*Error lfs*

*$ git lfs install –skip-smudge*

# Bash Commands:

Note that “~” corresponds to *c:\users\<username>*

$ cd ~

Remove a directory

$ rm –r folder

$ echo –n abcd > file01.txt //-n suppress newline

Creates four empty files

$ touch a b c d

Will display the line numbers

$ cat –n file01.txt

$ pwd

$ ls

$ ls –F //to help distinguish directories from regular files

$ ls –la

$ ls -a

$ touch text01.txt // Create empty file

$ echo “line 1” > test01.txt

$ echo “line 1” >> test01.txt

$ cat test01.txt

$ find .git/objects

# Git Commands

$ git <git-options> <command> <command-options> <operands>

$ git

$ git help glossary

$ git help –a //list of over 150 commands

$ git help –g //list of common guides

$ git help config

$ git config –h

$ git config --help

## Common Commands

|  |  |
| --- | --- |
| add | Add file contents to the index. |
| bisect | Find by binary search the change that introduced a bug. |
| branch | List, create, or delete branches. |
| checkout | Switch branches or restore working tree files. |
| cherry | Find commits yet to be applied to upstream (branch on the remote). |
| cherry-pick | Apply the changes introduced by some existing commits. |
| clone | Clone a repository into a new directory. |
| commit | Record changes to the repository. |
| config | Get and set repository or global options. |
| diff | Show changes between commits, commits and working tree, and so on. |
| fetch | Download objects and refs from another repository. |
| grep | Print lines matching a pattern. |
| help | Display help information. |
| log | Show commit logs. |
| merge | Join two or more development histories together. |
| mv | Move or rename a file, directory, or symlink. |
| pull | Fetch from, or integrate with, another repository or a local branch. |
| push | Update remote refs along with associated objects. |
| rebase | Forward-port local commits to the updated upstream head. |
| rerere | Reuse recorded resolution for merged conflicts. |
| reset | Reset current HEAD to the specified state. |
| revert | Revert some existing commits. |
| rm | Remove files from the working tree and from the index. |
| show | Show various types of objects. |
| status | Show the working tree status. |
| submodule | Initialize, update, or inspect submodules. |
| subtree | Merge subtrees and split repositories into subtrees. |
| tag | Create, list, delete, or verify a tagged object. |
| worktree | Manage multiple working trees. |

## Plumbing commands

$ git cat-file

$ git ls-files

the plumbing command, *cat-file*. You use two options here:

-t = type—shows the type of the object

-p = pretty—prints information about the object

|  |  |
| --- | --- |
| cat-file | Provide content or type and size information for repository objects |
| commit-tree | Create a new commit object. |
| count-objects | Count an unpacked number of objects and their disk consumption. |
| diff-index | Compare a tree to the working tree or index. |
| for-each-ref | Output information on each ref. |
| hash-object | Compute object ID and optionally create a blob from a file. |
| ls-files | Show information about files in the index and the working tree. |
| merge-base | Find as good common ancestors as possible for a merge. |
| read-tree | Read tree information into the index. |
| rev-list | List commit objects in reverse chronological order. |
| rev-parse | Pick out and massage parameters. |
| show-ref | List references in a local repository. |
| symbolic-ref | Read, modify, and delete symbolic refs. |
| update-index | Register file contents in the working tree to the index. |
| update-ref | Update the object name stored in a ref safely. |
| verify-pack | Validate packed Git archive files. |
| write-tree | Create a tree object from the current index. |

When both types are specified, if there is a possibility of Git not being able to tell the difference between a commit | branch | tag and one or more of the filenames or paths, then you can separate the two types using the special separation symbol “--”. Normally, this won’t be needed if a commit is expressed as a SHA1 value, but it may be needed if branch or tag names could be mistaken as names for files or paths.

As an example, the command $ git <command> a1b2c3d4 file1.txt might be clear enough, but git <command> my-tag-name -- my-file-name could be ambiguous enough when parsed to

require the “--” separator symbol.

