git-format-patch

- PATCH APPLICATION

git-format-patch(1) ====================

NAME ---- git-format-patch - Prepare patches for e-mail submission

SYNOPSIS --------

git format-patch [-k] [(-o|--output-directory) <dir> | --stdout] [--no-thread | --thread[=<style>]] [(--attach|--inline)[=<boundary>] | --no-attach] [-s | --signoff] [--signature=<signature> | --no-signature] [--signature-file=<file>] [-n | --numbered | -N | --no-numbered] [--start-number <n>] [--numbered-files] [--in-reply-to=<message-id>] [--suffix=.<sfx>] [--ignore-if-in-upstream] [--always] [--cover-from-description=<mode>] [--rfc[=<rfc>]] [--subject-prefix=<subject-prefix>] [(--reroll-count|-v) <n>] [--to=<email>] [--cc=<email>] [--[no-]cover-letter] [--quiet] [--commit-list-format=<format-spec>] [--[no-]encode-email-headers] [--no-notes | --notes[=<ref>]] [--interdiff=<previous>] [--range-diff=<previous> [--creation-factor=<percent>]] [--filename-max-length=<n>] [--progress] [<common-diff-options>] [<since> | <revision-range>]

DESCRIPTION -----------

Prepare each non-merge commit with its "patch" in one "message" per commit, formatted to resemble a UNIX mailbox. The output of this command is convenient for e-mail submission or for use with `git` `am`.

A "message" generated by the command consists of three parts:

- A brief metadata header that begins with `From` `<commit>` with a fixed `Mon` `Sep` `17` `00:00:00` `2001` datestamp to help programs like `file`(1) to recognize that the file is an output from this command, fields that record the author identity, the author date, and the title of the change (taken from the first paragraph of the commit log message).

- The second and subsequent paragraphs of the commit log message.

- The "patch", which is the `diff` `-p` `--stat` output (see linkgit:git-diff[1]) between the commit and its parent.

The log message and the patch are separated by a line with a three-dash line.

There are two ways to specify which commits to operate on.

1. A single commit, <*since*>, specifies that the commits leading to the tip of the current branch that are not in the history that leads to the <*since*> to be output.

2. Generic <*revision-range*> expression (see 'SPECIFYING REVISIONS' section in linkgit:gitrevisions[7]) means the commits in the specified range.

The first rule takes precedence in the case of a single <*commit*>. To apply the second rule, i.e., format everything since the beginning of history up until <*commit*>, use the `--root` option: `git format-patch --root <commit>`. If you want to format only <*commit*> itself, you can do this with `git format-patch -1 <commit>`.

By default, each output file is numbered sequentially from 1, and uses the first line of the commit message (massaged for pathname safety) as the filename. With the `--numbered-files` option, the output file names will only be numbers, without the first line of the commit appended. The names of the output files are printed to standard output, unless the `--stdout` option is specified.

If `-o` is specified, output files are created in <*dir*>. Otherwise they are created in the current working directory. The default path can be set with the `format.outputDirectory` configuration option. The `-o` option takes precedence over `format.outputDirectory`. To store patches in the current working directory even when `format.outputDirectory` points elsewhere, use `-o .`. All directory components will be created.

By default, the subject of a single patch is "[PATCH] " followed by the concatenation of lines from the commit message up to the first blank line (see the DISCUSSION section of linkgit:git-commit[1]).

When multiple patches are output, the subject prefix will instead be "[PATCH n/m] ". To force 1/1 to be added for a single patch, use `-n`. To omit patch numbers from the subject, use `-N`.

If given `--thread`, `git-format-patch` will generate `In-Reply-To` and `References` headers to make the second and subsequent patch mails appear as replies to the first mail; this also generates a `Message-ID` header to reference.

OPTIONS ------- :git-format-patch: 1 include::diff-options.adoc[]

`-<n>`
Prepare patches from the topmost <*n*> commits.

`-o <dir>`

`--output-directory=<dir>`
Use <*dir*> to store the resulting files, instead of the current working directory.

