GIT and GIT Remote Hosting

How to Get Started

TABLE OF CONTENTS (HIDE)

1. Introduction

GIT is a Version Control System (VCS) (aka Revision Control System (RCS), Source Code Manager (SCM)). A VCS serves as a *Repository* (or *repo*) of program codes, including all the historical revisions. It records all changes to files over time in a *log* such that you can recall a specific version.

Why VCS?

1. The Repository serves as the backup (in case of code changes or disk crash), or a living archive of all works (all historical revisions). It lets you revert back to a specific version, if the need arises.
2. It facilitates collaboration between team members, and serves as project management tool.
3. more...

Git was initially designed and developed by Linus Torvalds, in 2005, to support the development of the Linux kernel.

GIT is a *Distributed Version Control System* (DVCS). Other popular VCSes include:

1. The standalone and legacy Unix's RCS (Revision Control System).
2. Centralized Client-Server Version Control System (CVCS): CVS (Concurrent Version System), SVN (Subversion) and Perforce.
3. Distributed VCS (DVCS)): GIT, Merurial, Bazaar, Darcs.

The mother site for Git is http://git-scm.com.

2. Setting Up Git on your Local Machine

You need to setup Git on your local machine, as follows:

1. Download: Download the latest Git for your operating platform from http://git-scm.com/downloads (e.g., `Git-1.8.0-preview20121022.exe`).
2. Install: Run the downloaded installer to install Git.
3. Customize Git:
   For Windows, run "Git Bash" from the Git installed directory, and issue "`git config`" command:

```
// Set up your username (to properly label your commits)
$ git config --global user.name "Your Name Here"
// Set up your email
$ git config --global user.email "your_email@youremail.com"
// Cache username/password for default of 15 minutes
$ git config --global credential.helper cache
```

```
// Change the username/password cache time-out to 1 hour
$ git config --global credential.helper 'cache --timeout=3600'
```

The settings are kept in "<GIT_HOME>/etc/gitconfig" (of the GIT installed directory) and "<USER_HOME>/.gitconfig" (of the user's home directory. You can issue "git config --list" to list the settings:

```
$ git config --list
......
```

3. Git Basics

Help and Manual

The best way to get help these days is certainly *googling*.

To get help on Git commands:

```
$ git help <command>
// or
$ git <command> --help
```

The GIT manual is bundled with the software (under the "doc" directory), and also available online @ http://git-scm.com/docs.

Git Commands

Git provides a set of simple, distinct, standalone sub-commands developed according to the "Unix toolkit" philosophy - build small, interoperable tools.

The commonly-used commands include:

1. **init**: initialize Git repo.
2. **add**, **commit**, **reset**, **checkout**:
3. **status**, **log**, **diff**, **grep**, **show**:
4. **clone**, **branch**, **merge**, **push**, **fetch**, **pull**:
5. **tag**, **rebase**:
6. **mv**, **rm**: move (rename), or remove files.

3.1 Getting Started with Local Repo

Creating a new Git Repo for a Project (`git init`)

Suppose that we have started a project under the directory call "git_test", with one source file "Hello.java" (or "Hello.cpp", or "Hello.c"). The source file has been compiled into "Hello.class" (or "Hello.o" and "Hello.exe"). It is also highly recommended to provide a "README.md" file to describe your project.

The initial contents of the files are as follows:

```
// Hello.java
public class Hello {
   public static void main(String[] args) {
      System.out.println("Hello, world from GIT!");
   }
}
```

Compile the "Hello.java" into "Hello.class".

```
// README.md
This is the README file for the Hello-world project.
```

To turn a project into a Git repo, run "`git init`" command (via "Git Bash" shell) at the base of the project directory:

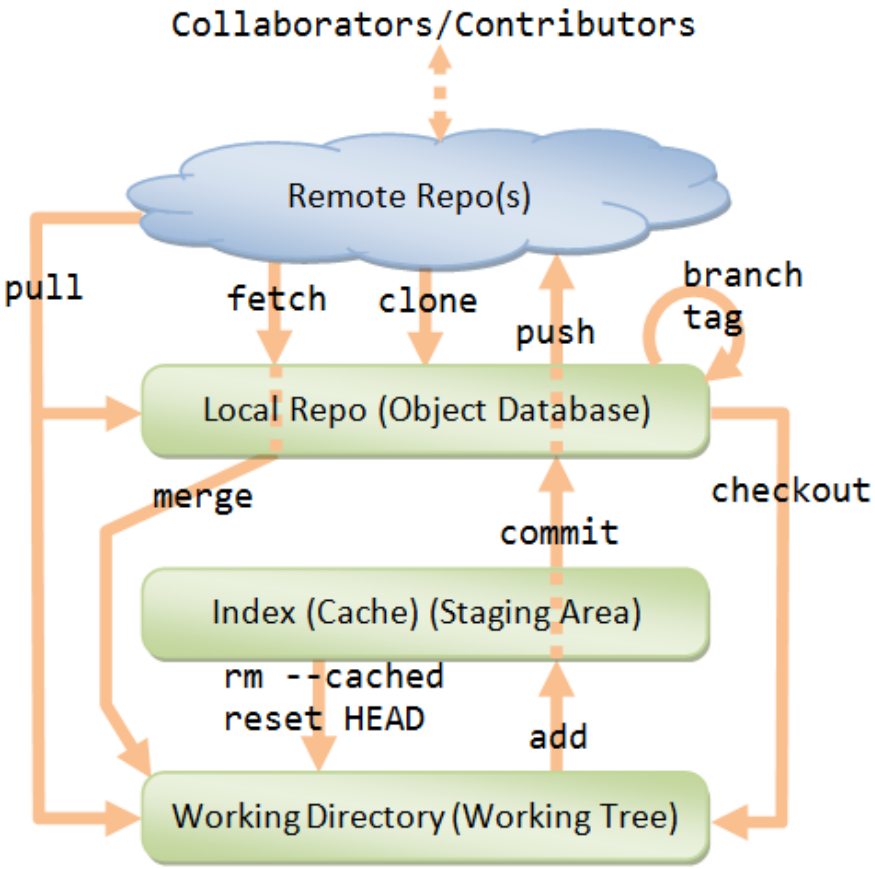
```
$ cd /path/to/git_test  // Change directory to the project directory
$ git init              // Initialize Git repo for this project
Initialized empty Git repository in path-to/git_test/.git/
```

A hidden sub-directory called "`.git`" will be created under your project directory.

