetch:  
want <ID of commit G>  
(flush)  
have <ID of commit C>  
done
```

If Alice still has commit *C* in her database, this is enough to minimise the pack: the range *C..G* gives commits *G* and *F*. Ordinarily, a repository will not manipulate its remote refs and they should always reflect a commit that was received from the remote. But it's possible that *C* no longer exists in Alice's repository, either because she ran a garbage-collection task to delete

unreachable objects, or she deleted her repository and re-cloned it using a method that only fetches reachable objects.

Figure 28.7. Alice's database with unreachable commits removed



In this case, Bob sending a message saying he has commit *C* is useless, because Alice no longer knows what that is. The range *C..G*, ignoring missing objects, will yield *G*, *F* and all their ancestors.

To try to prevent this happening, Bob could instead send a list of all the reachable commits he has; not just *C* but all the ancestors of *C*:

```
fetch:  
  
want <ID of commit G>  
(flush)  
have <ID of commit C>  
have <ID of commit B>  
have <ID of commit A>  
done
```

This may be inefficient, since Alice still has *B* and so sending ancestors of *B* is redundant. But, Bob sending a list of commit IDs will require much less data transfer than Alice sending the same commits as complete objects, including their trees and blobs. It will usually be more efficient for Bob to send all his commit IDs as have messages than for Alice to send the entire project history because she could not find a common ancestor between the want and have commits.

Alice can now run a RevList query with the want commits as included inputs, and the have commits as excluded inputs, that is, *G*, $\wedge C$, $\wedge B$, $\wedge A$, which yields commits *G* and *F* to be sent in the pack.

To further mitigate the cost of sending the have list, upload-pack supports a capability called `multi_ack`. If this is enabled, it means Bob can send a few have messages and then a flush-packet, and the remote agent will respond to say whether it recognises any of that batch of commits. If not, it sends the message NAK. If it recognises a commit, but requires more input to find a common ancestor for all the want commits, it sends ACK `<ID> continue`. And if it has all the commits it needs, it sends ACK `<ID>` for every further commit the client sends.

So in our example, Bob could send the first couple of his commits, and then a flush-packet:

```
fetch:  
  
want <ID of commit G>  
(flush)  
have <ID of commit C>  
have <ID of commit B>
```

```
(flush)
```

Since Alice has *B* and it's an ancestor of the requested commit *G*, she sends an ACK. Bob now knows he does not need to send any further commits, and can immediately send the done message and wait for the pack Alice will send.

```
fetch:          upload-pack:  
done           ACK <ID of commit B>
```

Git implements this feature by sending have commits in blocks of 32, but to get to a working implementation as soon as possible, we will not be implementing this optimisation. Further information about the protocol capabilities¹ and the pack protocol as a whole² is available in Git's documentation.

28.2. The fetch and upload-pack commands

We'll begin the `fetch` command by defining some options. The `--force` option is used in conjunction with `refsspecs`; if we fetch using a `refspec` with no leading `+` sign, this option makes `fetch` treat the `refspec` as forced anyway. The `--upload-pack` option sets the name of the remote agent program to run. So that we can interoperate with Git, this will default to `git-upload-pack`, but if we want to test our own remote agent implementation, we can pass `--upload-pack="jit upload-pack"`.

```
# lib/command/fetch.rb  
  
module Command  
  class Fetch < Base  
  
    def define_options  
      @parser.on("-f", "--force") { @options[:force] = true }  
  
      @parser.on "--upload-pack=<upload-pack>" do |uploader|  
        @options[:uploader] = uploader  
      end  
    end  
  
    # ...  
  end  
end
```

The `Command::Fetch#run` method outlines the overall operation of the `fetch` command. Some of the methods it calls are imported from the modules `FastForward`, `RemoteClient` and `ReceiveObjects`, which we'll define as we meet them. These methods will also be needed by the `push` and `receive-pack` commands, so I'm implementing them in shared modules here. The command begins by configuring itself — processing any command-line options and configurations to determine the location of the remote repository and which `refsspecs` we want to fetch. Then it starts the remote agent, the `git-upload-pack` process, using the name of the current command, the agent program name it has detected, and the URL.

¹<https://git-scm.com/docs/protocol-capabilities>

²<https://git-scm.com/docs/pack-protocol>

Once the connection is made, we go through a number of steps. First, we receive the references from the remote — every remote agent invocation begins by listing all the references in the remote repository and their commit IDs. As we do this, we'll compare this list to our refsspecs and determine which refs we want to download.

Once we've done that, we send a list of the commit IDs we want, and a list of the commit IDs we already have. From these lists, the remote agent will determine which commits it needs to send to complete our view of the history, and generate a pack containing those commits. We'll receive this pack, and then update our refs/remotes references to reflect the new content we've downloaded. Don't worry if this still feels vague --- we'll dig into each step of this process throughout this chapter.

```
# lib/command/fetch.rb

include FastForward
include ReceiveObjects
include RemoteClient

def run
  configure
  start_agent("fetch", @uploader, @fetch_url)

  recv_references
  send_want_list
  send_have_list
  recv_objects
  update_remote.refs

  exit (@errors.empty? ? 0 : 1)
end
```

The upload-pack command mirrors the sequence of steps in fetch. It accepts the client's connection, passing in its own name that will be passed to Remotes::Protocol. It sends the state of its references, and then receives the want and have lists from the client. It ends by generating and sending a pack of all the objects the client needs.

```
# lib/command/upload_pack.rb

module Command
  class UploadPack < Base

    include RemoteAgent
    include SendObjects

    def run
      accept_client("upload-pack")

      send_references
      recv_want_list
      recv_have_list
      send_objects

      exit 0
    end

    # ...
  end
end
```

```
    end  
end
```

accept_client is defined in the RemoteAgent module, and creates a Remotes::Protocol wrapper around the upload-pack process's standard input and output streams.

```
# lib/command/shared/remote_agent.rb  
  
module Command  
  module RemoteAgent  
  
    def accept_client(name, capabilities = [])  
      @conn = Remotes::Protocol.new(name, @stdin, @stdout, capabilities)  
    end  
  
    # ...  
  
  end  
end
```

Throughout the rest of this section we'll walk through each step of this conversation between the two commands. fetch begins by calling its configure method, which determines the URL of the remote repository and the list of refsspecs we want to use. The first argument to fetch can either be a literal URL, or the name of a remote in .git/config. The refsspecs are given either by the rest of the arguments, or by the remote.<name>.fetch config variable. If no arguments are given, we assume the default remote name, origin.

```
# lib/command/fetch.rb  
  
UPLOAD_PACK = "git-upload-pack"  
  
def configure  
  name = @args.fetch(0, Remotes::DEFAULT_REMOTE)  
  remote = repo.remotes.get(name)  
  
  @fetch_url = remote.fetch_url || @args[0]  
  @uploader = @options[:uploader] || remote.uploader || UPLOAD_PACK  
  @fetch_specs = (@args.size > 1) ? @args.drop(1) : remote.fetch_specs  
end
```

28.2.1. Connecting to the remote

After configure has run, we initiate a connection with the remote repository by calling start_agent, defined in the RemoteClient module. For now, we'll only make this work with repositories on the same machine, with URLs like file:///home/username/path/to/repo. This builds a command string like "git-upload-pack /home/username/path/to/repo" and runs this command using Open3. We use the Shellwords³ module to make sure any arguments containing spaces and other special characters are properly quoted for the shell.

```
# lib/command/shared/remote_client.rb  
  
module Command  
  module RemoteClient
```

³<https://docs.ruby-lang.org/en/2.3.0/Shellwords.html>

```
def start_agent(name, program, url, capabilities = [])
  argv = build_agent_command(program, url)
  input, output, _ = Open3.popen2(Shellwords.shelljoin(argv))
  @conn = Remotes::Protocol.new(name, output, input, capabilities)
end

def build_agent_command(program, url)
  uri = URI.parse(url)
  Shellwords.shellsplit(program) + [uri.path]
end

# ...

end
end
```

We wrap a `Remotes::Protocol` object around the I/O streams for this child process, and we're now ready to communicate with the remote repository.

28.2.2. Transferring references

Once the connection is established, the first part of the conversation consists of the remote agent sending its references to the client. The `upload-pack` command uses `RemoteAgent#send_references` to do this. This method gets a list of all the repository's refs from `Refs#list_all_refs`. Sorting them by their path, it sends the commit ID and path of each ref, if the ref points at a commit. If no refs were sent, we send the line `0000000... capabilities^{}}, which just exists as a first line of output to which to attach the repository's capability list. The client should ignore any refs with an all-zero commit ID. Once all the references have been sent, we send a flush-packet.`

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

ZERO_OID = "0" * 40

def send_references
  refs = repo.refs.list_all_refs
  sent = false

  refs.sort_by(&:path).each do |symref|
    next unless oid = symref.read_oid
    @conn.send_packet("#{ oid.downcase } #{ symref.path }")
    sent = true
  end

  @conn.send_packet("#{ ZERO_OID } capabilities^{})") unless sent
  @conn.send_packet(nil)
end
```

The above code uses the method `repo` to access the current `Repository` object. All the commands we've seen so far infer the path to the repository from the current working directory, and the `Command::Base#repo` method locates the repository on this assumption. However, remote agent commands like `upload-pack` do not have to be invoked with the repository as the current working directory. Instead, they are given the path to the repo as a command-line argument, and they need to infer the repository directory from that.

This argument might specify the `.git` directory itself, or the parent directory that contains both `.git` and the workspace. Git also supports *bare* repositories, which are just a `.git` directory with no workspace attached, and this repository can be named anything. So to infer the repository location from the argument, we check the given path and all its parent directories until we find one that contains something like looks like a Git repo — a file named `HEAD` and directories called `objects` and `refs` — or, contains a `.git` directory containing these files.

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

def repo
  @repo ||= Repository.new(detect_git_dir)
end

def detect_git_dir
  pathname = expanded_pathname(@args[0])
  dirs = pathname.ascend.flat_map { |dir| [dir, dir.join(".git")] }
  dirs.find { |dir| git_repository?(dir) }
end

def git_repository?(dirname)
  File.file?(dirname.join("HEAD")) and
  File.directory?(dirname.join("objects")) and
  File.directory?(dirname.join("refs"))
end
```

For example, if we're given the path `/home/jcoglan/jit`, we'd check that directory, and the paths `/home/jcoglan/jit/.git`, `/home/jcoglan/.git`, `/home/jcoglan/.git/.git`, `/home/.git`, `/home/.git/.git` and `/.git` until we find a directory containing `HEAD`, `objects` and `refs`.

On the client side, the `fetch` command uses `RemoteClient#recv_references` to read the refs sent by the remote agent, storing them in a hash mapping ref names to commit IDs. The zero ID is ignored if present.

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

REF_LINE = /^[0-9a-f]+) (.*)$/
ZERO_OID = "0" * 40

def recv_references
  @remote_refs = {}

  @conn.recv_until(nil) do |line|
    oid, ref = REF_LINE.match(line).captures
    @remote_refs[ref] = oid.downcase unless oid == ZERO_OID
  end
end
```

28.2.3. Negotiating the pack

The `fetch` command now needs to send information to `upload-pack` to indicate which objects it wants to receive, and what it already has. `send_want_list` deals with the first step. Supposing we have the following `refsspecs` and `refs` received from the remote:

```
@fetch_specs = ["+refs/heads/*:refs/remotes/origin/*"]
@remote_refs = {
```

```

    "HEAD"          => "5d826e972970a784bd7a7bdf587512510097b8c7",
    "refs/heads/maint" => "98cdfbb84ad2ed6a2eb43dafa357a70a4b0a0fad",
    "refs/heads/master" => "5d826e972970a784bd7a7bdf587512510097b8c7",
    "refs/heads/todo"   => "b2cc3488ba006e3ba171e85dffbe6f332f84bf9a"
  }
}

```

We need to build a list of the refs we'd like to download. Matching the refspecs against the names of the refs we've received will give us this data structure, which lists each ref that we'd like to update, along with its corresponding ref in the remote repo, and whether this update is forced.

```

@targets = {
  "refs/remotes/origin/maint" => ["refs/heads/maint", true],
  "refs/remotes/origin/master" => ["refs/heads/master", true],
  "refs/remotes/origin/todo"   => ["refs/heads/todo", true]
}

```

In `send_want_list`, we build a set of the object IDs we want by iterating over this structure, checking the remote's ID for each ref against our cached ID from the corresponding `refs/remotes` ref. Any commit IDs that differ from our own refs are added to the wanted set. Having constructed this set, we send each of its IDs to the remote, followed by a flush packet. If no want lines were sent, we exit immediately as there is nothing we need to download.

```

# lib/command/fetch.rb

def send_want_list
  @targets = Remotes::Refspec.expand(@fetch_specs, @remote_refs.keys)
  wanted   = Set.new

  @local_refs = {}

  @targets.each do |target, (source,_)|
    local_oid  = repo.refs.read_ref(target)
    remote_oid = @remote_refs[source]

    next if local_oid == remote_oid

    @local_refs[target] = local_oid
    wanted.add(remote_oid)
  end

  wanted.each { |oid| @conn.send_packet("want #{ oid }") }
  @conn.send_packet(nil)

  exit 0 if wanted.empty?
end

```

On the remote side in `upload-pack`, we read these want lines from the client. We read until we get a flush-packet, and match each line against the pattern `/^want ([0-9a-f]+)$/` to extract the commit ID. If we did not receive any wanted commits, we can immediately exit as there's nothing to send to the client.

```

# lib/command/upload_pack.rb

def recv_want_list
  @wanted = recv_oids("want", nil)
  exit 0 if @wanted.empty?

```

```
end

def recv_oids(prefix, terminator)
  pattern = /^#{ prefix } ([0-9a-f]+)$/
  result  = Set.new

  @conn.recv_until(terminator) do |line|
    result.add(pattern.match(line)[1])
  end
  result
end
```

Having sent its want list, the `fetch` command must now send its list of the commits it already has, and we can use the `RevList` class to do this. In Section 26.4, “Finding objects”, we added the `:all` option to make `RevList` search the history beginning with all the refs in the current repository. After sending all the commits and the done message, we’ll just skip over any messages the remote agent sends until we see the message `PACK`, indicating the start of the object data.

```
# lib/command/fetch.rb

def send_have_list
  options  = { :all => true, :missing => true }
  rev_list = ::RevList.new(repo, [], options)

  rev_list.each { |commit| @conn.send_packet("have #{ commit.oid }") }
  @conn.send_packet("done")

  @conn.recv_until(Pack::SIGNATURE) {}
end
```

Note that we’re not implementing the `multi_ack` optimisation here. We could interrupt sending the have list as soon as we detect an `ACK` message from the remote. However, this requires reading messages from the remote in a non-blocking manner while sending the have list, which is more complexity than I’d rather get into here. Do have a go at implementing this optimisation yourself, as it becomes more valuable as the repository gets larger.

To receive the have commit IDs, the `upload-pack` command can use the `recv_oids` method we defined above, just using the prefix “have” and the terminating message “done”. Since we’re not implementing any optimisation of the have negotiation, we’ll just send a `NAK` message rather than check for graph intersections so we can send `ACK` responses.

```
# lib/command/upload_pack.rb

def recv_have_list
  @remote_has = recv_oids("have", "done")
  @conn.send_packet("NAK")
end
```

This may seem like an oversight, but the `ACK` functionality is strictly an optimisation. If the client sends its entire history, we’ll still generate an optimal pack, we’ll just send more data than is strictly necessary during the negotiation phase. However, that data is small relative to the size of full objects, and is fairly quick to compute on the client site. This unoptimised implementation gets us to a working integration quicker, and as our repositories grow we can revisit the value of providing acknowledgement of have messages.

28.2.4. Sending object packs

Now upload-pack has all the information it needs to build a minimal set of objects and send them to the client. To calculate the set of commits the client wants, we can use its want IDs as the inputs to a RevList, along with the IDs from its have messages as excluded inputs, prefixed with `^`. This will generate the set of commits that are reachable from the want commits but not from the have commits, that is the set the client does not have yet.

```
# lib/command/upload_pack.rb

def send_objects
  revs = @wanted + @remote_has.map { |oid| "^#{ oid }" }
  send_packed_objects(revs)
end
```

`send_packed_objects` is a method in the shared `SendObjects` module. It takes a list of revisions and generates a pack containing all the commits identified by the revisions, including their trees and blobs that are not part of excluded commits. To set the compression level, it uses the `pack.compression` or `core.compression` setting, if present. The `Pack::Writer` class⁴ is given `@conn.output` as its output stream, so it will send data over the same channel that `Remotes::Protocol` was using.

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

module Command
  module SendObjects

    def send_packed_objects(revs)
      rev_opts = { :objects => true, :missing => true }
      rev_list = ::RevList.new(repo, revs, rev_opts)

      pack_compression = repo.config.get(["pack", "compression"]) ||
        repo.config.get(["core", "compression"])

      write_opts = { :compression => pack_compression }
      writer     = Pack::Writer.new(@conn.output, repo.database, write_opts)

      writer.write_objects(rev_list)
    end
  end
end
```

On the receiver side, the `fetch` command calls `recv_objects` to read the incoming pack and decode it into files in its local database. It passes `Pack::SIGNATURE` because we already read the bytes `PACK` from the input stream during `send_have_list`, and that needs to be routed to the pack reader so it can read the pack header correctly and include these bytes in its SHA-1 calculation to verify the received data.

```
# lib/command/fetch.rb

def recv_objects
  recv_packed_objects(Pack::SIGNATURE)
```

⁴Section 27.4.1, “Writing packs”

```
end
```

`recv_packed_objects` is defined in the shared module `ReceiveObjects`, and uses the `Pack::Reader`⁵ and `Pack::Stream`⁶ classes. For each object read, we save the object to our local database using `Database#store`.

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

module Command
  module ReceiveObjects

    def recv_packed_objects(prefix = "")
      stream = Pack::Stream.new(@conn.input, prefix)
      reader = Pack::Reader.new(stream)

      reader.read_header

      reader.count.times do
        record, _ = stream.capture { reader.read_record }
        repo.database.store(record)
      end
      stream.verify_checksum
    end
  end
end
```

The `prefix` argument is passed into `Pack::Stream`, which adds the given string to its initial buffered data so that this will be read before we start fetching more bytes from the `Remotes::Protocol` connection.

```
# lib/pack/stream.rb

def initialize(input, buffer = "")
  @input = input
  @digest = Digest::SHA1.new
  @offset = 0
  @buffer = new_byte_string.concat(buffer)
  @capture = nil
end
```

Recall from the previous chapter that `Pack::Reader#read_record` returns `Pack::Record` objects, not `Blob`, `Tree` or `Commit` objects from the `Database` namespace. These `Record` objects just expose `type` and `to_s` methods that return the right data to be written to disk correctly, without parsing their internal structure.

28.2.5. Updating remote refs

After saving all the packed objects to its local database, the `fetch` command's final task is to update the remote refs in its `refs/remotes` directory to reflect the refs that `upload-pack` initially sent. We stored those refs in `@remote_refs`, and a list of our own refs we want to update, with their current local values, in `@local_refs`. `@targets` contains a mapping from our

⁵Section 27.4.2, “Reading from packs”

⁶Section 27.4.3, “Reading from a stream”

refs/remotes refs to their remote refs/heads counterparts, along with a flag saying whether each mapping is subject to forced updates.

For example, let's use the @remote_refs and @targets values we saw above. Now imagine that the fetching repository already had an up-to-date copy of refs/remotes/origin/maint, so that ref does not appear in @local_refs. However, its copy of refs/remotes/origin/master is out of date and we decided to fetch 5d826e9... as a result, and it did not yet have the refs/remotes/origin/todo ref at all. @local_refs lists the refs we're updating, with our current values for those refs.

```
@remote_refs = {
    "HEAD"          => "5d826e972970a784bd7a7bdf587512510097b8c7",
    "refs/heads/maint" => "98cdfbb84ad2ed6a2eb43dafa357a70a4b0a0fad",
    "refs/heads/master" => "5d826e972970a784bd7a7bdf587512510097b8c7",
    "refs/heads/todo"   => "b2cc3488ba006e3ba171e85dffbe6f332f84bf9a"
}

@targets = {
    "refs/remotes/origin/maint" => ["refs/heads/maint", true],
    "refs/remotes/origin/master" => ["refs/heads/master", true],
    "refs/remotes/origin/todo"   => ["refs/heads/todo", true]
}

@local_refs = {
    "refs/remotes/origin/master" => "a3d959a5424171631aeeca59f29a7793fe9025b2",
    "refs/remotes/origin/todo"   => nil
}
```

To update our refs/remotes refs, update_remote_refs iterates over the fetched refs in @local_refs and calls attempt_ref_update with each one.

```
# lib/command/fetch.rb

def update_remote_refs
    @stderr.puts "From #{ @fetch_url }"

    @errors = {}
    @local_refs.each { |target, oid| attempt_ref_update(target, oid) }
end
```

attempt_ref_update, shown below, tries to update a refs/remotes ref, given the information it has about the current value of that ref and the value we want to replace it with. Let's say it's called with the values "refs/remotes/origin/master" and "a3d959a...", reflecting the current state of our copy of that ref. It will then consult the @targets mapping to discover this ref should reflect the value of refs/heads/master in the remote repo, and its update should be forced. Looking in @remote_refs will give the remote's value of the refs/heads/master ref, which is 5d826e9....

Having discovered these parameters, we call fast_forward_error("a3d959a...", "5d826e9..."), which returns some error if it cannot prove the old ID is an ancestor of the new one, and nil otherwise. If there is no error, or @targets says the ref is forced, or the --force option was used, then we update the refs/remotes/origin/master ref in our repo to 5d826e9.... Otherwise, we store the fast-forward error that was detected; adding to the @errors structure will make fetch exit with a non-zero status. Finally we call through to

report_ref_update, passing the names ["refs/heads/master", "refs/remotes/origin/master"], the fast-forward error, the old and new IDs a3d959a... and 5d826e9..., and a boolean that's true if there was no error detected.

```
# lib/command/fetch.rb

def attempt_ref_update(target, old_oid)
  source, forced = @targets[target]

  new_oid = @remote_refs[source]
  ref_names = [source, target]
  ff_error = fast_forward_error(old_oid, new_oid)

  if @options[:force] or forced or ff_error == nil
    repo.refs.update_ref(target, new_oid)
  else
    error = @errors[target] = ff_error
  end

  report_ref_update(ref_names, error, old_oid, new_oid, ff_error == nil)
end
```

The above procedure relies on a few methods in the shared module FastForward. These include some functionality that won't be relevant yet, but will be used by the push command when we come to implement it. fast_forward_error returns `nil` if `old_oid` is an ancestor of `new_oid`, or an error message if it cannot prove this is the case. If either ID is `nil` this is fine — it means we're either creating or deleting a ref. However if our database does not have `old_oid`, then we can't check its ancestry — this happens during a push if we don't have a local copy of the remote repository's head before we attempt to update it. We return an error if `fast_forward?` returns `false`, otherwise we implicitly return `nil`. `fast_forward?` uses `Merge::CommonAncestors`⁷ to determine whether one commit is an ancestor of the other.

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

module Command
  module FastForward

    def fast_forward_error(old_oid, new_oid)
      return nil unless old_oid and new_oid
      return "fetch first" unless repo.database.has?(old_oid)
      return "non-fast-forward" unless fast_forward?(old_oid, new_oid)
    end

    def fast_forward?(old_oid, new_oid)
      common = ::Merge::CommonAncestors.new(repo.database, old_oid, [new_oid])
      common.find
      common.marked?(old_oid, :parent2)
    end
  end
end
```

The Database#has? method tells us whether we have a copy of a certain object, which it does by seeing if the file that would contain that object exists in the .git/objects directory.

⁷Section 17.2, “Finding the best common ancestor”

```
# lib/database.rb

def has?(oid)
  File.file?(object_path(oid))
end
```

The final step in updating the refs/remotes refs is to display the updates we've made to the user. You may have seen Git print a message like this when pulling from a remote repository:

```
From: file:///home/username/path/to/repo
a3d959a..5d826e9 master -> origin/master
```

It shows the URL it's fetching from, and then a line for each ref it's updated, showing the revision range covered by the update, and the short names of the source and target refs. There are several possible formats for this message, and they include the following.

If the update is a fast-forward, then the revision range uses two dots, as above. If it's not a fast-forward, the range has three dots, a preceding + sign, and the message forced update following it.

```
+ a3d959a...5d826e9 master -> origin/master (forced update)
```

If the target ref did not previously exist, then the message * [new branch] is shown in place of a revision range.

```
* [new branch] master -> origin/master
```

If the source ref no longer exists and the target is therefore being deleted, the label - [deleted] is used.

```
- [deleted] master -> origin/master
```

And finally, if there was an error updating the ref, the message begins with ! [rejected] and is followed by the error message.

```
! [rejected] master -> origin/master (non-fast-forward)
```

`report_ref_update` in the `RemoteClient` module takes care of these situations by using the information gathered about the update by the `attempt_ref_update` method above. All arguments after the first two are optional, and the method encodes the decision procedure above. If there was an error, then we display the rejected notice with the error. If both IDs are equal, then there was no update and we exit without taking any action. If the old ID is `nil`, that means a new ref has been created, whereas if the new ID is `nil` it's being deleted. Otherwise, it's a revision range update, which is handled by a separate method.

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

def report_ref_update(ref_names, error, old_oid = nil, new_oid = nil, is_ff = false)
  return show_ref_update("!", "[rejected]", ref_names, error) if error
  return if old_oid == new_oid

  if old_oid == nil
    show_ref_update("*", "[new branch]", ref_names)
  elsif new_oid == nil
    show_ref_update("-", "[deleted]", ref_names)
  else
```

```
    report_range_update(ref_names, old_oid, new_oid, is_ff)
end
```

`report_range_update` handles the two revision range styles of update notice, based on whether the update is a fast-forward or not. If the update is forced, but is still a fast-forward, then it should be displayed as a fast-forward. Any non-fast-forward update should be assumed to be forced.

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

def report_range_update(ref_names, old_oid, new_oid, is_ff)
  old_oid = repo.database.short_oid(old_oid)
  new_oid = repo.database.short_oid(new_oid)

  if is_ff
    revisions = "#{old_oid}..#{new_oid}"
    show_ref_update(" ", revisions, ref_names)
  else
    revisions = "#{old_oid}...#{new_oid}"
    show_ref_update("+", revisions, ref_names, "forced update")
  end
end
```

`show_ref_update` underlies all the above logic, and takes the various fragments of text generated by `report_ref_update` and `report_range_update` and formats them into a single message. It converts the ref names to their short form and links them with `->`, and joins all the other parts of the message together before printing it.

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

def show_ref_update(flag, summary, ref_names, reason = nil)
  names = ref_names.compact.map { |name| repo.refs.short_name(name) }

  message = "#{flag} #{summary} #{names.join(" -> ")}"
  message.concat(" (#{reason})") if reason

  @stderr.puts message
end
```

The `fetch` and `upload-pack` commands are now functionality complete and can transfer data between two repositories on the same filesystem. One further change and we'll be able to fetch content over the network.

28.2.6. Connecting to remote repositories

In Section 27.1, “Programs as ad-hoc servers” we learned that two processes communicating via their standard I/O streams can communicate over a network, by tunneling their communication through an SSH connection. By running a program on a remote machine via SSH, we can interact with its I/O just as we did when it was on the same machine. The only thing we changed was that instead of running a local command like

```
ruby adder.rb
```

We ran the program on a remote machine using this command:

```
ssh 127.0.0.1 -p 2222 -l ubuntu "ruby /vagrant/adder.rb"
```

After that, everything else proceeded as before. We can add support for ssh: URLs to our `RemoteClient` module by expanding the `build_agent_command` method. When given a `file:` URL it constructs an array like `["git-upload-pack", "<path>"]`, which is then run as a command. To support ssh: URLs, for example `ssh://git@github.com/jcoglan/jit.git`⁸, we need to make it generate the array representing an ssh command, such as:

```
["ssh", "github.com", "-l", "git", "git-upload-pack /jcoglan/jit.git"]
```

The code below accomplishes just that.

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

def build_agent_command(program, url)
  uri = URI.parse(url)
  argv = Shellwords.shellsplit(program) + [uri.path]

  case uri.scheme
  when "file" then argv
  when "ssh"  then ssh_command(uri, argv)
  end
end

def ssh_command(uri, argv)
  ssh = ["ssh", uri.host]
  ssh += ["-p", uri.port.to_s] if uri.port
  ssh += ["-l", uri.user] if uri.user

  ssh + [Shellwords.shelljoin(argv)]
end
```

With this change, Jit is now able to fetch a repository from another machine, if it uses `--upload-pack="git upload-pack"`. Git's `upload-pack` command will apply delta compression to the objects it sends, and we cannot yet handle that. But for now, this is a good working implementation and we can move on to adding the `push` command.

28.3. Clone and pull

The `fetch` command underlies two other commonly used commands. `clone` is used to set up a new repository that's a copy of an existing one, and this can be accomplished using the `init`, `remote`, `fetch` and `reset` commands.

```
$ jit init fetch-test
$ cd fetch-test
$ jit remote add origin ssh://git@github.com/jcoglan/jit.git
$ jit fetch origin
$ jit reset --hard origin/master
```

This creates a new repository and adds an `origin` remote to it, with the given URL. The `fetch` command retrieves all the objects from that repository and generates refs in `refs/remotes` to mirror the `refs/heads` references in the remote repo. Finally, `reset --hard` is used to point our

⁸Git also supports writing this URL as `git@github.com: jcoglan/jit.git`, but to keep things simple I am only supporting URLs with an explicit `file:` or `ssh:` scheme.

current branch, `refs/heads/master`, at the same commit as `refs/remotes/origin/master`, and update the index and workspace to reflect its content. Our `master` branch now mirrors that in the remote repository.

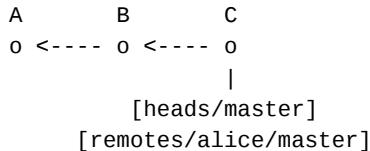
As we continue to work on the project, we'll want to fetch further updates from the remote repository and merge them into our own history. You may be familiar with the `pull` command, which is equivalent to running `fetch` and then `merge`. For example, if we're currently on the `master` branch and want to incorporate changes to `master` from the remote repo, we can run:

