maintain-git

From: Junio C Hamano <gitster@pobox.com> Date: Wed, 21 Nov 2007 16:32:55 -0800 Subject: Addendum to "MaintNotes" Abstract: Imagine that Git development is racing along as usual, when our friendly neighborhood maintainer is struck down by a wayward bus. Out of the hordes of suckers (loyal developers), you have been tricked (chosen) to step up as the new maintainer. This howto will show you "how to" do it. Content-type: text/asciidoc

How to maintain Git ==========

Activities ----------

The maintainer’s Git time is spent on three activities.

- Communication (45%)

```
Mailing list discussions on general design, fielding user
questions, diagnosing bug reports; reviewing, commenting on,
suggesting alternatives to, and rejecting patches.
```

- Integration (50%)

```
Applying new patches from the contributors while spotting and
correcting minor mistakes, shuffling the integration and
testing branches, pushing the results out, cutting the
releases, and making announcements.
```

- Own development (5%)

```
Scratching my own itch and sending proposed patch series out.
```

The Policy ----------

The policy on Integration is informally mentioned in "A Note from the maintainer" message, which is periodically posted to the mailing list after each feature release is made:

- Feature releases are numbered as vX.Y.0 and are meant to contain bugfixes and enhancements in any area, including functionality, performance and usability, without regression.

- Maintenance releases are numbered as vX.Y.Z (0 < Z) and are meant to contain only bugfixes for the corresponding vX.Y.0 feature release and earlier maintenance releases vX.Y.W (W < Z).

- The ’master’ branch is used to prepare for the next feature release. In other words, at some point, the tip of ’master’ branch is tagged as vX.(Y+1).0, when vX.Y.0 is the latest feature release.

- ’maint’ branch is used to prepare for the next maintenance release. After the feature release vX.Y.0 is made, the tip of ’maint’ branch is set to that release, and bugfixes will accumulate on the branch, and at some point, the tip of the branch is tagged with vX.Y.1, vX.Y.2, and so on.

- ’next’ branch is used to publish changes (both enhancements and fixes) that (1) have worthwhile goal, (2) are in a fairly good shape suitable for everyday use, (3) but have not yet demonstrated to be regression free. Reviews from contributors on the mailing list help to make the determination. After a topic is merged to ’next’, it is tested for at least 7 calendar days before getting merged to ’master’.

- ’seen’ branch is used to publish other proposed changes that do not yet pass the criteria set for ’next’ (see above), but there is no promise that ’seen’ will contain everything. A topic that had no reviewer reaction may not be picked up.

- A new topic will first get merged to ’seen’, unless it is trivially correct and clearly urgent, in which case it may be directly merged to ’next’ or even to ’master’.

- If a topic that was picked up to ’seen’ becomes and stays inactive for 3 calendar weeks without having seen a clear consensus that it is good enough to be moved to ’next’, the topic may be discarded from ’seen’. Interested parties are still free to revive the topic. For the purpose of this guideline, the definition of being "inactive" is that nobody has discussed the topic, no new iteration of the topic was posted, and no responses to the review comments were given.

- The tips of ’master’ and ’maint’ branches will not be rewound to allow people to build their own customization on top of them. Early in a new development cycle, ’next’ is rewound to the tip of ’master’ once, but otherwise it will not be rewound until the end of the cycle.

- Usually ’master’ contains all of ’maint’ and ’next’ contains all of ’master’. ’seen’ contains all the topics merged to ’next’, but is rebuilt directly on ’master’.

- The tip of ’master’ is meant to be more stable than any tagged releases, and the users are encouraged to follow it.

- The ’next’ branch is where new action takes place, and the users are encouraged to test it so that regressions and bugs are found before new topics are merged to ’master’.

- When a problem is found in a topic in ’next’, the topic is marked not to be merged to ’master’. Follow-up patches are discussed on the mailing list and applied to the topic after being reviewed and then the topic is merged (again) to ’next’. After going through the usual testing in ’next’, the entire (fixed) topic is merged to ’master’.

- One release cycle for a feature release is expected to last for eight to ten weeks. A few "release candidate" releases are expected to be tagged about a week apart before the final release, and a "preview" release is tagged about a week before the first release candidate gets tagged.