Arguments supplied either to Git or to Git commands can be abbreviated as a single letter or spelled out as words. One important note here is that if the argument is spelled out, you must precede it with two hyphens, as in --global. If the argument is abbreviated, only one hyphen is required, as in -a

To tell Git to ignore certain files (meaning not to track them), you just need to list them in a *Git ignore file*. This is a text file named *.gitignore* that is placed at the root (top level directory) of the local environment

# Git Configuration

List the git configurations

$ git config --local --list

$ git config --global --list

$ git config --system --list

Setting git configurations temporarily. Changing the behavior of git log via the -c switch

$ git -c log.date=relative log -2

Setting Git configurations permanently

$ git config --local log.date relative

The --local switch is the default, and it may be omitted

Resetting git configurations

$ git config --local --unset log.date

Edit git configurations files

$ git config --local --edit

Configuring Git’s default editor

$ git config core.editor

$ git config core.editor notepad

$ git –c core.editor=vi config –-local --edit

You can exit the editor vi without saving anything by typing this: :q!

$ git –c core.editor=echo config –-local --edit

The resulting git config command prints the name of the file.

# Git History

**-p.** This option stands for *patch*, meaning that the history output also displays the differences, or patches, between each change

**--stat**

**--pretty:** This option allows you to specify format strings.

**--format:** This option allows you to create your own custom output format to see the different

pieces of log output in nearly any format.

Display history concisely, using one line per each commit.

**-oneline:**

**--author:**

**--decorate:**

Show the history, displaying the parent commit’s SHA1 ID for each commit.

git log --parents

git log --patch Display the history, showing the file differences between each commit.

git log --stat Display the history, showing a summary of the file changes between each commit.

git log --patch-with-stat Display the history, combining patch and stat output.

git log --oneline file\_one Display the history for file\_one.

git rev-parse Translate a branch name or a tag name to a specific SHA1 ID.

git checkout YOUR\_SHA1ID Change your working directory to match the version specified in YOUR\_SHA1ID.

git tag TAG\_NAME -m "MESSAGE" YOUR\_SHA1ID

Create a tag named TAG\_NAME, pointing to YOUR\_SHA1ID. The tag will have a short MESSAGE associated with it.

git tag List all tags.

git show TAG\_NAME Show information about the tag named TAG\_NAME.

Time options

--since

$ git log --since 01.01.2015

$ git log --since “1 month”

--until

--after,

--before

History by files

$ git log file1.txt

will show only the log entries where the file build.gradle was involved

Adding the *--name-only* option to the log command will show the list of files changed with each commit along with the change information

$ git log --oneline --name-only

Adding a path onto the command (separated from the rest of the command by a double dash [--]) allows you to see just the commits that involved changes on that path.

$ git log --name-only -- web/src/main/webapp

Log Output Format

$ git log --pretty=format:"%h %ad | %s%d [%an]" --date=short

$ git log --pretty=format:"%h ( %ad %an ) %s %d" --graph --date=short

**--pretty** defines the output format

**%h** is the abbreviated hash of the commit

**%ad** is the commit date

**%an** is the name of the author

**%s** is the commit message

**%d** specifies to show commit decorations (for example, branch identifiers or tags)

Searching History

Two other options available with the log command facilitate searching for text in files

The first option, *-G*, takes a regular expression as an argument and searches for commits that added or removed occurrences of this text.

The second option, *-S*, takes a string and searches for commits that changed the number of occurrences of the string.

There are two differences here:

➤➤ -G is intended to take a regular expression, while -S normally takes a string.

➤➤ -S only detects situations where the before and after versions of a file have different counts of occurrences of the string.

Partial staging:

You can use the -p option to do this, as in

$ git add -p <file or . or pattern>.

Bypassing the Staging Area

The shortcut is to use the *-am* option on the command line when doing a commit, as in *git commit*

-am “comment”.

The one caveat with the *-am* shortcut is that it will not work for new content or files. The first time

a file is added to Git, it must have the *git add* command done first.

Resetting the Author Information

$ git commit --amend –reset-author

There are 2 ways for a user to tell Git to include a commit message template at the time of doing a commit:

1. Use the *-t* (--template) option on the commit command itself.

$ git commit -t <template file location>

2. Configure the default location of the template file globally.

$ git config --global commit.template <template file location>

GIT STATUS

git status [<options>...] [--] [<pathspec>...]

