Understanding the Git Workflow

笔记本： Git
创建时间： 2012/10/6 22:19
URL： http://sandofsky.com/blog/git-workflow.html

Understanding the Git Workflow

If you don't understand the motivation behind Git's design, you're in for a world of hurt. With enough flags you can force Git to act the way you think it should instead of the way it wants to. But that's like using a screwdriver like a hammer; it gets the job done, but it's done poorly, takes longer, and damages the screwdriver.

Consider how a common Git workflow falls apart.

> Create a branch off Master, do work, and merge it back into Master when you're done

Most of the time this behaves as you expect because Master changed since you branched. Then one day you merge a feature branch into Master, but Master hasn't diverged. Instead of creating a merge commit, Git points Master to the latest commit on the feature branch, or "fast forwards." (Diagram)

Unfortunately, your feature branch contained checkpoint commits, frequent commits that back up your work but captures the code in an unstable state. Now these commits are indistinguishable from Master's stable commits. You could easily roll back into a disaster.

So you add a new rule: "When you merge in your feature branch, use *--no-ff* to force a new commit." This gets the job done, and you move on.

Then one day you discover a critical bug in production, and you need to track down when it was introduced. You run bisect but keep landing on checkpoint commits. You give up and investigate by hand.

You narrow the bug to a single file. You run *blame* to see how it changed in the last 48 hours. You know it's impossible, but *blame* reports the file hasn't been touched in weeks. It turns out *blame* reports changes for the time of the initial commit, not when merged. Your first checkpoint commit modified this file weeks ago, but the change was merged in today.

The *--no-ff* band-aid, broken *bisect*, and *blame* mysteries are all symptoms that you're using a screwdriver as a hammer.

Rethinking Revision Control

Revision control exists for two reasons.

The first is to help the act of writing code. You need to sync changes with teammates, and regularly back up your work. Emailing zip files doesn't scale.

The second reason is configuration management . This includes managing parallel lines of development, such as working on the next major version while applying the occasional bug fix to the existing version in production. Configuration management is also used to figure out when exactly something changed, an invaluable tool in diagnosing bugs.

Traditionally, these two reasons conflict.

When prototyping a feature, you should make regular checkpoint commits. However, these commits usually break the build.

In a perfect world, every change in your revision history is succinct and stable. There are no checkpoint commits that create line noise. There are no giant, 10,000 line commits. A clean history makes it easy to revert changes or ~~cherry-pick~~ them between branches. A clean history is easy to later inspect and analyze. However, maintaining a clean history would mean waiting to check in changes until they're perfect.

So which approach do you choose? Regular commits, or a clean history?

If you're hacking on a two man pre-launch startup, clean history buys you little. You can get away with committing everything to Master, and deploying whenever you feel like it.

As the consequences of change increase, be it a growing development team or the size of your user base, you need tools and techniques to keep things in check. This includes automated tests, code review, and a clean history.

Feature branches seem like a happy middle ground. They solve the basic problems of parallel development. You're thinking of integration at the least important time, when you're writing the code, but it will get you by for some time.

When your project scales large enough, the simple branch/commit/merge workflow falls apart. The time for duct-tape is over. You need a clean revision history.

Git is revolutionary because it gives you the best of both worlds. You can regularly check in changes while prototyping a solution but deliver a clean history when you're finished. When this is your goal, Git's defaults make a lot more sense.

The Workflow

Think of branches in two categories: public and private.

Public branches are the authoritative history of the project. In a public branch, every commit should be succinct, atomic, and have a well documented commit message. It should be as linear as possible. It should be immutable. Public branches include Master and release branches.

A private branch is for yourself. It's your scratch paper while working out a problem.

It's safest to keep private branches local. If you do need to push one, maybe to synchronize your work and home computers, tell your teammates that the branch you pushed is private so they don't base work off of it.

You should never merge a private branch directly into a public branch with a vanilla *merge*. First, clean up your branch with tools like reset, rebase, squash merges, and commit amending.

Treat yourself as a writer and approach each commit as a chapter in a book. Writers don't publish first drafts. Michael Crichton said, "Great books aren't written– they're rewritten."

If you come from other systems, modifying history feels taboo. You're conditioned that anything committed is written in stone. By that logic we should disable "undo" in our text editors.

Pragmatists care about changes until the changes become noise. For configuration management, we care about big-picture changes. Checkpoint commits are just a cloud-backed undo buffer.

If you treat your public history as pristine, fast-forward merges are not only safe but preferable. They keep revision history linear and easier to follow.

The only remaining argument for *--no-ff* is "documentation." People may use merge commits to represent the last deployed version of production code. That's an antipattern. Use tags.

Guidelines and Examples

I use three basic approaches depending on the size of my change, how long I've been working on it, and how far the branch has diverged.

Short lived work

The vast majority of the time, my cleanup is just a squash merge.

Imagine I create a feature branch and create a series of checkpoint commits over the next hour:

```
git checkout -b private_feature_branch
touch file1.txt
git add file1.txt
git commit -am "WIP"
```

When I'm done, instead of a vanilla *git merge*, I'll run:

```
git checkout master
git merge --squash private_feature_branch
git commit -v
```

Then I spend a minute writing a detailed commit message.

Larger work

Sometimes a feature sprawls into a multi-day project, with dozens of small commits.

I decide my change should be broken into smaller changes, so squash is too blunt an instrument. (As a rule of thumb I ask, "Would this be easy to code review?")

If my checkpoint commits followed a logical progression, I can use ~~rebase~~'s Interactive Mode.

Interactive mode is powerful. You can use it to edit an old commits, split them up, reorder them, and in this case squash a few.

On my feature branch:

```
git rebase --interactive master
```

It then opens an editor with a list of commits. On each line is the operation to perform, the SHA1 of the commmit, and the current commit message. A legend is provided with a list of possible commands.

By default, each commit uses "pick," which doesn't modify the commit.

```
pick ccd6e62 Work on back button
pick 1c83feb Bug fixes
pick f9d0c33 Start work on toolbar
```

I change the operation to “squash,” which squashes the second commit into the first.

```
pick ccd6e62 Work on back button
squash 1c83feb Bug fixes
pick f9d0c33 Start work on toolbar
```

When I save and close, a new editor prompts me for the commit message of the combined commit, and then I’m done.

Declaring Branch Bankruptcy

Maybe my feature branch existed for a very long time, and I had to merge several branches into my feature branch to keep it up to date while I work. History is convoluted. It’s easiest to grab the raw diff create a clean branch.

```
git checkout master
git checkout -b cleaned_up_branch
git merge --squash private_feature_branch
git reset
```

I now have a working directory full of my changes and none of the baggage from the previous branch. Now I manually add and commit my changes.

Summary

If you’re fighting Git’s defaults, ask why.

Treat public history as immutable, atomic, and easy to follow. Treat private history as disposable and malleable.

The intended workflow is:

1. Create a private branch off a public branch.
2. Regularly commit your work to this private branch.
3. Once your code is perfect, clean up its history.
4. Merge the cleaned-up branch back into the public branch.

Special thanks to @joshaber and @jbarnette for providing feedback on an early draft.

— @sandofsky