Git Documentation

Prepared by: Chai Ying Hua

Date: 23th June 2018

Table Of Contents

1. Introduction

1.1 What is version control.

Version control systems help a software team manage changes to source code over time. It keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

Software teams that do not use any form of version control often run into problems like not knowing which changes that have been made are available to users or the creation of incompatible changes between two unrelated pieces of work that must then be painstakingly untangled and reworked. If you're a developer who has never used version control you may have added versions to your files, perhaps with suffixes like "final" or "latest" and then had to later deal with a new final version.

1.2 Benefits of version control.

a. **A complete long-term change history of every file.** This means every change made by many individuals over the years. Changes include the creation and deletion of files as well as edits to their contents.
b. **Branching and merging.** Creating a "branch" in VCS tools keeps multiple streams of work independent from each other while also providing the facility to merge that work back together, enabling developers to verify that the changes on each branch do not conflict. Many software teams adopt a practice of branching for each feature or perhaps branching for each release, or both.
c. **Traceability.** Being able to trace each change made to the software and connect it to project management and bug tracking software. Annotate each change with a message describing the purpose and intent of the change can help not only with root cause analysis and other forensics.

1.3 What is Git?

The most widely used modern version control system in the world today is **Git**. Developers who have worked with Git are well represented in the pool of available software development talent and it works well on a wide range of operating systems and IDEs (Integrated Development Environments).

1.4 Why do we need Git?

Continuous Integration and code reviews help ship sustainability (less bugs), efficiency (good performance) and maintainability (great customer satisfaction) software. Sustainability is about good estimation, effective branching strategies for managing code, automated testing to protect quality, and continuous deployment to get fast feedback from users. Adopting sustainable development practices requires a discipline most of us aspire to–but often struggle to realize–as individuals.

It empowers individuals with practices that build a solid technical foundation into their product and a culture of collaboration into their team. Software development is all about teamwork, which is no surprise since most software today is built by teams. Developers build strong relationships with product management, design, QA, and operations because writing sustainable code means staying connected to all facets of the project.

Better code, less redundancy and error (i.e. duplication of effort and/or conflicting streams of work), and more effective cross-functionalism are just a few of the benefits.

2. Window Installation

1. Download the latest Git for Windows installer via the link: https://gitforwindows.org
2. When you've successfully started the installer, you should see the Git Setup wizard screen. Follow the Next and Finish prompts to complete the installation. The default options are pretty sensible for most users.
3. Open a Command Prompt (or Git Bash if during installation you elected not to use Git from the Windows Command Prompt).
4. Configure your Git username and email using the following commands.

```
$ git config --global user.name “yinghua”
$ git config --global user.email “yinghua@smartblock.pro”
```

3. Mac Installation

3.1 Git for Mac Installer

1. Down the installer from : https://sourceforge.net/projects/git-osx-installer/files/
2. Follow the prompt to install Git.
3. Open the terminal and verify the installation was successful by typing the following command:

```
$ git --version
git version 2.18.0.
```

4. Configure your Git username and email using the following commands.

```
$ git config --global user.name “yinghua”
$ git config --global user.email “yinghua@smartblock.pro”
```

3.2 Install Git with Homebrew

1. If you have Homebrew package in OSX, follow these instruction to install Git.

```
$ brew install git
```

2. Verify the installation was successful by typing which git --version:

```
$ git --version
Git version 2.18.0
```

3. Configure your Git username and email using the following commands.

```
$ git config --global user.name “yinghua”
$ git config --global user.email “yinghua@smartblock.pro”
```

4. Sample Git commands.

4.1 Git Clone

Git clone is a Git command line utility which is used to target an existing repository and create a clone, or copy of the target repository and download into your local computer.

```
$ mkdir calculator-apps
$ cd calculator-apps/
$ git clone https://github.com/CodesAreHonest/tdd-calculator.git
Cloning into 'tdd-calculator'...
remote: Counting objects: 90, done.
remote: Total 90 (delta 0), reused 0 (delta 0), pack-reused 90
Unpacking objects: 100% (90/90), done.
```

4.2 Git Init

git init command creates a new Git repository. It can be used to convert an existing, unversioned project to a Git repository or initialize a new, empty repository. Most other Git commands are not available outside of an initialized repository, so this is usually the first command you'll run in a new project.

```
$ mkdir git-workshop-iscity
$ cd git-workshop-iscity
$ git init
Initialized empty Git repository in /Users/iscity/Documents/git-workshop-iscity/.git/
```

4.3 Git Status

Displays paths that have differences between the index file and the current HEAD commit, paths that have differences between the working tree and the index file, and paths in the working tree that are not tracked by Git.

```
$ cp -r tdd-calculator/ ~/Documents/git-workshop-iscity/
$ cd git-workshop-iscity
$ git status
On branch master
No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .bowerrc
        .gitignore
        .jshintrc
        .travis.yml
        LICENSE
        README.md
        app/
        bower.json
        doc/
        img/
        karma.conf.js
        package.json
nothing added to commit but untracked files present (use "git add" to track)
```