- After the preview release is tagged, topics that were well reviewed may be merged to ’master’ before spending the usual 7 calendar days in ’next’, with the expectation that any bugs in them can be caught and fixed in the release candidates before the final release.

- After the first release candidate is tagged, the contributors are strongly encouraged to focus on finding and fixing new regressions introduced during the cycle, over addressing old bugs and any new features. Topics stop getting merged down from ’next’ to ’master’, and new topics stop getting merged to ’next’. Unless they are fixes to new regressions in the cycle, that is.

- Soon after a feature release is made, the tip of ’maint’ gets fast-forwarded to point at the release. Topics that have been kept in ’next’ are merged down to ’master’ and a new development cycle starts.

Note that before v1.9.0 release, the version numbers used to be structured slightly differently. vX.Y.Z were feature releases while vX.Y.Z.W were maintenance releases for vX.Y.Z.

Because most of the lines of code in Git are written by individual contributors, and contributions come in the form of e-mailed patches published on the mailing list, the project maintains a mapping from individual commits to the Message-Id of the e-mail that resulted in the commit, to help tracking the origin of the changes. The notes in "refs/notes/amlog" are used for this purpose, and are published along with the broken-out branches to the maintainer’s repository.

A Typical Git Day ----------

A typical Git day for the maintainer implements the above policy by doing the following:

- Scan mailing list. Respond with review comments, suggestions etc. Kibitz. Collect potentially usable patches from the mailing list. Patches about a single topic go to one mailbox (I read my mail in Gnus, and type C-o to save/append messages in files in mbox format).

- Write his own patches to address issues raised on the list but nobody has stepped up to solve. Send it out just like other contributors do, and pick them up just like patches from other contributors (see above).

- Review the patches in the saved mailboxes. Edit proposed log message for typofixes and clarifications, and add Acks collected from the list. Edit patch to incorporate "Oops, that should have been like this" fixes from the discussion.
- Classify the collected patches and handle ’master’ and ’maint’ updates:
- Obviously correct fixes that pertain to the tip of ’maint’ are directly applied to ’maint’.
- Obviously correct fixes that pertain to the tip of ’master’ are directly applied to ’master’.
- Other topics are not handled in this step.

```
This step is done with "git am".

$ git checkout master    ;# or "git checkout maint"
$ git am -sc3 mailbox
$ make test

In practice, almost no patch directly goes to 'master' or
'maint'.

Applying the e-mailed patches using "git am" automatically records
the mappings from 'Message-Id' to the applied commit in the "amlog"
notes. Periodically check that this is working with "git show -s
--notes=amlog $commit".

This mapping is maintained with the aid of the "post-applypatch"
hook found in the 'todo' branch. That hook should be installed
before applying patches. It is also helpful to carry forward any
relevant amlog entries when rebasing, so the following config may
be useful:

  [notes]
rewriteRef = refs/notes/amlog

Avoid "cherry-pick", as it does not propagate notes by design. Use
either "git commit --amend" or "git rebase" to make corrections to
an existing commit, even for a single-patch topic.

Make sure that a push refspec for 'refs/notes/amlog' is in the
remote configuration for publishing repositories. A few sample
configurations look like the following:
```

```
  [remote "github"]
url = https://github.com/gitster/git
pushurl = github.com:gitster/git.git
mirror

  [remote "github2"]
url = https://github.com/git/git
fetch = +refs/heads/*:refs/remotes/github2/*
pushurl = github.com:git/git.git
push = refs/heads/maint:refs/heads/maint
push = refs/heads/master:refs/heads/master
push = refs/heads/next:refs/heads/next
push = +refs/heads/seen:refs/heads/seen
push = +refs/notes/amlog
```

- Review the last issue of "What’s cooking" message, review the topics ready for merging (topic→master and topic→maint). Use "Meta/cook -w" script (where Meta/ contains a checkout of the ’todo’ branch) to aid this step.

```
And perform the merge.  Use "Meta/Reintegrate -e" script (see
later) to aid this step.

$ Meta/cook -w last-issue-of-whats-cooking.mbox

$ git checkout master    ;# or "git checkout maint"
$ echo ai/topic | Meta/Reintegrate -e ;# "git merge ai/topic"
$ git log -p ORIG_HEAD.. ;# final review
$ git diff ORIG_HEAD..   ;# final review
$ make test              ;# final review

If the tip of 'master' is updated, also generate the preformatted
documentation and push the out result to git-htmldocs and
git-manpages repositories.
```

