GIT and GITHub Essentials

Distributed revision control and source code management (SCM)

Course Goals and Non Goals

- Course Goals
  - Learn what it Git and Github and how to use it for distributed revision control and source code management (SCM) system.

- Course Non Goals
  - Integration with any other tools is not part of this course

Intended Audience

- Developers, Test Engineers, Designers

Day Wise Schedule

- Duration – 8 Hrs

  Introduction to GIT

  Repositories and Branches

  Working with GIT Repositories

  Working with GIT Branches, merging branches and tags

  Git Stash

  .gitignore

  Inspecting a repository – status and log command

  Viewing old commits.

  Undoing changes.

  Rewriting history.

  Git hooks

  Merging vs rebasing

  Pull requests

References

- Pragmatic version control using GIT
- Vogella.com
- Pro-Git book

GIT and GitHub Essentials

Introduction to GIT

Overview of Git

- Git is a distributed revision control and source code management (SCM) system with an emphasis on speed, data integrity and support for distributed, non-linear workflows.
  - Git was initially designed and developed by Linus Torvalds for Linux kernel development in 2005, and has since become the most widely adopted version control system for software development.
- As with most other distributed revision control systems, and unlike most client–server systems, every Git working directory is a full-fledged repository with complete history and full version-tracking capabilities, independent of network access or a central server. Like the Linux kernel, Git is free software distributed under the terms of the GNU General Public License version 2.

Background

- Torvalds wanted a distributed system that he could use like BitKeeper, but none of the available free systems met his needs, particularly in terms of performance.
- Torvalds took an example of an SCM system requiring thirty seconds to apply a patch and update all associated metadata, and noted that this would not scale to the needs of Linux kernel development, where syncing with fellow maintainers could require 250 such actions at a time. He wanted patching to take three seconds,and had several other design criteria in mind:
  - take Concurrent Versions System (CVS) as an example of what *not* to do; if in doubt, make the exact opposite decision
  - support a distributed, BitKeeper-like workflow
  - very strong safeguards against corruption, either accidental or malicious.

Design Base

- Git's design was inspired by BitKeeper and Monotone.
- Git was originally designed as a low-level version control system engine on top of which others could write front ends, such as Cogito .
- The core Git project has since become a complete version control system that is usable directly.
- While strongly influenced by BitKeeper, Torvalds deliberately attempted to avoid conventional approaches, leading to a unique design.

Characteristics

- Strong support for non-linear development
  - Git supports rapid branching and merging, and includes specific tools for visualizing and navigating a non-linear development history. A core assumption in Git is that a change will be merged more often than it is written, as it is passed around various reviewers.
  - Branches in git are very lightweight: A branch in git is only a reference to a single commit. With its parental commits, the full branch structure can be constructed.
- Distributed development
  - Git gives each developer a local copy of the entire development history, and changes are copied from one such repository to another. These changes are imported as additional development branches, and can be merged in the same way as a locally developed branch.
- Compatibility with existing systems/protocols
  - Repositories can be published via HTTP, FTP, rsync, or a Git protocol over either a plain socket, or ssh. Git also has a CVS server emulation, which enables the use of existing CVS clients and IDE plugins to access Git repositories..

Characteristics… continued

- Efficient handling of large projects
  - Torvalds has described Git as being very fast and scalable and performance tests done by Mozilla showed it was an order of magnitude faster than some version-control systems, and fetching version history from a locally stored repository can be one hundred times faster than fetching it from the remote server
- Cryptographic authentication of history
  - The Git history is stored in such a way that the ID of a particular version (a *commit* in Git terms) depends upon the complete development history leading up to that commit. Once it is published, it is not possible to change the old versions without it being noticed.
- Toolkit-based design
  - Git was designed as a set of programs written in C, and a number of shell scripts that provide wrappers around those programs. Although most of those scripts have since been rewritten in C for speed and portability, the design remains, and it is easy to chain the components together.

Characteristics… continued

- Pluggable merge strategies
  - As part of its toolkit design, Git has a well-defined model of an incomplete merge, and it has multiple algorithms for completing it, culminating in telling the user that it is unable to complete the merge automatically and that manual editing is required.
- Garbage accumulates unless collected
  - Aborting operations or backing out changes will leave useless dangling objects in the database. These are generally a small fraction of the continuously growing history of wanted objects. Git will automatically perform garbage collection when enough loose objects have been created in the repository. Garbage collection can be called explicitly using git gc --prun.
- Periodic explicit object packing
  - Git stores each newly created object as a separate file. Although individually compressed, this takes a great deal of space and is inefficient. This is solved by the use of *packs* that store a large number of objects in a single file (or network byte stream) called*packfile*, delta-compressed among themselves.

GIT server

- As git is a distributed version control system, it can be used as server out of the box. Dedicated git server software helps, amongst other features, to add access control, display the contents of a git repository via web, and help managing multiple repositories.

GIT server … continued

- Remote file store and shell access
  - A git repository can be cloned to a shared file system, and accessed by other persons. It can also be accessed via remote shell just by having the git software installed and allowing a user to log in.
- Git daemon, instaweb
  - Git daemon allows users to share their own repository to colleagues quickly. Git instaweb allows users to provide web view to the repository. As of 2014-04 instaweb does not work on Windows. Both can be seen in the line of Mercurial's "hg serve".
- Gitolite
  - Gitolite is an access control layer on top of git, providing fine access control to git repositories. It relies on other software to remotely view the repositories on the server.
- Gerrit
  - Gerrit provides two out of three functionalities: access control, and managing repositories. It uses jGit. To view repositories it is combined e.g. with Gitiles or GitBlit.
- Gitblit
  - Gitblit can provide all three functions, but is in larger installations used as repository browser installed with gerrit for access control and management of repositories.
- Gitiles
  - Gitiles is a simple repository browser, usually used together with gerrit.
- Bonobo Git Server
  - Bonobo Git Server is a simple git server for Windows implemented as an ASP.NET gateway.It relies on the authentication mechanisms provided by Windows Internet Information Services, thus it does not support SSH access but can be easily integrated with Active Directory.
- Commercial solutions
  - Commercial solutions are also available to be installed on premises, amongst them GitHub Software (using native git, available as a vm),Stash (using jGit), Team Foundation Server (using libgit2).

What is GitHub

- GitHub (originally known as Logical Awesome LLC) is a web-based hosting service for version control using git. (https://github.com/)
- It offers all of the distributed version control and source code management (SCM) functionality of Git as well as adding its own features.
- It provides access control and several collaboration features such as bug tracking, feature requests, task management, and wikis for every project.
- GitHub offers plans for both private repositories and free accounts which are commonly used to host open-source software projects.
- As of April 2017, GitHub reports having almost 20 million users and 57 million repositories, making it the largest host of source code in the world.

Getting Started - Installing Git

- The most official build is available for download on the Git website.
- Just go to http://git-scm.com/download/win and the download will start automatically.
- Note that this is a project called Git for Windows, which is separate from Git itself; for more information on it, go to https://git-for-windows.github.io/.
- Another easy way to get Git installed is by installing GitHub for Windows. The installer includes a command line version of Git as well as the GUI. It also works well with Powershell, and sets up solid credential caching and sane CRLF settings. You can download this from the GitHub for Windows website, athttp://windows.github.com.

First-Time Git Setup

- Git comes with a tool called git config that lets you get and set configuration variables that control all aspects of how Git looks and operates.

- These variables can be stored in three different places:
  - /etc/gitconfig file: Contains values for every user on the system and all their repositories. If you pass the option --system to git config, it reads and writes from this file specifically.
  - ~/.gitconfig or ~/.config/git/config file: Specific to your user. You can make Git read and write to this file specifically by passing the --global option.
  - config file in the Git directory (that is, .git/config) of whatever repository you're currently using: Specific to that single repository.

- Each level overrides values in the previous level, so values in .git/config trump those in/etc/gitconfig.

- On Windows systems, Git looks for the .gitconfig file in the $HOME directory (C:\Users\$USER for most people). It also still looks for /etc/gitconfig. If you are using version 2.x or later of Git for Windows, there is also a system-level config file at C:\Documents and Settings\All Users\Application Data\Git\config .

First-Time Git Setup

- **Your Identity**
  - $ git config --global user.name “Anil"
  - $ git config --global user.email anil@example.com
- **Your Editor**
  - $ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -multiInst -nosession"
- **Checking Your Settings**
  - $ git config –list
  - You may see keys more than once, because Git reads the same key from different files (/etc/gitconfig and ~/.gitconfig, for example).
  - You can also check what Git thinks a specific key’s value is by typing git config <key>:
  - $ git config user.name

Getting Started - Getting Help

Getting Help

- If you ever need help while using Git, there are three ways to get the manual page (manpage) help for any of the Git commands:

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

- For example, you can get the manpage help for the config command by running

```
$ git help config
```

GIT

Repositories and branches

Repositories and Branches

- Repositories:
  - It is a collection of refs together with an object database containing all objects which are reachable from the refs, possibly accompanied by meta data from one or more porcelains. A repository can share an object database with other repositories via alternates mechanism.

Repositories and Branches

- What to store in repositories?
  - Anything, however any sort of editable files are preferred.

Git repositories

- How to get GIT repository:
  - $ git clone git://git.kernel.org/pub/scm/git/git.git

    It does approx. 225 MB download.

Git repositories

- Creating repositories :
  - at default location
    - git init
  - at particular location
    - git init c:/testGIT
  - Bare repository
    - git init --bare

Git Basics - Getting a Git Repository

- You can get a Git project using two main approaches.
  - The first takes an existing project or directory and imports it into Git.
  - The second clones an existing Git repository from another server.

Git Basics - Getting a Git Repository

- Initializing a Repository in an Existing Directory
  - If you're starting to track an existing project in Git, you need to go to the project's directory and type:

    ```
    $ git init
    ```

  - This creates a new subdirectory named .git that contains all of your necessary repository files – a Git repository skeleton.
- You can accomplish start version-controlling existing files that with a few git add commands that specify the files you want to track, followed by a git commit:

  ```
  $ git add *.c
  $ git add LICENSE
  $ git commit -m 'initial project version'
  ```

Git Basics - Getting a Git Repository

- Cloning an Existing Repository
  - If you want to get a copy of an existing Git repository – for example, a project you'd like to contribute to – the command you need is git clone [url] .

    ```
    $ git clone https://github.com/libgit2/libgit2
    ```

  - This creates a directory named "libgit2", initializes a .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version. If you go into the new libgit2 directory, you'll see the project files in there, ready to be worked on or used.

Recording Changes to the Repository

- Checking the Status of Your Files

```
$ git status
```

- Let’s say you add a new file to your project, a simple README file. If the file didn’t exist before, and you run git status, you see your untracked file like so:

```
$ echo 'My Project' > README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)
  README
```

Recording Changes to the Repository

- Tracking New Files
  - In order to begin tracking a new file, you use the command git add. To begin tracking the README file, you can run this:

```
$ git add README
```