4.4 Git Add

The git add command adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit. However, git add doesn't really affect the repository in any significant way—changes are not actually recorded until you run git commit.

```
$ git add .
// the ‘dots’ means everything that contains in your file.

$ git add
Nothing specified, nothing added.
// without the ‘dots’ nothing will be added onto the staging area.
```

4.5 Git Commit

Commits are the core building block units of a Git project timeline. Commits can be thought of as snapshots or milestones along the timeline of a Git project. Commits are created with the git commit command to capture the state of a project at that point in time. Git Snapshots are always committed to the local repository.

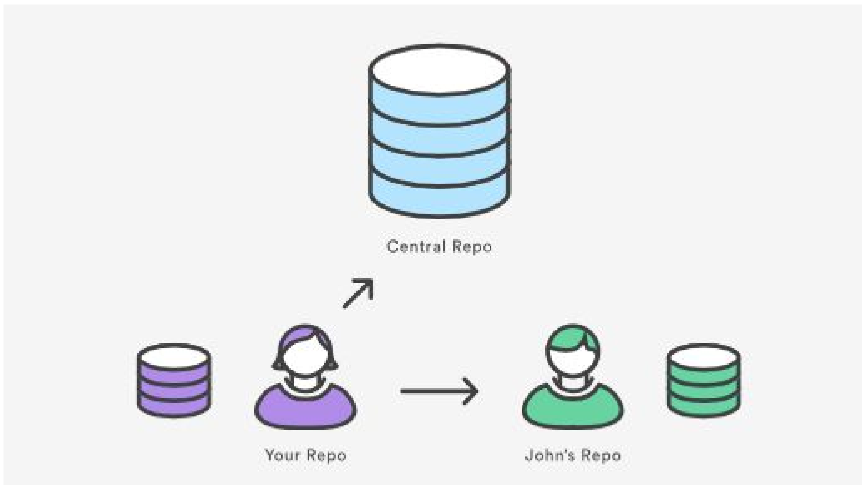
Git doesn't force you to interact with the central repository until you're ready. Just as the staging area is a buffer between the working directory and the project history, each developer's local repository is a buffer between their contributions and the central repository.

```
$ $ git commit -m "add calculator apps for initial commit to show how git works."
[master (root-commit) 14087a7] add calculator apps for initial commit to show how git works.
 36 files changed, 1237 insertions(+)

 create mode 100644 .bowerrc
 create mode 100644 .gitignore
 create mode 100644 .jshintrc
 create mode 100644 .travis.yml
 create mode 100644 LICENSE
 create mode 100644 README.md
 create mode 100644 app/calculator-app/calculator.css
 create mode 100644 app/calculator-app/calculator.html
 create mode 100644 app/calculator-app/calculator.js
 create mode 100644 app/calculator-app/calculatorKey.js
 create mode 100644 app/calculator-app/calculatorSpec.js
 create mode 100644 app/calculator-app/image/background.jpg
 create mode 100644 app/calculator-app/image/wallpaper.jpg

… Other files are not specified
```