```
$ jit fetch origin
$ jit merge origin/master
```

This will fetch all the updated refs from the `origin` remote, and then merge its `master` branch into our own. If we've not made any commits ourselves on `master` since we last fetched, then this will be a fast-forward and will simply move our `heads/master` pointer. If there are no new commits on the remote branch, then the `merge` command has no effect. If we've made some of our own commits on `master` then things become more complicated.

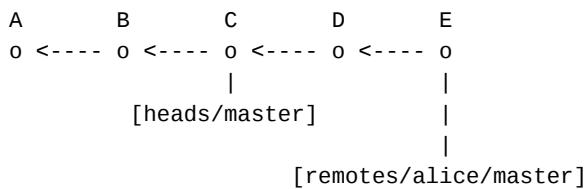
Consider the example that we began the chapter with, in which Bob has fetched a history from Alice. His own `master` branch and his copy of Alice's `master` branch both point at the same commit. For brevity I have left the prefix `refs/` out of all the ref annotations in the following diagrams.

Figure 28.8. Bob's repository after fetching from Alice



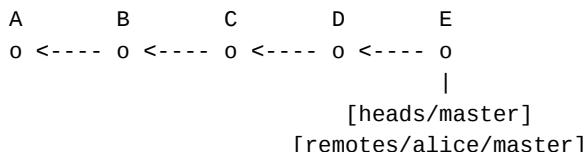
If Alice adds two further commits, `D` and `E`, to her `master` branch, and then Bob fetches again, his repository will move its `remotes/alice/master` pointer to include these new commits.

Figure 28.9. Bob fetches new commits from Alice



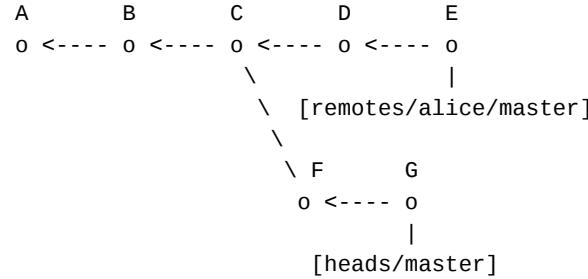
Because Bob has not made any of his own commits on top of `C`, running `merge alice/master` just fast-forwards his own `master` branch so that it agrees with Alice's.

Figure 28.10. Bob performs a fast-forward merge



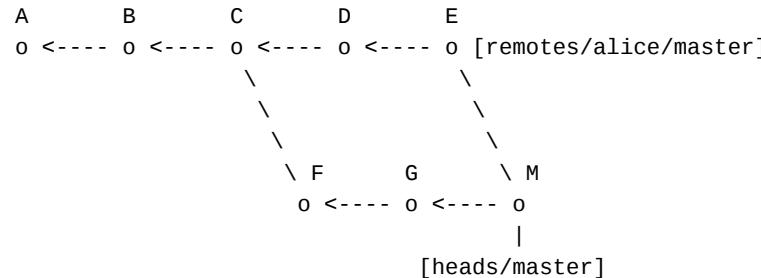
Now consider the case where, after Bob has fetched *C* from Alice, Alice adds commits *D* and *E* on top of *C*, while Bob adds *F* and *G* on top of *C*. If Bob now fetches from Alice, his own master branch and his copy of Alice's master branch will point at divergent histories.

Figure 28.11. Bob fetches divergent history



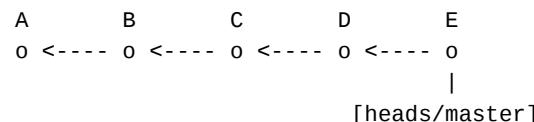
Bob can incorporate Alice's history into his own by merging *alice/master*. Since this is no longer a fast-forward, we must perform a true merge and generate a commit for it.

Figure 28.12. Bob merges Alice's changes into his master branch



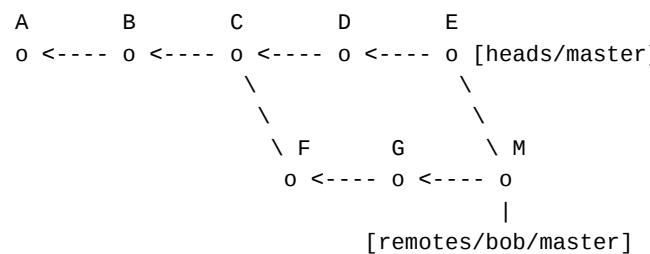
Although their histories have diverged, this merge allows Alice and Bob to reach consensus on the state of the project. Consider the situation from Alice's point of view, at the point where she has just written commit *E*. She has not yet seen any data from Bob's repository.

Figure 28.13. Alice's history



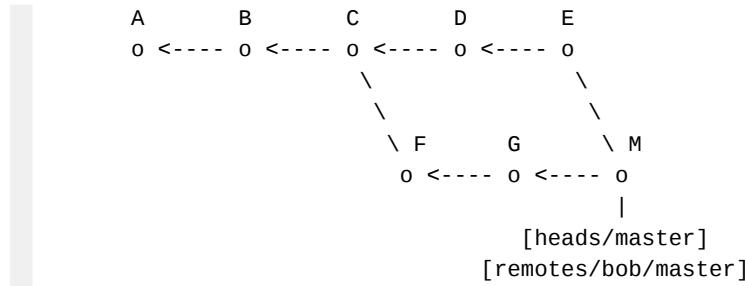
Now, Bob performs the above steps, fetching *E* from Alice and merging it with his own changes to form the merge commit *M*. If Alice now fetches from Bob, she'll receive that portion of the graph she does not have, that is, the range *E..M*.

Figure 28.14. Alice fetches from Bob



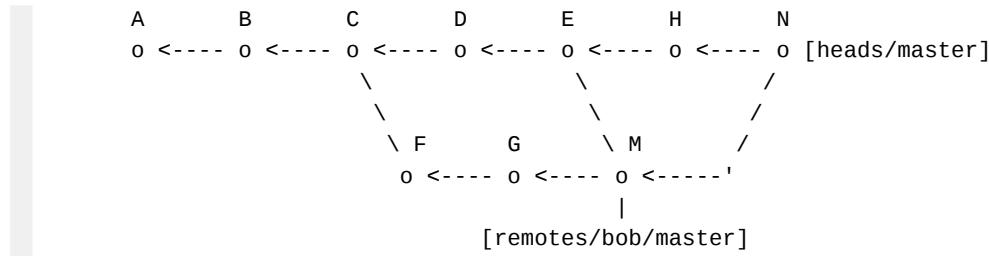
If Alice now runs `merge bob/master`, this causes a fast-forward and her own master branch moves to reflect Bob's view of the branch.

Figure 28.15. Alice fast-forwards to Bob's merge commit



If Alice has made further commits on top of *E* since Bob fetched from her, then running `merge bob/master` will produce a further merge commit incorporating both their changes. Bob can then fetch commit *N* from Alice, and the cycle continues.

Figure 28.16. Alice merges Bob's changes with her own



This process illustrates the role that merging plays in achieving consensus: when two parties have diverging histories, one of them can fetch the other's changes and merge them with their own. That merge can then be shared with the rest of the team, who collectively achieve eventual consistency⁹ over the state of the project.

28.3.1. Pulling and rebasing

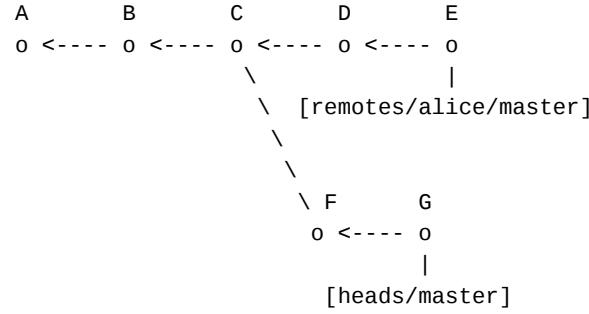
If team members are frequently fetching from one another while making their own changes, this can result in the history being littered with merge commits. Picture the diagram above extending to the right, forming a chain of merges going back and forth as Alice and Bob fetch from each other.

If those merge commits are making the history hard to understand, and they don't add any particularly useful information, it may be beneficial to avoid creating them. Git lets us do this using the `pull --rebase` command, which runs `rebase` instead of `merge` after fetching the remote's updates. As we saw in Section 24.2, "Rebase", we can replicate this behaviour using the `cherry-pick` command.

For example, picture the situation we saw above where Alice and Bob have divergent histories on their master branches, beginning with commit *C*. Bob creates commits *F* and *G*, and fetches from Alice to receive *D* and *E*.

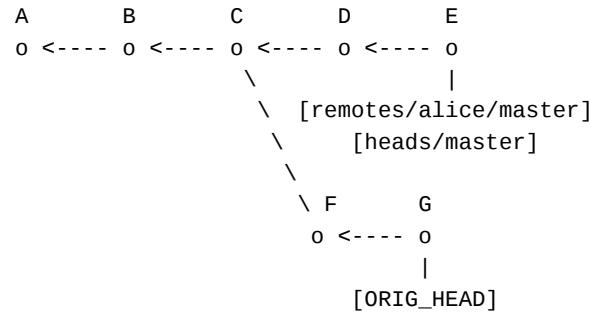
⁹https://en.wikipedia.org/wiki/Eventual_consistency

Figure 28.17. Bob's repository after fetching from Alice



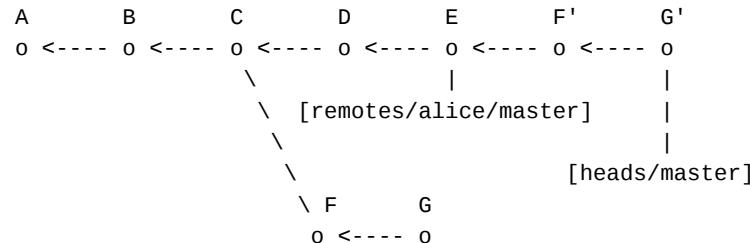
At this point, instead of running `merge alice/master` which would create a merge commit, Bob can create a flattened history by first running `reset --hard alice/master` to move his `master` branch to point at Alice's latest commit. This leaves the `ORIG_HEAD` ref pointing at `G`.

Figure 28.18. Bob resets his master branch to Alice's latest commit



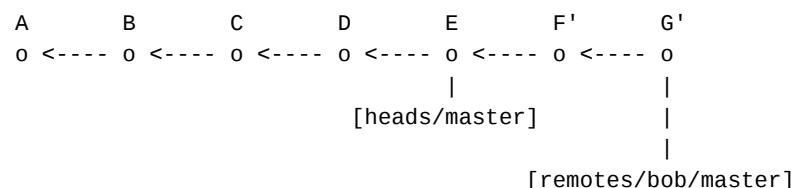
He can then run `cherry-pick ..ORIG_HEAD`, which will pick commits `F` and `G` on top of `E`, creating new commits `F'` and `G'`.

Figure 28.19. Bob rebases his branch



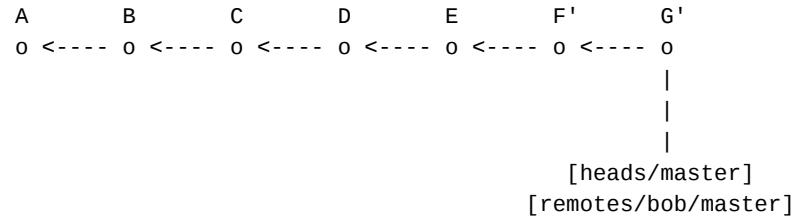
If Alice now fetches from Bob, she'll receive commits `F'` and `G'`:

Figure 28.20. Alice's repository after fetching Bob's rebased commits



And after fast-forwarding her `master` branch, the two once more agree on the state of the repository.

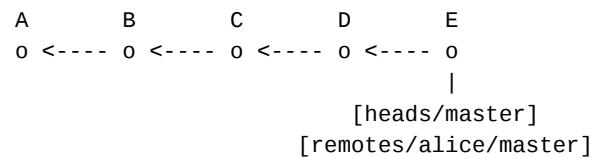
Figure 28.21. Alice fast-forwards her master branch



28.3.2. Historic disagreement

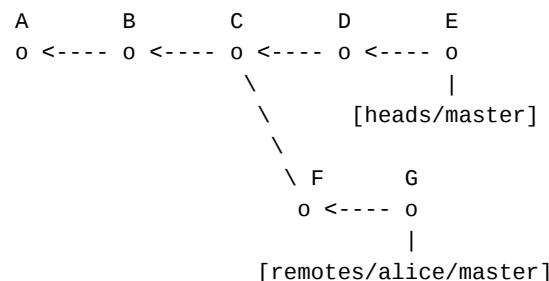
Although rebasing certainly helps with keeping the history tidy, it causes problems if one author discards commits that have already been shared and built upon by others. For example, let's examine the situation where Alice has authored commit *E*, and Bob fetches this from her.

Figure 28.22. Bob's repository after fetching from Alice



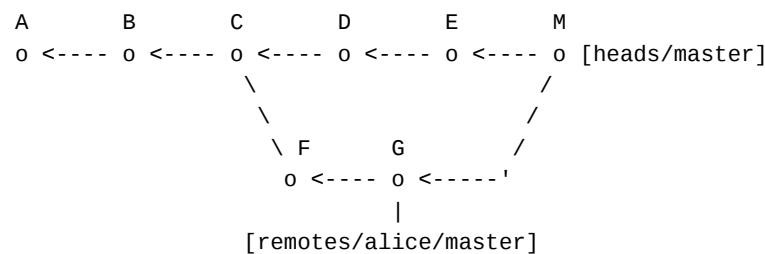
Now suppose that Alice discards commits *D* and *E* and begins a new history from commit *C*, creating commits *F* and *G*. Bob fetches these, and before he merges or rebases, his repository looks like this:

Figure 28.23. Bob fetches Alice's rewritten history



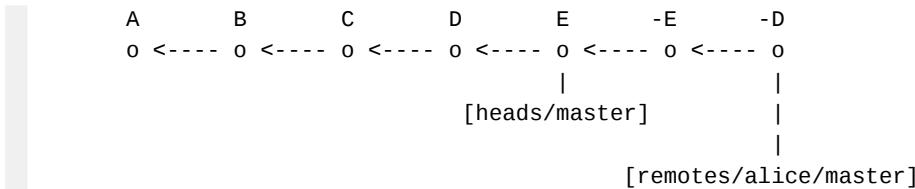
If Bob now merges Alice's branch with his own, the result is a merge commit that incorporates commits *D* and *E*. This happens because the commit graph doesn't record any notion of some commits being replacements for others, or of which commits came from whose repository. The graph simply says that Bob's head *E* and Alice's head *G* have a common ancestor *C*, and can be merged to produce *M*.

Figure 28.24. Bob merges Alice's old and new histories



If Alice now fetches from Bob, the state of the project will include content she wanted to get rid of — the changes in commits *D* and *E*. Because commit *E* had already been shared with Bob, it cannot easily be gotten rid of without telling Bob that's what was intended. If Alice wants to remove content in commits that have already been distributed, it would be safer to apply commits on top of *E* that revert *E* and *D*, and then have Bob fetch those.

Figure 28.25. Bob's repository after fetching Alice's revert commits



If Bob merges `alice/master` now, he will end up with a state in which the content from *D* and *E* has been removed.

Alice could have instead told Bob about what she'd done, and asked him to abandon commits D and E by performing a hard reset to commit G , and then re-applying any of his own changes on top of that. However, this requires more co-ordination and only gets harder as the team gains members. In general, it is safer to assume commits that have been incorporated into other people's histories cannot be removed, and should instead be undone by using the revert command.

29. Pushing changes

For users that can connect to one another's computers over a network, the `fetch` command works fine for sharing changes between a team. However, most personal computers and workstations are not directly accessible from the public internet, so it's not always possible to fetch content directly from another repository. In this case, it's easier to share changes by *pushing* them to a repository that's accessible to both parties. For example, rather than fetching the Jit repository directly from my personal laptop, I'd like you to be able to get it from the repository at `ssh://git@github.com/jcoglan/jit.git`.

The `push` command is similar to `fetch`, but sends object data in the other direction, from the local repository to the remote. Just as `fetch` uses the remote agent program `git-upload-pack`, `push` relies on `git-receive-pack`. It sends a pack of objects to the `git-receive-pack` process running in the remote repo, which unpacks the objects and updates its refs as requested.

However, there is an important difference in how `push` updates refs. The `fetch` command copies the `refs/heads` references from the remote to corresponding `refs/remotes/<name>` references in the local repository. These remote refs are a cache of the state of the remote repository's branches, and are distinct from the local repo's branches stored in `refs/heads`. `push`, on the other hand, modifies the `refs/heads` references in the *remote* repo, that is it changes the state of the branches in the remote repository. Because of this, it needs to take care not to overwrite changes written by other clients.

We'll explore this problem later in the chapter, but first we need to make some syntactic changes to the `Refspec` class.

29.1. Shorthand `refsspecs`

In the last chapter, we saw that the `fetch` command decides which refs to fetch from the remote by using `refsspecs`¹ stored in the `remote.<name>.fetch` configuration variable. There does exist a corresponding `remote.<name>.push` variable, but it's more common to give the `refspec` to `push` on the command-line. However, it is inconvenient to type out something like `refs/heads/master:refs/heads/master` every time we run `push`, and so Git provides some shorter ways of writing `refsspecs`.

For example, it is common to run `git push origin master` to push your `master` branch pointer to the `origin` remote. The second argument in this command, `master`, is a `refspec`, but it doesn't look like those we've seen before. It doesn't mention the `refs` directory, and there's no `:` separating a source and target name. This `refspec` is a shorthand, and it's been shortened in two distinct ways.

First, a `refspec` lacking a separating `:` is equivalent to one where the source and target are the same. So, `master` is a shorthand for `master:master`.

Second, each source and target is expanded to a canonical path under `refs` by examining its first path segment. If it begins with `refs/`, it is left unchanged. If it begins with `heads/` or `remotes/`,

¹Section 26.3, "Refspecs"

it is prefixed with `refs/`. Otherwise it is prefixed with `refs/heads/`. So, `master` is expanded to `refs/heads/master`, and the refspec `master` is short for `refs/heads/master:refs/heads/master`, and it instructs the push command to copy the local `refs/heads/master` ref to the same location in the remote repository.

The exception to this is that push accepts refspecs where the source is an arbitrary revision, for example `@~3:master`, meaning that our value for the revision `@~3` should be written to the remote's `refs/heads/master`. So if a ref is not a valid ref name, no prefix is added to it.

One final shorthand rule is that the source is allowed to be blank, as in `:master`. This expands to a refspec whose target is `refs/heads/master` but whose source is `nil`, which means we want to delete the target ref from the remote repository.

To accommodate these changes, we'll update `Remotes::Refspec.parse` so that `REFSPEC_FORMAT` allows the source to be empty, and the `:` and target are optional. The source and target are canonicalised, and if the target is missing it is made equal to the source.

```
# lib/remotes/refspec.rb

REFSPEC_FORMAT = /^(\+?)([^:]*)((:[^:]*)?)$/

Refspec = Struct.new(:source, :target, :forced) do
  def self.parse(spec)
    match = REFSPEC_FORMAT.match(spec)
    source = Refspec.canonical(match[2])
    target = Refspec.canonical(match[4]) || source

    Refspec.new(source, target, match[1] == "+")
  end

  #
end
```

`Refspec.canonical` converts a short path to its fully qualified name. It finds the first path out of `refs`, `refs/heads` and `refs/remotes` whose final segment matches the first segment of the given path. It prefixes the path as necessary to complete it, using the `refs/heads` directory if no matching prefix is found. If name is not a valid ref name (for example, it could be a revision like `@~2`) then it is left alone.

```
# lib/remotes/refspec.rb

def self.canonical(name)
  return nil if name.to_s == ""
  return name unless Revision.valid_ref?(name)

  first = Pathname.new(name).descend.first
  dirs = [Refs::REFS_DIR, Refs::HEADS_DIR, Refs::REMOTES_DIR]
  prefix = dirs.find { |dir| dir.basename == first }

  (prefix &. dirname || Refs::HEADS_DIR).join(name).to_s
end
```

A refspec's source can now be `nil`, and `Refspec#match_refs` already takes this into account. Usually, this method matches the source against a list of refs to generate the output. If we run `push foo:bar` then we want to generate the mappings:

```
{ "refs/heads/bar" => ["refs/heads/foo", false] }
```

However, if we run `push :bar`, then we want to update the target, `refs/heads/bar`, not from any of our refs but from *nothing*, i.e. we want to delete it. So in this case, the check at the beginning of `Refspec#match_refs` fails, and it returns the mapping `{ target => [nil, forced] }`.

```
# lib/remotes/refspec.rb

def match_refs(refs)
  return { target => [source, forced] } unless source.to_s.include?("*)

#
end
```

The `Refspec.parse` method now canonicalises shorthand refspecs for us, and we can use these shorthands on the command-line.

```
>> Remotes::Refspec.parse("master")
=> Refspec(source="refs/heads/master", target="refs/heads/master", forced=false)

>> Remotes::Refspec.parse(":master")
=> Refspec(source=nil, target="refs/heads/master", forced=false)

>> Remotes::Refspec.parse(":heads/master")
=> Refspec(source=nil, target="refs/heads/master", forced=false)

>> Remotes::Refspec.parse(":remotes/origin/master")
=> Refspec(source=nil, target="refs/remotes/origin/master", forced=false)

>> Remotes::Refspec.parse(":refs/heads/master")
=> Refspec(source=nil, target="refs/heads/master", forced=false)
```

29.2. The push and receive-pack commands

Just as we did for the `fetch` command, we'll define a `--force` option for `push`. This makes the command overwrite refs in the remote repository even if the requested update is not a fast-forward. The `--receive-pack` option can be used to set a different command to run for the remote agent, in place of the default of `git-receive-pack`. We can use this for testing our own `receive-pack` implementation.

```
# lib/command/push.rb

module Command
  class Push < Base

    def define_options
      @parser.on("-f", "--force") { @options[:force] = true }

      @parser.on "--receive-pack=<receive-pack>" do |receiver|
        @options[:receiver] = receiver
      end
    end

    #
  end
end
```

Again, following the pattern of the `fetch` command, `push` begins by configuring itself and making a connection to the remote repository. Here we're opting in to the `report-status` capability, which will make `receive-pack` send some messages indicating whether the objects unpacked successfully and whether each ref was updated. After making a connection, `Command::Push#run` sets out the high-level procedure for the command: it receives the current state of the remote's refs, then it sends a set of requests stating which refs it wants to update. After this it sends its pack of objects, begins printing a status report, and then receives the status messages from the remote.

```
# lib/command/push.rb

include FastForward
include RemoteClient
include SendObjects

CAPABILITIES = ["report-status"]

def run
  configure
  start_agent("push", @receiver, @push_url, CAPABILITIES)

  recv_references
  send_update_requests
  send_objects
  print_summary
  recv_report_status

  exit (@errors.empty? ? 0 : 1)
end
```

Note that `Command::Push` includes `SendObjects`, the module used by `Command::UploadPack` to send object data. Like `Command::Fetch`, it includes `FastForward` and `RemoteClient`, enabling it to check for fast-forward violations, make a connection to the remote repo, receive the initial reference list and print any ref updates that were applied.

The `receive-pack` command mirrors this sequence of steps, accepting the connection using `accept_client`, sending its reference list, receiving the update requests and object data, and then updating the refs in its own repository. It needs to advertise the capability `report-status` so that the client can send it back, and the `delete-refs` capability, since deleting refs is an optional feature and `push` will only attempt to use it if `receive-pack` says it supports it. We'll also add the `no-thin` capability; Git's `push` command assumes that `receive-pack` supports thin packs, a feature of the delta compression system, but since we don't yet support compression we need to tell the `push` command not to use this feature.

```
# lib/command/receive_pack.rb

module Command
  class ReceivePack < Base

    include ReceiveObjects
    include RemoteAgent

    CAPABILITIES = ["no-thin", "report-status", "delete-refs"]

    def run
```

```
accept_client("receive-pack", CAPABILITIES)

send_references
recv_update_requests
recv_objects
update_refs

exit 0
end

# ...

end
end
```

To configure itself, push needs to know where the remote repo is located, and which refs it should update. Like fetch, it gets the repo URL from the first argument, or from the configuration if this argument is the name of a configured remote. We need the fetch refsspecs, given by the `remote.<name>.fetch` variable, in order to update our local cache of the remote's refs. The receiver agent program is given either by the `--receive-pack` option, the `remote.<name>.receivepack` variable, or it defaults to `git-receive-pack`, and the push refsspecs are either given on the command-line or taken from the `remote.<name>.push` config variable.

```
# lib/command/push.rb

RECEIVE_PACK = "git-receive-pack"

def configure
  name    = @args.fetch(0, Remotes::DEFAULT_REMOTE)
  remote = repo.remotes.get(name)

  @push_url    = remote.push_url || @args[0]
  @fetch_specs = remote.fetch_specs || []
  @receiver    = @options[:receiver] || remote.receiver || RECEIVE_PACK
  @push_specs  = (@args.size > 1) ? @args.drop(1) : remote.push_specs
end
```

The `push_specs` and `receiver` methods in the above snippet are part of the `Remotes::Remote` class for interacting with remote configuration.

```
# lib/remotes/remote.rb

def push_specs
  @config.get_all(["remote", @name, "push"])
end

def receiver
  @config.get(["remote", @name, "receivepack"])
end
```

29.2.1. Sending update requests

After running `recv_references` to download the current ref state from the remote repo, the `push` command will have information that looks something like this:

```
@remote.refs = {
```

```

"HEAD"          => "5d826e972970a784bd7a7bdf587512510097b8c7",
"refs/heads/maint" => "98cdfbb84ad2ed6a2eb43dafa357a70a4b0a0fad",
"refs/heads/master" => "5d826e972970a784bd7a7bdf587512510097b8c7",
"refs/heads/todo"  => "b2cc3488ba006e3ba171e85dffbe6f332f84bf9a"
}

```

Say we've run the command `push origin master`. That means `@push_specs` contains the following, indicating we want to update the remote's `refs/heads/master` with our own value for that ref, without forcing a non-fast-forward update.

```

@push_specs = [
  Refspec(source="refs/heads/master", target="refs/heads/master", forced=false)
]

```

We use this list to select which of our local refs should be used in the update. We can get a list of all our local refs by calling `repo.refs.list_all_refs.map(&:path)`, which will return, for example, `["HEAD", "refs/heads/master", "refs/remotes/origin/master"]`. By matching this list against `@push_specs`, we can select those refs that should be pushed to the remote.

To send an update request, we need to send a message of the form `<old_id> <new_id> <ref>`. For example, if our own value of `refs/heads/master` is `3ab341d...`, then we need to send the following message, where the IDs have been abbreviated to save space.

```
5d826e9... 3ab341d... refs/heads/master
```

This message is interpreted by `receive-pack` as an instruction to update `refs/heads/master` from `5d826e9...` to `3ab341d...`. If the given ref does not currently exist in the remote repo, then the first ID should be given as `0000000...`. `push` must send one such message for every ref it wants to update, followed by a flush-packet.

In the `Command::Push` class, `send_update_requests` performs this task. Fetching a list of all the ref names present in the current repo, it matches them against the push specs to select which ones to push. Each of these targets is passed to `select_update`, which builds up the `@updates` structure recording the requests we want to make. Once this list is complete, each request is sent using `send_update`, and then a flush-packet is sent.

```

# lib/command/push.rb

def send_update_requests
  @updates = {}
  @errors = []

  local_refs = repo.refs.list_all_refs.map(&:path).sort
  targets    = Remotes::Refspec.expand(@push_specs, local_refs)

  targets.each do |target, (source, forced)|
    select_update(target, source, forced)
  end

  @updates.each { |ref, (*, old, new)| send_update(ref, old, new) }
  @conn.send_packet(nil)
end

```

`select_update` builds all the information required to send an update request for the given ref, as well as information we'll need when printing status updates later. It loads the old ID from the

references that receive-pack sent at the beginning, and determines the new ID by resolving it as a revision²—in the push command, a refspec's source can be any revision, not just a branch name.

It checks whether these values are equal, and skips the update if so. It also checks if the update is a fast-forward, using the `fast_forward_error` method³. If it is a fast-forward, or the update is forced, then we store the necessary data in the `@updates` hash, otherwise we record an error.

```
# lib/command/push.rb

def select_update(target, source, forced)
  return select_deletion(target) unless source

  old_oid = @remote_refs[target]
  new_oid = Revision.new(repo, source).resolve

  return if old_oid == new_oid

  ff_error = fast_forward_error(old_oid, new_oid)

  if @options[:force] or forced or ff_error == nil
    @updates[target] = [source, ff_error, old_oid, new_oid]
  else
    @errors.push([[source, target], ff_error])
  end
end
```

Note that if the old ID—the remote's current value for the ref—does not exist in the local repository, then `fast_forward_error` will return the message `fetch first`. Unless an update is forced, `push` will only apply updates in cases where it has the commit for the old ID, and has checked it's an ancestor of the new ID. This means that it can use the old ID as part of a `RevList` query to select which objects to send, and it will not cause any information to be lost from the remote.

For example, consider this repository in which the local `master` branch is in sync with its remote counterpart.

Figure 29.1. Repository in sync with the remote

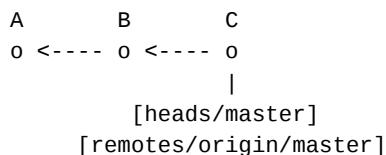
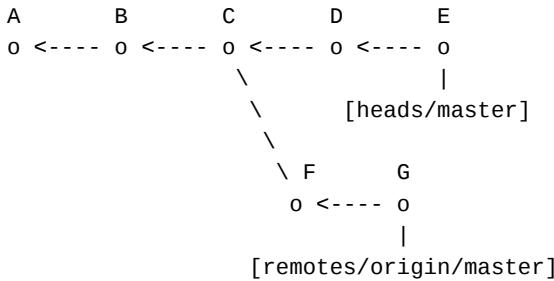


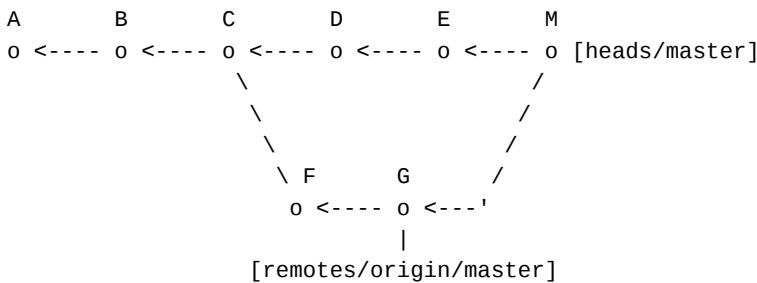
Figure 29.2. Local history diverges from the remote



If we had not yet fetched these new commits from the remote, then we would know from the initial ref list from `receive-pack` that the remote repo has `refs/heads/master` pointing at `G`, but we would not have `G` in our local database, so we'd be unable to find out whether it's an ancestor of our own `master` commit. After we've fetched, we can see that `G` is not an ancestor of our `master` commit `E`, and so we don't push `E`. If we force the update, that will make the remote's `master` branch point at `E`, and so `F` and `G` will be lost. If we don't want to lose other people's work, we need to fetch, merge the remote state with our own, and then push again.

Running `merge origin/master` produces this state in our local repo:

Figure 29.3. Merging the remote's changes



Since `G` is now an ancestor of our current `master` commit `M`, we can push our `master` to the remote without losing data. A similar result occurs if we rebase our changes instead of merging them.

If `select_update` is called with `nil` as the `source` argument, it delegates to the `select_deletion` method. This handles cases where the `refspec` source is blank, meaning we want to delete a ref from the remote. To do this, we need to check that the remote supports the `delete-refs` capability, and if so, we store an update with no source, no fast-forward error, the current commit ID but no new ID.

```

# lib/command/push.rb

def select_deletion(target)
  if @conn.capable?("delete-refs")
    @updates[target] = [nil, nil, @remote_refs[target], nil]
  else
    @errors.push([[nil, target], "remote does not support deleting refs"])
  end
end
  
```

Sending the update requests to the remote agent just requires converting any `nil` references to the all-zero ID, and then sending a message containing the old ID, the new ID, and the name of the ref.

```
# lib/command/push.rb

def send_update(ref, old_oid, new_oid)
  old_oid = nil_to_zero(old_oid)
  new_oid = nil_to_zero(new_oid)

  @conn.send_packet("#{ old_oid } #{ new_oid } #{ ref }")
end

def nil_to_zero(oid)
  oid == nil ? ZERO_OID : oid
end
```

The remote agent reads these messages until it gets a flush-packet, and stores them in the `@requests` variable, converting any all-zero IDs back to the Ruby `nil` value.

```
# lib/command/receive_pack.rb

def recv_update_requests
  @requests = {}

  @conn.recv_until(nil) do |line|
    old_oid, new_oid, ref = line.split(/ +/)
    @requests[ref] = [old_oid, new_oid].map { |oid| zero_to_nil(oid) }
  end
end

def zero_to_nil(oid)
  oid == ZERO_OID ? nil : oid
end
```

To determine which objects to send, the push command needs to select all those commits that are reachable from the refs it's pushing, that the remote does not already have. We know which commits the remote has from the initial ref list sent by `receive-pack` that we stored in `@remote.refs`, and if we're only pushing fast-forward updates, then we have those commits and all their history locally. The entries in the `@updates` hash consist of four values: the source ref name, a possible fast-forward error, the old ID and the new ID. So, we can get the objects we need using a `RevList` by starting with the last value from each update entry, and excluding all the commits we know the remote already has. If we're not sending any new commits, i.e. `@updates` is empty or contains only deletions, then we don't send any pack data.

```
# lib/command/push.rb

def send_objects
  revs = @updates.values.map(&:last).compact
  return if revs.empty?

  revs += @remote_refs.values.map { |oid| "^#{ oid }" }

  send_packed_objects(revs)
end
```

In the `receive-pack` command, we use the `recv_packed_objects` method from `ReceiveObjects` to read the pack that the push command sent, and send the status message `unpack ok` if it succeeded and `unpack <error message>` otherwise. `report_status` sends the given message if the remote client has enabled the `report-status` capability.

```
# lib/command/receive_pack.rb

def recv_objects
  @unpack_error = nil
  recv_packed_objects if @requests.values.any?(&:last)
  report_status("unpack ok")
rescue => error
  @unpack_error = error
  report_status("unpack #{error.message}")
end

def report_status(line)
  @conn.send_packet(line) if @conn.capable?("report-status")
end
```

The receive-pack process now has a set of update requests sent by the push command, and it's received all the objects necessary to complete those requests. The final step of the process is for receive-pack to update the refs in the remote repository and report the results back to the push process.

29.2.2. Updating remote refs

After receiving the update requests from the client, the receive-pack process will have a structure like this that maps ref names to pairs of old and new IDs.

```
@requests = {
  "refs/heads/master" => ["5d826e9...", "3ab341d..."]
}
```

Once it's extracted all the objects from the pack it received, it can apply these updates so that its refs point at the desired objects. After completing these changes, it sends a flush-packet if the client has asked for status reports.

```
# lib/command/receive_pack.rb

def update_refs
  @requests.each { |ref, (old, new)| update_ref(ref, old, new) }
  report_status(nil)
end
```

At this point we need to be very careful about how we update the refs in the remote repository, because multiple clients could try to push to the same repo concurrently. If Alice and Bob each start a push process at the same time on different machines, each will run a receive-pack process on the remote repo and will be told the current value of `refs/heads/master` is, say, `5d826e9...`. Each will then send an update request with their current value for the master branch. For example, Alice sends:

```
5d826e9... d11b2a4... refs/heads/master
```

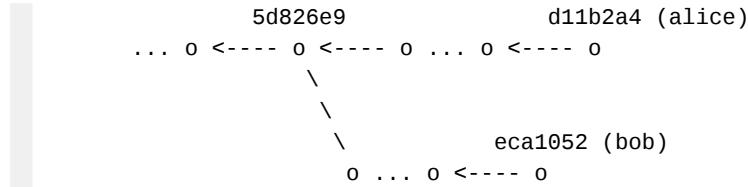
While Bob sends:

```
5d826e9... eca1052... refs/heads/master
```

Let's assume neither party is forcing the update, so they do not want the remote to lose any commits. Therefore, `d11b2a4...` and `eca1052...` are both descendants of `5d826e9...`. But, because

Alice has not yet seen Bob's latest commit and vice versa, neither is an ancestor of the other, and the combined history will look like this:

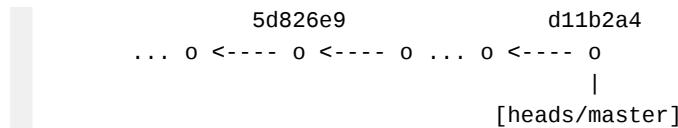
Figure 29.4. Alice and Bob's combined history



If both these updates are applied by the remote, then whichever one is received last will overwrite the other — `refs/heads/master` in the remote ends up pointing at either `d11b2a4...` or `eca1052...`, and the other commit and its history back to `5d826e9...` will effectively be lost. What we'd like to do is reject the second of these updates that we receive, and force its sender to merge the other party's changes before trying to push again.

For example, say Alice's request is received first, and the remote updates its `refs/heads/master` to `d11b2a4...`.

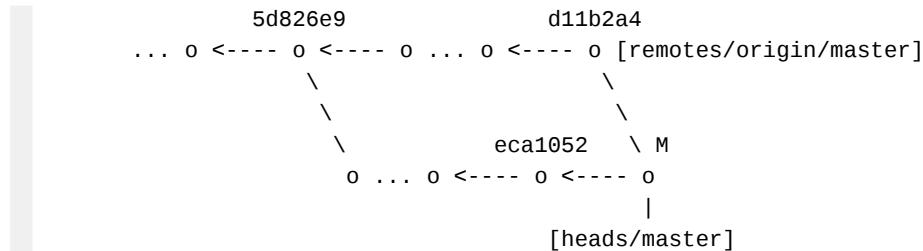
Figure 29.5. Alice's update is accepted



If we make sure Bob's request is not processed until the above update is complete, then we can compare his request to the current state. His request contains the old value `5d826e9...` for the `master` branch, but this is no longer its current value, so we can reject this update.

If Bob tries to push again, he'll find out that the new value of `refs/heads/master` is `d11b2a4...`, which he does not have, so he will fetch. If he makes another attempt, he'll learn that `d11b2a4...` is not an ancestor of his commit `eca1052...`, so he'll perform a merge.

Figure 29.6. Bob's repository after merging Alice's changes



Bob can now push commit `M` to the remote `master` branch since it's a descendant of `d11b2a4...`, and so the remote `master` now points at a commit that incorporates both Alice and Bob's changes, and no updates are lost.

This is a common problem in systems where two actors have concurrent access to modify a shared resource, in this case the remote's `refs/heads/master` pointer. Indeed we've seen it several times before in this project, and have used our `Lockfile` class to prevent concurrent processes from updating the refs, index or configuration at the same time. In those cases, the

lock prevents two processes from both reading a file, making some change to their in-memory copy, and writing the result back, with the last writer’s changes overwriting the others⁴.

In this situation, we will build on top of `Lockfile`, but the concurrency mechanism presented to the clients is different. The `push` command does not take out a lock on the refs it wants to update, preventing other parties from pushing. Instead, it states what the current state was at the time it began its request — it sends the value of the ref that `receive-pack` sent, `5d826e9...`, as part of its update request. `receive-pack` can then check this value against the current state, and if someone else has changed the ref while the `push` command was running, then it rejects the update.

To do this, it needs to check the old value the client sent against the value stored in the ref at the instant before it updates it. In particular, it needs to make sure no other updates to the ref are done between reading the current value and writing the new one. In `Command::ReceivePack#update_ref`, we’ll call a new method called `Refs#compare_and_swap` to perform this task. If the update succeeds, we send the status report `ok` with the name of the ref, otherwise we send `ng` with an error message.

```
# lib/command/receive_pack.rb

def update_ref(ref, old_oid, new_oid)
  return report_status("ng #{ref} unpacker error") if @unpack_error

  repo.refs.compare_and_swap(ref, old_oid, new_oid)
  report_status("ok #{ref}")
rescue => error
  report_status("ng #{ref} #{error.message}")
end
```

Compare-and-swap⁵ is a method of atomically updating a mutable value, only if its current value is equal to what we expected. The request update line `5d826e9... eca1052... refs/heads/master` can be read as, ‘update the contents of `refs/heads/master` to `eca1052...`, only if its current value is `5d826e9...`’. To make our ref updates atomic, we need to check the current value after acquiring a lock on the file, so that no other process can change the ref between the time we read it and the time we update it. This means our equality check remains valid until we ourselves change the value, thus avoiding a *time of check to time of use*⁶ race condition.

In the `Refs` class, the logic for updating a ref file is stored in the `update_ref_file` method, which opens a `Lockfile` for the given ref to update its contents. We’d like to add some new behaviour during the time the lock is held, and we can do this by yielding after the lock is acquired, if a block was given. If the block raises an error, we should call `lockfile.rollback` to release the lock before letting the error bubble up. We’ll also augment this method so that if `oid` is `nil`, the given file will be deleted.

```
# lib/refs.rb

def update_ref_file(path, oid)
  lockfile = Lockfile.new(path)

  lockfile.hold_for_update
```

⁴For example, Section 7.1.1, “Parsing .git/index”

⁵<https://en.wikipedia.org/wiki/Compare-and-swap>

⁶https://en.wikipedia.org/wiki/Time_of_check_to_time_of_use

```
yield if block_given?

if oid
  write_lockfile(lockfile, oid)
else
  File.unlink(path) rescue Errno::ENOENT
  lockfile.rollback
end

rescue Lockfile::MissingParent
  FileUtils.mkdir_p(path.dirname)
  retry
rescue => error
  lockfile.rollback
  raise error
end
```

This extension to `update_ref_file` lets us implement the `compare_and_swap` method. It calls `update_ref_file` with a block that, while the lock is held, reads the current ID that the ref points at. If it's not equal to the given value, we can raise an exception, and this prevents the new ID from being written. The error will be caught in `Command::ReceivePack#update_ref` and communicated back to the push client.

```
# lib/refs.rb

StaleValue = Class.new(StandardError)

def compare_and_swap(name, old_oid, new_oid)
  path = @pathname.join(name)

  update_ref_file(path, new_oid) do
    unless old_oid == read_symref(path)
      raise StaleValue, "value of #{name} changed since last read"
    end
  end
end
```

Although `receive-pack` is using a lock internally to guarantee atomic changes, the interface presented to the push command is not lock-based. Instead, the push command states what it believes the ref's current value is, and its request only succeeds if that is indeed the case. Whereas locking is a form of pessimistic concurrency control, this interface is a kind of *optimistic*⁷ control. Rather than preemptively locking resources, the caller attempts to make a change, and only succeeds if no other process has changed the same resource concurrently.

In general, this strategy is vulnerable to the *ABA problem*⁸; if during Bob's first push, Alice changes the remote's `master` from `5d826e9...` to `d11b2a4...` and then back again, it will appear as though the `master` ref has not changed, and Bob's push will succeed. However, since commit IDs represent immutable states of history, in practice this does not pose a data loss problem in Git.

Having sent back its `unpack/ok/ng` status messages, `receive-pack` is now finished and it's the job of the push client to interpret these messages. It begins its printed output by displaying the

⁷https://en.wikipedia.org/wiki/Optimistic_concurrency_control

⁸https://en.wikipedia.org/wiki/ABA_problem

URL it's pushing to, and any errors it detected itself before it attempted to send any update requests. This uses the `report_ref_update` method from `RemoteClient` that we saw in the previous chapter.

```
# lib/command/push.rb

def print_summary
  if @updates.empty? and @errors.empty?
    @stderr.puts "Everything up-to-date"
  else
    @stderr.puts "To #{ @push_url }"
    @errors.each { |ref_names, error| report_ref_update(ref_names, error) }
  end
end
```

Then, unless the remote agent does not support `report-status` or there were no updates sent, `recv_report_status` reads the status lines from the `receive-pack` command. If the `unpack` line says anything other than `ok` then it's printed as an error. Then we read until the `flush-packet` and call `handle_status` to process each individual ref update status.

