Version Control Systems

Introduction to Git

Dennis Klein

Scientific IT
GSI Darmstadt

Panda Computing Workshop 2-7 July 2017
Suranaree University of Technology (SUT), Nakhon Ratchasima

Outline

Introduction

What is Version Control?

Definition

Version Control is a system that records changes to a set of files in a repository.

What is Version Control?

Definition

Version Control is a system that records changes to a set of files in a repository.

In the context of this talk we assume the repository is a source code repository.

What is Version Control?

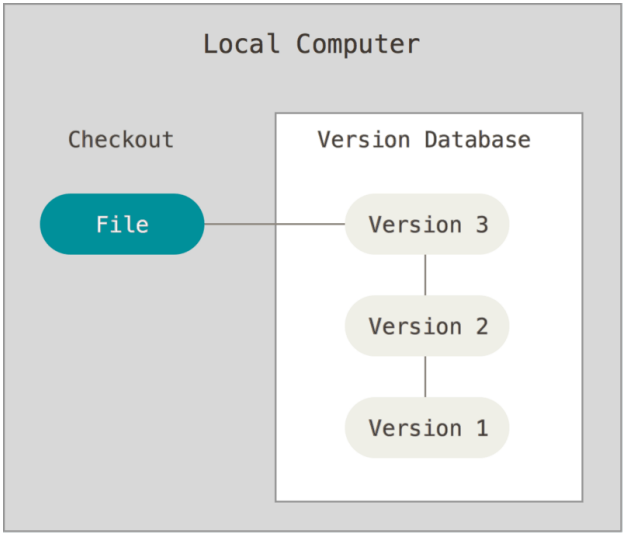
Definition

Version Control is a system that records changes to a set of files in a repository.

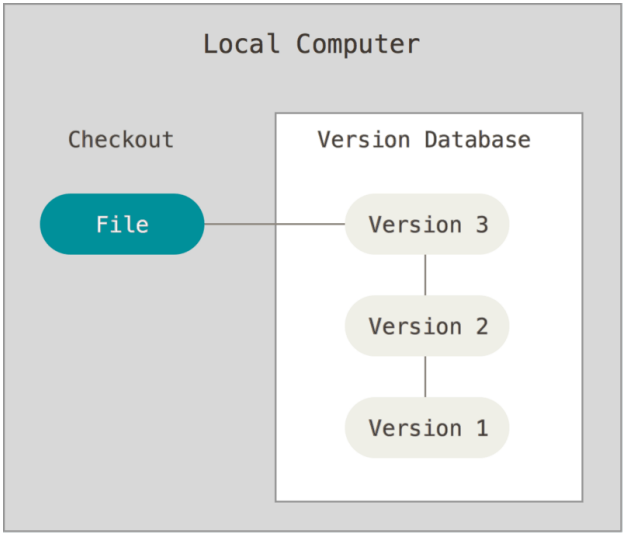
In the context of this talk we assume the repository is a source code repository. It allows to

- revert files back to a previous state,
- compare changes over time,
- keep track of who, when, and why changes are made,
- reference specific versions of the repository,
- and more.

