

# Learning by Doing

## A Short Introduction to git

Jianwen WEI

Shanghai Jiaotong University

April 5, 2012

# Outline

- 1 What is git?
- 2 Learning and USING git
- 3 Exercises
- 4 Recommended Materials

# Section 1

What is git?

# What does a Version Control System do?

- Track source code
  - ▶ Maintain code history, integrity, atomic change...
- Coordinate distributed development
  - ▶ branch, merge conflicts, tag...

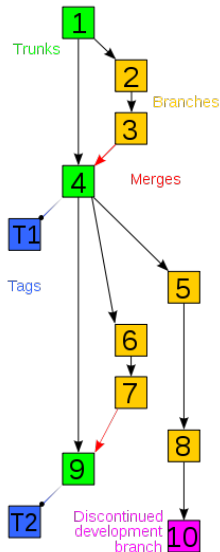


Figure: VCS work flow

# VCS Work Flow Categories

- Centralized: VSS, CVS, SVN
- Distributed<sup>1</sup>: BitKeeper, git, mercurial...

---

<sup>1</sup>Distributed VCSs support centralized work flow too.

# Why git is better than X (SVN, CVS, ...)

- git is super fast
- Full repository clone
- Local history: no need to connect to servers when viewing the revision history
- Cheap branch and easy merge
- **github**: social coding<sup>2</sup>
- Other things: tidy working directory, better compression, multi work flow support, ...

---

<sup>2</sup>**bitbucket**, **Google Code** support git too, but github in no doubt has more *fun*.

# General Advice on Learning git

- Try git and github
- Most graphical tool/plugin<sup>3</sup> *SUCK*. Please use the command-line git.
- Read git's prompts, run **git help** to get help.
- Find “how-to” on Google, StackOverflow, git book.

---

<sup>3</sup>tortoisegit, gitk, EGit, Snow Octocat... But please, oh please use the command-line tool.

# Rules of Thumb for git

- “A clear development flow is worth thousands of VCSs.”
- Modular design, avoid simultaneous source file editing by different members.
- Head version at trunk is always ready to deploy.
- Modification is made on branches, then merged into trunk.
- Stay on your own branch.
- Write comment to each commit.



## Section 2

### Learning and USING git

## To get started, I will...

- Illustrate git's various work flows.
- Explain the most frequently used git commands.
- Give exercises for self check. Some of the exercises require github access.

## git's stand-alone work flow

- You can use git on a stand-alone computer and easily integrate the code into a more sophisticated work flow (distributed or centralized) at a later time.

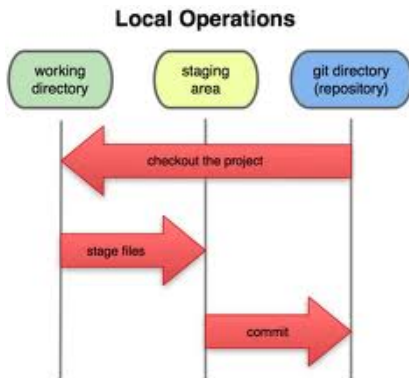


Figure: git's local work flow

## git's distributed work flow

- Every collaborator keeps a full clone of the repository.
- All repositories are peers.
- Repositories are not necessarily consistent at all time. Use push/pull to exchange changes when necessary.

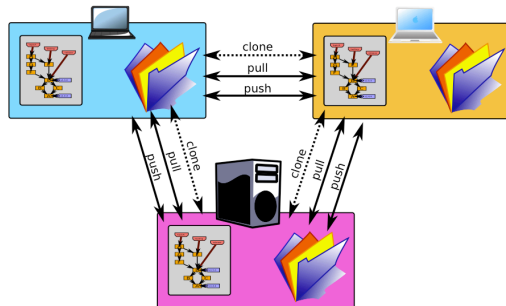


Figure: git's distributed work flow

git's emulation to the centralized work flow  
**(RECOMMENDED)**

- It's **emulation**, not *real*.
- The statement, "all repositories are peers.", still holds.
- We pretend that we see the central repo only, unaware of each other's peer repo.



Figure: git's centralized work flow for John and Jessica

# Set up git

- Please follow github's nice tutorials to set up<sup>4</sup> git on **Windows**, **Linux** or **Mac**.
- **Must-known things about SSH keys**: private key, public key, the pass phrase to access the private key, key fingerprint.
- Don't forget to set `user.name` and `user.email`<sup>5</sup> before your very first git commit.

---

<sup>4</sup>The email you fill in when signing up is used for web login and password reset only. github uses SSH keys for git authentication. Try to clarify the following *pass phrases*: your email account's pass phrase, your github account's pass phrase, and the pass phrase to access your SSH private key.

<sup>5</sup>Usernames and emails in git's configuration are for identification purpose only, not for sending emails. It is highly recommended that the email in git and SSH keeps the same.

# git command

- help
- init
- status
- add
- .gitignore
- commit
- diff
- tag
- Working with branch
- Working with remotes
- submodule
- Oh, there is a conflict!!!
- “Time Machine”

## help: Get help

`git help COMMAND` Get help from git.

- `git help add`
- `git help commit`
- ...