```
# lib/command/push.rb

UNPACK_LINE = /^unpack (.+)$/

def recv_report_status
  return unless @conn.capable?("report-status") and not @updates.empty?

  unpack_result = UNPACK_LINE.match(@conn.recv_packet)[1]

  unless unpack_result == "ok"
    @stderr.puts "error: remote unpack failed: #{ unpack_result }"
  end

  @conn.recv_until(nil) { |line| handle_status(line) }
end
```

The `handle_status` method takes a single status message from the remote, which will match either `ok <ref>`, or `ng <ref> <error>`. It extracts the information from the message, and calls `report_update` (defined below) to show the update to the user. Then it matches the given ref against the fetch refsspecs, so that we can keep our `refs/remotes` cache of the remote's state up to date. For example, if the remote says `ok refs/heads/master`, and we have a config variable `remote.origin.fetch` with value `+refs/heads/*:refs/remotes/origin/*`, then we'd like to update our `refs/remotes/origin/master` pointer to reflect the new state of the remote. We assume the remote's ref has been updated to the ID we requested, if the status is `ok`.

```
# lib/command/push.rb

UPDATE_LINE = /^(ok|ng) (\S+)(.*)$/

def handle_status(line)
  return unless match = UPDATE_LINE.match(line)

  status = match[1]
  ref    = match[2]
  error  = (status == "ok") ? nil : match[3].strip
```

```
@errors.push([ref, error]) if error
report_update(ref, error)

targets = Remotes::Refspec.expand(@fetch_specs, [ref])

targets.each do |local_ref, (remote_ref,_)|
  new_oid = @updates[remote_ref].last
  repo.refs.update_ref(local_ref, new_oid) unless error
end
end
```

report_update prints the update for the user to see. Because the status report only contains the name of the updated ref, we need to retrieve the IDs to show in the output from the @updates hash. We then use report_ref_update to print the update.

```
# lib/command/push.rb

def report_update(target, error)
  source, ff_error, old_oid, new_oid = @updates[target]
  ref_names = [source, target]
  report_ref_update(ref_names, error, old_oid, new_oid, ff_error == nil)
end
```

For example, if we had run `push origin foo:bar`, then @updates would contain an entry of the form:

```
"refs/heads/bar" => ["refs/heads/foo", nil, old_oid, new_oid]
```

We would have asked the remote to update its refs/heads/bar pointer from the value of our refs/heads/foo. In this case, we'd like to update our refs/remotes/origin/bar pointer to reflect the remote state, but print the update as `foo -> bar`. In effect we've copied our refs/heads/foo to the remote's refs/heads/bar, and then reflected that remote ref in our refs/remotes/origin/bar.

29.2.3. Validating update requests

As repositories that are being pushed to are typically a shared resource accessed and written to by multiple developers, it's sometimes desirable to restrict what types of updates can be pushed to them, to make sure each client's updates make sense and to prevent history being lost. Git provides a few configuration settings in the receive namespace that let you prevent deletions and non-fast-forward updates, and stop any changes at all being made to the current branch.

In `ReceivePack#update_ref`, we can add a call out to validate the given update. `validate_update` will throw an error if the update violates the repository's configuration, and that will be caught by the `rescue` block and relayed to the client.

```
# lib/command/receive_pack.rb

def update_ref(ref, old_oid, new_oid)
  return report_status("ng #{ ref } unpacker error") if @unpack_error

  validate_update(ref, old_oid, new_oid)
  repo.refs.compare_and_swap(ref, old_oid, new_oid)
  report_status("ok #{ ref }")
rescue => error
```

```
    report_status("ng #{ ref } #{ error.message }")
end
```

To begin with we'll perform a couple of safety checks to ensure that the client's update requests make sense — that the refs they use are valid and the object IDs point to objects we have in our database. If the ref is not checked, a client could send a name like `../topic` and write to a file outside of the `.git` directory, or overwrite system programs by using a name like `/bin/bash`. If the new object ID does not exist in our database after reading the pack from the client, then the client has made a mistake and not sent us the objects needed to complete its updates.

```
# lib/command/receive_pack.rb

def validate_update(ref, old_oid, new_oid)
  raise "funny refname" unless Revision.valid_ref?(ref)
  raise "missing necessary objects" if new_oid and not repo.database.has?(new_oid)

# ...
end
```

If the request contains valid data, we'd still like to be able to prevent a client erasing history. Two simple checks we can make are to block branch deletions and non-fast-forward updates. If `receive.denyDeletes` is set, then we'll raise an error if the new ID is `nil` indicating the ref should be deleted. If `receive.denyNonFastForwards` is set, we can call `fast_forward_error` to check the old and new IDs, and throw an error unless the new ID is a descendant of the old.

```
# lib/command/receive_pack.rb

include FastForward

def validate_update(ref, old_oid, new_oid)
  # ...

  if repo.config.get(["receive", "denyDeletes"])
    raise "deletion prohibited" unless new_oid
  end

  if repo.config.get(["receive", "denyNonFastForwards"])
    raise "non-fast-forward" if fast_forward_error(old_oid, new_oid)
  end
end
```

If the repository is one where a developer is working on the code, rather than a shared repo where no commit work happens directly, then we should not make any updates to the ref that `HEAD` currently points at. If we move `HEAD` while someone is making changes in the workspace and index, that will mess up the state of their commits, and possibly lead to them accidentally overwriting pushed changes.

Git defines a *bare* repository as one that is just the contents of a `.git` directory, with a database and refs, but no workspace and possibly no `HEAD` or index. It exists purely to store commits pushed from elsewhere, not to author commits directly. A repository where developers work is not bare, and this is indicated by setting `core.bare` to `false` in `.git/config`. In our `init` command, let's add a bit of code to set this variable when a new repo is created.

```
# lib/command/init.rb
```

```
def run
# ...

config = ::Config.new(git_path.join("config"))
config.open_for_update
config.set(["core", "bare"], false)
config.save

# ...
end
```

Git allows changes to the current branch — the ref linked to by `HEAD` — to be denied in non-bare repositories. `receive.denyCurrentBranch` prevents the current branch from being updated, while `receive.denyDeleteCurrent` stops it being deleted. These values are considered true unless explicitly set to `false`. Setting them to `true` in a bare repo has no effect. We can implement these by adding a section at the end of `validate_update` that's only triggered if `core.bare` is `false` and the given ref is equal to the repo's current ref.

```
# lib/command/receive_pack.rb

def validate_update(ref, old_oid, new_oid)
# ...

return unless repo.config.get(["core", "bare"]) == false and
repo.refs.current_ref.path == ref

unless repo.config.get(["receive", "denyCurrentBranch"]) == false
raise "branch is currently checked out" if new_oid
end

unless repo.config.get(["receive", "denyDeleteCurrent"]) == false
raise "deletion of the current branch prohibited" unless new_oid
end
end
```

It's common to disallow non-fast-forward updates on shared repos. As we saw in Section 28.3.2, “Historic disagreement”, abandoning commits that others have fetched and committed on top of makes it very hard to achieve consensus over the state of the repository, without asking your teammates to make tricky manual changes to their histories.

29.3. Progress meters

Most of the commands we've implemented so far either execute quickly, or they print output so even if they take a long time (for example, running `log --patch` on a large repo), we can see they're doing something. The `fetch` and `push` commands differ in that it can take them a long time to complete their task, depending on the speed of the network connection, and they don't currently display anything meaningful to the user. Without monitoring what's going on inside the `.git` directory, it would be easy to think the process was stuck.

So that we can get some basic visibility into what these commands are doing, let's add a progress meter. For example, when you clone a repository using Git, you'll see something like this in your terminal:

```
remote: Enumerating objects: 99, done.
```

```
remote: Counting objects: 100% (99/99), done.  
remote: Compressing objects: 100% (85/85), done.  
remote: Total 671065 (delta 36), reused 21 (delta 14), pack-reused 670966  
Receiving objects: 100% (671065/671065), 172.98 MiB | 6.07 MiB/s, done.  
Resolving deltas: 100% (495297/495297), done.  
Checking out files: 100% (4036/4036), done.
```

Most of these lines contain a progress meter: a message that updates in place, showing what is currently being done, its progress towards completion as a percentage and as a fraction of the total workload, and for some meters, a volume of data transferred. Ours won't be quite this detailed, but there are a few places where we'd benefit from adding them.

For example, in `Command::SendObjects` we have access to the current command's standard I/O streams, and we can wrap a `Progress` object around standard error and pass it into the `Pack::Writer` we create.

```
# lib/command/shared/send_objects.rb  
  
def send_packed_objects(revs)  
  # ...  
  
  writer = Pack::Writer.new(@conn.output, repo.database,  
                           :compression => pack_compression,  
                           :progress     => Progress.new(@stderr))  
  
  writer.write_objects(rev_list)  
end
```

`Pack::Writer` can then use this abstraction to emit notifications about what is going on inside it. To track how much data has been sent, we'll need to add an `@offset` variable to the class.

```
# lib/pack/writer.rb  
  
def initialize(output, database, options = {})  
  @output  = output  
  @database = database  
  @digest  = Digest::SHA1.new  
  @offset   = 0  
  
  @compression = options.fetch(:compression, Zlib::DEFAULT_COMPRESSION)  
  @progress    = options[:progress]  
end
```

That `@offset` should be updated whenever we write output to the pack.

```
# lib/pack/writer.rb  
  
def write(data)  
  @output.write(data)  
  @digest.update(data)  
  @offset += data.bytesize  
end
```

Let's now use the `Progress` object inside `Pack::Writer`. The `Pack::Writer` class is internal plumbing and is not concerned with user interface logic, so it should not emit progress in terms of printing output, per se. Instead it should just tell the `Progress` object what's going on, and the `Progress` object can display this as it sees fit.

For example, in `prepare_pack_list`, we'll set the current activity by calling `@progress&.start("Counting objects")`. We don't know how many objects there will be, so we can't set an expected total here. Each time we find an object we call `@progress&.tick`, and when we're finished we'll call `@progress&.stop`.

```
# lib/pack/writer.rb

def prepare_pack_list(rev_list)
  @pack_list = []
  @progress&.start("Counting objects")

  rev_list.each do |object, path|
    add_to_pack_list(object, path)
    @progress&.tick
  end
  @progress&.stop
end
```

A more detailed use case can be found in `write_entries`. This starts the progress meter with a count, since it knows how many objects there are. It only emits progress here if it's not writing to standard out, i.e. it's being called by `push` rather than by `upload-pack`. When we call `Progress#tick` here we can also pass in the total amount of data transferred.

```
# lib/pack/writer.rb

def write_entries
  count = @pack_list.size
  @progress&.start("Writing objects", count) unless @output == STDOUT

  @pack_list.each { |entry| write_entry(entry) }
  @progress&.stop
end

def write_entry(entry)
  # ...

  @progress&.tick(@offset)
end
```

These use cases are enough to flesh out a design for the `Progress` class. It will take an output stream that it should write to, and its `@message` property is initially `nil`.

```
# lib/progress.rb

class Progress
  def initialize(output)
    @output = output
    @message = nil
  end

  # ...
end
```

When we call `Progress#start`, we're asking for a new progress meter to begin. We set the meter's message and expected total from the arguments, and set the object and byte counts to zero. If the output is not a terminal, this method does nothing. In order to prevent the meter

being redrawn too frequently, which can make it flicker and results in the program spending a lot of time writing data to the terminal, we track the last time at which the meter was updated. `Process.clock_gettime`⁹ provides data from a *monotonic*¹⁰ clock, whose time always increases and is not affected by timezone changes or the user setting the time. This works better than the wall clock time for measuring durations.

```
# lib/progress.rb

def start(message, total = nil)
  return unless @output.isatty

  @message = message
  @total = total
  @count = 0
  @bytes = 0
  @write_at = get_time
end

def get_time
  Process.clock_gettime(Process::CLOCK_MONOTONIC)
end
```

When `tick` and `stop` are called, we can update the meter's state and print it to the screen. `tick` increments the object count and sets the byte count if given. It then clears the current line in the terminal and writes a new one in its place, if enough time has elapsed since the last update. `stop` sets the total to equal the object count, prints the status, and unsets `@message` so that no further calls to `tick` or `stop` will have any effect until `start` is called. Neither method does anything unless `@message` is set.

```
# lib/progress.rb

def tick(bytes = 0)
  return unless @message

  @count += 1
  @bytes = bytes

  current_time = get_time
  return if current_time < @write_at + 0.05
  @write_at = current_time

  clear_line
  @output.write(status_line)
end

def stop
  return unless @message

  @total = @count

  clear_line
  @output.puts(status_line)
  @message = nil
end
```

⁹https://docs.ruby-lang.org/en/2.3.0/Process.html#method-c-clock_gettime

¹⁰https://en.wikipedia.org/wiki/Monotonic_function

In Section 10.4, “Printing in colour” we encountered *escape codes*, sequences of special characters that affect how the terminal displays text. All such sequences begin with `\e[`, and the following characters encode formatting instructions. In `clear_line`, we use the code `G` to move the cursor to the beginning of the current line, and then `K` to clear it. Using `write` rather than `puts` in the `tick` method means there is no trailing line feed appended to the message, so the cursor remains on the same line.

```
# lib/progress.rb

def clear_line
  @output.write("\e[G\e[K")
end
```

The `status_line` method generates the text that should appear based on the meter’s current state. `format_count` returns the current count, with a percentage if an expected total has been set. `format_bytes` displays the number of bytes sent using an appropriate unit, from bytes to gigabytes.

```
# lib/progress.rb

UNITS = ["B", "KiB", "MiB", "GiB"]
SCALE = 1024.0

def status_line
  line = "#{@message}: #{format_count}"
  line.concat(", #{format_bytes}") if @bytes > 0
  line.concat(", done.") if @count == @total
  line
end

def format_count
  if @total
    percent = (@total == 0) ? 100 : 100 * @count / @total
    "#{percent}% (#{@count} / #{@total})"
  else
    "(#{@count})"
  end
end

def format_bytes
  power = Math.log(@bytes, SCALE).floor
  scaled = @bytes / (SCALE ** power)

  format("%.2f #{UNITS[power]}", scaled)
end
```

This completes the `Progress` class, and we can now spread it to other places in the codebase. For example, `ReceiveObjects` could use it to display its progress in extracting objects from the pack.

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

def recv_packed_objects(prefix = "")
  stream = Pack::Stream.new(@conn.input, prefix)
  reader = Pack::Reader.new(stream)
```

```
progress = Progress.new(@stderr) unless @conn.input == STDIN

reader.read_header
progress&.start("Unpacking objects", reader.count)

reader.count.times do
  record, _ = stream.capture { reader.read_record }
  repo.database.store(record)
  progress&.tick(stream.offset)
end
progress&.stop

stream.verify_checksum
end
```

As the pack writing process becomes more complex, these progress meters will become even more valuable in seeing what is going on inside, and seeing how our code is performing.

Our push command is now fully functional, and can interact with Git's `git-receive-pack` program. To demonstrate this, let's initialise a new repository, and add it as a remote to Jit's own repo.

```
$ jit init ../push-test
$ jit config --file ../push-test/.git/config receive.denyCurrentBranch false
$ jit remote add copy file://$PWD/..../push-test
```

If we push our `master` branch to this `copy`, we can see the objects being transferred, and how much data is sent.

```
$ jit push copy master

Counting objects: 100% (1619/1619), done.
Writing objects: 100% (1619/1619), 786.76 KiB, done.
To file:///Users/jcoglan/projects/building-git/jit/..../push-test
 * [new branch] master -> master
```

If we switch into the other repository, we can use `tree` to show the state of the `.git` directory. The `git/objects` directory has stored the packs we sent as files in their own right, rather than as one file per object, and we'll learn more about that when we look at storing packs in the database¹¹.

```
$ cd ..../push-test

$ tree .git
.git
├── HEAD
├── objects
│   └── pack
│       ├── pack-b3671307d1a15a297cdaa1cb554e2bc8aea26db5.idx
│       └── pack-b3671307d1a15a297cdaa1cb554e2bc8aea26db5.pack
└── refs
    └── heads
        └── master
```

To verify the data was transferred correctly, let's log the last few commits:

¹¹Chapter 32, *Packs in the database*

```
$ git log --oneline @~5..
```

```
5121326 (HEAD -> master) Display progress of pack transfers
045a839 Reject updates from `push` based on configuration
fbb572b Push content via the `push` and `receive-pack` commands
a9fa963 Update ref values using compare-and-swap
6768fa6 Retrieve the `remote.<name>.push, receivepack` variables
```

Indeed, everything appears to be working. And, since we added support for SSH URLs in the last chapter, we can run `git remote add origin ssh://git@github.com/jcoglan/jit.git`, and begin pushing the repository to GitHub. Running `git push origin heads/*` will push all our local branches up to the remote repository.

30. Delta compression

When sending a pack of objects over the network, it's likely that the pack will contain a lot of objects that are similar to one another. For example, if a sequence of commits all modify the same file, then we'll be sending multiple versions of that file in the same pack. Those versions may not differ very much from one another; if the changes are small, the versions will have a lot of text in common. Sending objects over the wire takes time, especially over a slow connection, and it would speed things up if we could avoid putting so much repeated data in our packs.

Although objects are compressed individually using zlib when sent over the network and stored on disk, there's an opportunity for shrinking them even more by exploiting similarities across objects. One technique we could use to make a pack smaller, which would be easy to implement, is to compress the entire thing with zlib. That would take advantage of similar strings between objects, as long as the similar objects are placed near each other in the pack¹.

However, as we'll see later, Git also uses packs as on-disk storage for objects² — if you clone a large repository you'll receive a single pack containing the entire history. If an entire pack were compressed with zlib, then we'd need to read the entire thing from the beginning to decompress it and retrieve objects.

Git's commands usually need access to arbitrary objects by ID, and it would slow things down terribly if we had to read the entire database in order to locate a single object. We'd like a way to compress objects that preserves our ability to access them at random. Delta compression does this by providing a way to represent one object as the difference between it and another, but in a way where object records in a pack can still be read from arbitrary locations in the pack, without needing to read everything up to that point first.

30.1. The XDelta algorithm

To compress objects, Git uses a variant on the XDelta algorithm, described in a paper by Joshua P. MacDonald³. It creates a representation of a *target* string in terms of a *source* string, by generating a sequence of operations for reconstructing the target from the source. These operations are either a *copy*, directing us to copy a substring from the source string, or an *insert*, telling us to insert newly added data.

The way XDelta works is best introduced with an example. Suppose we have the following two strings as our source and target. The aim is to find a representation of the target in terms of the source, where this new representation is smaller than the target.

source: the quick brown fox jumps over the slow lazy dog

target: a swift auburn fox jumps over three dormant hounds

We can see that these strings have some content in common. Copy operations consist of just two numbers, the offset and size of the substring to take from the source, and are therefore

¹To limit its memory use when processing large and possibly unending data streams, zlib uses a sliding window of recently seen strings to compress data. If similar substrings in the stream are too far apart, the first will fall out of this window and cannot be used to compress the next.

²Chapter 32, *Packs in the database*

³<http://www.xmailserver.org/xdfs.pdf>, section 3.1

much smaller than insert operations which contain literal chunks of text. To achieve good compression, we'd like to use copy operations as much as possible, so we need to find the common substrings between the two texts.

String searching is a well-studied problem⁴, and there are many well-known algorithms for finding a particular pattern within a string, for example Rabin-Karp⁵, Knuth-Morris-Pratt⁶, and Boyer-Moore⁷. Finding any possible substrings shared by two texts is even harder since there is no pre-defined pattern to search for.

The naive approach is to look at the first character of the target, a, and find where it occurs in the source. The letter a appears at offset 41 in the word `lazy`. In the target it's followed by a space, but in the source we have a `z`, so there's no meaningful substring there. Moving one place forward in the target, we have a space, which appears in the source at offsets 3, 9, 15, 19, 25, 30, 34, 39, and 44. In the target, the next letter is the s of `swift`, and if we examine each of these offsets in the source, we see the space at offset 34 is followed by the s of `slow`. There the similarity stops, so we've found a common substring of length 2.

The nested iteration through the target string, then the source string to find a matching character, then a character-by-character comparison at each matching offset, means this approach to finding common substrings is extremely slow. To get better performance, most search algorithms pre-process the source in some way to gather information about where substrings might start.

In XDelta, we begin by building an index of the source string based on splitting it into 16-byte blocks, and storing the offsets at which those blocks appear. A given block might be found at multiple locations, so this index is a map from blocks to arrays of offsets. For our source string, the index looks like this:

```
{  
    "the quick brown " => [ 0],  
    "fox jumps over t" => [16],  
    "he slow lazy dog" => [32]  
}
```

Only full blocks are indexed, so if the source length is not a multiple of 16, the last few bytes will not be indexed. This index can be reused across multiple target strings, so we only need to build it once for each source.

Having indexed the source, we iterate over the target using a *sliding window*, also of length 16. At each offset in the target, we look at the 16 bytes from that position, and see if that block appears in the index. If it does not, then we add the first byte to the *insert buffer*. Here are the first few steps of this process:

⁴https://en.wikipedia.org/wiki/String-searching_algorithm

⁵https://en.wikipedia.org/wiki/Rabin%20%93Karp_algorithm

⁶https://en.wikipedia.org/wiki/Knuth%20%93Morris%20%93Pratt_algorithm

⁷https://en.wikipedia.org/wiki/Boyer%20%93Moore_string-search_algorithm

Figure 30.1. First few iterations of the matching loop

offset	sliding window	insert buffer
0	"a swift auburn f"	""
1	" swift auburn fo"	"a"
2	"swift auburn fox"	"a "
3	"wift auburn fox "	"a s"
4	"ift auburn fox j"	"a sw"

Eventually, at offset 15 in the target, we find the block "fox jumps over t", which does appear in the index, with offset 16 in the source.

Figure 30.2. Finding the first matching block

offset	sliding window	insert buffer
13	"n fox jumps over"	"a swift aubur"
14	" fox jumps over "	"a swift auburn"
15	"fox jumps over t"	"a swift auburn "

So, we have found a common block between these two strings, as indicated by the parentheses:

source: the quick brown (fox jumps over t)he slow lazy dog

target: a swift auburn (fox jumps over t)hree dormant hounds

Having found this starting block, we then try to expand it. We can see that in both strings, the block is followed by the letter h, so we'll include that into the matching block:

source: the quick brown (fox jumps over th)e slow lazy dog

target: a swift auburn (fox jumps over th)ree dormant hounds

Working in the opposite direction, both blocks are preceded by the letter n and a space, so we'll include those into the block as well.

source: the quick brow(n fox jumps over th)e slow lazy dog

target: a swift aubur(n fox jumps over th)ree dormant hounds

As we've extended the match backwards, we also need to remove the matched string "n " from the end of the input buffer. We emit the input buffer as an *insert* operation, and then the matching block as a *copy*, giving the offset in the source string and the length of the block. We repeat this process after the matched block until we reach the end of the string, emitting anything remaining in the buffer as a final *insert* operation. The end result is:

```
insert "a swift aubur"
copy 14 19
insert "ree dormant hounds"
```

This has replaced a 19-byte chunk of the target with a copy operation, which as we'll see later occupies only 3 bytes. By maximising the amount of common text we find, we can substantially reduce the size of the target string.

30.1.1. Comparison with diffs

Conceptually, this problem looks very similar to generating diffs⁸ between the two strings. However, we need a specialised algorithm here because the algorithms used for diffing perform badly on this problem. While the Myers algorithm performs well on line-oriented text files, many things we'll want to compress, for example tree objects, are not line-oriented text but binary files. Increasing the granularity of the diff inputs from lines to bytes will typically make the input a couple of orders of magnitude larger and severely degrade the speed of the algorithm.

The other limitation of diffs is that they have constraints that are useful for human readability and for conducting merges, but do not produce optimal deltas. One of these constraints is that, for items to be considered equal, they must appear in the same order in both strings. For example, take these two strings that differ only in the order of their words.

```
source: the quick brown fox jumps over the slow lazy dog
```

```
target: over the slow lazy dog the quick brown fox jumps
```

If we take a word-wise diff between these, we get:

```
- the
- quick
- brown
- fox
- jumps
over
the
slow
lazy
dog
+ the
+ quick
+ brown
+ fox
+ jumps
```

A diff algorithm will recognise that over the slow lazy dog is common to the two strings, but treats the quick brown fox jumps as newly inserted words, because it doesn't model the concept of items being moved, copied or reordered. If we convert this into copy/insert operations, we get:

```
copy 26 22
insert " the quick brown fox jumps"
```

XDelta does not mind how many times a block appears in the target, or how blocks are ordered between the two versions. It will spot that the quick brown fox jumps can also be copied from the beginning of the source, replacing a 25-byte block with a 3-byte copy operation. It only generates an insert operation for the space between the two copied sections.

```
copy 26 22
insert " "
copy 0 25
```

⁸Chapter 11, *The Myers diff algorithm*

30.1.2. Implementation

To support the implementation of XDelta, we'll need a couple of structures to represent operations. I'm going to place these structures in the `Pack::Delta` namespace, which will contain logic to connect pairs of objects within a Git pack and apply compression to them. The compression algorithm itself will live in `Pack::XDelta`.

```
# lib/pack/delta.rb

module Pack
  class Delta

    Copy   = Struct.new(:offset, :size)
    Insert = Struct.new(:data)

  end
end
```

Because of the way that these operations are encoded within packs, the maximum space allocated for the size of a copy operation is 3 bytes, and so its maximum value is FFFFF_{16} , or $16,777,215_{10}$. The length of an insert block is contained in 7 bits, so an insert operation can have a size of at most $7F_{16}$, or 127_{10} .

```
# lib/pack.rb

MAX_COPY_SIZE  = 0xffffffff
MAX_INSERT_SIZE = 0x7f
```

To build an index for compressing various targets against a source, we'll have a method called `Pack::XDelta.create_index`. It takes a source string, and returns an `XDelta` instance that contains this source, and an index of its constituent blocks and their offsets.

```
# lib/pack/xdelta.rb

module Pack
  class XDelta

    BLOCK_SIZE = 16

    def self.create_index(source)
      blocks = source.bytesize / BLOCK_SIZE
      index = {}

      (0 ... blocks).each do |i|
        offset = i * BLOCK_SIZE
        slice = source.byteslice(offset, BLOCK_SIZE)

        index[slice] ||= []
        index[slice].push(offset)
      end

      XDelta.new(source, index)
    end

    def initialize(source, index)
      @source = source
      @index = index
    end
  end
end
```

```
    end

    #

end
end
```

A single `xDelta` instance contains a source string and its index, and can be used to generate compressed versions of arbitrary numbers of targets. The `xDelta#compress` method takes a target, and generates and returns a list of operations — `Copy` and `Insert` values — that represent the target in terms of the source. `@offset` tracks the current offset we've reached in the target string, and `@insert` stores the insert buffer.

```
# lib/pack/xdelta.rb

def compress(target)
  @target = target
  @offset = 0
  @insert = []
  @ops = []

  generate_ops while @offset < @target.bytesize
  flush_insert

  @ops
end
```

`generate_ops` is called as long as `@offset` is less than the size of the target. It calls `longest_match` to get the offset and size of the longest match we can find in the source, at our current position in the target. If no matching block is found, then we call `push_insert` which adds to the insert buffer and moves the `@offset` pointer. Otherwise, we try to expand the match in both directions, and then flush any existing insert bytes and emit a `Copy` operation for the offset and size of the match we found.

```
# lib/pack/xdelta.rb

def generate_ops
  m_offset, m_size = longest_match
  return push_insert if m_size == 0

  m_offset, m_size = expand_match(m_offset, m_size)

  flush_insert
  @ops.push(Delta::Copy.new(m_offset, m_size))
end
```

The `longest_match` method looks at the sliding window block at the current offset in the target string, and checks if that block appears in the index. If not, then it returns zero for the match offset and size, indicating no match was found.

If the block does appear in the index, then it tries to find the longest match beginning at any of the indexed positions. Looking up the block in the index gives every position that the block appears in the source. For each such position `pos`, we begin by calling `remaining_bytes` which returns the size of the longest possible copy operation that could begin here: the minimum of the remaining source bytes, the remaining target bytes, and the maximum copy operation size.

If this is smaller than the current best match size, then we skip this position. Otherwise we call `match_from`, which extends the match as far as possible in the forward direction and returns the offset in the source where the match ends. If the difference between this value and `pos` is greater than the current best match size, then we set `pos` as the best offset and update the best match size.

```
# lib/pack/xdelta.rb

def longest_match
    slice = @target.byteslice(@offset, BLOCK_SIZE)
    return [0, 0] unless @index.has_key?(slice)

    m_offset = m_size = 0

    @index[slice].each do |pos|
        remaining = remaining_bytes(pos)
        break if remaining <= m_size

        s = match_from(pos, remaining)
        next if m_size >= s - pos

        m_offset = pos
        m_size   = s - pos
    end

    [m_offset, m_size]
end
```

`remaining_bytes` encodes the calculation of how many bytes we could possibly include in a copy operation beginning at offset `pos` in the source string. This is limited by the number of bytes remaining in the source from `pos`, the bytes remaining in the target from `@offset`, and the maximum copy operation size.

```
# lib/pack/xdelta.rb

def remaining_bytes(pos)
    source_remaining = @source.bytesize - pos
    target_remaining = @target.bytesize - @offset

    [source_remaining, target_remaining, MAX_COPY_SIZE].min
end
```

The method `match_from` takes an offset in the source string, `pos`, and the maximum number of bytes we might include in a match, `remaining`. Starting at the current offsets in the source and target documents, we compare each byte and increment the two pointers as long as the bytes match. This should advance the pointers at least `BLOCK_SIZE` places, but may result in a longer match. At the end, it returns the source offset of the end of the match.

```
# lib/pack/xdelta.rb

def match_from(pos, remaining)
    s, t = pos, @offset

    while remaining > 0 and @source.getbyte(s) == @target.getbyte(t)
        s, t = s + 1, t + 1
        remaining -= 1
    end
```

```
    s  
end
```

The call to `longest_match` in `generate_ops` returns the longest match we can find by expanding a matching block forwards from an indexed position. `expand_match` then attempts to extend the block working backwards, by removing bytes from the insert buffer. As long as the match offset is positive and there is still content in the insert buffer, the target and match offsets are decremented, the match size is incremented, and the last byte is popped off the insert buffer. Once this is complete, we have found the best match, and we can advance the `@offset` pointer by the size of the match, and return the match offset and size to the caller. `generate_ops` takes these values, flushes the contents of the insert buffer, and then appends a `Copy` operation with the given values.

```
# lib/pack/xdelta.rb

def expand_match(m_offset, m_size)
  while m_offset > 0 and @source.getbyte(m_offset - 1) == @insert.last
    break if m_size == MAX_COPY_SIZE

    @offset -= 1
    m_offset -= 1
    m_size += 1

    @insert.pop
  end

  @offset += m_size
  [m_offset, m_size]
end
```

Finally, we have the methods for managing the insert buffer. `push_insert` pushes the byte at the current target offset into the buffer, and increments the pointer. It flushes the buffer if we've reached the maximum insert size. `flush_insert` appends an `Insert` operation with the contents of the buffer, and empties the buffer. If this method is given a size argument, it only flushes the buffer if it contains the given number of bytes.

```
# lib/pack/xdelta.rb

def push_insert
  @insert.push(@target.getbyte(@offset))
  @offset += 1
  flush_insert(MAX_INSERT_SIZE)
end

def flush_insert(size = nil)
  return if size and @insert.size < size
  return if @insert.empty?

  @ops.push(Delta::Insert.new(@insert.pack("C*")))
  @insert = []
end
```

This `XDelta` class can generate a list of operations, but is not concerned with how those operations are encoded when written into a pack. As such, we're able to inspect its output in a structured form and place the encoding logic elsewhere.

```
>> source = "the quick brown fox jumps over the slow lazy dog"
>> target = "a swift auburn fox jumps over three dormant hounds"
>>
>> index = Pack::XDelta.create_index(source)
>> index.compress(target)
=> [ Delta::Insert(data="a swift aubur"), Delta::Copy(offset=14, size=19),
    Delta::Insert(data="ree dormant hounds") ]
```

30.2. Delta encoding

To store a delta in a pack, we need to convert the `Copy` and `Insert` values into strings and concatenate them. An `Insert` operation is encoded directly as a byte containing its size, followed by its contents. The size has a maximum value of $7F_{16}$.

```
# lib/pack/delta.rb

Insert = Struct.new(:data) do
  def to_s
    [data.bytesize, data].pack("Ca*")
  end
end
```

`Copy` operations are more complicated. These values contain two numbers, an `offset` and a `size`. The encoding allocates up to four bytes for the offset and three bytes for the size, which together can be viewed as a seven-byte (56-bit) number. To get the greatest compression efficiency, the encoding uses a variable-length encoding — if we used the full seven bytes to encode the small numbers we saw in our opening example, most of the bytes would be zero. Therefore Git uses a method of packing a number into as few bytes as possible.

As an example, let's say we want to encode the number $2,634,033,908_{10}$, or $9D002AF4_{16}$. Representing this number directly using seven bytes, padding with leading zeroes, gives us:

```
00 00 00 9D 00 2A F4
```

Git's encoding removes all the zero bytes, leaving us with:

```
9D      2A  F4
```

To indicate which bytes of the number have been retained, it then prepends a header byte where a bit is set if the corresponding byte is present. In this example, we've kept the first, second and fourth least-significant bytes, and so the header byte reflects this with the value 00001011_2 . To distinguish `Copy` data from `Insert` data, the most-significant bit of this header is set; the highest value of an insert's first byte is $7F_{16}$, and setting the most-significant bit means a copy's first byte will be at least 80_{16} .

That gives us 10001011_2 as the header byte, or $8B_{16}$. The full encoding of the `Copy` operation consists of this header, followed by the retained bytes of the value in little-endian order, that is, `8B F4 2A 9D`.

The `Numbers::PackedInt56LE` module implements this encoding. Its `write` method returns an array of bytes, but does not set the most-significant bit on the first byte — this is left to the caller. The `read` method takes an `IO` object with a `readbyte` method, and a `header`, which is the first byte of the value. This makes it suitable for reading from a stream just as `Pack::Reader` does.

```
# lib/pack/numbers.rb

module PackedInt56LE
  def self.write(value)
    bytes = [0]

    (0...7).each do |i|
      byte = (value >> (8 * i)) & 0xff
      next if byte == 0

      bytes[0] |= 1 << i
      bytes.push(byte)
    end

    bytes
  end

  def self.read(input, header)
    value = 0

    (0...7).each do |i|
      next if header & (1 << i) == 0
      value |= input.readbyte << (8 * i)
    end

    value
  end
end
```

PackedInt56LE.write works by initialising the bytes array to contain the header byte (initially zero), and then iterating over the bytes of value, from least significant to most significant. For example, if value is 0x9d002af4, then byte inside the loop will take the values 0xf4, 0x2a, 0, 0x9d, 0, 0 and 0. For iterations where byte is non-zero, the corresponding bit in the header byte is set, and the current byte is pushed onto the result array. For example, the byte 0x9d occurs on the fourth iteration of the loop, so the fourth least-significant bit in the header byte is set by performing a bitwise-or with the value 0b1000.

Copy#to_s uses this number format by effectively concatenating its four-byte offset and three-byte size into a single seven-byte number. This is done by bit-shifting the size by 32 bits, and combining the result with the offset with a bitwise-or.

```
# lib/pack/delta.rb

Copy = Struct.new(:offset, :size) do
  def to_s
    bytes = Numbers::PackedInt56LE.write((size << 32) | offset)
    bytes[0] |= 0x80
    bytes.pack("C*")
  end
end
```

The final element of the delta encoding is that the sequence of operations is prefixed with two numbers: the sizes of the source and target strings, which let the receiver of the pack check they've decoded it properly. These numbers are encoded using the VarIntLE format that we

saw in the size headers for pack records⁹, with a small alteration. This format currently assumes the first byte of the encoding contains four bits from the value.

```
# lib/pack/numbers.rb

module VarIntLE
  def self.write(value)
    bytes = []
    mask = 0xf
    shift = 4

    # ...
  end

  def self.read(input)
    first = input.readbyte
    value = first & 0xf
    shift = 4

    # ...
  end
end
```

For delta size headers, the format uses the full seven data bits of the first byte, and we can arrange this by passing `shift` into this module's methods as a parameter, and having it calculate the mask dynamically.

```
# lib/pack/numbers.rb

module VarIntLE
  def self.write(value, shift)
    bytes = []
    mask = 2 ** shift - 1

    # ...
  end

  def self.read(input, shift)
    first = input.readbyte
    value = first & (2 ** shift - 1)

    # ...
  end
end
```

Callers can now use `VarIntLE.write(value, 4)` or `VarIntLE.write(value, 7)` as required.

