OneFlow - a Git branching model and workflow

posted on 2017/04/30

In this post I describe a Git branching model that I've been using successfully for years for both my side projects and professionally. I'm dubbing it OneFlow.

This workflow was first described in the 'GitFlow considered harmful' article on this blog in May of 2015. A more detailed description of the branching model was a very common request in the comments for that post - in fact, the name 'OneFlow' was suggested by the readers (I chose it from many candidates). This article is meant to be a comprehensive and detailed reference, but the core branching model is exactly the same as described in that original post.

Table of Contents

Introduction

OneFlow has been conceived as a simpler alternative to GitFlow. However, please don't read "simpler" as "less able". OneFlow's branching model is exactly as powerful as GitFlow's. There is not a single thing that can be done using GitFlow that can't be achieved (in a simpler way) with OneFlow. The description below goes into more detail.

As the name suggests, OneFlow's basic premise is to have one eternal branch in your repository. This brings a number of advantages (see below) without losing any expressivity of the branching model - the more advanced use cases are made possible through the usage of Git tags.

While the workflow advocates having one long-lived branch, that doesn't mean there aren't other branches involved when using it. On the contrary, the branching model encourages using a variety of support branches (see below for the details). What is important, though, is that they are meant to be short-lived, and their main purpose is to facilitate code sharing and act as a backup. The history is always based on the one infinite lifetime branch.

OneFlow advantages

Maintaining a single long-lived branch simplifies the versioning scheme and day-to-day operations that developers have to perform considerably.

It also makes the project history cleaner and more readable, and thus more useful.

When to use OneFlow

OneFlow is meant to be a drop-in replacement for GitFlow, which means it's suitable in all situations that GitFlow is. In fact, you can quite easily migrate a project that is using GitFlow to OneFlow.

The main condition that needs to be satisfied in order to use OneFlow for a project is that every new production release is based on the previous release (GitFlow has exactly the same requirement). The vast majority of software projects fulfill that condition. If your project is a web application, for instance, then OneFlow should be a great fit. Most open-source projects could be versioned using OneFlow as well.

When NOT to use OneFlow

While OneFlow is pretty flexible, it's not suitable for every project. OneFlow will be a bad fit in basically the same circumstances that GitFlow would be. There are 2 main reasons why that might be the case.

First, when the above condition ("every new production release is based on the previous one") is not satisfied. As an example, take the Python programming language. It has two incompatible versions, 2 and 3. Both of them receive bugfixes and security patches - however, that doesn't mean a new release of Python 3 is based on the commit of the latest release of Python 2. The two versions have diverged, and while they surely share a lot of code, you can't say that one is based on the other (talking from a purely version control perspective).

If your project needs to maintain multiple simultaneous yet incompatible release versions that way, then OneFlow won't work for you out of the box. You can surely use elements of it - probably each individual version can be managed using OneFlow, for example. However, in these types of projects, the main challenge is usually in the interactions between the versions and how to effectively share code between them, and OneFlow was not designed to be a solution to that problem.

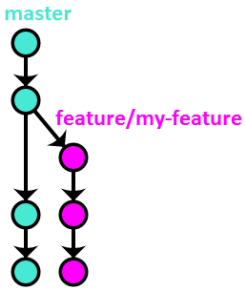
Second, if your project has a high degree of automation - uses Continuous Delivery, or even Continuous Deployment, for example - then this workflow will most likely be too heavy for you. Perhaps parts of it might still be useful, but other elements (like the release process, for instance) would have to be heavily modified to make sense when releasing on such a very frequent cadence.

Detailed description

The main branch

Like was explained before, the workflow uses only one eternal branch. The name doesn't matter, and can be anything you want. We will use `master` in this description, as it's probably the most common name, and is already a Git convention, but you can also use, for example, `current`, `default`, `mainline`, or anything else.

Feature branches



Feature branches (also sometimes called topic branches) are where the day-to-day development work happens - hence, they are by far the most common of all the support branches. They are used to develop new features and bugfixes for the upcoming release. They are usually named similarly to `feature/my-feature`.

Feature branches often exist only in the developer's repository, and are never pushed - however, if there are multiple people working on one feature, or if the feature will take a long time to develop, it's typical to push them to the central repository (if only to make sure the code isn't lost with a single disk failure).

Starting a feature branch

To start a feature branch, simply create a new branch from `master`:

```
$ git checkout -b feature/my-feature master
```

Finishing a feature branch

Once work on the given feature is done, it needs to be integrated back into `master`. There are several ways this can be accomplished.

Note: the choice of the feature branch integration method is immaterial as far as the workflow is concerned. It should be based on personal or team preference, however the branching model will work exactly the same, regardless of which option is chosen. My personal recommendation is to use option #1.