Like other commands, this command can take path specifications, but those are not required. The “--” is a separator used to note where options end and path specifications start. It sits in between and is not required if the specifications are unambiguous enough

# Git Diff

Starting at the Staging Area

From the command line, you execute *git diff --staged*.

Diffing against a Specific Version (SHA1)

The syntax is git diff <identifier>.

If you say *git diff HEAD* then instead of going up to the staging area to check for differences, Git will bypass the

staging area and compare the working directory against what’s pointed to by HEAD.

$ git diff HEAD

Diff Names Only

If you only want to see the names of the files that are different, you can use the --name-only option

$ git diff --name-only

Diffing Two Commits

The diff command can also be used to diff two different commits in the local repository

$ git diff c25a62d fc5c99f

Compare a specific file in the HEAD revision against the version in the working directory

$ git diff HEAD:file1.txt file1.txt

Note that you could also filter this by an individual file.

$ git diff fc5c99f c25a62d file2.txt

You could have also included the separator “--” between the second SHA1 and the filename, as follows:

$ git diff c25a62d fc5c99f -- file2.txt

However, that is not necessary in this case because the form of the filename is different enough from a SHA1 value to not be confused for a revision.

Git blame

$ git blame file1.txt

$ git blame –L5,10 file1.txt //Line 5 to 10

$ git blame –L5,+6 file1.txt //Line 5 to 10

$ git log --oneline file1.txt

$ git blame SHA1.. –- file1.txt

### Git reset

While reset is useful for rolling back to a certain point, it can also be problematic

Only update the HEAD of the local repository

$ git rest --soft SHA1

(default) update the HEAD of the local repository and the staging area

$ git rest --mixed SHA1 //Default

Update the HEAD of the local repository, the staging area and the working directory

$ git rest --hard SHA1

For these reasons, it is recommended to not use reset or any Git operations that change history and could cause difficult merge scenarios on that has been pushed to a remote history

### Git revert

TAGS

$ git tag RelCandidate1 <SHA1 value>.

An annotated tag

$ git tag -a <rtag> <SHA1 value> -m "message".

To verify the SHA1 of the commit associated with the tag

git rev-parse <tag>.

If you then try to update that tag to point to another revision, you need to supply the -f option.:

$ git tag -f tag1 4e430fe

To delete a tag, you can use the -d option

$ git tag -d tag1

To see the details of the tag

$ git show annotatedTag1

Chapter 8 Branches

Use the --force option to go ahead and do the switch , overwriting uncommitted changes

$ git checkout --force

$ git branch -v //Verbose

$ git branch --list //list the local branches

$ git branch --list t\* //list the local branches with a pattern as an option

$ git branch -d test //delete a branch

$ git log delete\_candidate\_branch ^current\_branch --no-merges

The caret (^) symbol at the start of the second branch argument means *not*. So the way to read this command is *Show me the commits that are in delete\_candidate\_branch and are NOT in current\_branch*. The --no-merges option tells Git not to include commits that have been merged in already

$ git log testing ^master --no-merges

git branch -d -f testing

Here, -f is short for --force and, as the name implies, it overrides the warning message and executes the operation. -D is an alias for *-d -f*.

Renaming a branch

$ git branch -m <current name> <new name>*.*

$ git branch -m –f <current name> <new name>*.*

Here, -f is short for --force and, as the name implies, it overrides the warning message and executes the operation. -M is an alias for -m -f.

Internally, in its .git structure, Git is storing the SHA1 values under refs/heads/<name> for branches and under refs/tags/<name> for tags.

Checking Out Individual Files

$ git checkout <reference | SHA1> <filename>*.*

### Chapter 9 Merging content

Cherry-Picking

*git cherry-pick b942f21*

Specifying a Range of Commits

*git cherry-pick -xtheirs b942f21..d9e8b2c*

The “-xtheirs” option here specifies a resolution strategy to solve merge issues

As it turns out, when you specify a range with cherry-pick, the range is interpreted as “everything *after* the starting value and up to and including the ending value.” In order to actually include the starting value, you have to tell Git to use the commit that’s one before that one.

