Git In-depth

Git In-depth

In the Software Engineering document, we have read about *Versioning Control Systems* and one of those systems is *Git*. This document is based on the Atlassian Git Tutorial which gives a more in-depth look at what is Git and how it is used within an organization. The comprehensive Git documentation is available here.

What is Git

Git is the most widely used modern version control system in the world today. It is an open source project originally developed in 2005 by Linus Torvalds, it is still still actively maintained today. Git has a distributed architecture therefore it is also a Distributed Version Control System.

Git is the de facto standard software versioning tool because of its performance, security and flexibility. The algorithms implemented inside Git are optimized to take advantage of the deep knowledge about common attributes of real source code file trees, how they are usually modified over time and what the access patterns are. The object format of Git's repository files uses a combination of delta encoding (storing content differences), compression and explicitly stores directory contents and version metadata objects.

For any organization that relies on software development, security is a top priority. The codes and change history have to be protected from both accidental and malicious changes and histories must remain fully traceable. Therefore, Git repositories are secured with a cryptographically secure hashing algorithm called SHA1.

Git is also flexible in several aspects, such as, support for various kinds of nonlinear development workflows, efficiency in both small and large projects and its compatibility with many existing systems and protocols. Git also has many integrations with IDEs, client tools (such as SourceTree), project tracking software (such as Jira) and code hosting services (such as GitHub or GitLab)

In addition to being open source, the Git project is well supported by maintainers to meet the long term needs of its users with regular releases that improve usability and functionality. Git also has great community support and a vast user base of documentation that includes books, tutorials (video and podcasts) and dedicated web sites.

Git for Developers

One of the biggest advantages of Git is its branching capabilities. Unlike centralized version control systems, Git branches are cheap and easy to merge. This facilitates the feature branch workflow popular with many Git users. Refer to figure 1 below.

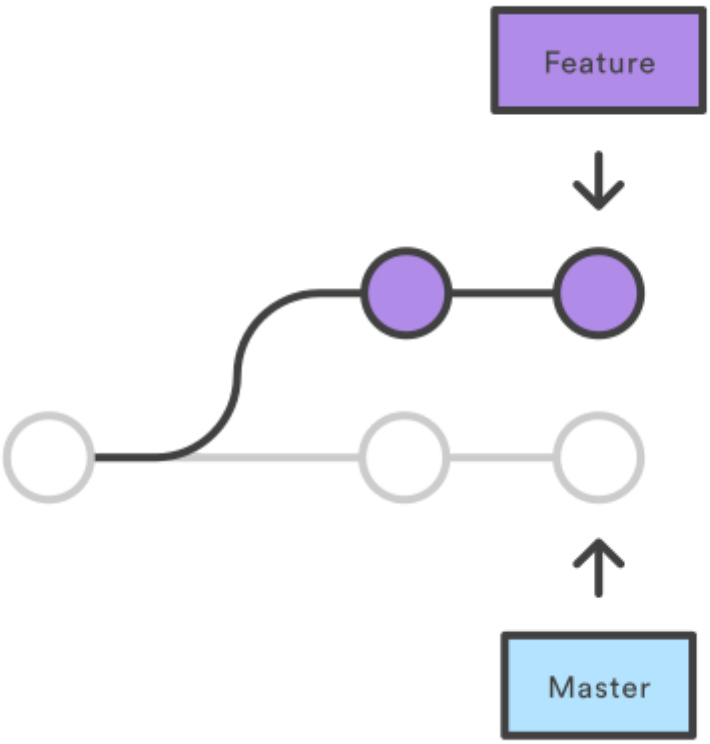

Figure 1: Feature branch in Git.

Feature branches provide an isolated environment for every change to your codebase. When a developer wants to start working on something (no matter how big or small) they create a new branch. This ensures that the master branch always contains production quality code. The reason for using feature branches is the added benefit of representing the development work at the same level of detail as the development task in an Agile Software Development workflow. It also helps if the organization uses Jira tickets to address feature branches.

With Git being a distributed version control system, every developer has their own local repository, complete with a full history of commits. This local repository allows local commits and inspection of file content differences & commits. Recovering from a messed up local repository is also as simple as cloning the repository from someone else's and starting anew.