Take note that each Git repo is associated with a project directory (and its sub-directories). The Git repo is completely contain within the project directory. Hence, it is safe to copy, move or rename the project directory. If your project uses more than one directories, you need to create one Git repo for each directory (or use symlinks to link up the directories).

Git Storage Levels

Now, we have created the "Local Repo" and having a working tree for the project `git_test`. Like all VCS, the repo retains a complete copy of the project throughout its life time, including all the revisions.



The local repo after "`git init`" is empty. You need to explicitly deposit files into the repo.

Adding Files for Tracking (`git add <file>`)

Let's issue a "`git status`" command to check the status of the files in the working tree:

```
$ git status
# On branch master
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#       Hello.class
#       Hello.java
#       README.md
nothing added to commit but untracked files present (use "git add" to track)
```

In Git, the files in the working tree is either *untracked* or *tracked*. Currently, all 3 files are *untracked*.

To add files for tracking, use "`git add <file>`" command.

```
// Add README.md file into Git repo
```

```
$ git add README.md

// You can use wildcard * in the filename
// Add all Java source files into Git repo
$ git add *.java
```

The command "`git add <file>`" takes a filename or pathname pattern with possibly wildcards. You can also use "`git add .`" to add all the files in the current directory (and all sub-directories) to the Git repo. But this will include "`Hello.class`", which we do not wish to be tracked.

When a file is added , it is *staged* (or *indexed*, or *cached*) in the *staging area*, rather than *committed*. Git provides two commands: "`git add`" to stage a file, and "`git commit`" to commit the file. You can check the status via "`git status`" command:

```
$ git status
# On branch master
# Initial commit
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#       new file:   Hello.java
#       new file:   README.md
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#       Hello.class
```

Recall that the files in the project directory are either *tracked* or *untracked*. In the example, the "`Hello.class`" is *untracked*. You need to issue "`git add`" command to track a file (e.g., "`Hello.java`" and "`README.md`"). For tracked files, it could be either *unmodified*, *modified*, *staged*, or *committed*.

Committing Files (`git commit`)

To commit all the files in the staging area, use "`git commit`" command, with a `-m` option to provide a message for the commit:

```
$ git commit -m "First commit"   // -m to specify the commit message
[master (root-commit) aa27391] first commit
 2 files changed, 8 insertions(+)
 create mode 100644 Hello.java
 create mode 100644 README.md

// Check the stutus
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#       Hello.class
nothing added to commit but untracked files present (use "git add" to track)
```

Viewing the Commit Information (`git log`) and Patches (`git log -p`)

Git records several pieces of metadata for every commit, which includes a log message, timestamp, the author's username and email (that have been set during customization). Every object in Git, including commit, is identified by a 40-hex-digit SHA-1 hash code.

You can use "`git log`" to list the commit information; or "`git log --stat`" to view the file statistics, as follows:

```
$ git log
commit aa27391512e39329edd89b14c52db741e49f20fa
Author: Username <Email>
Date:   Thu Nov 29 13:31:32 2012 +0800
    First commit

$ git log --stat
commit aa27391512e39329edd89b14c52db741e49f20fa
Author: Username <Email>
Date:   Thu Nov 29 13:31:32 2012 +0800
```

```
    First commit
 Hello.java | 6 ++++++
 README.md  | 2 ++
 2 files changed, 8 insertions(+)
```

Each commit is identified by a 40-hex-digit SHA-1 hash code, as highlighted. It also captured the author (username and email) and datetime of the commit.

To view the commit details, use "`git log -p`", which lists all the *patches*.

```
$ git log -p
commit aa27391512e39329edd89b14c52db741e49f20fa
Author: Username <Email>
Date:   Thu Nov 29 13:31:32 2012 +0800
    First commit
diff --git a/Hello.java b/Hello.java
new file mode 100644
index 0000000..dc8d4cf
--- /dev/null
+++ b/Hello.java
@@ -0,0 +1,6 @@
+// Hello.java
+public class Hello {
+   public static void main(String[] args) {
+      System.out.println("Hello, world from GIT!");
+   }
+}
diff --git a/README.md b/README.md
new file mode 100644
index 0000000..178ddc6
--- /dev/null
+++ b/README.md
@@ -0,0 +1,2 @@
+// README.md
+This is the README file for the Hello-world project.
```

The old files are listed as `---` and new file as `+++`. The addition is marked as `+` and deletion as `-`.

Git-Gui Graphical Tool

For convenience, Git provides a GUI tool, called git-gui, which can be used to perform all tasks and view the commit log graphically.

To run the git-gui, you can right-click on the project folder and choose "Git Gui"; or launch the Git-bash shell and run "`git gui`" command.

To view the log, choose "Repository" ⇒ "Visualize master's history", which launches the "gitk". You can view the details of each commit.

You can also view each of the file via "Repository" ⇒ "Browse master's Files" ⇒ Select a file.

File Status: Untracked, Unmodified, Modified, Staged and Commit

A file could be in various status: untracked, unmodified, modified, staged, and commit.

As seen in the previous "`git status`" output, after a commit, all files are set to the *unmodified* state.

Made some changes to the "`Hello.java`" source file, and check the status again:

```
// Hello.java
public class Hello {
   public static void main(String[] args) {
      System.out.println("Hello, world from GIT!");
      System.out.println("Changes after First commit!");
   }
}
```

```
$ git status
# On branch master
```

```
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#       modified:   Hello.java
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#       Hello.class
no changes added to commit (use "git add" and/or "git commit -a")
```

The "`Hello.java`" is marked *modified*, but not *staged* for *commit*. Stage the "`Hello.java`" by issuing the "`git add`" command:

```
$ git add Hello.java

$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#       modified:   Hello.java
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#       Hello.class
```

Commit all staged files via "`git commit`":

```
$ git commit -m "Second commit"
[master 7026f14] Second commit
 1 file changed, 1 insertion(+)

$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#       Hello.class
nothing added to commit but untracked files present (use "git add" to track)
```

Issue "`git log`" to list all the commits:

```
$ git log
commit 51c6b916a8d7969a2e9d25701415d51742f0989e
Author: Username <Email>
Date:   Thu Nov 29 14:09:46 2012 +0800
    Second commit

commit aa27391512e39329edd89b14c52db741e49f20fa
Author: Username <Email>
Date:   Thu Nov 29 13:31:32 2012 +0800
    First commit
```