`-n`

`--numbered`
Name output in '[PATCH n/m]' format, even with a single patch.

`-N`

`--no-numbered`
Name output in '[PATCH]' format.

`--start-number` `<n>`
Start numbering the patches at *<n>* instead of 1.

`--numbered-files`
Output file names will be a simple number sequence without the default first line of the commit appended.

`-k`

`--keep-subject`
Do not strip/add '[PATCH]' from the first line of the commit log message.

`-s`

`--signoff`
Add a `Signed-off-by` trailer to the commit message, using the committer identity of yourself. See the signoff option in linkgit:git-commit[1] for more information.

`--stdout`
Print all commits to the standard output in mbox format, instead of creating a file for each one.

`--attach[=<boundary>]`
Create multipart/mixed attachment, the first part of which is the commit message and the patch itself in the second part, with `Content-Disposition:` `attachment`.

`--no-attach`
Disable the creation of an attachment, overriding the configuration setting.

`--inline[=<boundary>]`
Create multipart/mixed attachment, the first part of which is the commit message and the patch itself in the second part, with `Content-Disposition:` `inline`.

`--thread[=<style>]`

`--no-thread`
Control addition of `In-Reply-To` and `References` headers to make the second and subsequent mails appear as replies to the first. Also controls generation of the `Message-ID` header to reference.

The optional *<style>* argument can be either `shallow` or `deep`. `shallow` threading makes every mail a reply to the head of the series, where the head is chosen from the cover letter, the `--in-`

`reply-to`, and the first patch mail, in this order. `deep` threading makes every mail a reply to the previous one.

The default is `--no-thread`, unless the `format.thread` configuration is set. `--thread` without an argument is equivalent to `--thread=shallow`.

Beware that the default for `git send-email` is to thread emails itself. If you want `git format-patch` to take care of threading, you will want to ensure that threading is disabled for `git send-email`.

`--in-reply-to=<message-id>`
Make the first mail (or all the mails with `--no-thread`) appear as a reply to the given *<message-id>*, which avoids breaking threads to provide a new patch series.

`--ignore-if-in-upstream`
Do not include a patch that matches a commit in *<until>*..*<since>*. This will examine all patches reachable from *<since>* but not from *<until>* and compare them with the patches being generated, and any patch that matches is ignored.

`--always`
Include patches for commits that do not introduce any change, which are omitted by default.

`--cover-from-description=<mode>`
Control which parts of the cover letter will be automatically populated using the branch's description.

<mode> can have the following values:

`message`;; `default`;; the cover letter subject will be populated with placeholder text. The body of the cover letter will be populated with the branch's description. This is the default mode when no configuration nor command line option is specified. `subject`;; the first paragraph of the branch description will populate the cover letter subject. The remainder of the description will populate the body of the cover letter. `auto`;; if the first paragraph of the branch description is greater than 100 bytes, then the mode will be `message`, otherwise `subject` will be used. `none`;; both the cover letter subject and body will be populated with placeholder text.

`--description-file=<file>`
Use the contents of *<file>* instead of the branch's description for generating the cover letter.

`--subject-prefix=<subject-prefix>`
Use '[<subject-prefix>]' instead of the standard '[PATCH]' prefix in the subject line. This can be used to name a patch series, and can be combined with the `--numbered` option.

The configuration variable `format.subjectPrefix` may also be used to configure a subject prefix to apply to a given repository for all patches. This is often useful on mailing lists which receive patches for several repositories and can be used to disambiguate the patches (with a value of e.g. "PATCH my-project").

`--filename-max-length=<n>`
Instead of the standard 64 bytes, chomp the generated output filenames at around *<n>* bytes (too short a value will be silently raised to a reasonable length). Defaults to the value of the `format.filenameMaxLength` configuration variable, or 64 if unconfigured.

`--rfc[=<rfc>]`
Prepends the string *<rfc>* (defaults to "RFC") to the subject prefix. As the subject prefix defaults to "PATCH", you'll get "RFC PATCH" by default.