Local Version Control System

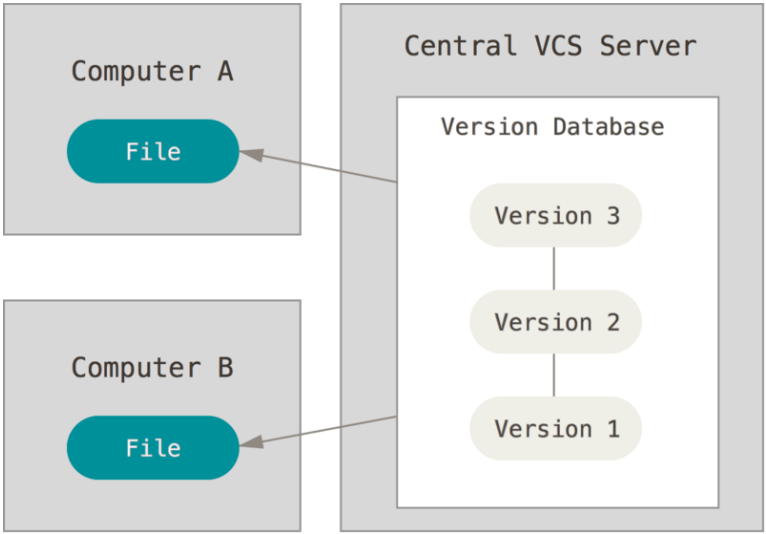


Local Version Control System

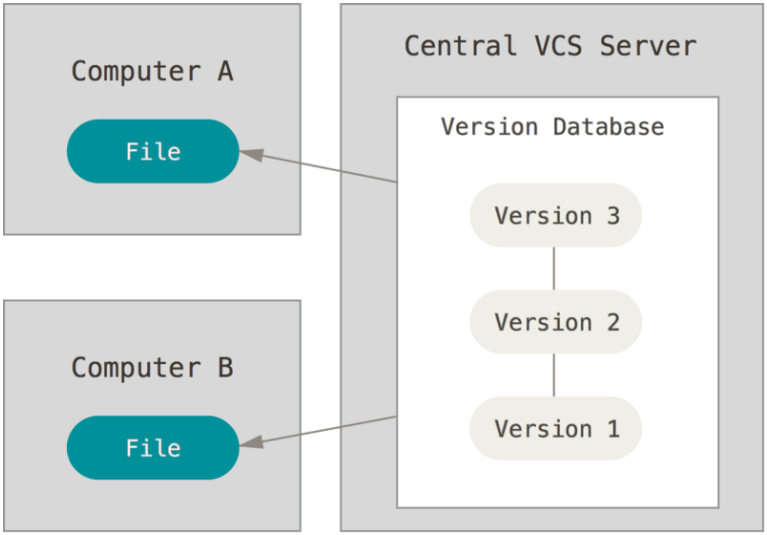


- Records patch sets (deltas) for each file on disk.
- Less error prone than manually copying files to seperate directories.
- E.g. RCS.

Centralized Version Control System (CVCS)

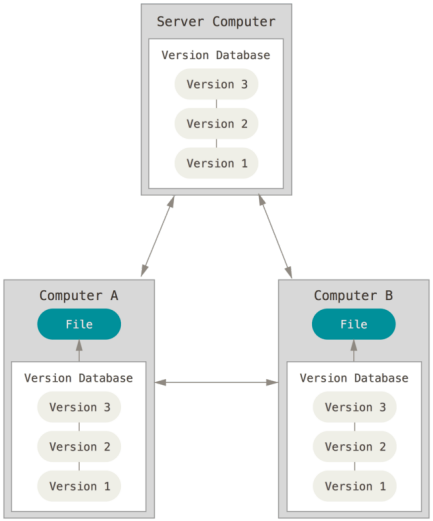


Centralized Version Control System (CVCS)

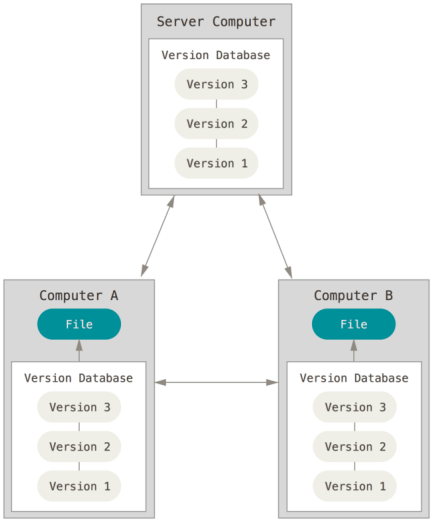


- Allows collaboration of developers.
- Server can facilitate fine-grained access control.
- E.g. CVS, Subversion, and Perforce.

Distributed Version Control System (DVCS)



Distributed Version Control System (DVCS)



- Each computer has a copy of the full history.
- E.g. Git and Mercurial.

About Git

- Written by Linus Torvalds in 2005
- Replace commercial DVCS BitKeeper for Linux development
- Design Goals were
  - Speed,
  - Simple design,
  - Strong support for non-linear development,
  - Fully distributed, and
  - Scalability for large projects like Linux.