Check the patches for the latest commit via "`git log -p -1`", with option `-n` to limit to the last `n` commit:

```
$ git log -p -1
commit 51c6b916a8d7969a2e9d25701415d51742f0989e
Author: Username <Email>
Date:   Thu Nov 29 14:09:46 2012 +0800
    Second commit
diff --git a/Hello.java b/Hello.java
index dc8d4cf..f4a4393 100644
--- a/Hello.java
+++ b/Hello.java
@@ -2,5 +2,6 @@
 public class Hello {
    public static void main(String[] args) {
       System.out.println("Hello, world from GIT!");
+      System.out.println("Changes after First commit!");
    }
 }
```

The `.gitignore` File

All the files in the Git directory are either *tracked* or *untracked*, as seen from the output of the "`git status`" command. To remove files (such as `.class`, `.o`, `.exe`) from the untracked, create a "`.gitignore`" file in your project directory, which list the files to be ignored, as follows:

```
# .gitignore

# Java class file
*.class

# Executable, object and archive files
*.exe
*.[oa]

# temp subdirectory
temp/
```

There should not be any trailing comments for filename. You can use regexe for matching the filename/pathname patterns, e.g. `[oa]` denotes either `o` or `a`. You can override the rules by using the inverted pattern (`!`), e.g., `!hello.exe` includes the `hello.exe` although `*.exe` are excluded.

Issue a "`git status`" command to check the untracked files.

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#       .gitignore
nothing added to commit but untracked files present (use "git add" to track)
```

Typically, we also track and commit the `.gitignore` file.

```
$ git add .gitignore

$ git commit -m "Adding .gitignore"
[detached HEAD fafbde8] Adding .gitignore
 1 file changed, 11 insertions(+)
 create mode 100644 .gitignore
```

3.2 Setting up Remote Repo

Setting up a Remote Repo at a Git Hosting Site

1. Signed up at a GIT hosting site such as Github https://github.com/signup/free (Unlimited for open-source projects; fee for private projects); or BitBucket @ https://bitbucket.org/ (Unlimited users for open-source projects; 5 free users for private projects; Unlimited for Academic Plan).
2. Create a new repo called "`test`" on the remote system.
3. On your project directory, set up the remote *name* and *URL* via "`git remote add <name> <URL>`" command. By convention, we shall name the remote as "`origin`". The hosting site shall provide the remote URL. For simplicity, use URL in HTTPS protocol.

   ```
   // for github
   $ git remote add origin https://github.com/username/test.git
   // for bitbucket
   $ git remote add origin https://username@bitbucket.org/username/test.git
   ```

   You can list all the remote name and URL via "`git remote -v`", for example,

   ```
   $ git remote -v
   origin  https://github.com/username/test.git (fetch)
   origin  https://github.com/username/test.git (push)
   ```

4. Push the committed files from the local repo to the remote repo via "`git push -u <remote-name> <branch-`

name>". By convention, we shall call the branch `master`.

```
$ git push -u origin master
Username for 'https://github.com': ******
Password for 'https://username@github.com': *******
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 712 bytes, done.
Total 7 (delta 1), reused 0 (delta 0)
To https://github.com/username/test.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

5. Login to the remote host, you shall find all the committed files of the project.
6. Make some change (e.g., on "`Hello.java`"); stage and commit on the local repo; and push it to the remote:

```
// Hello.java
public class Hello {
   public static void main(String[] args) {
      System.out.println("Hello, world from GIT!");
      System.out.println("Changes after First commit!");
      System.out.println("Changes after Pushing to remote!");
   }
}
```

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working dire
#       modified:   Hello.java
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#       .gitignore
no changes added to commit (use "git add" and/or "git commit -a")

$ git add *.java        // stage

$ git commit -m "Third commit"   // commit
[master d92ab19] Third commit
 1 file changed, 1 insertion(+)

$ git push origin master   // push to remote
Username for 'https://github.com': ******
Password for 'https://username@github.com': ******
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/username/test.git
   51c6b91..d92ab19  master -> master
```

Again, login to the remote to check the committed files.

Cloning a Project from a Repo (`git clone`)

Anyone (such as your team mates) having read access to the remote can *clone* your project, using "`git clone`" command. You can optionally choose your working directory name.

```
// cd to your base directory for the project working directory
// Clone into working directory called "test"
$ git clone https://github.com/xxxx/test.git
   // Use the URL for remote repo
   // You can also clone from a local repo by using the file path
```

```
// Clone into working directory called "git_clone_test"
$ git clone https://github.com/xxxx/test.git git_clone_test
Cloning into 'git_clone_test'...
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 10 (delta 2), reused 10 (delta 2)
Unpacking objects: 100% (10/10), done.
```

Forking a Repo in Fork-and-Pull Collaboration Model

There are many types of collaboration models. In the *Fork-and-Pull* model (or integration-manager workflow), the project maintainer (or integration manager) pushes the project to the public repo. Everyone else can contribute to the project by forking the project, making changes and sends a pull-request to the project maintainer. The project maintainer can then pull and merge the changes in his local repo, and then pushes the update to the public repo.

The steps are:

1. The project maintainer pushes the project to the remote public repo.
2. The contributor *forks* the project at the remote public repo, by pushing the *fork* button. The remote Git server creates and copies the project into the contributor's personal account.
3. The contribute clones the project from the remote fork into his local repo, via:

   ```
   $ cd base-directory-of-the-project-working-directory

   $ git clone https://github.com/contributor-username/project-name.git
   ```

4. When a fork is cloned, Git creates a remote called `origin` that points to the fork, not the original repo it was forked from. Git also creates a branch called `master` for your fork. To keep track of the original repo, create a remote named `upstream` and fetch all the changes:

   ```
   $ cd your-local-repo-of-the-fork

   // Create a remote called upstream pointing to the original repo
   $ git remote add upstream https://github.com/original-username/project-name.git

   // List the remote name and URL
   $ git remote -v

   // Fetch all changes from the original repo to local repo, without modifying the working tree
   $ git fetch upstream

   // Merge the changes into the working tree
   $ git merge upstream/master
   Already up-to-date.
   ```

5. Now, the contributor can made changes on its local repo (typically on a new branch), stage and commit the change, and push them to the forked remote `origin` branch `master`:

   ```
   // Create a new branch called mybranch
   $ git branch mybranch

