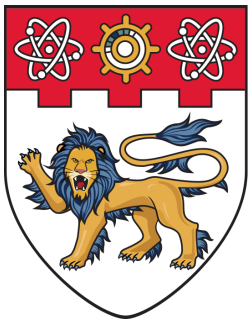

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CI6206 Technology Research Paper

Source code version control and social code sharing

NAME: XUANRAN HE
EMAIL: xhe015@e.ntu.edu.sg

Wee Kim Wee School of Communication and Information

October, 2020

Abstract

Version control systems are essential for co-ordinating work on a software project. A number of open- and closedsource projects are proposing to move, or have already moved, their source code repositories from a centralized version control system (CVCS) to a decentralized version control system (DVCS). In this paper we summarize the differences between a CVCS and a DVCS, and describe some of the rationales and perceived benefits offered by projects to justify the transition.We also provide examples of the use of DVCS to help network programming teams make better use of version control systems in real-world projects.

Contents

Chapter 1

Introdution

1.1 Development of version control systems

The central challenge in managing software development is scaling the change process up to large numbers of possibly geographically-distributed software developers without sacrificing quality or introducing undue overhead. That is why version control systems came into being.

Broadly speaking, the history of version control tools can be divided into three generations

The forty year history of version control tools shows a steady movement toward more concurrency.

- In first generation tools, concurrent development was handled solely with locks. Only one person could be working on a file at a time.
- The second generation tools are a fair bit more permissive about simultaneous modifications, with one notable restriction. Users must merge the current revisions into their work before they are allowed to commit.
- The third generation tools allow merge and commit to be separated.

As I write this in mid-2020, the world of version control is in a time of stable stage. The vast majority of professional programmers are using second generation tools but the third generation is growing very quickly in popularity.

Generation	Networking	Operations	Concurrency	Examples
First	None	One file at a time	Locks	RCS, SCCS
Second	Centralized	Multi-file	Merge before commit	CVS, SourceSafe,Subversion
Third	Distributed	Changesets	Commit before merge	Bazaar, Git, Mercurial

The most popular VCS on Earth is Apache Subversion, an open source second generation tool. The high-end of the commercial market is dominated by IBM and Microsoft, both of which are firmly entrenched in second generation tools. But at the community level, where developers around the world talk about what's new and cool, the buzz is all about Distributed Version Control Systems (DVCS). The three most popular DVCS tools are Bazaar Git and Mercurial.But Git is used the most.

1.2 Importance of version control

In a team-based software development,it is necessary to track changes and different code versions. That is why version control system(VCS) is so important.First and foremost, they should retain a long-term history of changes made on a project including creation, deletion, edits and more. This should include the author, date and any notes from the changes as well.Beyond that, they should have a solution for branching and merging new code changes to the main project to allow concurrent work from multiple members of a team. In my opinion,there are 3 reasons why VCS is Critical for team-based software development.

- Version control is a method of tracking changes to documents and files so that you always know which version is the current iteration. It also enables you to maintain old versions in case you want to see what's changed or need to restore a previous version.
- Projects typically result in the creation of a lot of documents, from project reports to deliverables. By using project management software with version control, or version management, you can effectively and efficiently track and control changes to these documents directly within your software.
- Version control can also apply to other files, such as videos and images, as well as software and any other deliverables that have multiple iterations.

Chapter 2

Effective tools of version control

2.1 Centralized VCSs

2.1.1 Work Principle

A centralized version control system works on a client-server model. There is a single, (centralized) master copy of the code base, and pieces of the code that are being worked on are typically locked, (or "checked out") so that only one developer is allowed to work on that part of the code at any one time. Access to the code base and the locking is controlled by the server. When the developer checks their code back in, the lock is released so it's available for others to check out.

These VCSs are centralized as they have a single canonical source repository. All developers work against this repository through a checkout taken from the repository, essentially a snapshot at some moment in time.Write-access to a group's repository is generally restricted to a set of known developers, or committers.The details of workflow can be seen in Figure 2.1.

