Intermediate Git

Rebase workflow & best practices

Gokul Das B

FSUG Trivandrum

Sunday 14th March, 2021

What after Beginners' Git?

- Be friendly to explorers
  - Don't surprise them!
  - Don't disappoint them! (with broken builds)
  - Don't frustrate them! (with experimental commits)
- Make it easy to collaborate
- Create a tolerable development history
  - Record the development history as if it's a document
  - Make it progress logically
  - Make it browsable
- Make it easy to use git tools that aid development

Introduction

Commits

A perspective on *commits*

Commits were designed to be sent as **text email patches**. All rules are based on this.

- A branch is a thread of emails (created with: `git format-patch`)
- A commit is a single mail in the thread
- There is usually a cover email at the head of the thread

Example (Patch Thread [1])

```
[PATCH 0/4] HTTPD: Add caching Sergey Ponomarev

    [PATCH 1/4] httpd:  Update to HTTP/1.1 Sergey Ponomarev

    [PATCH 2/4] httpd:  Don't add Date header to response Sergey Ponomarev

    [PATCH 3/4] httpd:  Don't add Last-Modified header to response Sergey Ponomarev

    [PATCH 4/4] httpd:  Support caching via ETag header Sergey Ponomarev
```

[1] From a patchset for BusyBox by Sergey Ponomarev

Cover letter

Example

```
From: "Sergey Ponomarev"<stokito@gmail.com>
Subject: [PATCH 0/4] HTTPD: Add caching
Date: Sat, 08 Aug 2020 22:23:30

Date and Last-Modified headers now can be disabled while there still enabled by default.
I hope in future versions they become disabled by default and later removed at all.
I checked and Cache-Control works fine in Chrome and Firefox even if Date is not present.
Last-Modified can be replaced with ETag.
Another difference with previous patch is that now ETag will be returned even in 304 response.
This is a requirement of spec and it makes sense because If-None-Match may have many ETags
but client should know which ETag matched.

You can clone the code from https://github.com/stokito/busybox/commits/caching


Sergey Ponomarev (4):
  httpd: Update to HTTP/1.1
  httpd: Don't add Date header to response
  httpd: Don't add Last-Modified header to response
  httpd: Support caching via ETag header

 networking/httpd.c | 131 ++++++++++++++++++++++++++++++++++++-------
 1 file changed, 111 insertions(+), 20 deletions(-)

--
2.25.1
```

Patch mail

Example

```
From: "Sergey Ponomarev"<stokito@gmail.com>
Subject: [PATCH 3/4] httpd: Don't add Last-Modified header to response
Date: Sat, 08 Aug 2020 22:23:33

The Last-Modified header is used for caching.  The client (browser) will
send back the received date to server via If-Modified-Since request
--** SNIP **--

Signed-off-by: Sergey Ponomarev <stokito at gmail.com>
---
 networking/httpd.c | 22 ++++++++++++++++++----
 1 file changed, 18 insertions(+), 4 deletions(-)

diff --git a/networking/httpd.c b/networking/httpd.c
index 7a429d2b5..1cea33ddd 100644
--- a/networking/httpd.c
+++ b/networking/httpd.c
@@ -215,6 +215,16 @@
 //config:
+//config:config FEATURE_HTTPD_LAST_MODIFIED
--** SNIP **--
+                len += sprintf(iobuf + len, "Last-Modified: %s\r\n", date_str);
+#endif
         /* This should be "Transfer-Encoding", not "Content-Encoding": 
--
2.25.1
```

Patch mail: An explanation

1. All text lines limited to **72 characters**
2. **Subject line**: Main commit message
   - Explains what the patch does, if applied (notice the imperative tone)
   - Limited to **50 characters** due to patch tags
3. **Body Main**: commit message
   - Be descriptive. Be free with tone
   - Limited to **72 characters** like rest of the text mail
4. Space after '---' can be used for unofficial messages

Example (Commit message of example patch)

```
httpd: Don't add Last-Modified header to response

The Last-Modified header is used for caching.  The client (browser) will
send back the received date to server via If-Modified-Since request
header.  But both headers MUST be an RFC 1123 formatted string.  And the
formatting consumes resources on request parsing and response
generation.  Instead we can use ETag header.  This simplifies logic and
the only downside is that in JavaScript the document.lastModified will
return null.
```