   // Switch to the new branch
   $ git checkout mybranch
   Switched to branch 'mybranch'

   // Make change, stage (git add) and commit (git commit)

   // Push changes to forked remote of the specific branch
   $ git push origin mybranch
   ```

   The contributor can also continue to fetch changes from `upstream`, and apply (merge) the changes to his local repo.

6. The contributor sends a *pull-request* to the project maintainer. He shall first choose the correct branch name, then push the *pull-request* button and fill up the information.
7. The project maintainer can review and decide whether to accept the changes. If he does, he could fetch and merge the changes:

```
// Checkout the master branch of the local repo
$ git checkout master

// Add a new remote pointing to the contributor's account
$ git remote add xxxx git://github.com/xxxx/project-name.git

// Fetch the changes into local repo from the remote
$ git fetch xxxx

// Merge the changes
$ git merge xxxx/branch-name

// Push the update to the remote public repo
$ git push origin master
```

Shared Repository Model

In *Shared Repository Model*, everyone is granted push access to a single shared repository and topic branches are used to isolate changes. Shared Repo is more prevalent with small teams on private projects.

[TODO]

4. More Git Commands

View Unstaged and Staged Changes (`git diff`)

To view what you have changed but have yet to stage, use command "`git diff`" with no argument. For example, make some changes to "`Hello.java`" and,

```
$ git diff
diff --git a/Hello.java b/Hello.java
index 59bdbfa..f9fd157 100644
--- a/Hello.java
+++ b/Hello.java
.......
```

Now, stage the "`Hello.java`". To view what you have staged but not commit, use command "`git diff --cached`" (or "`git diff --staged`"):

```
$ git add *.java      // stage
$ git diff --cached
diff --git a/Hello.java b/Hello.java
index 59bdbfa..f9fd157 100644
--- a/Hello.java
+++ b/Hello.java
......
```

If you modify the file again, the earlier changes are staged, but the later changes are not staged. You can use "`git diff`" to view the unstaged changes and "`git diff --cached`" (or "`git diff --staged`") to view the staged changes.

For convenience, you can use the "git-gui" tool to view the unstaged and staged changes.

Stage and Commit (`git commit -a`)

You can skip the staging (i.e., the "`git add`") and commit all *modified* files via "`git commit`" with `-a` option (or `--all`). For example,

```
$ git commit -a -m "Another commit"
```

Amending the commit (`git commit --amend`)

After a commit, you can amend the commit (e.g., you have forgotten to stage a file), if you did not make any change, via "`git commit --amend`"). For example,

```
$ git commit -m "A commit"
$ git add morefile          // stage the forgotten file
$ git commit --amend        // treat as one commit if no change to files
```

Unstage a Staged file (`git reset head <file>`)

Recall that you can use "`git add <file>`" to stage modified files into the staging area. To unstage staged files, use "`git reset head <file>`".

Unmodified a modified file (`git checkout -- <file>`)

After a commit, you may have modified some files. You can discard the changes by checking out the last commit via "`git checkout -- <file>`".

Removing File (`git rm <file>`)

"`git rm <file>`" is the reserve of "`git add <file>`".

To remove a file from the index (staging area), but keep the file on your working directory (e.g., you accidentally run a "`git add`" to stage a file), use "`git rm --cached <file>`". This unstages a staged file, i.e., reversing "`git add <file>`".

To remove a file both from the index (staging area) and the working directory, use "`git rm <file>`".

To remove a file after you have committed, run "`git rm <file>`" followed by a "`git commit`".

Before Git remove a file, it check if the version in the current branch (i.e., `HEAD`) matches the version in the working directory. If not, you have to use "`git rm -f`" to force the removal.

Moving (or Renaming) File (`git mv`)

To rename (or move) a file, you could do a "`git rm <file>`" followed by a "`git add <file>`"; or simply do a "`git mv <file>`". Again, you need to follow by a "`git commit`" to commit the changes.

5. Git GUI Tools

Git-gui

Git-gui is bundled with Git. To launch git-gui, right click on the working directory and choose "git gui", or run "`git gui`" command on the Git-Bash shell.

[TODO]

EGit Plugin for Eclipse

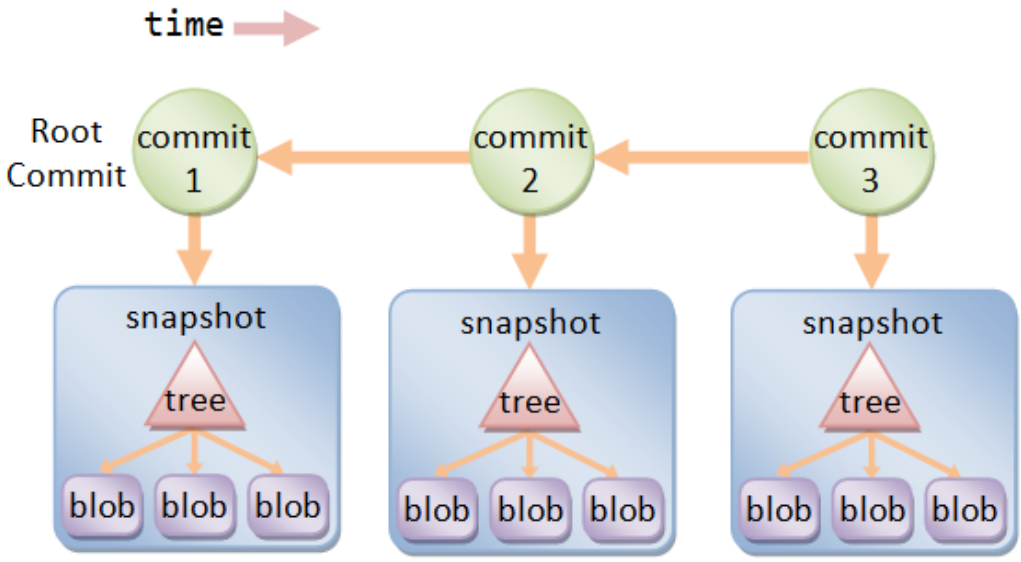
[TODO]

6. Git's Data Structures

Git has two primary data structures:

1. an immutable, append-only object database (or local repo) that stores all the commits and file contents;
2. a mutable index (or cache, or staging area) that caches the staged information.

The index serves as the connection between object database and working tree. It serves to avoid volatility, and allows you to stage ALL the files in *batch* before issuing a commit, instead of committing individual file. Changes to files that have been explicitly added to the index (staging area) via "`git add`" are called *staged changes*. Changes that have not been added are called *unstaged changes*. Staged and unstaged changes can co-exist. Performing a commit copies the statged changes into object database (local repo) and clears the index. The unstaged changes remain.



The object database contains these objects:

- Each version of a file is represented by a *blob* (binary large object - a file that can contain any data: binaries or characters). A blob holds the file data only, without any metadata - not even the filename.
- A *snapshot* of the working tree is represented by a *tree* object, which links the blobs and sub-trees for sub-directories.
- A *commit* object points to a tree object, i.e., the snapshot of the working tree at the point the commit was created. It holds metadata such as timestamp, log message, author's and committer's username and email. It also references its parent commit(s), except the root commit which has no parent. A normal commit has one parent; a merge commit could have multiple parents. A commit, where new branch is created, has more than one children. By referencing through the chain of parent commit(s), you can discover the history of the project.

Each object is identified (or named) by a 160-bit (or 40 hex-digit) SHA-1 hash value of its contents (i.e., a content-addressable name). Any tiny change to the contents produces a different hash value, resulted in a different object. Typically, we use the first 7 hex-digit prefix to refer to an object, as long as there is no ambiguity.

There are two ways to refer to a particular commit: via a branch or a tag.

- A branch is a mobile reference of commit. It moves forward whenever commit is made on that branch.
- A tag (like a label) marks a particular commit. Tag is often used for marking the releases.

7. Branching and Merging

Branching allows you and your team members to work on different aspects of the software *concurrently*, and merge into the `master` branch as and when they completes. Branching is the most important feature in a *concurrent* version control system.

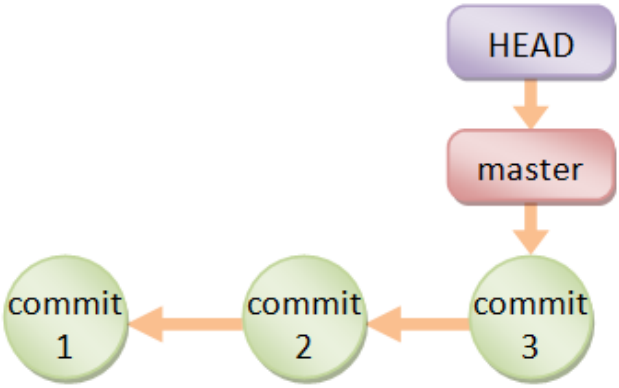
A branch in Git is a lightweight movable pointer to one of the commits. For the initial commit, Git assigns the default branch name called `master` and sets the `master` branch pointer at the initial commit. As you make further commits on the `master` branch, the `master` branch pointer move forward accordingly. Git also uses a special pointer called `HEAD` to keep track of the branch that you are currently working on. The `HEAD` always refers to the latest commit on the current branch. Whenever you switch branch, the `HEAD` also switches to the latest commit on the new branch.

For example, let create a Git-managed project called `git_branch_test` with only the a single-line `README.md` file:

```
This is the README. My email is xxx@somewhere.com
```

```
$ git init
$ git add README.md
$ git commit -m "Commit 1"

// Add a line: This line is added after Commit 1
$ git add README.md
$ git commit -m "Commit 2"
```

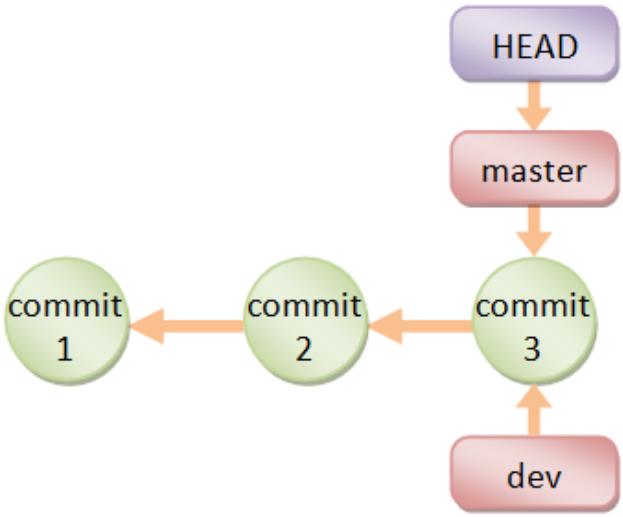
```
// Add a line: This line is added after Commit 2
$ git add README.md
$ git commit -m "Commit 3"
```



Creating a new Branch (`git branch <branch-name>`)

You can create a new branch via "`git branch <branch-name>`" command. When you create a new branch (says `development` or `dev`), Git creates a new branch pointer for the branch `development`, pointing initially at the latest commit on the current branch (e.g., `master`).