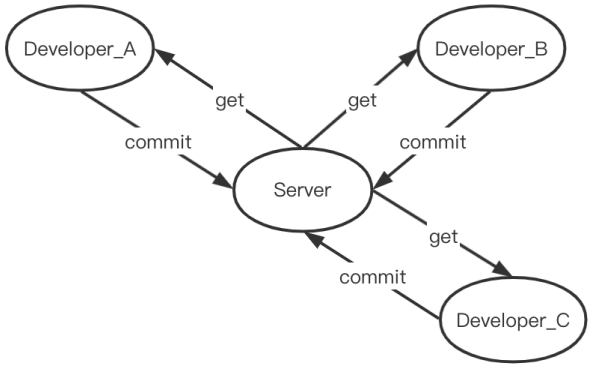

Figure 2.1: Workflow of Centralized VCS

In open-source situations, committers generally earn and maintain write-access through the submission of highquality patches to demonstrate programming prowess and understanding of the project development conventions.Teams generally establish coding conventions and practices to control how changes are made to the repository to preserve the quality of the source code.Modern VCSs support parallel evolution of a repository's contents through the use of branches. One wide-spread practice is to maintain a mainline branch to represent the current development efforts, and creating new branches of the mainline to represent released versions of the product and track bug fixes to the released product.Branches are also often used for undertaking a substantial change of some long duration, with the goal of being merged back into the mainline.

2.1.2 Advantage and Disadvantage

Some advantage of Centralized Version Control Systems:

- **Easy to understand** Centralized version control system has clear logic, only one Server makes it has high code consistency, suitable for small development team project development management convenient, in line with common people's habit of thinking.
- **High safety** Centralized version control system requires authorization When the code is submitted, if there is no write permission, the transaction fails, ensuring the security of the code base.
- **Rigorous Authority** The centralized version control system adopts folder level permission control with small granularity of permission control.Hierarchical management can be achieved.
- **Easy to use** Centralized version control systems require little in the way of client configuration and do not require the storage of a full set of code.

Some disadvantage of Centralized Version Control Systems:

- **Rely on the network** Network environment is demanding, relevant personnel must depend on network to commit the branch.Once the network delay, no developer can commit the branch.
- **Rely on the server** A single point of failure on a central server affects the whole world, and if the server goes down, no one can work.

- **Unfriendly to open source project** In an open source project, the large number of people involved in the development is a huge challenge to the authority management of a centralized version control system and a burden for administrators.

2.2 Decentralized VCSs

2.2.1 What is Decentralized VCSs

With a DVCS, each checkout is itself a first-class repository in its own right, a copy containing the complete commit history. Write-access is no longer an issue as each developer is a first-class committer to their personal repository, regardless of whether they are an accepted committer to the project.Because each DVCS repository is a full-fledged repository, there are no enforced master branches.Instead a canonical branch is identified by convention within a development group or community, and some projects may have several principal branches.
For example, the Linux kernel's authoritative branch is Linus Torvalds' branch, but there are also several other important branches, such as Andrew Morton's -mm branch, that are used for staging and evaluating proposed changes to the kernel. Once the patches in these intermediate branches have been stabilized and proved their value,Torvalds will select the result to merge to his master tree.
Since the DVCS repositories contain all the revision history, they lend themselves very well to distributed and disconnected development.
Reliable Software needed to cater to teams developing across distributed locations, where the latency to access a central repository was prohibitive. Using a DVCS naturally leads to the source code repository being replicated to a number of places, a side-benefit that reduces the risk of some disaster scenarios.
DVCSs maintain sufficient information to support easy branching and easy, repeated merging of branches.The ease of branching has encouraged a practice called feature branches, where every prospective change is done within a branch and then merged into the mainline, rather than being directly developed against the mainline.
The details of workflow can be seen in Figure 2.2.

