NAME

gitworkflows - An overview of recommended workflows with Git

SYNOPSIS

git *

DESCRIPTION

This document attempts to write down and motivate some of the workflow elements used for git.git itself. Many ideas apply in general, though the full workflow is rarely required for smaller projects with fewer people involved.

We formulate a set of *rules* for quick reference, while the prose tries to motivate each of them. Do not always take them literally; you should value good reasons for your actions higher than manpages such as this one.

SEPARATE CHANGES

As a general rule, you should try to split your changes into small logical steps, and commit each of them. They should be consistent, working independently of any later commits, pass the test suite, etc. This makes the review process much easier, and the history much more useful for later inspection and analysis, for example with **git-blame(1)** and **git-bisect(1)**.

To achieve this, try to split your work into small steps from the very beginning. It is always easier to squash a few commits together than to split one big commit into several. Don't be afraid of making too small or imperfect steps along the way. You can always go back later and edit the commits with git rebase --interactive before you publish them. You can use git stash save --keep-index to run the test suite independent of other uncommitted changes; see the EXAMPLES section of **git-stash(1)**.

MANAGING BRANCHES

There are two main tools that can be used to include changes from one branch on another: **git-merge(1)** and **git-cherry-pick(1)**.

Merges have many advantages, so we try to solve as many problems as possible with merges alone. Cherry-picking is still occasionally useful; see Merging upwards below for an example.

Most importantly, merging works at the branch level, while cherry-picking works at the commit level. This means that a merge can carry over the changes from 1, 10, or 1000 commits with equal ease, which in turn means the workflow scales much better to a large number of contributors (and contributions). Merges are also easier to understand because a merge commit is a promise that all changes from all its parents are now included.

There is a tradeoff of course: merges require a more careful branch management. The following subsections discuss the important points.

Graduation

As a given feature goes from experimental to stable, it also graduates between the corresponding branches of the software. git.git uses the following *integration branches*:

- *maint* tracks the commits that should go into the next maintenance release, i.e., update of the last released stable version;
- *master* tracks the commits that should go into the next release;
- *next* is intended as a testing branch for topics being tested for stability for master.

There is a fourth official branch that is used slightly differently:

- *pu* (proposed updates) is an integration branch for things that are not quite ready for inclusion yet (see Integration Branches below).

Each of the four branches is usually a direct descendant of the one above it.

Conceptually, the feature enters at an unstable branch (usually *next* or *pu*), and graduates to *master* for the next release once it is considered stable enough.

Merging upwards

The downwards graduation discussed above cannot be done by actually merging downwards, however, since that would merge *all* changes on the unstable branch into the stable one. Hence the following:

Example 1. Merge upwards

Always commit your fixes to the oldest supported branch that require them. Then (periodically) merge the integration branches upwards into each other.

This gives a very controlled flow of fixes. If you notice that you have applied a fix to e.g. *master* that is also required in *maint*, you will need to cherry-pick it (using **git-cherry-pick(1)**) downwards. This will happen a few times and is nothing to worry about unless you do it very frequently.

Topic branches

Any nontrivial feature will require several patches to implement, and may get extra bugfixes or improvements during its lifetime.

Committing everything directly on the integration branches leads to many problems: Bad commits cannot be undone, so they must be reverted one by one, which creates confusing histories and further error potential when you forget to revert part of a group of changes. Working in parallel mixes up the changes, creating further confusion.

Use of topic branches solves these problems. The name is pretty self explanatory, with a caveat that comes from the merge upwards rule above:

Example 2. Topic branches

Make a side branch for every topic (feature, bugfix, ...). Fork it off at the oldest integration branch that you will eventually want to merge it into.

Many things can then be done very naturally:

- To get the feature/bugfix into an integration branch, simply merge it. If the topic has evolved further in the meantime, merge again. (Note that you do not necessarily have to merge it to the oldest integration branch first. For example, you can first merge a bugfix to *next*, give it some testing time, and merge to *maint* when you know it is stable.)
- If you find you need new features from the branch *other* to continue working on your topic, merge *other* to *topic*. (However, do not do this just habitually, see below.)
- If you find you forked off the wrong branch and want to move it back in time, use **git-rebase(1)**.