RFC means "Request For Comments"; use this when sending an experimental patch for discussion rather than application. "--rfc=WIP" may also be a useful way to indicate that a patch is not complete yet ("WIP" stands for "Work In Progress").

If the convention of the receiving community for a particular extra string is to have it *after* the subject prefix, the string *<rfc>* can be prefixed with a dash ("-") to signal that the rest of the *<rfc>* string should be appended to the subject prefix instead, e.g., `--rfc='-(WIP)'` results in "PATCH (WIP)".

`-v` `<n>`

`--reroll-count=<n>`
Mark the series as the *<n>*-th iteration of the topic. The output filenames have `v<n>` prepended to them, and the subject prefix ("PATCH" by default, but configurable via the `--subject-prefix` option) has `v<n>` appended to it. E.g. `--reroll-count=4` may produce `v4-0001-add-makefile.patch` file that has "Subject: [PATCH v4 1/20] Add makefile" in it. *<n>* does not have to be an integer (e.g. `--reroll-count=4.4`, or `--reroll-count=4rev2` are allowed), but the downside of using such a reroll-count is that the range-diff/interdiff with the previous version does not state exactly which version the new iteration is compared against.

`--to=<email>`
Add a `To:` header to the email headers. This is in addition to any configured headers, and may be used multiple times. The negated form `--no-to` discards all `To:` headers added so far (from config or command line).

`--cc=<email>`
Add a `Cc:` header to the email headers. This is in addition to any configured headers, and may be used multiple times. The negated form `--no-cc` discards all `Cc:` headers added so far (from config or command line).

`--from`

`--from=<ident>`
Use *<ident>* in the `From:` header of each email. In case of a commit email, if the author ident of the commit is not textually identical to the provided *<ident>*, place a `From:` header in the body of the message with the original author. If no *<ident>* is given, or if the option is not passed at all, use the ident of the current committer.

Note that this option is only useful if you are actually sending the emails and want to identify yourself as the sender, but retain the original author (and `git am` will correctly pick up the in-body header). Note also that `git send-email` already handles this transformation for you, and this option should not be used if you are feeding the result to `git send-email`.

`--force-in-body-from`

`--no-force-in-body-from`
With the e-mail sender specified via the `--from` option, by default, an in-body "From:" to identify the real author of the commit is added at the top of the commit log message if the sender is different from the author. With this option, the in-body "From:" is added even when the sender and the author have the same name and address, which may help if the mailing list software mangles the sender's identity. Defaults to the value of the `format.forceInBodyFrom` configuration variable.

`--add-header=<header>`
Add an arbitrary header to the email headers. This is in addition to any configured headers, and may be used multiple times. For example, `--add-header="Organization: git-foo"`. The negated form `--no-add-header` discards **all** (`To:`, `Cc:`, and custom) headers added so far from config or command line.

`--cover-letter`

`--no-cover-letter`
In addition to the patches, generate a cover letter file containing the branch description, commit list and the overall diffstat. You can fill in a description in the file before sending it out.

`--commit-list-format=<format-spec>`
Specify the format in which to generate the commit list of the patch series. The accepted values for format-spec are `shortlog`, `modern` or a format-string prefixed with `log:`. E.g. `log: %s (%an)`. `modern` is the same as `log:%w(72)[%(count)/%(total)] %s`. The `log:` prefix can be omitted if the format-string has a `%` in it (expecting that it is part of `%<placeholder>`). Defaults to the `format.commitListFormat` configuration variable, if set, or `shortlog`. This option given from the command-line implies the use of `--cover-letter` unless `--no-cover-letter` is given.

`--encode-email-headers`

`--no-encode-email-headers`
Encode email headers that have non-ASCII characters with "Q-encoding" (described in RFC 2047), instead of outputting the headers verbatim. Defaults to the value of the `format.encodeEmailHeaders` configuration variable.

`--interdiff=<previous>`
As a reviewer aid, insert an interdiff into the cover letter, or as commentary of the lone patch of a 1-patch series, showing the differences between the previous version of the patch series and the series currently being formatted. *<previous>* is a single revision naming the tip of the previous series which shares a common base with the series being formatted (for example `git format-patch --cover-letter --interdiff=feature/v1 -3 feature/v2`).