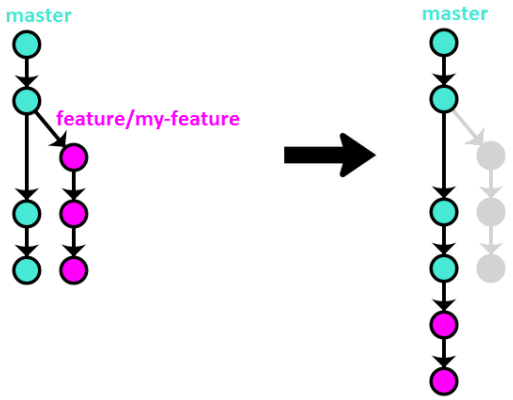
Option #1 - rebase

This method uses Git's `rebase` command (with the `-i`, meaning interactive, switch) to integrate the feature branch with `master`:

```
$ git checkout feature/my-feature
$ git rebase -i master
$ git checkout master
$ git merge --ff-only feature/my-feature
$ git push origin master
$ git branch -d feature/my-feature
```

If you're not yet well acquainted with the `rebase` command, I recommend this chapter from the Pro Git SCM book.

Here's a visual illustration of how that method works:



Advantages of this method:

- Rebasing before integrating with master allows you to clean up the branch history before making it public, resulting in better final history landing on `master`.
- Linear history makes things simpler and easier to find, especially when looking at the per-file history.

Disadvantages:

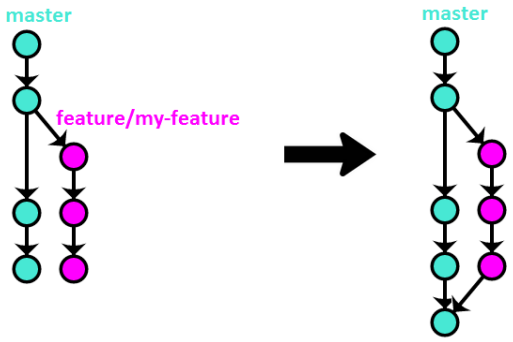
- Reverting the entire feature requires reverting multiple commits.

Option #2 - merge --no-ff

This is the method that GitFlow advocates.

```
$ git checkout master
$ git merge --no-ff feature/my-feature
$ git push origin master
$ git branch -d feature/my-feature
```

Visually:



Advantages of this method:

- Reverting the entire feature requires reverting only one commit (the merge commit).

Disadvantages:

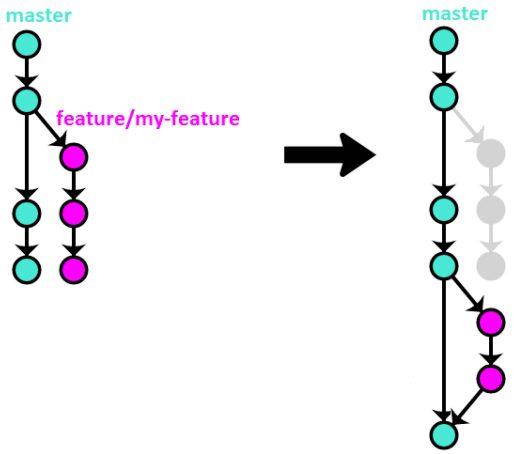
- The feature branch history, which is often messy, gets put directly on `master`.
- The proliferation of merge commits (especially as the number of developers on a project grows) makes the history unmanageable.

Option #3 - rebase + merge --no--ff

This method is a combination of the previous two, trying to keep their advantages while simultaneously getting rid of the disadvantages:

```
$ git checkout feature/my-feature
$ git rebase -i master
$ git checkout master
$ git merge --no-ff feature/my-feature
$ git push origin master
$ git branch -d feature/my-feature
```

Visually:



Advantages of this method:

- Clean and almost linear history.
- Easy to revert an entire feature with one commit.

Disadvantages:

- It's difficult to enforce this method programmatically, so it has to rely on best-effort convention.

Finally, regardless of the method used, if the feature branch was pushed to the central repository, you need to now remove it:

```
$ git push origin :feature/my-feature
```

Release branches

Release branches are created to prepare the software for being released. Obviously, what exactly that means varies on a project-per-project basis. This could be as simple as bumping the version number in the configuration, or involve things like code freezes, producing Release Candidates, and having a full QA process. The important thing is all that happens on a separate branch, so that day-to-day development can continue as usual on `master`.

The naming convention for these is `release/<version-number>`.

Starting a release branch

Release branches also start from `master`, however they often don't start from the tip - instead, they have their origin in whatever commit on `master` you think contains all of the features that you want to include in the given release.