  - If you run your status command again, you can see that your README file is now tracked and staged to be committed:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)

 new file:   README
```

Recording Changes to the Repository

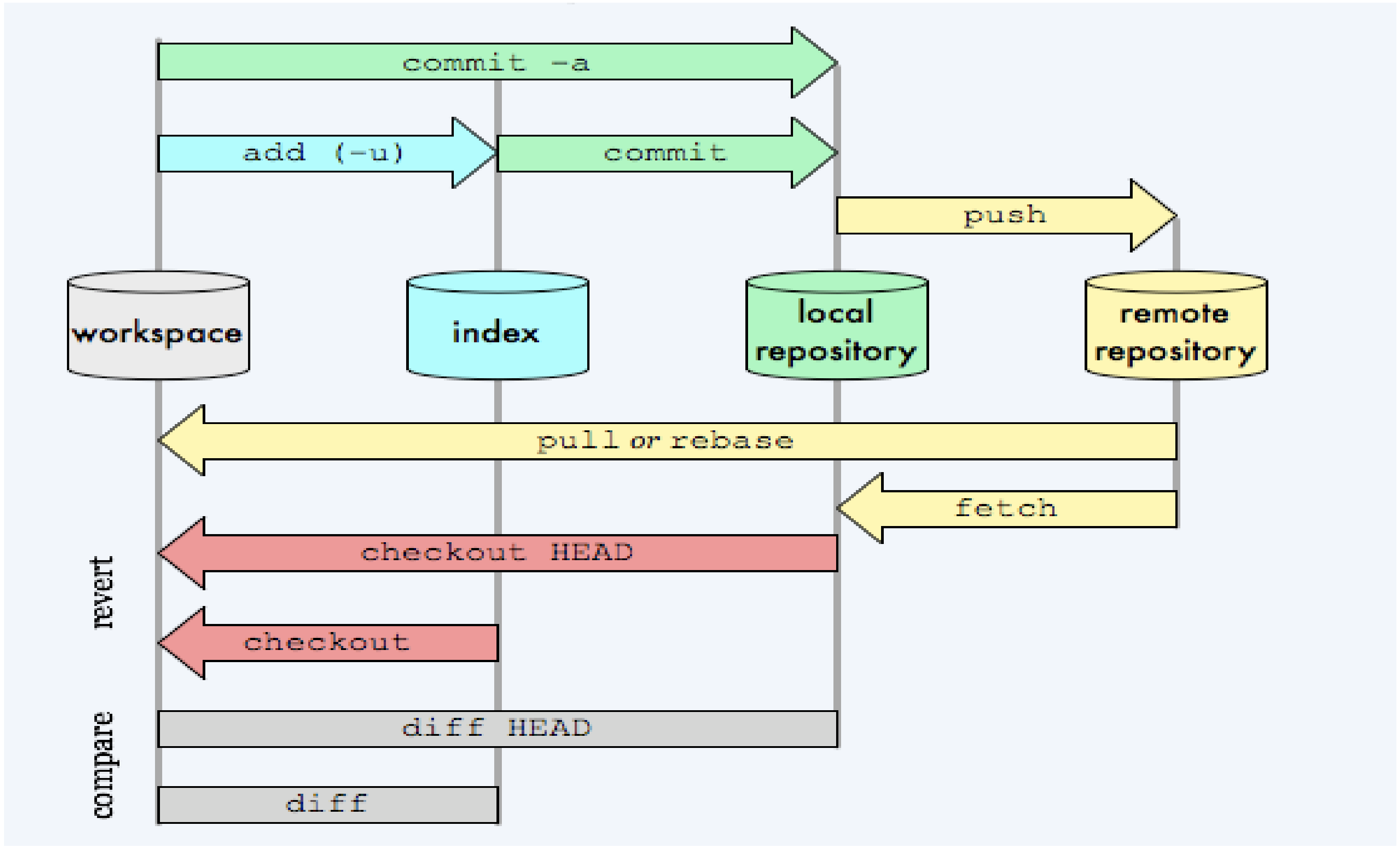
- Staging Modified Files
  - Let’s change a file that was already tracked. If you change a previously tracked file (say CONTRIBUTING.md) and then run your git status command again,

```
$ git status
```

  - The CONTRIBUTING.md file appears under a section named “Changes not staged for commit” – which means that a file that is tracked has been modified in the working directory but not yet staged. To stage it, run the git add command.

```
$ git add CONTRIBUTING.md
$ git status
```

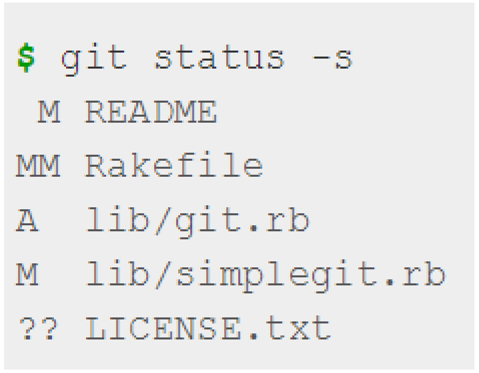
Typical Workflow



Recording Changes to the Repository

➢Short Status

- Git also has a short status flag so you can see your changes in a more compact way. If you run git status -s or git status --short you get a far more simplified output from the command:



```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

- New files that aren't tracked have a ?? next to them, new files that have been added to the staging area have an A, modified files have an M and so on.
- There are two columns to the output - the left-hand column indicates the status of the staging area and the right-hand column indicates the status of the working tree.
- So for example in that output, the README file is modified in the working directory but not yet staged, while the lib/simplegit.rb file is modified and staged. The Rakefile was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

Recording Changes to the Repository

- **Ignoring Files**
- Often, you'll have a class of files that you don't want Git to automatically add or even show you as being untracked. These are generally automatically generated files such as log files or files produced by your build system.
- In such cases, you can create a file listing patterns to match them named .gitignore. Here is an example .gitignore file:

```
$ cat .gitignore
*.[oa]
*~
```

- The first line tells Git to ignore any files ending in ".o" or ".a" – object and archive files that may be the product of building your code. The second line tells Git to ignore all files whose names end with a tilde (~).
- You may also include a log, tmp, or pid directory; automatically generated documentation; and so on.
- Setting up a .gitignore file is a good idea so you don't accidentally commit files that you really don't want in your Git repository.
- The rules for the patterns you can put in the .gitignore file are as follows:
  - Blank lines or lines starting with # are ignored.
  - Standard glob patterns work.
  - You can start patterns with a forward slash (/) to avoid recursivity.
  - You can end patterns with a forward slash (/) to specify a directory.
  - You can negate a pattern by starting it with an exclamation point (!).

Recording Changes to the Repository

➢Viewing Your Staged and Unstaged Changes

- To see what you’ve changed but not yet staged, type git diff with no other arguments:

```
$ git diff
```

- This command compares what is in your working directory with what is in your staging area. The result tells you the changes you’ve made that you haven’t yet staged.

- If you want to see what you’ve staged that will go into your next commit, you can use git diff --staged. This command compares your staged changes to your last commit:

```
$ git diff --staged
```

- (--staged and --cached are synonyms)

Recording Changes to the Repository

- Committing Your Changes
  - Now that your staging area is set up the way you want it, you can commit your changes.
  - Remember that anything that is still unstaged – any files you have created or modified that you haven't run git add on since you edited them – won't go into this commit. They will stay as modified files on your disk. In this case, let's say that the last time you ran git status, you saw that everything was staged, so you're ready to commit your changes.
  - The simplest way to commit is to type git commit:

    ```
    $ git commit
    ```
  - Doing so launches your editor of choice.
- Alternatively, you can type your commit message inline with the commit command by specifying it after a -m flag, like this:

```
$ git commit -m "Story 182: Fix benchmarks for speed”
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

Recording Changes to the Repository

- Adding the -a option to the git commit command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the git add part:

```
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

Recording Changes to the Repository

➢Removing Files

- To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The git rm command does that, and also removes the file from your working directory so you don't see it as an untracked file the next time around.

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    PROJECTS.md
```

- The next time you commit, the file will be gone and no longer tracked. If you modified the file and added it to the index already, you must force the removal with the -f option. This is a safety feature to prevent accidental removal of data that hasn't yet been recorded in a snapshot and that can't be recovered from Git.

Recording Changes to the Repository

- If you want to keep the file in your working tree but remove it from your staging area. In other words, you may want to keep the file on your hard drive but not have Git track it anymore.
- This is particularly useful if you forgot to add something to your .gitignore file and accidentally staged it, like a large log file or a bunch of .a compiled files.
- To do this, use the --cached option:

```
$ git rm --cached README
```

- You can pass files, directories, and file-glob patterns to the git rm command. That means you can do things such as:

```
$ git rm log/\*.log
```

- This command removes all files that have the .log extension in the log/ directory.

```
$ git rm \*~
```

- This command removes all files whose names end with a ~.

Recording Changes to the Repository

- Moving Files

```
$ git mv file_from file_to
```

- Unlike many other VCS systems, Git doesn’t explicitly track file movement.
- Example

```
$ git mv README.md README
```

- And is equivalent to running something like this:

```
$ mv README.md README
$ git rm README.md
$ git add README
```

- Git figures out that it’s a rename implicitly, so it doesn’t matter if you rename a file that way or with the mv command.
- The only real difference is that mv is one command instead of three – it’s a convenience function.

Viewing the Commit History

- After you have created several commits, or if you have cloned a repository with an existing commit history, you'll probably want to look back to see what has happened.
- The most basic and powerful tool to do this is the git log command.

```
$ git log
$ git log -p -2
$ git log --stat
$ git log --pretty=oneline
$ git log --pretty=format:"%h - %an, %ar : %s”
```

- Useful options for git log --pretty=format
  - %H : Commit hash , %h : Abbreviated commit hash, %T : Tree hash, %t :Abbreviated tree hash,%P : Parent hashes, %p : Abbreviated parent hashes, %an : Author name, %ae: Author email, %ad: Author date (format respects the --date=option), %ar: Author date, relative, %cn : Committer name, %ce : Committer email, %cd : Committer date, %cr: Committer date, relative, %s:Sub

Undoing Things

- One of the common undos takes place when you commit too early and possibly forget to add some files, or you mess up your commit message. If you want to try that commit again, you can run commit with the --amend option:

  $ git commit –amend

- This command takes your staging area and uses it for the commit. If you've made no changes since your last commit (for instance, you run this command immediately after your previous commit), then your snapshot will look exactly the same, and all you'll change is your commit message.
- if you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:

  $ git commit -m 'initial commit'

  $ git add forgotten_file

  $ git commit --amend

- You end up with a single commit – the second commit replaces the results of the first.

Unstaging a Staged File and Unmodifying a Modified File

- Unstaging a Staged File
  - use git reset HEAD <file>... to unstage
- Unmodifying a Modified File
  - use "git checkout -- <file>..." to discard changes in working directory
  - It’s important to understand that git checkout -- <file> is a dangerous command. Any changes you made to that file are gone – Git just copied another file over it. (use branching - to get it in different branch)

Working with Remotes

➢Showing Your Remotes

- To see which remote servers you have configured, you can run the git remote command.
- It lists the shortnames of each remote handle you've specified. If you've cloned your repository, you should at least see origin – that is the default name Git gives to the server you cloned from.

- You can also specify -v, which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote.

➢Adding Remote Repositories

```
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
```

Working with Remotes

➢ Pushing to Your Remotes

- If you want to push your master branch to your origin server (again, cloning generally sets up both of those names for you automatically), then you can run this to push any commits you've done back up to the server:

```
$ git push origin master
```

- This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You'll have to fetch their work first and incorporate it into yours before you'll be allowed to push.

Working with Remotes

- Inspecting a Remote
  - If you want to see more information about a particular remote, you can use the git remote show [remote-name] command. If you run this command with a particular shortname, such as origin, you get something like this:

```
$ git remote show origin
```

- Removing and Renaming Remotes
  - You can run git remote rename to change a remote's shortname. For instance, if you want to rename pb to paul, you can do so with git remote rename:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

Git Pull

- git pull pull down from a remote whatever you ask (so, whatever trunk you're asking for) and instantly merge it into the branch you're in when you make the request.
- Pull is a high-level request that runs 'fetch' then a 'merge' by default, or a rebase with '–rebase'. You could do without it, it's just a convenience.

```
%> git checkout localBranch
%> git pull origin master
%> git branch
master
* localBranch
```

- The above will merge the remote "master" branch into the local "localBranch".

Git fetch

- Fetch is similar to pull, except it won’t do any merging.

```
%> git checkout localBranch
 %> git fetch origin remoteBranch
%> git branch
master
* localBranch
remoteBranch
```

- So, the fetch will pull down the remoteBranch and put it into a local branch called “remoteBranch”.
- It creates a local copy of a remote branch which you shouldn’t manipulate directly; instead create a proper local branch and work on that.
- ‘git checkout’ has a confusing feature though. If you ‘checkout’ a local copy of a remote branch, it creates a local copy and sets up a merge to it by default.

Git clone

- Git clone will clone a repo int a newly created directory.