`--range-diff=<previous>`
As a reviewer aid, insert a range-diff (see linkgit:git-range-diff[1]) into the cover letter, or as commentary of the lone patch of a 1-patch series, showing the differences between the previous version of the patch series and the series currently being formatted. *<previous>* can be a single revision naming the tip of the previous series if it shares a common base with the series being formatted (for example `git format-patch --cover-letter --range-diff=feature/v1 -3 feature/v2`), or a revision range if the two versions of the series are disjoint (for example `git format-patch --cover-letter --range-diff=feature/v1~3..feature/v1 -3 feature/v2`).

Note that diff options passed to the command affect how the primary product of `format-patch` is generated, and they are not passed to the underlying `range-diff` machinery used to generate the cover-letter material (this may change in the future).

`--creation-factor=<percent>`
Used with `--range-diff`, tweak the heuristic which matches up commits between the previous and current series of patches by adjusting the creation/deletion cost fudge factor. See linkgit:git-range-diff[1]) for details.

Defaults to 999 (the linkgit:git-range-diff[1] uses 60), as the use case is to show comparison with an older iteration of the same topic and the tool should find more correspondence between the two sets of patches.

`--range-diff-notes=<ref>`

`--no-range-diff-notes`
Used with `--range-diff`, tweak what notes to display in the range diff.

The default behavior is to display the same notes in the range diff as on the patches; see `--notes`. Use `--range-diff-notes=<ref>` to use *<ref>* for the range diff instead. This option can be given multiple times to show notes from multiple refs. Use `--no-range-diff-notes` to disable notes in the range diff.

`--notes[=<ref>]`

`--no-notes`
Append the notes (see linkgit:git-notes[1]) for the commit after the three-dash line.

The expected use case of this is to write supporting explanation for the commit that does not belong to the commit log message proper, and include it with the patch submission. While one can simply write these explanations after `format-patch` has run but before sending, keeping them as Git notes allows them to be maintained between versions of the patch series (but see the discussion of the `notes.rewrite` configuration options in linkgit:git-notes[1] to use this workflow).

The default is `--no-notes`, unless the `format.notes` configuration is set.

`--signature=<signature>`

`--no-signature`
Add a signature to each message produced. Per RFC 3676 the signature is separated from the body by a line with "-- " on it. If the signature option is omitted the signature defaults to the Git version number.

`--signature-file=<file>`
Works just like `--signature` except the signature is read from a file.

`--suffix=.<sfx>`
Instead of using `.patch` as the suffix for generated filenames, use specified suffix. A common alternative is `--suffix=.txt`. Leaving this empty will remove the `.patch` suffix.

Note that the leading character does not have to be a dot; for example, you can use `--suffix=-patch` to get `0001-description-of-my-change-patch`.

`-q`

`--quiet`
Do not print the names of the generated files to standard output.

`--no-binary`
Do not output contents of changes in binary files, instead display a notice that those files changed. Patches generated using this option cannot be applied properly, but they are still useful for code review.

`--zero-commit`
Output an all-zero hash in each patch’s From header instead of the hash of the commit.

`--no-base`

`--base[=<commit>]`
Record the base tree information to identify the state the patch series applies to. See the BASE TREE INFORMATION section below for details. If *<commit>* is `auto`, a base commit is automatically chosen. The `--no-base` option overrides a `format.useAutoBase` configuration.

`--root`
Treat the revision argument as a *<revision-range>*, even if it is just a single commit (that would normally be treated as a *<since>*). Note that root commits included in the specified range are always formatted as creation patches, independently of this flag.

`--progress`
Show progress reports on stderr as patches are generated.