For example, here we start the branch for the version `2.3.0` release on a commit with the hash `9efc5d`:

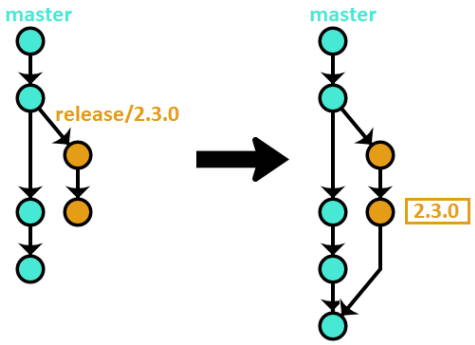
```
$ git checkout -b release/2.3.0 9efc5d
```

Finishing a release branch

Once whatever process you use for releasing is finished, the tip of the branch is tagged with the version number. After that, the branch needs to be merged into `master` to be versioned permanently:

```
$ git checkout release/2.3.0
$ git tag 2.3.0
$ git checkout master
$ git merge release/2.3.0
$ git push --tags origin master
$ git branch -d release/2.3.0
```

Here's a diagram illustrating the above commands (assuming the release took two commits):



Again, if you pushed the release branch to the central repository, you now need to delete it:

```
$ git push origin :release/2.3.0
```

Hotfix branches

Hotfix branches are very similar to release branches - they result in a new version of the project being released. Where they differ is their intentions - while release branches signify a planned production milestone, hotfix branches are most often an unwanted but necessary exception to the usual release cadence, typically because of some critical defect found in the latest release that needs to be fixed as soon as possible.

They are named `hotfix/<version-number>`. Note that if you use Semantic Versioning, regular releases bump either the Major or Minor number, while hotfixes bump the Patch number.

Starting a hotfix branch

Hotfix branches are cut from the commit that the latest version tag points to. Continuing our example from the release branch:

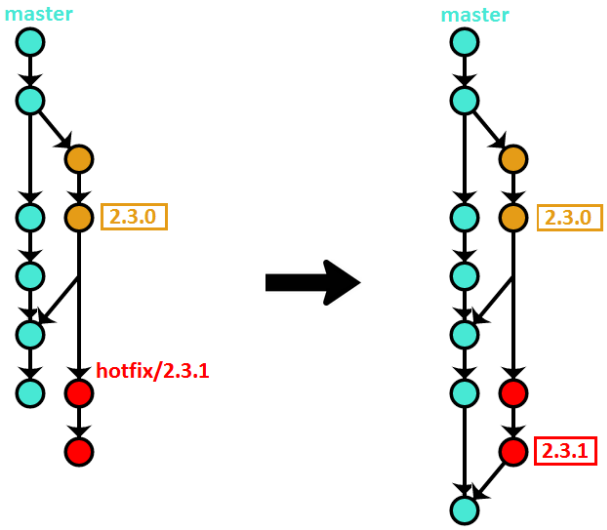
```
$ git checkout -b hotfix/2.3.1 2.3.0
```

Finishing a hotfix branch

Finishing a hotfix branch is pretty much the same as finishing a release branch: tag the tip, merge it to master, then delete the branch.

```
$ git checkout hotfix/2.3.1
$ git tag 2.3.1
$ git checkout master
$ git merge hotfix/2.3.1
$ git push --tags origin master
$ git branch -d hotfix/2.3.1
```

Here's a visual illustration:



There is one special case when finishing a hotfix branch. If a release branch has already been cut in preparation for the next release before the hotfix was finished, you need to merge the hotfix branch not to `master`, but to the release branch. Otherwise, the new release will bring back the original bug that the hotfix corrected. The fix will eventually get to `master` - when the release branch is merged back to it.

As always, if the hotfix branch was pushed to the central repository, you need to remove it now:

```
$ git push origin :hotfix/2.3.1
```

Variation - develop + master

There is one small wrinkle with the branching model described above. In order to find the latest production version of the code, you need to look at all of the tags in the repository, and checkout the latest one.

This problem has a very simple solution. You add another, 'latest', long-lived branch, whose only purpose is to point to the last released commit. Each time the version number is bumped, the 'latest' branch is fast-forwarded to the newly created tag.

So, that's all great, but there is one small issue left. It would be very cool, especially for open-source projects, if the default branch that people got when cloning the repository was this 'latest' branch, which contains stable code, instead of the 'working' (what was called `master` in the above description) branch, which contains the work-in-progress on the yet-unreleased next version, which might not be very stable.

The simplest solution to this problem is to take advantage of the fact that `master` is the default branch in Git. So, we call the 'latest' branch `master`. However, that means we need to find a new name for the 'working' branch. In this description, we will re-use the GitFlow convention, and call it `develop` (of course, you are free to call it whatever you want in your project).

For clarity, I will show all of the workflow operations again, this time using this new naming. However, I want to emphasize that, other than a slight name change and the introduction of a new 'marker' branch, the workflow is exactly the same as was described above.

The main branches

This variation uses two branches: `develop`, which plays the same role as `master` above, and `master`, which points at the latest release tag.