We now have all the pieces necessary to serialise a delta. Let's look at the example that we opened the chapter with, using a short script that uses `Pack::XDelta` and `Pack::Numbers`.

```
source = "the quick brown fox jumps over the slow lazy dog"
target = "a swift auburn fox jumps over three dormant hounds"

index = Pack::XDelta.create_index(source)
delta = index.compress(target)
```

⁹Section 27.4, “The pack format”

```
s = Pack::Numbers::VarIntLE.write(source.bytesize, 7)
t = Pack::Numbers::VarIntLE.write(target.bytesize, 7)

delta = (s + t).pack("C*") + delta.join("")

print delta
```

If we run this program and pipe it into hexdump we can see that the result is 26_{16} or 38_{10} bytes long, down from the original 50.

```
00000000 30 32 0d 61 20 73 77 69 66 74 20 61 75 62 75 72 |02.a swift aubur|
00000010 91 0e 13 12 72 65 65 20 64 6f 72 6d 61 6e 74 20 |....ree dormant |
00000020 68 6f 75 6e 64 73                                |hounds|
00000026
```

The first two bytes encode the sizes of the strings: 30_{16} is 48_{10} , and 32_{16} is 50_{10} .

```
00000000 30 32                                         |02|
```

The next 14 bytes are an insert operation: $0D_{16}$ is less than 80_{16} , and equals 13_{10} , the length of the following data.

```
00000000          0d 61 20 73 77 69 66 74 20 61 75 62 75 72 | .a swift aubur|
```

After this, we see a 3-byte copy operation. 91_{16} is 10010001_2 , indicating the true little-endian value is $0e\ 00\ 00\ 00\ 13\ 00\ 00$. The first four bytes are the offset, and the rest are the size: E_{16} is 14_{10} , and 13_{16} is 19_{10} .

```
00000010 91 0e 13 |...|
```

Finally, the last insert operation appears, with length header 12_{16} or 18_{10} .

```
00000010          12 72 65 65 20 64 6f 72 6d 61 6e 74 20 | .ree dormant |
00000020 68 6f 75 6e 64 73                                |hounds|
```

What about the example where we reordered the words? If we change the target to over the slow lazy dog the quick brown fox jumps, then the output is:

```
00000000 30 30 91 1a 16 01 20 90 19 |00....|
```

```
00000009
```

This is just 9 bytes to represent a 48-byte string. It consists of $91\ 1a\ 16$, a copy operation of offset 26_{10} and size 22_{10} ; followed by $01\ 20$, an insert operation consisting of a single space; then $90\ 19$, a copy operation with offset 0 and size 25_{10} . Note that the offset of zero means this last operation uses no bytes to encode the offset.

These examples give some indication of why the block size of 16 is used. The block size represents the minimum size of a matching substring that can be found, and making it larger decreases the likelihood of finding matches. But making it smaller damages performance; a block size of 1 is equivalent to the naive approach. We need a trade-off between speed and compression. Since copy operations take up between two and eight bytes, a minimum match size of 16 means compression will make a significant reduction even to the shortest possible matches, while achieving a good-enough speed for practical use. It's also small enough to detect matching entries in tree objects, enabling them to compress very well.

30.3. Expanding deltas

When a delta is read by the receiver—a fetch or receive-pack process—it must be re-applied to the source text to regenerate the target. Let's create a `Pack::Expander` class for performing this task. `Expander.expand` will take two strings, the source and the delta, and initialise a new `Expander` to perform the process. The `Expander` begins by reading the two string sizes from the beginning of the delta. `StringIO` wraps a string with the `I0` interface, letting us call `readbyte` and `read` to progress through the string; this saves us from having to explicitly track the current position in the delta.

```
# lib/pack/expander.rb

module Pack
  class Expander

    attr_reader :source_size, :target_size

    def self.expand(source, delta)
      Expander.new(delta).expand(source)
    end

    def initialize(delta)
      @delta = StringIO.new(delta)

      @source_size = read_size
      @target_size = read_size
    end

    #
    # ...
    #

  end
end
```

`read_size` uses the `VarIntLE.read` method we defined above, which returns the first byte of the number, and the full numeric value. We only want the second value here.

```
# lib/pack/expander.rb

def read_size
  Numbers::VarIntLE.read(@delta, 7)[1]
end
```

The main work of interpreting the delta is done in `Expander#expand`. It takes the source string and scans through the delta, reconstructing the target as it goes. Until we reach the end of the delta, we read a single byte. If this is less than 80_{16} , that means the next chunk is an `Insert` operation, and we pass that first byte and the delta to `Delta::Insert.parse` to read the string data from the operation. If the first byte is at least 80_{16} , then a `Copy` follows. Calling `Delta::Copy.parse` gives us a `Copy` value, and we use its `offset` and `size` to copy a chunk out of the source string.

```
# lib/pack/expander.rb

def expand(source)
  check_size(source, @source_size)
  target = ""
```

```
until @delta.eof?
  byte = @delta.readbyte

  if byte < 0x80
    insert = Delta::Insert.parse(@delta, byte)
    target.concat(insert.data)
  else
    copy = Delta::Copy.parse(@delta, byte)
    target.concat(source.byteslice(copy.offset, copy.size))
  end
end

check_size(target, @target_size)
target
end
```

At either end of this method, we check the sizes of the strings against the sizes recorded at the beginning of the delta. If they disagree, an error is thrown.

```
# lib/pack/expander.rb

def check_size(buffer, size)
  raise "failed to apply delta" unless buffer.bytesize == size
end
```

Parsing an Insert operation is as straightforward as serialising it: we read a number of bytes given by the value of the first byte.

```
# lib/pack/delta.rb

Insert = Struct.new(:data) do
  def self.parse(input, byte)
    Insert.new(input.read(byte))
  end
end
```

To parse a Copy operation, we can use PackedInt56LE.read to read the seven-byte number described by the leading byte. The least-significant four bytes of this number are the offset, and we can select those via a bitwise-and with $FFFFFFFFFF_{16}$. The next three bytes are the size, and we can get this by bit-shifting the number to the right by four bytes, or 32 bits.

```
# lib/pack/delta.rb

Copy = Struct.new(:offset, :size) do
  def self.parse(input, byte)
    value = Numbers::PackedInt56LE.read(input, byte)
    offset = value & 0xffffffff
    size   = value >> 32

    Copy.new(offset, size)
  end
end
```

We now have all the tools we need to generate and interpret deltas. Given some pair of strings, we can compress one of them by generating a delta against the other. To use this to compress

a pack, we need to find pairs of objects that will compress well against one another, and that's what we'll deal with in the next chapter.

31. Compressing packs

In the last chapter, we saw how the XDelta algorithm can be used to compress a target string by expressing it as the difference between it and a source string. If the source and target have a lot of text in common, this can produce a huge size reduction in the target by reducing it to a series of pointers to substrings from the source. To apply this technique to Git packs, we need a method of finding pairs of objects that will form good pairs of strings for XDelta. I will call such pairs of objects *delta pairs*.

In Section 27.4.1, “Writing packs” we introduced the `Pack::Writer` class, which takes a `RevList` and writes all the objects it yields into a pack to be sent over the network. As an example, I’ve picked a sequence of three commits from Jit’s history and generated a `RevList` from them, with the `:objects` option enabled. The following is a list of all the objects that would be packaged up if those three commits were sent in a pack.

be6da09	caef97	6953b7b	4aff4e1	9bd6e40	fa24535
d43eb8a	73eefdf6	2057254	dfe0449	9dd940f	a5986de
6f25967	4e4e36a	cb366b6	65aebd9	3190036	8dad25c
38ccace	5b48c0f	dad209f	dd45083	ac373a6	3c51d0b
220d9c9	6fce34	3a92035	af2a5a6	6b9fcfc	46c59ee
ea59a5c	073c69a	184a945			

These three commits have yielded 33 objects, and to make a start on compressing the pack we need to identify which of these would form good delta pairs. That is, for each target object in the pack, which other object could we use as the source so that XDelta shrinks the target object as much as possible?

Generating a delta is expensive, and trying every possible pair of objects is unfeasible, as the number of possible pairs grows with the square of the number of objects — we say this approach has *quadratic complexity*, or it runs in $O(n^2)$ time¹. A set of 10 objects yields 45 pairs, 100 objects yields 4,950 pairs, 1,000 objects yields 499,500 pairs and so on. In general, algorithms with quadratic complexity cannot be used because as the input size grows, they take an unacceptable amount of time to run.

Acceptable performance on large problems often means finding an $O(n)$ or *linear time* solution — one where if you make the input ten times larger, the algorithm takes ten times longer, rather than a hundred times. We could achieve this if every object were compared to a fixed number of other objects, regardless of the size of the set, rather than trying every possible pair. To do this, we need some way of determining which objects are most likely to make good delta pairs, without actually generating deltas for them.

31.1. Finding similar objects

By reading just the beginning of each object, we can get a little bit of information about it — its type and size — without reading the whole object from disk into memory. By examining how the objects are connected to each other, we can also reconstruct a filename for each tree and blob. Below are the three commits, in ascending chronological order, along with a tree of the new objects that each commit introduces. Each tree entry lists its ID and name; the name / denotes the root tree of a commit.

¹https://en.wikipedia.org/wiki/Time_complexity

```
commit: 6953b7b4a4372e395db1c4ac45563859c3202ee0
└ [6fce434] /
  └ [3a92035] lib
    | └ [fa24535] command.rb
    | └ [af2a5a6] command
    |   └ [6b9fcfc] config.rb
    |   └ [46c59ee] config.rb
    |   └ [dfe0449] config
    |     └ [9dd940f] stack.rb
    |     └ [a5986de] repository.rb
  └ [ea59a5c] test
    └ [073c69a] command
      └ [184a945] config_test.rb

commit: caeef97016c08abad94c8191e4edcd3e3eff5a84
└ [3190036] /
  └ [8dad25c] lib
    | └ [38ccace] command
    |   └ [5b48c0f] config.rb
    | └ [dad209f] config.rb
  └ [dd45083] test
    └ [ac373a6] command
      └ [3c51d0b] config_test.rb
    └ [220d9c9] config_test.rb

commit: be6da0954076535bb33e755ba2e0824fe9fd2c5
└ [4aff4e1] /
  └ [9bd6e40] lib
    | └ [d43eb8a] command
    |   └ [73eefd6] config.rb
    | └ [2057254] config.rb
  └ [6f25967] test
    └ [4e4e36a] command
      └ [cb366b6] config_test.rb
    └ [65aebd9] config_test.rb
```

Blobs and trees do not have filenames built into them; the same blob or tree may be referenced many times from different places, if the same file content has been moved or copied around the project. However, we can generate a *possible* filename for the objects in a pack by looking at how they are reachable just from the commits we're interested in. For example, the objects `46c59ee`, `dad209f` and `2057254` are reachable only via the path `lib/config.rb` from the root tree in their respective commits, so we could reasonably say these blobs represent versions of the file called `lib/config.rb`.

Now, using the object header information and the pathnames above, we can get a more detailed picture of the objects we're dealing with. Below is a list of each object ID along with its type, size, and derived pathname.

be6da09	commit	356	
caeef97	commit	590	
6953b7b	commit	901	
4aff4e1	tree	166	
9bd6e40	tree	845	lib
fa24535	blob	1305	lib/command.rb
d43eb8a	tree	577	lib/command
73eefd6	blob	3249	lib/command/config.rb
2057254	blob	5608	lib/config.rb

dfe0449	tree	36	lib/config
9dd940f	blob	824	lib/config/stack.rb
a5986de	blob	1033	lib/repository.rb
6f25967	tree	356	test
4e4e36a	tree	539	test/command
cb366b6	blob	3478	test/command/config_test.rb
65aebd9	blob	5589	test/config_test.rb
3190036	tree	166	
8dad25c	tree	845	lib
38ccace	tree	577	lib/command
5b48c0f	blob	2750	lib/command/config.rb
dad209f	blob	5204	lib/config.rb
dd45083	tree	356	test
ac373a6	tree	539	test/command
3c51d0b	blob	3118	test/command/config_test.rb
220d9c9	blob	5294	test/config_test.rb
6fceea34	tree	166	
3a92035	tree	845	lib
af2a5a6	tree	577	lib/command
6b9fcfc	blob	2451	lib/command/config.rb
46c59ee	blob	4812	lib/config.rb
ea59a5c	tree	356	test
073c69a	tree	539	test/command
184a945	blob	2364	test/command/config_test.rb

With this information, we can make some reasonable guesses about which objects will be most similar. For a start, blobs are unlikely to have much content in common with trees or commits, and vice versa, so we can rule some pairs out by only combining objects of the same type. Objects with the same path are most likely to be similar to one another: versions of the same file will typically have small differences, and long stretches of common text, so they'll compress very well.

Within versions of the same file, we're likely to get the best compression by combining versions that are next to each other in the commit sequence, as the differences between adjacent versions will usually be smallest. But, because some commits may be on concurrent branches, it's not possible to totally order the versions of each file in this way. Fortunately, there's a good proxy for the historical order of versions: their file size.

Most of the time, software projects and the files within them tend to get larger. This is not always true — code can be deleted, moved, refactored, and so on — but it's true often enough to be a good heuristic for finding similar versions. If we examine the `lib/config.rb` objects in the above list, in the order they appear in the commits, we'll see that their size does indeed increase with time.

46c59ee	blob	4812	lib/config.rb
dad209f	blob	5204	lib/config.rb
2057254	blob	5608	lib/config.rb

So, to prioritise objects that might be similar to each other, Git first sorts the objects in reverse order by type, name and size, producing the following list. By *type* we mean the numeric type: 1 for commits, 2 for trees, 3 for blobs. And by *name* we mean just the filename, not including its parent directories, so all the trees named `command` sort together, as do all the blobs named `config.rb`.

9dd940f	blob	824	lib/config/stack.rb
a5986de	blob	1033	lib/repository.rb

cb366b6	blob	3478	test/command/config_test.rb
3c51d0b	blob	3118	test/command/config_test.rb
184a945	blob	2364	test/command/config_test.rb
65aebd9	blob	5589	test/config_test.rb
220d9c9	blob	5294	test/config_test.rb
73eef6d	blob	3249	lib/command/config.rb
5b48c0f	blob	2750	lib/command/config.rb
6b9fcfc	blob	2451	lib/command/config.rb
2057254	blob	5608	lib/config.rb
dad209f	blob	5204	lib/config.rb
46c59ee	blob	4812	lib/config.rb
fa24535	blob	1305	lib/command.rb
ea59a5c	tree	356	test
dd45083	tree	356	test
6f25967	tree	356	test
3a92035	tree	845	lib
9bd6e40	tree	845	lib
8dad25c	tree	845	lib
4e4e36a	tree	539	test/command
073c69a	tree	539	test/command
ac373a6	tree	539	test/command
38ccace	tree	577	lib/command
af2a5a6	tree	577	lib/command
d43eb8a	tree	577	lib/command
4aff4e1	tree	166	
3190036	tree	166	
6fce34	tree	166	
6953b7b	commit	901	
caef97	commit	590	
be6da09	commit	356	

We've also left one object out of this set: `dfe0449` has a size of just 36 bytes, and is too small to substantially benefit from delta compression. It would be quicker to send this data over the network as-is rather than spend time trying to optimise it.

Objects likely to be similar to each other are now close together in the list, and we can compare each object to a fixed number of its neighbours, rather than to the entire set.

31.1.1. Generating object paths

To implement this sorting method, we're going to need to make an update to the `RevList` class. It currently yields either `Database::Commit` objects, or `Database::Entry` objects representing the ID and type of trees and blobs, without their full content. To generate the filename for each object, we'll need `RevList` to record and emit the path by which it found each tree and blob that it yields. In the `initialize` method we'll create a new hash called `@paths` that will store the first path we saw for each object. In `traverse_tree`, we can then add entries to `@paths` as we read tree entries.

```
# lib/rev_list.rb

def initialize(repo, revs, options = {})
  # ...
  @paths  = {}

  #
end
```

```
def traverse_tree(entry, path = Pathname.new(""))
  @paths[entry.oid] ||= path

  return unless yield entry
  return unless entry.tree?

  tree = @repo.database.load(entry.oid)

  tree.entries.each do |name, item|
    traverse_tree(item, path.join(name)) { |object| yield object }
  end
end
```

Then when we are emitting the pending objects in `RevList#each`, we'll yield the path for each object as a second argument.

```
# lib/rev_list.rb

def each
  limit_list if @limited
  mark_edges_uninteresting if @objects
  traverse_commits { |commit| yield commit }
  traverse_pending { |object| yield object, @paths[object.oid] }
end
```

31.1.2. Sorting packed objects

We can now use the paths yielded by `RevList` to augment the `Entry` class used by `Pack::Writer`. At first, this was a simple struct holding only an object's ID and its type, both of which could be inferred from the output of `RevList`, without consulting the database.

```
# lib/pack/writer.rb

Entry = Struct.new(:oid, :type)
```

To implement the list ordering we saw above, we will also need the object's size. This is not contained in the `RevList` output so we'll need to ask the database for this information. However, we'd like to avoid reading every object into memory fully until we really need it, especially if the repository contains any very large blobs. Therefore, let's add a method called `Database#load_info`, which will act like the `load_raw` method, only without reading the entire object. The argument to `read_object_header` limits how much of the file is read, and we read just enough to get data that will decompress to give us the type and size.

```
# lib/database.rb

def load_info(oid)
  type, size, _ = read_object_header(oid, 128)
  Raw.new(type, size)
end
```

We're now ready to update `Pack::Writer` to generate richer `Entry` objects with all the information we need. In `prepare_pack_list` we'll receive paths for each blob and tree, and in `add_to_pack_list` we can use `Database#load_info` to get the type and size of the object, rather than examining the type of object.

```
# lib/pack/writer.rb
```

```
def prepare_pack_list(rev_list)
  @pack_list = []
  @progress&.start("Counting objects")

  rev_list.each do |object, path|
    add_to_pack_list(object, path)
    @progress&.tick
  end
  @progress&.stop
end

def add_to_pack_list(object, path)
  info = @database.load_info(object.oid)
  @pack_list.push(Entry.new(object.oid, info, path))
end
```

The Entry class will no longer be a simple struct but a rich class, that takes as input an object ID, a Database::Raw value with type and size attributes, and a Pathname. The delta and depth variables will be used to store connections between delta pairs, but can be ignored for the time being.

```
# lib/pack/entry.rb

module Pack
  class Entry

    extend Forwardable
    def_delegators :@info, :type, :size

    attr_reader :oid, :delta, :depth

    def initialize(oid, info, path)
      @oid = oid
      @info = info
      @path = path
      @delta = nil
      @depth = 0
    end

    #
    #
    #

    end
  end
end
```

Ruby's Forwardable² module is a quick way of defining methods that delegate to an object's internal variables. The line `def_delegators :@info, :oid, :type, :size` is equivalent to defining the following methods, that let us get object metadata from the Entry itself, for example by calling `entry.type` rather than `entry.info.type`. This is a common strategy to avoid leaking the internal structure of an object out to its callers.

```
def oid
  @info.oid
end

def type
```

²<https://docs.ruby-lang.org/en/2.3.0/Forwardable.html>

```
    @info.type
  end

  def size
    @info.size
  end
```

The Entry class now exposes all the information we need and we can start work on the Compressor class. Our Pack::Writer will pass Entry objects into this class, and its job is to compress the objects as best it can by generating deltas between them, before the Writer writes them into the pack. It takes a Database and Progress as input, and creates a Window object, which we'll take a look at shortly. The Compressor#add method adds an Entry to its @objects list, if its size is within a range from 50 bytes to 512 megabytes. Objects outside this range are either not worth compressing, or would take so long to compress that Git doesn't try. Files larger than 512 MiB are likely to be binary files that don't compress well anyway.

```
# lib/pack/compressor.rb

module Pack
  class Compressor

    OBJECT_SIZE = 50..0x20000000
    WINDOW_SIZE = 8

    def initialize(database, progress)
      @database = database
      @window = Window.new(WINDOW_SIZE)
      @progress = progress
      @objects = []
    end

    def add(entry)
      return unless OBJECT_SIZE.include?(entry.size)
      @objects.push(entry)
    end

    #
    end
  end
end
```

After adding all its entries to the Compressor, Writer will invoke build_deltas to start the compression process. This begins by sorting the objects as described above, and then trying to generate a delta for each one.

```
# lib/pack/compressor.rb

def build_deltas
  @progress&.start("Compressing objects", @objects.size)

  @objects.sort_by!(&:sort_key)

  @objects.reverse_each do |entry|
    build_delta(entry)
    @progress&.tick
  end
  @progress&.stop
```

```
end
```

The `sort_by!` call above sorts the `@objects` list by the result of `Entry#sort_key`. This contains an array of the sorting values — the numeric type from `TYPE_CODES`, the filename, and the size. I've also included object's directory name, so that files with the same basename are grouped by which directory they're in. Git itself just uses the last 16 bytes of the full pathname here.

```
# lib/pack/entry.rb

def sort_key
  [packed_type, @path&.basename, @path&.dirname, @info.size]
end

def packed_type
  TYPE_CODES.fetch(@info.type)
end
```

31.2. Forming delta pairs

Now that it has sorted the list of objects, `Compressor#build_deltas` can get to work on compressing them. Here's the first eight objects from our sorted list:

9dd940f	blob	824	lib/config/stack.rb
a5986de	blob	1033	lib/repository.rb
cb366b6	blob	3478	test/command/config_test.rb
3c51d0b	blob	3118	test/command/config_test.rb
184a945	blob	2364	test/command/config_test.rb
65aebd9	blob	5589	test/config_test.rb
220d9c9	blob	5294	test/config_test.rb
73eef6	blob	3249	lib/command/config.rb

To compress the objects, Compressor uses a sliding window, which we'll model as a simplified ring buffer³. To begin with, picture an 8-element array, where all the slots are empty. A variable called `offset` points at the beginning of the array.

Figure 31.1. Empty sliding window

```
[ _____, _____, _____, _____, _____, _____, _____, _____ ]  
  ^  
  offset
```

The compressor iterates over the sorted list, adding each one to the window and comparing it to the window's other contents. Initially the window is empty, so we just place the first object `9dd940f lib/config/stack.rb` in it and move the `offset` pointer. When we add an object to the window, we load its complete data from `.git/objects`, but this data is only retained while the object remains in the window, so we don't run out of memory by loading the entire repository at once.

Figure 31.2. Sliding window with one element

```
[ 9dd940f, _____, _____, _____, _____, _____, _____, _____ ]  
  ^  
  offset
```

Next we add the second object `a5986de lib/repository.rb` to the window. As the window still contains the previous object, we can try to make a pair from them.

³https://en.wikipedia.org/wiki/Circular_buffer

Figure 31.3. Sliding window with two elements

[9dd940f, a5986de, _____, _____, _____, _____, _____]
 ^
 offset

A diagram showing a horizontal array of seven slots. The first two slots contain object IDs: 9dd940f and a5986de. The remaining five slots are represented by underscores. An upward-pointing arrow labeled "offset" is positioned above the third slot.

If we use XDelta to generate the delta between 9dd940f and a5986de, we get a string of length 1,000. This is less than the original size of a5986de (1,033), and so we store a delta pair: a5986de is based on 9dd940f.

Now, we add cb366b6 test/command/config_test.rb, with a size of 3,478. This is considerably larger than the other two objects and it does not compress to less than its original size against either of them, so no delta pair is formed.

Figure 31.4. Sliding window with three elements

[9dd940f, a5986de, cb366b6, _____, _____, _____, _____]
 ^
 offset

A diagram showing a horizontal array of seven slots. The first three slots contain object IDs: 9dd940f, a5986de, and cb366b6. The remaining four slots are represented by underscores. An upward-pointing arrow labeled "offset" is positioned above the fourth slot.

We continue in this manner, adding objects to the sliding window and comparing them against the others, keeping the best delta we can find that's smaller than the original object size. After compressing a few more objects, the window is in this state:

Figure 31.5. Sliding window with one slot empty

[9dd940f, a5986de, cb366b6, 3c51d0b, 184a945, 65aebd9, 220d9c9, _____]
 ^
 offset

A diagram showing a horizontal array of eight slots. The first seven slots contain object IDs: 9dd940f, a5986de, cb366b6, 3c51d0b, 184a945, 65aebd9, and 220d9c9. The eighth slot is represented by an underscore. An upward-pointing arrow labeled "offset" is positioned above the eighth slot.

If we add one more object, the offset pointer wraps back around to the start:

Figure 31.6. Sliding window after becoming full

[9dd940f, a5986de, cb366b6, 3c51d0b, 184a945, 65aebd9, 220d9c9, 73eef6]
 ^
 offset

A diagram showing a horizontal array of eight slots. All slots contain object IDs: 9dd940f, a5986de, cb366b6, 3c51d0b, 184a945, 65aebd9, 220d9c9, and 73eef6. An upward-pointing arrow labeled "offset" is positioned above the first slot.

Inserting new objects will then begin replacing existing ones. These lost objects can no longer be used as bases for delta pairs, and their full object data will drop out of memory and be freed by Ruby's garbage collector⁴.

Figure 31.7. Sliding window overwriting old elements

[5b48c0f, a5986de, cb366b6, 3c51d0b, 184a945, 65aebd9, 220d9c9, 73eef6]
 ^
 offset

A diagram showing a horizontal array of eight slots. The first slot contains 5b48c0f, and the remaining seven slots contain the same objects as in Figure 31.6: a5986de, cb366b6, 3c51d0b, 184a945, 65aebd9, 220d9c9, and 73eef6. An upward-pointing arrow labeled "offset" is positioned above the first slot.

Following this process, each object in the list is compressed against one of its near neighbours. Each object is compared to a fixed number of others, and object data is only kept in memory while being used to form new deltas. This gives us linear-time performance and limits the amount of memory we need to complete the process.

Sometimes, these objects will form chains. For example, it may turn out that the three versions of test/command/config_test.rb are chained so that 184a945 is compressed against 3c51d0b, and 3c51d0b in turn is based on cb366b6. Decompressing an object requires fully reconstructing

⁴[https://en.wikipedia.org/wiki/Garbage_collection_\(computer_science\)](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science))

the text of the object it's based on, so if that base object is itself compressed, we need to read the object *it* is based on, and so on. So that reading objects from the pack does not become too expensive, we should cap the length of these chains somehow.

31.2.1. Sliding-window compression

To implement the sliding window, we saw above that Compressor creates a Window object with size 8 in its initialize method. The Window class takes this number and creates an array of that many elements; initially all its slots will contain nil. The @offset variable tracks the current insert position.

```
# lib/pack/window.rb

module Pack
  class Window

    def initialize(size)
      @objects = Array.new(size)
      @offset = 0
    end

    # ...

  end
end
```

In Compressor#build_data, we take a single Entry and try to compress it. First, we load its full data from the database using Database#load_raw. Then we pass the Entry and this data string to Window#add, which returns a target object, wrapping the entry and its data. For each other object currently in the Window, we attempt to form a delta pair from them.

```
# lib/pack/compressor.rb

def build_delta(entry)
  object = @database.load_raw(entry.oid)
  target = @window.add(entry, object.data)

  @window.each { |source| try_delta(source, target) }
end
```

Window#add takes an Entry and a string of data, and returns an Unpacked struct wrapping these values. The purpose of Unpacked is simply to extend an Entry by attaching object data to it—we delegate type, size, delta and depth to the Entry. Unpacked also has a delta_index attribute, which we'll use to store the pre-processed index used in the XDelta algorithm.

```
# lib/pack/window.rb

Unpacked = Struct.new(:entry, :data) do
  extend Forwardable
  def_delegators :entry, :type, :size, :delta, :depth

  attr_accessor :delta_index
end

def add(entry, data)
  unpacked = Unpacked.new(entry, data)
```

```
@objects[@offset] = unpacked
@offset = wrap(@offset + 1)

    unpacked
end

def wrap(offset)
  offset % @objects.size
end
```

To iterate over the other objects in the window, we want to visit the most-recently inserted objects first. As the offset is one position after the just-inserted object, that object will be at offset - 1, and the first one we want to try to pair it with is at offset - 2. So starting at offset - 2, we iterate a cursor downwards until we reach offset - 1, wrapping around to the end of the array.

```
# lib/pack/window.rb

def each
  cursor = wrap(@offset - 2)
  limit = wrap(@offset - 1)

  loop do
    break if cursor == limit

    unpacked = @objects[cursor]
    yield unpacked if unpacked

    cursor = wrap(cursor - 1)
  end
end
```

Compressor#build_delta calls try_delta to attempt to combine two entries in a delta pair. This method takes two Unpacked objects; target is the one we just inserted, and source is one of the others that were already in the window. If the object types differ, or the source depth has overshot the maximum chain length, then we don't attempt to pair these objects. Otherwise, we create a new Delta from them, and check its size. If it's bigger than the target's best delta or original size, it's no good. If it's equal to this size, but would increase the depth (the chain length) of the target, we also ignore it. If all these checks pass, then we attach the Delta to the target's Entry.

```
# lib/pack/compressor.rb

MAX_DEPTH = 50

def try_delta(source, target)
  return unless source.type == target.type
  return unless source.depth < MAX_DEPTH

  delta = Delta.new(source, target)
  size = target.entry.packed_size

  return if delta.size > size
  return if delta.size == size and delta.base.depth + 1 >= target.depth

  target.entry.assign_delta(delta)
```

```
end
```

Pack::Delta is the namespace where we placed the Copy and Insert structs in the previous chapter, and I'm now going to turn it into a class in its own right. It takes two Unpacked values, a source and target, and generates the appropriate delta header for their two sizes, using the Numbers::VarIntLE module. If the source has not already been pre-processed, we pass its data to XDelta.create_index and store the result on the source object — caching this value means each possible source is only indexed once and can be reused to compress multiple targets. Finally, we call XDelta#compress to compress the target, and add all the serialised operations onto the delta.

```
# lib/pack/delta.rb

module Pack
  class Delta

    extend Forwardable
    def_delegator :@data, :bytesize, :size

    attr_reader :base, :data

    def initialize(source, target)
      @base = source.entry
      @data = sizeof(source) + sizeof(target)

      source.delta_index ||= XDelta.create_index(source.data)

      delta = source.delta_index.compress(target.data)
      delta.each { |op| @data.concat(op.to_s) }
    end

    private

    def sizeof(entry)
      bytes = Numbers::VarIntLE.write(entry.size, 7)
      bytes.pack("C*")
    end
  end
end
```

The other methods used in Compressor#try_delta are Entry#packed_size and Entry#assign_delta. packed_size returns the compressed size of the entry if it has a Delta assigned, and its original size otherwise. assign_delta binds a Delta to the Entry and sets the entry's depth to one greater than the source entry's depth. We can access the Entry that another has been compressed against by calling entry.delta.base.

```
# lib/pack/entry.rb

def packed_size
  @delta ? @delta.size : @info.size
end

def assign_delta(delta)
  @delta = delta
  @depth = delta.base.depth + 1
end
```

```
end
```

31.2.2. Limiting delta chain length

The upload-pack and push commands have not yet been updated to use our Compressor class, but with the code we've defined so far it is possible to write a short script that seeds a Compressor with a set of objects and runs it. The below program finds the last commit to modify the file lib/command/config.rb, selects the last three commits up to that point, loads all the objects from this range and feeds them into a Compressor. In fact, this is how I generated the examples for this chapter.

```
git_path = Pathname.new(Dir.getwd).join(".git")
repo     = Repository.new(git_path)

commit = RevList.new(repo, ["lib/command/config.rb"]).first
revs   = "#{ commit.oid }~3..#{ commit.oid }"
range  = RevList.new(repo, [revs], :objects => true)

compress = Pack::Compressor.new(repo.database, nil)
entries  = []

range.each do |object, path|
  info  = repo.database.load_info(object.oid)
  entry = Pack::Entry.new(object.oid, info, path)

  compress.add(entry)
  entries.push(entry)
end

compress.build_deltas
```

By examining the state of the entries list after running this script, it's possible to construct a tree of the delta links between objects. The follow tree shows each object in the pack, with those that use it as a delta base listed as children. For example a5986de has been compressed as a delta against 9dd940f. Each object is followed by its name, and its change in size (or just its original size for uncompressed objects).

Figure 31.8. Delta dependencies from compressing three commits

```
└─ [9dd940f] lib/config/stack.rb (824)
  └─ [a5986de] lib/repository.rb (1033 -> 1000)
  └─ [cb366b6] test/command/config_test.rb (3478)
    └─ [3c51d0b] test/command/config_test.rb (3118 -> 458)
      └─ [184a945] test/command/config_test.rb (2364 -> 12)
    └─ [65aebd9] test/config_test.rb (5589 -> 5126)
      └─ [220d9c9] test/config_test.rb (5294 -> 405)
        └─ [73eef6] lib/command/config.rb (3249 -> 3084)
          └─ [5b48c0f] lib/command/config.rb (2750 -> 50)
            └─ [6b9fcfc] lib/command/config.rb (2451 -> 25)
        └─ [2057254] lib/config.rb (5608 -> 5521)
          └─ [dad209f] lib/config.rb (5204 -> 12)
            └─ [fa24535] lib/command.rb (1305 -> 1039)
              └─ [46c59ee] lib/config.rb (4812 -> 12)
  └─ [ea59a5c] test (356)
    └─ [dd45083] test (356 -> 66)
    └─ [6f25967] test (356 -> 66)
    └─ [4e4e36a] test/command (539 -> 513)
      └─ [073c69a] test/command (539 -> 31)
      └─ [ac373a6] test/command (539 -> 31)
  └─ [3a92035] lib (845)
    └─ [9bd6e40] lib (845 -> 68)
    └─ [8dad25c] lib (845 -> 68)
  └─ [38ccace] lib/command (577)
    └─ [af2a5a6] lib/command (577 -> 32)
    └─ [d43eb8a] lib/command (577 -> 32)
  └─ [4aff4e1] / (166)
    └─ [3190036] / (166 -> 58)
    └─ [6fceaa34] / (166 -> 58)
  └─ [6953b7b] (commit) (901)
    └─ [caefef97] (commit) (590 -> 505)
      └─ [be6da09] (commit) (356 -> 267)
```

This demonstrates some impressive results, for example the object `dad209f` has been shrunk from 5,204 bytes to just 12. This illuminates why the objects are sorted in order of decreasing size. The smaller version will usually be the previous version in the history, and the larger blob will be a copy of that, plus a few lines added. That means the smaller blob can be turned into a series of copy operations and no inserts, which is very small. It also means that later objects get written out uncompressed and are quicker to read, while older objects require delta expansion.

However, some of these delta chains are quite long. To decompress object `6b9fcfc` (`lib/command/config.rb`) we first need to read objects `cb366b6`, `65aebd9`, `220d9c9`, `73eef6` and `5b48c0f`, reconstructing the full text of each object in the chain so we can apply the next delta. This additional expense might be worth it if the resulting objects were highly compressed, but this is not the case. Object `65aebd9` has been modestly shrunk from 5,589 to 5,126 bytes, and `73eef6` from 3,249 to 3,084. Requiring five additional object reads to shrink one object by 5% does not seem like a good trade-off.

One observation we can make from the above data is that objects tend to compress either very well or very poorly — there is not much middle ground. Most of the objects in our example are either compressed by at least 90%, or less than 10%. So one thing we could do to discourage long chains from forming is to discard deltas that do not shrink an object by at least one-half. Then we'd be restricting delta chains to those that provide real compression value.

Below is a method we'll add to `Compressor`, which works out a size threshold that a delta must beat in order to be retained. When the target does not already have an assigned `Delta`, this threshold is half the target's size, minus 20 bytes for the base object ID that will need to be written into the pack. When it does have an existing `Delta`, we just need to improve on that delta's size.

Git goes one step further by adjusting the `max_size` threshold based on the depth of the two `Entry` objects. It is multiplied by the ratio of the source and target depths' difference from `MAX_DEPTH`. If `source.depth` is greater than `target.depth`, this ratio will be less than 1, meaning the source needs to generate an even smaller delta in order to justify the cost of increasing the chain length. But if `source.depth` is less than `target.depth`, the ratio is greater than 1 and the threshold rises; we'll allow a larger delta than the existing one if it results in making the chain shorter.

```
# lib/pack/compressor.rb

def max_size_heuristic(source, target)
  if target.delta
    max_size = target.delta.size
    ref_depth = target.depth
  else
    max_size = target.size / 2 - 20
    ref_depth = 1
  end

  max_size * (MAX_DEPTH - source.depth) / (MAX_DEPTH + 1 - ref_depth)
end
```

When trying to form delta pairs, Git will reject a delta that does not meet this size limit, and it also performs a few checks to avoid generating a delta at all if it will probably be a waste of time. If the `max_size` is zero, this is an impossible target, so we skip this pair. If the target is more than `max_size` bytes larger than the source, it's unlikely we'll be able to generate a delta smaller than `max_size`, so we don't bother. Finally if the target is very much smaller than the source, then they're unlikely to be very similar.

```
# lib/pack/compressor.rb

def try_delta(source, target)
  return unless source.type == target.type
  return unless source.depth < MAX_DEPTH

  max_size = max_size_heuristic(source, target)
  return unless compatible_sizes?(source, target, max_size)

  delta = Delta.new(source, target)
  size = target.entry.packed_size

  return if delta.size > max_size
  return if delta.size == size and delta.base.depth + 1 >= target.depth

  target.entry.assign_delta(delta)
end

def compatible_sizes?(source, target, max_size)
  size_diff = [target.size - source.size, 0].max
```

```
    return false if max_size == 0
    return false if size_diff >= max_size
    return false if target.size < source.size / 32

  true
end
```

After applying these changes, the resulting delta tree has a very different shape. It is flatter; the longest chains are of depth 2 rather than 5. Each chain contains only versions of the same file, as the objects need to be more similar to one another to form pairs. And, although fewer objects have been compressed, the ones that have been are those that compress really well, where most of the size reduction in the overall pack comes from. Applying these constraints results in a marginally larger pack after zlib compression, but it has fewer long chains and those chains that are present are providing real value.