After a feature branch is done, a *pull request* is requested to merge the feature branches into the main repository. During these *pull requests* the code is reviewed by the team lead before before merging into the main repository. This makes it easier for project leads to keep track of changes, and conduct code reviews. Junior developers can be confident that they aren't destroying the entire project by treating pull requests as a formal code review.

Git can also be configured to deploy the most recent commit from the development branch to a test server whenever anyone merges a pull request into it. Combining this kind of build automation with peer reviews means that the product moves from development to staging to production with the highest possible assurance.

Installation

Git can be installed on all 3 platforms: Mac OS X, Windows, Linux. There are 2 types of Git interfaces: Terminal or User interface. Only the Windows Installation is listed, for the other platform installation, refer to this website.

Git Stand-alone Installer

This is the official Git Stand-alone Installer from Git. It can be downloaded from here.

1. Download the latest Git release.
2. Install Git by following the instructions on the Git Setup wizard.
3. Open a Command Prompt (or Git Bash if during installation you elected not to use Git from the Windows Command Prompt).
4. Run the following commands to configure your Git username and email (make sure to include the double quotation marks). These details will be associated with any commits that you create.

```
$ git config --global user.name "<your name>"
$ git config --global user.email "<your email>"
```

5. Install the **optional** *Git credential manager for Windows*. This will be required if you are interacting with a remote repository that requires you to supply a username & password combination. This credential manager is used to store these credentials so that you do not have to enter it every time you interact with the remote repository.

Git with Atlassian Sourcetree

There are not many free Git clients for Windows. Sourcetree from Atlassian is one of them. It can be downloaded from here.

Getting Started

Setting up a Repository

A repository can be created from 2 ways: locally or by cloning an existing repository.

Creating a Local Repository

Assuming that you already have an existing project folder that you would like to create a repository within, navigate to that folder with the change directory command (`cd`) in Terminal then execute the `git init` command.

```
$ cd /path/to/your/existing/code
$ git init
```

The `git init` command will initialize the folder with the required Git files. Executing this command in any existing project directory will execute the same initialization setup as mentioned above, but scoped to that project directory.

Cloning an Existing Repository

If a project has already been set up in a central repository, the clone command is the most common way for users to obtain a local development clone. Like `git init`, cloning is generally a one-time operation and done in the project directory where the repository is supposed to be housed on your local machine. Remember to navigate to the project directory before cloning.

```
$ git clone <repo url>
```

`git clone` is used to create a copy or clone of remote repositories. You pass it a repository URL. Git supports a few different network protocols and corresponding URL formats. If the organization is using the Git SSH protocol, the Git SSH URLs will follow a template of:

```
git@HOSTNAME:USERNAME/REPONAME.git
```

where the template values match:

- `HOSTNAME: bitbucket.org`
- `USERNAME: rhyolight`
- `REPONAME: javascript-data-store`

If the organization is using an internal `HTTPS` proxy, you will need to setup the proxy configuration for Git otherwise Git will connect to the Internet and try to search for the internal proxy URL.

```
# for a general http proxy server
$ git config --global http.proxy
http://proxyUsername:proxyPassword@proxy.server.com:port

# for a specific domain on the https proxy server
$ git config --global http.https://domain.com.proxy
http://proxyUsername:proxyPassword@proxy.server.com:port
```

After setting up the proxy settings, use the `git clone` command and enter the repository URL. Proxy servers repository URL have the same from as normal web URLs except that it ends with a `.git`.

```
$ git clone https://somedomain/somefolder/REPONAME.git
```

By default, the repository name is the `REPONAME` defined in the URL but you can change it by adding a different folder name as the last parameter:

```
$ git clone https://somedomain/somefolder/REPONAME.git new-REPONAME
```

After execution, the latest version of the remote repository files on the master branch will be cloned and added to a new folder. The new folder will be named after the `REPONAME`. The folder will contain the full history of the remote repository and a newly created master branch.

Saving Changes

When working in Git, or other version control systems, the concept of "saving" is similar but not the same as the process of saving in a word processor or other traditional file editing applications. The traditional software expression of "saving" is synonymous with the Git term "committing" where a Git commit is an operation that acts upon a collection of files and directories instead of the regular file system operation.