Introduction

Ideal History

What we want to happen

1. Every commit must work
   - Imagine that a clone or pull doesn't build or work?
   - Each commit should be free of bugs
2. Each commit must contain all related changes
   - Partial changes violate rule: 1
3. Commits/changes should be in logical order
   - Out of order changes are confusing
   - `git bisect` is easier when changes are in order

- The guidelines work well for any workflow
- The guidelines are especially important for email-patch workflow
- Important for *master* branch. Not so much for feature branches

What actually happens!

- Actual development workflow is never ideal
- Commit as often as possible
- Commit in the same order as you develop
- Don't bother with aesthetics:
  - Forget about logical order
  - Forget about combining related changes
- Revert buggy commits
  - This will create *reversion commits*

Reconciling contradictions

1. Create **development**/**feature** branches
2. Follow natural style in those branches
3. **Edit the history** of feature branch
   - **Rebase** feature branch onto master
   - Every commit is self-contained
4. Merge rebased branch to master

Operations

Commits

Interactive staging: Made too many changes?

Stage 'hunks' (blocks of changes) in related groups

Commands

```
# Do this only for new/untracked files
git add -N <filename>

# Stage changes in a file piece-by-piece
git add -p [filename]

# Get a lot more options
git add -i [filename]
```

Patch staging (`git add -p`) allows you to stage, unstage, ignore or split *hunks*

In case of untracked (new files), do (`git add -N`) first

Interactive staging (`git add -i`) gives you a lot more options including patch staging. Other options are status, update, revert and diff

Commit amending: Made a mistake?

In case of mistake in last commit, amend it

Command

```
git add <filename>

git commit --amend
```

1. Modify mistakes in file
2. Stage (`git add`) the corrected files
   - Skip steps 1 & 2 if you want to just modify the commit message
3. Do: `git commit --amend`
4. Modify the commit message

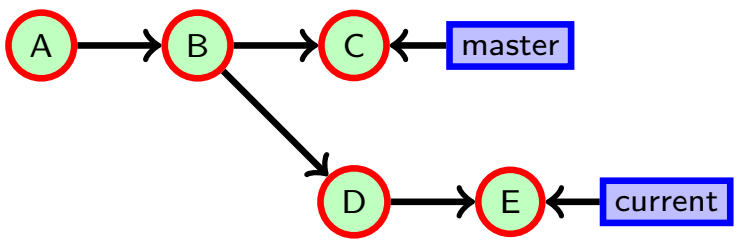
This will overwrite the last commit with corrected content and message

Operations

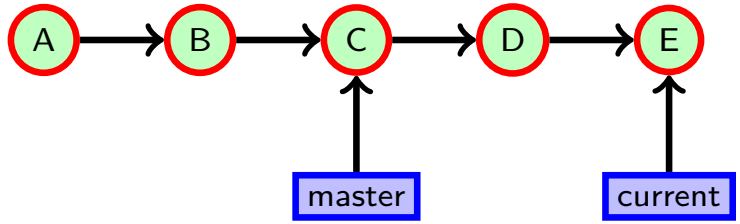
Rebase Workflow

Rebasing: Basic idea

Before rebase



After rebase



- Eg: `git rebase master` (from `current` branch)
- This rebases changes in current branch on master branch
- Rebasing works based on changesets

RULES:

1. Always use fresh branch for commits meant to be rebased
2. NEVER rebase a published branch

Interactive rebasing: Basic idea

Edit history with:

```
git rebase -i master
```

1. A rebase plan/todo will open automatically in a text editor
2. Edit the plan as necessary
3. Save rebase plan
4. Rebase will start, interrupting when manual intervention is needed

Rebase plan: Original

```
pick 032932a 03: Line to be edited
pick 55bdd80 04: Two lines to be split
pick 6814493 05: A buffer line
pick f643d0c 06: This commit should be combined with previous
pick 3148420 07: This commit will be deleted
pick e7b5df9 08: This commit will be switched with next one
pick 874883d 09: This commit will be switched with previous one
pick 14c6893 10: This is a placeholder commit
```