Feature branches

Feature branches work exactly the same as already explained, except you need to substitute `master` with `develop` in the description above.

Starting a feature branch

```
$ git checkout -b feature/my-feature develop
```

Finishing a feature branch

Option #1 -rebase

```
$ git checkout feature/my-feature
$ git rebase -i develop
$ git checkout develop
$ git merge --ff-only feature/my-feature
$ git push origin develop
$ git branch -d feature/my-feature
```

Option #2 - merge --no-ff

```
$ git checkout develop
$ git merge --no-ff feature/my-feature
```

```
$ git push origin develop
$ git branch -d feature/my-feature
```

Option #3 - rebase + merge --no--ff

```
$ git checkout feature/my-feature
$ git rebase -i develop
$ git checkout develop
$ git merge --no-ff feature/my-feature
$ git push origin develop
$ git branch -d feature/my-feature
```

Release branches

Release branches work the same as described above (with `master` substituted for `develop`, of course), except one small detail. There is an extra step when finishing the release branch: fast-forwarding the marker branch to the newly created release tag.

Starting a release branch

```
$ git checkout -b release/2.3.0 9efc5d
```

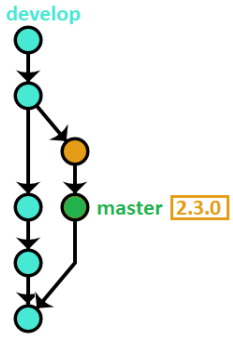
Finishing a release branch

```
$ git checkout release/2.3.0
$ git tag 2.3.0
$ git checkout develop
$ git merge release/2.3.0
$ git push --tags origin develop
$ git branch -d release/2.3.0
```

And here is the extra step - fast-forwarding `master` to the latest release tag:

```
$ git checkout master
$ git merge --ff-only 2.3.0
```

Here's a visualization of the state of the repository after finishing the release branch:



Hotfix branches

Hotfix branches, because they result in publishing a new version as well, also require the additional step of fast-forwarding `master` to the newly created tag.

Starting a hotfix branch

Because `master` always tracks the latest tag, creating a hotfix branch is a tiny bit easier in this variant (note however that you still need to look at the tags to determine what the previous version number was, in order to name your hotfix branch correctly):

```
$ git checkout -b hotfix/2.3.1 master
```

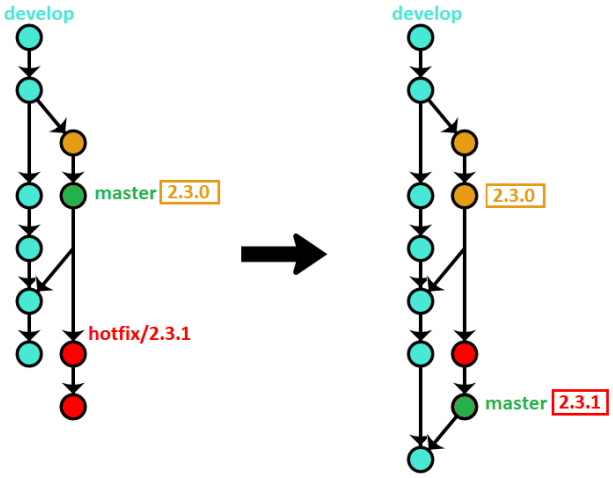
Finishing a hotfix branch

```
$ git checkout hotfix/2.3.1
$ git tag 2.3.1
$ git checkout develop
$ git merge hotfix/2.3.1
$ git push --tags origin develop
$ git branch -d hotfix/2.3.1
```

And here is the additional step of fast-forwarding `master` to the latest release tag:

```
$ git checkout master
$ git merge --ff-only 2.3.1
```

If we continue our release example, the visual illustration looks something like this:



Summary

So, this is OneFlow in a nutshell. If you have any questions about the workflow, or if something is unclear in the description, please let me know in the comments below. I'll try to answer as best as I can.

I also wanted to ask you, dear reader, one thing. GitFlow has a set of command-line tools that help with managing the workflow. Personally, I'm not a huge fan of that, as I think relying on the tools makes people never learn the actual concepts behind the workflow they're using, and when things go wrong (which they seem to invariably do with these tools - for example, if you happen to execute the commands in the wrong order), they have no idea how to fix it, and wind up with their repository in a really weird state. However, I also recognize that these tools help with the adoption of the workflow, and are useful when, for example, trying to enforce team-wide standards.

So, here's my question to you: do you want to see a similar command-line tool for working with OneFlow? If the answer is 'yes', please let me know in the comments. If enough people express interest in using a project like that, then I'll devote some time to creating that tool.

Credits: the diagrams used in this article were created using the awesome GitGraph.js library.

this strategy.