The commands: `git add` and `git commit` are all used in combination to save a snapshot of a Git project's current state. In addition, `git stash` is used to store changes that are not ready to be committed. A Git repository can also be configured to ignore specific files or directories using the `.gitignore` file. Content to be ignored are listed using the relative path from the project directory to either the file or the folder itself. This will prevent Git from saving changes to any ignored content.

The `git add` and `git commit` commands compose the fundamental Git workflow. These are the two commands that every Git user needs to understand, regardless of their team's collaboration model. They are the means to record versions of a project into the repository's history.

- `git add` will add the changed files to a staging area to save a copy of the current state of the project. Eg: `git add hello.py`
- `git commit` will capture a snapshot of the state of the project at that point in time and save it to the local repository. DO make sure that you write a proper description for each commit via the `-m` options as it will help your team members know what you have changed. Eg: `git commit -m "<message>"`
- `git reset` is used to undo a commit or staged snapshot done via `git add`.

Once the changes are done the command `git push` is used to send the committed changes to remote repositories for collaboration. This enables other team members to access a set of saved changes.

Comparing changes

Let's say that we would like to compare a file between 2 commits or a local vs remote repository. This can be done via the `git diff` command where it takes two input data sets and outputs the changes between them.

Do note that the output from `git diff` is very different between terminal and user interface (UI) clients. Executing a `git diff` in a terminal window will produce the following output:

```
1 diff --git a/diff_test.txt b/diff_test.txt
2 index 6b0c6cf..b37e70a 100644
3 --- a/diff_test.txt
4 +++ b/diff_test.txt
5 @@ -1 +1 @@
6 -this is a git diff test example
7 +this is a diff example
```

where each line explanations are as follows:

1. **Line 1** - The input sources to the `diff` command. In this case, `a/diff_test.txt` and `b/diff_test.txt` files.
2. **Line 2** - Internal Git metadata. You will most likely not need this information. The numbers in this output correspond to Git object version hash identifiers.

3. **Lines 3 and 4** - The legend of the symbols given to each input source. In this case, changes in file `a/diff_test.txt` are marked by the subtract (`-`) symbol and in the file `b/diff_test.txt`, changes are marked by the plus (`+`) symbol.
4. **Lines 5 to 7** - These are diff "chunks" where it lists the sections of file that have changes. Each chunk is prepended by a header enclosed within `@@` symbols. The content of the header is the summary of changes made to the file. In this case, `-1 +1` means line 1 of the file had changes. A more realistic header would be `@@ -34,6 +34,8 @@` where it means that at line 34, 6 lines have been extracted and 8 lines have been added.

   The remaining content of the diff chunk displays the recent changes. Each changed line is prepended with a `+` or `-` symbol indicating which version of the diff input the changes come from.

With a user interface Git client, the differences can be seen on both the history tab of the project or from the staged area before committing the snapshot of project. The `git add` command is done automatically by the UI Git client. Refer to figure 2 below.