- Handle the remaining patches:

- Anything unobvious that is applicable to ’master’ (in other words, does not depend on anything that is still in ’next’ and not in ’master’) is applied to a new topic branch that is forked from the tip of ’master’ (or the last feature release, which is a bit older than ’master’). This includes both enhancements and unobvious fixes to ’master’. A topic branch is named as ai/topic where "ai" is two-letter string named after author’s initial and

"topic" is a descriptive name of the topic (in other words, "what's the series is about").

- An unobvious fix meant for 'maint' is applied to a new topic branch that is forked from the tip of 'maint' (or the oldest and still relevant maintenance branch). The topic may be named as ai/maint-topic.
- Changes that pertain to an existing topic are applied to the branch, but:
- obviously correct ones are applied first;
- questionable ones are discarded or applied to near the tip;
- Replacement patches to an existing topic are accepted only for commits not in 'next'.

```
The initial round is done with:

$ git checkout ai/topic ;# or "git checkout -b ai/topic master"
$ git am -sc3 --whitespace=warn mailbox

and replacing an existing topic with subsequent round is done with:

$ git checkout master...ai/topic ;# try to reapply to the same base
$ git am -sc3 --whitespace=warn mailbox

to prepare the new round on a detached HEAD, and then

$ git range-diff @{-1}...
$ git diff @{-1}

to double check what changed since the last round, and finally

$ git checkout -B @{-1}

to conclude (the last step is why a topic already in 'next' is
not replaced but updated incrementally).

Whether it is the initial round or a subsequent round, the topic
may not build even in isolation, or may break the build when
merged to integration branches due to bugs.  There may already
be obvious and trivial improvements suggested on the list.  The
maintainer often adds an extra commit, with "SQUASH???" in its
title, to fix things up, before publishing the integration
```

```
branches to make it usable by other developers for testing.
These changes are what the maintainer is not 100% committed to
(trivial typofixes etc. are often squashed directly into the
patches that need fixing, without being applied as a separate
"SQUASH???" commit), so that they can be removed easily as needed.
The expectation is that the original author will make corrections
in a reroll.
```

- By now, new topic branches are created and existing topic branches are updated. The integration branches ’next’, ’jch’, and ’seen’ need to be updated to contain them.
- If there are topics that have been merged to ’master’ and should be merged to ’maint’, merge them to ’maint’, and update the release notes to the next maintenance release.
- Review the latest issue of "What’s cooking" again. Are topics that have been sufficiently long in ’next’ ready to be merged to ’master’? Are topics we saw earlier and are in ’seen’ now got positive reviews and are ready to be merged to ’next’?
- If there are topics that have been cooking in ’next’ long enough and should be merged to ’master’, merge them to ’master’, and update the release notes to the next feature release.
- If there were patches directly made on ’maint’, merge ’maint’ to ’master’; make sure that the result is what you want.

```
$ git checkout master
$ git merge -m "Sync with 'maint'" --no-log maint
$ git log -p --first-parent ORIG_HEAD..
$ make test
```

- Prepare to update the ’jch’ branch, which is used to represent somewhere between ’master’ and ’seen’ and often is slightly ahead of ’next’, and the ’seen’ branch, which is used to hold the rest.

```
$ Meta/Reintegrate master..jch >Meta/redo-jch.sh

The result is a script that lists topics to be merged in order to
rebuild the current 'jch'.  Do the same for 'seen'.
```

- Review the Meta/redo-jch.sh and Meta/redo-seen.sh scripts. The former should have a line ’### match next'---the idea is that merging the topics listed before the line on top of ’master’ should result in a tree identical to that of ’next’.

- As newly created topics are usually merged near the tip of ’seen’, add them to the end of the Meta/redo-seen.sh script. Among the topics that were in ’seen’, there may be ones that are not quite ready for ’next’ but are getting there. Move them from Meta/redo-seen.sh to the end of Meta/redo-jch.sh. The expectation is that you’d use ’jch’ as your daily driver as the first guinea pig, so you should choose carefully.