Git Basics

Configuration

For installation, see `https://git-scm.com/downloads`

Minimal configuration

```
git config --global user.name "John Doe"
git config --global user.email "johndoe@example.com"
git config --global core.editor "nvim"
```

- Per user config in `~/.gitconfig`
- Per repo config in `/REPOPATH/.git/config`
- Use *Git Attributes* for per directory config

Check config with `git config --list`

Advanced Configuration

Recommended configuration

```
git config --global alias.st "status -bs"
git config --global alias.l "log --pretty=format:\
'%C(yellow)%h %Cred%ad %Cblue%an%Cgreen%d %Creset%s'\
--graph"
```

- Concise status output with `git st`
- Compact history view with `git l`

Init

Initialize a git repo in an existing directory

```
mkdir demo && cd demo
git init
echo ".swp" > .gitignore
git add .gitignore
git commit -m "Initial commit"
```

- Initializes `.git` subdirectory
- Initial commit
- No explicit repo name

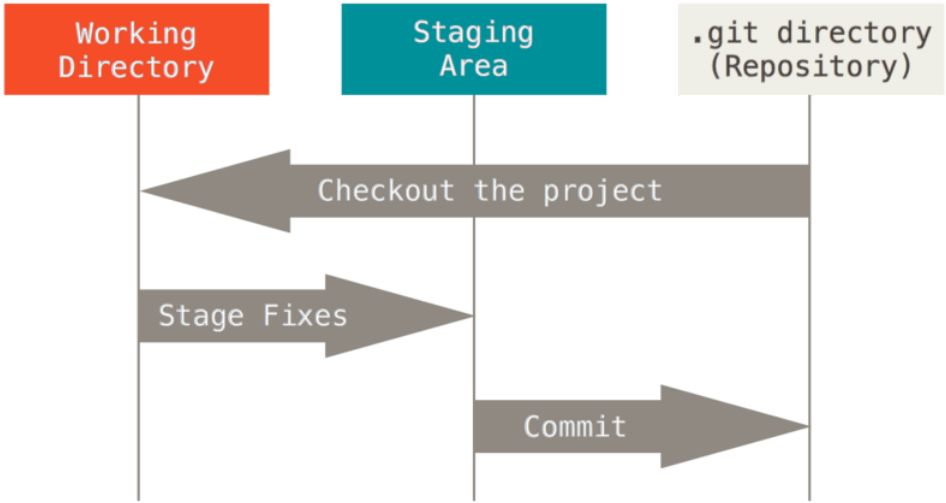
Clone

Clone an existing repo

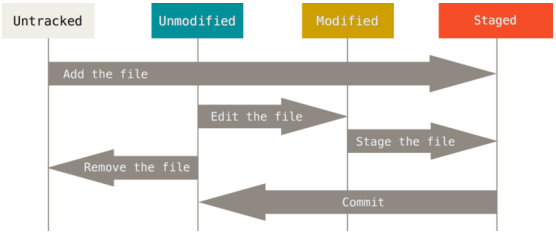
```
git clone https://github.com/FairRootGroup/FairRoot
git clone git@github.com:FairRootGroup/FairRoot
git clone git://github.com/FairRootGroup/FairRoot
git clone /some_fs/FairRoot
```

- Clone with different transfer protocols:
  - HTTP,
  - SSH,
  - GIT, and
  - filesystem

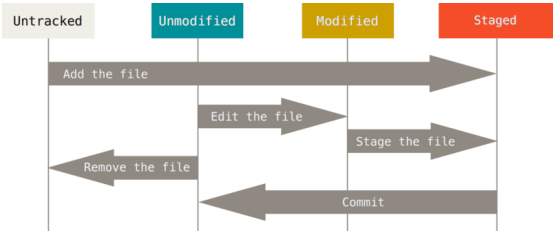
Areas


Working Directory
Staging Area
.git directory (Repository)
Checkout the project
Stage Fixes
Commit

Lifecycle


Untracked
Unmodified
Modified
Staged
Add the file
Edit the file
Stage the file
Remove the file
Commit

Lifecycle



Check current status with `git status`

- Add: `git add`
- Stage: `git add`
- Edit: your favourite editor
- Remove (tracked): `git rm`
- Rename (tracked): `git mv`
- Unstage: `git reset HEAD`
- Unmodify: `git checkout --`

Diff

- `git diff` to view unstaged changes
- `git diff --staged` (or `--cached`) to view staged changes

Diff

- `git diff` to view unstaged changes
- `git diff --staged` (or `--cached`) to view staged changes

Best practise

Before each commit, view `git diff --staged` to verify and be sure what you are about to commit.