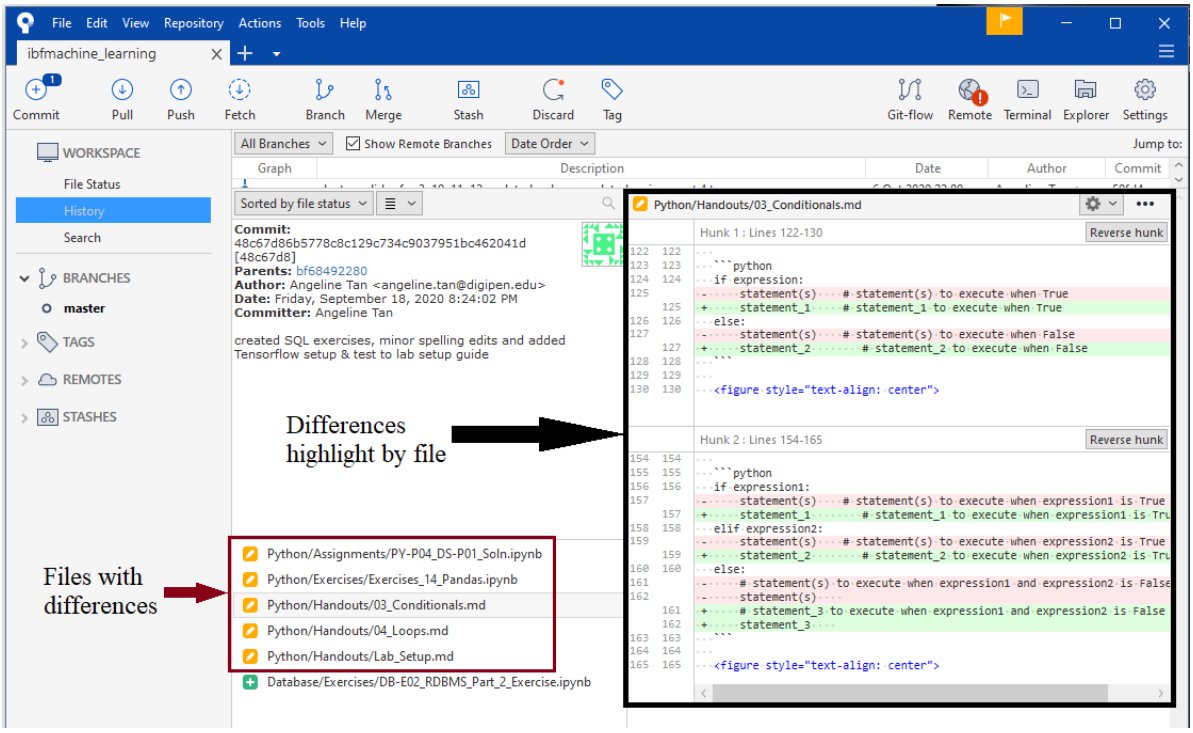

Figure 2: Screenshot of the SourceTree Git client. The project's history is shown.

Stashes

Stashes are temporarily shelves (or *stashes*) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on. Stashing is used when you need to switch between your current edited work and a *clean working copy* of your previous work. This is mainly used in scenarios where branches are *merged* or *pulling* from a remote branch or *checking out* a different branch.

The command used is `git stash`. Once your work has been stashed, you are free to make changes, create new commits, switch branches, and perform any other Git operations; then come back and re-apply your stash when you're ready. **Note** that the stash is local to your Git repository; stashes are not transferred to the server when you push.

The common `git stash` options are:

- `git stash pop` - removes the changes from your **most recent** stash and reapplies them to your working copy.

- `git stash apply` - reapply the changes to your working copy **and** still keep them in your stash.
- `git stash save "<message>"` - creates multiple stashes, each with a description. **Note** that to reapply from a particular stash, you need to past it's identifier as the last argument; for example, `git stash pop stash@{2}` will reapply changes from stash with the id 2.
- `git stash list` - to list all the stashes
- `git stash drop stash@{n}` - to delete a particular stash with then *nth* id. To delete all the stashes, use `git stash clear`.

Inspecting a repository

There are 2 commands that you can use to check the status of a Git repository: `git status` & `git log`. Both commands displays different outputs:

- `git status` - displays the **state of the working directory and the staging area**. It lets you see which changes have been staged, which haven't, and which files aren't being tracked.
- `git log` - displays the **committed snapshots**. It lets you list the project history, filter it, and search for specific changes.

It's good practice to check the state of your repository using `git status` before committing changes so that you don't accidentally commit something you don't mean to. While `git status` is mainly used before and after committing, `git log` is used mainly to search for specific changes in the project's history via feature branches, author's name, author's email, commit message, etc.

Tagging

Tagging is used set major reference points (like version releases) in the project's Git history. There are 2 types of tags: annotated and lightweight tags. The difference between the 2 tags are the amount of extra meta data they store. **Annotated tags** store extra meta data such as: the tagger name, email, and date. They are also normally regarded as public. **Lightweight tags** are essentially "bookmarks" to a commit, they are just a name and a pointer to a commit, useful for creating quick links to relevant commits. They are also normally regarded as private.

Git tags are created via the `git tag` command:

- `git tag -a <tag name>` - for annotated tags. Note that messages can be added to annotated tags via the `-m` option
- `git tag <tag name>` - for lightweight tags

Generally, it is better to use annotated tags over lightweight tags as more information can be stored with them.