- Now we are ready to start rebuilding ’jch’ and merging topics to ’next’. For each branch whose tip is not merged to ’next’, one of three things can happen:

- The commits are all next-worthy; merge the topic to next;
- The new parts are of mixed quality, but earlier ones are next-worthy; merge the early parts to next;
- Nothing is next-worthy; do not do anything.

```
This step is aided with Meta/redo-jch.sh script created earlier.
If a topic that was already in 'next' gained a patch, the script
would list it as "ai/topic~1".  To include the new patch to the
updated 'next', drop the "~1" part; to keep it excluded, do not
touch the line.

If a topic that was not in 'next' should be merged to 'next', add
it before the '### match next' line.  Then:

$ git checkout --detach master
$ sh Meta/redo-jch.sh -c1

to rebuild the 'jch' branch from scratch.  "-c1" tells the script
to stop merging at the first line that begins with '###'
(i.e. the "### match next" line you added earlier).

At this point, build-test the result.  It may reveal semantic
conflicts (e.g. a topic renamed a variable, another added a new
reference to the variable under its old name), in which case
prepare an appropriate merge-fix first (see appendix), and
rebuild the 'jch' branch from scratch, starting at the tip of
'master', this time without using "-c1" to merge all topics.

Then do the same to 'next'.

$ git checkout next
$ sh Meta/redo-jch.sh -c1 -e
```

```
The "-e" option allows the merge message that comes from the
history of the topic and the comments in the "What's cooking" to
be edited.  The resulting tree should match 'jch^{/^### match next'}'
as the same set of topics are merged on 'master'; otherwise there
is a mismerge. Investigate why and do not proceed until the mismerge
is found and rectified.

If 'master' was updated before you started redoing 'next', then

$ git diff 'jch^{/^### match next}' next

would show differences that went into 'master' (which 'jch' has,
but 'next' does not yet---often it is updates to the release
notes).  Merge 'master' back to 'next' if that is the case.

$ git merge -m "Sync with 'master'" --no-log master

When all is well, clean up the redo-jch.sh script with

$ sh Meta/redo-jch.sh -u

This removes topics listed in the script that have already been
merged to 'master'.  This may lose '### match next' marker;
add it again to the appropriate place when it happens.
```

- Rebuild ’seen’ on top of ’jch’.

```
$ git checkout -B seen jch
$ sh Meta/redo-seen.sh

When all is well, clean up the redo-seen.sh script with

$ sh Meta/redo-seen.sh -u

Double check by running

$ git branch --no-merged seen '??/*'

to see there is no unexpected leftover topics.
```

```
At this point, build-test the result for semantic conflicts, and
if there are, prepare an appropriate merge-fix first (see
appendix), and rebuild the 'seen' branch from scratch, starting at
the tip of 'jch'.
```

- Update "What’s cooking" message to review the updates to existing topics, newly added topics and graduated topics.

```
This step is helped with Meta/cook script.

$ Meta/cook

This script inspects the history between master..seen, finds tips
of topic branches, compares what it found with the current
contents in Meta/whats-cooking.txt, and updates that file.
Topics not listed in the file but are found in master..seen are
added to the "New topics" section, topics listed in the file that
are no longer found in master..seen are moved to the "Graduated to
master" section, and topics whose commits changed their states
(e.g. used to be only in 'seen', now merged to 'next') are updated
with change markers "<<" and ">>".

Look for lines enclosed in "<<" and ">>"; they hold contents from
old file that are replaced by this integration round.  After
verifying them, remove the old part.  Review the description for
each topic and update its doneness and plan as needed.  To review
the updated plan, run

$ Meta/cook -w

which will pick up comments given to the topics, such as "Will
merge to 'next'", etc. (see Meta/cook script to learn what kind
of phrases are supported).
```

- Compile, test and install all four (five) integration branches; Meta/Dothem script may aid this step.
- Format documentation if the ’master’ branch was updated; Meta/dodoc.sh script may aid this step.
- Push the integration branches out to public places; Meta/pushall script may aid this step.

Observations ----------

Some observations to be made.