Reply • Share ›

Adam Ruka Mod → ryanmulligan • 4 months ago

Thanks. I'm putting you down for a +1 for the OneFlow CLI tool then.

1 Reply • Share ›

Mauro Farah → Adam Ruka • a year ago

Is the first who win? I hope more people say yes! :)

Reply • Share ›

mwrren → Mauro Farah • a year ago

I would also really like one please! Great article!

Reply • Share ›

Adam Ruka Mod → mwrren • a year ago

Noted. Thanks for the comment :)

Reply • Share ›

Agustín Cruz Lozano → Adam Ruka • 8 months ago

Also would like to see a CLI tool

Reply • Share ›

Adam Ruka Mod → Agustín Cruz Lozano • 7 months ago

Noted, thanks!

1 Reply • Share ›

Joe K • a year ago

Excellent. Thank you. I have always found GitFlow way too complicated, and super hard for those new to Git to learn, so I've been recommending something very very similar to this to my team for years, and I love that I finally have a post that I can point to and say, "like this!"

8 Reply • Share ›

Adam Ruka Mod → Joe K • a year ago

Thanks for the comment :)

Reply • Share ›

Phil Ruggera • a month ago

Is there a way to get the "develop + master" variation to work for immutable builds? This means the commit SHA that was used to build the deployable object is the commit SHA that ends up in the master branch.

2 Reply • Share ›

Adam Ruka Mod → Phil Ruggera • a month ago

Hey Phil,

yes, you get that behavior out of the box. Because updating master is always just a fast-forward merge in that scheme, the commit that will eventually end up on master is exactly the same commit that the deployable artifacts were built from.

Reply • Share ›

Sean Brown • a year ago

Hello, I'm in the market for a new workflow. I like the simplicity of this one! Our current workflow is a custom one that's loosely based on git flow. The main motivation for customizing it was to have the ability to easily change the scope of a release. We achieve this by keeping feature branches independent of each other. They get merged into a release branch, but we don't merge in the other direction. With each feature branch being independent, I can easily toss the current release branch, create a new one, and only merge in the subset of features that made the cut.

For example, say feature branches feature1, feature2, & feature3 are all in development, and they've all been merged into release1 and are undergoing testing. Features 1 & 2 are close, but 3 is not nearly ready. Management decides that features 1

release1 and are undergoing testing. Features 1 & 2 are close, but 3 is not nearly ready. Management decides that features 1 & 2 need to go live ASAP and that 3 can wait. So now we can change direction by creating release1.b and merging in feature1 & feature2 (the suffix, ".b" is not important and doesn't correspond to a release number).

This is great, and we have taken advantage of this feature of our workflow to quickly change course. However, it is a burden to have features being developed independently. Developers are not working in conjunction with other work being done; every feature is developed from the currently live code. It also makes for a relatively high number of merge conflicts when features get merged into the release branch.

Does your workflow support changing the scope of a release like that? Or are you familiar with any workflows that do?
2 ▲ | ▼ • Reply • Share ›

Adam Ruka Mod ➚ Sean Brown • 10 months ago

Hi Sean,

unfortunately, neither my workflow, nor any other that I know of, allows you to do that. The problem is that what you've described is antithetical to Continuous Integration, a pretty fundamental practice in modern software development. The burdens that you're experiencing are a direct consequence of not following that process.

May I suggest you look into Feature Flags? They have many uses (A/B testing, for example), but one of them is I think exactly what you are looking for - dynamic releases. In that model, new features are pushed continuously to master (and then to production, if you follow Continuous Deployment), however, they are not activated. Only after the code has been deployed and verified working correctly, is the feature flag turned on, at which point the users can interact with the new feature. This way decouples the source-code changes from deployment, and from the feature set active in the product. The activation can even be done by somebody non-technical, at which point releases become a business, instead of technical, decision.

In your situation, management would be able to activate the features they want independently of their version-control status, with a high confidence in it working (they can test it in production by overridding the treatment to force the new code paths, not yet available for the general public).

Does this make sense?

Thanks,
Adam
1 ▲ | ▼ • Reply • Share ›

Amaury Fages ➚ Adam Ruka • 7 months ago

The day when every feature will be "flippable" is not yet to come.
You all live in an ideal world or never ever saw a complex project with hard clients.
▲ | ▼ • Reply • Share ›

Adam Ruka Mod ➚ Amaury Fages • 7 months ago

I would say feature flags are more useful the more complex the project is... And the harder the client is (rolling back is just a matter of turning off the feature flag back off).
▲ | ▼ • Reply • Share ›

Amaury Fages ➚ Adam Ruka • 7 months ago

Yes, that's true. But also not easily feasible the more it's complex. Especially when you consider the reality on the ground, or behind the scenes. Unless you are working for a startup with his own "software path" and no "real client", you'll always driven by budget and planning. No matter what you try or you want to convince yourself with things like "agile / complexity", you'll always be driven by what V cycles were made for. In a 8 years of projects with 5 different clients, I never saw a single one where team is truly free. Reviews, technical important things who don't bring business value is put aside in profit of planning. So I'm saying those ideal model work well for your own project without any SLA OR with a 100% test coverage automation tools (good luck to find budget, this is heavy cost to engage and to negotiate with ROI. Plus, there are some cases where you still need manual testing, 100% is not achievable. Where do you put those in the so called "Github model" ?

