Lab Report for Software Engineering course
Lab 1: Git & Dev Cloud

Wang, Chen
16307110064
School of Software
Fudan University

March 9, 2019

Contents

Chapter 1

Background Knowledge of the lab

1.1 Version Control System

1.1.1 The application of VCS

A component of software configuration management, version control, also known as revision control or source control, is the management of changes to documents, computer programs, large web sites, and other collections of information. Changes are usually identified by a number or letter code, termed the "revision number", "revision level", or simply "revision". For example, an initial set of files is "revision 1". When the first change is made, the resulting set is "revision 2", and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

The need for a logical way to organize and control revisions has existed for almost as long as writing has existed, but revision control became much more important, and complicated when the era of computing began. The numbering of book editions and of specification revisions are examples that date back to the print-only era. Today, the most capable (as well as complex) revision control systems are those used in software development, where a team of people may concurrently make changes to the same files.

Version control systems (VCS) most commonly run as stand-alone applications, but revision control is also embedded in various types of software such as word processors and spreadsheets, collaborative web docs and in various content management systems, e.g., Wikipedia's page history. Revision control allows for the ability to revert a document to a previous revision, which is critical for allowing editors to track each other's edits, correct mistakes, and defend against vandalism and spamming in wikis.

1.1.2 The general usage of VCS

In computer software engineering, revision control is any kind of practice that tracks and provides control over changes to source code. Software developers

sometimes use revision control software to maintain documentation and configuration files as well as source code.

As teams design, develop and deploy software, it is common for multiple versions of the same software to be deployed in different sites and for the software's developers to be working simultaneously on updates. Bugs or features of the software are often only present in certain versions (because of the fixing of some problems and the introduction of others as the program develops). Therefore, for the purposes of locating and fixing bugs, it is vitally important to be able to retrieve and run different versions of the software to determine in which version(s) the problem occurs. It may also be necessary to develop two versions of the software concurrently: for instance, where one version has bugs fixed, but no new features (branch), while the other version is where new features are worked on (trunk).

At the simplest level, developers could simply retain multiple copies of the different versions of the program, and label them appropriately. This simple approach has been used in many large software projects. While this method can work, it is inefficient as many near-identical copies of the program have to be maintained. This requires a lot of self-discipline on the part of developers and often leads to mistakes. Since the code base is the same, it also requires granting read-write-execute permission to a set of developers, and this adds the pressure of someone managing permissions so that the code base is not compromised, which adds more complexity. Consequently, systems to automate some or all of the revision control process have been developed. This ensures that the majority of management of version control steps is hidden behind the scenes.

Moreover, in software development, legal and business practice and other environments, it has become increasingly common for a single document or snippet of code to be edited by a team, the members of which may be geographically dispersed and may pursue different and even contrary interests. Sophisticated revision control that tracks and accounts for ownership of changes to documents and code may be extremely helpful or even indispensable in such situations.

Revision control may also track changes to configuration files, such as those typically stored in /etc or /usr/local/etc on Unix systems. This gives system administrators another way to easily track changes made and a way to roll back to earlier versions should the need arise.

1.1.3 Various tools for VCS

Differences between distributive and non-distributive VCS systems

Distributed version control systems (DVCS) takes a peer-to-peer approach to version control, as opposed to the client–server approach of centralized systems. Distributed revision control synchronizes repositories by exchanging patches from peer to peer. There is no single central version of the codebase; instead, each user has a working copy and the full change history.

Advantages of DVCS (compared with centralized systems) include:

1. Allows users to work productively when not connected to a network.
2. Common operations (such as commits, viewing history, and reverting changes) are faster for DVCS, because there is no need to communicate

with a central server. With DVCS, communication is only necessary when sharing changes among other peers.

3. Allows private work, so users can use their changes even for early drafts they do not want to publish.
4. Working copies effectively function as remote backups, which avoids relying on one physical machine as a single point of failure.
5. Allows various development models to be used, such as using development branches or a Commander/Lieutenant model.
6. Permits centralized control of the "release version" of the project
7. On FOSS software projects it is much easier to create a project fork from a project that is stalled because of leadership conflicts or design disagreements.

Disadvantages of DVCS (compared with centralized systems) include:

1. Initial checkout of a repository is slower as compared to checkout in a centralized version control system, because all branches and revision history are copied to the local machine by default.
2. The lack of locking mechanisms that is part of most centralized VCS and still plays an important role when it comes to non-mergeable binary files such as graphic assets.
3. Additional storage required for every user to have a complete copy of the complete codebase history.

Some originally centralized systems now offer some distributed features. For example, Subversion is able to do many operations with no network. Team Foundation Server and Visual Studio Team Services now host centralized and distributed version control repositories via hosting Git.

1.2 Git

1.2.1 The feature of Git in comparison to other VCS tools

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. In Git, a core assumption is that a change will be merged more often than it is written, as it is passed around to various reviewers. In Git, branches are very lightweight: a branch is only a reference to one commit. With its parental commits, the full branch structure can be constructed.

Distributed development

Like Darcs, BitKeeper, Mercurial, SVK, Bazaar, and Monotone, Git gives each developer a local copy of the full development history, and changes are copied from one such repository to another. These changes are imported as added development branches and can be merged in the same way as a locally developed branch.

Compatibility with existent systems and protocols

Repositories can be published via Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), rsync (removed in Git 2.8.0), or a Git protocol over either a plain socket, or Secure Shell (ssh). Git also has a CVS server emulation, which enables the use of extant CVS clients and IDE plugins to access Git repositories. Subversion and svk repositories can be used directly with git-svn.

Efficient handling of large projects

Torvalds has described Git as being very fast and scalable, and performance tests done by Mozilla showed that it was an order of magnitude faster than some version-control systems; 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. The structure is similar to a Merkle tree, but with added data at the nodes and leaves. (Mercurial and Monotone also have this property.)