$ git cherry-pick -Xtheirs b942f21^..d9e8b2c

### Undoing merge operations

$ git reset --hard <SHA1 value that was current before the merge operation>

ORIG\_HEAD

I’ve talked before about the special pointer HEAD that Git maintains to point to the current branch and current commit. Git stores the corresponding reference for whatever HEAD points to as data in the file .git/HEAD. This is generally a reference to another reference for the branch, such as refs/heads/master. If you then look at .git/refs/heads/master, you can actually get the designated SHA1 value for the commit that HEAD (ultimately) references.

When a merge operation happens in Git, Git also saves off another reference named ORIG\_HEAD.This value is stored in .git/ORIG\_HEAD.

The command is git reset --merge ORIG\_HEAD.

It is not strictly required, but you will also use a different option to reset instead of --hard. That option is *--merge*. The difference between --hard and --merge is that --hard discards all changes in the working directory, while --merge does not discard changes that haven’t been staged. So, if you have other changes you’ve started making since the merge, using the --merge option allows you to keep those changes.

Reflog

While there is only one current value for any of these references, a *reflog* for each reference records the values as they change over time

$ git reflog

Aborting the merging operation

$ git merge --abort

$ git rebase --abort

$ git cherry-pick --abort

*Besides the conflict markers in the local file, there are other ways to look at the diffs when you have a merge conflict. One way is to use the command*

gitlog --merge -p <path>*.*

$ git log --merge –p file1.txt

*You can also use the* git show *command to see the different versions. If you pass* :#:filepath *as the arguments, then you can see the various versions involved in the merge. Using 1 for the # in the expression shows the common ancestor; using 2 for the # shows the tip of the current branch; and using 3 for the # shows the result after the merge was attempted.*

$ git show :1:file1.txt

master

$ git show :2:file1.txt

still master

$ git show :3:file1.txt

Feature

Resolution Options and Strategies

because recursive is the default strategy, you can just pass the -X<option> without including -s to specify the strategy.

➤➤ **Ours:** This option tells Git that when a file has been modified on both branches, resulting in a conflict, it must use the version from the current (destination) branch as the resolution.

➤➤ **Theirs:** This option tells Git that when a file has been modified on both branches, resulting in a conflict, it must use the version from the source branch as the resolution

*$ git cherry-pick -Xtheirs d9e8b2c*.

Advanced Rebasing Scenario

The advanced syntax looks like this:

$ git rebase --onto newbase branch2 [ branch1]

The way you interpret this is *Rebase branch1 off of newbase, but exclude any commits that branch1 and branch2 have in common*

So, if you issue the command for the advanced rebase as *git rebase --onto master feature topic*, then this tells Git to rebase the parts of *topic* that are not shared with *feature* onto *master*.

You can also see the set of commits that are in *topic* but not in *feature* by using one of the advanced forms of the log command.

$ git log --oneline topic ^feature

### Interactive Rebasing:

$ git rebase –i <start commit>

$ git rebase –i HEAD~3

### Chapter 11 Doing more with git

After creating the stash and saving the uncommitted content, Git is basically doing a

$ git reset --hard HEAD operation

When stashing content with Git, by default, it ignores untracked files. In order for Git to stash untracked files, it is necessary to include the *-u* (*--include-untracked*) option.

you can look at what you have in the queue.

$ git stash list

*More details*

$ git stash list –oneline

$ git stash show <stash>

$ git stash show –p <stash>

Restore changing

$ git stash apply stash@{1}

$ git stash pop stash@{2}

Renaming Content

$ git mv file1 file2

Delete Content

*git rm*

If you want to override Git and remove the content, you can use the -f option to force the removal. If

you have a staged version and want to just remove it from the staging area, you can use the --cached

option. (Recall from my discussion on git diff that the term *cache* is another [historical] name

for the staging area.) Here again, you would do a git commit to finalize this change in the local

repository.

COMMANDS FOR SEARCHING

$ git grep database -- \*.java

These options simply tells the command to search for all instances of the expression (*database*) in files with the .java extension. Notice the use of the double hyphen (--) separator here. As with other Git commands, the double hyphen separates the command from path-limiting options. The part before the double hyphen is the actual command, and the part after the double hyphen is the selected set for the command to operate against.