Figure 31.9. Delta dependencies with compression size limits

```
└── [cb366b6] test/command/config_test.rb (3478)
  └── [3c51d0b] test/command/config_test.rb (3118 -> 458)
    └── [184a945] test/command/config_test.rb (2364 -> 12)
└── [65aebd9] test/config_test.rb (5589)
  └── [220d9c9] test/config_test.rb (5294 -> 405)
└── [73eefd6] lib/command/config.rb (3249)
  └── [5b48c0f] lib/command/config.rb (2750 -> 50)
    └── [6b9fcfc] lib/command/config.rb (2451 -> 25)
└── [2057254] lib/config.rb (5608)
  ├── [dad209f] lib/config.rb (5204 -> 12)
  └── [46c59ee] lib/config.rb (4812 -> 12)
└── [ea59a5c] test (356)
  ├── [dd45083] test (356 -> 66)
  └── [6f25967] test (356 -> 66)
└── [3a92035] lib (845)
  ├── [9bd6e40] lib (845 -> 68)
  └── [8dad25c] lib (845 -> 68)
└── [4e4e36a] test/command (539)
  ├── [073c69a] test/command (539 -> 31)
  └── [ac373a6] test/command (539 -> 31)
└── [38ccace] lib/command (577)
  ├── [af2a5a6] lib/command (577 -> 32)
  └── [d43eb8a] lib/command (577 -> 32)
└── [4aff4e1] / (166)
  └── [3190036] / (166 -> 58)
    └── [6fce34] / (166 -> 58)
```

Git also uses some other tricks to shorten chains. For example, if an object is selected as a delta base, it is shifted to the front of the sliding window so it's kept around for longer and is more likely to be used as the base for more deltas. Pushing more deltas to use the same object as their base results in a flatter tree.

31.3. Writing and reading deltas

Now that we have a means of compressing a set of `Pack::Entry` objects, we can integrate it into `Pack::Writer` and update `Pack::Reader` so it can read the results. We'll start by adding a new step to the work performed by the `Writer` class:

```
# lib/pack/writer.rb

def write_objects(rev_list)
  prepare_pack_list(rev_list)
  compress_objects
  write_header
  write_entries
  @output.write(@digest.digest)
end
```

`compress_objects` uses `Compressor` to shrink the objects in `@pack_list`. We just need to add each object and call `build_deltas` to kick off the compression process.

```
# lib/pack/writer.rb

def compress_objects
  compressor = Compressor.new(@database, @progress)
  @pack_list.each { |entry| compressor.add(entry) }
  compressor.build_deltas
end
```

We must also update `write_entry` so that it writes the entry's delta, rather than its original content, if a delta exists. The current implementation writes out each object as a header containing its type and size, followed by the zlib-compressed object data. For so-called *deltified* objects, we still write the type/size header, where the type is 7, and the size is that of the delta, not the original object — the object's original size is contained in the delta data. This is followed by the 20-byte ID of the object the delta is based on, and the zlib-compressed delta data.

For example, to encode object `dad209f`, we'd write a header for type 7 and size 12, which encodes to the single byte `7c`. This is followed by the ID of the base object `2057254`, so we write the bytes `20 57 25` and so on. Then we compress the 12-byte delta, `e8 2b d4 28 b0 5a 09 b3 ee 0a fa 0a`, using zlib and that completes this object.

In order for the receiver to read this object, it needs to resolve the delta relative to the object `2057254`, and so it needs to have a copy of `2057254` before it attempts to read `dad209f`. That means, the receiver must already have `2057254` in its database, or `2057254` must appear before `dad209f` in the pack. Usually, Git packs are *self-contained*, meaning any object used as a delta base appears in the same pack, so the pack can be decoded without reference to other sources. Making the pack readable thus means every object must appear before any others that use it as a delta base.

In `write_entry`, we can ensure this by recursively calling `write_entry` with the entry's base, if it's deltified, before writing the current entry. To prevent objects being included multiple times, we store the offset at which each object is written on the `Entry` object, and bail out if the `Entry` already has an `offset` assigned.

After that, we can proceed to write out the object record. To get the record's data, we either use the entry's delta, or get its original data by calling `Database#load_raw`. We put the `packed_size` and `packed_type` in the header; these return the compressed size and delta type for deltified objects, and the original size and type otherwise. After the header, we write the `delta_prefix`, which is the base object ID for deltified objects and empty otherwise. Finally we write out the object data.

```
# lib/pack/writer.rb

def write_entry(entry)
  write_entry(entry.delta.base) if entry.delta

  return if entry.offset
  entry.offset = @offset

  object = entry.delta || @database.load_raw(entry.oid)

  header = Numbers::VarIntLE.write(entry.packed_size, 4)
  header[0] |= entry.packed_type << 4

  write(header.pack("C*"))
  write(entry.delta_prefix)
  write(Zlib::Deflate.deflate(object.data, @compression))

  @progress&.tick(@offset)
end
```

To support this method, we've amended `Entry#packed_type` to return the value `REF_DELTA` if the entry is deltified. Similarly, `delta_prefix` returns the base object's ID, packed to 20 bytes. We also need to add the `offset` attribute to store where each entry was placed.

```
# lib/pack/entry.rb

attr_accessor :offset

def packed_type
  @delta ? REF_DELTA : TYPE_CODES.fetch(@info.type)
end

def delta_prefix
  @delta ? [@delta.base.oid].pack("H40") : ""
end
```

`REF_DELTA` is another object type constant stored alongside the basic object types.

```
# lib/pack.rb

COMMIT = 1
TREE   = 2
BLOB   = 3

REF_DELTA = 7
```

When reading the pack, we need to detect these records with the `REF_DELTA` type, and handle them properly. These records are not full objects and need further processing, so we distinguish them by returning a different type of value: a `RefDelta`, which contains the base object ID and the delta data.

```
# lib/pack/reader.rb

def read_record
  type, _ = read_record_header

  case type
  when COMMIT, TREE, BLOB
```

```
    Record.new(TYPE_CODES.key(type), read_zlib_stream)
when REF_DELTA
  read_ref_delta
end
end

def read_ref_delta
  base_oid = @input.read(20).unpack("H40").first
  RefDelta.new(base_oid, read_zlib_stream)
end
```

RefDelta is a simple struct that holds the two values necessary to rebuild the original object. We could make Pack::Reader rebuild the object itself, but as we'll see in the following chapter, there are multiple ways of storing the contents of the pack, requiring different methods of storing and retrieving the contents of base objects. Rather than coupling Pack::Reader to these concerns, we'll just make it return its immediate content with enough structure that other modules can process it further.

```
# lib/pack.rb

RefDelta = Struct.new(:base_oid, :delta_data)
```

Pack::Reader#read_record can now return non-object values, and its caller ReceiveObjects#recv_packed_objects needs to accommodate this change. Before this glue module becomes more complicated, we'll extract the logic for unpacking the objects from the pack into its own class, Pack::Unpacker. This class takes a Database, Pack::Reader, Pack::Stream and Progress as inputs, and progress is not used if the pack is being read from standard input.

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

def recv_packed_objects(prefix = "")
  stream  = Pack::Stream.new(@conn.input, prefix)
  reader  = Pack::Reader.new(stream)
  progress = Progress.new(@stderr) unless @conn.input == STDIN

  reader.read_header

  unpacker = Pack::Unpacker.new(repo.database, reader, stream, progress)
  unpacker.process_pack
end
```

The Unpacker class takes the Database, Pack::Reader, Pack::Stream and Progress, and unpacks the contents of the pack into the database. Each value returned by Reader#read_record is passed to a method called resolve, which converts any deltified records into full objects. The result of this resolve call is stored in the database.

```
# lib/pack/unpacker.rb

module Pack
  class Unpacker

    def initialize(database, reader, stream, progress)
      @database = database
      @reader = reader
      @stream = stream
```

```
    @progress = progress
end

def process_pack
  @progress&.start("Unpacking objects", @reader.count)

  @reader.count.times do
    process_record
    @progress&.tick(@stream.offset)
  end
  @progress&.stop

  @stream.verify_checksum
end

def process_record
  record, _ = @stream.capture { @reader.read_record }

  record = resolve(record)
  @database.store(record)
end

# ...
end
end
```

The Unpacker#resolve method returns Record values unmodified. For RefDelta values, it looks up the given base ID in the database, and uses Pack::Expander to apply the delta to that object. It returns the result as a Record, ready for saving to the database.

```
# lib/pack/unpacker.rb

def resolve(record)
  case record
  when Record then record
  when RefDelta then resolve_ref_delta(record)
  end
end

def resolve_ref_delta(delta)
  resolve_delta(delta.base_oid, delta.delta_data)
end

def resolve_delta(oid, delta_data)
  base = @database.load_raw(oid)
  data = Expander.expand(base.data, delta_data)

  Record.new(base.type, data)
end
```

This demonstrates why objects need to be written to the pack with delta base objects coming before those that are based on them. That call to Database#load_raw with the base object ID needs to succeed before we can rebuild the current object, so we must have already extracted the base object from the pack. If the objects were not ordered this way, then we'd need to scan through the pack multiple times to identify all the non-delta records, save them, and use them to expand the remaining records.

A quick test of our push command shows the difference that delta compression makes to the pack size. On the commit before we enabled compression, this is the result of pushing the whole history to another repository:

```
Counting objects: 100% (1684/1684), done.  
Writing objects: 100% (1684/1684), 820.84 KiB, done.  
To file:///Users/jcoglan/projects/building-git/jit/..push-test  
* [new branch] master -> master
```

After enabling compression, with one more commit introducing 12 new objects, we see the amount of data transferred is reduced from 821 KiB to 333 KiB:

```
Counting objects: 100% (1696/1696), done.  
Compressing objects: 100% (1662/1662), done.  
Writing objects: 100% (1696/1696), 333.11 KiB, done.  
To file:///Users/jcoglan/projects/building-git/jit/..push-test  
* [new branch] master -> master
```

So even though the records stored in packs are already using zlib compression, the use of XDelta to reduce duplicate content makes a very big difference to the amount of data we need to send over the network to synchronise repositories.

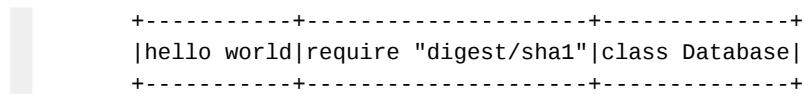
32. Packs in the database

At the end of Chapter 29, *Pushing changes* we tried out our new push command and saw that `git-receive-pack` saved our pack directly to disk in the directory `.git/objects/pack`, rather than extracting its contents into a single file per object. It does this because packs are a much more efficient representation of the contents of a version control repository than individual files, for two reasons.

First, many of the objects in a repository will have large blocks of content in common, and storing each version of a file in full means we're storing a lot of duplicate content. We used delta compression¹ to exploit this duplication and make packs much smaller.

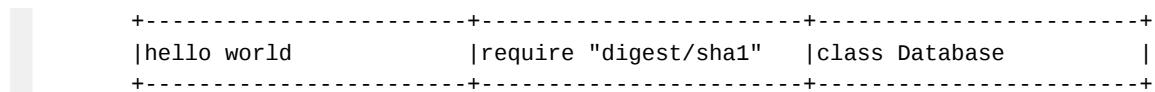
The second reason is down to how filesystems work. When the computer allocates storage space for a file, it does so in fixed-size *blocks*², rather than storing each file exactly end-to-end. This makes it easy to allow files to change size without chopping them into lots of tiny pieces. Imagine some files were stored with no space between the end of one file and the start of another:

Figure 32.1. Files stored end-to-end



If we want to add some more content to the end of the first file, we could store that somewhere else and link the two chunks back together later. However, reading files goes faster when each file is stored contiguously on disk, without fragmentation. To avoid fragmenting the file, we should move all the files after it to create space, but that is a costly operation. Instead, files are allocated in blocks, so a file may not use all its allocated space, but it's easier to resize it before more blocks are required.

Figure 32.2. Files stored in fixed-size blocks



Right now, I am using a MacBook with the stock filesystem settings, and it allocates at least 4 KiB (4,096 bytes) for each file. Using `find` and `du`³ we can see how many files are currently stored in `.git/objects`, and how much disk space they're taking up.

```
$ find .git/objects -type f | wc -l  
1696  
  
$ du -hd 0 .git/objects  
6.6M .git/objects
```

6.6 MiB is 4 KiB for each file, but it turns out that all the files in `.git/objects` are actually smaller than 4 KiB. In fact the median size is around 400 bytes, just one tenth of the space

¹Chapter 30, *Delta compression*

²[https://en.wikipedia.org/wiki/Block_\(data_storage\)](https://en.wikipedia.org/wiki/Block_(data_storage))

³<https://manpages.ubuntu.com/manpages/bionic/en/man1/du.1.html>

allocated for the file. The following command shows us the total size of the content of the files; `stat -f %z` prints each file's size in bytes⁴, and we use a little awk⁵ program to sum these values.

```
$ find .git/objects -type f | xargs stat -f '%z' | awk '{ s += $1 } END { print s }'  
912248
```

This is just 891 KiB of actual content, for which we've used 6.6 MiB of disk space. As our repositories get larger this may become a serious problem, and a more compact storage format becomes necessary.

If we push the current history to another repository, we can see the size of the pack that's sent over the wire.

```
$ jit init ../push-test  
  
$ jit push file://$PWD/..../push-test master  
Counting objects: 100% (1696/1696), done.  
Compressing objects: 100% (1662/1662), done.  
Writing objects: 100% (1696/1696), 333.11 KiB, done.
```

Both `.git/objects` files and packs use zlib compression, but delta compression between objects has shrunk 891 KiB of content down to 333 KiB. Compared to the 6.6 MiB that `.git/objects` is currently consuming, this represents a 95% reduction in size. In a packed repository, it's not uncommon for the pack containing the entire history to consume less space than the latest commit when decompressed and checked out into the workspace.

To use packs as a primary storage system for repositories, we need two new pieces of infrastructure. First, recall that packs do not contain object IDs — the IDs are regenerated from the content by the receiving repository. In order to find objects quickly, we'll need an index saying where in the file each object begins, otherwise we'd need to scan every object in a pack from the beginning until we found one that hashed to the requested ID. Second, we need to extend the `Database` class so that it can serve objects from packs, as well as from individual object files, without any changes to the classes that call the `Database`.

32.1. Indexing packs

Currently, the `recv_packed_objects` method used by `fetch` and `receive-pack` uses the `Pack::Unpacker` class to read each object out of the pack and save it to the database, as a file in `.git/objects`. When a delta is encountered, its base object is reloaded from disk and used to expand the delta.

```
# lib/command/shared/receive_objects.rb  
  
def recv_packed_objects(prefix = "")  
  stream = Pack::Stream.new(@conn.input, prefix)  
  reader = Pack::Reader.new(stream)  
  progress = Progress.new(@stderr) unless @conn.input == STDIN  
  
  reader.read_header
```

⁴<https://manpages.ubuntu.com/manpages/bionic/en/man1/stat.1.html>

⁵<https://manpages.ubuntu.com/manpages/bionic/en/man1/awk.1posix.html>

```
unpacker = Pack::Unpacker.new(repo.database, reader, stream, progress)
unpacker.process_pack
end
```

To keep the pack on disk rather than unpacking it into individual objects, we'll need to allocate a temporary file in `.git/objects/pack`. Once the whole pack has been received, we'll rename this temporary file to `pack-<hash>.pack`, where `<hash>` is the checksum at the end of the file. This ensures each pack we receive gets a unique filename, just like objects do. Next to this file we'll generate another called `pack-<hash>.idx`. This will contain an index of all the object IDs that exist in the pack, and where each object begins. Using this index we'll be able to look up the offset for an object, then `seek()` to that position in the `.pack` file and read the requested record.

32.1.1. Extracting TempFile

To create these temporary files, we can reuse some logic we already created in the very first commit⁶. The `Database#write_object` method contains code that allocates a file called `tmp_obj_<random_id>`, writes to it, then renames it. This process makes the creation of the object file appear atomic.

```
# lib/database.rb

TEMP_CHARS = ("a".."z").to_a + ("A".."Z").to_a + ("0".."9").to_a

def write_object(oid, content)
  path = object_path(oid)
  return if File.exist?(path)

  dirname = path.dirname
  temp_path = dirname.join(generate_temp_name)

  begin
    flags = File::RDWR | File::CREAT | File::EXCL
    file = File.open(temp_path, flags)
    rescue Errno::ENOENT
      Dir.mkdir(dirname)
      file = File.open(temp_path, flags)
    end

    compressed = Zlib::Deflate.deflate(content, Zlib::BEST_SPEED)
    file.write(compressed)
    file.close

    File.rename(temp_path, path)
  end

  def generate_temp_name
    "tmp_obj_#{(1..6).map { TEMP_CHARS.sample }.join("")}"
  end
end
```

Let's extract all the file-handling logic into a new class called `TempFile`, leaving only the code for writing the necessary content and selecting the ultimate name of the file.

```
# lib/database.rb
```

⁶Section 3.2.1, “Storing blobs”

```
def write_object(oid, content)
  path = object_path(oid)
  return if File.exist?(path)

  file = TempFile.new(path.dirname, "tmp_obj")
  file.write(Zlib::Deflate.deflate(content, Zlib::BEST_SPEED))
  file.move(path.basename)
end
```

TempFile will be created taking a Pathname for the directory in which to create the file, and the prefix of the file's name. It appends a random ID to that name and stores the result as @path.

```
# lib/temp_file.rb

class TempFile
  TEMP_CHARS = ("a".."z").to_a + ("A".."Z").to_a + ("0".."9").to_a

  def initialize(dirname, prefix)
    @dirname = dirname
    @path = @dirname.join(generate_temp_name(prefix))
    @file = nil
  end

  def generate_temp_name(prefix)
    id = (1..6).map { TEMP_CHARS.sample }.join("")
    "#{prefix}_#{id}"
  end

  #
end
```

Its write method opens the file for writing if it does not already exist, implicitly creating its parent directory, before writing the given content to it. move closes the open file and renames it as requested.

```
# lib/temp_file.rb

def write(data)
  open_file unless @file
  @file.write(data)
end

def move(name)
  @file.close
  File.rename(@path, @dirname.join(name))
end

def open_file
  flags = File::RDWR | File::CREAT | File::EXCL
  @file = File.open(@path, flags)
rescue Errno::ENOENT
  Dir.mkdir(@dirname)
  retry
end
```

This class is a sufficient abstraction for us to store packs and indexes as we receive them.

32.1.2. Processing the incoming pack

In Git, a process that's receiving a pack will sometimes unpack it into individual objects, and sometimes it will store it verbatim and generate an index for it. It decides on a course of action using the `unpackLimit` variables: `fetch.unpackLimit` controls what the `fetch` command does, and `receive.unpackLimit` controls `receive-pack`. Both of these default to the value of `transfer.unpackLimit` if it is set, or 100 otherwise.

As the pack begins with a header giving the number of objects it contains, the receiver can decide how to process the pack before reading any of its contents. If the number of objects is less than the unpack limit, then it's extracted into individual files. Otherwise, the pack is stored directly.

At first, we won't use the default value of 100. We currently do not have any logic to let the `Database` read objects from packs, so saving them to disk is fairly useless. We'll only do this if an `unpackLimit` variable is explicitly set, letting us opt into this behaviour and test it, before making it the default. In the `fetch` command, we'll pass the relevant variable into `recv_packed_objects`:

```
# lib/command/fetch.rb

def recv_objects
  unpack_limit = repo.config.get(["fetch", "unpackLimit"])
  recv_packed_objects(unpack_limit, Pack::SIGNATURE)
end
```

And similarly in `receive-pack`:

```
# lib/command/receive_pack.rb

def recv_objects
  @unpack_error = nil
  unpack_limit = repo.config.get(["receive", "unpackLimit"])
  recv_packed_objects(unpack_limit) if @requests.values.any?(&:last)
  report_status("unpack ok")
rescue => error
  @unpack_error = error
  report_status("unpack #{error.message}")
end
```

The `ReceiveObjects` module can now accept this argument and make a decision about processing the pack. Instead of always creating a `Pack::Unpacker` to handle the incoming stream, the `select_processor_class` method now chooses between `Pack::Unpacker` and the new class `Pack::Indexer`, based on the given unpack limit or `transfer.unpackLimit`. Once a class is selected, it's instantiated as before with a `Database`, `Pack::Reader`, `Pack::Stream` and `Progress` and then we call `process_pack` on the resulting object.

```
# lib/shared/receive_objects.rb

def recv_packed_objects(unpack_limit = nil, prefix = "")
  stream = Pack::Stream.new(@conn.input, prefix)
  reader = Pack::Reader.new(stream)
  progress = Progress.new(@stderr) unless @conn.input == STDIN
```

```
reader.read_header

factory = select_processor_class(reader, unpack_limit)
processor = factory.new(repo.database, reader, stream, progress)

processor.process_pack
end

def select_processor_class(reader, unpack_limit)
  unpack_limit ||= transfer_unpack_limit

  if unpack_limit and reader.count > unpack_limit
    Pack::Indexer
  else
    Pack::Unpacker
  end
end

def transfer_unpack_limit
  repo.config.get(["transfer", "unpackLimit"])
end
```

All we need now is a `Process::Indexer` class that has the same interface for `new` and `process_pack` as `Pack::Unpacker`, but does something different with its inputs.

32.1.3. Generating the index

Whereas `Pack::Unpacker` converts the incoming pack into a set of files in `.git/objects`, each object in its own file, the job of `Pack::Indexer` is to write the pack directly to disk in `.git/objects/pack`, and generate an index for it. To do this, it will need to extract information from the pack as it's streaming in. It will need to determine each object's ID, which means any deltified objects need to be reconstructed and then hashed, and it will need to record how many bytes into the pack each object begins, so it can be found without re-parsing the entire file.

We'll start by storing the inputs the `Indexer` class is given, and creating some variables to hold the metadata that's gathered while scanning the pack. `@index` will store the ID and starting offset of each non-delta object, and `@pending` will store a list of positions of deltified objects that are based on a certain ID. A first scan through the pack will populate these structures, and we can make a second pass to reconstruct any pending deltified objects.

```
# lib/pack/indexer.rb

module Pack
  class Indexer

    def initialize(database, reader, stream, progress)
      @database = database
      @reader = reader
      @stream = stream
      @progress = progress

      @index = {}
      @pending = Hash.new { |hash, oid| hash[oid] = [] }

      @pack_file = PackFile.new(@database.pack_path, "tmp_pack")
```

```
    @index_file = PackFile.new(@database.pack_path, "tmp_idx")
  end

  #
end
end
```

Database#pack_path returns the path to the directory where packs and their indexes should be stored:

```
# lib/database.rb

def pack_path
  @pathname.join("pack")
end
```

The PackFile class is a decorator for the TempFile class that computes the SHA-1 hash of the written content and stores it at the end of the file. Both the .pack and .idx file end with a hash of their contents, and the hash of the .pack is used to name both the final files.

```
# lib/pack/indexer.rb

class PackFile
  attr_reader :digest

  def initialize(pack_dir, name)
    @file = TempFile.new(pack_dir, name)
    @digest = Digest::SHA1.new
  end

  def write(data)
    @file.write(data)
    @digest.update(data)
  end

  def move(name)
    @file.write(@digest.digest)
    @file.move(name)
  end
end
```

The Pack::Indexer#process_pack method is what recv_packed_objects calls to set the indexer or unpacker running. In this class, it performs a few steps that can be divided into three phases. In the first phase, we read in the incoming pack, write its objects to disk, and while doing this we gather information about the pack's contents. In the second phase, we use the non-delta records to expand any deltified objects that depend on them to fully recover all the pack's object IDs. Finally, the generated index is written to a file stored next to the pack.

```
# lib/pack/indexer.rb

def process_pack
  write_header
  write_objects
  write_checksum

  resolve_deltas
```

```
    write_index  
end
```

The `write_header` method is much the same as that in `Pack::Writer`. We take the object count we've read from the beginning of the pack and write it to the file.

```
# lib/pack/indexer.rb  
  
def write_header  
  header = [SIGNATURE, VERSION, @reader.count].pack(HEADER_FORMAT)  
  @pack_file.write(header)  
end
```

The more interesting work begins in `write_objects`. For the number of objects given in the pack's header, we call `index_object`, wrapped in a progress meter that shows how much data we've received using the `offset` property from the `Pack::Stream`. `index_object` records this offset, which is the position in the pack that the next object begins. It then calls `Reader#read_record` via `Stream#capture` to parse the next object and get the chunk of the pack data that represents it. The data is written to the pack file on disk, and then we take action based on the kind of record we received.

If the object is a `Record`, a non-delta object, then we can immediately hash it and store the offset and CRC32⁷ for the object's ID. The index stores a CRC32 for each record as a checksum on its pack data, so that records can potentially be copied in compressed form between packs, without being expanded and then re-compressed. If the object is a `RefDelta`, then we can't calculate its ID yet as that would require expanding it. So, we just store its offset and CRC32 against the ID of the object it's based on in the `@pending` set.

```
# lib/pack/indexer.rb  
  
def write_objects  
  @progress&.start("Receiving objects", @reader.count)  
  
  @reader.count.times do  
    index_object  
    @progress&.tick(@stream.offset)  
  end  
  @progress&.stop  
end  
  
def index_object  
  offset      = @stream.offset  
  record, data = @stream.capture { @reader.read_record }  
  crc32       = Zlib.crc32(data)  
  
  @pack_file.write(data)  
  
  case record  
  when Record  
    oid = @database.hash_object(record)  
    @index[oid] = [offset, crc32]  
  when RefDelta  
    @pending[record.base_oid].push([offset, crc32])  
  end
```

⁷https://en.wikipedia.org/wiki/Cyclic_redundancy_check

```
end
```

We don't expand deltas right away, as that requires retrieving the data for the base object. We don't want to hold the whole pack in memory just in case each object might be used as a delta base later on, so we instead record this information in the first pass and resolve deltas after all the pack data has been stored on disk.

Once all the pack's objects have been read, `write_checksum` writes the hash at the end of the pack file and renames it, using the hash to generate the filename. To complete the next phase, we'll want to retrieve objects from this file, so we replace our current Reader — the one connected to the incoming data stream — with a new one that reads from the file on disk.

```
# lib/pack/indexer.rb

def write_checksum
  @stream.verify_checksum

  filename = "pack-#{ @pack_file.digest.hexdigest }.pack"
  @pack_file.move(filename)

  path      = @database.pack_path.join(filename)
  @pack    = File.open(path, File::RDONLY)
  @reader = Reader.new(@pack)
end
```

We can now reload records from the pack by seeking to a known offset in the pack file, and asking the new Reader to read from that position.

```
# lib/pack/indexer.rb

def read_record_at(offset)
  @pack.seek(offset)
  @reader.read_record
end
```

32.1.4. Reconstructing objects

After writing the whole pack out to disk, we have a set of metadata about its contents. `@index` contains the object IDs of all non-delta objects and their offsets in the pack file, and `@pending` contains the IDs of all objects used as delta bases, with the offsets of all the objects based on each ID. To reconstruct all the delta objects and discover their IDs, we can take all the non-delta objects and use them to expand all the deltas based on them.

```
# lib/pack/indexer.rb

def resolve_deltas
  deltas = @pending.reduce(0) { |n, (_, list)| n + list.size }
  @progress&.start("Resolving deltas", deltas)

  @index.to_a.each do |oid, (offset, _)|
    record = read_record_at(offset)
    resolve_delta_base(record, oid)
  end
  @progress&.stop
end
```

`resolve_delta_base` takes a `Pack::Record` and its object ID, and checks if there are any pending deltas that are based on it — if `@pending` does not contain the given ID, then we can return. Otherwise, for each delta record that's based on this object, we call `resolve_pending`.

```
# lib/pack/indexer.rb

def resolve_delta_base(record, oid)
  pending = @pending.delete(oid)
  return unless pending

  pending.each do |offset, crc32|
    resolve_pending(record, offset, crc32)
  end
end
```

`resolve_pending` takes a `Record` containing the data for the base object, and the offset and CRC32 of the delta record. It reloads the `RefDelta` from the pack on disk, and uses `Expander` to rebuild the deltified object's original content. From this we can build a new `Record` and hash it to get its object ID, and we can store this ID and the offset into `@index`.

Not all deltas will be based on non-delta records — some will be based on objects that are themselves deltified. So, while we have the reconstructed object in memory, and we've discovered its ID, we can use it to expand further deltas via a recursive call to `resolve_delta_base`, passing the `Record` we just generated. In this way, long chains of dependent deltas can be rebuilt.

```
# lib/pack/indexer.rb

def resolve_pending(record, offset, crc32)
  delta = read_record_at(offset)
  data = Expander.expand(record.data, delta.delta_data)
  object = Record.new(record.type, data)
  oid = @database.hash_object(object)

  @index[oid] = [offset, crc32]
  @progress&.tick

  resolve_delta_base(object, oid)
end
```

Note that since this process of resolving deltas is done once the whole pack is on disk, it does not actually matter if a delta record appears before the object it's based on — it's resilient against implementations that don't sort their packs effectively. However, to make it easier for other clients to read packs, we'll continue to have the `Writer` class sort objects so that deltas always appear after their base objects.

At the end of this process, `@pending` should be empty and `@index` should contain the offset of every object in the pack, and we can now write that information out to a file.

32.1.5. Storing the index

We now have a structure mapping every object ID in the pack to its offset and the CRC32 of its packed bytes. In Ruby, this structure looks something like this:

```
{
```

```
"cffacfe48418bf5b6e2821d760b6685e9e117a1a" => [ 12, 1483935663],
"2d2735624992d5ee494b2b3692287bae3500360c" => [ 613, 994999864],
"fe3702f7d6b929af0612da7597a395178920ab33" => [1120, 3642074317],
"54be6da57e3212fb5108b0c7cf72739f1c4dfe8" => [2110, 4083994585],
"22223621ee0300b08cc89f154cf8686653b6aaa" => [2476, 2463337663],
"7e19f7a6ce33d3755c3f75c3cc74c5757b055da7" => [2944, 979754138],
"8dd4387d6802bc3a1e1ec1c795de27bbf4c9aaaf7" => [3863, 1020950003],
"e41723ba17427114634aec6ee1806933e734369" => [4395, 3322252285],
# ...
}
```

This structure is a *hash table*⁸, and most high-level languages provide some built-in form of it. Given some key, this structure is designed to take *constant time* on average to locate the key's value, meaning it takes the same amount of time no matter how many entries are present. If this data was in an array, it would take *linear time* to find an entry as we'd need to search through the array one place at a time.

We've already encountered a couple of file formats that represent data that we hold in memory as tables: the `.git/index` file which lists every checked-out file in the workspace, and tree objects, which list the version of each file in a directory. These files are usually relatively small, and so reading the whole file from disk and turning it into a table is fine. In contrast, pack indexes can be very large, for example, a recent clone of the Linux kernel repository results in a pack file of over 2 GiB, and its index is almost 200 MiB. The `.git/index` for a recent commit is around 6 MiB. The `.git/index` size grows with the number of files in the latest commit, whereas pack indexes grow with the number of objects in the entire history.

We need a file format that lets us look up an object ID quickly without reading the entire file from disk. Specifically we need it to be faster than a linear scan that reads items sequentially until it finds the right ID. The most common way to do this is to store the values in sorted order, for example:

```
0017c6dc181fd13df81fe804c21b82af0f6db5f2
001ddd0b5ed61683f8d75fe16c661f90893728dc
002f3d6c5bd6de6715c3a44c7c56711b823532ee
007b7c8aca328362060efffc2697b6ff6f4653ef0
0148dff020d1ce12464239c792e23820183c0829
016b21200c0ab572aec57315c9106e312de92cc8
01dbb5bd6aa8d19294402104f2d009dc2c92d075
02b807f0364391f42cbc11eaed97470b4232ea4
...
fe3702f7d6b929af0612da7597a395178920ab33
fe47a1517afc2d2c44c47faafb53edac4d690019
ff01d05dc53ce90f85cd8cdeebbb4fdd93d5f4dc
ff3734e5558bd9d9f909c8f7a0eddeaaf46e9e62
ff47f6fabf7ef3e7c5e5be150d5f398607458899
ff67445e02dadd2c575239c3386dc2d989591219
ffd3bad2a54f2efe27d8f85334a9000b33403d3e
fff1eeb9a60df7667516ede342300f67362be8a6
```

Sorting the IDs lets us use *binary search*⁹ to find items: we begin in the middle of the list, and see if the value there is higher or lower than the one we're looking for. If it's too high, then we narrow the search to the first half of the list, too low and we search the second half. We keep

⁸https://en.wikipedia.org/wiki/Hash_table

⁹https://en.wikipedia.org/wiki/Binary_search_algorithm

cutting the list in half until we either find what we were looking for, or reduce the list down to one element and discover the list does not contain the desired value.

This takes *logarithmic time*; a list of length 16 will be cut in half an average of 4 times to find a single item, a list of 32 will take 5 times, a list of 64 will take 6 times and so on. Rather than doubling the number of steps required, doubling the size of the list only adds one more step to the search, so it scales very well as the index grows. This is exactly what databases do to index tables: they create a structure that allows particular keys to be found quicker than it would take via a row-by-row scan of the whole table.

To make things even faster, Git pack indexes begin with a fixed-size 256-element list that says how many objects exist with IDs beginning `00`, `01`, `02` and so on up to `ff`. Each value in this list tells us the position in the full sorted list that each ID prefix ends at. For example, say the list for our example pack contained the following numbers:

4	7	9	12	16	17	19	21	26	28	30	41
44	50	54	58	60	62	64	67	72	78	83	86
88	92	100	104	108	111	114	116	120	123	130	136
140	144	150	157	159	160	164	167	176	181	187	189
190	193	194	196	198	204	207	211	214	218	221	224
229	230	235	238	241	246	248	249	251	257	262	266
268	269	273	276	280	282	285	287	290	294	295	296
301	304	309	312	314	317	324	329	329	332	336	339
342	348	351	354	360	362	364	370	372	377	379	382
386	389	393	398	401	405	412	415	416	416	420	423
427	430	435	438	438	438	443	447	450	455	458	459
462	466	470	475	479	481	483	488	493	495	501	506
510	512	512	515	516	519	522	524	528	532	539	543
546	548	548	554	557	561	565	568	569	573	576	578
582	585	590	595	597	599	602	603	608	615	620	625
630	632	635	644	645	648	651	655	659	662	665	667
671	675	678	680	683	685	691	698	700	708	709	713
715	715	717	721	723	728	730	734	738	739	742	746
750	754	755	759	763	764	766	777	779	784	789	794
798	804	808	813	814	814	816	823	827	832	838	840
843	848	858	858	860	868	870	873	875	884	893	897
899	901	903	909								

Say we want to find object `0148dff...`. Its prefix is `01`, and the table above tells us that these IDs begin at position 4 and end before position 7. Similarly, to find `fe47a15...`, we'd look at the prefix `fe`, the penultimate item in the table, and see that this ID must be somewhere between position 901 and 903 in the full list. Splitting the full list into 256 buckets effectively skips 8 rounds of cutting the list in half, and also gives us an easily accessible count of the objects the pack contains.

Git calls this initial structure the *fanout table*. After this it writes three further lists to the file: the full sorted list of object IDs, which are 20 bytes each, then the CRC32 of each object in corresponding order, using 4 bytes per number, and finally the offset of each object, again using 4-byte numbers. If the pack is especially large and an offset is larger than 31 bits (i.e. the pack is larger than 4 GiB), then the offset is added to a list of 8-byte numbers at the end of the index, and its position in that list is stored in the main offset list instead.

To write the @index to disk then, we perform the following steps: we generate a sorted list of all the IDs in the pack, then we write the object table: the 256-value fanout table and the full

list of IDs. After that we write the CRC32 and offset lists, and we finish the file off with the checksum of the .pack file.

```
# lib/pack/indexer.rb

def write_index
  @object_ids = @index.keys.sort

  write_object_table
  write_crc32
  write_offsets
  write_index_checksum
end
```

Writing the object table requires first writing a header that identifies this as a version-2 pack index file. `VERSION` has the value 2, and `IDX_SIGNATURE` is a special value $FF744F63_{16}$. This is chosen because version-1 packs begin with the fanout table and no header, and $FF744F63_{16}$ is an extremely unlikely value for the first fanout bucket given this is near the maximum number of objects a single index can hold.

Having written the header, we need to build the fanout table. We first build a list of the number of IDs that begin with each prefix from `00` to `ff`, and store those in counts. Then we iterate over counts to generate a running total, and write each value of this total to the file as we go through the buckets. After the fanout table, we write all the object IDs, packed to 20 bytes.

```
# lib/pack/indexer.rb

def write_object_table
  header = [IDX_SIGNATURE, VERSION].pack("N2")
  @index_file.write(header)

  counts = Array.new(256, 0)
  total = 0

  @object_ids.each { |oid| counts[oid[0..1].to_i(16)] += 1 }

  counts.each do |count|
    total += count
    @index_file.write([total].pack("N"))
  end

  @object_ids.each do |oid|
    @index_file.write([oid].pack("H40"))
  end
end
```

Writing the CRC32 values is straightforward, we simply iterate over the object IDs in order, grab the CRC32 for each one, and write it as a 4-byte big-endian number.

```
# lib/pack/indexer.rb

def write_crc32
  @object_ids.each do |oid|
    crc32 = @index[oid].last
    @index_file.write([crc32].pack("N"))
  end
```

```
end
```

Writing the offsets is a little more complicated. We still iterate over the object IDs as we did in the previous method, but now we need to check each offset we have stored. If it's less than `IDX_MAX_OFFSET`, which is 80000000_{16} , then we can write it into the first 4-byte offset list. If it's higher than this value, we push it onto the `large_offsets` array, and write its position in the `large_offsets` array, after a bitwise-or with `IDX_MAX_OFFSET` which sets the most-significant bit and identifies this offset as being stored in the extended list. After writing all the normal offsets as 4-byte integers, we write out the large offsets as 8-byte values — `Q>` is the pack formatting instruction for a big-endian 64-bit unsigned integer.