Good commit messages

`https://chris.beams.io/posts/git-commit/`

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain what and why vs. how

Good commit messages

`https://chris.beams.io/posts/git-commit/`

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain what and why vs. how

Subject line template

If applied, this commit will *your subject line here*

Good commit messages

`https://chris.beams.io/posts/git-commit/`

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain what and why vs. how

Subject line template

If applied, this commit will *your subject line here*

```
Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
subject of the commit and the rest of the text as the body. The
blank line separating the summary from the body is critical (unless
you omit the body entirely); various tools like `log`, `shortlog`
and `rebase` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you
are making this change as opposed to how (the code explains that).
Are there side effects or other unintuitive consequences of this
change? Here's the place to explain them.

Further paragraphs come after blank lines.

 - Bullet points are okay, too

 - Typically a hyphen or asterisk is used for the bullet, preceded
   by a single space, with blank lines in between, but conventions
   vary here

If you use an issue tracker, put references to them at the bottom,
like this:

Resolves: #123
See also: #456, #789
```

Committing

Command

```
git commit
```

Opens your editor and lets you type the commit message

Committing

Command

```
git commit
```

Opens your editor and lets you type the commit message

Command

```
git commit -m "add a feature"
```

Shortcut for simple commit messages

Committing

Command

```
git commit
```

Opens your editor and lets you type the commit message

Command

```
git commit -m "add a feature"
```

Shortcut for simple commit messages

Command

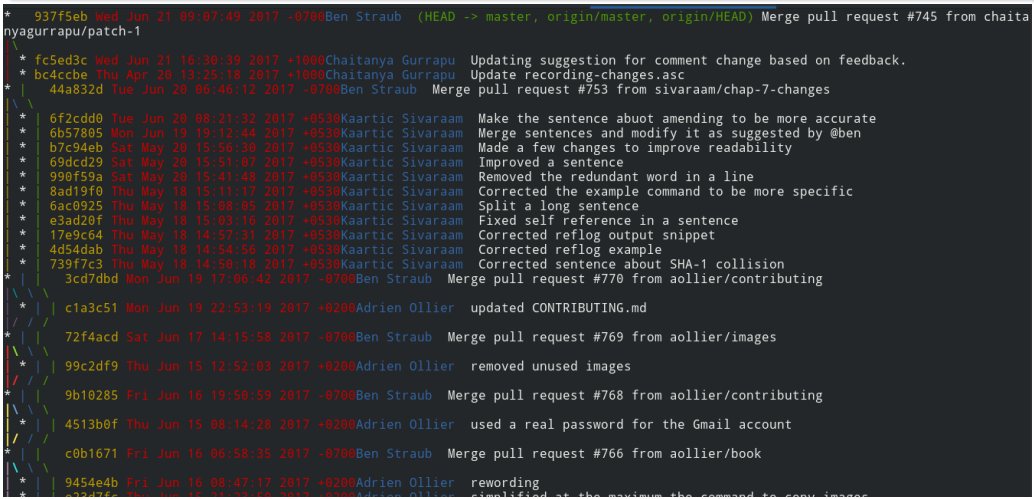
```
git commit --amend
```

Recommit the last commit with the current staged changes

Viewing the history

Command

```
git l(og)
```



Viewing a single commit

Command

`git show` *commit hash*

```
commit 6f2cdd07f1649c10ba5e0f62e12b6f9a98971de0
Author: Kaartic Sivaraam <kaarticsivaraam91196@gmail.com>
Date:   Tue Jun 20 08:21:32 2017 +0530

    Make the sentence abuot amending to be more accurate

diff --git a/book/07-git-tools/sections/rewriting-history.asc b/book/07-git-tools/sections/rewriting-history.asc
index 9819e09..2f1d2d2 100644
--- a/book/07-git-tools/sections/rewriting-history.asc
+++ b/book/07-git-tools/sections/rewriting-history.asc
@@ -25,7 +25,7 @@ That drops you into your text editor, which has your last commit message in it,
 When you save and close the editor, the editor writes a new commit containing that message and makes it your new last commit.

 If you've committed and then you want to change the snapshot you committed by adding or changing files, possibly because you forgot t
o add a newly created file when you originally committed, the process works basically the same way.
-You stage the changes you want by editing a file and running `git add` on it or `git rm` to a tracked file, and the subsequent `git c
ommit --amend` takes your current staging area and makes it the snapshot for the new commit.
+You stage the changes you want by editing a file and running `git add` on it or `git rm` to a tracked file, and the subsequent `git c
ommit --amend` takes your current staging area and adds it to the last snapshot making it the snapshot of the last commit.