```
$ git branch development
```



Take note that when you create a new branch, the HEAD pointer is still pointing at the current branch.

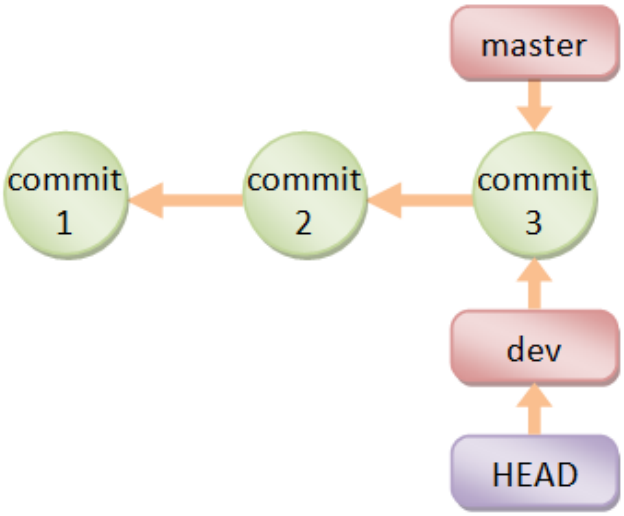
Branch Names Convention

- `master` branch: the production branch with tags for the various releases.
- `development` (or `next` or `dev`) branch: developmental branch, to be merged into master if and when completes.
- topics branch: a short-live branch for a specific topics, such as introducing a feature (for the `development` branch) or fixing a bug (for the `master` branch).

Switching to a Branch (`git checkout <branch-name>`)

Git uses a special pointer called `HEAD` to keep track of the branch that you are working on. The "`git branch`" command simply create a branch, but does not switch to the new branch. To switch to a branch, use "`git checkout <branch-name>`" command. The `HEAD` pointer will be pointing at the switched branch (e.g., `development`).

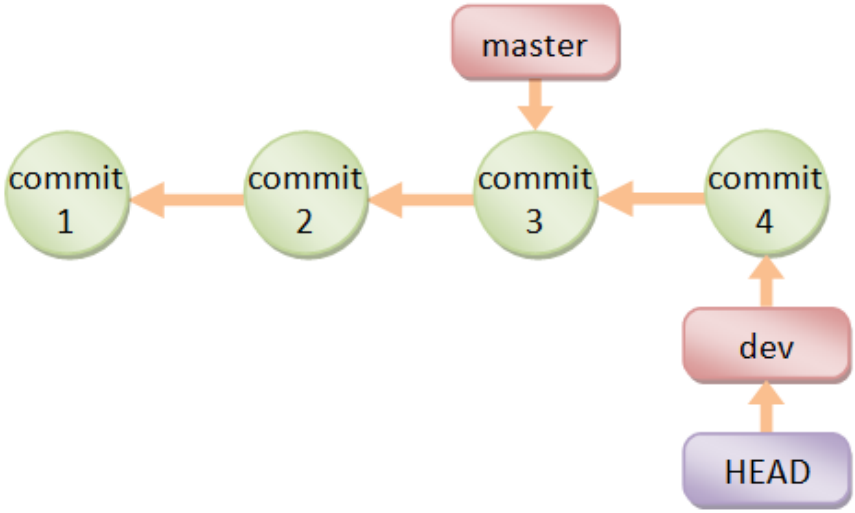
```
$ git checkout development
Switched to branch 'development'
```



Alternatively, you can use "`git checkout -b <branch-name>`" to create a new branch and switch into the new branch.

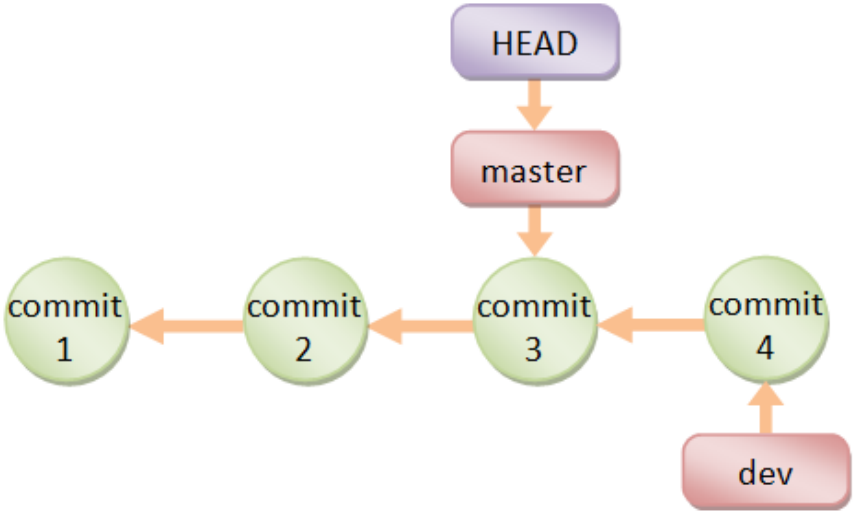
If you switch to a branch (via "`git checkout`", e.g., `development`), make changes and commit. The `HEAD` pointer moves forward in the branch e.g., `development`.

```
// Add a line: This line is added on development branch after Commit 3
$ git add README.md
$ git commit -m "Commit 4"
```



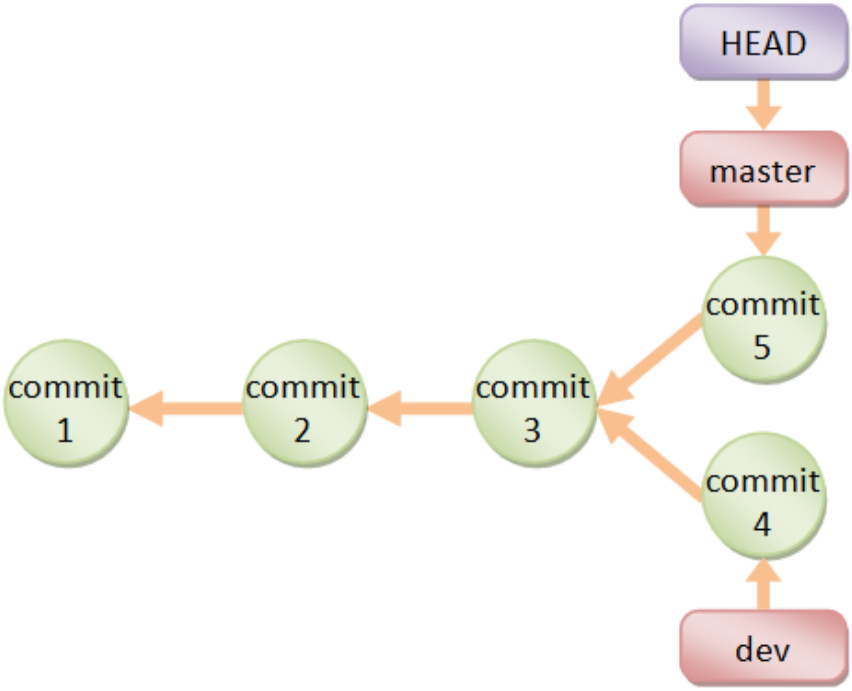
You can switch back to the `master` branch via "`git checkout master`". The `HEAD` pointer moves back to the last commit of the `master` branch, and the working directory is *rewinded* back to the lastest commit on the `master` branch.

```
$ git checkout master
Switched to branch 'master'
// Check the content of the README.md, which is reminded back to Commit 3
```



If you continue to work on the `master` branch and commit, the `HEAD` pointer moves forward on the `master` branch. The two branches are now diverged.