To see line numbers when running the command, you normally need to pass the -n option. However, if you want this to always be the default option, you can configure the git config setting of git.lineNumber to true. So, both of the following operations would result in the output including line numbers:

$ git grep -n database -- \*.java

or

$ git config grep.lineNumber true

To make the output easier to read, you can use the --break option, which prints a separator line

between matches from different files, and the --heading option, which prints the filename as a

header above the matches for a file instead of on each line.

$ git grep -n --break --heading database -- \*.java

You can also use Boolean operators to specify how to grep for multiple instances

When you do use Boolean operators, you need to meet a couple of requirements:

➤➤ The expressions must be surrounded with quotes.

➤➤ The expressions must have the -e option in front of them.

$ git grep -e 'database' --and -e 'access' -- \*.java

The or option is just --or,

you can tell Git to grep in the HEAD revision or in one further back.

$ git grep database HEAD -- \*.java

$ git grep database b2e575a -- \*.java

There is also an option to tell Git to search in the index (staging area), --cached.

$ git grep database b2e575a –cached -- \*.java

Git Log Searches

The first option is the -S option. By default, this option takes a string (not a regular expression) and searches the commit history for commits that add or delete that string

$ git log --oneline -S "line 5"

If you want to use the pickaxe option but supply a regular expression, you can add the –pickaxeregex option. As the name suggests, this allows you to pass a regular expression for the argument to the pickaxe (-S) option.

$ git log --oneline --pickaxe-regex -S "line [1-3]"

A similar option that Git provides with the log command is -G.

Clean

Clean up untracked files from your working directory

$ git clean –f

Remove untracked subdirectories

$ git clean -d

The clean command provides two options for working with the .gitignore file: *-x* and *-X*. Both of these options tell clean to also clean the items in the .gitignore file.

The difference between *-x* and *-X* is that *-x* allows for cleaning out everything, including what’s ignored, and *-X* only allows for cleaning out what’s ignored.

Gc

*gc* stands for *garbage collection* and, unlike the clean command, which cleans up files in your working directory, gc cleans up internal content in the repository. It has two main functions:

➤➤ To compress revisions of files to save space and increase performance

➤➤ To remove objects that are no longer reachable by any connection in Git

Notes

At some point after making a commit, you may decide that there are additional comments or other non-code information that you’d like to add with the commit. If this is the most recent commit, you can use the git amend functionality and update the commit message.

$ git notes add -m "This is an example of a note" 2f2ea1e

If you want to look at a specific note, you can use the show subcommand.

$ git notes show 2f2ea1e

This is an example of a note

With an additional option, you can create notes in a custom namespace. For example, if you want to create a notes namespace for reviews of previous commits, you can supply that namespace to the --ref option.

$ git notes --ref=review add -m "Looks ok to me" f3b05f90

The simplest way to see notes is to use git log. By default, this command shows notes in the default namespace. To see other namespaces, you can use the --show-notes=<namespace> or --show-notes=\* option to see all of them.

Quickly finding where a problem a problem or change was introduced

$ git bisect

Chapter 12:

$ git remote –v

$ git remote add version2 https://

$ git remote rename version2 origin2

$ git remote rm version2

Clone options

The –bare option tells Git to create a bare repository

Viewing information about remote branches

You can easily see the remote tracking branches after a clone by using the –r option

$ git branch –r

IF you want to see the local branches as well as the remote ones you can use the –a option

$ git branch –av (-v verbose)

There is one more variation you can use with the branch command to find out additional information: the -vv flag. (Note that this is two *v*’s side by side, not one *w*.) The two *v*’s tell Git to show extra information that is very useful—namely, any tracking connections between local branches and the remote tracking (upstream) branches

$ git branch –vv

One other way to get the list of branches indirectly is to use the show option for your remote.

$ git remote show origin

Finally, if you take a look at the local config file, you can see where this configuration information is actually stored.

$ cat .git/config

Automatic Mapping

For other remote tracking branches, if you attempt to start working with a local branch with the same name, Git may establish the tracking relationship for you

Manual Mapping

$ git branch test origin2/test

The same result occurs if you explicitly include the --track option.

$ git branch --track test origin2/test

There is also an option named --set-upstream, which is the same as --track in most cases. However, this option is deprecated in Git and will be unsupported at some point, so it is best not to use it.