 You need to be careful with this technique because amending changes the SHA-1 of the commit.
 It's like a very small rebase – don't amend your last commit if you've already pushed it.
(END)
```

Remotes

Definition

Remotes are repository-local names which refer to other git repositories.

Remotes

Definition

Remotes are repository-local names which refer to other git repositories.

Command

```
git remote -v
```

```
dklein@gamma ~/FairRoot ‹ruby-2.3.4› ‹plugins›
➤ git remote -v
dklein  git@github.com:dennisklein/FairRoot (fetch)
dklein  git@github.com:dennisklein/FairRoot (push)
origin  git@github.com:FairRootGroup/FairRoot (fetch)
origin  git@github.com:FairRootGroup/FairRoot (push)
rbx     https://github.com/rbx/FairRoot (fetch)
rbx     https://github.com/rbx/FairRoot (push)
```

origin is the default remote after cloning a repository

Remotes

Definition

Remotes are repository-local names which refer to other git repositories.

Command

```
git remote -v
```

```
dklein@gamma ~/FairRoot ‹ruby-2.3.4› ‹plugins›
➤ git remote -v
dklein  git@github.com:dennisklein/FairRoot (fetch)
dklein  git@github.com:dennisklein/FairRoot (push)
origin  git@github.com:FairRootGroup/FairRoot (fetch)
origin  git@github.com:FairRootGroup/FairRoot (push)
rbx     https://github.com/rbx/FairRoot (fetch)
rbx     https://github.com/rbx/FairRoot (push)
```

origin is the default remote after cloning a repository

Command

```
git remote add url
git remote remove name
```

Fetch and Push

Command

`git fetch` *remote*

Downloads all objects (e.g. commits) and refs (e.g. tags) from the remote repository that you do not have yet. Only deltas are transferred.

Command

`git push` *remote branch*

Commit hash

Definition

The *commit hash* is the result of applying a cryptographic hash function on the metadata and patch set of a git commit. It "uniquely" identifies a git commit and is usually represented as a character string.

Commit hash

Definition

The *commit hash* is the result of applying a cryptographic hash function on the metadata and patch set of a git commit. It "uniquely" identifies a git commit and is usually represented as a character string.

- Used to reference a certain version of the repository.
- Easy detection of repository corruption/tampering.
- No natural order of the commit hash space that follows the commit history.
- One can use a short version in most cases.

Tagging I

Definition

A *tag* is an arbitrary repository-local name that points to a commit hash.

Tagging I

Definition

A *tag* is an arbitrary repository-local name that points to a commit hash.

- Used to point out important points in the history.
- Usually software projects apply their versioning scheme via tags.

Tagging II

Command

```
git tag
git tag -l "v1.5.*"
```

List tags.

Tagging II

Command

```
git tag
git tag -l "v1.5.*"
```

List tags.

Command

```
git tag name [commit hash]
git tag -a name -m "Major release"
git tag -d name
```

Create and delete lightweight and annotated tags.

Tagging II

Command

```
git tag
git tag -l "v1.5.*"
```

List tags.

Command

```
git tag name [commit hash]
git tag -a name -m "Major release"
git tag -d name
```

Create and delete lightweight and annotated tags.

Command

```
git push remote tag
git push remote --tags
```

Push single or all tags to the remote.

Git Workflow

Branches

Definition

A *branch* is similar to a lightweight tag. It is a name pointing to a commit hash. In addition, the pointer tracks (is automatically updated to) the most recent commit.

