Elegant code management using *Git*

Kameswararao Anupindi *

September 25, 2011

Contents

*Graduate Research Assistant, CFD Laboratory,
School of Mechanical Engineering, Purdue University, West Lafayette, IN - 47906

1 *Git* - what is it?

Git is a distributed version control system, which can be used for maintaining a set of files that get revised quite often. These set of files can be for example research source code, or document files shaping up for a book or a thesis. Git stores and maintains revisions for each of the files thus relieving the programmer from the process of making different copies of the same file with small changes and calling them with slightly different which are hard to differentiate at a later time. Revisions, milestones that are reached in the code development, bug fixing all can be stored in *git* for future retrieval.

2 Initial Setup

The normal flow of *git* is described here. In the folder that contains the file to be tracked we have to first initialize git using the following command.

```
>git init
Initializing empty Git repository in PWD
```

Then you will see a (hidden) folder named *.git* being created in the present working directory (PWD). The contents of *.git* are responsible for storing and retrieving different versions of the files etc., so it is important not to alter them. Git only stores the differences between the current version and the previous version of the file hence making it memory efficient.
Let's say the PWD contains three files as shown below:

```
>ls
main.f90   Makefile   subroutines.f90
```

We want to track these files using git, so the first step is to add them to the repository that we just initialized.

```
>git add main.f90 Makefile subroutines.f90
```

or

```
>git add .
```

Subsequently, we can use the status command to see the what git knows about the files that we just added.

```
>git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   Makefile
#       new file:   main.f90
#       new file:   subroutines.f90
#
```

As we can see from the above status message from git, there are three new files which were added to the repository but not committed yet. Next, we will commit the files through an initial commit. So, adding the files is a one time job meaning each time we create a new file and want it to be tracked by git then we first need to add the file to git. But, commit on the other hand is done each time the file is modified. So a commit corresponds to a revision of the file that was already added to the git repository before. Next, we proceed to commit the three files to the repository

```
>git commit .
```

This opens up **vim** editor in the following form with a provision to write a *log* for our commit. In this case, we just write a log message of *Adding new files* and **:wq** the editor.

```
Adding new files
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# Explicit paths specified without -i nor -o; assuming --only paths...
#
# Committer: Kameswararao Anupindi <kanupind@cortado.ecn.purdue.edu>
#
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   Makefile
#       new file:   main.f90
#       new file:   subroutines.f90
#
```

Upon committing the files the following messages are displayed.

```
[master (root-commit) e94f48a] Adding new files
 2 files changed, 6 insertions(+), 0 deletions(-)
 create mode 100644 Makefile
 create mode 100644 main.f90
 create mode 100644 subroutines.f90
```

Now, a git status will tell us that there are no changes to commit, i.e. we just committed our files to *git* repository and did not modify the files after our last commit (as shown below):

```
>git status
# On branch master
nothing to commit (working directory clean)
```

3 Normal Work Flow

Normal work flow involves making changes to the files using your favorite editor (vim/emacs) and committing changes to *git* after realizing that they are worth a commit (meaning the code compiles fine etc.).

```
>vim main.f90 ...edit
>git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   main.f90
#
no changes added to commit (use "git add" and/or "git commit -a")
```

So, as seen above we can keep editing the files and commit them at one point.

```
>git commit -a
```

```
minor changes related to input file reading
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Committer: Kameswararao Anupindi <kanupind@cortado.ecn.purdue.edu>
        #
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   main.f90
#
```

```
[master 93cdbce] minor changes related to input file reading
 1 files changed, 1 insertions(+), 1 deletions(-)
```

4 Dealing with Branches

During the process of code development or maintenance, there are instances where one wants to try out a new idea, or fix a bug, or perform a new implementation (such as parallel version of the code) while retaining the current working version so that at any point one can quickly switch between both of these versions. This can be accomplished in *git* using the concept of **branches**. From the previous log messages produced by *git* we can see that we are on a branch called **master**. The same can be verified using git as follows:

```
>git branch
* master
```

The ***** in front of the master denotes that we are currently on branch master, i.e. the files that we have correspond to this branch. In the present case we have only one branch so there is no confusion. Say, we want to create a new branch (to make the code parallel) from the existing copy of the serial version of the code. The first step is to create a new branch and call it parallel

```
>git branch parallel
```

So we now have two branches, with same content as of now, since we did not make any changes yet to the newly created branch.

```
>git branch -a
* master
  parallel
```

Now, we want to checkout the branch parallel and make changes to it.

```
>git checkout parallel
Switched to branch 'parallel'
```

As we did not have outstanding changes (i.e. the master branch was clean before checking out the parallel branch) we were able to checkout parallel branch, otherwise we first need to commit the outstanding changes to the current branch we are on. Only then *git* will allow us to checkout a different branch as otherwise these changes will be lost.
Now, let us make a new file related to the parallel implementation and commit it to the parallel branch. After creating a new file, as shown below status tells us that there is a new file parallel.f90 which is not being tracked. We will add the parallel.f90 and commit so that *git* starts tracking this file under the branch parallel.

```
git status
# On branch parallel
```

```
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       parallel.f90
nothing added to commit but untracked files present (use "git add" to track)
```

```
>git add parallel.f90
>git commit .
```

Lets say we want to switch back and forth between the two branches and see the contents of the folder. So, we perform:

```
>ls
main.f90   Makefile   parallel.f90   subroutines.f90
```

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

```
> git branch -a
* master
  parallel
```

```
>ls
main.f90   Makefile   subroutines.f90
```

As can be seen from above the parallel.f90 disappeared from the present folder because right now we are on **master** branch which does not have (need) this file. If we checkout **parallel** branch then all the files that belong to it will be retrieved. This way we can maintain different versions of the entire project (folder) efficiently and switch between them easily. At some point, if a branch is deemed to be not needed or to be merged with the master branch that can be easily performed using *git* but not discussed here. It is always a good idea to keep the branch count as small as possible and delete unwanted branches. Also while on any branch we can go back to previous revisions which is also not discussed here.

5 Concluding Remarks

Here, few items which are not covered in this document but of help while working with *git* are discussed.

- Maintaining a central repository where all the developers can commit does not seem to be very easy with git, but it can be certainly achieved. Text books on *git* or on line tutorials are left as points of contact for this purpose.
- A graphical interface for checking the differences between versions and the previous commits can be launched using **gitk**.
- Manually removing the *.git* folder will get rid off all the previous versions, so it should be only performed with caution and if we do not want to use a version control system. Before doing such a step we should also checkout what version/branch we want to retain.
- Developers working on the same project can resort to a central repository or can pull/push changes from each other (which is where the power of distributed version control system like *git* comes into play). References should be consulted for more on this.

6 Collection of Commands

Finally, a collection of *git* commands (discussed in the present document) are provided here for quick reference.

```
git init
git status
git branch -a
git branch newbranch
git checkout branch-name
git add .
git commit .
gitk
```

References

[1] *http://git-scm.com/documentation*

[2] *http://progit.org/book/*

[3] Scott Chacon, *Pro Git*. Apress, SpringerLink