Configuration branches with Git

Burkart Lingner

November 3, 2011

Introduction

Every day that I use Git in my projects I wonder how I ever managed to do without it. Easy branching is arguably one of Git's best features – and I discovered a new way to utilize it.

It's common to use 'topic branches' to try out new features you're in the process of developing before you deem them worthy enough to be included in the `master` branch. That way it's easy to work on different parts of a project in parallel, with Git doing the work when it comes time to merge all those different branches back together.

My use case is somewhat different. I precisely *don't* want to have several code bases but rather the same code used with different settings. While the `master` branch with its set of default settings is used to develop the code itself, there are also a number of configuration branches that use the same code with their respective settings. This allows me to use the same software with an arbitrary number of different configurations to e.g. work on different data sets or have the same data processed in different ways by varying some parameters.

The key implementation problem, I figured, would be to maintain the once-changed settings in a certain configuration branch with as little human intervention as possible while also keeping up to date with the code changes from the `master` branch. However, thanks to Git's abilities it's really easy to achieve that goal. This article shows the general approach to do so and also covers a few of the problems that might occur along the way. Needless to say that even those cases are easily handled with just an additional command or two.

A simple example

Let's first set up a Git repository in the newly created directory *gitcfgbrn*.

```
$ mkdir gitcfgbrn
$ cd gitcfgbrn
$ git init
Initialized empty Git repository in /home/username/gitcfgbrn/.git/
```

The example project consists of the simple shell script *program.sh* and another file *settings.conf* that contains the configuration for the script. The script sources the *settings.conf* file to load the configuration, displays a welcome message, the names of all files in the current directory, and a goodbye message.

```
$ cat <<EOF > program.sh
> #!/bin/bash
> . ./settings.conf
> echo \$welcome
> ls \$files
> echo \$goodbye
> EOF
$ cat <<EOF > settings.conf
> welcome="Welcome␣user!"
> goodbye="That's␣it.␣Goodbye"
> files="*"
> EOF
$ chmod a+x program.sh
$ ./program.sh
Welcome user!
program.sh  settings.conf
That's it. Goodbye
```

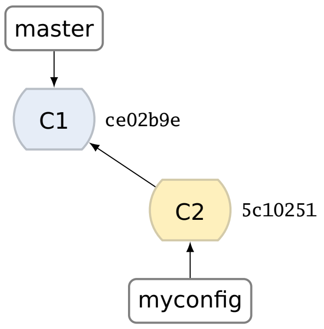
After the script has been executed to test its functionality, we'll commit both files to the repository.

```
$ git add .
$ git commit -am "Initial␣commit"
[master (root-commit) ce02b9e] Initial commit
 2 files changed, 8 insertions(+), 0 deletions(-)
 create mode 100755 program.sh
 create mode 100644 settings.conf
```

Now the configuration branch comes into play. Its configuration deviates from the standard one in the master branch by a personalized `welcome` message. To do so we first create the branch myconfig and then change the `welcome` parameter in *settings.conf*. After verifying the new functionality by running *program.sh*, we finally commit the change.

```
$ git checkout -b myconfig
Switched to a new branch 'myconfig'
$ sed -i 's/Welcome␣user/Welcome␣Burkart/' settings.conf
$ grep ^welcome= settings.conf
welcome="Welcome␣Burkart!"
$ ./program.sh
Welcome Burkart!
program.sh  settings.conf
That's it. Goodbye
$ git commit -am "Personalize␣welcome␣message"
[myconfig 5c10251] Personalize welcome message
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Thereafter the commit graph looks as follows:



So far so good. Now let's go back to the master branch to verify that it still contains the old `welcome` message.

```
$ git checkout master
Switched to branch 'master'
$ grep ^welcome= settings.conf
welcome="Welcome␣user!"
```

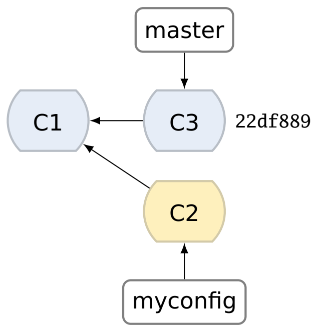
And just as expected it does.

Keep up with changes in the master branch

The next step will be to change something in the master branch and make sure that change is also applied to the configuration branch myconfig. We could change something in *program.sh* but it's also perfectly reasonable to make changes in *settings.conf* while on the master branch – those changes then reflect a different default behavior of the program. What we'll do is make sure that *program.sh* only prints the names of shell scripts from now on. Once the change is tested it can be committed.