4.6 Git remote add

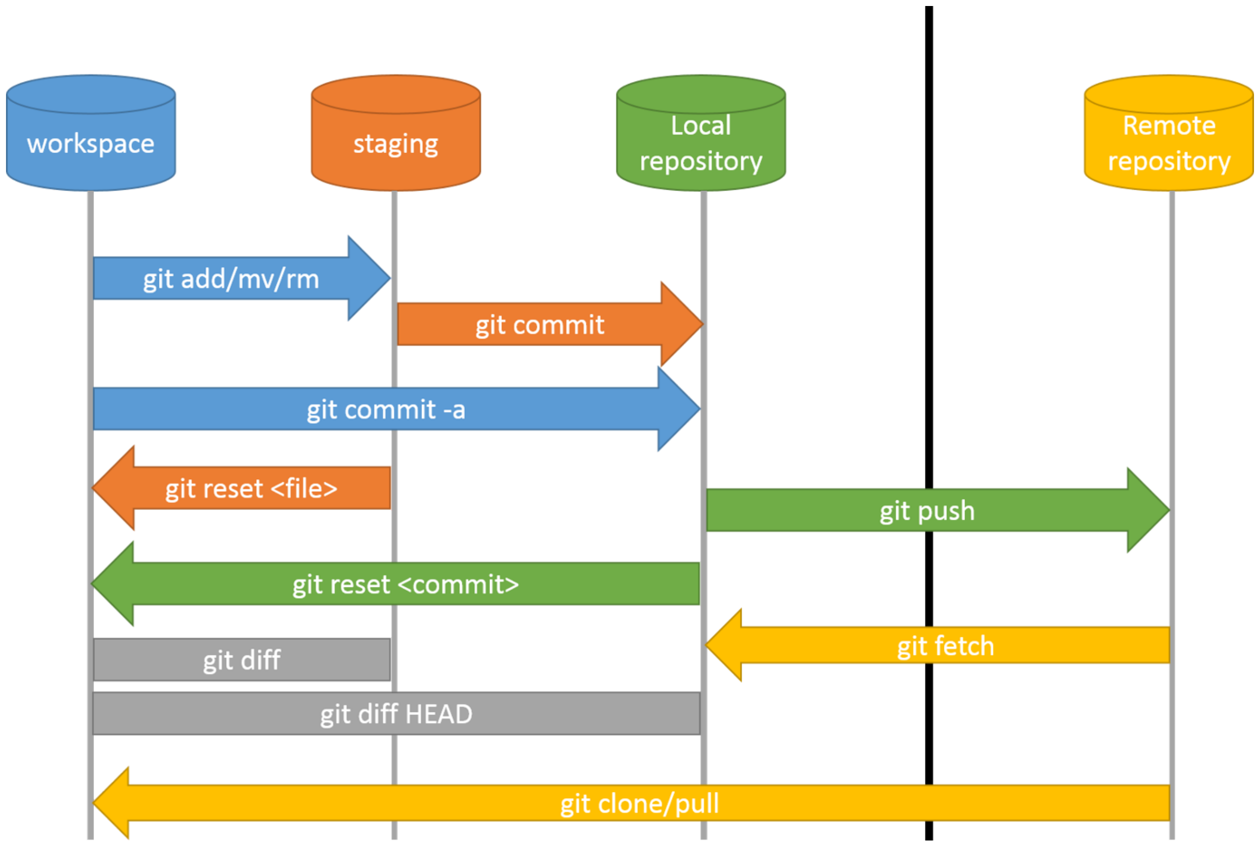
The git remote command is one piece of the broader system which is responsible for syncing changes and establish connection to other repositories in Github.

Records registered through the git remote command are used in conjunction with the git fetch, git push, and git pull commands. These commands all have their own syncing responsibilities which can be explored on the corresponding links. Remote connections are more like bookmarks rather than direct links into other repositories.



```
$ git remote -v
yinghua:git-workshop-iscity iscity$ git remote add origin
https://github.com/chai-yinghua/git-workshop-iscity.git

// list out all the available remote
yinghua:git-workshop-iscity iscity$ git remote -v
origin  https://github.com/chai-yinghua/git-workshop-iscity.git (fetch)
origin  https://github.com/chai-yinghua/git-workshop-iscity.git (push)
```

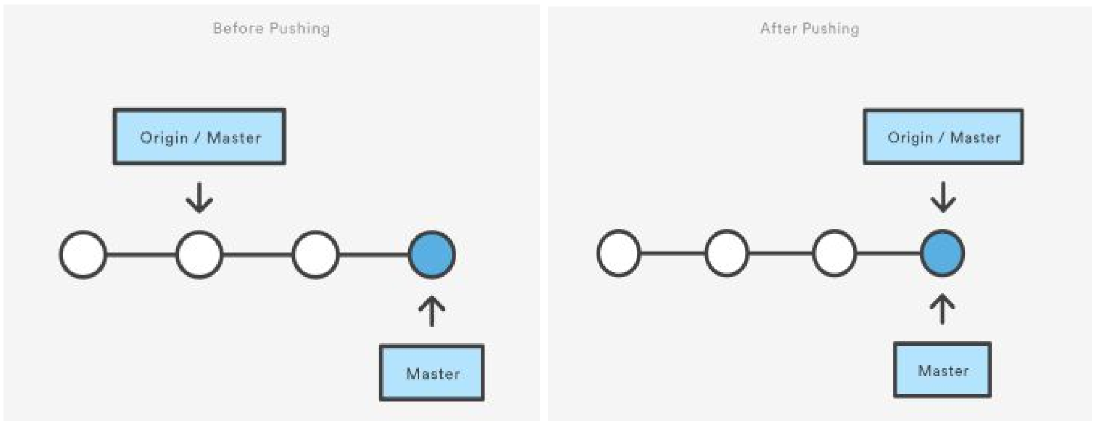
4.7 Git remote add



5. Synchronization with repositories.

5.1 Git push

The git push command is used to upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repo.



git push is most commonly used to publish an upload local changes to a central repository. After a local repository has been modified a push is executed to share the modifications with remote team members.

```
$ git push
fatal: The current branch master has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin master

$ git push -u origin master
Counting objects: 48, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (44/44), done.
Writing objects: 100% (48/48), 770.14 KiB | 10.41 MiB/s, done.
Total 48 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), done.
To https://github.com/chai-yinghua/git-workshop-iscity.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

5.2 Git pull

The git pull command is used to fetch and download content from a remote repository and immediately update the local repository to match that content.