Toolkit-based design

Git was designed as a set of programs written in C and several 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.

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 needed.

Garbage accumulates until 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 –prune.

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 delta-compressed among

themselves in one file (or network byte stream) called a packfile. Packs are compressed using the heuristic that files with the same name are probably similar, but do not depend on it for correctness. A corresponding index file is created for each packfile, telling the offset of each object in the packfile. Newly created objects (with newly added history) are still stored as single objects, and periodic repacking is needed to maintain space efficiency. The process of packing the repository can be very computationally costly. By allowing objects to exist in the repository in a loose but quickly generated format, Git allows the costly pack operation to be deferred until later, when time matters less, e.g., the end of a work day. Git does periodic repacking automatically, but manual repacking is also possible with the git gc command. For data integrity, both the packfile and its index have an SHA-1 checksum inside, and the file name of the packfile also contains an SHA-1 checksum. To check the integrity of a repository, run the git fsck command.

1.2.2 The application of Git

Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Its goals include speed, data integrity, and support for distributed, non-linear workflows. Git was created by Linus Torvalds in 2005 for development of the Linux kernel, with other kernel developers contributing to its initial development. Its current maintainer since 2005 is Junio Hamano. As with most other distributed version-control systems, and unlike most client–server systems, every Git directory on every computer is a full-fledged repository with complete history and full version-tracking abilities, independent of network access or a central server. Git is free and open-source software distributed under the terms of the GNU General Public License version 2.

1.2.3 The general usage of Git

Working with local repositories

git init This command turns a directory into an empty Git repository. This is the first step in creating a repository. After running git init, adding and committing files/directories is possible. Usage:

```
# change directory to codebase                    1
$ cd /file/path/to/code                           2
                                                  3
# make directory a git repository                 4
$ git init                                        5
```

git add Adds files in the to the staging area for Git. Before a file is available to commit to a repository, the file needs to be added to the Git index (staging area). There are a few different ways to use git add, by adding entire directories, specific files, or all unstaged files. Usage:

```
$ git add <file or directory name>                1
```

git commit Record the changes made to the files to a local repository. For easy reference, each commit has a unique ID. It's best practice to include a message with each commit explaining the changes made in a commit. Adding a commit message helps to find a particular change or understanding the changes. Usage:

```
# Adding a commit with message
$ git commit -m "Commit message in quotes"
```

git status This command returns the current state of the repository. *git status* will return the current working branch. If a file is in the staging area, but not committed, it shows with *git status*. Or, if there are no changes it'll return *nothing to commit, working directory clean*. Usage:

```
$ git status
```

git config With Git, there are many configurations and settings possible. *git config* is how to assign these settings. Two important settings are user user.name and user.email. These values set what email address and name commits will be from on a local computer. With *git config*, a *–global* flag is used to write the settings to all repositories on a computer. Without a *–global* flag settings will only apply to the current repository that you are currently in. There are many other variables available to edit in *git config*. From editing color outputs to changing the behavior of *git status*. Usage:

```
$ git config <setting> <command>
```

git branch To determine what branch the local repository is on, add a new branch, or delete a branch. Usage:

```
# Create a new branch
$ git branch <branch_name>

# List all remote or local branches
$ git branch -a

# Delete a branch
$ git branch -d <branch_name>
```

git checkout To start working in a different branch, use *git checkout* to switch branches. Usage:

```
# Checkout an existing branch
$ git checkout <branch_name>

# Checkout and create a new branch with that name
$ git checkout -b <new_branch>
```

git merge Integrate branches together. *git merge* combines the changes from one branch to another branch. For example, merge the changes made in a staging branch into the stable branch. Usage:

```
# Merge changes into current branch
$ git merge <branch_name>
```

Working with remote repositories

git remote To connect a local repository with a remote repository. A remote repository can have a name set to avoid having to remember the URL of the repository. Usage:

```
# Add remote repository
$ git remote <command> <remote_name> <remote_URL>

# List named remote repositories
$ git remote -v
```

git clone To create a local working copy of an existing remote repository, use *git clone* to copy and download the repository to a computer. Cloning is the equivalent of *git init* when working with a remote repository. Git will create a directory locally with all files and repository history. Usage:

```
$ git clone <remote_URL>
```

git pull To get the latest version of a repository run *git pull*. This pulls the changes from the remote repository to the local computer. Usage:

```
$ git pull <branch_name> <remote_URL/remote_name>
```

git push Sends local commits to the remote repository. *git push* requires two parameters: the remote repository and the branch that the push is for. Usage:

```
$ git push <remote_URL/remote_name> <branch>

# Push all local branches to remote repository
$ git push -all
```

1.2.4 The usage of Git in this lab

AAAW

1.3 Java EE

1.3.1 The needs and application of Java EE

1.3.2 Common frameworks of Java EE

1.4 Spring Boot

1.4.1 The feature of Spring Boot in comparison to other Java EE frameworks

1.4.2 The usage of Spring Boot in this lab

1.5 Classroom platform of Huawei Cloud

1.5.1 The functionalities of Huawei Cloud

1.5.2 The functions of Huawei Cloud used in this lab

1.6 Life cycle in software engineering

1.6.1 The meaning of life cycle in software engineering

1.6.2 The life cycle displayed in this lab

AAAW

Chapter 2

Specification of the Lab

2.1 Platform of the operation

2.2 Guideline of the operation steps

2.3 Hand in method and materials

Chapter 3

Steps of accomplishing this Lab

3.1 Git operations

3.2 SpringBoot framework construction

3.3 Executions on the SpringBoot framework

Chapter 4

Significance of different parts of the lab

Chapter 5

Conclusion