The other available option is the newer *--set-upstream-to* option. This option performs a similar function to *--track* and *--set-upstream* but is clearer.

$ git branch test

$ git branch -vv | grep test

$ git branch --set-upstream-to=origin/test test

This option can be abbreviated as *-u*.

$ git branch test

$ git branch -vv | grep test

$ git branch --set-upstream-to=origin/test test

*If you are creating a new project or you want to share an existing project with others, you may want to create a new remote repository from your existing code base in your local repository.*

1. In the remote area for repositories, make a directory for the repository, change into that directory, and run the following command:

$ git init --bare --shared<options>

The --shared option isn’t required, but it does allow you to configure permissions and access at creation time. See the git init --help page for an explanation of --shared and its possible values.

2. Back in your local environment, add the URL (path) for the new remote area you created as origin (or whatever remote reference name you’d like) to your repository.

$ git remote add origin http://mygitserver.com/myrepopath

3. Now you have a connection to an empty remote. You can use the push operation to push your content over to the remote (which I’ll talk about shortly) using commands like this:

$ git push origin master

# Push

So, in most cases, you would use this form:

$ git push <remote repository> <remote branch>

You can, however, also use this form if you are pushing from a local branch that is named differently from the targeted remote branch:

$ git push <remote repository> <local branch>:<remote branch>

The all option pushes all branches.

$ git push –all

The delete option deletes the reference on the remote repository. For example, the following command deletes a remote branch:

$ git push --delete origin testing

By default, push does not push tags over to the remote side. The option *--tags* tells Git to push tags.

The mirror option tells Git to push everything (tags, remotes, heads, and so on) over. You typically use the --mirror option if you need to migrate or move an entire repository somewhere.

### Chapter 13 Understanding remotes

Setup a local feature branch to track the remote feature branc

$ git branch features origin/features

Abort a merge

$ git merge --abort

$ git rebase –-abort

$ git rebase –Xours origin/master

$ git rebase --onto newbase branch2 [ branch1]

The way you interpret this is *Rebase branch1 off of newbase, but exclude any commits that branch1 and branch2 have in common*.

So, if you issue the command for the advanced rebase as *git rebase --onto master feature topic*, then this tells Git to rebase the parts of *topic* that are not shared with *feature* onto *master*

# Chapter 14 Working with trees and modules in Git

Worktree

$ git worktree add <path> <branch>

$ git worktree add --force <path> <branch>

Create a new worktree with a different branch name based on a existing branch

$ git worktree add -b docs2 ../tmparea4 docs

The -B option allows you to force having a new branch with the same name as an

existing branch.

$ git worktree add -B docs2 ../tmparea4 docs

What happens if you don’t supply a branch name to create? The worktree command creates a new branch with the same name as the target area and based on whatever branch is current in the main working tree.

$ git worktree add ../tmparea5

*Git stores the information about working trees in the Git directory area. Assuming the Git directory maps to .git, the working tree information is stored in .git/worktrees/<*name of worktree*>.*

Listing Out the Working Trees

$ git worktree list

There is only one option for list: *porcelain*. This option lists the worktree information in a more verbose format

$ git worktree list –porcelain

Pruning a Worktree

As its name implies, the prune subcommand removes worktree information. However, it only removes the information from the Git directory (.git) *after* the actual worktree subdirectory has been manually removed. Here is an example from the main worktree:

$ rm -rf ../tmparea6

$ git worktree prune

-n (--dry-run)—This option tells Git to not execute, but to just explain what it would do.

$ git worktree prune –n

-v (--verbose)—This option tells Git to be more verbose in explaining what it’s doing.