```
%> cd newfolder
%> git clone git@github.com:whatever/something.git
%> git branch
 * master
remoteBranch
```

- Git clone additionally creates a remote called ‘origin’ for the repo cloned from, sets up a local branch based on the remote’s active branch (generally master), and creates remote-tracking branches for all the branches in the repo

Repositories and Branches

- Branches:
  - A "branch" is an active line of development.
  - The most recent commit on a branch is referred to as the tip of that branch.
  - The tip of the branch is referenced by a branch head, which moves forward as additional development is done on the branch.
  - A single git repository can track an arbitrary number of branches, but your working tree is associated with just one of them (the "current" or "checked out" branch), and HEAD points to that branch

Repositories and Branches

- Getting different versions of project:
  - Git is best thought of as a tool for storing the history of a collection of files. It stores the history as a compressed collection of interrelated snapshots of the project's contents. In git each such version is called a commit.
  - Those snapshots aren't necessarily all arranged in a single line from oldest to newest; instead, work may simultaneously proceed along parallel lines of development, called branches, which may merge and diverge.
  - A single git repository can track development on multiple branches. It does this by keeping a list of heads which reference the latest commit on each branch; the git-branch(1) command shows you the list of branch heads:

  - $ git branch * master
  - A freshly cloned repository contains a single branch head, by default named "master", with the working directory initialized to the state of the project referred to by that branch head.

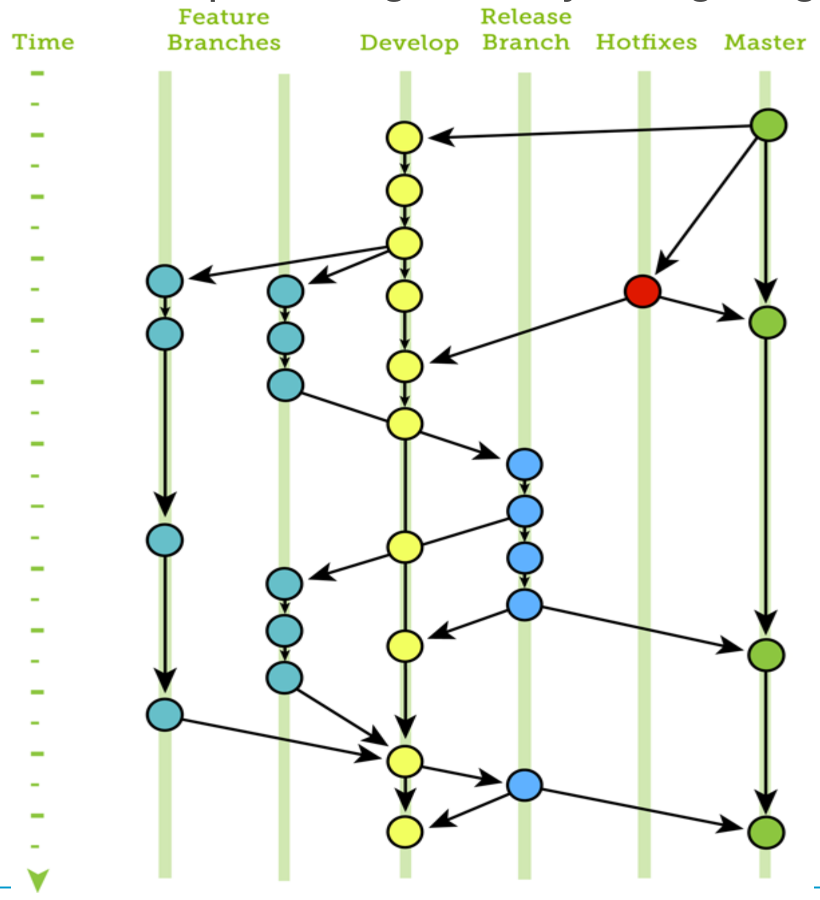
  - Most projects also use tags. Tags, like heads, are references into the project's history, and can be listed using thegit-tag(1) command:
  - $ git tag -l

Understanding History

- Commits:
  - Every change in the history of a project is represented by a commit. The git-show(1) command shows the most recent commit on the current branch:
  - $ git show
  - Every commit (except the very first commit in a project) also has a parent commit which shows what happened before this commit. Following the chain of parents will eventually take you back to the beginning of the project.
  - However, the commits do not form a simple list; git allows lines of development to diverge and then reconverge, and the point where two lines of development reconverge is called a "merge". The commit representing a merge can therefore have more than one parent, with each parent representing the most recent commit on one of the lines of development leading to that point.
  - The best way to see how this works is using the gitk(1)command; running gitk now on a git repository and looking for merge commits will help understand how the git organizes history.
  - In the following, we say that commit X is "reachable" from commit Y if commit X is an ancestor of commit Y. Equivalently, you could say that Y is a descendant of X, or that there is a chain of parents leading from commit Y to commit X.

Understanding History

- History Diagrams:
  - We will sometimes represent git history using diagrams like the one below:



Working with GIT

- *# switch to home* cd ~/ *# create a directory and switch into it*
- mkdir ~/repo01
- cd repo01 *# create a new directory* mkdir datafiles
- # Initialize the Git repository# for the current directory
- git init
- *# switch to your new repository*
- cd ~/repo01 *# create another directory # and create a few files*
- mkdir datafiles
- touch test01
- touch test02
- touch test03
- touch datafiles/data.txt
- *# Put a little text into the first file*
- ls >test01
- *# add all files to the index of the # Git repository*
- git add .

Working with GIT

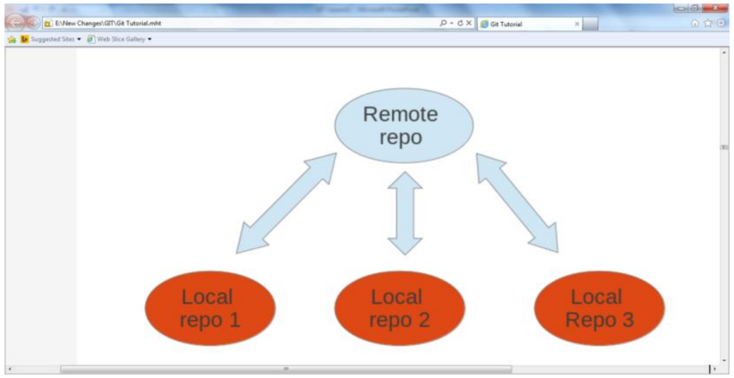
- *# commit your file to the local repository*
- git commit -m "Initial commit"
- *# show the Git log for the change*
- git log
- *# Create a file and commit it*
- touch nonsense2.txt git add . && git commit -m "more nonsense"
- *# remove the file via*
- *Git* git rm nonsense2.txt
- *# commit the removal*
- git commit -m "Removes nonsense2.txt file"
- *# Create a file and put it under version control*
- touch nonsense.txt git add . && git commit -m "a new file has been created"
- *# Remove the file*
- rm nonsense.txt
- *# Try standard way of committing -> will NOT work*
- git add . && git commit -m "a new file has been created"

Working with GIT

- *# commit the remove with the -a flag*
- git commit -a -m "File nonsense.txt is now removed"
- *# alternatively you could add deleted files to the staging index via*
- *# git add -A .*
- *# git commit -m "File nonsense.txt is now removed"*
- # create a file and add to index
- touch unwantedstaged.txt
- git add unwantedstaged.txt
- # remove it from the index
- git reset unwantedstaged.txt
- # to cleanup, delete it
- rm unwantedstaged.txt
- *# assume you have something to commit*
- git commit -m "message with a tpyo here"
- git commit --amend -m "More changes - now correct"

Working with GIT – Remote repositories

- Remote repositories are repositories that are hosted on the Internet or network. Such remote repositories can be use to synchronize the changes of several Git repositories. A local Git repository can be connected to multiple remote repositories and you can synchronize your local repository with them via Git operations.
- I is possible that users connect their individual repositories directly, but a typically Git workflow involve one or more remote repositories which are used to synchronize the individual repositories.



Working with GIT – Remote repositories

- # create a bare repository
- git init --bare
- *# switch to the first repository*
- cd ~/repo01
- *# create a new bare repository by cloning the first one*
- git clone --bare ../remote-repository.git
- *# check the content, it is identical to the .git directory in repo01*
- ls ~/remote-repository.git
- *# Add ../remote-repository.git with the name origin*
- git remote add origin ../remote-repository.git
- *# do some changes*
- echo "I added a remote repo" > test02
- *# commit*
- git commit -a -m "This is a test for the new remote origin"
- *# to push use the command:*
- *# git push [target]*
- *# default for [target] is origin*
- git push origin

Working with GIT – Remote repositories

- *# show the details of the remote repo called origin*
- git remote show origin
- # show the existing defined remote repositories
- git remote
- # show details about the remote repos
- git remote -v
- *# Switch to home* cd ~ *# Make new directory*
- mkdir repo02
- *# Switch to new directory* cd ~/repo02 *# Clone*
- git clone ../remote-repository.git .
- *# Make some changes in the first repository* cd ~/repo01
- *# Make some changes in the file*
- echo "Hello, hello. Turn your radio on" > test01
- echo "Bye, bye. Turn your radio off" > test02
- *# Commit the changes, -a will commit changes for modified files # but will not add automatically new files*
- git commit -a -m "Some changes"
- *# Push the changes*
- git push ../remote-repository.git

Working with GIT – Remote repositories

- *# switch to second directory*
- cd ~/repo02
- *# pull in the latest changes of your remote repository*
- git pull
- *# make changes*
- echo "A change" > test01
- *# commit the changes*
- git commit -a -m "A change"
- *# push changes to remote repository*
- *# origin is automatically created as we cloned original from this repository*
- git push origin
- # switch to the first repository and pull in the changes
- cd ~/repo01
- git pull ../remote-repository.git/
- # check the changes
- git status

GIT

Working with GIT Tags &
Branches

Tags

- Git has the option to tag a commit in the repository history so that you find them more easily at a later point in time. Most commonly, this is used to tag a certain version which has been released.

Tagging

- Git has the ability to tag specific points in history as being important. Typically people use this functionality to mark release points (v1.0, and so on).

- Listing Your Tags
  - Listing the available tags in Git is straightforward. Just type git tag:

```
$ git tag
```

  - This command lists the tags in alphabetical order; the order in which they appear has no real importance.

  - You can also search for tags with a particular pattern.

```
$ git tag -l "v1.8.5*“
```

Tagging

- Creating Tags
  - Git uses two main types of tags: lightweight and annotated.
  - A lightweight tag is very much like a branch that doesn’t change – it’s just a pointer to a specific commit.
  - Annotated tags, however, are stored as full objects in the Git database. They’re checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG).
- Annotated Tags
  - Creating an annotated tag in Git is simple. The easiest way is to specify -a when you run the tag command:

```
$ git tag -a v1.4 -m "my version 1.4"
$ git tag
```

- You can see the tag data along with the commit that was tagged by using the git show command:

```
$ git show v1.4
```

Tagging

- Lightweight Tags
  - Another way to tag commits is with a lightweight tag. This is basically the commit checksum stored in a file – no other information is kept. To create a lightweight tag, don't supply the -a, -s, or -m option:

```
$ git tag v1.4-lw
$ git tag
```

- This time, if you run git show on the tag, you don't see the extra tag information. The command just shows the commit.

- Tagging Later
  - You can also tag commits after you've moved past them.

```
$ git log --pretty=oneline

$ git tag -a v1.2 9fceb02

$ git tag
```

Tagging

- Sharing Tags
  - By default, the git push command doesn't transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them. This process is just like sharing remote branches – you can run git push origin [tagname].

```
$ git push origin v1.5
```

  - If you have a lot of tags that you want to push up at once, you can also use the --tags option to the git push command. This will transfer all of your tags to the remote server that are not already there.