Note that the last point clashes with the other two: a topic that has been merged elsewhere should not be rebased. See the section on RECOVERING FROM UPSTREAM REBASE in **git-rebase(1)**.

We should point out that habitually (regularly for no real reason) merging an integration branch into your topics — and by extension, merging anything upstream into anything downstream on a regular basis — is frowned upon:

Example 3. Merge to downstream only at well-defined points

Do not merge to downstream except with a good reason: upstream API changes affect your branch; your branch no longer merges to upstream cleanly; etc.

Otherwise, the topic that was merged to suddenly contains more than a single (well-separated) change. The many resulting small merges will greatly clutter up history. Anyone who later investigates the history of a file will have to find out whether that merge affected the topic in development. An upstream might even inadvertently be merged into a more stable branch. And so on.

Throw–away integration

If you followed the last paragraph, you will now have many small topic branches, and occasionally wonder how they interact. Perhaps the result of merging them does not even work? But on the other hand, we want to avoid merging them anywhere stable because such merges cannot easily be undone.

The solution, of course, is to make a merge that we can undo: merge into a throw-away branch.

Example 4. Throw-away integration branches

To test the interaction of several topics, merge them into a throw-away branch. You must never base any work on such a branch!

If you make it (very) clear that this branch is going to be deleted right after the testing, you can even publish this branch, for example to give the testers a chance to work with it, or other developers a chance to see if their in-progress work will be compatible. git.git has such an official throw-away integration branch called *pu*.

Branch management for a release

Assuming you are using the merge approach discussed above, when you are releasing your project you will need to do some additional branch management work.

A feature release is created from the *master* branch, since *master* tracks the commits that should go into the next feature release.

The *master* branch is supposed to be a superset of *maint*. If this condition does not hold, then *maint* contains some commits that are not included on *master*. The fixes represented by those commits will therefore not be included in your feature release.

To verify that *master* is indeed a superset of *maint*, use git log:

Example 5. Verify *master* is a superset of *maint*

git log master..maint

This command should not list any commits. Otherwise, check out *master* and merge *maint* into it.

Now you can proceed with the creation of the feature release. Apply a tag to the tip of *master* indicating the release version:

Example 6. Release tagging

git tag -s -m Git X.Y.Z vX.Y.Z master

You need to push the new tag to a public Git server (see DISTRIBUTED WORKFLOWS below). This makes the tag available to others tracking your project. The push could also trigger a post-update hook to perform release-related items such as building release tarballs and preformatted documentation pages.

Similarly, for a maintenance release, *maint* is tracking the commits to be released. Therefore, in the steps above simply tag and push *maint* rather than *master*.

Maintenance branch management after a feature release

After a feature release, you need to manage your maintenance branches.

First, if you wish to continue to release maintenance fixes for the feature release made before the recent one, then you must create another branch to track commits for that previous release.

To do this, the current maintenance branch is copied to another branch named with the previous release version number (e.g. maint-X.Y.(Z-1) where X.Y.Z is the current release).

Example 7. Copy maint

git branch maint-X.Y.(Z-1) maint

The *maint* branch should now be fast-forwarded to the newly released code so that maintenance

fixes can be tracked for the current release:

Example 8. Update maint to new release

- git checkout maint
- git merge --ff-only master

If the merge fails because it is not a fast-forward, then it is possible some fixes on *maint* were missed in the feature release. This will not happen if the content of the branches was verified as described in the previous section.

Branch management for next and pu after a feature release

After a feature release, the integration branch *next* may optionally be rewound and rebuilt from the tip of *master* using the surviving topics on *next*:

Example 9. Rewind and rebuild next

- git checkout next
- git reset --hard master
- git merge ai/topic_in_next1
- git merge ai/topic_in_next2
- ...