Rebase plan: Edited

```
edit 032932a 03: Line to be edited
edit 55bdd80 04: Two lines to be split
pick 6814493 05: A buffer line
squash f643d0c 06: This commit should be combined with previous
pick 874883d 09: This commit will be switched with previous one
pick e7b5df9 08: This commit will be switched with next one
pick 14c6893 10: This is a placeholder commit
```

Editing commit

Do this when there is any mistake in the content or message of a commit

Rebase plan: edit

```
edit 032932a 03:  Line to be edited
```

1. The rebase will be interrupted for edit
2. Modify the files that needs change
3. Stage the file: `git add <filename>`
4. Continue the rebase: `git rebase --continue`

Splitting commit

Do this to split a commit with unrelated changes

1. The option is same as `edit`
2. The rebase will be interrupted for edit
3. Do a soft reset: `git reset`
4. Do an interactive staging for first commit: `git add -p <filename>`
5. Commit the partial changes: `git commit`
6. Stage and commit the rest of the changes
7. Continue the rebase: `git rebase --continue`

Squash commit

Do this to combine commits with related partial changes

Rebase plan: squash

```
squash f643d0c 06:  This commit should be combined with previous
```

1. The commit will be combined with the previous commit
2. You will be asked for an updated commit message

Delete commit

Do this to delete mistakes and test commits

- Just delete the commit line from the rebase plan

Reorder commits

Do this when the changes are not in logical order

Rebase plan

```
pick 874883d 09:  This commit will be switched with previous one
pick e7b5df9 08:  This commit will be switched with next one
```

- Rearrange the lines in the rebase plan as needed

Final step

Final step is to update the target branch (master) to the current branch

1. Switch to target branch: `git switch master`
2. Run a fast-forward merge: `git merge current`

Fast-forward merge means:

- Target branch (master) will update to the feature branch (current)
- The history will be linear
- No merge commits (no commits with multiple parents)

Operations

Email-Patch Workflow

Prerequisites for email-patch workflow

Git should be configured to send out emails.
Check git-send-email.io for available options.

Step 1: Create patchset

Command

```
git format-patch -o /tmp/patches --cover-letter master..current
```

1. Rebase the feature branch (current) on the target branch (master)
   - Clean up the history in the process
2. Run `git format-patch` command to create patch files
   - Each patch file has one commit in email format
   - `-o /tmp/patches`: Specifies folder to save patch files
   - `--cover-letter`: Specifies a cover letter should be created
   - `master..current`: Specifies starting and ending commits of patchset
   - The patchset excludes `master` and includes `current`
   - Revision selector, like commit hash can be used instead of branch name
   - `-v2`, `-v3`, etc can be used to indicate revised patchsets

Step 2: Edit patchset

1. Patch files in destination folder can be edited with a regular editor
2. Edit subject and body of cover letter (if opted for)
3. Modify patch email if needed (not usually necessary)
   - The subject line corresponds to first line (subject) of commit message
   - The body of email before `---` corresponds to body of commit message
   - Body of email after `---` and before patch are non-git messages
4. Don't edit any machine formatted part of the messages. That includes:
   - Patch and version tags on subject lines
   - Patches/diffs in patch files
   - Footer area of cover letter

Step 3: Send patches

Command

```
git send-email --to=<maintainer-email> --cc=<mailinglist-address> /tmp/patches/*.patch
```

- All patch files will appear as threaded to the cover letter
- The patches can be sent to any number of recipients, including redundant addresses

Applying email patches as maintainer

Command

```
git am /tmp/incoming/*.patch
```

- This requires access to mailbox directory containing patch files
- One method is to 'pipe' email from client (eg: mutt) through `git am`
- This will apply the patches to current branch
- It is recommended to apply patches on a fresh branch, rather than master

Conclusion

Rebase workflow and **email-patch workflow** offer a lot of advantages over normal workflows:

1. Higher quality commit history
2. Less frustration for readers
3. Full freedom during development
4. Resiliance in collaboration
5. Less dependence on platforms

The additional work in these workflows pay off in the way of developer satisfaction