```
$ git push origin --tags
```

- Checking out Tags
  - You can't really check out a tag in Git, since they can't be moved around. If you want to put a version of your repository in your working directory that looks like a specific tag, you can create a new branch at a specific tag with git checkout -b [branchname] [tagname]:

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

Working with tags

- Creating tags
  - You can create a new tag via the git tag command. Via the -m parameter, you specify the description of this tag. The following command tags the current active HEAD.
  - git tag version1.6 -m 'version 1.6' You can also create tags for a certain commit id.
  - git tag version1.5 -m 'version 1.5' [commit id]
- If you want to use the code associated with the tag, use:
  - git checkout <tag_name>
- Sharing tags
  - By default the git push command does not transfer tags to remote repositories. You explicitly have to push the tag with the following command.
  - git push origin [tagname]
- You can list the available tags via the following command:
  - git tag

Branches

- Git allows you to create *branches*, i.e. copies of the files from a certain commit. These branches can be changed independently from each other. The default branch is called *master*.
- Git allows you to create branches very fast and cheaply in terms of resource consumption. Developers are encouraged to use branches frequently.
- If you decide to work on a branch, you *checkout* this branch. This means that Git moves the *HEAD* pointer to the latest commit of the branch and populates the *working tree* with the content of this commit.
- Untracked files remain unchanged and are available in the new branch. This allows you to create a branch for unstaged and uncommited changes at any point in time.

Working with Branches

- Create new branch
  - You can create a new branch via the git branch [newname] command. This command allows optionally to specify the commit id, if not specified the currently checked out commit will be used to create the branch.
  - git branch testing
  - git checkout testing
  - echo "Cool new feature in this branch" > test01
  - git commit -a -m "new feature"
  - git checkout master
  - cat test01
- To create a branch and to switch to it at the same time you can use the git checkout command with the *-b* parameter.
  - git checkout -b bugreport12
  - git checkout -b mybranch master~1

Working with Branches

- Rename a branch
  - Renaming a branch can be done with the following command
  - git branch -m [old_name] [new_name]
- Delete a branch
  - To delete a branch which is not needed anymore, you can use the following command.
  - git branch -d testing
  - git branch
- Push a branch to remote repository
  - By default Git will only push matching branches to a remote repository. That means that you have to manually push a new branch once. Afterwards "git push" will also push the new branch.
  - git push origin testing
  - git checkout testing
  - echo "News for you" > test01
  - git commit -a -m "new feature in branch"
  - git push

Working with Branches

- Difference between branches
  - To see the difference between two branches you can use the following command.
  - git diff master your_branch
- Remote tracking branches
  - Your local Git repository contains references to the state of the branches on the remote repositories to which it is connected. These local references are called *remote tracking branches*.
  - You can see your *remote tracking branches* with the following command
  - git branch -r
  - To see all branches or only the local branches you can use the following commands
  - *# list all local branches*
  - git branch
  - *# list local and remote braches*
  - git branch -a
- Delete a remote branch
  - It is also save to delete a remote branch in your local Git repository. You can use the following command for that.
  - git branch -d -r origin/<remote branch>

Working with Branches

- You can create new *tracking branches* by specifying the *remote branch* during the creation of a branch. The following example demonstrates that.
- # setup a tracking branch called newbrach# which tracks origin/newbranch
- git checkout -b newbranch origin/newbranch

Working with Branches

- Updating your remote branches with git fetch
  - You can update your remote branches with the git fetch command.
  - The git fetch command updates your *remote branches*. The fetch command only updates the *remote branches* and none of the local branches and it does not change the working tree of the Git repository. Therefore you can run the git fetch command at any point in time.
  - After reviewing the changes in the remote tracking branch you can merge or rebase these changes onto your local branches. Alternatively you can also use the git cherry-pick "sha" command to take over only selected commits
- Compare remote tracking branch with local branch
  - The following code shows a few options how you can compare your branches
  - *# show the long entries between the last local commit and the # remote branch*
  - git log HEAD..origin
  - *# show the diff for each patch*
  - git log -p HEAD..origin *# show a single*
  - *diff* git diff HEAD...origin

Working with Branches

- Rebase your local branch based on the remote tracking branch
  - You can apply the changes of the *remote branches* on your current local branch for example with the following command
  - *# assume you want to rebase master based on the latest fetch # therefore check it out* git checkout master
  - *# update your remote tracking branch*
    git fetch
  - *# rebase your master onto origin/master*
  - git rebase origin/master

Working with Branches

- Fetch vs. pull
  - The git pull command performs a git fetch and git merge (or git rebase based on your Git settings). The git fetch does not perform any operations on your local branches. I can always run the fetch command and review the incoming changes.

Working with Branches

➢ Merging branches

- Git allows to combine the changes of two branches. This process is called *merging*.
- The git merge command performs a merge. If the commits which are merged are direct successors of the HEAD pointer of the currentbranch, Git simplifies things by performing a so-called *fast forward merge*. This *fast forward merge* simply moves the *HEAD* pointer of the current branch to the last commit which is being merged.
- This process is depicted in the following graphics. Assume you want to merge the changes of branch into your master branch. Each commits points to its successor.
- You can merge changes from one branch to the current active one via the following command.
- *# merges into your current selected branch*
- git merge testing

Working with Branches

- Merge conflicts
  - A merge conflicts occurs, if two people have modified the same content and Git cannot automatically determine how both changes should be applied.
  - If a merge conflict occurs Git will mark the conflict in the file and the programmer has to resolve the conflict manually. After resolving it, he can add the file to the staging index and commit the change.
- Example process for solving a merge conflict
  - In the following example you first create a merge conflict and afterwards you resolve the conflict and apply the change to the Git repository.
  - The following code creates a merge conflict

```
# Switch to the first directory
cd ~/repo01
# Make changes
echo "Change in the first repository" >
mergeconflict.txt
# Stage and commit
git add . && git commit -a -m "Will create
merge conflict 1"

# Switch to the second directory
cd ~/repo02
# Make changes
touch mergeconflict.txt
echo "Change in the second repository"
> mergeconflict.txt
# Stage and commit
git add . && git commit -a -m "Will create
merge conflict 2"
# Push to the master repository
git push
```

```
# Now try to push from the first
directory
# Switch to the first directory
cd ~/repo01
# Try to push --> you will get an error
message
git push


# Get the changes via a pull
# this creates the merge conflict in
your
# local repository
git pull origin master
```

Git marks the conflict in the affected file. This file looks like the following.