The other useful tagging commands are:

- `git tag` - lists all stored tags in a repository. Note that it has **no other options** appended. To refine the list of tags the `-l` option can be passed with a wild card expression. Eg: `git tag -l *-rc*` where `*-rc*` is the wildcard expression that returns a list of all tags marked with a `-rc` prefix
- `git tag -d <tag name>` - deletes the tag with that tag identifier

It is definitely possible to tag old commits but you would need to get the commit's SHA hash number via the `git log` and pass that to `git tag`. Eg: `git tag -a <tag name> <commit's SHA hash>`.

Undoing Changes

Important note Git does not have a traditional "undo'"system like those found in a word processing application therefore do not compare Git undo operations to a traditional sense. Additionally, Git has its own nomenclature for "undo" operations, such as reset, revert, checkout, clean, etc.

Think of Git as a timeline management utility. Commits are snapshots of a point in time or points of interest along the timeline of a project's history. Additionally, multiple timelines can be managed through the use of branches. When "undoing" in Git, you are usually moving back in time, or to another timeline where mistakes didn't happen.

How to find old commits

We know from the previous section that we can use the `git log` command to find out the commit history of the current branch. Of course `git log` is able to view all commits across all branches using the the option `--branches` and a wildcard character. Eg: `git log --branches=*`. To swap branches, we can use the `git branch` command which we will cover in the section of *Branching* in the *Collaborating* section.

Checkout

Once you have found a commit reference to the point in history you want to visit, you can utilize the `git checkout` command to visit that commit. `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. Refer to the blue dot in figure 3 below.

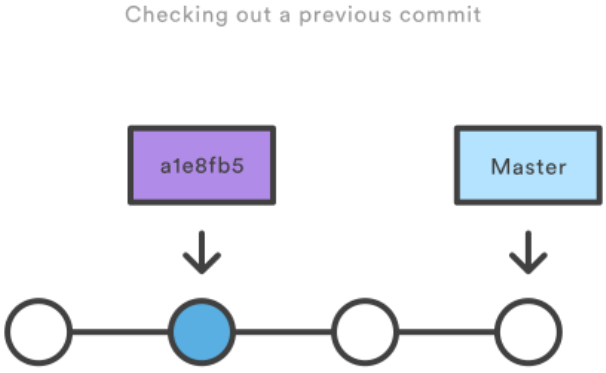

Figure 3: Checking out a previous commit.

The blue dot is called a **detached HEAD** state. However, checking out an old file does not move the `HEAD` pointer. It remains on the same branch and same commit, avoiding a "detached head" state. The reason why we **want to avoid** a detached head state is because changes made to that particular commit will get lost if you re-commit new changes to it. The changes made, **do not propagate** to the other commits between the master and your current point. It is always better to create a branch from that commit, make your changes then merge it back to the master branch, once done. Refer to the code snippet below.

```
$ git checkout -b test-branch a1e8fb5
# ... do your thing ...
$ git checkout master
$ git branch -d test-branch
```

Alternatively, checking out an old file **does not move** the `HEAD` pointer. It remains on the same branch and same commit, avoiding a "detached head" state. You can then commit the old version of the file in a new snapshot as you would any other changes. So, in effect, this usage of `git checkout` on a file, serves as a way to revert back to an old version of an individual file.

The other "undo" Git operations are:

- `git revert` - used for undoing changes to a repository's commit history but does not move the HEAD pointer to the particular commit. The command will inverse the changes from that commit, create a new "revert commit" and update the HEAD pointer to point at this new revert commit therefore this revert commit is now at the tip of the branch. The other commits between the 2 points remain unchanged.
- `git clean` - removes all untracked files. Rarely used.
- `git reset` - it is used to reset the a previous commit or add operation but take caution when done on a shared remote repository.

Collaborating

By this point we should know that Git gives every developer their own copy of the repository, complete with its own local history and branch structure. This section details how Git is used collaborate between each team member's work.