In fact, I'm just saying there are many devil advocates on the internet who destroy git flow whereas it's still the safest model for control over the contract, SLA, protection ...
▲ | ▼ • Reply • Share ›

Adam Ruka Mod ➚ Amaury Fages • 7 months ago

I don't think you understood the article, so let me quote it directly:

> There is not a single thing that can be done using GitFlow that can't be achieved (in a simpler way) with OneFlow

Reply • Share ›

Raman Gupta ➚ Sean Brown • 6 months ago

Yes, the workflow used by the git.git project itself supports exactly this. Having used it myself on multiple real enterprise projects, I know it works and works well. I also wrote about it here: https://hackernoon.com/how-.... This flow is not at all antithetical to the concept of Continuous Integration, and actually supports true CI even better than OneFlow and GitFlow in my opinion.

Reply • Share ›

Adam Ruka Mod ➚ Raman Gupta • 5 months ago

Hey Raman,

thanks a lot for the comment, and for the interesting article. However, reading it, gitworkflow seems incredibly complex. From my experience, people already have trouble using GitFlow, which is many, many times simpler than that. I can't imagine gitworkflow will catch on in the same capacity as GitFlow has.

Thanks,
Adam

2 Reply • Share ›

Raman Gupta ➚ Adam Ruka • 5 months ago

I wouldn't say *incredibly* complex. It probably is higher complexity than GitFlow, with certainly an increase in capability and flexibility. But yes, I agree it fits a niche: teams that know, or are willing to learn, git really well. BTW, since I last posted, I've created a Github repo to serve as a home for docs and tools related to gitworkflow: https://github.com/rocketra....

Reply • Share ›

Glenn Austin • a year ago

Having used large projects (both single- and multi-contributor), I greatly prefer gitflow-organized projects that merge without squashing. Why?
- *All* of the code history is there. More than a few times we've had to go back to a bug report that asks for the reverse behavior that was requested *by the same person* during feature development.
- You have what is shipping (master) and what has passed requirements for future releases (develop), the *minor* changes necessary for the next release (release/version_number), any explicit hot fixes (hotfix), and features still under development. If you're having to go through a release process, having a separate release branch means that the release process itself doesn't block other development.
- Git encourages small, atomic commits. Squashing violates that basic tenet (and makes it no different than a centralized, managed version control system). Rebasing basically does the same, and adds the hell of updating *every* commit on the branch being rebased on top of the new destination.
- Merge/squash means that you can *never* be certain that what you just committed to your long-lived branches actually matches the feature you just worked on (tested, approved).
- Encouraging rebase also (by inference) encourages the use of `push --force`. Enough said.

1 Reply • Share ›

Hjulle ➚ Glenn Austin • a year ago

Huh? No. That is not what Adam is saying.
* Nowhere in this this article does he promote squashing. You can squash if you want to during the interactive rebase, but there you can also preserve partial commits in any way you want. You can even split it into more parts during that step.
* That was not super clear in the article, but only feature/hotfix branches should be rebased. Release branches should always be merged onto master, otherwise the tag wouldn't be contained on the One Branch.
* The trick about using rebase here is that you *delete* the branch (both locally and remote) after rebasing. Since the old branch is gone, there is no need to force push.

Reply • Share ›

Adam Ruka Mod ➚ Hjulle • a year ago

Hey Hjulle,

thanks for the comment. I hope I made it clear in the description that only feature branches (**not** hotfixes!) can be rebased, release and hotfix branches are always merged back to master.

^ | v • Reply • Share ›

Glenn Austin ➚ Hjulle • a year ago

Having done both merge from "main" to feature branches and rebasing feature branches, rebasing is NOT an easy thing to manage, as you have to then go back and verify *each re-commit* during the rebase process to make sure you haven't introduced a new error, or lost code somewhere. I *guarantee* that if you do not verify each step, you can end up with an un-usable branch, with no history as to where the corruption happened -- since you just "rewrote history." If, instead, you merge from "main" to the feature branch, you have the complete history including what changes needed to be resolved in the feature branch to be "clean" against the "main" branch. Plus, you can resolve any issues *on the feature branch* before committing the entire feature back to your "main" branch.
Plus, if you look at Adam's response to Jason's message -- he *does* promote squashing the final commit. I understand the reasoning behind it -- if you're just squashing to make the log easier to "parse," simply add the branch name to your git log command. If you're squashing to make future reverts easy, you can *never* be certain that your squash commit is exactly the same code that you have already written and tested. It *should* be, but because it's a roll-up of all of the changes from an older "main" version there's no guarantee that it *will* be (and I've seen -- and fixed -- the results of those squash commit failures more than a few times).