The advantage of doing this is that the history of *next* will be clean. For example, some topics merged into *next* may have initially looked promising, but were later found to be undesirable or premature. In such a case, the topic is reverted out of *next* but the fact remains in the history that it was once merged and reverted. By recreating *next*, you give another incarnation of such topics a clean slate to retry, and a feature release is a good point in history to do so.

If you do this, then you should make a public announcement indicating that *next* was rewound and rebuilt.

The same rewind and rebuild process may be followed for *pu*. A public announcement is not necessary since *pu* is a throw-away branch, as described above.

DISTRIBUTED WORKFLOWS

After the last section, you should know how to manage topics. In general, you will not be the only person working on the project, so you will have to share your work.

Roughly speaking, there are two important workflows: merge and patch. The important difference is that the merge workflow can propagate full history, including merges, while patches cannot. Both workflows can be used in parallel: in git.git, only subsystem maintainers use the merge workflow, while everyone else sends patches.

Note that the maintainer(s) may impose restrictions, such as Signed-off-by requirements, that all commits/patches submitted for inclusion must adhere to. Consult your project's documentation for more information.

Merge workflow

The merge workflow works by copying branches between upstream and downstream. Upstream can merge contributions into the official history; downstream base their work on the official history.

There are three main tools that can be used for this:

- **git-push(1)** copies your branches to a remote repository, usually to one that can be read by all involved parties;
- **git-fetch(1)** that copies remote branches to your repository; and
- **git-pull(1)** that does fetch and merge in one go.

Note the last point. Do *not* use *git pull* unless you actually want to merge the remote branch.

Getting changes out is easy:

Example 10. Push/pull: Publishing branches/topics

git push <remote> <branch> and tell everyone where they can fetch from.

You will still have to tell people by other means, such as mail. (Git provides the **git-request-pull(1)** to send preformatted pull requests to upstream maintainers to simplify this task.)

If you just want to get the newest copies of the integration branches, staying up to date is easy too:

Example 11. Push/pull: Staying up to date

Use git fetch <remote> or git remote update to stay up to date.

Then simply fork your topic branches from the stable remotes as explained earlier.

If you are a maintainer and would like to merge other people's topic branches to the integration branches, they will typically send a request to do so by mail. Such a request looks like

Please pull from
<url> <branch>

In that case, *git pull* can do the fetch and merge in one go, as follows.

Example 12. Push/pull: Merging remote topics

git pull <url> <branch>

Occasionally, the maintainer may get merge conflicts when he tries to pull changes from downstream. In this case, he can ask downstream to do the merge and resolve the conflicts themselves (perhaps they will know better how to resolve them). It is one of the rare cases where downstream *should* merge from upstream.

Patch workflow

If you are a contributor that sends changes upstream in the form of emails, you should use topic branches as usual (see above). Then use **git-format-patch(1)** to generate the corresponding emails (highly recommended over manually formatting them because it makes the maintainer's life easier).

Example 13. format-patch/am: Publishing branches/topics

- git format-patch -M upstream..topic to turn them into preformatted patch files
- git send-email --to=<recipient> <patches>

See the **git-format-patch(1)** and **git-send-email(1)** manpages for further usage notes.

If the maintainer tells you that your patch no longer applies to the current upstream, you will have to rebase your topic (you cannot use a merge because you cannot format-patch merges):

Example 14. format-patch/am: Keeping topics up to date

git pull --rebase <url> <branch>

You can then fix the conflicts during the rebase. Presumably you have not published your topic other than by mail, so rebasing it is not a problem.

If you receive such a patch series (as maintainer, or perhaps as a reader of the mailing list it was sent to), save the mails to files, create a new topic branch and use *git am* to import the commits:

Example 15. format-patch/am: Importing patches

git am < patch

One feature worth pointing out is the three-way merge, which can help if you get conflicts: git am -3 will use index information contained in patches to figure out the merge base. See **git-am(1)** for other options.

SEE ALSO

gittutorial(7), **git-push(1)**, **git-pull(1)**, **git-merge(1)**, **git-rebase(1)**, **git-format-patch(1)**, **git-send-email(1)**, **git-am(1)**

GIT

Part of the **git(1)** suite.