- Each topic is tested individually, and also together with other topics cooking first in ’seen’, then in ’jch’ and then in ’next’. Until it matures, no part of it is merged to ’master’.

- A topic already in ’next’ can get fixes while still in ’next’. Such a topic will have many merges to ’next’ (in other words, "git log --first-parent next" will show many "Merge branch ’ai/topic’ to next" for the same topic.

- An unobvious fix for ’maint’ is cooked in ’next’ and then merged to ’master’ to make extra sure it is Ok and then merged to ’maint’.

- Even when ’next’ becomes empty (in other words, all topics prove stable and are merged to ’master’ and "git diff master next" shows empty), it has tons of merge commits that will never be in ’master’.

- In principle, "git log --first-parent master..next" should show nothing but merges (in practice, there are fixup commits and reverts that are not merges).

- Commits near the tip of a topic branch that are not in ’next’ are fair game to be discarded, replaced or rewritten. Commits already merged to ’next’ will not be.

- Being in the ’next’ branch is not a guarantee for a topic to be included in the next feature release. Being in the ’master’ branch typically is.

- Due to the nature of "SQUASH???" fix-ups, if the original author agrees with the suggested changes, it is OK to squash them to appropriate patches in the next round (when the suggested change is small enough, the author should not even bother with "Helped-by"). It is also OK to drop them from the next round when the original author does not agree with the suggestion, but the author is expected to say why somewhere in the discussion.

Appendix --------

Preparing a "merge-fix" ~~~~~~~~~~~~~~~~~~~~~~~~

A merge of two topics may not textually conflict but still have conflict at the semantic level. A classic example is for one topic to rename a variable and all its uses, while another topic adds a new use of the variable under its old name. When these two topics are merged together, the reference to the variable newly added by the latter topic will still use the old name in the result.

The Meta/Reintegrate script that is used by redo-jch and redo-seen scripts implements a crude but usable way to work around this issue. When the script merges branch $X, it checks if "refs/merge-fix/$X" exists, and if so, the effect of it is squashed into the result of the mechanical merge. In other words,

```
$ echo $X | Meta/Reintegrate
```

is roughly equivalent to this sequence:

```
$ git merge --rerere-autoupdate $X
$ git commit
$ git cherry-pick -n refs/merge-fix/$X
$ git commit --amend
```

The goal of this "prepare a merge-fix" step is to come up with a commit that can be squashed into a result of mechanical merge to correct semantic conflicts.

After finding that the result of merging branch "ai/topic" to an integration branch had such a semantic conflict, say seen~4, check the problematic merge out on a detached HEAD, edit the working tree to fix the semantic conflict, and make a separate commit to record the fix-up:

```
$ git checkout seen~4
$ git show -s --pretty=%s ;# double check
Merge branch 'ai/topic' to seen
$ edit
$ git commit -m 'merge-fix/ai/topic' -a
```

Then make a reference "refs/merge-fix/ai/topic" to point at this result:

```
$ git update-ref refs/merge-fix/ai/topic HEAD
```

Then double check the result by asking Meta/Reintegrate to redo the merge:

```
$ git checkout seen~5 ;# the parent of the problem merge
$ echo ai/topic | Meta/Reintegrate
$ git diff seen~4
```

This time, because you prepared refs/merge-fix/ai/topic, the resulting merge should have been tweaked to include the fix for the semantic conflict.

Note that this assumes that the order in which conflicting branches are merged does not change. If the reason why merging ai/topic branch needs this merge-fix is because another branch merged earlier to the integration branch changed the underlying assumption ai/topic branch made (e.g. ai/topic branch added a site to refer to a variable, while the other branch renamed that variable and adjusted existing use sites), and if you changed redo-jch (or redo-seen) script to merge ai/topic branch before the other branch, then the above merge-fix should not be applied while merging ai/topic, but should instead be applied while merging the other branch. You would need to move the fix to apply to the other branch, perhaps like this:

```
$ mf=refs/merge-fix
$ git update-ref $mf/$the_other_branch $mf/ai/topic
$ git update-ref -d $mf/ai/topic
```

□□□ #2
□ □□□ □□□ 2026-10-31 20:14:49 UTC
□ □□□ □□□ 2026-10-31 20:14:50 UTC