```
<<<<<<< HEAD
Change in the first repository
=======
Change in the second repository
>>>>>>> b29196692f5ebfd10d8a9ca1911c8b08127c85f8
```

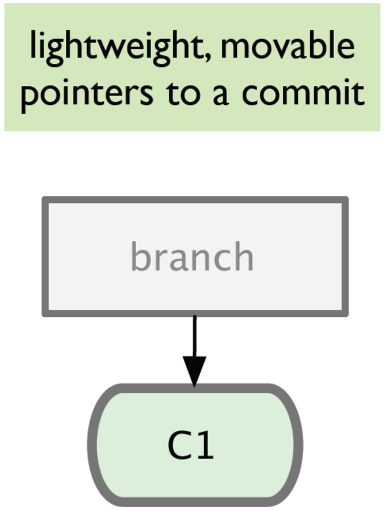
- The above is the part from your repository and the below one from the remote repository. You can edit the file manually and afterwards commit the changes.
- Alternatively, you could use the git mergetool command. git mergetool starts a configurable merge tool that displays the changes in a split screen. git mergetool is not always available but it is save to edit the file with merge conflicts by hand.
- # Either edit the file manually or use
- git mergetool
- # You will be prompted to select which merge tool you want to use
- # For example on Ubuntu you can use the tool "meld"
- # After merging the changes manually, commit them
- git commit -m "merged changes"

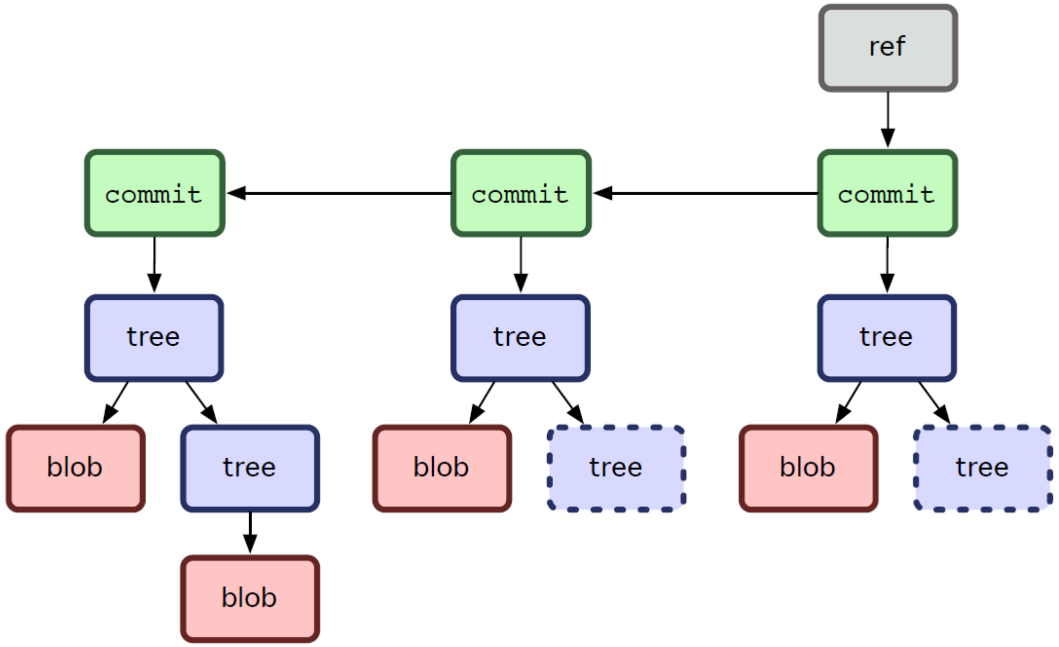
Rebase

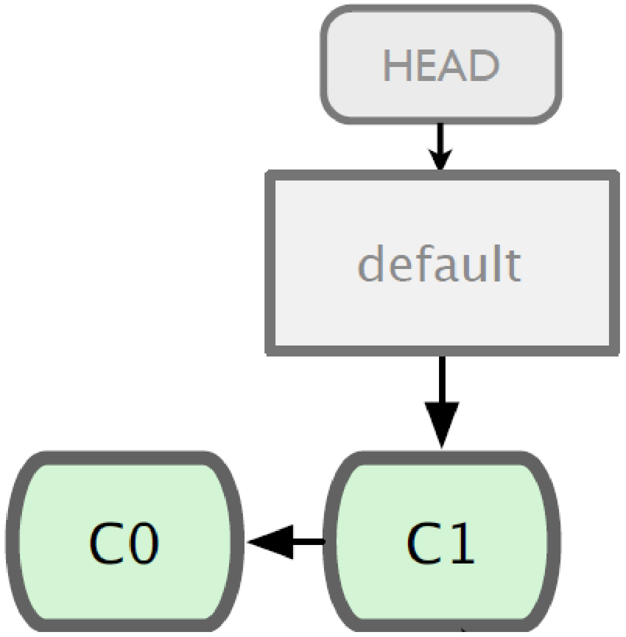