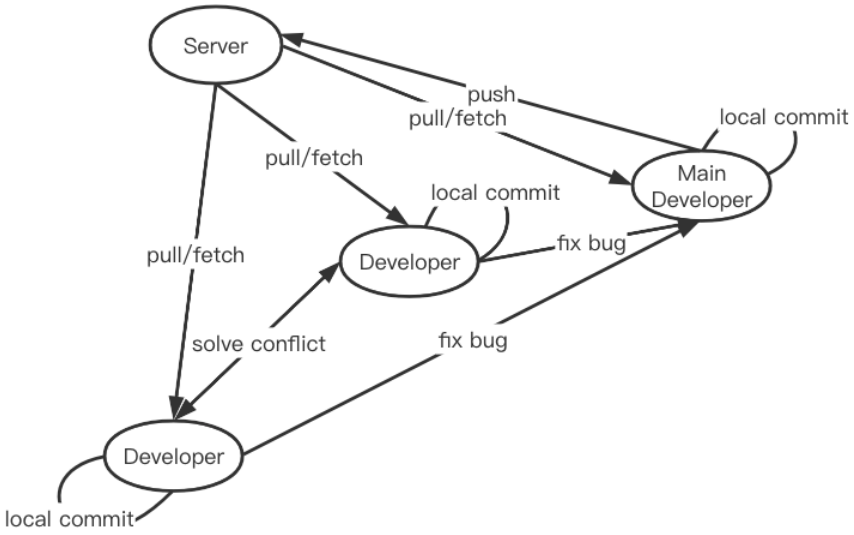

Figure 2.2: Workflow of Centralized DVCS

2.2.2 Work Principle of Git

By far, the most widely used modern version control system in the world today is Git which has a distributed architecture. In Git, every developer's working copy of the code is also a repository that can contain the full history of all changes.In git,everything can be considered as an object.The commit object can be seen as a snapshot of the git repository which contains all the files of this project at the same time.There are two repositories one is local and another is remote.The workplace is a local area of the developer,by checkout different branches,the developer can choose the most suitable workplace.The details of working principle can be seen in Figure 2.3.

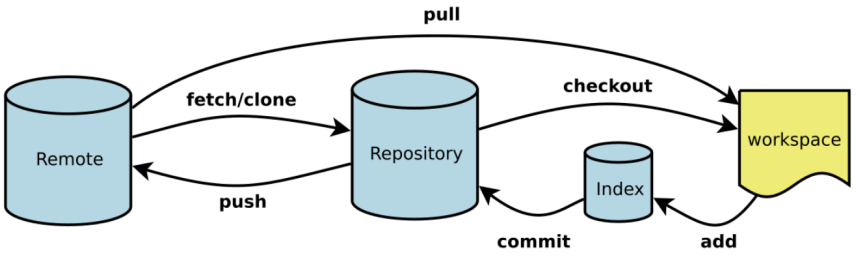

Figure 2.3: Workflow of Centralized DVCS

Conception	Comment
Workspace	The work place for developers
Index / Stage	Save the changes for a moment
Repository	Local repository for developers
Remote	Push commit to remote

2.2.3 Advantage and Disadvantage

Some advantage of Decentralized Version Control Systems:

- **High Performance** In distributed development, You can clone a local version of Git, and then commit the operation locally. You can complete the version control locally, no need to rely on the network, and you can use Git to push to the remote location when publishing.
- **Flexibility** Git is flexible in several respects: in support for various kinds of nonlinear development workflows, in its efficiency in both small and large projects and in its compatibility with many existing systems and protocols.
- **Conflict resolution** Conflicts can easily occur in multi-person development, and git has the advantage of resolving conflicts. Developers can first pull remote branch to the local, and then merge branches locally to resolve conflicts, and then push together to the remote.
- **Do not rely on Server** In Git,everyone has complete code in their repository, it's not too stressful or heavy on the public server.
- **Friendly to Open Source** Git emphasizes individual contributions and is easy to manage for open source projects.

Some disadvantage of Decentralized Version Control Systems:

- **Hard to learn** Git does not conform to conventional thinking, plus the need to operate on the shell, the learning threshold is high.
- **Bad Secrecy** In Git, you pull all the project code from the repository, so keeping it secret is hard for git projects.
- **Low permission controls** In Git, there is no strict control over permissions, and all developers merge into the master branch through a unified

review process. The roles of each branch are agreed upon and adhered to by the developers in the project.

Chapter 3

Use of git

3.1 Objects in Git

In the Chapter2,we mentioned that everything in git can be seen as an object.There are four objects in git: blob,tree,commit,tag.