```
$ sed -i 's|\*|\*.sh|' settings.conf
$ grep ^files= settings.conf
files="*.sh"
$ ./program.sh
Welcome user!
program.sh
That's it. Goodbye
$ git commit -am "List␣only␣shell␣scripts"
[master 22df889] List only shell scripts
 1 files changed, 1 insertions(+), 1 deletions(-)
```

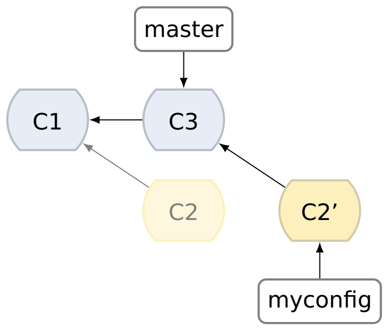
The new commit C3 is a child of the initial commit C1. The commit graph shows that the branches master and myconfig have diverted.



An important aspect of the methodology presented here is to easily encompass changes made in the master branch into the configuration branch. Now that the master branch has advanced from where myconfig branched off, such an operation is required. Git provides us with the *rebase* command to do just what we want to do, i.e. apply the configuration changes in commit C2 to C3 instead of to C1 as before. As you can see, Git is smart enough to mix the changes to *settings.conf* from both C2 and C3: the personalized `welcome` message and the shell-script `files` wildcard, respectively.

```
$ git checkout myconfig
Switched to branch 'myconfig'
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Personalize welcome message
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging settings.conf
$ cat settings.conf
welcome="Welcome␣Burkart!"
goodbye="That's␣it.␣Goodbye"
files="*.sh"
```

As you can see in the commit graph, the old commit C2 has been removed and replaced with C2′ which contains the same change to the `welcome` message but has C3 as its parent.



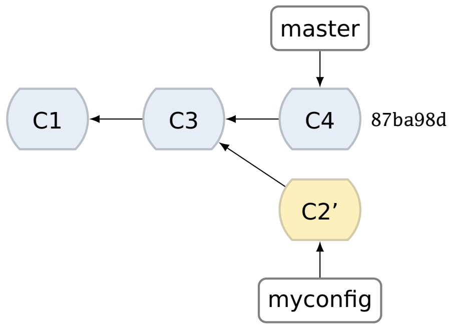
Merge conflict upon rebasing

So far everything has been pretty straightforward. Things get more complicated if the master branch modifies those parts of the configuration that are changed in myconfig as well.

To create such a situation let's go back to the master branch and make the `welcome` message more informative.

```
$ git checkout master
Switched to branch 'master'
$ sed -i 's/Welcome␣user!/Welcome␣user!␣These␣are␣the␣files␣you'\'\
> 're␣looking␣for:/' settings.conf
$ grep ^welcome= settings.conf
welcome="Welcome␣user!␣These␣are␣the␣files␣you're␣looking␣for:"
$ git commit -am "Clarify␣welcome␣message"
[master 87ba98d] Clarify welcome message
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Once more this makes the master branch advance from a common ancestor with the myconfig branch, as shown in the commit graph.



We do the usual to encompass the changes from master into myconfig: checkout myconfig and rebase. But wait... Rebasing fails due to a merge conflict this time. As both branches edited the same line in *settings.conf* (the `welcome` string), Git can't automatically determine what to do. Just like with any other merge conflict, Git puts special markers in the file and asks the user to manually resolve the conflict.

```
$ git checkout myconfig
Switched to branch 'myconfig'
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Personalize welcome message
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging settings.conf
CONFLICT (content): Merge conflict in settings.conf
Failed to merge in the changes.
Patch failed at 0001 Personalize welcome message

When you have resolved this problem run "git␣rebase␣--continue".
If you would prefer to skip this patch, instead run "git␣rebase␣--skip".
To restore the original branch and stop rebasing run "git␣rebase␣--abort".

$ cat settings.conf
<<<<<<< HEAD
welcome="Welcome␣user!␣These␣are␣the␣files␣you're␣looking␣for:"
=======
welcome="Welcome␣Burkart!"
>>>>>>> Personalize welcome message
goodbye="That's␣it.␣Goodbye"
files="*.sh"
```

The conflict resolution is most easily done using **git** mergetool.