Branches

Definition

A *branch* is similar to a lightweight tag. It is a name pointing to a commit hash. In addition, the pointer tracks (is automatically updated to) the most recent commit.

A built-in ref named HEAD can point to a branch. This branch is then the current branch.

Branches

Definition

A *branch* is similar to a lightweight tag. It is a name pointing to a commit hash. In addition, the pointer tracks (is automatically updated to) the most recent commit.

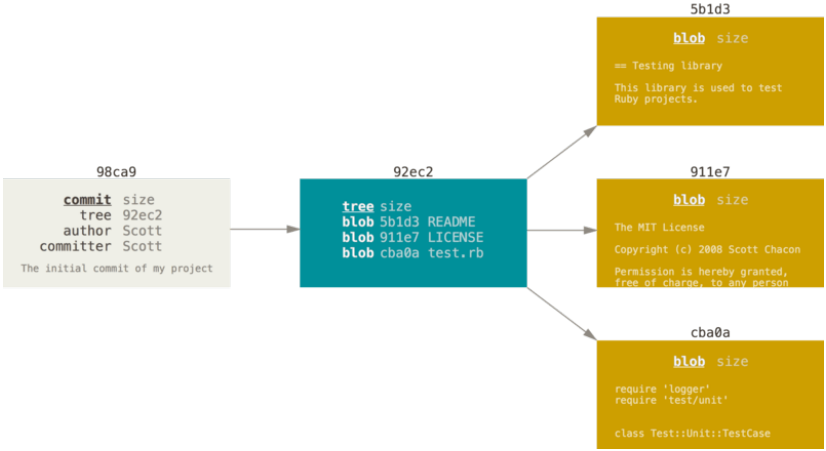
A built-in ref named HEAD can point to a branch. This branch is then the current branch.

Command

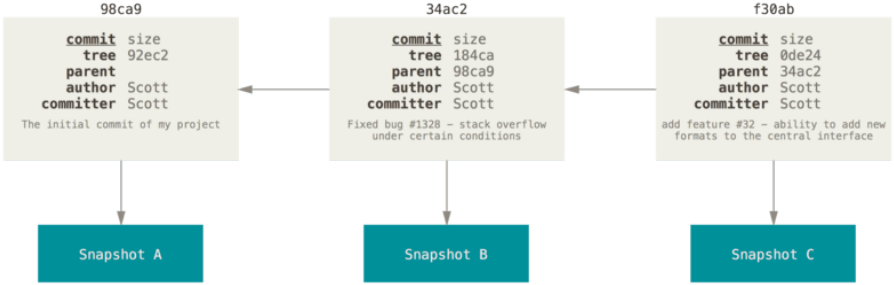
git branch
git checkout *branch name*
git checkout -b *branch name*

List branches, set current branch (HEAD), and create a new branch.

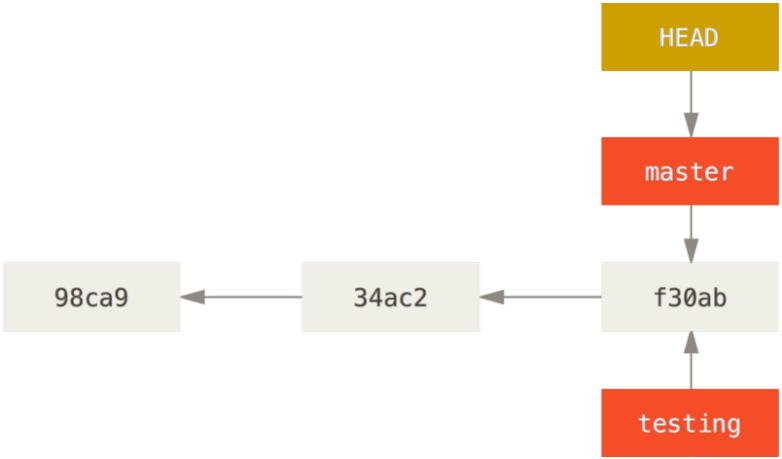
Internals I


98ca9
commit size
tree 92ec2
author Scott
committer Scott
The initial commit of my project
92ec2
tree size
blob 5b1d3 README
blob 911e7 LICENSE
blob cba0a test.rb
5b1d3
blob size
== Testing library
This library is used to test
Ruby projects.
911e7
blob size
The MIT License
Copyright (c) 2008 Scott Chacon
Permission is hereby granted,
free of charge, to any person
cba0a
blob size
require 'logger'
require 'test/unit'
class Test::Unit::TestCase