- You can use Git to rebase one branches on another one. As described the merge command combines the changes of two branches. If you rebase a branch called A onto another, the git rebase command takes the changes introduced by the commits of branch A and applies them based on the HEAD of the other branch. This way the changes in the other branch are also available in branch A.
  - # create new branch
  - git checkout -b rebasetest
  - # To some changes
  - touch rebase1.txt
  - git add . && git commit -m "work in branch“
  - # do changes in master
  - git checkout master# make some changes and commit into testing
  - echo "This will be rebased to rebasetest" > rebasefile.txtgit add rebasefile.txt
  - git commit -m "New file created“
  - # rebase the rebasetest onto master
  - git checkout rebasetest
  - git rebase master
  - # now you can fast forward your branch onto master
  - git checkout master
  - git merge rebasetest

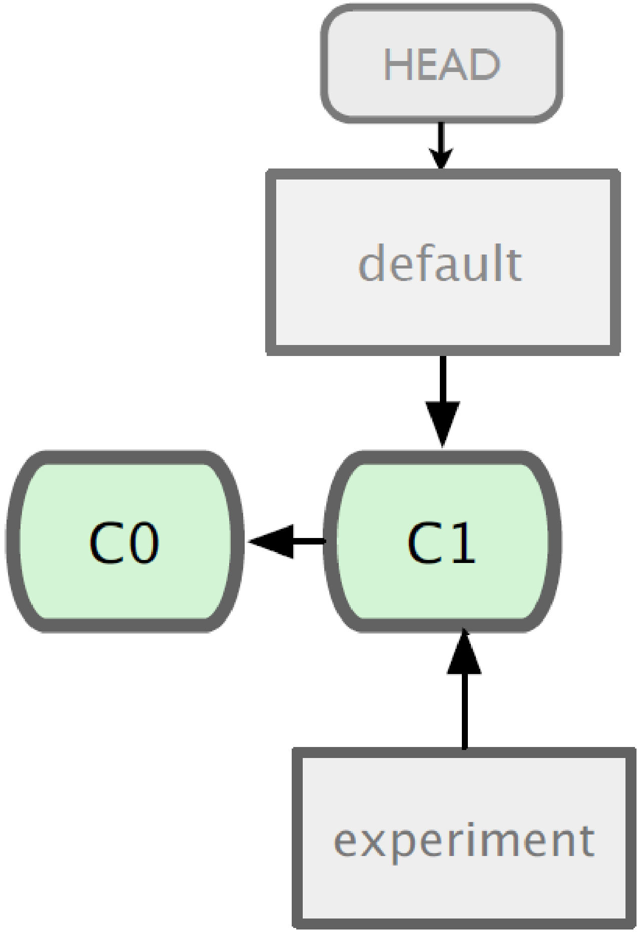
Branching

- Git sees commit this way…
- Branch annotates which commit we are working on

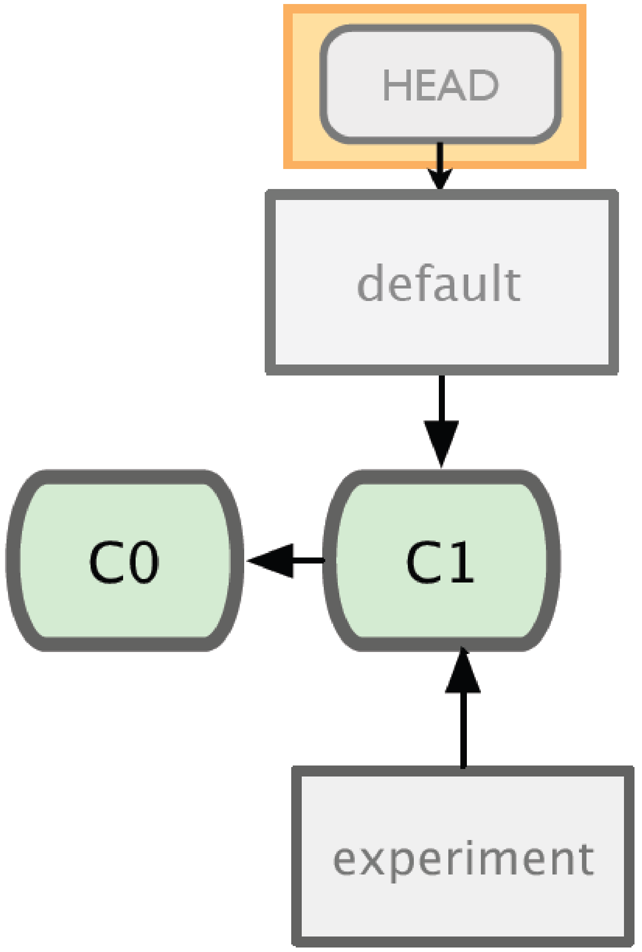




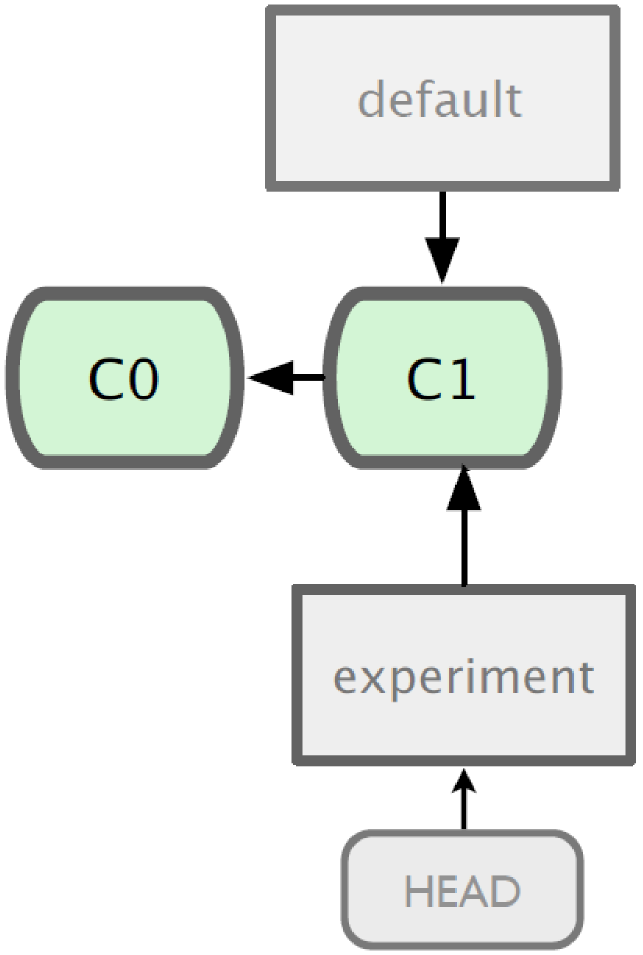

HEAD
default
C0
C1