## init: Initialize a local git repo for your project

init command will create a .git dir on the top level of your project.

1. `cd YOUR_PROJ_DIR`
2. `git init .`

## status: Show the status of your repo

`git status`

- status tells you how to **UNDO** the last operation on git
- **File status**: untracked, unstaged, staged (indexed), committed<sup>6</sup>

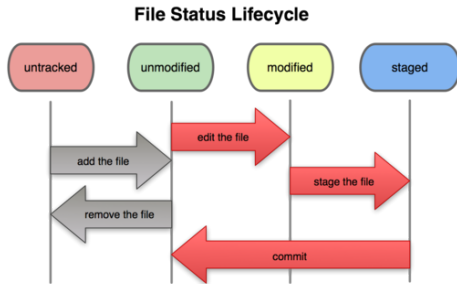


Figure: File Status Lifecycle

<sup>6</sup>The *committed* status simply displays nothing when running `git status`.

## add: A multi-function git command

```
git add FILES_OR_DIR
```

- For untracked files: *add* them to git's control
- For unstaged changes: *add* them to the staged area
- For conflicted files: *add* marks them as “resolved”

## .gitignore: Ignore files

- Specify file types to ignore in .gitignore (a normal text file)
  - ▶ \*.tmp (Ignore all \*.tmp files)
  - ▶ \*.bak (Ignore all \*.bak files)
- git has an internal .gitignore. You can ask git NOT to ignore some file types in .gitignore
  - ▶ !\*.o (*Don't* ignore \*.o files)
- git help gitignore

## commit: Store the status (snapshot) permanently

- `git commit -m "YOUR_COMMENT"`
  - ▶ `git commit` Stores the STAGED changes only
  - ▶ `git commit -a` Stores all the STAGED and UNSTAGED changes.
- Please write comment for each of your commit.
- Each commit is identified by a **UNIQUE** SHA-1 ID of 40 ASCII characters.

```
commit dd5f924c40096b9cda27ffd1cfd1205822ab3c70
```

```
Author: Github Support <me@github.com>
```

```
Date: Sun Apr 1 19:38:37 2012 +0800
```

Restart the `git-tutorial` project.

## diff: Find differences

- `git diff`
  - ▶ changes between the staged and working files
- `git diff --staged`
  - ▶ changes between the HEAD and the staged files
- `git diff HEAD`
  - ▶ changes between the HEAD and the working files
- `git diff COMMIT_ID COMMIT_ID`
  - ▶ changes between two commits

## tag: A milestone version

- `git tag`
  - ▶ See all the tag
- `git show TAG_NAME`
  - ▶ See a tag in detail
- `git tag TAG_NAME`
  - ▶ Add a “lightweight” tag
- `git tag -a TAG_NAME -M YOUR_COMMENT`
  - ▶ Add an annotated tag
- `git tag -d TAG_NAME`
  - ▶ Delete a tag

## Submodule: Integrate multi git repos

- `git help submodule`
- Repo in Repo
- Manage other repos as “submodules” in your project



# Working with **branch**: branch, checkout, merge

A branch-based development flow:

1. Create a branch
2. Switch to the newly-created branch
3. Modify and commit on the branch
4. Merge branch's changes into trunk.

## Working with **branch**: branch, checkout, merge

- `git branch`
  - ▶ See all the branches
- `git branch BRANCH_NAME`
  - ▶ Create a branch
- `git branch -d BRANCH_NAME` Delete a branch
  - ▶ Delete a branch
- `git branch -D BRANCH_NAME` Force delete a branch
  - ▶ Force delete a branch

## Working with **branch**: branch, checkout, merge

- `git checkout BRANCH_NAME`
  - ▶ Switch to a branch. The working files will change.<sup>7</sup>
- `git checkout -f BRANCH_NAME`
  - ▶ Force switch to a branch
- `git checkout master`
  - ▶ Go back to trunk, named *master* in git.
- `git checkout -b BRANCH_NAME`
  - ▶ Create a branch then switch to it.

---

<sup>7</sup>Don't confuse git's term `checkout` here with Subversion's `checkout`.

## Working with **branch**: branch, checkout, merge

- `git merge BRANCH_A BRANCH_B`
  - ▶ Merge branch\_a's and branch\_b's changes into *current* branch
- `git checkout master, git merge master BRANCH_NAME`
  - ▶ Merge changes into trunk, the master branch.

## Working with **remotes**: clone, remote, push, pull

- `git clone REPO_URL` Full clone of a repo.
- URL can be in forms of local dir (`~/proj`), git (`git://xxx`), SSH (`ssh://xxx`), https (`http://xxx`)...

# Working with **remotes**: clone, *remote*, push, pull

- `remote` Manages the set of tracked repositories.<sup>8</sup>
- `git remote`
  - ▶ Show all the tracked repositories.
- `git remote show REPO_NAME`
  - ▶ Show the repo's details.
- `git remote add REPO_NAME REPO_URL`
  - ▶ Add a remote repo to tracked list.
- `git remote -d REPO_NAME`
  - ▶ Remove a remote repo from the tracked list.
- `git remote rename REPO_OLD REPO_NEW`
  - ▶ Rename a repo.