```
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/chai-yinghua/git-workshop-iscity
   09a0842..8eb8fbe  master     -> origin/master
Updating 09a0842..8eb8fbe
Fast-forward
 README.md | 4 +---
 1 file changed, 1 insertion(+), 3 deletions(-)

$ git pull origin master
From https://github.com/chai-yinghua/git-workshop-iscity
 * branch            master     -> FETCH_HEAD
Already up to date.
```

5.3 Git fetch

The git fetch command downloads commits, files, and refs from a remote repository into your local repo. Fetching is what you do when you want to see what everybody else has been working on.

```
$ cd ~/Documents/gcox_exchange // Change directory to GCOX Exchange for example.
$ git fetch
remote: Counting objects: 48, done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 48 (delta 34), reused 48 (delta 34), pack-reused 0
Unpacking objects: 100% (48/48), done.
From https://github.com/smartblock/gcox-acm
   2f5934e..fb74ad8  implement_new_kyc -> origin/implement_new_kyc
```

6. Inspect a repositories.

6.1 Git Status

The usage is mentioned in section 4.c

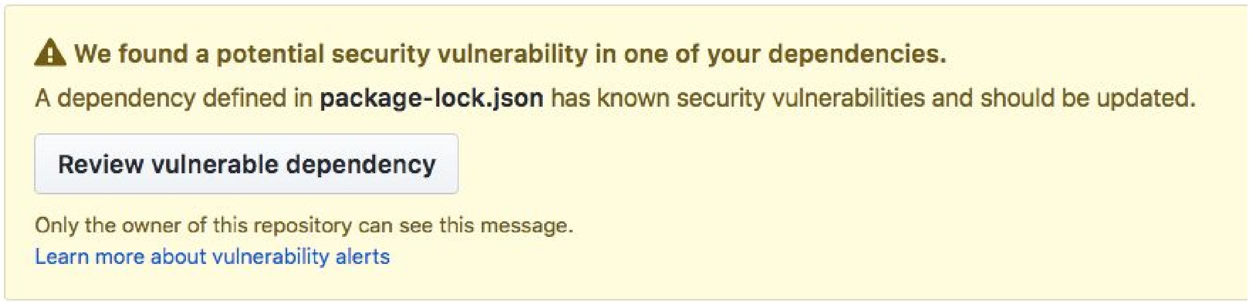
6.2 Git log

Git log allow you to navigate all the history of changes in your codes. This gives you the power to go back into your project history to see who contributed what, figure out where bugs were introduced, and revert problematic changes with git reset.

```
$ git log
commit a2d795eb2c68ac5cfab4d6dde6e8ba433d542a82 (HEAD -> master, origin/master)
Author: yinghua <yinghua@smartblock.pro>
Date:   Sun Jun 24 22:35:29 2018 +0800

    Revert "Update README.md"

    This reverts commit 8eb8fbeebb22773026a845396e33494e16b0cb2b.

commit f80e2d99cca2388ddaaeb8c148f09c038299ed43
Author: yinghua <yinghua@smartblock.pro>
Date:   Sun Jun 24 22:33:58 2018 +0800

    Revert "ignore package-lock.json to prevent angular's dependencies exposed"

    This reverts commit 09a084290842d408d342a6e57976104f1e1f5fd3.

// pretty print the log in one line
$ git log --oneline
a2d795e (HEAD -> master, origin/master) Revert "Update README.md"
f80e2d9 Revert "ignore package-lock.json to prevent angular's dependencies exposed"
8eb8fbe (tag: v0.1.0) Update README.md
09a0842 ignore package-lock.json to prevent angular's dependencies exposed
14087a7 add calculator apps for initial commit to show how git works.
```

6.3 Git ignore

Specifies the file and directory name in .gitignore files to untracked by Git to prevent push onto the Github server.



We found a potential security vulnerability in one of your dependencies.

A dependency defined in **package-lock.json** has known security vulnerabilities and should be updated.

Only the owner of this repository can see this message.
Learn more about vulnerability alerts

```
// Inside .gitignore files.

logs/*
!.gitkeep
node_modules/
e2e-tests/
bower_components/
Package-lock.json ← new lines is added to untrack the files.