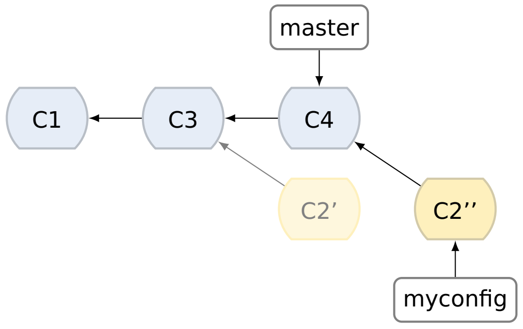
```
$ git mergetool
merge tool candidates: meld opendiff kdiff3 tkdiff xxdiff tortoisemerge ↙
    ↳ gvimdiff diffuse ecmerge p4merge araxis emerge vimdiff
Merging the files: settings.conf

Normal merge conflict for 'settings.conf':
  {local}: modified
  {remote}: modified
Hit return to start merge resolution tool (meld):
$ grep ^welcome= settings.conf
welcome="Welcome␣Burkart!␣These␣are␣the␣files␣you're␣looking␣for:"
```

In my case *meld* was used as the merge tool. It created a backup copy of *settings.conf* named *settings.conf.orig* which can be deleted.

```
$ rm settings.conf.orig
$ git rebase --continue
Applying: Personalize welcome message
```

Once Git continues the rebase operation, the result is analogous to the rebase cycle without a merge conflict, yielding the following commit graph:



Migrating a commit from config branch to master

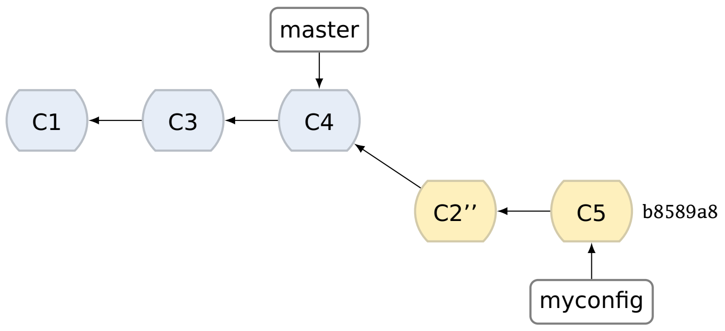
Let's now make another edit on the file *program.sh*.

```
$ sed -i 's/^\./source/' program.sh
$ cat program.sh
#!/bin/bash
source ./settings.conf
echo $welcome
ls $files
echo $goodbye
$ git commit -am "Use␣source␣command␣instead␣of␣(dot)␣for␣clarity"
[myconfig b8589a8] Use source command instead of (dot) for clarity
 1 files changed, 1 insertions(+), 1 deletions(-)
```

But oops, we're still on the myconfig branch, aren't we? Let's make sure.

```
$ git branch
  master
* myconfig
```

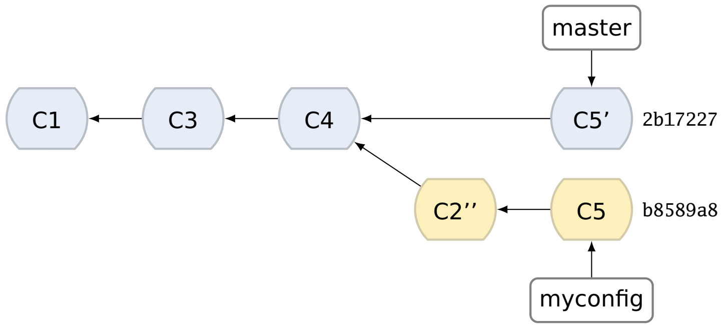
As the asterisk in front of myconfig indicates we are in fact still on that branch. That means the last commit went to the myconfig branch, making the commit tree look like this:



Unfortunately this is in violation with the methodology of having code modifications in the master branch whereas the myconfig branch is reserved for a modified configuration. Nevertheless the most recent commit C5 is valuable, it's just in the wrong place. Thankfully Git offers a way to copy (not move) commits to another branch in the form of the command **git** cherry-pick.

```
$ git checkout master
Switched to branch 'master'
$ git cherry-pick b8589a8
Finished one cherry-pick.
[master 2b17227] Use source command instead of (dot) for clarity
 1 files changed, 1 insertions(+), 1 deletions(-)
```

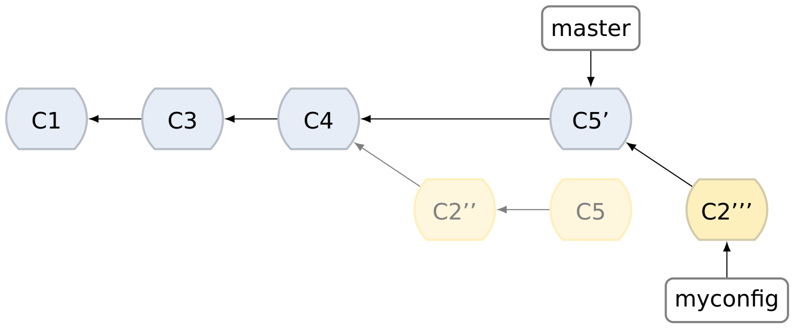
After this operation the master branch contains the additional commit C5′ with the same changes to *program.sh* as C5 but with a different SHA1 hash.



To remove this duplicate commit (actually the original C5 in the myconfig branch) we rebase myconfig onto master once more.

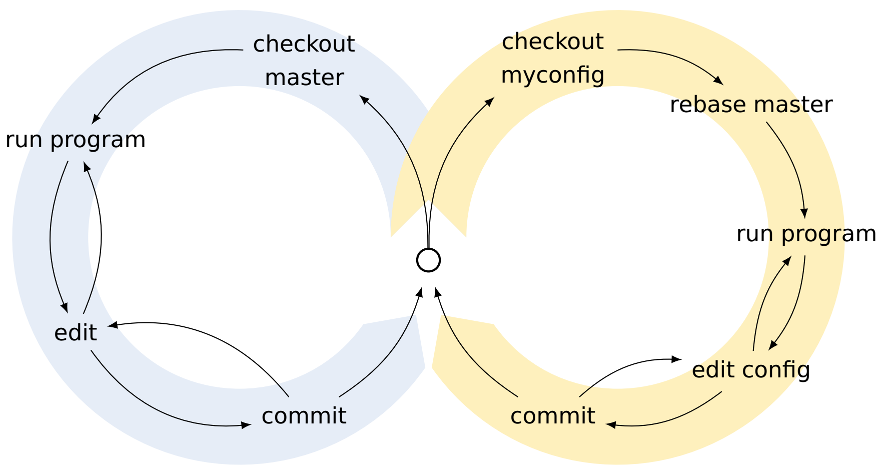
```
$ git checkout myconfig
Switched to branch 'myconfig'
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Personalize welcome message
```

Now everything is as it is supposed to be again, with the code modification history in the master branch and configuration changes in the myconfig branch.



Summary

The workflow presented in this article harnesses the branching power of Git to easily switch between differently configured variants of the same code. The key to this approach is to strictly differentiate between general (programming) work in i.e. the `master` branch and configuration-only modifications in the configuration branch. This allows you to easily use the most up-to-date code base in the configuration branch after just one simple rebasing operation.



I like to think of working on the two branches as two circles connected at one point, just like in the figure above. You start out in the center by deciding what you intend to do. Either you take the blue route to run the program with default settings and work on the code itself, or you take the yellow route to run the program with non-standard settings, possibly editing that configuration as you go.

Each of the two routes begins by checking out the appropriate branch and ends by committing whatever changes you may have made before going back to the junction from the beginning where you decide which configuration you'd like to work on next. What differentiates the two routes is whether you're going to edit the code (blue) or just the configuration (yellow). In the latter case it's recommended to first rebase the configuration branch to keep up with any code changes that happened since the last rebase operation.