CONFIGURATION ---------- You can specify extra mail header lines to be added to each message, defaults for the subject prefix and file suffix, number patches when outputting more than one patch, add "To:" or "Cc:" headers, configure attachments, change the patch output directory, and sign off patches with configuration variables.

```
[format]
    headers = "Organization: git-foo\n"
    subjectPrefix = CHANGE
    suffix = .txt
    numbered = auto
    to = <email>
    cc = <email>
    attach [ = mime-boundary-string ]
    signOff = true
    outputDirectory = <directory>
    coverLetter = auto
    commitListFormat = shortlog
    coverFromDescription = auto
```

DISCUSSION ----------

The patch produced by `git` `format-patch` is in UNIX mailbox format, with a fixed "magic" time stamp to indicate that the file is output from format-patch rather than a real mailbox, like so:

```
From 8f72bad1baf19a53459661343e21d6491c3908d3 Mon Sep 17 00:00:00 2001
From: Tony Luck <tony.luck@intel.com>
Date: Tue, 13 Jul 2010 11:42:54 -0700
Subject: [PATCH] =?UTF-8?q?[IA64]=20Put=20ia64=20config=20files=20on=20the=20?=
 =?UTF-8?q?Uwe=20Kleine-K=C3=B6nig=20diet?=
MIME-Version: 1.0
Content-Type: text/plain; charset=UTF-8
Content-Transfer-Encoding: 8bit

arch/arm config files were slimmed down using a python script
(See commit c2330e286f68f1c408b4aa6515ba49d57f05beae comment)

Do the same for ia64 so we can have sleek & trim looking
...
```

Typically it will be placed in a MUA’s drafts folder, edited to add timely commentary that should not go in the changelog after the three dashes, and then sent as a message whose body, in our example, starts with "arch/arm config files were...". On the receiving end, readers can save interesting patches in a UNIX mailbox and apply them with linkgit:git-am[1].

When a patch is part of an ongoing discussion, the patch generated by `git` `format-patch` can be tweaked to take advantage of the `git` `am` `--scissors` feature. After your response to the discussion comes a line that consists solely of “-- >8 --” (scissors and perforation), followed by the

patch with unnecessary header fields removed:

```
...
> So we should do such-and-such.


Makes sense to me.  How about this patch?


-- >8 --
Subject: [IA64] Put ia64 config files on the Uwe Kleine-König diet


arch/arm config files were slimmed down using a python script
...
```

When sending a patch this way, most often you are sending your own patch, so in addition to the “From $SHA1 $magic_timestamp” marker you should omit `From:` and `Date:` lines from the patch file. The patch title is likely to be different from the subject of the discussion the patch is in response to, so it is likely that you would want to keep the Subject: line, like the example above.

Checking for patch corruption ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ Many mailers if not set up properly will corrupt whitespace. Here are two common types of corruption:

- Empty context lines that do not have *any* whitespace.
- Non-empty context lines that have one extra whitespace at the beginning.

One way to test if your MUA is set up correctly is:

- Send the patch to yourself, exactly the way you would, except with `To:` and `Cc:` lines that do not contain the list and maintainer address.
- Save that patch to a file in UNIX mailbox format. Call it a.patch, say.
- Apply it:

```
$ git fetch <project> master:test-apply
$ git switch test-apply
$ git restore --source=HEAD --staged --worktree :/
$ git am a.patch
```

If it does not apply correctly, there can be various reasons.

- The patch itself does not apply cleanly. That is *bad* but does not have much to do with your MUA. You might want to rebase the patch with linkgit:git-rebase[1] before regenerating it in this case.

- The MUA corrupted your patch; "am" would complain that the patch does not apply. Look in the `.git/rebase-apply/` subdirectory and see what `patch` file contains and check for the common corruption patterns mentioned above.

- While at it, check the `info` and `final-commit` files as well. If what is in `final-commit` is not exactly what you would want to see in the commit log message, it is very likely that the receiver would end up hand editing the log message when applying your patch. Things like "Hi, this is my first patch.n" in the patch e-mail should come after the three-dash line that signals the end of the commit message.

MUA-SPECIFIC HINTS ---------- Here are some hints on how to successfully submit patches inline using various mailers.

GMail ~~~~~ GMail does not have any way to turn off line wrapping in the web interface, so it will mangle any emails that you send. You can however use `git` `send-email` and send your patches through the GMail SMTP server, or use any IMAP email client to connect to the google IMAP server and forward the emails through that.