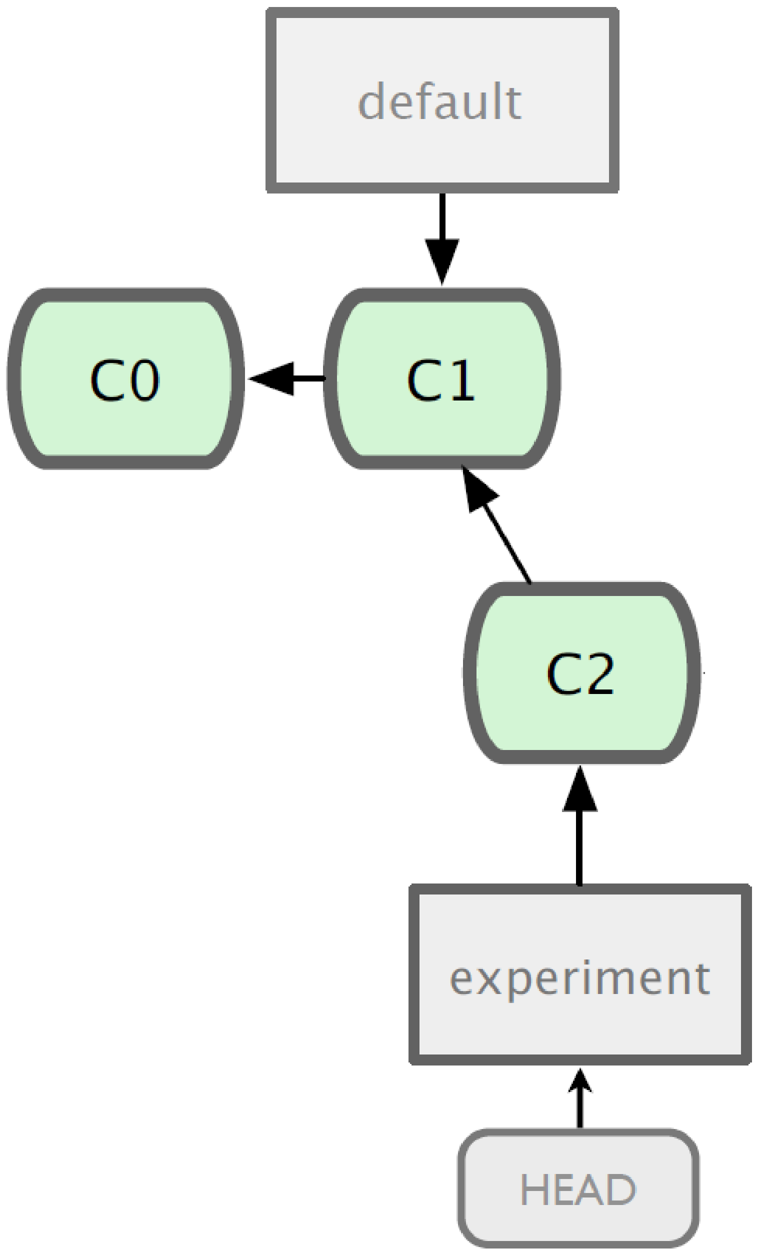
git branch experiment



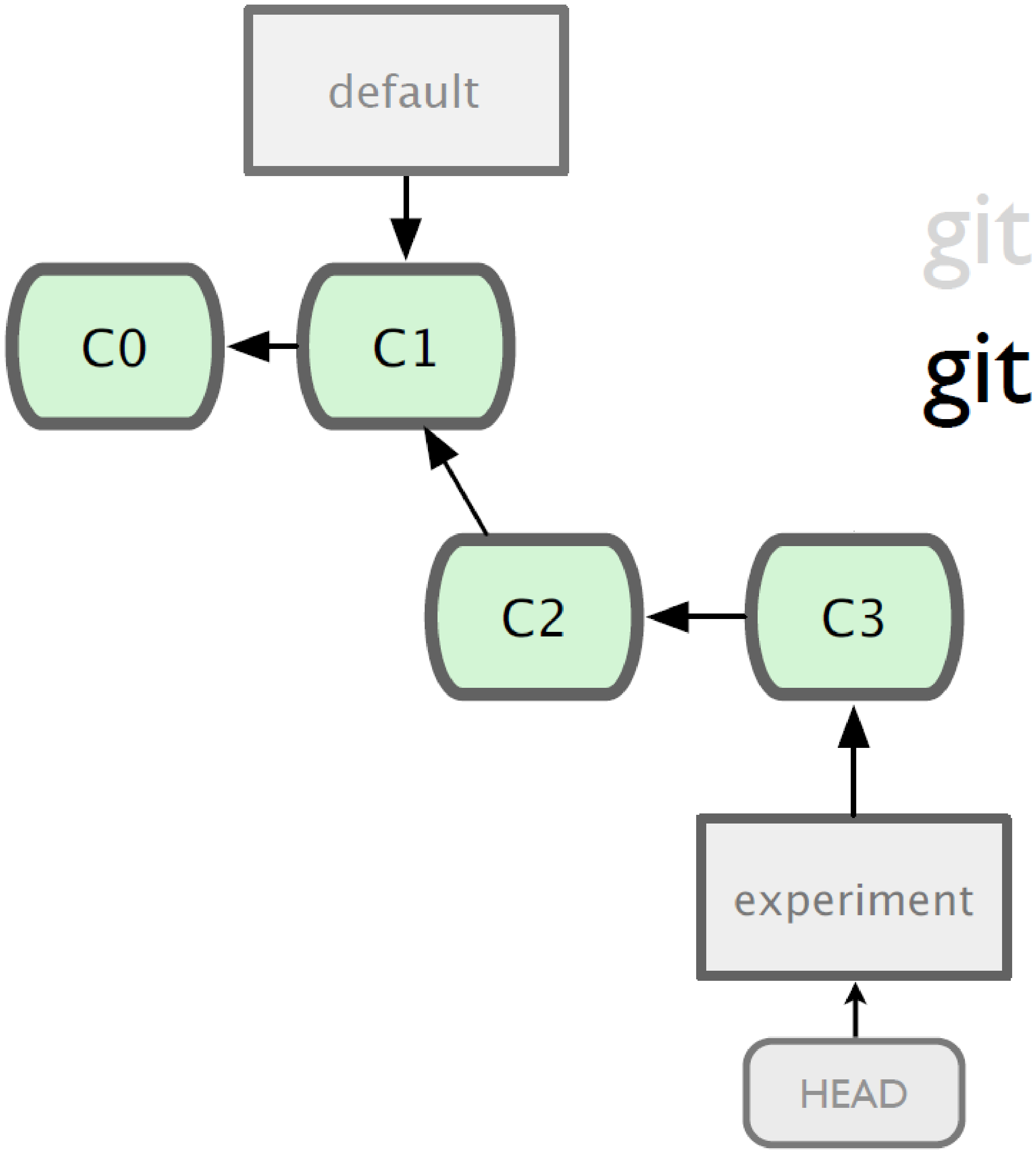
```
$ git branch
* default
  experiment
```



git checkout experiment

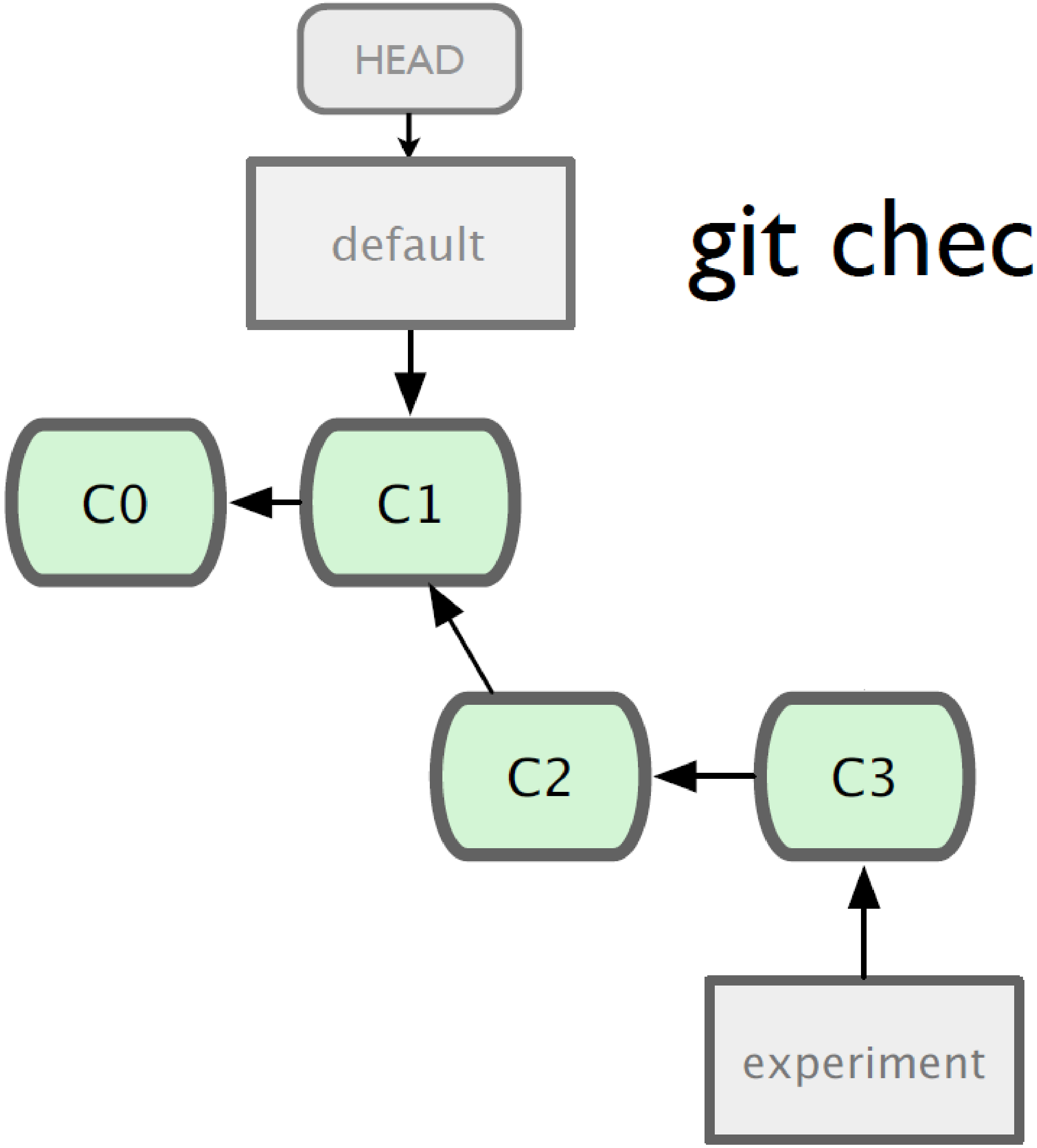


git commit



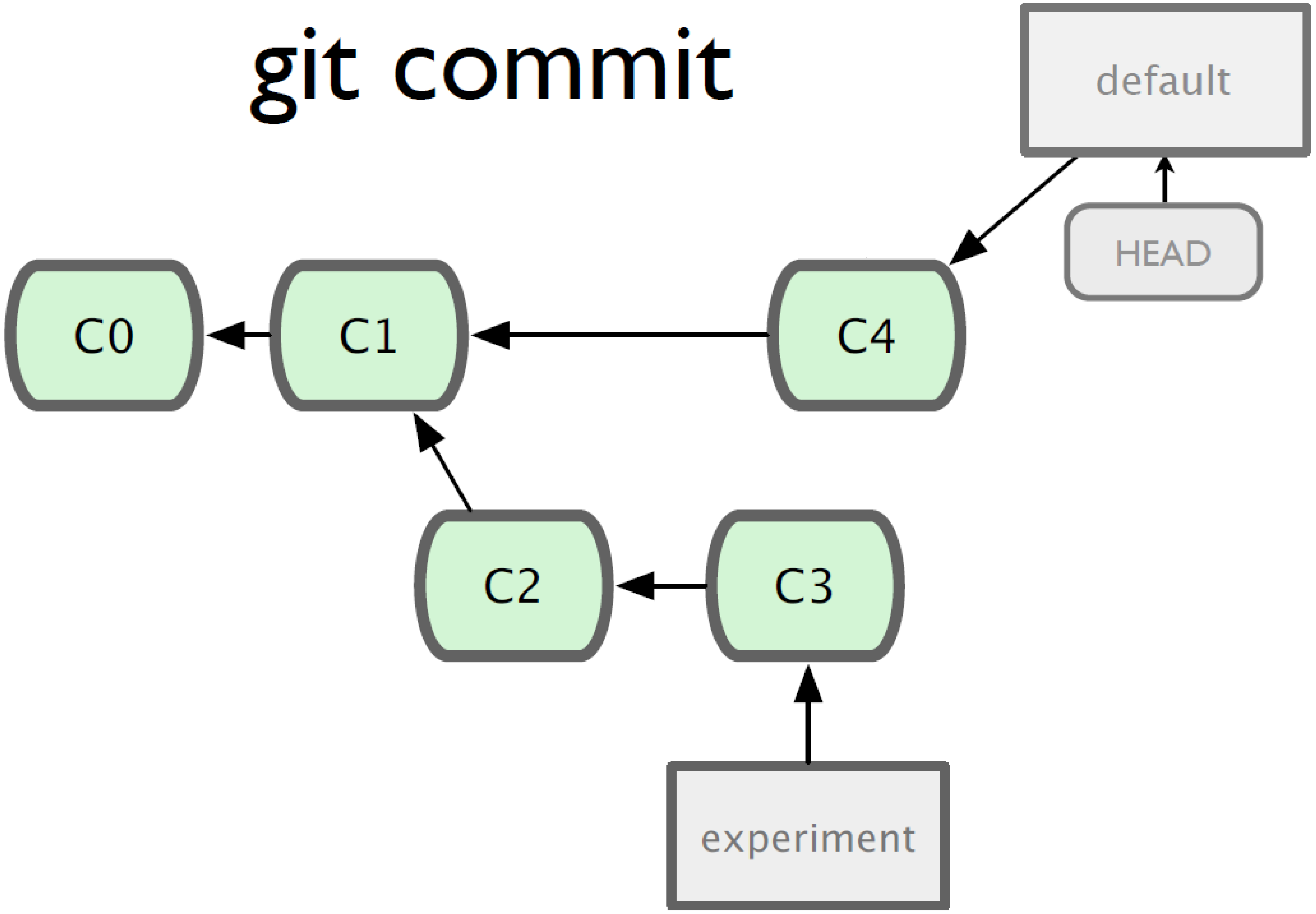
git commit

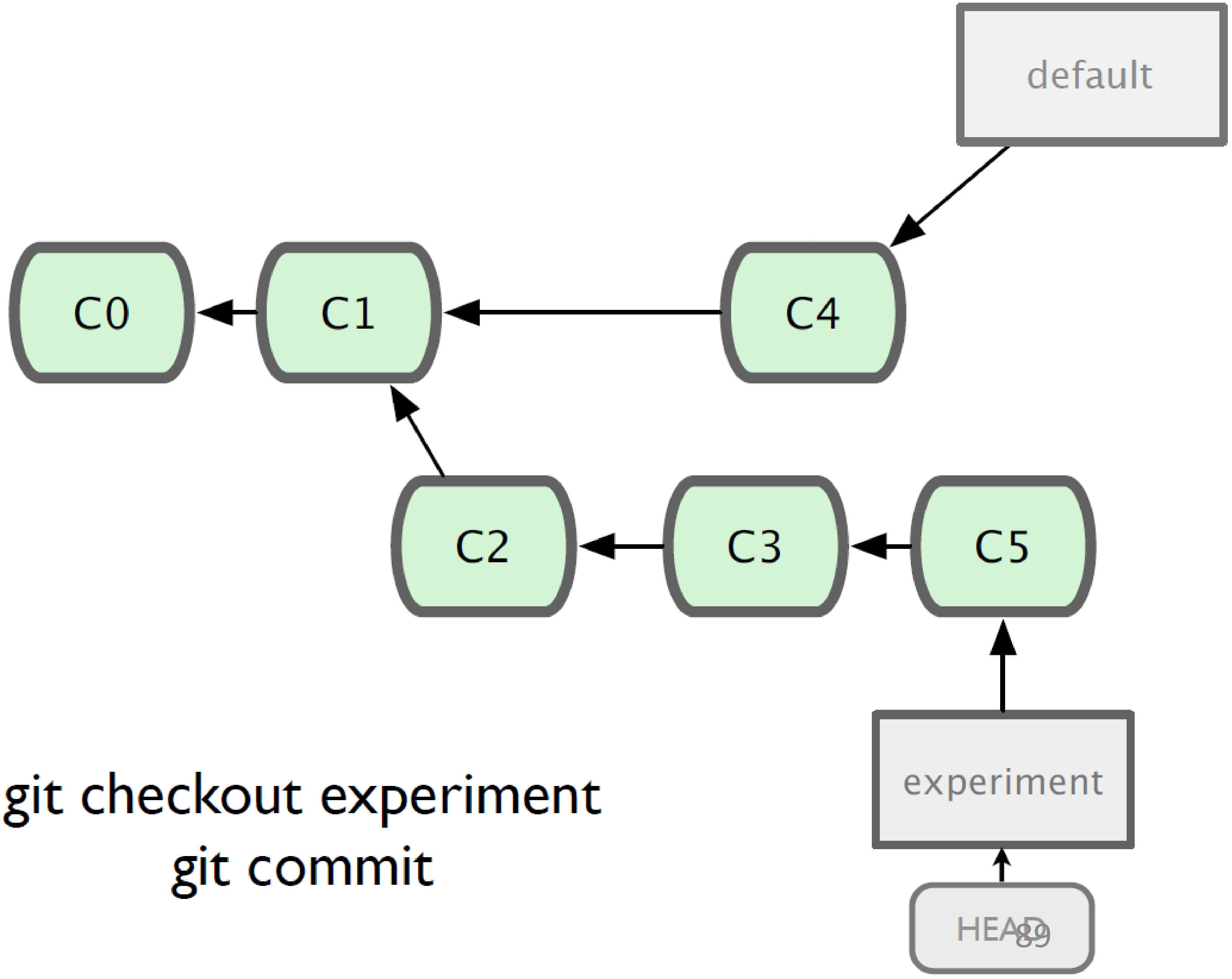
git commit



git checkout default

git commit

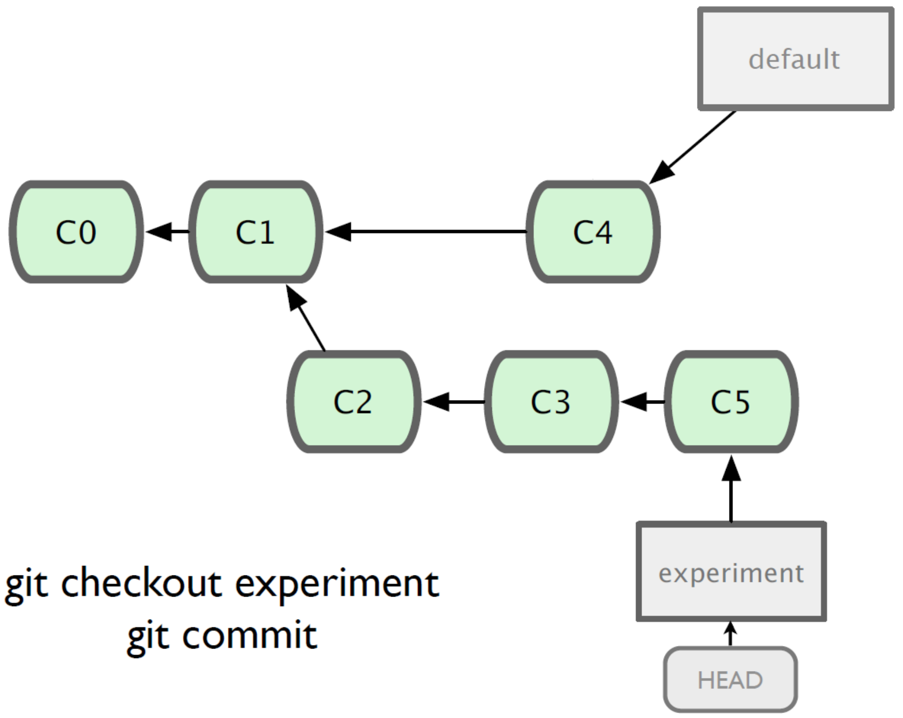




git checkout experiment
git commit

Merging

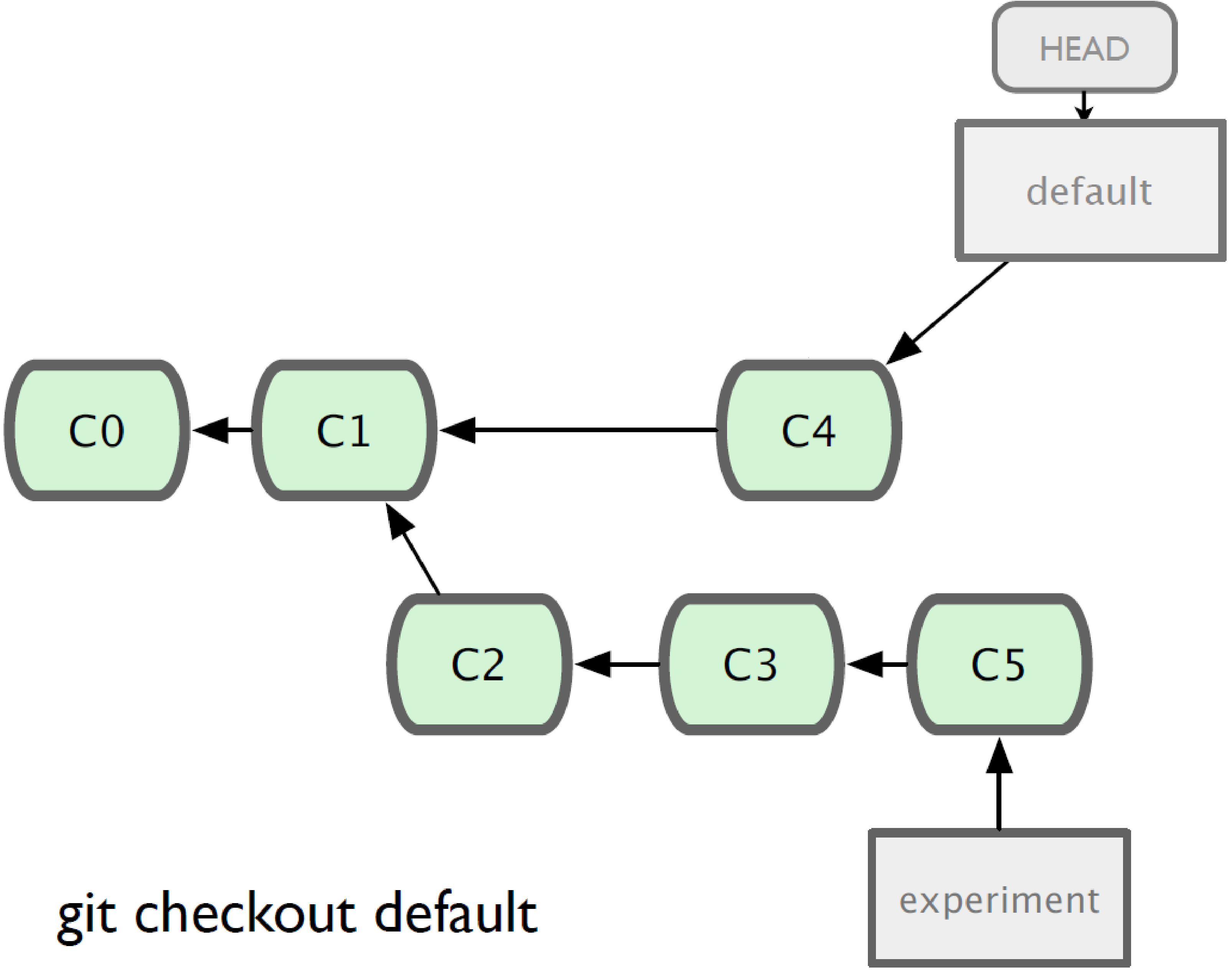
- What do we do with this mess?
  - Merge them



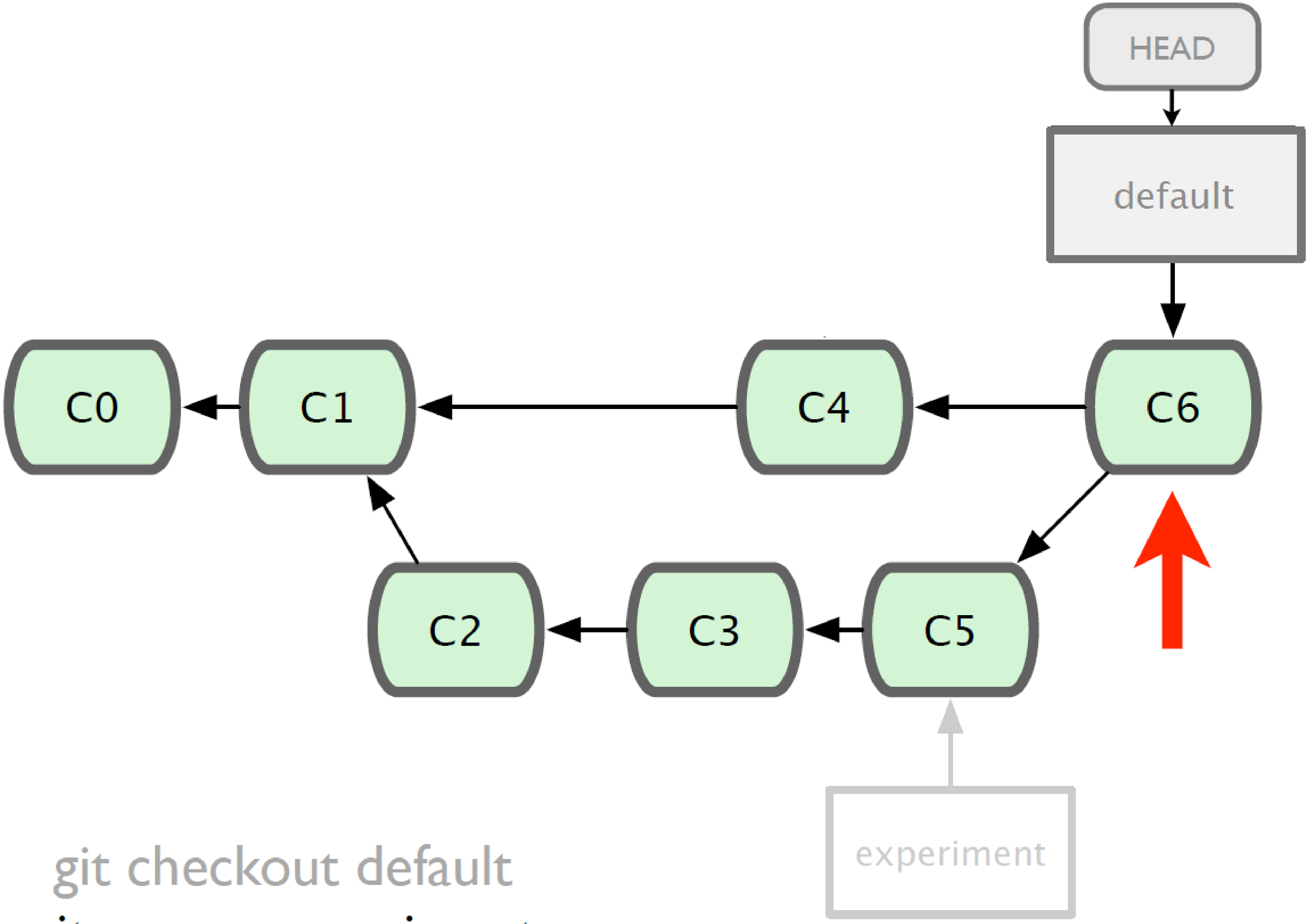
git checkout experiment
git commit

Merging

- Steps to merge two branch
  - Checkout the branch you want to merge onto
  - Merge the branch you want to merge

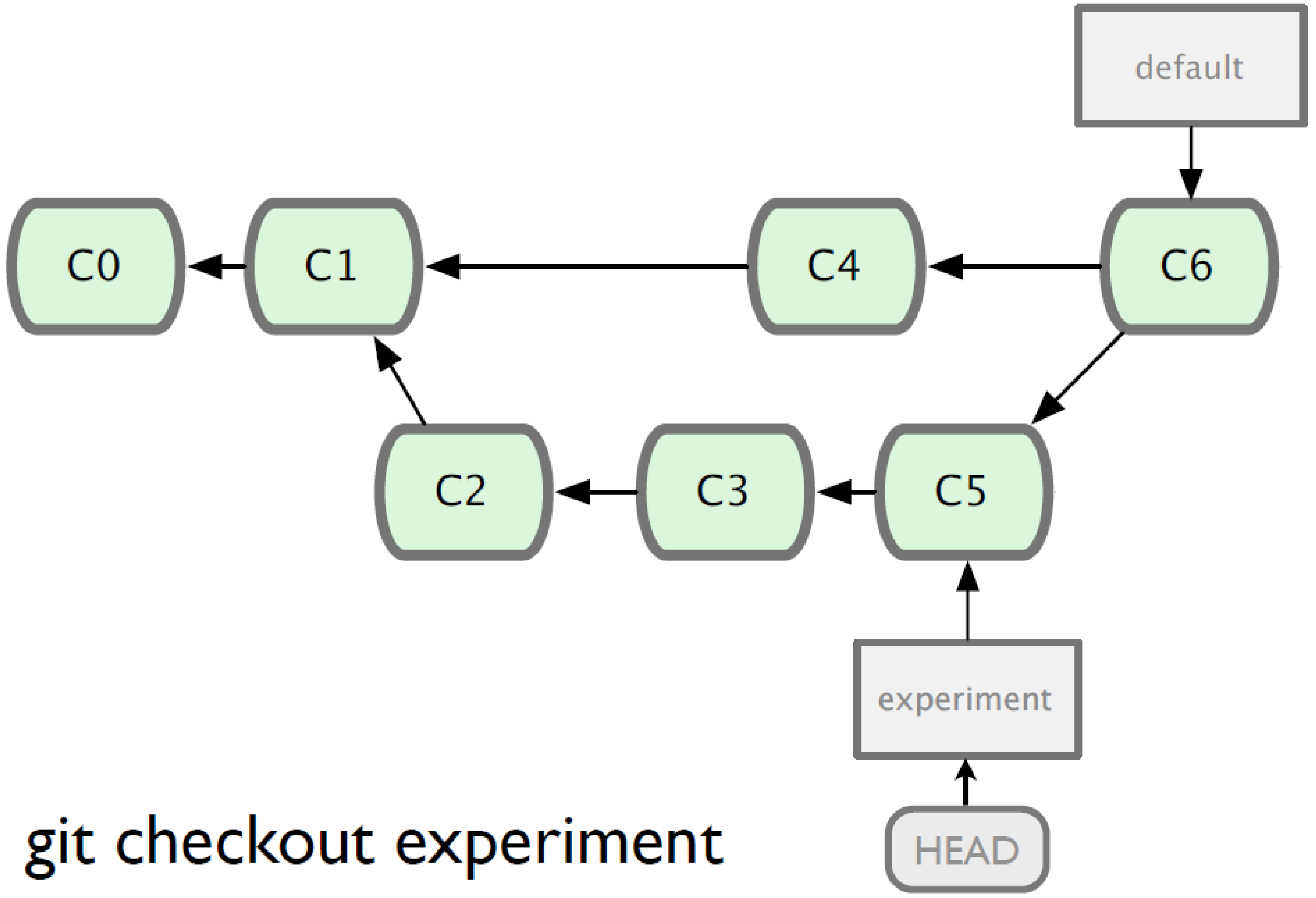


git checkout default

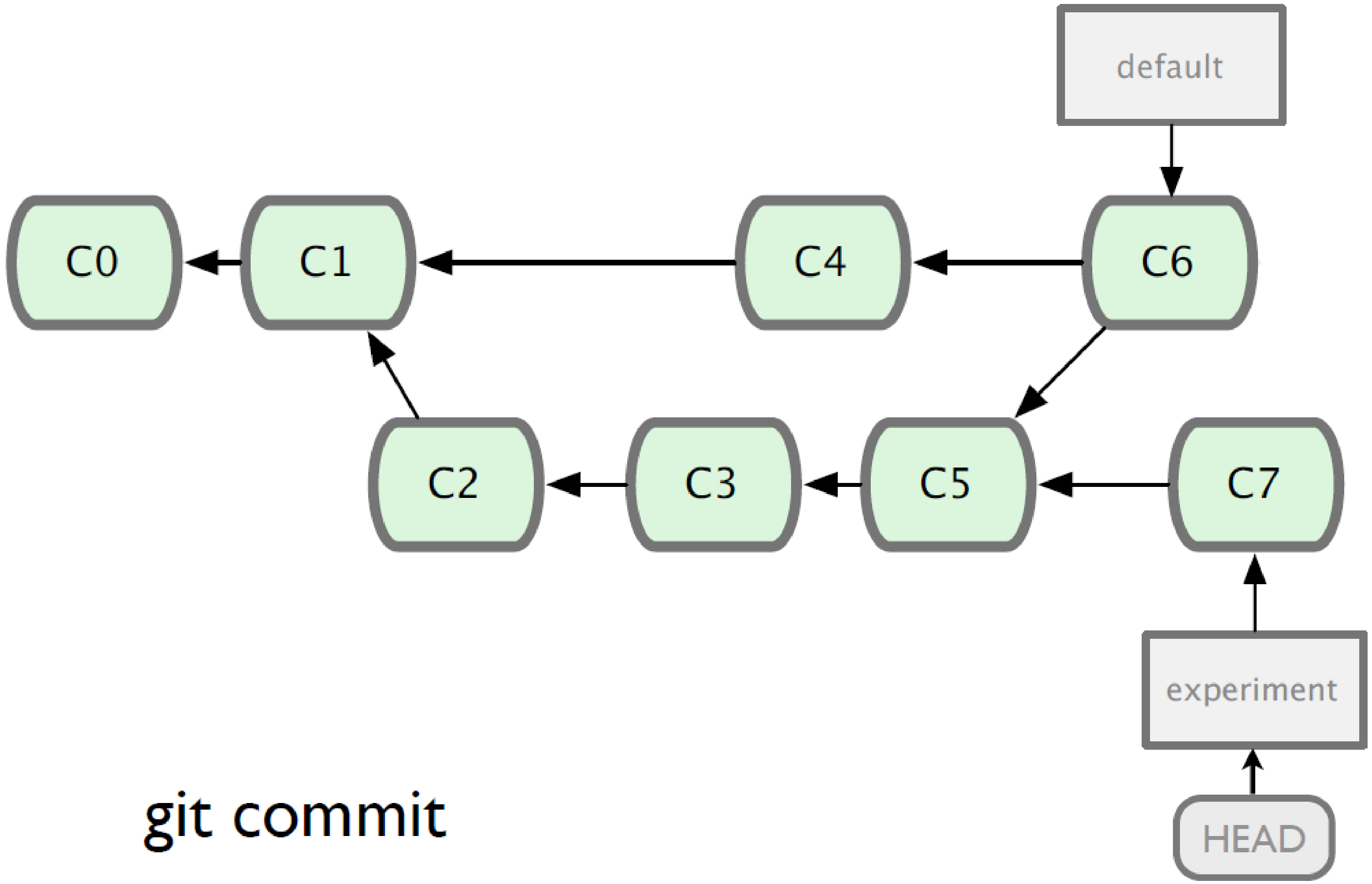


git checkout default

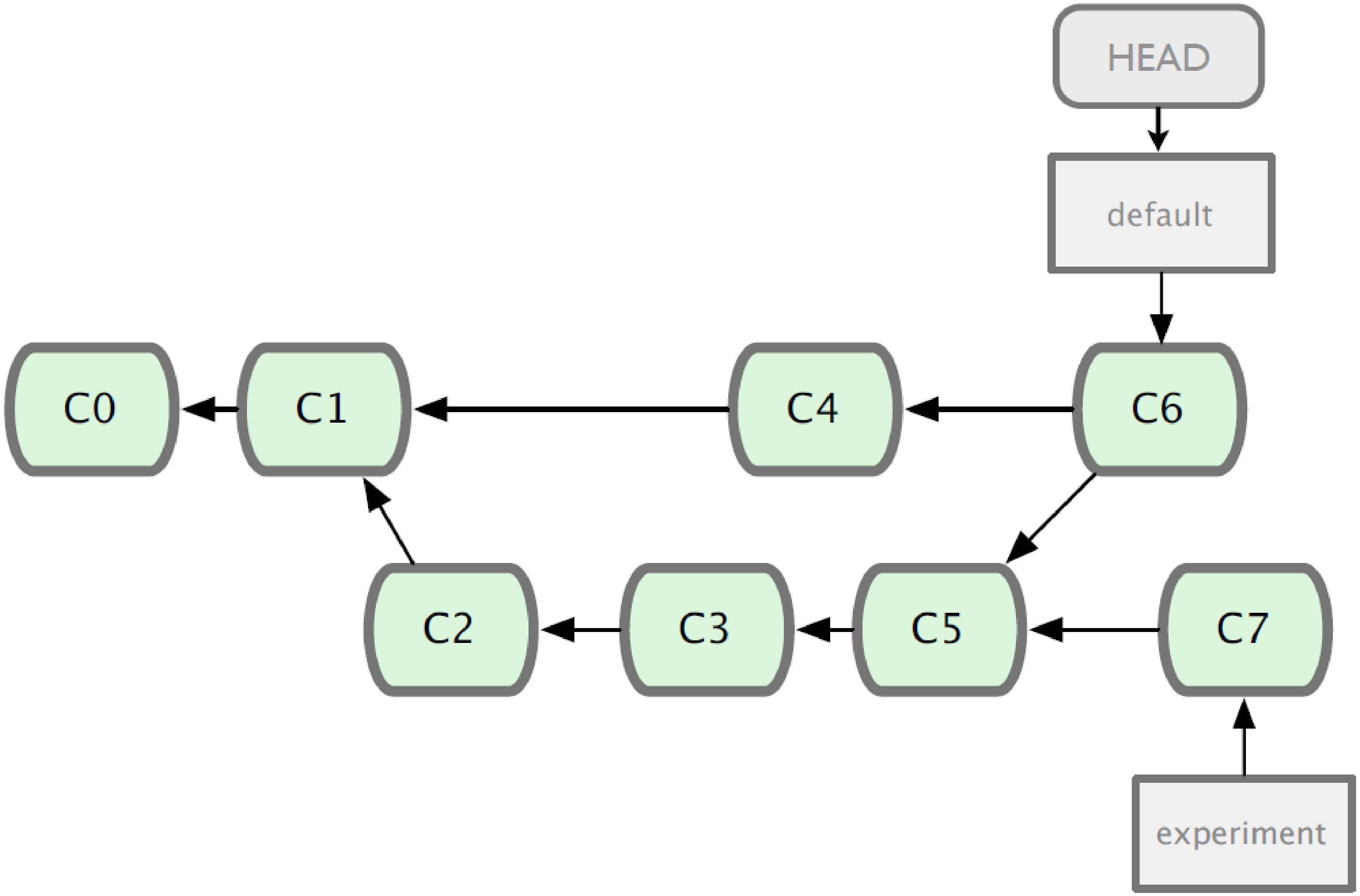
git merge experiment



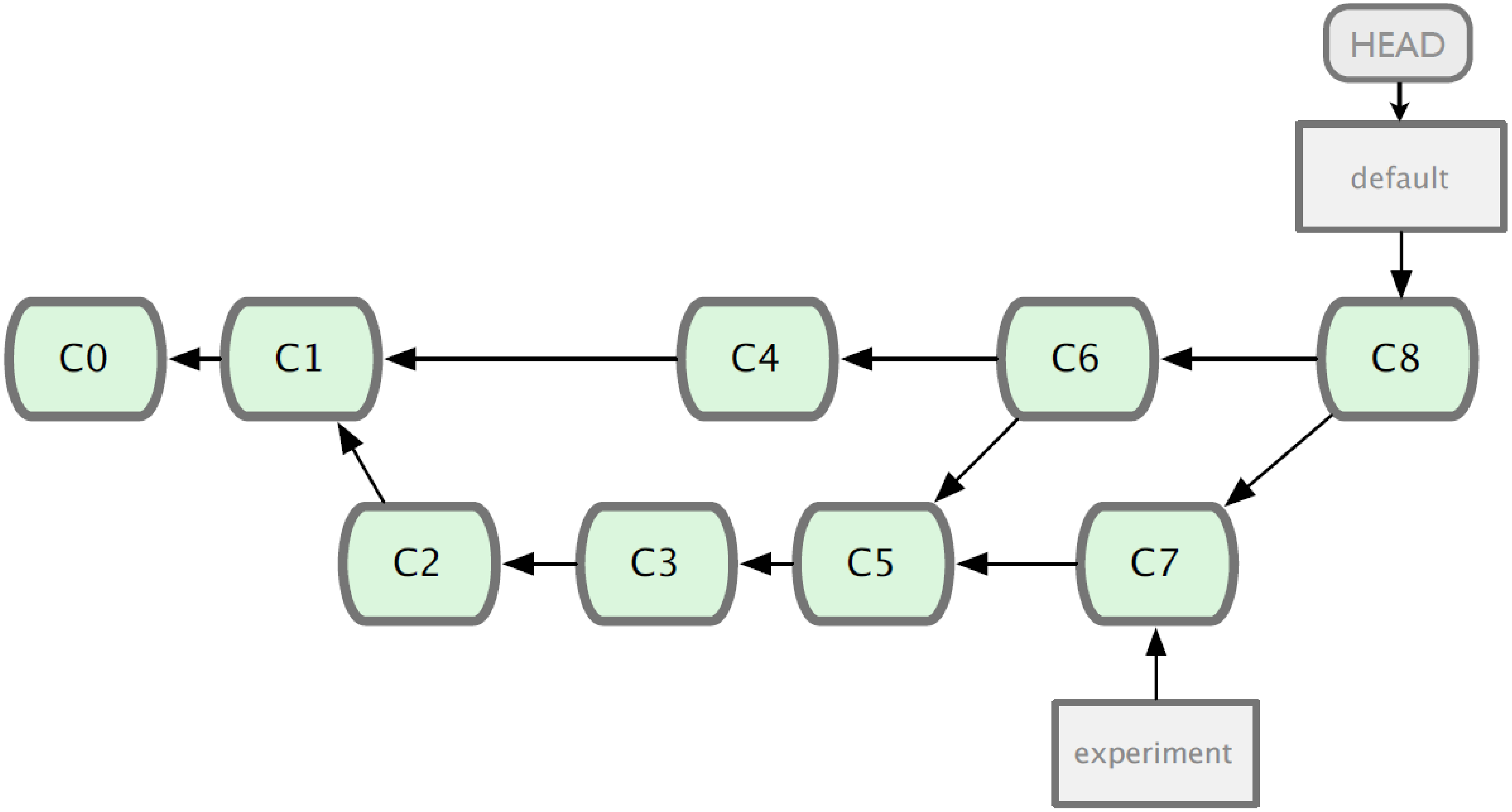
git checkout experiment



git commit



git checkout default