```
# lib/pack/indexer.rb

def write_offsets
    large_offsets = []

    @object_ids.each do |oid|
        offset = @index[oid].first

        unless offset < IDX_MAX_OFFSET
            large_offsets.push(offset)
            offset = IDX_MAX_OFFSET | (large_offsets.size - 1)
        end
        @index_file.write([offset].pack("N"))
    end

    large_offsets.each do |offset|
        @index_file.write([offset].pack("Q>"))
    end
end
```

Finally, we write the checksum from the `.pack` file, and rename the `.idx` file so its name matches that of the pack it represents.

```
# lib/pack/indexer.rb

def write_index_checksum
    pack_digest = @pack_file.digest
    @index_file.write(pack_digest.digest)

    filename = "pack-#{ pack_digest.hexdigest }.idx"
    @index_file.move(filename)
end
```

This index file can now be used to find out whether a pack contains a given object ID, and find its offset in the pack if so. That means we can use these files to retrieve objects, rather than reading from the individual files generated by the Unpacker.

32.2. A new database backend

In order to make the packs we're storing on disk useful, and enable this behaviour by default, we need the rest of the Jit codebase to be able to load objects from them. Since a huge amount of its functionality involves loading objects, this change should be as transparent as possible: most of the code should have no idea that the way object loading works has changed, it should just keep working.

The whole codebase uses the `Database` class to load objects. This class has quite a broad interface, but its methods can be split into three main categories. First, there are methods like `hash_object` and `short_oid`, which do not talk to the disk storage at all, they only perform in-memory operations. These methods are unaffected by a change in the underlying storage. Then there are those methods that do rely on talking to disk: `load`, `has?`, `load_raw`, `prefix_match`. Finally there are methods that *indirectly* rely on the storage but are built on top of the aforementioned methods, for example `load_tree_list` and `tree_diff`. These methods will be unaffected by a change in the storage if the `load` method continues to respond as it currently does.

This means there's a core of methods in `Database` that we can identify as relying directly on the filesystem, rather than relying on it indirectly or not at all. Git refers to objects stored in a single file per object as *loose* objects, and so I'm going to create a class called `Database::Loose` that encapsulates the management of such objects. Its methods are verbatim copies of those from `Database`, so I won't repeat their implementation here.

```
# lib/database/loose.rb

class Database
  class Loose

    def initialize(pathname)
      @pathname = pathname
    end

    def has?(oid)
    def load_info(oid)
    def load_raw(oid)
    def prefix_match(name)
    def write_object(oid, content)

    private

    def object_path(oid)
    def read_object_header(oid, read_bytes = nil)

  end
end
```

This class only deals with the translation between the Jit object model and the filesystem. It does not deal with serialising and hashing objects; `Database#store` performs those tasks and then calls `write_object` with the results. We can then delete these methods from `Database` and instead delegate them to an instance of `Database::Loose`. To an external observer, the interface and behaviour of `Database` has not changed, we have only moved some of its methods to another class.

```
# lib/database.rb

extend Forwardable
def_delegators :@backend, :has?, :load_info, :load_raw, :prefix_match

def initialize(pathname)
  @pathname = pathname
  @objects = {}
  @backend = Loose.new(pathname)
```

```
end
```

I have made one implementation change in `Database`. The `load` method calls `read_object`, which was previously built on top of `read_record_header`. As we'll see later, that implementation won't translate well to reading from packs, and so I've instead implemented `read_object` on top of `load_raw`. The `Database::Loose` class implements `load_raw` which loads unparsed objects, and then the `Database` class takes care of parsing it into a `Blob`, `Tree` or `Commit`.

```
# lib/database.rb

def read_object(oid)
  raw      = load_raw(oid)
  scanner = StringScanner.new(raw.data)

  object = TYPES[raw.type].parse(scanner)
  object.oid = oid

  object
end
```

Extracting the `Loose` class has exposed an interface where we can exploit polymorphism. If we introduce another class that implements the same interface as `Loose`, but is based on reading packs instead of loose objects, we can swap this new class in where `Loose` currently is and everything will continue working.

32.2.1. Reading the pack index

To be able to read an object from a `.pack` file, we need to find out where in the file that object begins. We get this information from the associated `.idx` file, so first we need a class for reading pack indexes. All this class needs as input is an `IO` object representing the file it's reading from. It begins by loading the fanout table — this is just 256 numbers so it's no problem reading it into memory and saving a few disk reads on each object lookup.

```
# lib/pack/index.rb

module Pack
  class Index

    HEADER_SIZE = 8
    FANOUT_SIZE = 1024

    def initialize(input)
      @input = input
      load_fanout_table
    end

    def load_fanout_table
      @input.seek(HEADER_SIZE)
      @fanout = @input.read(FANOUT_SIZE).unpack("N256")
    end

    #
  end
end
```

```
end
```

The main job of the `Pack::Index` class is to take an object ID and tell us its offset, that is the byte position in the pack file where that object begins. Using the file format we saw above in the `Indexer` class, we know this will involve a few steps. First, we consult the fanout table to see roughly where in the sorted list the ID might appear. Having found the right region, we perform a binary search within it to find the exact position of our ID. Once we know its position, we look up the corresponding offset, and if it denotes an offset above `IDX_MAX_OFFSET`, then we expand it.

The `oid_offset` method below outlines this process. It tries to find out the position in the index of the given ID, and returns nothing if a negative value is returned. Then, it reads a 32-bit integer from the offset list (`OFS_LAYER`), and returns it if it's less than `IDX_MAX_OFFSET`. If not, we select the low 31 bits of the number via a bitwise-and with `IDX_MAX_OFFSET - 1`. We `seek()` to that position in the extended offset list (`EXT_LAYER`), and read a 64-bit integer from that position.

```
# lib/pack/index.rb

OID_LAYER = 2
CRC_LAYER = 3
OFS_LAYER = 4
EXT_LAYER = 5

def oid_offset(oid)
  pos = oid_position(oid)
  return nil if pos < 0

  offset = read_int32(OFS_LAYER, pos)

  return offset if offset < IDX_MAX_OFFSET

  pos = offset & (IDX_MAX_OFFSET - 1)
  @input.seek(offset_for(EXT_LAYER, pos))
  @input.read(8).unpack("Q>").first
end
```

The `LAYER` constants help us find positions to read from in the `.idx` file. We know the file begins with an 8-byte header, followed by the 1,024-byte fanout table. If N is the number of objects, given by the last fanout entry, there then follow N 20-byte object IDs, N 4-byte CRC32 values, N 4-byte offsets, and some number of extended offsets. So to find the offset of the n^{th} item in a layer, we begin with `offset` set to $8 + 1,024$, then add N times the item size for each of the previous layers. Finally we add n times the item size of the requested layer to find the desired offset.

```
# lib/pack/index.rb

SIZES = {
  OID_LAYER => 20,
  CRC_LAYER => 4,
  OFS_LAYER => 4,
  EXT_LAYER => 8
}

def offset_for(layer, pos)
  offset = HEADER_SIZE + FANOUT_SIZE
```

```
    count = @fanout.last

    SIZES.each { |n, size| offset += size * count if n < layer }

    offset + pos * SIZES[layer]
end
```

`read_int32` is a little sugar over this that seeks to the required position and decodes a 4-byte integer from there.

```
# lib/pack/index.rb

def read_int32(layer, pos)
  @input.seek(offset_for(layer, pos))
  @input.read(4).unpack("N").first
end
```

To find the exact position of a given ID `oid_offset` calls the `oid_position` method. This begins by converting the first two digits of the ID to a number from 0 to 255, and then looks up the corresponding lower and upper bound for the object's position from the fanout table. Finally it performs a binary search between these positions until it either finds the ID or runs out of search space.

```
# lib/pack/index.rb

def oid_position(oid)
  prefix = oid[0..1].to_i(16)
  packed = [oid].pack("H40")

  low = (prefix == 0) ? 0 : @fanout[prefix - 1]
  high = @fanout[prefix] - 1

  binary_search(packed, low, high)
end
```

`binary_search` implements the method of binary searching. While `low` is not greater than `high`, at each turn we calculate `mid`, the point halfway between the current limits. We then read the object ID at that position from `OID_LAYER`, and compare it to the target ID using the `<=>` operator that `Comparable`¹⁰ values support. A result of `-1` means the current ID is less than the target, so we set `low` to one place greater than `mid`. Conversely, a result of `1` means our current value is too high, so we move `high` to exclude the upper half of the list. A result of `0` means the IDs are equal, and `mid` is the position of the ID in the index.

```
# lib/pack/index.rb

def binary_search(target, low, high)
  while low <= high
    mid = (low + high) / 2

    @input.seek(offset_for(OID_LAYER, mid))
    oid = @input.read(20)

    case oid <=> target
    when -1 then low = mid + 1
    when 0 then return mid
    end
  end
end
```

¹⁰<https://docs.ruby-lang.org/en/2.3.0/Comparable.html>

```
    when 1 then high = mid - 1
  end
end

-1 - low
end
```

If the requested ID is not found, `binary_search` returns a negative number: -1 minus the final lower bound (which, if we've exited the loop, will exceed the upper bound). This indicates the position at which the ID would need to be inserted to keep the list sorted, for example a result of -5 means the ID should be inserted at position 4. This is done not because we're actually going to insert new objects — packs and indexes are not modified once created — but because we need this to implement `prefix_match`.

`prefix_match` is a method relied upon by the `Revision` class. It lets us give a partial object ID and have it expanded to a full one, by matching against the IDs that exist in the database. Any incomplete ID will not exist in the index, but if it did, it would sort before any full IDs that begin with it as a prefix. Therefore, if `oid_position` returns a negative number, we can start inspecting the IDs from that position and collect any that begin with the given string.

```
# lib/pack/index.rb

def prefix_match(name)
  pos = oid_position(name)
  return [name] unless pos < 0

  @input.seek(offset_for(OID_LAYER, -1 - pos))
  oids = []

  loop do
    oid = @input.read(20).unpack("H40").first
    return oids unless oid.start_with?(name)
    oids.push(oid)
  end
end
```

That completes all the functionality we need from the index, and we can now use it to retrieve objects from `.pack` files.

32.2.2. Replacing the backend

We can now combine `Pack::Reader` with `Pack::Index` and retrieve objects from pairs of `.pack` and `.idx` files. Let's use this ability to create a new class, `Database::Packed`, that replicates the interface of `Database::Loose` and can be used to replace it. This class will take the path to a `.pack` file, and instantiate a `Pack::Reader` for it and a `Pack::Index` for its corresponding `.idx`. It delegates the `prefix_match` method to the `Index` object, as this does not require loading any information from the pack itself.

```
# lib/database/packed.rb

class Database
  class Packed

    extend Forwardable
    def_delegators :@index, :prefix_match
```

```
def initialize(pathname)
  @pack_file = File.open(pathname, File::RDONLY)
  @reader    = Pack::Reader.new(@pack_file)

  @index_file = File.open(pathname.sub_ext(".idx"), File::RDONLY)
  @index      = Pack::Index.new(@index_file)
end

# ...

end
end
```

Let's start with a simple method. `has?` needs to return `true` only if the database contains the given object, so for a pack this just means the index returns an offset for the ID.

```
# lib/database/packed.rb

def has?(oid)
  @index.oid_offset(oid) != nil
end
```

Implementing `load_raw` is a little more work. Just as `Pack::Unpacker` and `Pack::Indexer` had to reconstruct delta objects, so does `Database::Packed`. It begins by fetching the object's offset from the `.idx` file, returning `nil` if the object is not in this pack. If an offset is found, then we seek to the right offset in the `.pack` file and use our `Pack::Reader` to read the record there. If it's a `Record`, we can return it immediately — `Pack::Record` has the same type and data methods as `Database::Raw`, which is what everything that calls `load_raw` relies on.

If we get a `RefDelta` then we need to expand it. Unlike `Pack::Unpacker` which loads the base object from the loose object store, this backend can load the base object from the same pack. We recursively call `load_raw` and use the data from the base object to expand the delta, and return a `Record` containing the result.

```
# lib/database/packed.rb

def load_raw(oid)
  offset = @index.oid_offset(oid)
  offset ? load_raw_at(offset) : nil
end

def load_raw_at(offset)
  @pack_file.seek(offset)
  record = @reader.read_record

  case record
  when Pack::Record
    record
  when Pack::RefDelta
    base = load_raw(record.base_oid)
    expand_delta(base, record)
  end
end

def expand_delta(base, record)
  data = Pack::Expander.expand(base.data, record.delta_data)
```

```
Pack::Record.new(base.type, data)
end
```

Implementing `load_raw` means the `Database` class can now use the `Packed` class to load both raw and parsed objects, and that means we can get most of the codebase working on top of packed data stores. `Packed` does not have a `write_object` method because new objects are always written as loose files, not added to existing packs.

Earlier, we extracted the `Database::Loose` class and delegated some old `Database` methods to it. To integrate `Database::Packed` as a source of objects, we now need to replace `Loose` with another class that can load *both* loose and packed objects. It can use an instance of `Loose` to continue reading loose objects, and an instance of `Packed` for every `.pack` file in `.git/objects/pack`.

Below is the beginning of a `Database::Backends` class, which creates a `Loose` instance from the given path, and one `Packed` for each packfile. It delegates `write_object` to the `Loose` class, as that's where new objects are written. It caches all the stores it can find in `@stores` when it's first instantiated, because we don't want to be re-scanning the `.git/objects/pack` directory and re-reading every index fanout table, every single time an object is requested. However, sometimes we will need to reload this set when a new packfile is created, so we expose a `reload` method for doing this.

```
# lib/database/backends.rb

class Database
  class Backends

    extend Forwardable
    def_delegators :@loose, :write_object

    def initialize(pathname)
      @pathname = pathname
      @loose = Loose.new(pathname)

      reload
    end

    def reload
      @stores = [@loose] + packed
    end

    def pack_path
      @pathname.join("pack")
    end

    def packed
      packs = Dir.entries(pack_path).grep(/\^.pack$/)
        .map { |name| pack_path.join(name) }
        .sort_by { |path| File.mtime(path) }
        .reverse

      packs.map { |path| Packed.new(path) }
    end

    rescue Errno::ENOENT
    []
  end
end
```

```
    end

    #

  end
end
```

For the methods that read object information, we need an implementation of the `Loose` interface that can use multiple data sources, rather than just one. For example, `has?` returns `true` if any of the backing stores return `true` for this method.

```
# lib/database/backends.rb

def has?(oid)
  @stores.any? { |store| store.has?(oid) }
end
```

For the `load_info` and `load_raw` methods, we can attempt to load the requested object from each store in turn, returning the first one we find. It's possible that an object exists in multiple places, but all copies of it ought to be identical or else we'd have calculated different IDs for each copy.

```
# lib/database/backends.rb

def load_info(oid)
  @stores.reduce(nil) { |info, store| info || store.load_info(oid) }
end

def load_raw(oid)
  @stores.reduce(nil) { |raw, store| raw || store.load_raw(oid) }
end
```

For `prefix_match`, we want to find all the IDs from all the stores that match the given input. Therefore we need to call `prefix_match` on every backing store and concatenate the results. We filter out duplicate results in case an object exists in multiple places; in some cases returning multiple results from this method will make `Revision` complain about ambiguous refs, so we should avoid false failures.

```
# lib/database/backends.rb

def prefix_match(name)
  oids = @stores.reduce([]) do |list, store|
    list + store.prefix_match(name)
  end

  oids.uniq
end
```

In `Database`, we can now replace `Loose` with `Backends`, and continue delegating all the previous methods, as well as `pack_path` and `reload`, to the new backend that covers all the objects we have stored. Anything that calls `Database#load` can now recover objects from pack files, as well as loose objects, and none of the classes that call `Database` need to know about the internal `Backends`, `Loose` or `Packed` classes.

```
# lib/database.rb
```

```
extend Forwardable
def_delegators :@backend, :has?, :load_info, :load_raw,
                 :prefix_match, :pack_path, :reload

def initialize(pathname)
  @objects = {}
  @backend = Backends.new(pathname)
end
```

We've seen one place where we'll need to call `Database#reload` because a new packfile has been stored on disk. The `receive-pack` command stores a new pack sent by the client and needs to check that the client's ref updates point to objects that actually exist. Those objects will probably be in the pack the client just sent, so the `Database` needs to be reloaded to read from the newly stored pack before the validations in `receive-pack` take place. This can be handled by calling `Database#reload` at the end of `recv_packed_objects`:

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

def recv_packed_objects(unpack_limit = nil, prefix = "")
  # ...

  processor.process_pack
  repo.database.reload
end
```

There remains one part of the codebase that won't work: `Pack::Writer` uses the `load_info` method to get just the type and size of an object, and `Database::Packed` does not yet implement that method. Without it, it's impossible to push objects to another repository if we only have copies of them in pack files. To make `Packed#load_info` work, we need a supporting method on `Pack::Reader`, one that just reads the type and size of a record and doesn't read any of its data.

Like `read_record`, the `Pack::Reader#load_info` method can begin by reading the type and size of the object from its header. The size in the header is the size of the object before zlib compression, so we don't need to read the full object to get its size. If the record is a commit, tree or blob then we can immediately return a `Record`, using the `data` field to store the size. If we get a delta, then we can read the object's original size from the beginning of the delta data, using the `Expander` class. However, to get the original type requires looking up the base object, so we'll return a `RefDelta` giving the base object ID and the target size.

```
# lib/pack/reader.rb

def read_info
  type, size = read_record_header

  case type
  when COMMIT, TREE, BLOB
    Record.new(TYPE_CODES.key(type), size)

  when REF_DELTA
    delta = read_ref_delta
    size = Expander.new(delta.delta_data).target_size

    RefDelta.new(delta.base_oid, size)
  end
```

```
end
```

We can now build `Database::Packed#load_info` on top of `Pack::Reader#load_info`. Like `load_raw`, it gets the offset from the index and gets `Pack::Reader` to load the info from there. If we get a `Record`, we can convert it to a `Raw` object with the size pulled from the `data` field. If we get a `RefDelta`, then we have the original object size but we need to recurse to find the original type.

```
# lib/database/packed.rb

def load_info(oid)
  offset = @index.oid_offset(oid)
  offset ? load_info_at(offset) : nil
end

def load_info_at(offset)
  @pack_file.seek(offset)
  record = @reader.read_info

  case record
  when Pack::Record
    Raw.new(record.type, record.data)
  when Pack::RefDelta
    base = load_info(record.base_oid)
    Raw.new(base.type, record.delta_data)
  end
end
```

The `Database::Packed` class is now complete, and it can back any data access operation the codebase needs. We can fetch objects in a pack from someone else, and then push data from that pack to another repository. When Git does this, it can attempt to reuse deltas from existing packs to populate new ones, but that's strictly an optimisation. Our implementation is enough to get all the functionality working.

Now that Jit fully supports delta compression and storing packs received from a remote, we can use it to clone arbitrary repositories. For example, it can clone the Git repository in a couple of minutes:

```
$ jit init git
$ cd git
$ jit remote add origin ssh://git@github.com/git/git.git
$ time jit fetch
Receiving objects: 100% (263704/263704), 114.59 MiB, done.
Resolving deltas: 100% (196002/196002), done.
From ssh://git@github.com/git/git.git
 * [new branch] maint -> origin/maint
 * [new branch] master -> origin/master
 * [new branch] next -> origin/next
 * [new branch] pu -> origin/pu
 * [new branch] todo -> origin/todo

real    1m39.750s
```

Running `jit log --patch origin/master` immediately starts printing the recent commits with their diffs, confirming the pack has been accurately indexed and deltified records can be expanded.

Just like we did in the `Config::Stack` class¹¹, `Database::Backends` uses the composite pattern to make a collection of things look like a single object. When combining multiple objects into a composite, we need to decide between several strategies for each method: dispatching it to a specific backend as in `write_object`, querying all the backends and using one of the results as in `has?` or `load_raw`, or querying everything and combining the results as in `prefix_match`.

32.3. Offset deltas

When a pack is stored on disk and used as backing storage for the database, it can become expensive to load an object at the end of a long chain of deltas — every object it depends on must be loaded and reconstructed before it can apply the next delta in the chain. On top of this expense, locating each base object requires a call into the pack’s index, and this becomes more expensive the larger the index is. We’ve done our best to make it efficient to search, but it would be better if this cost could be eliminated entirely.

As it happens, Git has a second kind of delta record: the *ofs-delta*, short for *offset delta*. Whereas a ref-delta record gives the ID of the base object, whose location must be looked up in the index, an ofs-delta gives the offset of the base object. These records let us jump directly to the base object without consulting the index, saving a little time when rebuilding objects, and they also make the pack a little smaller as the offset numbers tend to be smaller than 20-byte object IDs.

I won’t walk through the implementation here as it’s conceptually similar to handling ref-deltas. Ofs-delta records have numeric type 6, and if a receiver advertises the `ofs-delta` capability, then the sender can use ofs-delta records in its pack. To briefly summarise the required changes:

- `Pack::Writer` and `Pack::Entry` must be changed so that if ofs-deltas are enabled, they write the type 6 and the offset of the base object, instead of 7 and the base’s ID. The offset uses a number format similar to `VarIntLE`, except its bytes are in the opposite order.
- `Pack::Reader` needs to return a new kind of value when it reads an ofs-delta record.
- `Pack::Unpacker` must remember the offset of each object ID it sees, so that if it reads an ofs-delta record it can recall the ID for that offset and load the base object from disk.
- `Pack::Indexer` needs to track deltas that depend on a given offset as well as the ID-based dependencies it already tracks. For each expanded object, it needs to expand any deltas depending on that object’s ID and offset.
- `Database::Packed` needs to handle ofs-delta records from `Pack::Reader` by using `load_raw_at` or `load_info_at` immediately, rather than consulting the index to find the base object’s offset.

The Jit repository contains a commit that implements these changes, but they are mostly boilerplate on top of the functionality we’ve already developed. The addition of this feature reduces the packed size of the commits so far from 346 KiB to 324 KiB.

¹¹Section 25.2.3, “The configuration stack”

33. Working with remote branches

Over the course of this part of the book, we have developed the `fetch` and `push` commands for transferring content between repositories. The `fetch` command creates a cache of the remote’s `refs/heads` directory in the local `refs/remotes` directory, and `push` lets us modify the `refs/heads` branch pointers in a remote repository. Although this machinery works fine, the state of any remote branches, and our local branches’ relationship to them is not presented anywhere in the user interface. To make `fetch` and `push` truly usable, we need to present this state to the user, and provide some conveniences for associating local branches with the remotes they merge with.

33.1. Remote-tracking branches

After fetching from a remote repository, there will be a set of objects in the database and refs in `.git/refs/remotes` that are completely invisible to the user. These refs are called *remote-tracking* branches, or *remote* branches for short. They cache the state of the refs in a remote repository.

The `branch` command doesn’t list remote-tracking branches, and the `log` command doesn’t display them as distinct from local branches. Without these affordances, it’s not clear to the user that they can type `origin/master` to refer to a remote branch. Let’s fix this visibility problem so users can begin to meaningfully interact with remote branches.

33.1.1. Logging remote branches

It is already possible to run `log origin/master` and see commits leading to that remote ref, and the logs will annotate the commits with the remote branches that point to them, like it does for local branches¹. This is because the decoration feature uses `Refs#reverse_refs`, which lists every ref in the `.git/refs` directory. In Section 26.1, “Storing remote references”, we also amended `Refs#short_name` so that `refs/remotes` would be stripped from the beginning of remote refs. To make it a little clearer that these are remote branches, Git displays these decorations in red rather than green, and to support this we need to be able to tell what kind of ref a `SymRef` object represents.

Let’s add two methods to `Refs::SymRef`: `branch?` returns `true` for refs in `refs/heads`, and `remote?` for those in `refs/remotes`.

```
# lib/refs.rb

SymRef = Struct.new(:refs, :path) do
  #
  # ...

  def branch?
    path.start_with?("refs/heads/")
  end

  def remote?

```

¹Section 16.1.4, “Branch decoration”

```

    path.start_with?("refs/remotes/")
  end
end

```

The log command can then use these methods in its `ref_color` method to decide how to present each ref decoration.

```

# lib/command/log.rb

def ref_color(ref)
  return [:bold, :cyan]  if ref.head?
  return [:bold, :green] if ref.branch?
  return [:bold, :red]   if ref.remote?
end

```

Another useful feature Git provides is the ability to select entire sets of refs to be displayed by the log command. Rather than specifying individual branches, you can pass `--all` to show the logs reachable from any ref, `--branches` to follow refs in `refs/heads`, and `--remotes` to log the refs in `refs/remotes`. To support this, we need to add a `list_remotes` method alongside `list_all_refs` and `list_branches` in the `Refs` class.

```

# lib/refs.rb

def list_all_refs
  [SymRef.new(self, HEAD)] + list_refs(@refs_path)
end

def list_branches
  list_refs(@heads_path)
end

def list_remotes
  list_refs(@remotes_path)
end

```

The log command can then define these options, using them to populate a set of options that's passed into the `RevList` used to find the commits.

```

# lib/command/log.rb

def define_options
  @rev_list_opts = {}
  @parser.on("--all") { @rev_list_opts[:all] = true }
  @parser.on("--branches") { @rev_list_opts[:branches] = true }
  @parser.on("--remotes") { @rev_list_opts[:remotes] = true }

  #
end

def run
  #

  @rev_list = ::RevList.new(repo, @args, @rev_list_opts)
  @rev_list.each { |commit| show_commit(commit) }

  exit 0
end

```

In RevList, these options are used to set the start points of the search by including everything the relevant Refs method returns.

```
# lib/rev_list.rb

def initialize(repo, revs, options = {})
  # ...

  include_refs(repo.refs.list_all_refs) if options[:all]
  include_refs(repo.refs.list_branches) if options[:branches]
  include_refs(repo.refs.list_remotes) if options[:remotes]

  revs.each { |rev| handle_revision(rev) }
  handle_revision(Revision::HEAD) if @queue.empty?
end
```

The log command can now easily show content from any set of local or remote branches, and it will annotate the commits clearly with remote refs, so the user can see the state of their remote repositories.

33.1.2. Listing remote branches

We can make a similar addition to the branch command. When run without arguments, it lists all the local branches from refs/heads, and optionally displays the commit each one points at. We can get it to include remote branches, or display only remote branches, using the --all and --remotes options.

```
# lib/command/branch.rb

def define_options
  @parser.on("-a", "--all") { @options[:all] = true }
  @parser.on("-r", "--remotes") { @options[:remotes] = true }

  # ...
end
```

To include the remote branches in what's listed, we need to replace the call to Refs#list_branches in Command::Branch#list_branches with a call to a new method called branch_refs, which decides which refs the command should display. Other than changing this selection decision, all the other logic remains the same: if --verbose is set, these remote refs will also have their referenced commit looked up and displayed.

```
# lib/command/branch.rb

def list_branches
  current = repo.refs.current_ref
  branches = branch_refs.sort_by(&:path)
  max_width = branches.map { |b| b.short_name.size }.max

  # ...
end

def branch_refs
  branches = repo.refs.list_branches
  remotes = repo.refs.list_remotes
```

```
    return branches + remotes if @options[:all]
    return remotes if @options[:remotes]

  branches
end
```

Again, to make the distinction between local and remote refs clearer, we can colour remote branches in red. The decision for how to present each branch's name is done in `Branch#format_ref`, and we add a clause to use red text for remote branches.

```
# lib/command/branch.rb

def format_ref(ref, current)
  if ref == current
    "* #{fmt :green, ref.short_name}"
  elsif ref.remote?
    " #{fmt :red, ref.short_name}"
  else
    "#{ref.short_name}"
  end
end
```

The `refs/remotes` branches are now more visible to the user, and this makes it clearer that they can use, say, `merge origin/master` to incorporate new commits from the remote into their local history.

33.2. Upstream branches

We've made remote branches easier to see and work with, but we can go further. Currently, there's no connection between local and remote refs — we can merge from any ref we choose, regardless of which branch we currently have checked out, and we can push our commits to any ref we like in the remote repository.

However, this is not how projects typically work, particularly when people are collaborating via a shared repository that they all push to and pull from. If you've fetched a branch from a remote repository, it's common to create your own local branch based on it, add some more commits, and then push your changes back to the same branch you based your commits on. You're typically not merging in arbitrary branches from the remote on a regular basis, or pushing to arbitrary branches. Instead, local branches tend to be *linked* with a single remote branch; we always merge new commits from that branch and push our commits back to that same branch. The local branch does not need to have the same name as the remote one, but local branches are usually associated with remotes in this way.

In Git, this association between a local branch and another branch in a remote repository is referred to as an *upstream*. The local branch's *upstream branch* is the branch in a remote repository that the local branch is usually merged with, either by fetching new commits from the remote or pushing new commits to it. An upstream branch is identified by the name of the remote it comes from, and its ref name.

To make this more concrete, suppose Alice has a repository with a local branch, `ref/heads/salt`. She also has a remote called `bob` set up to point at her colleague's copy of the project.

Bob has a branch called pepper, and Alice usually syncs her salt branch with Bob's pepper branch². When she wants to pull changes from Bob, she runs `fetch bob` (which updates her `refs/remotes/bob/pepper` ref) followed by `merge bob/pepper`. To send commits to Bob she runs `push bob salt:pepper`, short for `push bob refs/heads/salt:refs/heads/pepper`. She would like to associate these branches so that her local salt branch has `bob/pepper` as its upstream branch, and will push to it by default.

33.2.1. Setting an upstream branch

An upstream branch is represented by the name of the remote it belongs to, and the name of its ref in the remote repository. Let's say Alice has the `bob` remote set up in her configuration:

```
# .git/config

[remote "bob"]
    url = ssh://alice@example.com/bob/repo
    fetch = +refs/heads/*:refs/remotes/bob/*
```

If she runs `git branch --set-upstream-to bob/pepper` while her salt branch is checked out, Git will add the following lines to the local config:

```
# .git/config

[branch "salt"]
    remote = bob
    merge = refs/heads/pepper
```

This configuration is derived from the state of Alice's `refs` directory, and her `remote.bob` configuration. When she gives `bob/pepper` as the name of the upstream branch, this resolves to the file `refs/remotes/bob/pepper` if we search the `refs` directory for it. The variable `remote.bob.fetch` tells us that `refs/remotes/bob/pepper` is derived from `refs/heads/pepper` in the `bob` remote. And so, Git sets `branch.salt.remote` to `bob` and `branch.salt.merge` to `refs/heads/pepper`.

The `branch` command has a few options for setting upstream branches. `--set-upstream-to` sets the upstream for an existing branch, while `--track` can be used to create an upstream when a new branch is created from a remote ref. For example, `branch --track salt bob/pepper` creates the `refs/heads/salt` ref, with the initial value taken from `refs/remotes/bob/pepper`, and sets up an upstream relationship for the `salt` branch as above. The `--unset-upstream` option can be used to remove the upstream association for a branch.

Let's declare these options in the `Command::Branch` class:

```
# lib/command/branch.rb

def define_options
    #
    # ...

    @parser.on "-u <upstream>", "--set-upstream-to=<upstream>" do |upstream|
        @options[:upstream] = upstream
```

²I have used different branch names here to avoid the ambiguity of talking about the 'local' or 'remote' master branch, or Alice's or Bob's master branch. These names make it clear we're talking about two distinct refs, but it's common for branches in connected repositories to share names.

```

    end

    @parser.on("-t", "--track") { @options[:track] = true }
    @parser.on("--unset-upstream") { @options[:upstream] = :unset }
end

```

If the `:upstream` option is set, then we'll dispatch to a new method called `set_upstream_branch`.

```
# lib/command/branch.rb

def run
  if @options[:upstream]
    set_upstream_branch
  elsif @options[:delete]
    delete_branches
  elsif @args.empty?
    list_branches
  else
    create_branch
  end

  exit 0
end

```

If a branch name is specified as well as the `--set-upstream-to` argument, then the upstream is configured for that branch, otherwise the current branch is used. If `:upstream` is set to `:unset`, then we call `unset_upstream` on the `Remotes` class, otherwise we call through to a helper method `set_upstream` with the selected branch name and the value of the `:upstream` option.

```
# lib/command/branch.rb

def set_upstream_branch
  branch_name = @args.first || repo.refs.current_ref.short_name

  if @options[:upstream] == :unset
    repo.remotes.unset_upstream(branch_name)
  else
    set_upstream(branch_name, @options[:upstream])
  end
end

```

We'll also call `set_upstream` from the `create_branch` method, passing the name of the new branch, and the name of the start point as the upstream branch, if the `--track` option is used.

```
# lib/command/branch.rb

def create_branch
  branch_name = @args[0]
  start_point = @args[1]

  #

  repo.refs.create_branch(branch_name, start_oid)
  set_upstream(branch_name, start_point) if @options[:track]

  #
end

```

`set_upstream` will do the actual work of configuring the upstream. Let's say that Alice has run branch `--set-upstream-to bob/pepper`, while her salt branch is checked out. The inputs to this method are then "salt" and "bob/pepper". The given upstream name first needs to be expanded to its full ref name, `refs/remotes/bob/pepper`, which we'll do by calling `Refs#long_name`, defined below. We'll then pass the values "salt" and "`refs/remotes/bob/pepper`" through to `Remotes#set_upstream`, also defined below, and it will return the name of the remote and the upstream ref that it discovered for the given inputs. On success we tell the user the upstream was configured, and on errors we exit with the error message.

```
# lib/command/branch.rb

def set_upstream(branch_name, upstream)
  upstream      = repo.refs.long_name(upstream)
  remote, ref = repo.remotes.set_upstream(branch_name, upstream)

  base = repo.refs.short_name(ref)

  puts "Branch '#{branch_name}' set up to track remote " +
    "branch '#{base}' from '#{remote}'."

rescue Refs::InvalidBranch => error
  @stderr.puts "error: #{error.message}"
  exit 1

rescue Remotes::InvalidBranch => error
  @stderr.puts "fatal: #{error.message}"
  exit 128
end
```

`Refs#long_name` expands a short name like `bob/pepper` to a complete ref name like `refs/remotes/bob/pepper` by searching the `refs` directory. It does this by calling `path_for_name`, which returns a full system path to the ref, and then takes the relative path from the `.git` directory. If no matching file is found, an exception is raised.

```
# lib/refs.rb

def long_name(ref)
  path = path_for_name(ref)
  return path.relative_path_from(@pathname).to_s if path

  raise InvalidBranch,
    "the requested upstream branch '#{ref}' does not exist"
end
```

Now we come to the logic in the `Remotes` class for managing upstream branches. `Remotes#set_upstream` loads each `Remote` that's configured for the repository in turn, and calls `set_upstream` on each one. This method tries to match the given `upstream` with its `remote.<name>.fetch` variable, to see if it's responsible for that ref. If it is, we return the remote's name and the source ref it gave. In our example, the `bob` remote should return the source ref `refs/heads/pepper` for the target ref `refs/remotes/bob/pepper`, and so `Remotes#set_upstream` will return `["bob", "refs/heads/pepper"]`. If no remote recognises the given upstream ref, then an exception is raised.

```
# lib/remotes.rb
```

```
InvalidBranch = Class.new(StandardError)

def set_upstream(branch, upstream)
  list_remotes.each do |name|
    ref = get(name).set_upstream(branch, upstream)
    return [name, ref] if ref
  end

  raise InvalidBranch,
    "Cannot setup tracking information; " +
    "starting point '#{upstream}' is not a branch"
end
```

Remotes::Remote#set_upstream takes the branch and upstream, and tries to match the upstream ref name against its fetch variable. Recall that the bob remote is configured like so:

```
[remote "bob"]
  url = ssh://alice@example.com/bob/repo
  fetch = +refs/heads/*:refs/remotes/bob/*
```

Its fetch setting tells us that our ref refs/remotes/bob/pepper comes from the refs/heads/pepper ref in the remote repository. To find this out, we need to *invert* the refspec, that is, match its target against an input and return the corresponding source. Remote#set_upstream calls Refspec.invert to perform this task, and if that returns a result, then we set the branch.<name>.remote and branch.<name>.merge variables in the config file, and return the merge ref name.

```
# lib/remotes/remote.rb

def set_upstream(branch, upstream)
  ref_name = Refspec.invert(fetch_specs, upstream)
  return nil unless ref_name

  @config.open_for_update
  @config.set(["branch", branch, "remote"], @name)
  @config.set(["branch", branch, "merge"], ref_name)
  @config.save

  ref_name
end
```

Refspec.invert works much like Refspec.expand³, except it works in the opposite direction: rather than matching each spec's source against the given ref, it matches the target and returns the generated source value. To do this, it parses each refspec and then swaps its source and target fields before performing the match. Returning the first key in the result gives us the ref name we were looking for.

```
# lib/refspec.rb

def self.invert(specs, ref)
  specs = specs.map { |spec| Refspec.parse(spec) }

  map = specs.reduce({}) do |mappings, spec|
    spec.source, spec.target = spec.target, spec.source
```

³Section 26.3, “Refspecs”

```

    mappings.merge(spec.match_refs([ref]))
  end

  map.keys.first
end

```

For example, say the inputs to this method are `["+refs/heads/*:refs/remotes/bob/*"]` and `"refs/remotes/bob/pepper"`. Parsing the `refspec` gives us:

```
Refspec(source = "refs/heads/*", target = "refs/remotes/bob/*", forced = true)
```

Swapping the source and target fields and then matching against the ref `refs/remotes/bob/pepper` gives the mapping:

```
{ "refs/heads/pepper" => ["refs/remotes/bob/pepper", true] }
```

Returning the first key of this hash gives us the source ref name we need to configure the upstream. This value is passed back to `Remote#set_upstream`, which stores it in the configuration, and to `Command::Branch` which displays it to the user.