Syncing

For syncing, the Git uses the commands:

- `git remote` - used for creating, viewing, and deleting connections to other repositories. These connections are more like bookmarks rather than direct links into other repositories. Remember when you clone a repository with `git clone`, this automatically creates a remote connection called **origin** pointing back to the cloned repository. You can add other connections to in addition to the origin repositories by using `git remote`.
- `git fetch` - downloads commits, files, and refs from a remote repository into your local repo. This is done to see what are the commits that your other team members have done. This step isolates the new content from your local content. Git will only merge the new content if a `git pull` command is executed after.
- `git push` - used to upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repo. Pushing has the potential to overwrite changes, caution should be taken when pushing.
- `git pull` - used to fetch and download content from a remote repository and immediately update the local repository to match that content. This command is more aggressive alternative to `git fetch`. Uses a combination of `git fetch` and `git merge` commands.

Branching

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.

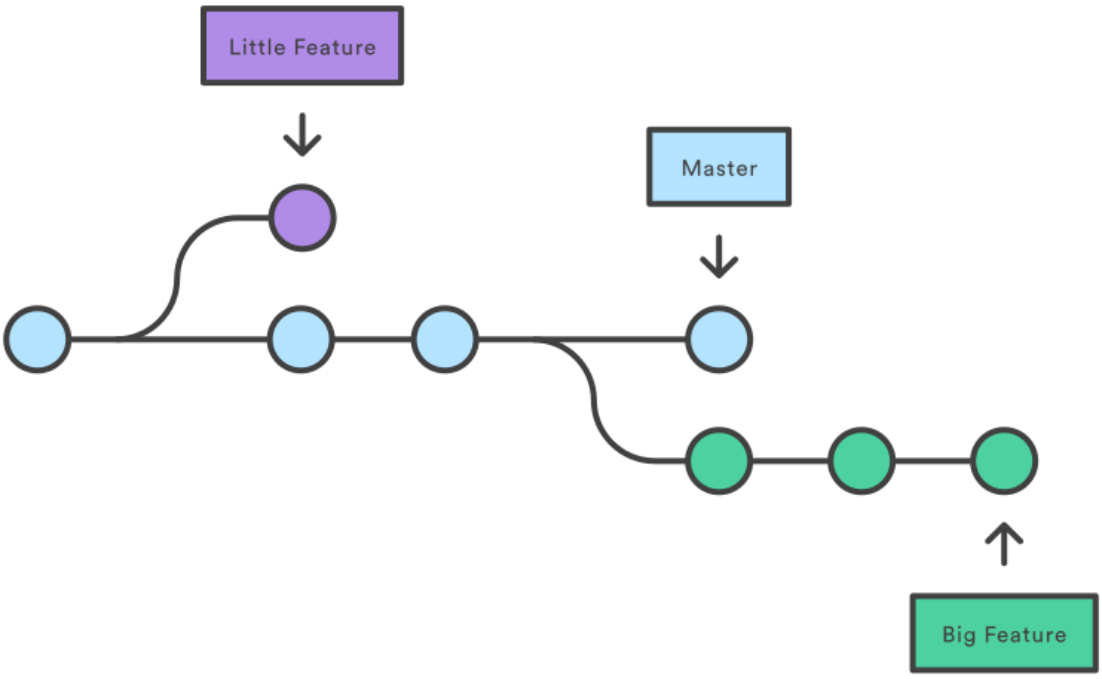

Figure 4: Branching from the main repository.

Figure 4 above visualizes a repository with two isolated lines of development, one for a little feature, and one for a longer running feature. By developing them in branches, it's not only possible to work on both of them in parallel, but it also keeps the main `master` branch free from questionable code.

A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process. You can think of them as a way to request a brand new working directory, staging area, and project history. New commits are recorded in the history for the current branch, which results in a fork in the history of the project.

Git commands for branching are as follows:

- `git branch` - used for creating, listing, renaming and deleting branches. Note that this command does not allow you to switch between branches. Eg: `git branch <branch name>`.
- `git checkout` - from the section on checkout, we have seen the usage of this command on files & commits and its effects. This command command lets you navigate between the branches created by `git branch`. Checking out a branch updates the files in the working directory to match the version stored in that branch, and it tells Git to record all new commits on that branch, essentially creating a new line of development.

  Try not to confuse this command with `git clone`. The difference between the two commands is that clone works to fetch code from a remote repository, alternatively checkout works to switch between versions of code already on the local system.
- `git merge` - merges independent lines of development created by `git branch` and integrate them into a single branch.

Merge Conflicts

From the previous section, we see that `git merge` is used by `git pull` to merge the remote repository content into your active local repository and it is also used in branching to integrate independent lines of development into a single branch (refer to figure 5 below).