```
// Add a line: This line is added on master branch after Commit 4
$ git add README.md
$ git commit -m "Commit 5"
```



If you check out the development branch, the file contents will be rewinded back to Commit -4

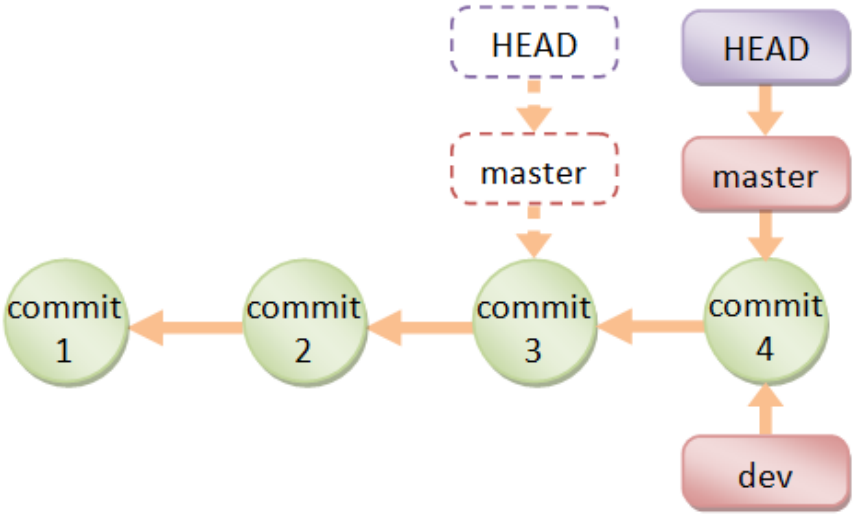
```
$ git checkout development
// Check file contents
```

Merging Two Branches (`git merge <branch-name>`)

To merge two branches, says `master` and `development`, check out the first branch (via "`git checkout <branch-name>`") and merge with another branch via command "`git merge <branch-name>`".

If the branch to be merged is a direct descendant, Git performs *fast forward* by moving the `HEAD` pointer forward. For example, suppose that you are currently working on the `development` branch at commit-4, and the master branch's latest commit is at commit-3:

```
$ git checkout master

// Let discard the Commit-5 totally and rewind to commit-3 on master branch
// This is solely for illustration!!! Do this with great care!!!
$ git reset --hard HEAD~1
HEAD is now at 7e7cb40 Commit 3
  // HEAD~1 moves the HEAD pointer back by one commit
  // --hard resets the working tree
  // --mixed (default) undoes the commit but keeps the working tree, not keeping the index
  // --soft undoes the commit, keeps the working tree and index

$ git merger development
Updating 7e7cb40..4848c7b
Fast-forward
 README.md | 1 +
 1 file changed, 1 insertion(+)

// Check the file contents
```



If the two branches are diverged, git automatically searches for the common ancestor and performs a three-way merge. If git detects a conflict, it will pause the merge and issue a merge conflict and ask you to resolve the conflict manually. The file is marked as *unmerged*. You can issue "`git status`" to check the unmerged files, study the details of the conflict, and decide which way to resolve the conflict. Once the conflict is resolve, stage the file (via "`git add`"). Finally, run a "`git commit`" to finalize the three-way merge.

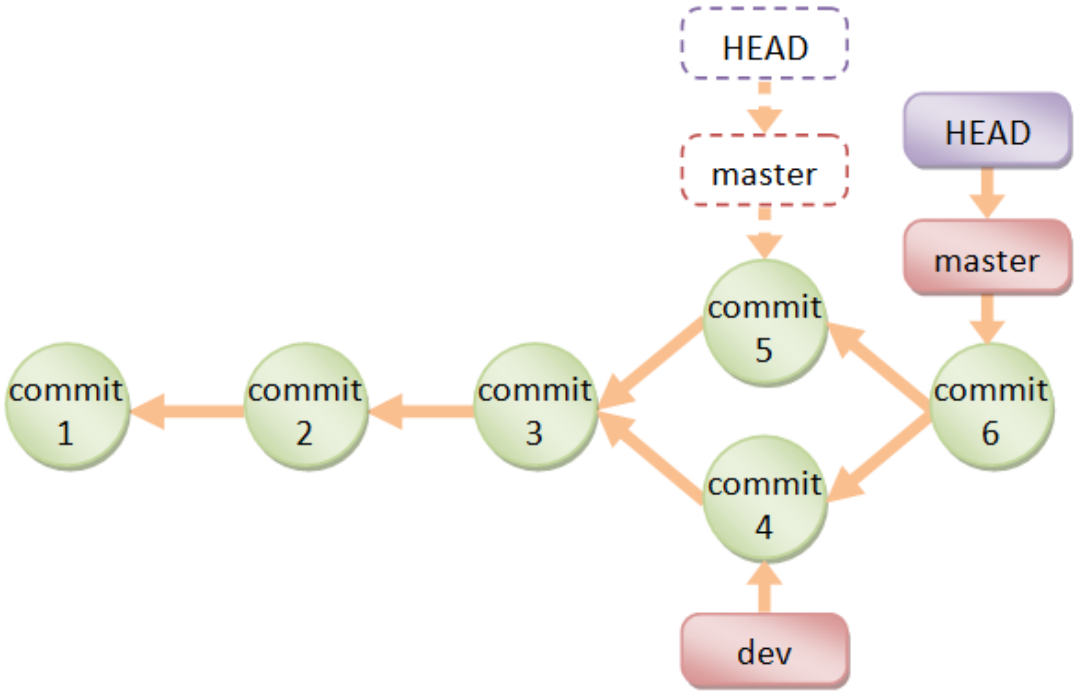
```
$ git checkout master
// undo the Commit-4, back to Commit-3
$ git reset --hard HEAD~1
HEAD is now at 7e7cb40 Commit 3

// Change the email to abc@abc.com
$ git add README.md
$ git commit -m "Commit 5"

$ git checkout development
// undo the Commit-4, back to Commit-3
$ git reset --hard HEAD~1
// Change the email to xyz@xyz.com to trigger conflict
$ git add README.md
$ git commit -m "Commit 4"

// Let's do a 3-way merge with conflict
$ git checkout master
$ git merge development
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.

$ git status
# On branch master
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#       both modified:      README.md
no changes added to commit (use "git add" and/or "git commit -a")
```

The conflict file is marked as follows:

```
<<<<<<< HEAD
This is the README. My email is abc@abc.com
=======
This is the README. My email is xyz@xyz.com
>>>>>>> development
This line is added after Commit 1
This line is added after Commit 2
```

You need to manually decide which way to take, or you could discard both by setting the email to zzz@nowhere.com.