- **Blob** Used to store file data, usually a file.
- **Tree** Tree solves the problem of storing the filename and also allows you to store a group of files together. Git stores content in a manner similar to a UNIX filesystem, but a bit simplified. All the content is stored as tree and blob objects, with trees corresponding to UNIX directory entries and blobs corresponding more or less to inodes or file contents. A single tree object contains one or more entries, each of which is the SHA-1 hash of a blob or subtree with its associated mode, type, and filename. For example, the most recent tree in a project may look something like this:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152     README
100644 blob 8f94139338f9404f2     Rakefile
040000 tree 99f1a6d12cb4b6f19     lib
```

  The master $\hat{\{}$tree$\}$ syntax specifies the tree object that is pointed to by the last commit on your master branch.

- **Commit** A "commit" only points to a "tree", which is used to mark the status of the project at a particular point in time. This includes metadata on time, such as timestamps, the author of the most recent submission, Pointers explaining why previous submissions were submitted, etc.
- **Tag** A "tag" is a method to mark a commit.

The connection among the four objects can be seen in the Figure 3.1.

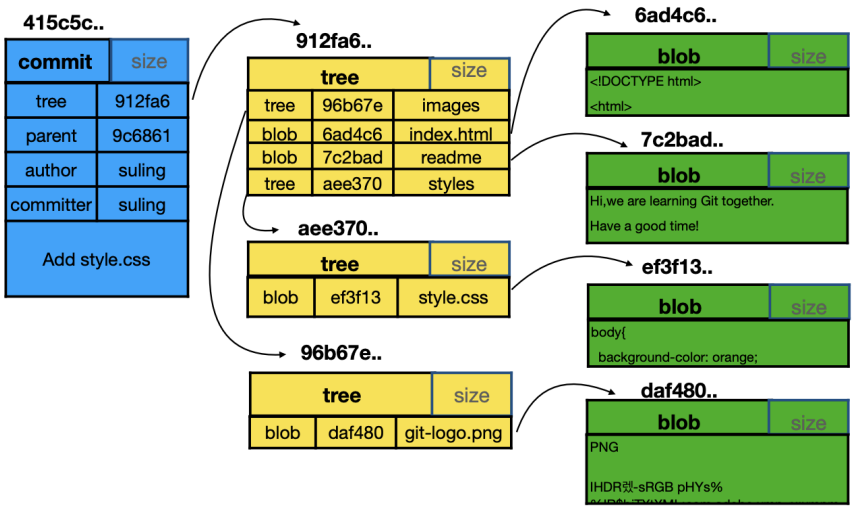

Figure 3.1: Workflow of Centralized VCS

3.2 Workflow of git

3.2.1 Start a git project

Initializing a Repository in an Existing Directory

```
$ git init # Getting a Git Repository
$ git add *.c  # Save files to index/stage
$ git add LICENSE # Save files to index/stage
$ git commit -m 'version' # Create a snapshot
```

Cloning an Existing Repository

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

3.2.2 Status of your files

Remember that each file in your working directory can be in one of two states: tracked or untracked. Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about. Untracked files are everything else—any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository,all of your files

will be tracked and unmodified because Git just checked them out and you haven't edited anything. As you edit files, Git sees them as modified, because you've changed them since your last commit. As you work, you selectively stage these modified files and then commit all those staged changes, and the cycle repeats. In a word,there are four status of files: Untracted, Unmodified, Modefied, Staged.The connection of the four status can be seen in Figure3.2.

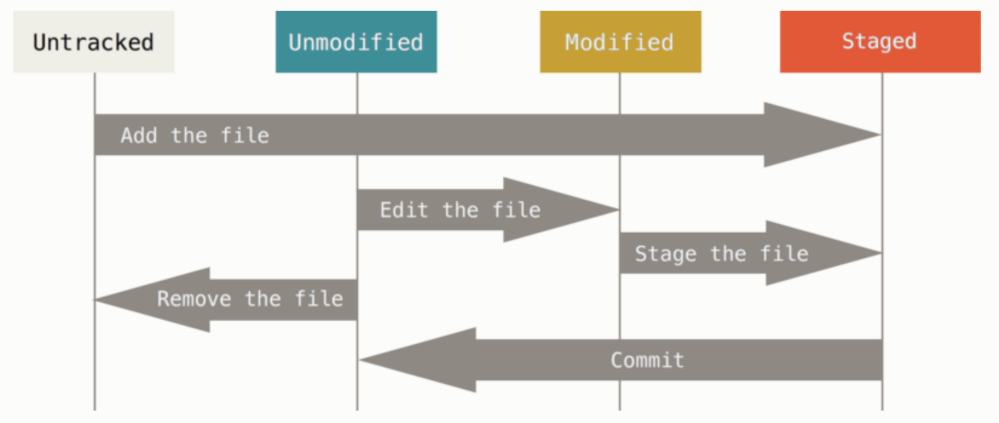

Figure 3.2: Two branches pointing into the same series of commits

The main tool you use to determine which files are in which state is the **git status** command. If you run this command directly after a clone, you should see something like this:

```
$ git status # View the status of all files
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
(use git push to publish your local commits)
nothing to commit, working tree clean
```

3.2.3 View history of commit

By default, with no arguments, git log lists the commits made in that repository in reverse chronological order; that is, the most recent commits show up first. As you can see, this command lists each commit with its SHA-1 checksum, the author's name and email, the date written, and the commit message.