When `--unset-upstream` is used, we call `Remotes#unset_upstream`, which deletes the relevant config variables for the given branch.

```

# lib/remotes.rb

def unset_upstream(branch)
  @config.open_for_update
  @config.unset(["branch", branch, "remote"])
  @config.unset(["branch", branch, "merge"])
  @config.save
end

```

The final operation we need in the `Remotes` class is a means of getting the name of the remote-tracking branch for the upstream ref for a given branch. The `branch.<name>.merge` variable gives the name of the upstream ref in the *remote* repository, not the name of our remote-tracking ref for it. That is, we want a function that, given `refs/heads/salt`, returns `refs/remotes/bob/pepper`, not `refs/heads/pepper`.

`Remotes#get_upstream` will load the remote named by the `branch.<name>.remote` variable, and ask it to convert the ref name:

```

# lib/remotes.rb

def get_upstream(branch)
  @config.open
  name = @config.get(["branch", branch, "remote"])
  get(name)&.get_upstream(branch)
end

```

`Remote#get_upstream` will take the ref name, and use `Refspec` to expand it via its `remote.<name>.fetch` variable(s). The remote-tracking name is the first key in the resulting hash.

```
# lib/remotes/remote.rb
```

```
def get_upstream(branch)
    merge = @config.get(["branch", branch, "merge"])
    targets = Refspec.expand(fetch_specs, [merge])

    targets.keys.first
end
```

With these operations in place, it's now possible to fully replicate the behaviour of Git's clone command. First we use init to create a new repo, and change into its directory. Then, remote add sets up the URL we want to clone from, and fetch downloads the contents of that repository, which should produce a ref/remotes/origin/master ref in our repository. branch --track master origin/master creates the master branch, with the same value as origin/master, and links them together. As HEAD points at refs/heads/master, origin/master is now the current commit, and reset --hard populates the index and workspace with its content.

```
$ jit init <name>
$ cd <name>
$ jit remote add origin <url>
$ jit fetch
$ jit branch --track master origin/master
$ jit reset --hard
```

33.2.2. Safely deleting branches

When we first added the --delete option to the branch command⁴, we didn't have a merge command, and so we didn't implement this option fully. When used without --force, it's supposed to check whether the given branch is merged into its upstream branch, or HEAD of it has no upstream. We now have all the ingredients to implement this properly.

We can use Remotes#get_upstream to find the upstream branch name for the given branch. If it has one, then we read its value from Refs, otherwise we read HEAD. Using the FastForward module, we can check whether the upstream is a fast-forward of the given branch, meaning the branch has been merged. If it's not, then we'll raise an error instead of deleting anything.

```
# lib/command/branch.rb

include FastForward

def delete_branch(branch_name)
    check_merge_status(branch_name) unless @options[:force]

    #
end

def check_merge_status(branch_name)
    upstream = repo.remotes.get_upstream(branch_name)
    head_oid = upstream ? repo.refs.read_ref(upstream) : repo.refs.read_head
    branch_oid = repo.refs.read_ref(branch_name)

    if fast_forward_error(branch_oid, head_oid)
        stderr.puts "error: The branch '#{branch_name}' is not fully merged."
        exit 1
    end

```

⁴Section 15.5.3, “Deleting branches”

```
end
```

33.2.3. Upstream branch divergence

If we've linked a local branch with an upstream remote branch, we can report on the state of these branches relative to each other, that is, the relative position of the commits referenced by these branches in the commit history graph. If salt is an ancestor of bob/pepper, then our local branch is behind its upstream and we can fast-forward it using `merge`. If bob/pepper is an ancestor of salt, then our branch is ahead of its upstream branch and can be pushed without using `--force`. If neither is an ancestor of the other, then our branch and its upstream have *diverged*: they contain concurrent edits that must be truly merged before we can push to the upstream again.

To provide this information, we'll add a method to `Repository` that takes a `Refs::SymRef` and returns an object that tells us the name of its upstream, and the relative state of the two branches.

```
# lib/repository.rb

def divergence(ref)
  Divergence.new(self, ref)
end
```

The `Divergence` class takes a `Repository` and a `SymRef`, and gets the remote-tracking upstream name for it. If an upstream branch exists, then we want to calculate how far ahead the local branch is, that is the size of the range `bob/pepper..salt`, and how far behind it is, that is the size of the range `salt..bob/pepper`. We'll do this by reading the object ID of each ref, and then running `Merge::CommonAncestors` on these inputs. When it's finished, we can ask it to count the number of commits it found that were reachable from only one of the inputs.

```
# lib/repository/divergence.rb

class Repository
  class Divergence

    attr_reader :upstream, :ahead, :behind

    def initialize(repo, ref)
      @upstream = repo.remotes.get_upstream(ref.short_name)
      return unless @upstream

      left   = ref.read_oid
      right  = repo.refs.read_ref(@upstream)
      common = Merge::CommonAncestors.new(repo.database, left, [right])

      common.find
      @ahead, @behind = common.counts
    end

  end
end
```

`Merge::CommonAncestors#counts` takes the state generated by the common ancestor search⁵, and counts how many entries therein are flagged as reachable from only a single parent.

⁵Section 17.2, "Finding the best common ancestor"

```
# lib/merge/common_ancestors.rb

def counts
  ones, twos = 0, 0

  @flags.each do |oid, flags|
    next unless flags.size == 1
    ones += 1 if flags.include?(:parent1)
    twos += 1 if flags.include?(:parent2)
  end

  [ones, twos]
end
```

Given how we're calling it, `ones` tells us how far ahead our branch is, and `twos` how far behind we are. In Git, this is actually done by calling `rev-list` with the *symmetric range* salt...`bob/pepper` (note the three dots), which gives all the commits reachable from *either* ref, but excludes their common ancestors. Our technique is less general but more direct and does not require multiple scans of the commit graph.

There are a couple of places this information is usually displayed by Git. First, calling `branch` with `-v` or `-vv` displays the divergence of a branch from its upstream, with the name of the upstream if `-vv` is used. We'll detect this by using a counter on our `--verbose` option.

```
# lib/command/branch.rb

def define_options
  @parser.on("-a", "--all") { @options[:all] = true }
  @parser.on("-r", "--remotes") { @options[:remotes] = true }

  @options[:verbose] = 0
  @parser.on("-v", "--verbose") { @options[:verbose] += 1 }

  #
end
```

In `extended_branch_info`, we'll return nothing unless at least one `--verbose` flag is used, and call `upstream_info` to generate the divergence information.

```
# lib/command/branch.rb

def extended_branch_info(ref, max_width)
  return "" unless @options[:verbose] > 0

  commit = repo.database.load(ref.read_oid)
  short = repo.database.short_oid(commit.oid)
  space = " " * (max_width - ref.short_name.size)
  upstream = upstream_info(ref)

  "#{ space }#{ short }#{ upstream }#{ commit.title_line }"
end
```

`upstream_info` will call `Repository#divergence` to gather the required information for the given branch, returning nothing if no upstream is configured. Otherwise, we construct a string from the upstream name (if `-vv` is used), and the commit counts from the `Divergence` class, and return it.

```
# lib/command/branch.rb

def upstream_info(ref)
  divergence = repo.divergence(ref)
  return "" unless divergence.upstream

  ahead = divergence.ahead
  behind = divergence.behind
  info = []

  if @options[:verbose] > 1
    info.push(fmt(:blue, repo.refs.short_name(divergence.upstream)))
  end
  info.push("ahead #{ ahead }") if ahead > 0
  info.push("behind #{ behind }") if behind > 0

  info.empty? ? "" : "[#{ info.join(", ") }]"
end
```

The other place this information is displayed is in the status command, when using the long format. After the name of the current branch is displayed, it displays the state of the branch's upstream.

```
# lib/command/status.rb

def print_long_format
  print_branch_status
  print_upstream_status
  print_pending_commit_status

  #
end
```

`print_upstream_status` retrieves the same information as `Branch#upstream_info`, and displays one of a choice of messages depending on whether the local branch is ahead, behind, or divergent relative to its upstream.

```
# lib/command/status.rb

def print_upstream_status
  divergence = repo.divergence(repo.refs.current_ref)
  return unless divergence.upstream

  base = repo.refs.short_name(divergence.upstream)
  ahead = divergence.ahead
  behind = divergence.behind

  if ahead == 0 and behind == 0
    puts "Your branch is up to date with '#{ base }'."
  elsif behind == 0
    puts "Your branch is ahead of '#{ base }' by #{ commits ahead }."
  elsif ahead == 0
    puts "Your branch is behind '#{ base }' by #{ commits behind }, " +
      "and can be fast-forwarded."
  else
    puts <<~MSG
      Your branch and '#{ base }' have diverged,
      and have #{ ahead } and #{ behind } different commits each, respectively.
    MSG
  end
end
```

```
    MSG
  end

  puts ""
end

def commits(n)
  n == 1 ? "1 commit" : "#{n} commits"
end
```

Displaying this information helps the user understand what will happen if they run the push command, or try to merge the upstream into their local branch.

33.2.4. The @{upstream} revision

To make it easier to refer to a branch's upstream counterpart without needing to remember its name, Git's revisions notation⁶ provides the syntax <branch>@{upstream}. The branch name is optional, and defaults to HEAD if omitted. The word upstream is case-insensitive, and can be abbreviated to merely {@u}. Using this expression in a revision passed to branch, checkout, log, merge and friends gives the commit ID that the remote-tracking upstream of the given branch points at. In our example then salt@{@u} returns the commit ID given by refs/remotes/bob/pepper that is currently cached on disk, without re-fetching from the remote.

To parse this syntax, we need a regular expression that matches a possibly-empty string followed by {@{} with either u or upstream within the braces. We'll also add the mapping "" => HEAD to REF_ALIASES so that the empty string is replaced with HEAD when the branch name is parsed.

```
# lib/revision.rb

UPSTREAM = /^(.*)@{\u(pstream)?}\$/i

REF_ALIASES = {
  "@" => HEAD,
  ""  => HEAD
}

def self.parse(revision)
  if match = PARENT.match(revision)
    #
  elsif match = UPSTREAM.match(revision)
    rev = Revision.parse(match[1])
    rev ? Upstream.new(rev) : nil
  else
    #
  end
end
```

Upstream has a single child node, which should be a Ref — a branch name, rather than an object ID, or parent or ancestor expression. It looks up the remote-tracking upstream name via Remotes#get_upstream, and then reads the ID from that ref.

```
# lib/revision.rb
```

⁶Section 13.3, “Setting the start point”

```
Upstream = Struct.new(:rev) do
  def resolve(context)
    name = context.upstream(rev.name)
    context.read_ref(name)
  end
end

def upstream(branch)
  branch = @repo.refs.current_ref.short_name if branch == HEAD
  @repo.remotes.get_upstream(branch)
end
```

This expression makes it easy to merge in new commits from the upstream. After running `fetch`, `merge @{u}` will merge the latest commits from the upstream branch, performing a fast-forward if possible and doing nothing if there are no new commits. If you'd rather rebase your local changes, you can do this by running `reset --hard @{u}` followed by `cherry-pick ..ORIG_HEAD`.

33.2.5. Fetching and pushing upstream

The final function of upstream branches is to make it more convenient to push and pull content between repositories. If an upstream is configured for the current branch, then running `fetch` without arguments will use the `branch.<name>.remote` as the remote to pull from, rather than defaulting to `origin`. This can be implemented by looking up the value of this variable at the beginning of `Command::Fetch#configure`.

```
# lib/command/fetch.rb

def configure
  current_branch = repo.refs.current_ref.short_name
  branch_remote = repo.config.get(["branch", current_branch, "remote"])

  name = @args.fetch(0, branch_remote || Remotes::DEFAULT_REMOTE)
  remote = repo.remotes.get(name)

  #
end
```

It also makes it possible to run `push` without arguments. What happens in this event is determined by the value of `push.default`⁷, which defaults to `simple`. What we'll implement here is the `upstream` setting, which means that the current branch is pushed to its upstream, if one is configured.

To achieve this, we load the `branch.<name>.remote` and `branch.<name>.merge` variables from the config. In our example, these will be `bob` and `refs/heads/pepper` respectively. The `remote` variable is used in place of `origin` if no remote is specified on the command-line. The `merge` variable is used in place of the configured push specs for the remote, if no `refsspecs` are given in the command. If no `refsspecs` are given, then we build an unforced `Refspec` with the current branch as its source and the `merge` variable as its target.

```
# lib/command/push.rb
```

⁷<https://git-scm.com/docs/git-config#git-config-pushdefault>

```
def configure
  current_branch = repo.refs.current_ref.short_name
  branch_remote  = repo.config.get(["branch", current_branch, "remote"])
  branch_merge   = repo.config.get(["branch", current_branch, "merge"])

  name    = @args.fetch(0, branch_remote || Remotes::DEFAULT_REMOTE)
  remote = repo.remotes.get(name)

  @push_url    = remote&.push_url || @args[0]
  @fetch_specs = remote&.fetch_specs || []
  @receiver    = @options[:receiver] || remote&.receiver || RECEIVE_PACK

  if @args.size > 1
    @push_specs = @args.drop(1)
  elsif branch_merge
    spec = Remotes::Refspec.new(current_branch, branch_merge, false)
    @push_specs = [spec.to_s]
  else
    @push_specs = remote&.push_specs
  end
end
```

The effect of these decisions is that push with no arguments pushes the current branch to its upstream in the remote repo. This push will not be forced, unless the --force option is used.

After all these changes, it is no longer necessary to remember the name of the upstream branch at all, once it has been configured. fetch and push will use the configured upstream if run without arguments, and merge and cherry-pick can use the @{u} revision to incorporate commits from the upstream branch. In Git, the fetch command places some metadata about what was fetched in .git/FETCH_HEAD, and the merge and pull commands use this to decide what to merge, if no arguments were given.

34. ...and everything else

Over the course of this book, we've developed a perfectly capable Git implementation. It is by no means complete, but it contains all the essential building blocks to be useful for version control, and to interoperate with the canonical Git software. It can store commits, create and merge branches, and distribute content over the network. We could fill several more books attempting to replicate all of Git's functionality. But the tools we have developed cover most of the core concepts in Git's internals, and it should be straightforward to implement further features on top of them, should you want to explore further.

For example, some Git commands are sugar over commands we already have. `clone` can be implemented using `init`, `remote`, `fetch`, `branch` and `reset`. `pull` is the same as running `fetch` followed by either `merge` or `rebase`, and `rebase` itself can be simulated using `reset` and `cherry-pick`.

Some commands have options that translate into lower-level workflows. For example, `commit --all` is equivalent to calling `add` or `rm` with all modified or deleted files before running `commit`. `checkout -b` is the same as creating a new branch and then immediately checking it out. If `checkout` is called with a remote-tracking branch like `origin/master`, then it creates a new local branch with the remote as its upstream, as in `branch --track master origin/master`, and then checks this new branch out. If `checkout` is given a path, it behaves like `reset`, only instead of setting the index equal to `HEAD` for the given path, it affects the index *and* the workspace file. In fact `checkout` and `reset` have a lot of overlapping functionality related to equating entries in the workspace, index and `HEAD`, and this can make them confusing to learn.

There are some commands and options that we could build on top of our existing internal tools. For example, `add --patch` lets you add only some of the changes in a file to the index, and this can be implemented by using `Diff.diff_hunks` to find the changes between the index and the workspace, asking the user to approve each one, and applying the selected edits to a copy of the workspace file before storing it as a blob. The `blame` command¹ annotates each line of a file with the last commit that changed it, and this can be built by using `RevList` to find all the commits that change a file, then using `Diff` on consecutive versions to find out which lines each commit introduced. The `bisect` tool² is used to find which commit introduced a bug — if you know commit *G* was good and commit *B* is bad, then `RevList` can calculate the range *G..B* and pick a point halfway between them to test. If the selected commit contains the bug, then it replaces *B* in the range, and if it does not then it's added to the set of *G* commits that are excluded from the search. This process just requires storing the lists of good and bad commits on disk while pausing to let the user run tests.

Many commands have options that tweak the core behaviour that we've implemented here. `diff` (and commands that use `diff` internally, such as `merge`) have options for determining whether lines are equal, for example by ignoring changes to whitespace. `diff` can also use one of several algorithms — we've discussed the Myers algorithm here, but there are others with different characteristics, and they can be used to improve merge results if the default algorithm doesn't produce a good diff. The `log` command has numerous options for filtering

¹<https://git-scm.com/docs/git-blame>

²<https://git-scm.com/docs/git-bisect>

and displaying commits, including a `--graph` option to show the commit graph, and there are many more features in the revision notation that we've not looked at here.

A few commands require infrastructure we've not looked at. `tag` creates *tag* objects, which are essentially objects that point to commits and give them a memorable name — often used to record released versions of the software. `reflog` lets you search the history of the commits that files in `refs` have pointed at, and to support that the `Refs` class needs to be updated so that whenever it changes the content of a `.git/refs` file, it appends the change to a corresponding file in `.git/logs`. This helps you find commits that have become unreachable, and it also helps the `rebase` command find the point where a branch started, if the commits it's based on have been discarded by cherry-picking³.

Finally, there are some Git features that, like the compression functionality we have implemented, are optimisations rather than new user-facing behaviour. For example, the index supports an extension that caches the IDs of trees when `commit` is run, so that it doesn't need to recompute trees that have not changed on each commit. The pack protocol supports a capability called *thin packs*, which means a pack can include deltas that are based on objects that aren't in the pack, but which the sender knows the receiver has. This allows a pack to be compressed even further, although the receiver has to append any delta bases that are not included in order to make the pack self-contained before storing it.

All in all, there is plenty of scope to expand Jit and learn more about how Git works on your own. Read the documentation for a command you use every day and see if you can implement some of its options. Think of a command you find confusing and try to improve its user interface. With the copy of Jit included with this book, you can start committing right away and take it in whatever direction you want.

³<https://git-scm.com/docs/git-rebase#git-rebase---fork-point>

Part IV. Appendices

Appendix A. Programming in Ruby

Jit is written in Ruby¹, a dynamically typed object-oriented programming language. Ruby is widely used in web applications, as well as automation tools like Chef² and Homebrew³. Its design is influenced by Perl, Smalltalk and Lisp, and is fairly similar to other commonly used object-oriented languages such as Python and JavaScript. My intent is that you don't need to know Ruby to read this book, but having some familiarity with it or a similar language will certainly help.

This appendix covers everything you need to know about Ruby to understand the Jit codebase. It does not attempt to exhaustively document the language, and Jit has been designed to make effective use of the language while keeping this reference as brief as possible. Full documentation is available online⁴, and in numerous books, for example *The Ruby Programming Language*⁵.

If you have Ruby installed, you can also find documentation using the `ri` command-line tool. For example, to see an overview of the `Array` class, type `ri "Array"` into your terminal. To see information about a particular method, `Array#map` for example, type `ri "Array#map"`. If you don't know which class a method belongs to, you can just enter its name, as in `ri "map"`, to see all the implementations of that method in the Ruby standard library.

If you want to try out code to see what it does, Ruby comes with an interactive environment called IRB⁶. This lets you enter expressions and have them immediately evaluated. Type `irb` into your terminal and IRB will present you with a prompt where you can enter Ruby code. Code entered here behaves similarly to code placed in a file and executed.

A.1. Installation

Jit requires Ruby 2.3.0 or later, and it targets Unix-like environments. To make sure you can run Jit, type the following into your terminal:

```
$ ruby --version
```

This should print a version like `ruby 2.3.7p456` followed by some other build information. The version should be 2.3.0 or later.

If you are using macOS High Sierra (10.13) or later, then Ruby 2.3 or a later version should be pre-installed. If not, you can get it from Homebrew by typing `brew install ruby` to get the latest version.

If you are on a Linux distribution such as Ubuntu, your package manager will likely carry a `ruby` package. Ubuntu 16.04 and later releases have a `ruby` package that is at least version 2.3.0.

¹<https://www.ruby-lang.org/>

²<https://www.chef.io/chef/>

³<https://brew.sh/>

⁴<https://docs.ruby-lang.org/en/2.3.0/>

⁵<http://shop.oreilly.com/product/9780596516178.do>

⁶<https://docs.ruby-lang.org/en/2.3.0/IRB.html>

As a tool designed to teach Unix programming concepts, Jit does not support Windows, but if you would like to tinker with the code you can use RubyInstaller⁷ to set up a Ruby environment on your system.

Ruby has a couple of tools for installing third-party libraries: RubyGems⁸ and Bundler⁹. Ruby is usually distributed with a few commonly used libraries, and Jit uses a couple of them to run its tests. If you run the following command, you should see `rake` and `minitest` listed in the output.

```
$ gem list
```

At runtime, Jit only uses the Ruby standard library, not any third-party libraries. So don't worry too much if you don't have `gem` or these packages installed.

A.2. Core language

Ruby does not have a compiler or any other build tooling. To run Ruby code, you just save it to a file with a `.rb` extension, and execute that file. Here is 'hello world' in Ruby:

```
# hello.rb  
puts "hello world"
```

Running this file using `ruby` prints the expected message:

```
$ ruby hello.rb  
hello world
```

As in other dynamic imperative languages, a Ruby program is a sequence of statements that the interpreter executes from top to bottom. Statements are terminated by line breaks; semicolons are not required but can be used if desired. In the above program, "hello world" is a literal denoting a `String` value, and `puts` is a call to a method that prints its argument. A great deal of functionality in Ruby is accomplished via method calls, but the language does have a few built-in keywords. We'll look at these, and then examine how to define our own objects.

A.2.1. Control flow

Conditional blocks are defined using the `if` and `unless` keywords. This keyword is followed by an expression, which does not need surrounding parentheses, then a sequence of statements, then the keyword `end`. The statements are executed if the condition is true, or in the case of `unless`, if the condition is false.

```
if 1 < 2  
  puts "correct!"  
  puts "one is less than two"  
end  
  
unless 3 < 4  
  puts "something seems to be wrong"
```

⁷<https://rubyinstaller.org/>

⁸<https://rubygems.org/>

⁹<https://bundler.io/>

```
end
```

The keyword `else` can be used to define statements that execute if the condition is not true.

```
if 1 > 2
  puts "that's not how maths works"
else
  puts "one is not greater than two"
end
```

If you have multiple conditions to check, use `elsif` to introduce all but the first condition. The conditions will be tried in order, and the first one that's true will have its statements executed. No further conditions will be tested after one has succeeded.

```
if 1 > 2
  puts "this should not execute"
elsif 3 > 4
  puts "neither should this"
else
  puts "this _will_ be printed"
end
```

A condition can be placed after a statement to make that single statement conditional, for example:

```
puts "your computer is broken" unless 1 + 1 == 2
```

This is equivalent to:

```
unless 1 + 1 == 2
  puts "your computer is broken"
end
```

Ruby supports the *ternary* operator `? :` to select one of two expressions based on a condition. The general form is `x ? y : z` where `x`, `y` and `z` are expressions. The value of the whole expression is `y` if `x` is true, and `z` otherwise. The expression that is not selected is not executed. The condition will sometimes be enclosed in parentheses for clarity, but this is not essential.

```
message = (1 + 1 == 2) ? "working" : "broken"
```

A more complicated form of conditional execution is provided by the `case` keyword. This is followed by an expression, and it compares the value of this expression with a sequence of values. For each `when` clause, the expression is compared to the value following the `when`, and if they match then the statements up to the next `when` or `else` are executed and no further clauses are tried. If no `when` clause matches, then the optional `else` block is executed. Unlike in the `switch/case` construct in C-like languages, `when` clauses do not fall-through into the next block.

```
case x
when 1
  puts "one"
when 2
  puts "two"
else
  puts "many"
end
```

If the block for a clause is a single statement, it can be placed on the same line as the `when`, prefixed with the keyword `then`.

```
case x
when 1 then puts "one"
when 2 then puts "two"
else      puts "many"
end
```

A `when` can be followed by multiple values, and it will match if the expression matches any of these values.

```
case x
when 1, 3 then puts "odd"
when 2, 4 then puts "even"
end
```

`case` does not only compare values for exact equality (the `==` operator). It uses a distinct operator called *case equality*, written `==`. This roughly means that a value matches a pattern, or is a member of a group. For example, if a `when` is followed by the name of a class, then it matches if the `case` expression evaluates to an instance of that class.

```
case x
when Integer then puts "it's a number"
when String   then puts "it's a string"
end
```

Other commonly used `when` values are regular expressions and numeric ranges.

The `while` and `until` keywords are used for iteration. `while` repeatedly checks its condition and executes the following statements if it's true, and `until` executes the body if the condition is false.

```
x = 0

while x < 10
  x = x + 1
end
```

Like `if` and `unless`, `while` and `until` can be placed after a single statement in order to execute that statement in a loop.

```
x = 0
x = x + 1 until x == 10
```

There is also the unconditional loop construct, `loop`. This takes a block (we'll introduce blocks properly later), and executes it until a keyword like `break`, `return` or `raise` is used to escape it.

```
loop do
  puts "are we there yet?"
end
```

Inside a loop, the `next` keyword skips to the next iteration, and `break` escapes the loop entirely. Ruby does not have loop labels like you might have seen in C-like languages, so `next` and `break` only affect the immediately enclosing loop.

A.2.2. Error handling

Ruby uses *exceptions*¹⁰ to handle errors. The `raise` keyword takes the name of an error type, and a message, and throws the exception. This aborts the current method, and the method that called it, and so on up the stack until a `rescue` clause appears.

```
raise NameError, "that's not my name"
```

To catch potential exceptions, use the `begin` and `rescue` keywords. This construct works like the `try/catch` statement you may have seen in other languages. It runs the statements after `begin`, until one of them produces an exception. If this happens, then the `rescue` block is executed.

```
begin
  set_name("Jim")
rescue
  puts "something bad happened"
end
```

You can optionally follow `rescue` with an arrow and a variable name, and the exception object will become bound to that name.

```
begin
  set_name("Jim")
rescue => error
  puts error.message
end
```

`rescue` can also be followed by the name of the error type, and the exception will be matched against each `rescue` clause to decide which block to execute.

```
begin
  set_name("Jim")
rescue NameError => error
  puts "bad name: #{error.message}"
rescue TypeError => error
  puts "bad type: #{error.message}"
end
```

Inside a method body, the `begin` and `end` can be omitted. If a method body contains a `rescue` clause, it behaves as though the entire method body was wrapped in `begin...end`.

```
def foo
  do_something(x)
rescue => e
  handle_error(e)
end
```

Within a `rescue` clause, the `retry` keyword causes the `begin` segment to be re-executed from the beginning. As well as `rescue` clauses, these blocks can include `ensure` clauses, which define code that's executed whether or not an error is thrown. This is good for cleaning up resources.

```
begin
  file = File.open("config.json")
  config = JSON.parse(file.read)
```

¹⁰https://en.wikipedia.org/wiki/Exception_handling

```
rescue
  puts "invalid JSON"
ensure
  file.close
end
```

Ruby has many built-in error types, and its libraries define their own custom errors. The `Errno` module¹¹ defines various errors that represent the low-level errors in the underlying C system code. For example, if you try to open a file that does not exist, an `Errno::ENOENT` error is raised.

A.2.3. Objects, classes, and methods

To build meaningful programs, we need some units of abstraction. In Ruby, those units are objects and methods. All values are objects, and all interaction with values occurs via calls to an object's methods. Objects are created from classes, which are defined using the `class` keyword, and those classes define the methods for their instances using `def`.

Here is a class with one method that takes no arguments:

```
class Robot
  def talk
    "beep beep"
  end
end
```

Method names can end with a `?`, usually used for methods that return a boolean, or a `!`, which by convention is used when the method mutates the receiving object and a method without a `!` exists that does not mutate. For example, if `bot = "robot"`, then `bot.gsub("o", "i")` returns a new string `"ribit"`, while `bot.gsub!("o", "i")` changes the string `bot` itself so it now contains `"ribit"`.

Objects are created by instantiating a class, by calling its `new` method. Calling a method on an object is done by following the object with a dot `(.)`, and the name of the method. The return value of the method is the value of the last expression its body executes, unless the method contains an explicit `return` statement.

```
bot = Robot.new
bot.talk # returns "beep beep"
```

The dot operator raises an error if the object does not implement, or ‘respond to’, the requested method. The *safe navigation* operator `&.` works like `..`, but if the value on its left is `nil`, then it silently returns `nil` rather than raising an error because `nil` doesn't have the requested method.

```
bot = nil

bot.talk
# NoMethodError (undefined method `name' for nil:NilClass)

bot.&.talk # returns nil
```

This is useful for invoking methods on optional values that might not be assigned; it's equivalent to the expression `bot == nil ? nil : bot.talk`.

¹¹<https://docs.ruby-lang.org/en/2.3.0/Errno.html>

If a method takes arguments, those are specified after its name in the definition.

```
class Robot
  def crunch_numbers(x, y)
    x + y
  end
end
```

To pass arguments in a method call, we also follow its name with the arguments.

```
bot.crunch_numbers(1, 2) # returns 3
```

The parentheses around the method arguments are optional, so the above can also be written:

```
bot.crunch_numbers 1, 2
```

For clarity, most method calls taking arguments do use parentheses. Method calls taking no arguments usually do not; `bot.talk` means the same thing as `bot.talk()`, and `def talk` is the same as `def talk()`. Ruby does not have public object properties or attributes as some other object-based languages do; you can only interact with an object by calling its methods.

Arguments are always passed to methods by reference, without being copied. So if you pass a value to a method and the method mutates it, that change will be visible to the caller.

```
class ListChanger
  def change_list(list)
    list.push(0)
  end
end

changer = ListChanger.new
array   = [1, 2, 3]

changer.change_list(array)

# array now contains [1, 2, 3, 0]
```

Methods can have optional parameters. In a method definition, parameters followed by an assignment are optional; the assignment specifies their default value.

```
class ListChanger
  def change_list(list, value = 0)
    list.push(value)
  end
end

changer = ListChanger.new
array   = [1, 2, 3]

changer.change_list(array, 4)
changer.change_list(array)

# array now contains [1, 2, 3, 4, 0]
```

The expression for the default value is evaluated every time the method is called, not when it is defined. So the method definition below creates a new array every time it is invoked.

```
def change_list(list = [], value = 0)
```

It is an error to call a method with too few or too many arguments. A method must receive at least as many arguments as it has non-optional parameters, and must not receive more arguments than the total number of parameters it has. A method call with the wrong number of arguments will raise an exception.

Arguments can also be made optional via the `*` or *splat* operator. If the last parameter to a method is prefixed with `*`, then any arguments surplus to the required parameters are supplied as a list.

```
class ListChanger
  def change_list(list, *values)
    values.each { |v| list.push(v) }
  end
end

changer = ListChanger.new
array = [1, 2, 3]

changer.change_list(array, 4, 5, 6)

# array now contains [1, 2, 3, 4, 5, 6]
```

An array can also be passed to a method expecting multiple arguments by prefixing it with `*`.

```
args = [list, 7]
changer.change_list(*args)
```

This can also be used to flatten arrays, for example `[1, 2, *[3, 4]]` gives `[1, 2, 3, 4]`.

To do anything interesting, objects need internal state. In Ruby this state is held in *instance variables*, which are prefixed with the symbol `@`. These do not need to be declared in advance, and can be assigned and accessed in any of an object's methods. It's common to assign them in the `initialize` method, which is implicitly called on new objects created by calling `new`.

```
class Robot
  def initialize(name)
    @name = name
  end

  def say_hello
    puts "hello, I'm #{ @name }"
  end
end

bot = Robot.new("Marvin")
bot.say_hello # prints "hello, I'm Marvin"
```

Instance variables are scoped to the object; each instance of a class carries its own values for its instance variables, and those variables can be accessed from any of the object's methods. A reference to an instance variable that is not defined evaluates to `nil` rather than raising an error, though Ruby may print a warning about it.

Local variables — those not beginning with an `@` — are scoped to a method body between `def` and `end`. They don't persist after the method returns. It is an error to reference a local variable that does not exist. A variable may be defined, but have the value `nil`; this is distinct from the variable not existing at all.

Global variables are prefixed with a \$ sign, and are visible throughout the program. Some operations, for example those used in matching regular expressions, implicitly create global variables containing the matches they found, but we will not be using those.

Sometimes we want to be able to read an instance variable from outside an object. We could write a method for doing this:

```
class Robot
  def initialize(name)
    @name = name
  end

  def name
    @name
  end
end

bot = Robot.new("Eve")
bot.name # returns "Eve"
```

However, Ruby provides a shorthand for doing this, called `attr_reader`. The following class is equivalent to the above:

```
class Robot
  attr_reader :name

  def initialize(name)
    @name = name
  end
end
```

`attr_reader` takes a list of names and generates a method for each one that exposes the instance variable of the same name. Similarly, `attr_writer` generates a method for assigning an instance variable from outside an object. The expression `:name` is a *symbol*, which we'll discuss in more detail below.

```
class Robot
  attr_writer :name

  bot = Robot.new
  bot.name = "Eve"

  # bot now has an instance variable @name with the value "Eve"
```

It looks like `bot.name = "Eve"` is an assignment to a property, but it's really a method call; it's invoked a method called `name=`. We could write this explicitly without using `attr_writer` as follows.

```
class Robot
  def name=(name)
    @name = name
  end
end
```

A reader and writer can be defined in a single expression by using `attr_accessor`.

Some data types allow their members to be accessed via bracket notation, as in `array[0]`, or assigned via `array[0] = x`. These are also method calls, invoking the methods `[]` and `[]=`.

```
class List
  # called when `list[n]` is used
  def [](n)
    #
  end

  # called when `list[n] = x` is used
  def []=(n, x)
    #
  end
end
```

An object's instance variables are completely private — they cannot be accessed directly from outside the object. An object can also have private methods, that is methods that can only be invoked by other methods in the same object, not called from the outside. The keyword `private` causes all the methods after it to be made private.

```
class Robot
  def initialize(name)
    set_name(name)
  end

  private

  def set_name(name)
    @name = name
  end
end
```

When calling another method in the same class, the dot operator is not needed. `set_name(name)` might look like a call to a global function, but it's not. When a method call appears with no object preceding it, there's an implicit `self`. that's been omitted. Knowing this, we can see that the hello-world program at the beginning of this guide is really doing this:

```
self.puts("hello world")
```

`puts` is not a keyword, or a global function. It's a method defined on all objects. Calls to `self` made at the top level, outside of any method definition, are dispatched to the `main` object, a special object that represents the global scope.

Since `self.` is usually unnecessary, you won't often see it used. It does show up in conjunction with `attr_writer` methods: `self.x = y` means a method call, `self.x=(y)`, whereas `x = y` is an assignment to the local variable `x`.

Methods defined using `def` as above become available on instances of the class, that is objects created by calling `new` on the class. But classes can also have *class methods*, which are not called on instances but on the class itself. This is sometimes used to define *factory methods*¹² that make some decision before returning an instance of the class. Or, they might not create any objects at all but provide some utility behaviour.

```
class Robot
```

¹²[https://en.wikipedia.org/wiki/Factory_\(object-oriented_programming\)](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming))

```

def self.create(name, is_loud)
  new(is_loud ? name.upcase : name.downcase)
end

# ...
end

Robot.create("Marvin", true) # Robot(@name = "MARVIN")
Robot.create("Eve", false)   # Robot(@name = "eve")

```

The call to `new` in `Robot.create` has an implied `self.`, so it's a call to `Robot.new`. In a class, `self` refers to the class itself, while in normal methods, `self` refers to the object that received the method call.

When referring to methods in documentation, it is conventional to denote instance methods with the `#` sign, as in `Robot#name`. This format is recognised by `ri`. Class methods are denoted with a dot, as in `Robot.create`.

A class can be defined as a subclass of another, via the `<` operator.

```

class Transformer < Robot
# ...
end

```

The subclass inherits all the methods that its parent class has, and it can override the parent's methods and add new ones of its own. Any instance variables set by the parent class's methods are visible inside the subclass. If the subclass overrides an inherited method, it can use `super` to invoke its parent's implementation of it.

```

class Transformer < Robot
attr_reader :name

private
def set_name(name)
  super(name.reverse)
end
end

bot = Transformer.new("Blaster")
bot.name # returns "retsalB"

```

If `super` is used without any arguments following it, the arguments to the current method are implicitly passed. So above, `super` would be equivalent to `super(name)`.

Finally, classes may be nested as a way of providing namespacing. Nesting one class inside another does not affect its behaviour, it only changes its name. For example:

```

class MyApp
  class Robot
    # ...
  end
end

```

The `Robot` class does not inherit any behaviour from `MyApp`, or gain visibility of its instance variables, or anything like this. This nesting is nothing like inner classes in Java. All it means is that, outside of the `MyApp` class, code must refer to the `Robot` class via the name `MyApp::Robot`.

This provides a means of keeping names out of the global scope to prevent clashes, if a lot of program components use classes with similar names, or a class is only used internally by another and is not visible to the rest of the system.