For hints on using `git` `send-email` to send your patches through the GMail SMTP server, see the EXAMPLE section of linkgit:git-send-email[1].

For hints on submission using the IMAP interface, see the EXAMPLE section of linkgit:git-imap-send[1].

Thunderbird ~~~~~~~~~~~ By default, Thunderbird will both wrap emails as well as flag them as being `format=flowed`, both of which will make the resulting email unusable by Git.

There are three different approaches: use an add-on to turn off line wraps, configure Thunderbird to not mangle patches, or use an external editor to keep Thunderbird from mangling the patches.

Approach #1 (add-on) ^^^^^^^^^^^^^^^^^^^^^^

Install the Toggle Line Wrap add-on that is available from

https://addons.thunderbird.net/thunderbird/addon/toggle-line-wrap It adds a button "Line Wrap" to the composer's toolbar that you can tick off. Now you can compose the message as you otherwise do (cut + paste, `git` `format-patch` | `git` `imap-send`, etc), but you have to insert line breaks manually in any text that you type.

As a bonus feature, the add-on can detect patch text in the composer and warns when line wrapping has not yet been turned off.

The add-on requires a few tweaks of the advanced configuration (`about:config`). These are listed on the download page.

Approach #2 (configuration) ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Three steps:

1. Configure your mail server composition as plain text: Edit...Account Settings...Composition & Addressing, uncheck "Compose Messages in HTML".

2. Configure your general composition window to not wrap.
   In Thunderbird 2: Edit..Preferences..Composition, wrap plain text messages at 0
   In Thunderbird 3: Edit..Preferences..Advanced..Config Editor. Search for "mail.wrap_long_lines". Toggle it to make sure it is set to `false`. Also, search for "mailnews.wraplength" and set the value to 0.

3. Disable the use of format=flowed: Edit..Preferences..Advanced..Config Editor. Search for "mailnews.send_plaintext_flowed". Toggle it to make sure it is set to `false`.

After that is done, you should be able to compose email as you otherwise would (cut + paste, `git format-patch` | `git imap-send`, etc), and the patches will not be mangled.

Approach #3 (external editor) ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

The following Thunderbird extensions are needed: AboutConfig from

https://mjg.github.io/AboutConfig/ and External Editor from

https://globs.org/articles.php?lng=en&pg=8

1. Prepare the patch as a text file using your method of choice.

2. Before opening a compose window, use Edit→Account Settings to uncheck the "Compose messages in HTML format" setting in the "Composition & Addressing" panel of the account to be used to send the patch.

3. In the main Thunderbird window, *before* you open the compose window for the patch, use Tools→about:config to set the following to the indicated values:

   ```
   mailnews.send_plaintext_flowed  => false
   mailnews.wraplength             => 0
   ```

4. Open a compose window and click the external editor icon.

5. In the external editor window, read in the patch file and exit the editor normally.

Side note: it may be possible to do step 2 with about:config and the following settings but no one's tried yet.

```
mail.html_compose                       => false
mail.identity.default.compose_html      => false
mail.identity.id?.compose_html          => false
```

There is a script in contrib/thunderbird-patch-inline which can help you include patches with Thunderbird in an easy way. To use it, do the steps above and then use the script as the external editor.

KMail ~~~~~ This should help you to submit patches inline using KMail.

1. Prepare the patch as a text file.

2. Click on New Mail.

3. Go under "Options" in the Composer window and be sure that "Word wrap" is not set.

4. Use Message → Insert file… and insert the patch.

5. Back in the compose window: add whatever other text you wish to the message, complete the addressing and subject fields, and press send.

BASE TREE INFORMATION ----------

The base tree information block is used for maintainers or third party testers to know the exact state the patch series applies to. It consists of the "base commit", which is a well-known commit that is part of the stable part of the project history everybody else works off of, and zero or more "prerequisite patches", which are well-known patches in flight that is not yet part of the "base commit" that need to be applied on top of "base commit" in topological order before the patches can be applied.