$ git worktree prune –v

SUBMODULES

Git creates and manages a .gitmodules file at the root of the repository

$ cat .gitmodules

[submodule "BVUtils"]

path = BVUtils

url = ../bvutils

branch = master

Add a submodule

$ git submodule add <Remote path> name

$ git submodule status

The output also includes a simple prefix character, defined as:

“-“ if the submodule is not initialized

“+” if the submodule’s current version that’s checked out is different from the SHA1 in the containing repository

“U” if there are merge conflicts in the submodule

*Internally, Git stores module information for submodules in a directory named .git/modules. Inside this area, there is a separate subdirectory for each of the modules attached to this project*

Cloning with submodules

$ git clone <url>

$ git submodule init

$ git submodule update

There is a shortcut

$ git submodule update --init

Even more helpful git provides an option –recursive to the clone command

$ git clone --recursive <url>

Processing multiple submodules

$ git submodule foreach git log --oneline

Git also provides several variables populated with information that you can use when constructing commands

$name: the name of the submodule

$path: the path of the submodule relative to the superproject

$sha1: the current SHA1 value as recorded in the superproject

$toplevel: the absolute path to the superproject

$ git submodule –-quiet foreach ‘echo $path $sha1’

Updating submodules from their remotes. Multiple approaches

1. 1ere approach

$ git pull --recurse-submodules

However this operation does not check out updated references in your submodules. Your submodules are still registering the previous commits as current.

To get the latest commits registered and finish the update, you can just run the submodule command to check out the updates references from the submodule

$ git submodule update

1. 2approach

$ git submodule update --remote

3 approach

$ git submodule foreach git pull origin master

*The default branch for a submodule is assumed to be master. If you need to change that, you can do it through a simple git config command, such as the following:*

$ git config -f .gitmodules submodule.mod2.branch testbranch

*Here, the* -f *option is simply pointing to a different file—the .gitmodules file—and setting the value for the key triple submodule->mod2->branch. Afterward,your .gitmodules file looks like this:*

Viewing Submodule Differences

$ git submodule status

$ git diff --submodule

If you run this command (whitout –remote option), this tells Git to update the submodules to the references (SHA1 values) that are current in the superproject.

$ git submodule update

Pushing changes from submodules

*The check option argument tells the push command to verify that in each submodule where code has been committed, the commit has also been pushed to at least one remote associated wit the submodule*

$ git push --recurse-submodules=check

*The on-demand argument tells the push command to try pushing any commits that need to be pushed for the submodules at that point*

$ git push --recurse-submodules=on-demand

Unregistering a submodule

$ git submodule deinit

TIP *Here is a summary of the basic rules for dealing with submodules and superprojects:*

If you update something in a submodule, follow these steps:

1. *In the submodule directory, commit and push it out to the submodule’s remote.*

2. *Go back to the superproject. The superproject should show that that particularsubmodule area has changed—almost like a file in the repository with the submodule name.*

3. *Stage and commit that changed area (submodule name) in the superproject to ensure that the superproject points to the updated commit in the submodule.*

4. *Push out that change in the superproject to the superproject’s remote. This ensures that anyone cloning or pulling the superproject gets a version that points to the latest updates in the submodules.*

If you pull an update of the superproject, follow these steps:

1. *Ensure that you have also pulled the latest versions of the submodules (using the recurse-submodules option or foreach subcommand, or by pulling each area).*

2. *In the superproject, run the submodule update to check out the commit in the submodule that corresponds to the submodule references in the superproject.*

SUBTREES

Another kind of functionality is available in Git that provides a similar working model without the worry of trying to keep things synchronized: subtrees

When you add a subtree, by default, all of the project’s history is also added in the subdirectory. To avoid adding all of the history, you can use a squash option. This squash option is similar to the squash option you used in the interactive rebasing functionality. It compresses the history for the project that is being added into one commit.

~/subtrees/local/myproject$ git remote add sub\_origin ~/subtrees/remotes/subproj.git

~/subtrees/local/myproject$ git subtree add --prefix subproject --squash sub\_origin master

If you later need to pull some changes into your subtree, you can use a similar version of the subtree command with pull.

$ git subtree pull --prefix subproject sub\_origin master –squash

$ git --version

Professional git

The general form of commands is as follows:

$ git <git-options> <command> <command-options> <operands>

Undoing a configuration setting

$ git config --unset --global user.name

$ git config --list

$ git config --global –list

One short configuration

$ git -c <configuration setting>=<value> <rest of command line>

*Aliases*

git config <scope option> alias.<name> <command string>.

To seee the stored objects:

$ find .git/objects -type f

After

$ git cat-file –p 45dddf display its contents

$ git cat-file –t 45dddf returns the type of an object

*One shot configuration*