Internals II



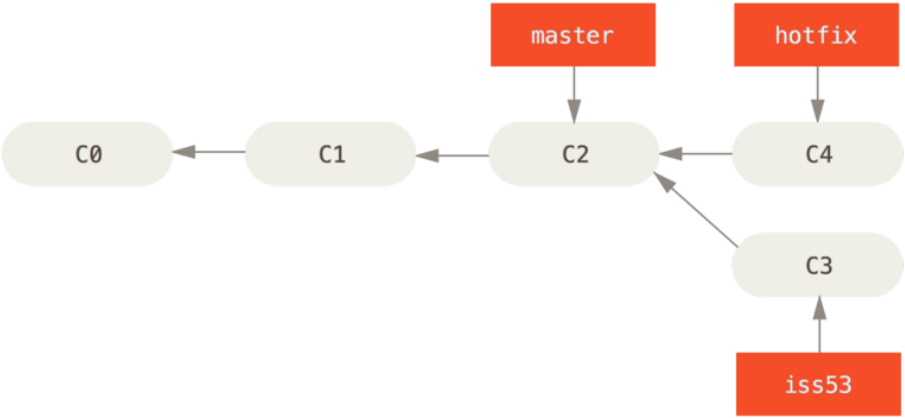
Internals III



Merge fast-forward I

Command

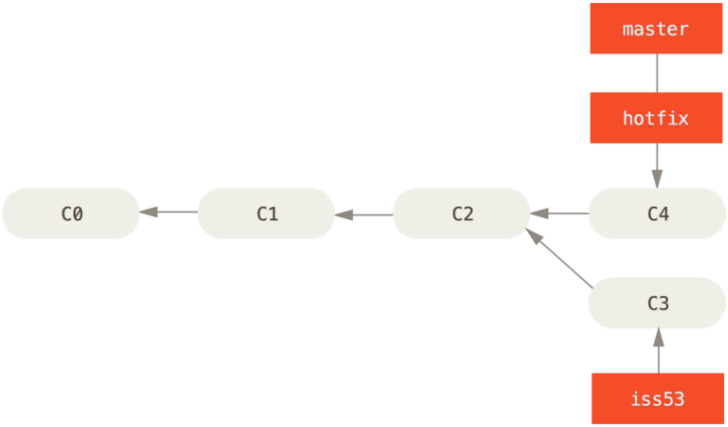
```
git checkout master
git merge hotfix
```



Merge fast-forward II

Command

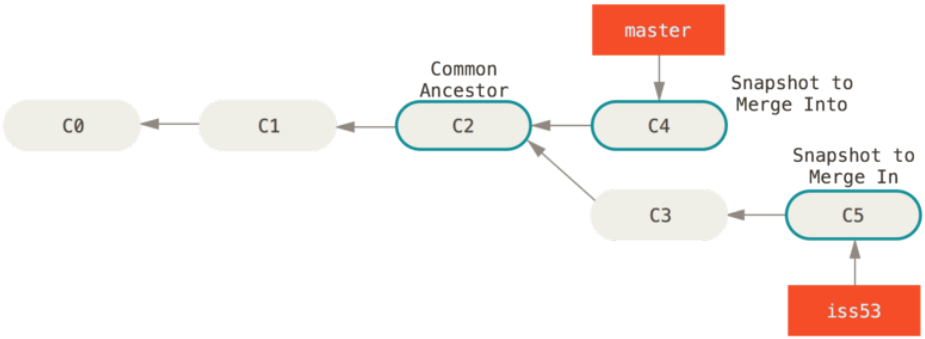
```
git checkout master
git merge hotfix
```