^ | v • Reply • Share ›

Adam Ruka Mod ➚ Glenn Austin • a year ago

Hi Glenn,

thanks for your comments. Just to make a few things clear:

1. I don't "promote" squashing. The commenter was just asking whether it's something that's compatible with the workflow, that's all.

Let me quote the article directly:

> The choice of the feature branch integration method is immaterial as far as the workflow is concerned.
> It should be based on personal or team preference,
> however the branching model will work exactly the same,
> regardless of which option is chosen.
> My personal recommendation is to use option #1.

2. I fundamentally disagree with you on maintaining useful history. If you're merging every feature branch, then

see more

^ | v • Reply • Share ›

Glenn Austin ➚ Adam Ruka • a year ago

Merging requires only one resolve step, rebase requires a resolve step *for every commit on the feature branch.* Because of this, there are far more opportunities for mistakes to happen.
Having developed and maintained fairly large projects, I've seen the benefits of "WIP commits" because they can indicate the though processes of the original developer -- which is FAR more beneficial than the "sanitized" result of rebasing or squashing. Those thought processes are crucial when addressing issues in the future -- because *nobody* writes perfect code against perfect specifications with perfect architecture. Therefore, *ANY* clue as to the development process of the feature is critical when addressing issues with that feature.
As far as the argument against "spaghetti history" -- you *do* know you can look at just the logs of a single branch, so it's easy to limit your log message list to just that branch, which will ignore the logs of the feature branch?

1 • Reply • Share ›

Mattias Blom ➚ Adam Ruka • a year ago

> you push garbage, "work-in-pogress" commits [...] This doesn't help you - on the contrary, it makes things more difficult to find

This garbage is many times a great explanation/documentation of how a bug came into existence. Mind you I am a proponent of TDD so bugs should be a rare occurrence and therefore the time needed to examine the full unadulterated log shouldn't be hard to justify. Contrived example: Feature has a bad bug when persisting data. Full unsquashed log shows tens of WIP-commits before noon going back and forth on gui and colors. Backend code is developed and commited in a single commit at 4pm. Test is added last at 5pm and code was merged/pushed 5.15. This tells the story of a developer that has the wrong priorities. Lessons could be learned and team culture could be improved (cause it isn't the individual developer that has a problem, it's the team). Had it been one squashed and rebased commit, there wouldn't be a lesson and more bugs would be introduced. (but hey, at least they can be found quickly in the git history!)

• Reply • Share ›

Belal Mostafa • a year ago

in case we used the develop and master branch as you suggested where develop contains the latest version of our code, and master branch points to the latest tag in production , how would that be different than gitFlow, it is very similar to gitFlow in this case, am I right ?

1 • Reply • Share ›

Adam Ruka Mod ➚ Belal Mostafa • a year ago

Hey Belal,

the flows are similar, but not the same (otherwise writing this long article would make little sense ;p). The release and hotfix process will be simpler. The history will be cleaner (and more useful as a result). The day-to-day operations will be easier for the devs. But you'll get the same versioning power as with GitFlow.

• Reply • Share ›

Belal Mostafa ➚ Adam Ruka • a year ago

I will be happy as well to contribute to the source code of the CLI tool which you are planning to develop

• Reply • Share ›

Adam Ruka Mod ➚ Belal Mostafa • a year ago

Thanks, that's good to know :)

• Reply • Share ›

Belal Mostafa ➚ Adam Ruka • a year ago

yes, actually I have read the three articles you wrote about your gitflow and your branching model, I admit it is too informative to me, and show me some disadvantages about gitFlow that I would have never notice unless I fall into it. so, of course, thank you after all. but I still have one quesiton. when you say that the history will be cleaner then the only option is to do the rebasing + --no-ff merge option, right ? otherwise, i will end up with the gitFlow issues again, right ?

• Reply • Share ›

Adam Ruka Mod ➚ Belal Mostafa • a year ago

No, the history will be cleaner also because each release and hotfix will only create at most one new commit (the merge to develop, the master one will be always a fast-forward). With GitFlow, each release/hotfix results in two new merge commits, making the history much less clean.

• Reply • Share ›

Belal Mostafa ➚ Adam Ruka • a year ago

yeah, yeah, ,you are right, I got what you mean now , thanks Adam

• Reply • Share ›

Matias • a year ago

"Feature branches often exist only in the developer's repository, and are never pushed"... Isn't this advocating against code reviewing? Normally I would say that local feature branches should be mirrored by remote ones (or vice versa), thus allowing pull requests to exist.