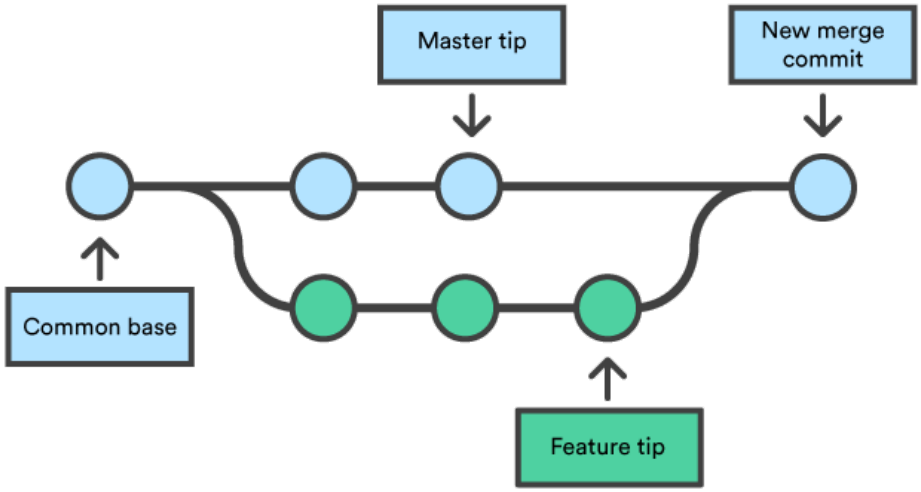

Figure 5: Marging a feature branch to the master.

The general steps one would take to prepare for a merge are:

1. Use `git status` to ensure that `HEAD` is pointing to the correct merge-receiving branch. Use `git checkout` to switch to the receiving branch if required.
2. Fetch the latest remote commits to make sure that the receiving branch and the merging branch are up-to-date with the latest remote changes. Advisable to use a `git fetch` before a `git pull`.
3. Execute a `git merge <feature branch name>` to combine the feature branch with the master branch. **Note** that your current branch must be the receiving branch.

When a merge is requested, Git will attempt to auto magically merge the separate histories together but if Git encounters a piece of data that is changed in both histories (ie, changes made to the same part of the same file) it will be unable to automatically combine them. This results in a **merge conflict** which will need user intervention to continue.

There are 2 types of merge conflicts:

- **Before start of merge** - happens when the working directory or staging area of the current project is not up-to-date due to uncommitted local changes. To stabilize the local state, use either `git stash`, `git checkout`, `git commit` or `git reset` or a combination of those commands.
- **During merging** - happens when there is a conflict between the current local branch and the branch being merged. This indicates a conflict with another developer's code. Git will do its best to merge the files but will leave things for you to resolve manually in the conflicted files.

Git will produce some descriptive output letting us know that a conflict has occurred. We can gain further insight by running the `git status` command. For example, in the code snippet below, it shows a merge conflict during a merge.

```
$ git status
On branch master
You have unmerged paths.
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)

Unmerged paths:
(use "git add <file>..." to mark resolution)

both modified:   merge.txt
```

The output from `git status` indicates that there are unmerged paths due to a conflict. The `merge.text` file now appears in a modified state. Let's examine the file and see what is modified.

```
$ cat merge.txt
<<<<<<< HEAD
this is some content to mess with
content to append
=======
totally different content to merge later
>>>>>>> new_branch_to_merge_later
```

The `cat` command is used to put out the contents of the `merge.txt` file to the terminal window. The new additions

- `<<<<<<< HEAD`
- `=======`
- `>>>>>>> new_branch_to_merge_later`

can be treated as "conflict dividers" where the `=======` line is the "center" of the conflict. All the content between the center and the `<<<<<<< HEAD` line is content that exists in the current branch master which the `HEAD` ref is pointing to. Alternatively all content between the center and `>>>>>>> new_branch_to_merge_later` is content that is present in our merging branch.

The most direct way to resolve a merge conflict is to edit the conflicted file. Open the `merge.txt` file in your favorite editor, edit it then do a commit and push operation. The tutorial gives more details on the Git commands that can be used to help resolve merge conflicts.