```
git merge experiment
```

Git Hooks

- Git has a way to fire off custom scripts when certain important actions occur.
- There are two groups of these hooks: client-side and server-side.
- Client-side hooks are triggered by operations such as committing and merging, while server-side hooks run on network operationssuch as receiving pushed commits.
- You can use these hooks for all sorts of reasons.

Git Hooks

- .git/hooks : When you initialize a new repository with git init, Git populates the hooks directory with a bunch of example scripts.
- To enable a hook script, put a file in the hooks subdirectory of your .git directory that is named appropriately (without any extension) and is executable.

Client-Side Hooks

- COMMITTING-WORKFLOW HOOKS
- pre-commit
  - The pre-commit hook is run first, before you even type in a commit message.
  - It’s used to inspect the snapshot that’s about to be committed, to see if you’ve forgotten something, to make sure tests run, or to examine whatever you need to inspect in the code. Exiting non-zero from this hook aborts the commit, although you can bypass it with git commit --no-verify.
  - You can do things like check for code style (run lint or something equivalent), check for trailing whitespace (the default hook does exactly this), or check for appropriate documentation on new methods.

Client-Side Hooks

- COMMITTING-WORKFLOW HOOKS
- commit-msg
  - The commit-msg hook takes one parameter, which again is the path to a temporary file that contains the commit message written by the developer.
  - If this script exits non-zero, Git aborts the commit process, so you can use it to validate your project state or commit message before allowing a commit to go through.

Client-Side Hooks

- COMMITTING-WORKFLOW HOOKS
- post-commit
  - After the entire commit process is completed, the post-commit hook runs.
  - It doesn’t take any parameters, but you can easily get the last commit by run-Git Hooks
  - git log -1 HEAD.
  - Generally, this script is used for notification or something similar.

Git Stash

- Use git stash when you want to record the current state of the working directory and the index, but want to go back to a clean working directory. The command saves your local modifications away and reverts the working directory to match the HEAD commit.

Thanks