The "base commit" is shown as "`base-commit:` " followed by the 40-hex of the commit object name. A "prerequisite patch" is shown as "`prerequisite-patch-id:` " followed by the 40-hex "patch id", which can be obtained by passing the patch through the `git` `patch-id` `--stable` command.

Imagine that on top of the public commit P, you applied well-known patches X, Y and Z from somebody else, and then built your three-patch series A, B, C, the history would be like:

```
---P---X---Y---Z---A---B---C
```

With `git` `format-patch` `--base=P` `-3` `C` (or variants thereof, e.g. with `--cover-letter` or using `Z..C` instead of `-3` `C` to specify the range), the base tree information block is shown at the end of the first message the command outputs (either the first patch, or the cover letter), like this:

```
base-commit: P
prerequisite-patch-id: X
prerequisite-patch-id: Y
prerequisite-patch-id: Z
```

For non-linear topology, such as

```
---P---X---A---M---C
    \         /
     Y---Z---B
```

You can also use `git` `format-patch` `--base=P` `-3` `C` to generate patches for A, B and C, and the identifiers for P, X, Y, Z are appended at the end of the first message.

If set `--base=auto` in cmdline, it will automatically compute the base commit as the merge base of tip commit of the remote-tracking branch and revision-range specified in cmdline. For a local branch, you need to make it to track a remote branch by `git` `branch` `--set-upstream-to` before using this option.

EXAMPLES --------

- Extract commits between revisions R1 and R2, and apply them on top of the current branch using `git` `am` to cherry-pick them:

  ```
  $ git format-patch -k --stdout R1..R2 | git am -3 -k
  ```

- Extract all commits which are in the current branch but not in the origin branch:

  ```
  $ git format-patch origin
  ```

  For each commit a separate file is created in the current directory.

- Extract all commits that lead to `origin` since the inception of the project:

  ```
  $ git format-patch --root origin
  ```

- The same as the previous one:

  ```
  $ git format-patch -M -B origin
  ```

  Additionally, it detects and handles renames and complete rewrites intelligently to produce a renaming patch. A renaming patch reduces the amount of text output, and generally makes it easier to review. Note that non-Git `patch` programs won't understand renaming patches, so use it only when you know the recipient uses Git to apply your patch.

- Extract three topmost commits from the current branch and format them as e-mailable patches:

  ```
  $ git format-patch -3
  ```

CAVEATS -------

Note that `format-patch` will omit merge commits from the output, even if they are part of the requested range. A simple "patch" does not include enough information for the receiving end to reproduce the same merge commit.

PATCH APPLICATION

The output from linkgit:git-format-patch[1] can lead to a different commit message when applied with linkgit:git-am[1]. The patch that is applied may also be different from the one that was generated, or patch application may fail outright. ifdef::git-am[] See the DISCUSSION section above for the syntactic rules. endif::git-am[]

ifndef::git-am[] include::format-patch-end-of-commit-message.adoc[] endif::git-am[]

Note that this is especially problematic for unindented diffs that occur in the commit message; the diff in the commit message might get applied along with the patch section, or the patch application machinery might trip up because the patch target doesn’t apply. This could for example be caused by a diff in a Markdown code block.

The solution for this is to indent the diff or other text that could cause problems.

This loss of fidelity might be simple to notice if you are applying patches directly from a mailbox. However, changes originating from Git could be applied in bulk, in which case this would be much harder to notice. This could for example be a Linux distribution which uses patch files to apply changes on top of the commits from the upstream repositories. This goes to show that this behavior does not only impact email workflows.

Given these limitations, one might be tempted to use a general-purpose utility like `patch`(1) instead. However, `patch`(1) will not only look for unindented diffs (like linkgit:git-am[1]) but will try to apply indented diffs as well.

SEE ALSO -------- linkgit:git-am[1], linkgit:git-send-email[1]

GIT --- Part of the linkgit:git[1] suite

□□□ #1
□ □□□ □□□ 2026-10-31 20:10:59 UTC
□ □□□ □□□ 2026-10-31 20:10:59 UTC