```
$ git log
```

```
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

3.2.4 Branches in git

Branching means you diverge from the main line of development and continue to do work without messing with that main line. In many VCS tools, this is a somewhat expensive process, often requiring you to create a new copy of your source code directory, which can take a long time for large projects.
A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is master. As you start making commits, you're given a master branch that points to the last commit you made. Every time you commit, the master branch pointer moves forward automatically.
If you want to create a new branch named testing,you can use [git branch] command.This creates a new pointer to the same commit you're currently on.The current architecture can be seen in Figure 3.3.

```
$ git branch testing
```

Git use a special pointer called HEAD to tag the branch you are currently on. To switch to an existing branch, you run the git checkout command. Let's

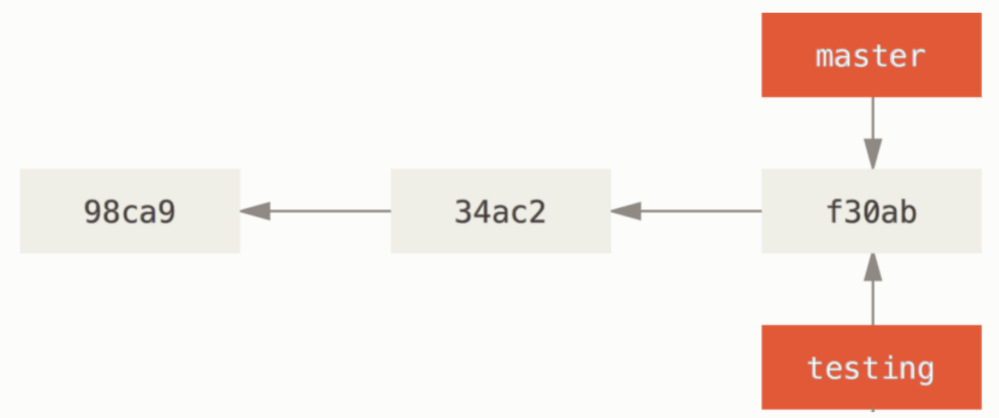

Figure 3.3: Two branches pointing into the same series of commits

switch to the new testing branch:

```
$ git checkout testing
```

Figure 3.4: HEAD points to the current branch

3.3 Basic commands

1. To synchronize your work with a given remote, you run a git fetch ¡remote¿ command (in our case, git fetch origin). This command looks up which server "origin" is (in this case, it's git.ourcompany.com), fetches any data from it that you don't yet have, and updates your local database, moving your

origin/master pointer to its new, more up-to-date position. Run git fetch teamone to fetch everything the remote teamone server has that you don't have yet.

```
$ git fetch teamone
```

2. If you want to make the history of git visualization,you can run gitk to see the graphical interface.

```
$ gitk
```

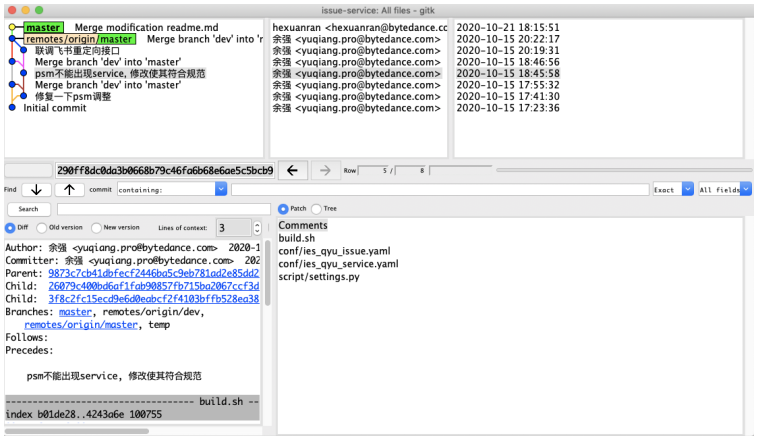

Figure 3.5: Graphical interface

3. With the rebase command, you can take all the changes that were committed on one branch and replay them on a different branch.

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of
   it...
Applying: added staged command
```

4. It often happens that while working on one project, you need to use another project from within it. Perhaps it's a library that a third party developed or that you're developing separately and using in multiple parent projects. A common issue arises in these scenarios: you want to be able to treat the two projects as separate yet still be able to use one from within the other.