---

<sup>8</sup>Remote repos in git are just references or pointers, so you lose or gain *nothing* when adding or removing a remote repo.

## Working with **remotes**: clone, remote, *push*, *pull*

- `git pull REPO_NAME REMO_BRANCH`
  - ▶ Merge remote branch's changes into current branch.
- `git push REPO_NAME REMO_BRANCH`
  - ▶ Push current branch's changes to the remote branch.
- `git push REPO_NAME :REMO_BRANCH`
  - ▶ Delete a remote branch.

# Oh, there is a conflict!!!

- A conflict looks like:

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

- Conflicts arise when git cannot automatically merge changes at merge or pull operations.
- Don't panic. Conflicts are no big deal, sometimes even inevitable.
- What you should do: merge the conflicts, mark the files as "resolved", then commit the changes.



# Working with conflicts: merge, resolve, commit

1. You can
  - ▶ Merge the conflicts *manually* diff can help you. Or
  - ▶ `git checkout --theirs FILES` Replace the conflicted files with *theirs*.
  - ▶ `git checkout --ours FILES` Replace the conflicted files with *ours*.
2. `git add CONFLICT_FILES` Mark the file as resolved.
3. `git commit -m "YOUR_COMM"` Commit changes to the repo.

## “Time Machine”: `stash`, `checkout`

`stash` saves your temporary work and resets the files to HEAD version. You can handle some emergency fix first then continue to hack at a latter time.

1. `git stash`
  - ▶ Save the temp changes.
2. `git stash list`
  - ▶ Check the stash list.
3. EDIT and COMMIT your emergency fix.
4. `git stash pop`
  - ▶ Continue to hack

## “Time Machine”: stash, checkout

checkout enable you to go backward and forward in the revision history.

1. `git checkout COMMITID_OR_TAGNAME`<sup>9</sup>
  - ▶ Time Machine starts up.
2. You are on a `unnamed` branch with file status dating back. Do anything you want.
3. `git checkout master`
  - ▶ Come back to master.

---

<sup>9</sup>The full commit ID is 40 characters long. But you may type a short prefix (like 4~6 characters) to refer a commit uniquely.

## Section 3

### Exercises

# Exercise 1: Set up git environment

1. Set up git on your computer, and sign up a github account.
2. Initialize a local project as git repo, make your first git.
3. Email your SSH public key file as an attachment to **me**. Name your pubkey file after “YOUR@EMAIL.pub”, e.g., you should rename your `id_rsa.pub` to `xxx@sjtu.edu.cn.pub` and send it to me.

## Exercise 2: git basics

Be familiar with `status`, `add`, `commit`, `diff`, `tag`.

## Exercise 3: Branch-based development

1. Create a branch.
2. Checkout to that branch.
3. Merge the changes into trunk (master).
4. Delete the branch.

## Exercise 4: Be social on github

1. Follow **me** on github.<sup>10</sup>
2. I will add you as a collaborator. Please wait for my message on github before proceeding to next step.
3. Clone the **GitForBeginners** project with Read+Write access.
4. Write something into the README.mkd (DON'T destroy the description header). add, commit, pull, push.

---

<sup>10</sup>Please feel free to unfollow me when finishing all the required exercises.



## Exercise 5: Manage remotes

1. You clone the remote repo `GitForBeginners` on github. Try `git remote`.
2. Copy the `REPO_URL` to somewhere else.
3. Delete the remote repo.<sup>11</sup>
4. Add the remote repo `REPO_URL` with a name you prefer, such as `myrepo`.
5. Rename the remote repo to its original name – `origin`.

---

<sup>11</sup>Don't worry. It is just a reference.

## Exercise 6: Remote branch

1. Create a local branch with your full name, such as `zhangsan`.
2. Write something into `README.mkd` on the branch. `add`, `commit`, `pull`, `push` to the remote branch.
3. Leave the branch on github as a mark of “I finish the homework”. Please recreate the remote branch if you’ve tried the *delete remote branch* command.

## Exercise 7: Handle conflicts

1. Clone **GitForBeginners** twice into two separate projects, namely proj\_A and proj\_B.
2. In proj\_A, modify README.mkd. add, commit, pull, push
3. In proj\_B, modify the *SAME* lines of README.mkd as you do in proj\_A. add, commit, pull
4. A *conflict* towards README.mkd arises in proj\_B.
5. Resolve the conflict, then add, commit, pull, push to github.

## Exercise 8: Time Machine

Use `stash`, `checkout` to do time travel.

## Section 4

### Recommended Materials

# Recommended Materials for Learning git

- “Git Tutorials” by Li Yanrui
- `github:help`
- Pro Git On line
- Video: “Git the basics” by Bart Trojanowski
- O'Reilly Book: Version Control With Git, 2nd Edition

# Acknowledgment

- The slides are composed with **Markdown** language, and converted to **latex beamer** with **pandoc**.
- **XeTeX** is a nice typesetting system. **latexmk** helps to hide the complexity of compilation.
- The slides, along with the **project**, are hosted on **github**.
- Feedback is always welcomed. Write **me** an email or open an issue on the **project** homepage.