$ git commit -m "iscity$ untrack package-lock.json in .gitignore to remove security
vulnerabilities."
[master 88ff5a4] iscity$ untrack package-lock.json in .gitignore to remove security
vulnerabilities.
 1 file changed, 1 insertion(+)
yinghua:git-workshop-iscity iscity$ git push origin master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 349 bytes | 349.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/chai-yinghua/git-workshop-iscity.git
   7660727..88ff5a4  master -> master

// Removed the file in Github.
$ git rm --cached package-lock.json
rm 'package-lock.json'
```

7. Undoing changes in a repository.

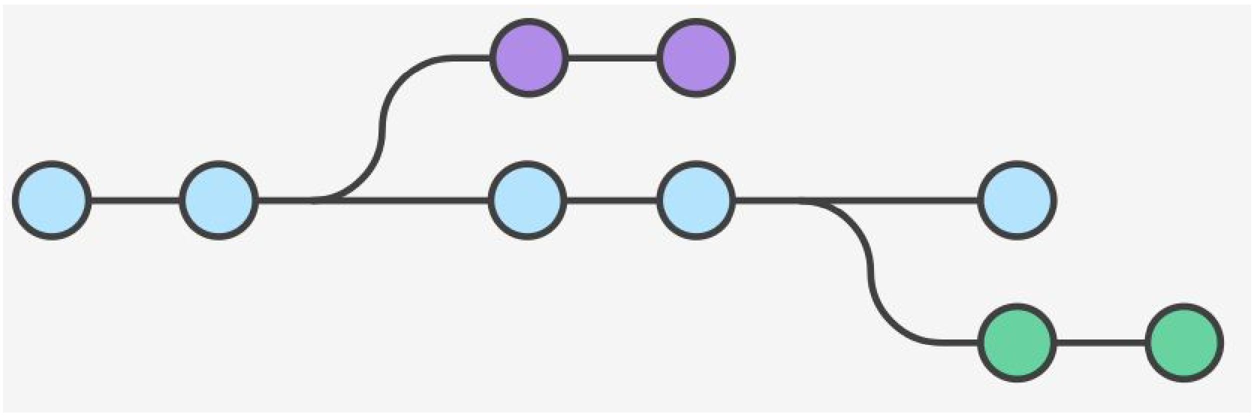
7.1 Git checkout

Git checkout is an easy way to “load” any of these saved snapshots onto your development machine. During the normal course of development, the HEAD usually points to master or some other local branch, but when you check out a previous commit, HEAD no longer points to a branch—it points directly to a commit.

In Git terms, a "checkout" is the act of switching between different versions of a target entity. The git checkout command operates upon three distinct entities: files, commits, and branches. (will be discussed in later section)

```
$ git log --oneline
b818da2 (HEAD -> master, origin/master) remove several information in README.md
5fa1dbe remove several information in README.md
88ff5a4 iscity$ untrack package-lock.json in .gitignore to remove security vulnerabilities.

$ git checkout 88ff5a4
Note: checking out '88ff5a4'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 88ff5a4 iscity$ untrack package-lock.json in .gitignore to remove security
vulnerabilities

$ git status
HEAD detached at 88ff5a4
nothing to commit, working tree clean

// return to original HEAD
$ git checkout master
Previous HEAD position was 88ff5a4 iscity$ untrack package-lock.json in .gitignore to remove
security vulnerabilities.
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```

8. Branches with Git



In Git, branches are a part of your everyday development process. Git branches are effectively a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to encapsulate your changes. This makes it harder for unstable code to get merged into the main code base, and it gives you the chance to clean up your future's history before merging it into the main branch.

a. List all the branch

```
$ git branch
```

b. Create a new branch

```
$ git branch new-branch
$ git branch
* master
  new-branch
```

c. Rename a branch

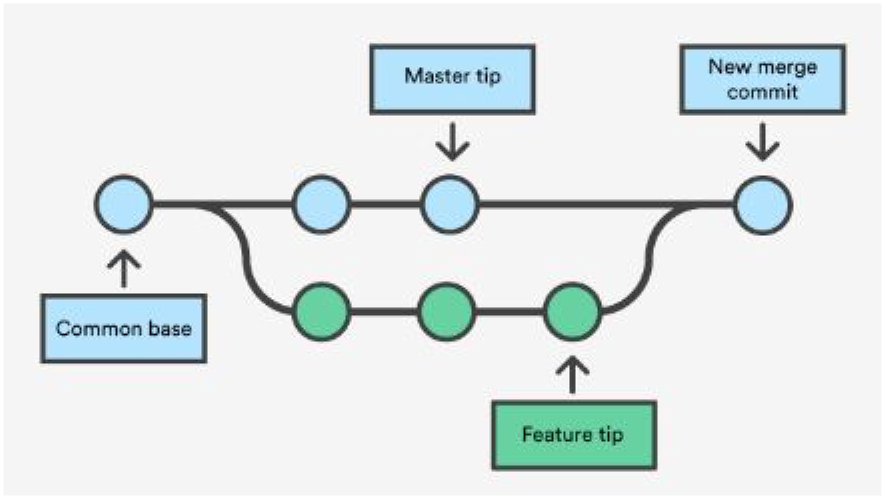
```
$ git branch -m new-branch old-branch
$ git branch
* master
  old-branch
```

d. Switch to a branch

```
$ git checkout old-branch
Switched to branch 'old-branch'
```

