git-revise

Release 0.6.0

Jun 07, 2020

Contents:

`git-revise` is a *`git(1)`* subcommand, and *`python(1)`* library for efficiently updating, splitting, and rearranging commits.

`git revise` is open-source, and can be found on GitHub

CHAPTER 1

Installing

git-revise can be installed from PyPi. Python 3.6 or higher is required.

```
$ pip install --user git-revise
```

CHAPTER 2

`git-revise(1)` – Efficiently update, split, and rearrange git commits

2.1 SYNOPSIS

git revise [<options>] [<target>]

2.2 DESCRIPTION

`git revise` is a *`git(1)`* subcommand to efficiently update, split, and rearrange commits. It is heavily inspired by *`git-rebase(1)`*, however tries to be more efficient and ergonomic for patch-stack oriented workflows.

By default, **`git revise`** will apply staged changes to <target>, updating `HEAD` to point at the revised history. It also supports splitting commits, rewording commit messages.

Unlike *`git-rebase(1)`*, **`git revise`** avoids modifying working directory and index state, performing all merges in-memory, and only writing them when necessary. This allows it to be significantly faster on large codebases, and avoid invalidating builds.

If *`--autosquash`* or *`--interactive`* is specified, the <target> argument is optional. If it is omitted, **`git revise`** will consider a range of unpublished commits on the current branch.

2.3 OPTIONS

2.3.1 General options

`-a, --all`
: Stage changes to tracked files before revising.

`-p, --patch`
: Interactively stage hunks from the worktree before revising.

--no-index
Ignore staged changes in the index.

--reauthor
Reset target commit's author to the current user.

--ref <gitref>
Working branch to update; defaults to HEAD.

2.3.2 Main modes of operation

-i, --interactive
Rather than applying staged changes to <target>, edit a todo list of actions to perform on commits after <target>. See *INTERACTIVE MODE*.

--autosquash, --no-autosquash
Rather than directly applying staged changes to <target>, automatically perform fixup or squash actions marked with `fixup!` or `squash!` between <target> and the current HEAD. For more information on what these actions do, see *INTERACTIVE MODE*.

These commits are usually created with `git commit --fixup=<commit>` or `git commit --squash=<commit>`, and identify the target with the first line of its commit message.

This option can be combined with *--interactive* to modify the generated todos before they're executed.

If the *--autosquash* option is enabled by default using a configuration variable, the option *--no-autosquash* can be used to override and disable this setting. See *CONFIGURATION*.

-c, --cut
Interactively select hunks from <target>. The chosen hunks are split into a second commit immediately after the target.

After splitting is complete, both commits' messages are edited.

See the "Interactive Mode" section of *git-add(1)* to learn how to operate this mode.

-e, --edit
After applying staged changes, edit <target>'s commit message.

This option can be combined with *--interactive* to allow editing of commit messages within the todo list. For more information on, see *INTERACTIVE MODE*.

-m <msg>**, --message** <msg>
Use the given <msg> as the new commit message for <target>. If multiple *-m* options are given, their values are concatenated as separate paragraphs.

--version
Print version information and exit.

2.4 CONFIGURATION

Configuration is managed by *git-config(1)*.

revise.autoSquash
If set to true, imply *--autosquash* whenever *--interactive* is specified. Overridden by *--no-autosquash*. Defaults to false. If not set, the value of `rebase.autoSquash` is used instead.

2.5 CONFLICT RESOLUTION

When a conflict is encountered, **git revise** will attempt to resolve it automatically using standard git mechanisms. If automatic resolution fails, the user will be prompted to resolve them manually.

There is currently no support for using *git-mergetool(1)* to resolve conflicts.

No attempt is made to detect renames of files or directories. **git revise** may produce suboptimal results across renames. Use the interactive mode of *git-rebase(1)* when rename tracking is important.

2.6 NOTES

A successful **git revise** will add a single entry to the reflog, allowing it to be undone with `git reset @{1}`. Unsuccessful **git revise** commands will leave your repository largely unmodified.

No merge commits may occur between the target commit and `HEAD`, as rewriting them is not supported.

See *git-rebase(1)* for more information on the implications of modifying history on a repository that you share.