```
$ git add README.md
$ git commit -m "Commit 6"
```



Viewing the Commit Graph (`gitk`)

You can use the "git-gui" "gitk" tool to view the commit graph.

To run the git-gui, you can right-click on the project folder and choose "Git Gui"; or launch the Git-bash shell and run "`git gui`" command.

To view the commit graph, choose "Repository" ⇒ "Visualize master's history", which launches the "gitk". You can view the details of each commit.

Deleting Merged Branch (`git branch -d <branch-name>`)

The merged branch (e.g., development) is no longer needed. You can delete it via "`git branch -d <branch-name>`".

```
$ git branch -d development
Deleted branch development (was a20f002).

// Create the development branch again at the latest commit
$ git branch development

// View via git-gui
```

8. Tagging the Releases at Commits

Tag (or label) can be used to tag a specific commit as being important, for example, to mark a particular release. The release is often marked in this format: *`version-number.release-no.modificaton-no`* (e.g., `v1.1.5`) or or *`version-number.release-no.upgrade-no_modificaton-no`* (e.g., `v1.7.0_26`).

Listing Tags (`git tag`)

To list the existing tags, use "`git tag`" command.

Types of Tags - Lightweight Tags and Annotated Tags

There are two kinds of tags: lightweight tag and annotated tag. Lightweight tag is simply a pointer to a commit. Annotated tag contains annotations (meta-data) and can be digitally signed and verified.

Creating Annotated Tags (`git tag -a <tag-name> -m <message>`)

To create an annotated tag at the latest commit, use "`git tag -a <tag-name> -m <message>`", where `-a` option specifies annotation tag having meta-data. For example,

```
$ git tag -a v1.0.0 -m "Version 1.0.0"

// List all tags
$ git tag
v1.0.0

// Show tag details
$ git show v1.0.0
   // Show the commit point and working tree
```

To create a tag for earlier commit, you need to find out the commit's first seven character hash code (via "`git log`"), and issue "`git tag -a <tag-name> -m <message> <commit-hash-code-7>`". For example,

```
$ git log
......
commit 7e7cb40a9340691e2b16a041f7185cee5f7ba92e
......
    Commit 3

$ git tag -a "v0.9.0" -m "Version 0.9.0" 7e7cb40

// List all tags
$ git tag
v0.9.0
v1.0.0

// Show details of a tag
$ git show v0.9.0
......
```

Creating Lightweight Tags (`git tag <tag-name>`)

To create a lightweight tag (without meta-data), use "`git tag <tag-name>`" without the `-a` option. The lightweight tag stores only the commit hash code.

Signed Tags

You can signed your tags with your private key, with `-s` option instead of `-a`.

To verify a signed tag, use `-v` option and provide the signer's public key.

[TODO] Example

Pushing to Remote Repo

By default, Git does not push tags (and branches) to remote repo. You need to push them explicitly, via "`git push origin <tag-name>`" for a particular tag or "`git push origin --tags`" for all the tags.

9. Working with Remote Repo

Adding Remote Repo (`git remote add <remote-name> <url>`)

[TODO]

Listing Remote-Name and URL (`git remote -v`)

[TODO]

Pushing to Remote Repo (`git push <remote-name> <branch-name>`)

```
$ git push origin master    // remote-name is origin, branch-name is master
```

[TODO]

Cloning a Remote Repo (`git clone <url>`)

The "`git clone`" automatically set up a local `master` branch to track the remote `master` branch.

Pulling from Remote Repo (`git pull`)

The "`git pull`" automatically fetch and merge the remote branch with the local branch.

[TODO]

Fetching from Remote Repo (`git fetch <remote-name>`)

The "`git fetch`" fetches the update but does not merge the remote branch with the local branch.

[TODO]

Merging the Fetched Changes (`git merge <remote-name>/<branch-name>`)

The "`git fetch`" fetches the update but does not merge the remote branch with the local branch.

[TODO]

10. How-To

10.1 How to Amend the Last Commit (`git commit --amend`)

If you make a commit but want to change the commit message:

```
$ git commit --amend -m "message"
```

If you make a commit but realize that you have not staged some files, you can also do it with `--amend`:

```
$ git add morefile
$ git commit --amend
```

You can also make some changes to working tree, stage, and amend the last commit

```
// Edit morefile and make some changes
$ git add morefile
$ git commit --amend
```

10.2 How to Undo the Previous Commit(s) (`git reset`)

The "`git commit --amend`" can be used to *amend* the last commit.

To undo previous commit(s):

```
// Reset the HEAD to the previous commit
// --soft to keep the working tree and index
$ git reset --soft HEAD~1   // Windows
$ git reset --soft HEAD^    // Unix

// Make changes
......

// Stage
$ git add ......
```

```
// Commit
$ git commit -c ORIG_HEAD
```

The "`git reset --hard HEAD~1`" moves the `HEAD` to the previous commit, restore the working tree and discard the index (i.e., discard all change after the previous commit). Instead of `HEAD~n`, you can also specify the commit hash code.

The "`git rest HEAD~1`" with default `--mixed` moves the `HEAD` to the previous commit, keep the working tree and discard the index

The "`git reset --soft HEAD~1`" moves the `HEAD` to the previous commit, keep the working tree and the index (i.e., keep all changes after the previous commit).

[TODO] Examples, diagrams and "`git status`" outputs.

For a public repo, you probably need to make another commit and push the commit to the public repo, or ...

10.3 Relative Commit Names

A commit is uniquely and absolutely named using a 160-bit (40-hex-digit) SHA-1 hash code of its contents. You can always refer to a commit via its hash value or abbreviated hash value (such as the first 7 hex-digit) if there is no ambiguity.

You can also refer to a commit *relatively*, e.g., `master~1` (Windows), `master^` (may not work in Windows), `master^1` refers to the *previous* (parent) commit on the `master` branch; `master~2`, `master^^` refers to the previous of the previous (grandparent) commit, and etc. If a commit has multiple parents (e.g., due to merging of branches), `^1` refers to the first parent, `^2` refers to the second parent, and so on.

REFERENCES & RESOURCES

1. GIT Documentation @ http://git-scm.com/doc.
2. Git User's Manual @ http://www.kernel.org/pub/software/scm/git/docs/user-manual.html.
3. Scott Chacon, "Pro Git", Apress, 2009.
4. Jon Loeliger and Matthew McCullough, "Version Control with Git", 2nd Eds, O'reilly, 2012.
5. GitHub @ https://github.com/.
6. Bitbucket 101 @ https://confluence.atlassian.com/display/BITBUCKET/Bitbucket+101.