e. Merge a branch

In the most frequent use cases, git merge is used to combine two branches.



```
// Step 1 - Confirm the receiving branch
$ git branch
* master
  old-branch

$ git merge old-branch
Updating b818da2..6cd7f45
Fast-forward
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

f. Delete a branch

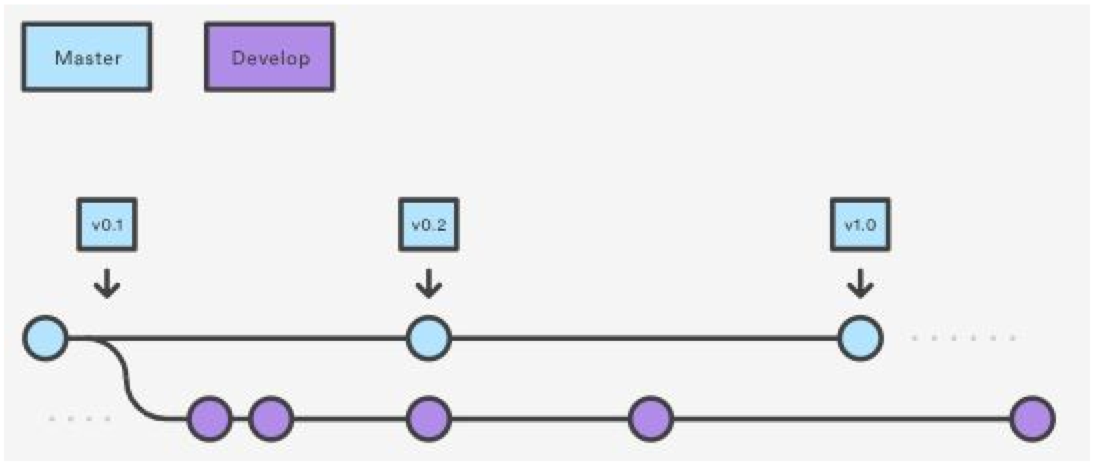
```
$ git branch
  master
* old-branch

$ git branch -d old-branch
error: Cannot delete branch 'old-branch' checked out at
'/Users/iscity/Documents/git-workshop-iscity'

$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

$ git branch
* master
  old-branch

$ git branch -d old-branch
Deleted branch old-branch (was b818da2).

$ git branch
* master
```

9. Git flow workflow.

Install the Git Flow extension to initialize the workflow.

```
$ git flow init

Which branch should be used for bringing forth production releases?
   - develop
   - master
Branch name for production releases: [master]

Which branch should be used for integration of the "next release"?
   - develop
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
```

9.1 The main branches.



- Master.

We consider origin/master to be the main branch where the source code of HEAD always reflects a production-ready state.

- Develop.

We consider origin/develop to be the main branch where the source code of HEAD always reflects a state with the latest delivered development changes for the next release.

When the source code in the develop branch reaches a stable point and is ready to be released, all of the changes should be merged back into master somehow and then tagged with a release number.

```
$ git branch develop // create a develop branch
$ git push -u origin develop // push the change to develop branch
```

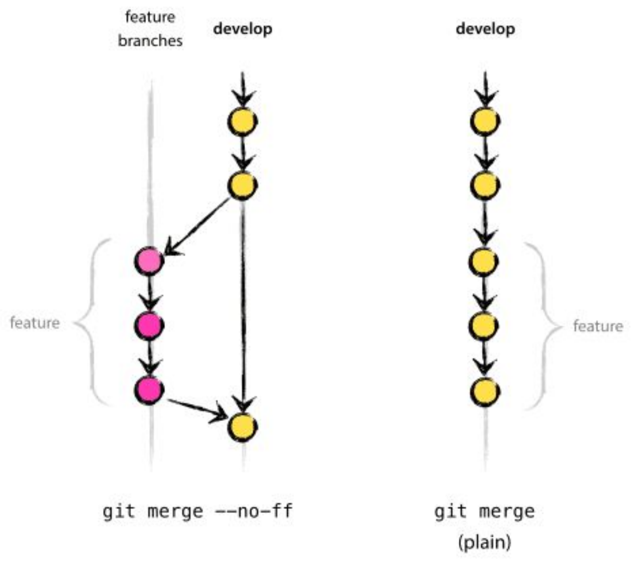
9.2 The supporting branches.

The different types of branches we may use are:

- Feature branches
- Release branches
- Hotfix branches

9.2.1 Feature branch

When starting work on a new feature, branch off from the develop branch.