A.2.4. Blocks

Methods are not the only type of code encapsulation in Ruby. Code can also be bundled up into *blocks*, which let you pass executable chunks of code to methods. Earlier we saw the loop construct for writing an indefinite loop:

```
loop do
  puts "are we there yet?"
end
```

The keywords do and end and the code between them form a block. They can either be written with do...end, or with braces, like so:

```
loop { puts "are we there yet?" }
```

It's conventional to use do...end for multi-line blocks and braces for one-liners, but they mean exactly the same thing. They package up a chunk of code, and pass it as an implicit argument to the loop method. loop can invoke the block using the yield keyword. To make the block execute repeatedly, loop could be implemented like this:

```
def loop
  yield while true
end
```

You can think of blocks as first-class functions that are passed as implicit arguments to methods. Ruby has a for loop construct, but most iteration over collections is done using a method called each. It takes a block and invokes it once for every item in the collection. Whereas methods specify their parameters between parentheses, blocks put them between pipes, after the do or opening brace.

```
ingredients = ["oil", "vinegar", "mustard"]

ingredients.each { |thing| puts thing }
```

This program prints each ingredient in turn. The implementation of this in Array might look like this:

```
def each
  n = 0
  until n == self.size
    yield self[n]
    n += 1
  end
end
```

This method invokes the block it's given once for each item in the array. The arguments following the yield keyword become bound to the parameters between the block's pipes. The value of a yield expression is the return value of the block it's calling, that is the value of the block's final statement. Unlike methods, it is not an error to invoke a block with the wrong number of arguments.

You cannot return from a block. The `return` keyword always returns from the enclosing method. The method below would return "vinegar", and the final iteration of the loop inside `Array#each` will be skipped.

```
def find_acidic
  ingredients = ["oil", "vinegar", "mustard"]

  ingredients.each do |thing|
    return thing if thing.start_with?("v")
  end
end
```

You can interrupt a block using the `next` keyword. `next` inside a block works like `return` inside a method, returning control to the block's caller. In practice this means loops written using `each` can use `next` to skip elements.

A method can check whether it was passed a block by calling `block_given?`. It can get an explicit reference to the block by defining a final parameter prefixed with `&`. This allows an object to retain a reference to the block and invoke it later.

```
def handle_event(type, &block)
  @handler = block
end
```

The block can be invoked later using `@handler.call(args)`. A block can be passed to another method that expects a block by prefixing it with `&`, as in `list.each(&@handler)`.

A.2.5. Constants

In Ruby, a constant is any named binding beginning with an uppercase letter. Whereas local variables must begin with lowercase letters, a name like `Robot` or `MAX_SIZE` denotes a constant. We've seen some constants already: `class` defines a constant whose value is the class itself, that is `class Robot` is like saying:

```
Robot = Class.new do
  # ...
end
```

In general, a constant can reference any value. Ruby will print a warning if a constant is reassigned, but will not raise an error. The values that constants refer to can still be mutated; it is only the reference that is constant, not the thing being referred to.

Constants are scoped by classes. The following defines a top-level constant called `VERSION`, and a constant inside the `Index` class called `MAX_SIZE`.

```
VERSION = "2.3.0"

class Index
  MAX_SIZE = 1024
end
```

Code inside the `Index` class, that is any code appearing between `class Index` and `end`, even inside nested classes, can refer to `MAX_SIZE` by just that name. All code outside the `Index` class must use `Index::MAX_SIZE`. If there is a constant named `MAX_SIZE` outside the `Index` class, then

code inside `Index` must use `::MAX_SIZE` to refer to it — this is sometimes necessary to refer to class names that are used in multiple modules.

A.3. Built-in data types

As well as the core language semantics defined above, the Ruby runtime includes numerous built-in classes for different types of data and other common functionality. Below are listed the data types used in Jit.

A.3.1. `true`, `false` and `nil`

The boolean values are denoted `true` and `false`, and have their usual meaning. They result from any comparison like `1 + 1 == 2` or `1 < 2`. The special value `nil` denotes the absence of a value.

Conditional constructs like `if`, `unless`, `while` and `until` can accept conditions that evaluate to any value, not just literal `true` and `false`. For logical purposes, the values `false` and `nil` are considered untruthful, and all other values are considered truthful. This includes zero, the empty string, the empty array, and other empty collections that are treated as untruthful in some languages.

Values can be logically combined using `and`, `or` and `not`. `x and y` first evaluates `x`, and if it is truthful returns the value of `y`. `x or y` first evaluates `x`, and if it is not truthful it returns the value of `y`. `not x` evaluates to `false` if `x` is truthful, and `true` otherwise.

Ruby also has operators `&&` and `||`, which mean the same as `and` and `or` but have higher precedence. They are usually used in assignments, as in:

 `x = y && z`

This means the same as `x = (y && z)`, i.e. assign the value of `y && z` to `x`. Whereas, `x = y and z` means `(x = y) and z`, that is, assign the value of `y` to `x`, and if `y` is truthful then evaluate `z`.

A.3.2. Integer

Integers are arbitrary-precision signed whole numbers. They are denoted by their decimal representation, as in `1024`. Numerals can include underscores for legibility, which Ruby ignores, as in `120_000`. A numeral beginning `0b` is a binary¹³ representation, as in `0b1101` is the same as `13`. A numeral beginning `0x` is a hexadecimal¹⁴ representation; `0x9f` is equal to `159`. A numeral beginning with just `0` is octal¹⁵, so `0755` and `493` are the same.

Addition, subtraction and multiplication of integers (`+`, `-`, and `*`) all produce integers. Division (`/`) on integers also produces integers; `x / y` is the value of `x` divided by `y`, rounded down to the nearest whole number.

Other numeric operators include `x % y`, which gives `x` modulo `y`, and `x ** y` which returns `x` to the power `y`. Integers also have a method called `times`, which executes the given block the given number of times, with an incrementing argument beginning at zero.

¹³https://en.wikipedia.org/wiki/Binary_number

¹⁴<https://en.wikipedia.org/wiki/Hexadecimal>

¹⁵<https://en.wikipedia.org/wiki/Octal>

```
3.times { |i| puts i }
# prints 0, 1, and 2
```

Calling `to_s` on an integer returns a string containing the number's decimal representation. `to_s` optionally takes an argument specifying the base, for example `n.to_s(2)` returns the binary representation of `n`. Calling `to_i` (also with an optional base) on a string reverses this conversion.

A.3.3. String

Strings represent blobs of textual or binary data, and are written by placing the text between double-quotes¹⁶. Within a double-quoted string, the syntax `#{ ... }` can be used to evaluate an expression and interpolate the result into the text.

```
"the result is #{ 1 + 1 }"
# == "the result is 2"
```

The `+` operator can also be used to concatenate strings, but the above syntax automatically converts the expression's value to a string, whereas `+` does not. All Ruby objects have a method called `to_s` that converts them to strings.

Some characters can be prefixed with a backslash to denote non-printing characters. Common examples include `\t` for tab, `\n` for line feed, `\r` for carriage return, and `\e` for the escape character.

Ruby also supports *heredocs*¹⁷, a more convenient method of writing multi-line strings. The variation I have used in this text is `<<~`, which strips any common leading spaces from all the lines, letting you indent the string consistently with the surrounding code. The method below returns `"alfa\n beta\ngamma\n"`.

```
def to_s
<<~STR
  alfa
  beta
  gamma
STR
end
```

A string is represented internally as an array of bytes, plus an encoding tag that indicates how the bytes should be interpreted into text. The default Ruby source code encoding is UTF-8. The encoding `Encoding::ASCII_8BIT` indicates the string represents binary data, not text, although when printed it will be interpreted as ASCII where possible.

A single character can be selected from a string using brackets, `"hello"[1]` gives `"e"`. A negative index reads from the end of the string; `"world"[-3]` is `"r"`. A range selects a sequence of characters: `"Ruby"[2..-1]` gives `"by"`. The `byteslice` method performs the same operation but selects a range of bytes, not characters — a character may be represented by multiple bytes. `size` gives the string's length in characters, and `bytesize` the length in bytes.

Strings are mutable. New data can be added to the end of a string by calling `concat`, or to the beginning by calling `prepend`.

¹⁶Ruby strings can also be single-quoted, but they behave differently and I have not used them here.

¹⁷https://en.wikipedia.org/wiki/Here_document

```
str = "hello "
str.concat("world")
# str == "hello world"
```

You can find more information about the methods supported by `String` in the online documentation¹⁸ or by running `ri String`.

A.3.4. Regexp

Regular expressions¹⁹ are written delimited by slashes, as in `/(R|r)uby/`. An `i` after the trailing slash means the pattern is case-insensitive.

To match a string against a pattern we use the `match` method. The expression `/(R|r)uby/.match("ruby")` returns a `MatchData` object, `<MatchData "ruby" 1:"r">`. Bracket notation can be used to select portions of the string captured by the pattern: `/(R|r)uby/.match("ruby")[1]` returns `"r"`. To check whether a string matches a pattern we can also use the `=~` operator. `/(R|r)uby/ =~ "ruby"` returns `nil` if the string does not match, and a number otherwise. The number indicates the offset in the string where the pattern begins.

More information about the `Regexp` and `MatchData` classes is available online²⁰ or by running `ri Regexp`.

A.3.5. Symbol

Symbols are words prefixed with a colon, as in `:burger`. They might look like strings, but they're not. We don't typically perform string operations on them, and they cannot be mutated. They are just used to name things, often to refer to the program's methods as in `attr_reader`, or to act as keys in a hash, described below.

Symbols can be used anywhere a block is expected. If you see something like `list.each(&:thing)`, that will invoke the `thing` method on each item in the list. In general, a symbol prefixed with `&` is the same as a block that takes an argument and returns the result of calling the named method on that argument. For example, these two expressions are equivalent:

```
list.each(&:do_something)

list.each { |value| value.do_something }
```

Symbols can be generated from strings by calling `to_sym` on the string, but it is rare to do this. Symbols are typically hard-coded into the program, and not generated from user input.

A.3.6. Array

An array is a dynamically sized ordered sequence of values. An array literal is a list of expressions, separated by commas and enclosed in brackets, for example:

```
list = [true, 6, "git"]
```

¹⁸<https://docs.ruby-lang.org/en/2.3.0/String.html>

¹⁹https://en.wikipedia.org/wiki/Regular_expression

²⁰<https://docs.ruby-lang.org/en/2.3.0/Regexp.html>

Arrays can also be created using `Array.new`, passing the initial size and an optional initial value:

```
Array.new(3, 0) # returns [0, 0, 0]
```

Values can be retrieved by position starting from zero; `list[1]` is 6. A negative position reads from the end of the array; `list[-1]` is "git". A range can be used to select multiple values: `list[1..-1]` gives [6, "git"]. Giving a position that's out of bounds produces `nil` as the result, rather than throwing an error. If you want to get an error if a non-existent position is accessed, use `fetch`: `list.fetch(3)` will raise an exception. Calling `list.sample` returns a random element from the array.

An array's elements can also be accessed via *destructuring*. Assigning an array to multiple variables assigns its elements in order to each one.

```
x, y, z = list

# x == true, y == 6, z == "git"
```

If you use an array literal on the right of the assignment, the braces can be omitted, as in `x, y = :foo, :bar`. This is used to swap the values of variables, i.e. `x, y = y, x`.

This kind of destructuring is not performed when an array is passed as an argument to a method, but can happen when an array is passed to a block. For example, if an array of two elements is passed as a single argument to a block of the form `{ |x, y| ... }`, then `x` and `y` take on the values of the array's elements.

Subarrays can be generated by `take` and `drop`. `list.take(n)` returns an array containing the first `n` elements of `list`, and `list.drop(n)` returns everything *except* the first `n` elements. These methods return a new array, which does not share structure with the original one.

The `+` operator concatenates two arrays. A new array is formed containing the contents of each input, but neither input is modified. In contrast, the `concat` method modifies its receiver:

```
list.concat([:burger, false])
# list == [true, 6, "git", :burger, false]
```

The `&` operator returns the elements that are common to two arrays, for example, `list & ["git", 42, 6] == [6, "git"]`.

The `delete` method removes any items equal to the input from the list. For example, `list.delete(6)` leaves the list holding [true, "git"]. The `-` operator returns a new list formed by removing the elements in one list from another, for example `list - [true, "git"] == [6]`. `compact` removes any `nil` items from the array, and `list.uniq` removes any duplicate values.

New elements can be added at the end by calling `push`, and removed using `pop`. Items can be removed from the beginning using `shift`. New items can be inserted at an arbitrary position using `insert`.

```
list = [true, 6, "git"]

list.pop          # returns "git", list is now [true, 6]
list.push(:burger) # list is now [true, 6, :burger]
list.shift        # returns true, list is now [6, :burger]
```

```
list.insert(1, "jit") # list is now [6, "jit", :burger]
```

We can check whether an array contains a value by calling, say, `list.include?(:fries)`. `find_index` returns the position of the first element for which the given block returns true, for example `list.find_index { |v| v.is_a? Symbol }` returns `1`. `list.empty?` returns `true` if the array contains no items, and `size` returns its length.

Arrays can be transformed in various ways. `list.reverse` returns a copy of the array with its elements in reverse order. `list.reverse!` reverses the array itself without copying it. `list.sort` returns a sorted copy of the array, and `list.sort!` sorts the array in place. `transpose` swaps the rows and columns of a two-dimensional array:

```
grid = [[3, 7, 8, 4],
        [1, 5, 9, 6]]

grid.transpose == [[3, 1],
                  [7, 5],
                  [8, 9],
                  [4, 6]]
```

`zip` combines two arrays pairwise, that is: `[1, 2, 3].zip([:a, :b, :c])` returns `[[1, :a], [2, :b], [3, :c]]`.

An array may be converted into a string by calling `join`, which takes an argument specifying the text to insert between each element. It can also be serialised using `pack`²¹, a powerful tool for formatting various data types as byte sequences. Its effect can be reversed using `String#unpack`.

Finally, arrays can be compared for equality. Two different array objects are equal according to `==` if all their corresponding elements are equal. They can also be compared for sorting: array `x` sorts before array `y` if its first element sorts before the first of `y`. If the first elements are equal, the second elements are compared and so on.

The `Array` class is documented in full online²², or you can find out more by running `ri Array`.

A.3.7. Range

A range is a lightweight representation for a sequence of consecutive values. It is denoted by two expressions separated by two or three dots, as in `1 .. 10` or `"a" ... "d"`. Two dots means the range includes the end value, and three dots means it does not. For example:

```
("a" ... "d").to_a # returns ["a", "b", "c"]
```

The range only stores its start and end points, not all the values in between. It can be used to check if a value is within certain bounds, for example `(1..10).include?(5)` returns `true`. It can also iterate the range while skipping elements, for example:

```
(1 .. 10).step(3) { |n| puts n }

# prints 1, 4, 7 and 10
```

²¹<https://docs.ruby-lang.org/en/2.3.0/Array.html#method-i-pack>

²²<https://docs.ruby-lang.org/en/2.3.0/Array.html>

Run `ri Range` or visit the online documentation²³ to learn more.

A.3.8. Hash

A hash is an associative data structure that maps keys to values. In other languages it's known as a hash table²⁴, a map, a dictionary, or an associative array. In Ruby they are denoted using braces surrounding a comma-separated list of key-value pairs, joined using `=>`. The keys can be any kind of value that can be compared for equality, but symbols and strings are most common.

```
stats = {  
  :tests => 487,  
  :assertions => 788,  
  :failures => 0,  
  :skipped => 0  
}
```

Like arrays, hash values are retrieved and set using bracket notation, as in `hash[key]` and `hash[key] = value`. Looking up a key that is not present results in `nil` rather than an error. Use `hash.fetch` if you want to force an error if the key is missing, or generate a default value using a block:

```
hash.fetch(:errors) { 1 + 1 } # returns 2
```

A general default behaviour can be set for missing keys when a hash is created using `Hash.new`. This is usually used to assign a starting value for hashes that accumulate data.

```
stats = Hash.new { |hash, key| hash[key] = 0 }  
  
stats[:tests] += 1  
stats[:tests] += 1  
stats[:assertions] += 3  
  
# stats is now { :tests => 2, :assertions => 3 }
```

`hash.keys` returns an array of the keys in the hash, which will be unique. `hash.values` returns an array of the values, which may not be unique. `hash.size` returns the number of keys in the hash. To check if a hash contains a certain key, use `hash.has_key?(key)`. To look up the key pointing to a certain value, it's `hash.key(value)`.

The `merge` method combines two hashes together, returning a new hash, not modifying either of the inputs. Calling `hash.delete(:tests)` removes the `:tests` key from the hash and returns its associated value.

When a hash is iterated using `each`, pairs of its keys and values are passed to the block.

```
total = 0  
  
stats.each do |name, count|  
  total += count  
end  
  
# total is now 5
```

²³<https://docs.ruby-lang.org/en/2.3.0/Range.html>

²⁴https://en.wikipedia.org/wiki/Hash_table

The `each_key` and `each_value` methods iterate over just the keys or values respectively.

When a hash is passed as the last argument to a method, its braces can be omitted. That is, `print("results", :tests => 1)` means the same thing as `print("results", { :test => 1 })`.

Use `ri Hash` to learn more about hashes, or look up the documentation online²⁵.

A.3.9. Struct

Whereas a hash is an open-ended collection that contains an arbitrary set of keys, a struct is an associative structure with fixed, known fields. Ruby's `Struct` class acts as a shorthand for creating a class with attribute accessors for the given names, for example:

```
Stats = Struct.new(:tests, :assertions, :failures)
```

This is equivalent to the following class:

```
class Stats
  attr_accessor :tests, :assertions, :failures

  def initialize(tests, assertions, failures)
    self.tests      = tests
    self.assertions = assertions
    self.failures   = failures
  end
end
```

`Struct` also implements equality. By default, two arbitrary Ruby objects are not considered equal to each other. Equality is achieved by classes like `Integer`, `String`, `Array` and so on implementing their own equality methods. If you use `Struct` to generate a class, members of that class will be equal if their fields are all equal. This means structs can be easily compared for equality, used as hash keys, stored in sets, and so on.

Additional methods can be added to a `Struct` class by passing a block to `Struct.new`.

```
Stats = Struct.new(:tests, :assertions, :failures) do
  def passes
    tests - failures
  end
end
```

When creating instances of a `Struct` class, you do not have to pass all the constructor parameters, for example it is legal to call `Stats.new(2, 3)`, omitting the `failures` field. Any omitted fields will have their values set to `nil`.

A.4. Mixins

Earlier we saw how subclassing can be used to inherit methods from an existing class. Every class has only one parent class (the `Object` class if none is specified), and so this inheritance technique is inherently hierarchical.

²⁵<https://docs.ruby-lang.org/en/2.3.0/Hash.html>

Ruby has another way of sharing methods that is non-hierarchical, which is referred to as *mixins* or *modules*. A module is like a class, in the sense that it defines a set of methods. The difference is that a module cannot create new objects, and a class may inherit from multiple modules.

Many Ruby modules add behaviour based on a class having some particular trait. For example, if your class has an `each` method that iterates over its contents, then you can include the `Enumerable` module and get a lot of collection-related functionality. An implementation of `Enumerable#map` might look like this:

```
module Enumerable
  def map
    results = []
    each { |value| results.push(yield value) }
    results
  end
end
```

In any class with an `each` method, we can write `include Enumerable` and gain access to this `map` implementation. The `map` method of `Enumerable` will use the `each` method from whatever class it has been included into.

There are two ways of using a module to import methods. Inside a class or module, `include TheModule` makes all the methods in `TheModule` available as instance methods in the current class. The other way is to use `extend`. Calling `object.extend(TheModule)` on any object adds the methods in `TheModule` to just that object and no others. This can be used to add class methods that are defined in a module.

Ruby has a number of built-in modules that provide functionality to its core data types. The ones you'll encounter most commonly are `Enumerable` and `Comparable`.

Sometimes, modules are used merely as namespaces. That is, they're never included into another class, but they have classes and methods nested inside them just to group related functionality. You'll see this in the `Command`, `Diff` and `Pack` sections of the Jit codebase.

A.4.1. Enumerable

The `Enumerable` module provides methods that are useful for collections: arrays, hashes, and so on. It can be included into any class that has an `each` method. `each` defines an iteration procedure for the class, and `Enumerable` builds on that to provide a common interface across all collection types.

One of its most commonly used methods is `map`. This builds an array by applying a block to every member of the collection. For example, the following code converts a list of strings into uppercase copies of the original data.

```
["apples", "oranges", "bananas"].map { |word| word.upcase }

# returns ["APPLES", "ORANGES", "BANANAS"]
```

The `Symbol` shorthand is frequently used with `map`; the above could be written as `map(&:upcase)`. Some collections also provide `map!`, which replaces the original content of the

array with the result rather than returning a new array. A variant of `map` is `flat_map`, which is useful when the block you're using produces arrays and you want to concatenate them.

```

["hello", "world"].map(&:chars)
# returns [["h", "e", "l", "l", "o"], ["w", "o", "r", "l", "d"]]

["hello", "world"].flat_map(&:chars)
# returns ["h", "e", "l", "l", "o", "w", "o", "r", "l", "d"]

```

Many methods exist for querying the contents of a collection. `any?` takes a block and returns `true` if the block returns `true` for any member of the collection. `all?` returns `false` if the block returns `false` for any member of the collection, and `true` otherwise. `none?` does the opposite of this; it returns `false` if the block returns `true` for any member.

```

[1, 3, 5].any?(&:even?)    # false
[1, 3, 5].all?(&:odd?)     # true
[1, 3, 5].none?(&:even?)   # true

```

`select` returns an array containing the elements of the collection for which the block returns `true`, and `reject` returns a list of the elements for which the block is `false`. These both have a mutating variant, `select!` and `reject!`, which modifies the collection rather than returning a new one. `partition` returns a pair of arrays that combines the results of `select` and `reject`. `find` is like `select`, but only returns the first matching element. `grep` returns an array containing the elements that are case-equal (as in the `case` statement) to its argument.

```

(1..8).select(&:even?)      # [2, 4, 6, 8]
(1..8).reject(&:even?)       # [1, 3, 5, 7]
(1..8).partition(&:even?)    # [[2, 4, 6, 8], [1, 3, 5, 7]]
(1..8).find(&:even?)        # 2
["hello", "world"].grep(/r/)  # ["world"]

```

The `reduce` method takes a collection and reduces it to a single value by applying a block to each consecutive element. For example, to sum all the numbers in a range, we write:

```

(1..8).reduce(0) { |x, y| x + y }

# returns 36

```

`reduce` takes the starting value, `0`, and the first element of the collection, `1`, and applies the block `{ |x, y| x + y }` to these inputs, giving `1`. This result becomes the first argument for the next iteration, and the second argument is the next element, `2`. Calling the block with `1` and `2` gives `3`, and so on. In general, combining a list by calling some method on its members, like this:

```
[a, b, c, d].reduce(z) { |x, y| x.f(y) }
```

Is the same as chaining all the elements together with that method: `z.f(a).f(b).f(c).f(d)`.

`Enumerable` contains a number of methods for dealing with collections of sortable values. `max` returns the greatest value, and `min` the smallest. `sort` returns a sorted array of the collection's contents, and `sort!` sorts the collection in place. These methods both take a block, which is used to compare the elements if no pre-defined sort order exists for them. `sort_by` sorts a collection by the result of a block. For example, to sort a list of strings by their length, it's `strings.sort_by(&:size)`.

Finally, there are methods for iterating the collection in different ways. `reverse` returns a copy of the collection in reverse order, while `reverse_each` iterates the collection in reverse order. `each_with_index` is like `each` but yields an additional parameter to the block: a zero-based counter that increments with each iteration. `with_index` can also be combined with other methods. For example, `map.with_index` maps over a collection, giving the block the counter value as well as the current element. The following code selects those numbers whose value is less than their position in the array:

```
list = [7, 6, 8, 3, 1, 4, 5, 2]
list.select.with_index { |n, i| n < i }

# returns [1, 4, 5, 2]
```

Finally, `to_a` converts any collection to an array, based on its `each` implementation. Run `ri Enumerable` or visit the online docs²⁶ to find out more.

A.4.2. Comparable

`Comparable` is the module that's included into any class of values that can be sorted, for example `Integer`, `String`, and `Array`. To define a sort order for a class, we implement the `<=>` operator, which should return a negative value if `self` is less than the argument, a positive value if `self` is greater than the argument, and zero if they're equal. For example, let's say our `Robot` class can be sorted by its name. We'd implement this as:

```
class Robot
  include Comparable

  attr_reader :name

  def initialize(name)
    @name = name
  end

  def <=>(other)
    if @name < other.name
      -1
    elsif @name > other.name
      1
    else
      0
    end
  end
end
```

Including `Comparable` makes all the usual comparison operators (`<`, `<=`, `>`, `>=`) work with the class, and means its members can be sorted. The `<=>` operator can be invoked directly, so the above could be shortened to:

```
class Robot
  include Comparable

  attr_reader :name
```

²⁶<https://docs.ruby-lang.org/en/2.3.0/Enumerable.html>

```
def initialize(name)
  @name = name
end

def <=>(other)
  @name <=> other.name
end
end
```

A.5. Libraries

Everything we've covered above is part of Ruby's default runtime environment — it's available out of the box without loading any additional libraries. As well as these features, Ruby comes with a rich standard library of packages that provide commonly used data structures, algorithms, and utilities. I will not document these extensively, but simply mention the packages that Jit uses, so that you know what they are and can consult the documentation for further information.

As usual, you can use `ri` to look up the docs for any of these features, and you can load them into an IRB session either by using the `-r` switch when you launch IRB:

```
$ irb -r pathname -r zlib
```

Or, by using the `require` keyword during your IRB session.

```
>> require "pathname"
>> require "zlib"
```

A.5.1. Digest

The `Digest` module²⁷ provides implementations of various cryptographic hash functions. Git uses SHA-1, which can be loaded by requiring `"digest/sha1"` and is found in the `Digest::SHA1` class. This class can either hash an entire string all at once, as in `Digest::SHA1.hexdigest(string)`, or hash data that is streamed in incrementally.

```
digest = Digest::SHA1.new
until file.eof?
  digest.update(file.readline)
end
digest.hexdigest
```

The `hexdigest` method returns the hash as a hexadecimal string, and `digest` returns it in binary format.

A.5.2. FileUtils

`FileUtils`²⁸ provides a collection of helper functions on top of the lower-level `File`²⁹ and `Dir`³⁰ classes used for interacting with the filesystem. We use the `mkdir_p` method for creating missing directories, and `rm_rf` for recursively deleting them.

²⁷<https://docs.ruby-lang.org/en/2.3.0/Digest.html>

²⁸<https://docs.ruby-lang.org/en/2.3.0/FileUtils.html>

²⁹<https://docs.ruby-lang.org/en/2.3.0/File.html>

³⁰<https://docs.ruby-lang.org/en/2.3.0/Dir.html>

A.5.3. Forwardable

Forwardable³¹ provides a shorthand for defining methods that delegate to an object's instance variables. For example, our Robot might have a @brain, to which it delegates anything to do with making decisions.

```
class Robot
  def work_out(*args, &block)
    @brain.compute(*args, &block)
  end
end
```

Forwardable allows this method to be defined more concisely, by specifying the name of the variable being delegated to, the method to call on it, and the method to define in the current class.

```
class Robot
  extend Forwardable
  def_delegator :@brain, :compute, :work_out
end
```

Multiple delegated methods can be defined at once using `def_delegators`. All the methods listed after the instance variable will have delegator methods generated for them, with the same names in the current class.

```
class Robot
  extend Forwardable
  def_delegators :@brain, :compute, :sleep, :wake_up
end
```

A.5.4. open3

The `Open3` module³² is a high-level wrapper around the process and I/O management associated with launching a child process. Its `popen2` method takes a command to execute, starts a process from that command, and returns the standard input and output streams for that new process.

A.5.5. OptionParser

`OptionParser`³³, which is loaded by requiring "optparse", allows for declarative parsing of command-line inputs. A Ruby process has access to all the arguments given to it in the constant `ARGV`, and `OptionParser` can recognise the common syntax for option names and their arguments in this list of strings. Its `Command` classes use it to process their inputs.

A.5.6. Pathname

In programming, many things that are structured data are represented as strings. Filesystem paths are one of these things, and there's a surprising amount of complexity in handling them correctly. The `Pathname` class³⁴ wraps a string representing a path in an object that provides

³¹<https://docs.ruby-lang.org/en/2.3.0/Forwardable.html>

³²<https://docs.ruby-lang.org/en/2.3.0/Open3.html>

³³<https://docs.ruby-lang.org/en/2.3.0/OptionParser.html>

³⁴<https://docs.ruby-lang.org/en/2.3.0/Pathname.html>

methods for manipulating it. Pathnames can be combined via `join`, and split into their segments using `basename`, `dirname`, `ascend`, `descend`, and `each_filename`. The file extension can be replaced using `sub_ext`, and the relative path from one file to another can be computed using `relative_path_from`.

A.5.7. Set

A `Set`³⁵ is a kind of collection, like `Array` and `Hash` that are part of the core language. A `Set` is an unordered collection of unique values. This means if you add a new object to a set and it returns true for the `==` operator with any object already in the set, the new object is silently discarded. Being unordered means that, unlike with arrays, you should not assume items are yielded by the `each` method in any particular order.

Sets are useful when you're collecting some values and only want to know if a value is present or not, without knowing how many times it occurs. Checking whether a set contains a value is not affected by the set's size, whereas finding a value in an array takes more time the longer the array gets. We say sets find values in *constant time* while arrays take *linear time*.

`SortedSet` is like the `Set` class and its interface is identical, but it makes sure items come out of its each iteration in sorted order.

A.5.8. Shellwords

The `Shellwords` module³⁶ is like `Pathname` in the sense that's a safety feature for processing string data. If you're constructing commands to be executed from dynamic input, it's unsafe to use string interpolation. For example:

```
system "rm -rf #{filename}"
```

If `filename` has the value `"* bar.txt"`, this command will delete all the files in the current directory, rather than the single file named `* bar.txt`. Instead, the command should be built as an array and then joined using `shelljoin`³⁷.

```
command = ["rm", "-rf", filename]
system Shellwords.shelljoin(command)
```

This makes sure that the arguments are correctly quoted so they're not misparsed. `Shellwords.shellsplit` does the opposite of this, letting you parse a string in the same way the shell does.

A.5.9. StringIO

`StringIO`³⁸ wraps a string and lets you treat it as an `IO` object, which is usually used for reading from files, or standard input. You can use `read`, `readbyte` and other methods to walk through the string without needing to track the current read position yourself.

³⁵<https://docs.ruby-lang.org/en/2.3.0/Set.html>

³⁶<https://docs.ruby-lang.org/en/2.3.0/Shellwords.html>

³⁷The `system` method can take multiple arguments and perform this safety task for you, but Jit also uses other mentions for running subprocesses that don't have this feature.

³⁸<https://docs.ruby-lang.org/en/2.3.0/StringIO.html>

A.5.10. StringScanner

The `StringScanner` class³⁹, loaded by requiring "strscan", is helpful for stateful parsing tasks. It keeps a cursor into the string and lets you search through it for the next occurrence of a string or regular expression. The next scan will begin from the position of the last match found.

A.5.11. Time

`Time` is a class in the Ruby core that represents times. For example, `Time.now` returns the current system time as a `Time` value. Requiring "time" makes some additional methods available for serialising and parsing times⁴⁰. In Jit we use the `strptime` method for parsing times in Git commit objects.

A.5.12. URI

The `URI` class⁴¹ parses strings representing URIs⁴² into an object to allow the scheme, host, path and other elements to be correctly extracted.

A.5.13. Zlib

The `zlib` module⁴³ provides an interface to the zlib compression library⁴⁴. Like `Digest::SHA1`, its `Deflate` and `Inflate` classes can either process an entire string at once, or incrementally compress or decompress a string, useful for streaming applications.

³⁹<https://docs.ruby-lang.org/en/2.3.0/StringScanner.html>

⁴⁰<https://docs.ruby-lang.org/en/2.3.0/Time.html>

⁴¹<https://docs.ruby-lang.org/en/2.3.0/URI.html>

⁴²https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

⁴³<https://docs.ruby-lang.org/en/2.3.0/Zlib.html>

⁴⁴<https://zlib.net/>

Appendix B. Bitwise arithmetic

As well as the usual arithmetic operations, integers support *bitwise operations*¹. These operations show up when processing binary file formats, and when representing sets of function options in low-level languages like C. These operations aren't specific to Ruby, so if you're already familiar with them, feel free to skip this section.

& is the bitwise-and operator, and | is bitwise-or. These operators take the binary representations of two numbers, and compare their respective bits, performing a logical-and or a logical-or operation on each pair of bits. For example, take the numbers 26 and 9, which in binary notation are `0b11010` and `0b01001`, padding each to the same length with leading zeroes. To take the bitwise-and of these numbers, we look at their binary representations and make a new binary numeral with a 1 in each column in which both numbers have a 1, and 0 in all other columns. To take the bitwise-or, we place a 1 in any column in which either numeral has a 1.

Figure B.1. Bitwise-and and bitwise-or of two numbers

26:	11010	11010
9:	AND 01001	OR 01001
	-----	-----
	01000	11011

`0b01000` is equal to the decimal number 8, while `0b11011` equals 27. So, `26 & 9` equals 8, and `26 | 9` is 27. These results do not depend on how the numbers are originally written down; however the inputs to these operators were generated, the operators treat them as though they are represented in binary.

These operations are often used to represent sets of options using a *bitmask*². For example, the `File.open`³ method has an associated set of constants used to set the mode in which the file is opened. To open a file for writing, creating it if it does not exist, and raising an error if it already exists, we write:

```
file = File.open("file.txt", File::WRONLY | File::CREAT | File::EXCL)
```

This interface is a direct copy of the underlying C function⁴:

```
file = open("file.txt", O_WRONLY | O_CREAT | O_EXCL);
```

The way this works is that each `File` or `O_` constant has a different value and each is a power of 2, meaning its binary representation is a one followed by all zeroes. The bitwise-or of these numbers gives a result whose bits indicate which settings were used.

Figure B.2. Setting a `File.open` mode with bitwise-or

File::WRONLY		1		1
File::CREAT		64		10000000
File::EXCL		128		100000000
	-----+-----+			-----
bitwise-or		193		110000001

¹https://en.wikipedia.org/wiki/Bitwise_operations_in_C

²[https://en.wikipedia.org/wiki/Mask_\(computing\)](https://en.wikipedia.org/wiki/Mask_(computing))

³<https://docs.ruby-lang.org/en/2.3.0/File.html#method-c-open>

⁴<https://manpages.ubuntu.com/manpages/bionic/en/man2/open.2.html>

To find out if a setting was used, we take the bitwise-and of this result with the relevant constant. If the setting is enabled, this will give a non-zero value.

Figure B.3. Checking a `File.open` mode with bitwise-and

		193		11000001
File::CREAT		64		10000000
- - - - -	+-----+-----+			
bitwise-and		64		10000000

Any setting that's not enabled will give zero when we compute the bitwise-and of its constant with the value.

Figure B.4. Checking a non-matching `File.open` mode

		193		11000001
File::RDWR		2		10
- - - - -	+-----+-----+			
bitwise-and		0		0

The `<<` and `>>` operators perform left- and right-bitshift operations on integers. The left-shift operator effectively moves the bits of the number one place to the left, inserting zeroes in the rightmost column. For example, take 27, which is `0b11011` in binary. Left-shifting it by two places gives `0b1101100`, or 108. Therefore, `27 << 2` gives 108. The right-shift operator moves the bits in the opposite direction, dropping the rightmost digits. Right-shifting `0b11011` by two places gives `0b110`, or 6, so `27 >> 2` is equal to 6.

These are used to pack multiple numbers into a single one, often used in binary file formats. For example, Git's pack file format includes some bytes that contain a boolean flag, a three-bit number, and a four-bit number, all packed into one 8-bit byte. Say we want to pack boolean flag (`0b1`), the number 6 (`0b110`) and the number 11 (`0b1011`) into a single byte. We can concatenate these bits to get `0b11101011`, or 235. The way this concatenation is done is by left-shifting each value by the appropriate amount, and taking the bitwise-or of the results.

The following table shows the values we're storing, their binary representations, and finally those binary values left-shifted by some amount.

Figure B.5. Packing values into a byte

true		1		10000000	(<code><< 7</code>)
6		110		1100000	(<code><< 4</code>)
11		1011		1011	(no shift)
- - - - -	+-----+-----+				
bitwise-or:					<code>11101011</code>

To read values back out of this byte, we use the `>>` and `&` operators. Say we went to read the number 6 back out — that's stored in the second, third and fourth most-significant bits, with a further four less-significant bits after it. So, first we right-shift four places: `235 >> 4` gives 14, or `0b1110`. We then select the three least significant bits of this value using the bitwise-and with `0b111`, or 7: `14 & 7` gives 6. So to read values out of packed bytes, we right-shift past the less-significant bits we're not interested in, then bitwise-and to select those we want.

Numbers used in bitwise operations are usually written in binary, since that gives the most direct information about what's happening, or in hexadecimal, which is more compact. Hexadecimal

has the property that, since 16 is a power of 2, each hexadecimal digit stands for exactly four bits in the binary representation.

Figure B.6. Decimal, hexadecimal and binary representations of numbers

Dec	Hex	Bin	Dec	Hex	Bin
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	a	1010
3	3	0011	11	b	1011
4	4	0100	12	c	1100
5	5	0101	13	d	1101
6	6	0110	14	e	1110
7	7	0111	15	f	1111

So, you'll often see a bitmasking operation use `0xff` to select an eight-bit byte's worth of information from the end of a value. `0x7` selects three bits, so `0x7f` selects seven bits. It's also worth knowing that powers of 2 are equivalent to 1 left-shifted the appropriate amount: `2 ** n == 1 << n`.

These techniques are historically used in data formats and low-level programming because they let us pack numbers into much smaller amounts of space than if we used more complicated structures, or string representations. Bitwise operations are also very efficiently carried out by a CPU.