```
$ git submodule add https://github.com/chaconinc/
   DbConnector
Cloning into 'DbConnector'...
```

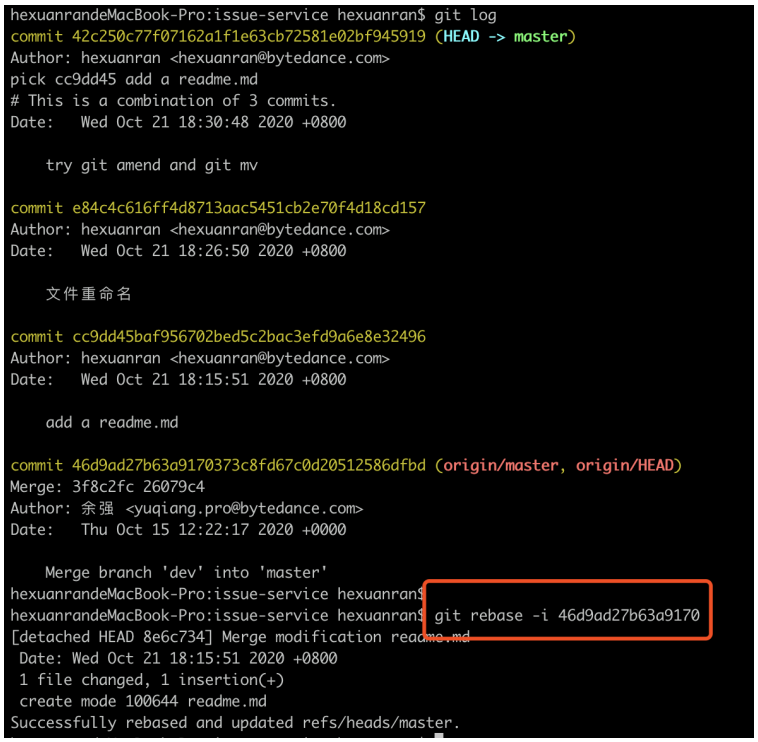

Figure 3.6: Graphical interface

```
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

5. If you want to compare branching differences,you can use git diff.For example if you want to compare Head and the father of Head,you can use command below.

```
$ git diff HEAD HEAD~1
```

6. In our work, we often need to remove useless branches, in which case, we can use git branch -d/D.For example,if we want to delete 'test' branch.We can use the command below:

```
git branch -d test
```

3.4 Git Internals

When you run git init in a new or existing directory, Git creates the .git directory, which is where almost everything that Git stores and manipulates is located. If you want to back up or clone your repository, copying this single directory elsewhere gives you nearly everything you need. This entire chapter basically deals with what you can see in this directory.The core of Git is a simple key-value data store. Here's what a newly-initialized .git directory typically looks like:

```
config # configuration
description
HEAD # Latest COMMIT to the current branch
inedx # Save information for the staging area
hooks #
info
exclude
objects #The object file type is tree
info
pack
refs # The pointer type for the branch and label is
  COMMIT
heads
tags
remotes
```

Chapter 4

Conclusion

Version control plays a vital role in team collaboration, not only improving team development efficiency, but also reducing the frequency of failures.

Distributed version control systems, such as Git, are more recommended in large team and open source projects. Centralized control systems are recommended for projects with small teams that require high levels of secrecy.

DVCSs have captured a large mindshare, and many projects are at least debating the merits of transitioning their code repositories to a DVCS. Our study has provided some insights into the limitations of CVCSs and the consequences those limitations have for development teams.Although DVCSs address some of the problems of CVCSs, particularly the difficulty in repeated merges of branches, our findings suggest that DVCS may also introduce new issues. require high levels of secrecy.

In the actual network project development process, we need to combine our own requirements and comprehensively consider the advantages and disadvantages of CVCS and DVCS to make a choice.

Chapter 5

References

[1] http://www.catb.org/ esr/writings/version-control/version-control.html — I don't remember for sure. I may have gotten this notion of three generations from Eric Raymond's "Understanding Version-Control Systems". Either way, it's a good read.

[2] https://medium.com/weareservian/importance-of-version-control-and-why-you-need-it-aae53dac208a - Importance of Version Control and Why You Need It

[3] https://git-scm.com/book/en/v2/Git-Internals-Git-Objects - Pro Git

[4] Hammack S G, Jundt L O, Lucas J M, et al. Version control and audit trail in a process control system: U.S. Patent 6,449,624[P]. 2002-9-10.

[5] Haikin J S. Version control system for software code: U.S. Patent 6,757,893[P]. 2004-6-29.

[6] B. Milewski. Distributed source control system. In Proc. ICSE Worksh. on System Configuration Management (SCM-7), pages 98–107, 1997