```
$ git flow feature start new-feature
Switched to a new branch 'feature/new-feature'

Summary of actions:
- A new branch 'feature/new-feature' was created, based on 'develop'
- You are now on branch 'feature/new-feature'

Now, start committing on your feature. When done, use:

     git flow feature finish new-feature / git checkout develop && git merge new-feature
```

Finished features may be merged into the develop branch to definitely add them to the upcoming release.

May branch off from:	develop
Must merge back into:	develop
Branch naming convention:	anything except master, develop, release-*, or hotfix-*

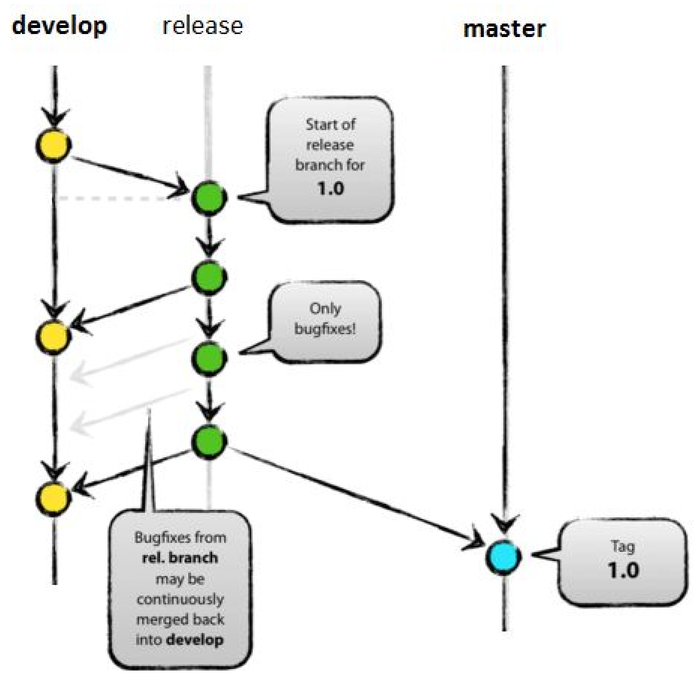
Feature branches (or sometimes called topic branches) are used to develop new features for the upcoming or a distant future release.

When starting development of a feature, the target release in which this feature will be incorporated may well be unknown at that point. The essence of a feature branch is that it exists as long as the feature is in development, but will eventually be merged back into develop (to definitely add the new feature to the upcoming release) or discarded (in case of a disappointing experiment).

```
$ git flow feature finish new-feature
Switched to branch 'develop'
Already up to date.
Deleted branch feature/new-feature (was 6cd7f45).

Summary of actions:
- The feature branch 'feature/new-feature' was merged into 'develop'
- Feature branch 'feature/new-feature' has been removed
- You are now on branch 'develop'
```

Feature branches typically exist in developer repos only, not in origin.

9.2.2 Release branch


develop
release
master
Start of release branch for 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Release branches support preparation of a new production release. They allow for minor bug fixes and preparing meta-data for a release (version number, build dates, etc.). By doing all of this work on a release branch, the develop branch is cleared to receive features for the next big release.

The key moment to branch off a new release branch from develop is when develop (almost) reflects the desired state of the new release. At least all features that are targeted for the release-to-be-built must be merged in to develop at this point in time. All features targeted at future releases may not—they must wait until after the release branch is branched off.

It is exactly at the start of a release branch that the upcoming release gets assigned a version number—not any earlier. Up until that moment, the develop branch reflected changes for the "next release", but it is unclear whether that "next release" will eventually become 0.3 or 1.0, until the release branch is started. That decision is made on the start of the release branch and is carried out by the project's rules on version number bumping.

Create a release branch

```
$ git flow release start v1.0.0
Summary of actions:
- A new branch 'release/v1.0.0' was created, based on 'develop'
- You are now on branch 'release/v1.0.0'

Follow-up actions:
- Bump the version number now!
- Start committing last-minute fixes in preparing your release
- When done, run:

     git flow release finish 'v1.0.0'
```

Add a tag for the release branch to indicate a version of the system.

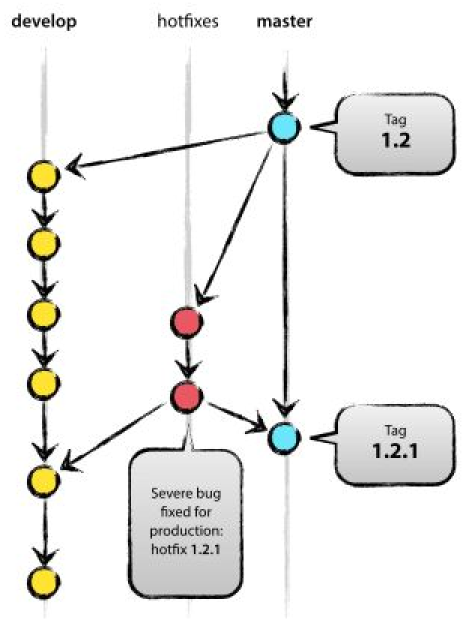
```
$ git tag v1.0.0
$ git push origin v1.0.0
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/chai-yinghua/git-workshop-iscity.git
 * [new tag]         v1.0.0 -> v1.0.0
```

Merge the release branch into Develop and Master.