1 • Reply • Share ›

Adam Ruka Mod → Matias • a year ago

Hey Matias,

no, not at all. I'm all for using Pull Requests (that's how I work day-to-day). The two aren't really related. Many tools (Gerrit and ReviewBoard, for example) allow you to submit Pull Requests without pushing any code to a central repository.

1 • Reply • Share ›

Maxim Kulkin → Adam Ruka • a year ago

That is not true. Since they show it to reviewers they have to store it somewhere on server side. Thing is that Gerrit does not work on top of vanilla Git, but rather itself being a Git implementation, it supports "hidden" branches, that are not shown with regular branch commands.

• Reply • Share ›

Adam Ruka Mod → Maxim Kulkin • a year ago

That's just nitpicking. The important thing is that you can submit Pull Requests without pushing code. How they implement it is irrelevant.

• Reply • Share ›

Maxim Kulkin → Adam Ruka • a year ago

No, the important thing is that you ALWAYS need to push branch to remote repository. How you think (and tell people) about it is irrelevant.

• Reply • Share ›

Adam Ruka Mod → Maxim Kulkin • a year ago

What BS is this? No, you do not "ALWAYS need to push branch to remote repository". ReviewBoard, for example (which I already mentioned above), does not require this. Unless you know every single code review tool in existence, your comment is pure nonsense.

• Reply • Share ›

Maxim Kulkin → Adam Ruka • a year ago

I do not need to know every tool, I know how computers (and networking) works. Unless you're reviewing your code by yourself on your own computer, you need to upload (push) it to a server. You do not need a CS degree to understand that. Whether you do it through VCS itself or through a custom (HTTP) API - that does not matter. Although tool that uses custom API while Git itself has everything needed kind of ridiculous. It's like editing C++ code in a notepad while there are specialized IDEs for that, programming in C++ and not using classes.

• Reply • Share ›

Adam Ruka Mod → Maxim Kulkin • a year ago

The original comment was in the context of a very particular question - about pushing feature branches. That's it. Not other "uploads". This sort of nitpick comments are a waste of time.

1 • Reply • Share ›

Seb Kécetac • 21 days ago

I'm interested in this paragraph:
"If your project needs to maintain multiple simultaneous yet incompatible release versions that way, then OneFlow won't work for you out of the box..."
Well, the Gitflow "support" branch was kinda intended to do that, but in the end it's just a branch created from a master tag that you're never supposed to merge back anywhere and certainly not on master. Would you know a branching model more adapted to maintain multiple simultaneous yet incompatible release versions? Or any advice to deal with this huge problematic that you described as "the main challenge is usually in the interactions between the versions and how to effectively share code between them"?

• Reply • Share ›

deevel oper • a month ago

So, basically this was created to control how a repo history looks? Seems like your fighting against the very nature of git so that you can have a clean git diagram. Honestly, I can't say that I've ever "struggled" with git history other than someone deleting / or rebasing incorrectly. I sort of felt the whole purpose of git was to maintain ALL the history. If you need something, you search for it, and if it takes you an extra 5 to 10 minutes to find it because there's a lot of stuff in the history, oh well. At least it's there and not gone, and you can tell who did it and when they did it.

Don't get me wrong, if it works for you and others, great, keep using it. I don't plan on using it, however; I commend you for putting so much time and effort into it. Anything that helps people adds value to the community. Great write up.

• Reply • Share ›

Adam Ruka Mod ➚ deevel oper • a month ago

I don't agree with your conclusions about the difficulty of searching through history, but I can respect your opinion. Thanks for the kind words!

6 • Reply • Share ›

Starman • 2 months ago

This is a much better explaination than the original article. It's great that you talk about the tradeoffs, and when not to use this branching model.

• Reply • Share ›

Adam Ruka Mod ➚ Starman • 2 months ago

Thank you, I really appreciate it :-)

• Reply • Share ›

jimstigler • 2 months ago

I have a specific question regarding OneFlow. Let's say I have a release (tagged 1.0) on master. I keep working on the project, adding several new features which I then merge into master. I then find a bug that needs urgent fixing for everyone using version 1.0. I make a branch called hotfix, from the tagged commit on master, and fix the bug. Now comes my question: I'd like to merge the fix into master and tag the new tip of master as version 1.1. But, I don't want all of the other changes in master that have been added after version 1.0 - I want to save these other things for version 2.0. In my understanding of OneFlow, if I merge my hotfix back into master, then tag the tip, version 1.1 will include not only the hotfix but also everything else I've added to master since version 1.0. I confess I'm a git newbie, so maybe I misunderstand. But any advice would be welcome. I like your idea of simplifying GitFlow. But I need to understand how my particular need would be addressed. Thanks!

• Reply • Share ›