Merge commit I

Command

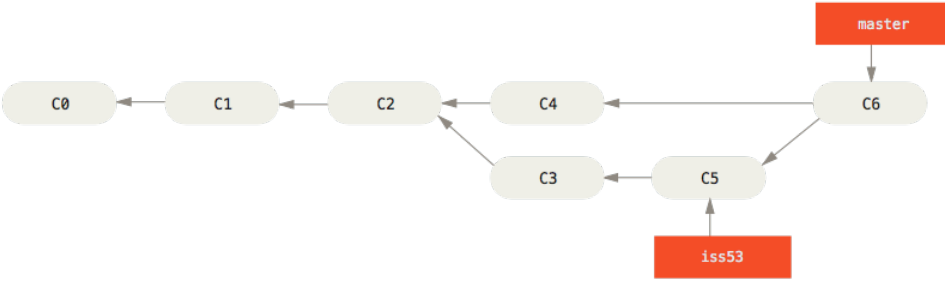
```
git checkout master
git merge iss53
```



Merge commit II

Command

```
git checkout master
git merge iss53
```



Pull Requests

Definition

A git *pull* is a shortcut for a *fetch* and *merge*.

Pull Requests

Definition

A git *pull* is a shortcut for a *fetch* and *merge*.

Definition

A *pull request* is the managed process of a git merge (usually via a webbased software tool).

Pull Requests

Definition

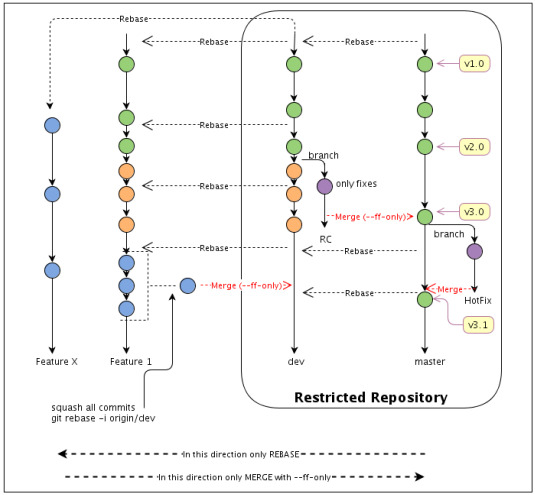
A git *pull* is a shortcut for a *fetch* and *merge*.

Definition

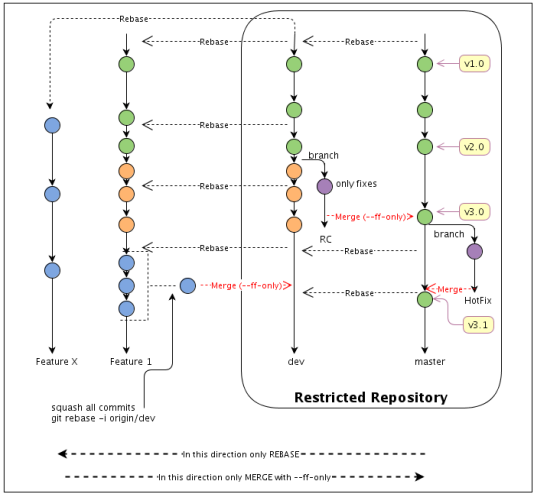
A *pull request* is the managed process of a git merge (usually via a webbased software tool).

- Asynchronous
- Distributed
- Usually combined with a review process

FairRoot git workflow



FairRoot git workflow

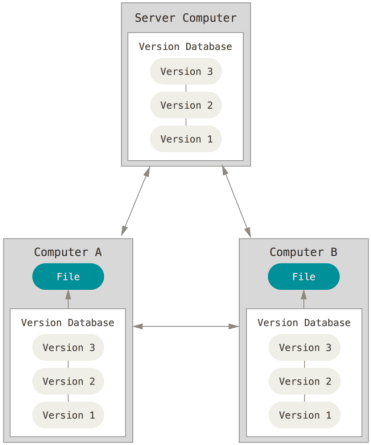


https://github.com/AnarManafov/GitWorkflow

Rebase

Golden Rule of rebase

"No one shall rebase a shared branch"—*Everyone about rebase*



Exercises

1. Hosting a git repo on Gitlab.org
2. Resolve merge conflicts
3. Interactive rebase (`git rebase -i`)

Material

- `https://git-scm.com`
- Progit2 - `https://git-scm.com/book`
- `man git`

Attribution