```
$ git flow release finish v1.0.0
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
Deleted branch release/v1.0.0 (was 6cd7f45).

Summary of actions:
- Latest objects have been fetched from 'origin'
- Release branch has been merged into 'master'
- The release was tagged 'v1.0.0'
- Release branch has been back-merged into 'develop'
- Release branch 'release/v1.0.0' has been deleted
```

May branch off from:	develop
Must merge back into:	Develop and master
Branch naming convention:	release-*

9.2.3 Hotfix branch



When a critical bug in a production version must be resolved immediately, a hotfix branch may be branched off from the corresponding tag on the master branch that marks the production version.

The essence is that work of team members (on the develop branch) can continue, while another person is preparing a quick production fix.

May branch off from:	master
Must merge back into:	develop and master (situational)
Branch naming convention:	hotfix-*

Create a hotfix branch from master branch:

```
$ git flow hotfix start fix-calculator-button
Summary of actions:
- A new branch 'hotfix/fix-calculator-button' was created, based on 'master'
- You are now on branch 'hotfix/fix-calculator-button'

Follow-up actions:
- Bump the version number now!
- Start committing your hot fixes
- When done, run:

     git flow hotfix finish 'fix-calculator-button'
```

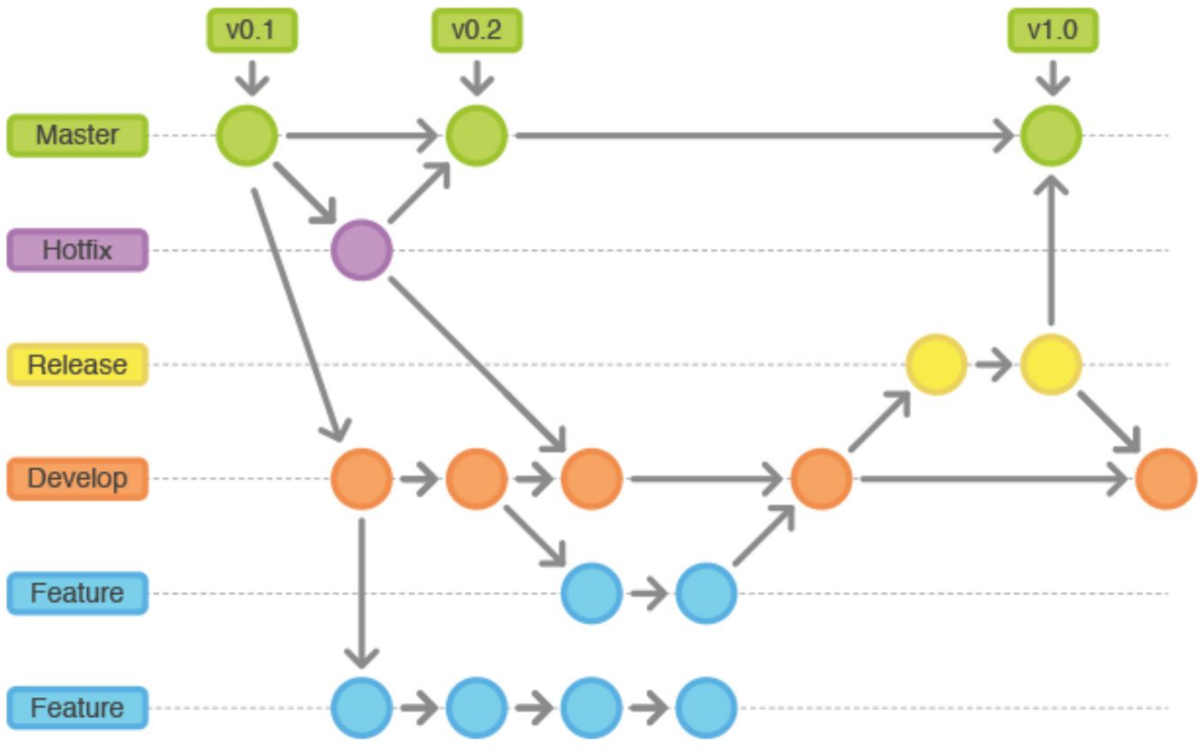
Add a tag for the release branch to indicate a version of the system.

```
$ git tag v1.0.1 -m "fix calculator button"
$ git push origin hotfix/fix-calculator-button
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/chai-yinghua/git-workshop-iscity.git
 * [new tag]         v1.0.1 -> v1.0.1
```

Merge the release branch into Master.

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

$ git merge fix-calculator-button
$ git branch -d fix-calculator-button
```

10. Summary



The overall flow of Gitflow is:

- A develop branch is created from master
- A release branch is created from develop
- Feature branches are created from develop
- When a feature is complete it is merged into the develop branch
- When the release branch is done it is merged into develop and master
- If an issue in master is detected a hotfix branch is created from master
- Once the hotfix is complete it is merged to both develop and master