2.7 INTERACTIVE MODE

git revise supports an interactive mode inspired by the interactive mode of *git-rebase(1)*.

This mode is started with the last commit you want to retain "as-is":

```
git revise -i <after-this-commit>
```

An editor will be fired up with the commits in your current branch after the given commit. If the index has any staged but uncommitted changes, a `<git index>` entry will also be present.

```
pick 8338dfa88912 Oneline summary of first commit
pick 735609912343 Summary of second commit
index 672841329981 <git index>
```

These commits may be re-ordered to change the order they appear in history. In addition, the `pick` and `index` commands may be replaced to modify their behaviour. If present, `index` commands must be at the bottom of the list, i.e. they can not be followed by non-index commands.

If `-e` was specified, the full commit message will be included, and each command line will begin with a `++`. Any changes made to the commit messages in this file will be applied to the commit in question, allowing for simultaneous editing of commit messages during the todo editing phase.

```
++ pick 8338dfa88912
Oneline summary of first commit

Body of first commit

++ pick 735609912343
Summary of second commit

Body of second commit

++ index 672841329981
<git index>
```

The following commands are supported in all interactive modes:

index
: Do not commit these changes, instead leaving them staged in the index. Index lines must come last in the file.

pick
: Use the given commit as-is in history. When applied to the generated `index` entry, the commit will have the message `<git index>`.

squash
: Add the commit's changes into the previous commit and open an editor to merge the commits' messages.

fixup
: Like squash, but discard this commit's message rather than editing.

reword
: Open an editor to modify the commit message.

cut
: Interactively select hunks from the commit. The chosen hunks are split into a second commit immediately after it.

  After splitting is complete, both commits' messages are edited.

  See the "Interactive Mode" section of *git-add(1)* to learn how to operate this mode.

2.8 REPORTING BUGS

Please report issues and feature requests to the issue tracker at https://github.com/mystor/git-revise/issues.

Code, documentation and other contributions are also welcomed.

2.9 SEE ALSO

git(1) *git-rebase(1)* *git-add(1)*

CHAPTER 3

Performance

Note: These numbers are from an earlier version, and may not reflect the current state of *git-revise*.

With large repositories such as `mozilla-central`, **`git revise`** is often significantly faster than *`git-rebase(1)`* for incremental, due to not needing to update the index or working directory during rebases.

I did a simple test, applying a single-line change to a commit 11 patches up the stack. The following are my extremely non-scientific time measurements:

Command	Real Time
`git rebase -i --autosquash`	16.931s
`git revise`	0.541s

The following are the commands I ran:

```
# Apply changes with git rebase -i --autosquash
$ git reset 6fceb7da316d && git add .
$ time bash -c 'TARGET=14f1c85bf60d; git commit --fixup=$TARGET; EDITOR=true git␣
↪rebase -i --autosquash $TARGET~'
<snip>

real    0m16.931s
user    0m15.289s
sys     0m3.579s

# Apply changes with git revise
$ git reset 6fceb7da316d && git add .
$ time git revise 14f1c85bf60d
<snip>

real    0m0.541s
user    0m0.354s
sys     0m0.150s
```

3.1 How is it faster?

In-Memory Cache

To avoid spawning unnecessary subprocesses and hitting disk too frequently, **`git revise`** uses an in-memory cache of objects in the ODB which it has already seen.

Intermediate git trees, blobs, and commits created during processing are helds exclusively in-memory, and only persisted when necessary.

Custom Merge Algorithm

A custom implementation of the merge algorithm is used which directly merges trees rather than using the index. This ends up being faster on large repositories, as only the subset of modified files and directories need to be examined when merging.

Note: Currently this algorithm is incapable of handling copy and rename operations correctly, instead treating them as file creation and deletion actions. This may be resolveable in the future.

Avoiding Index & Working Directory

The working directory and index are never examined or updated during the rebasing process, avoiding disk I/O and invalidating existing builds.

CHAPTER 4

The `gitrevise` module

Python modules for interacting with git objects used by *git-revise(1) – Efficiently update, split, and rearrange git commits*.

4.1 `merge` – Quick in-memory merges

This module contains a basic implementation of an efficient, in-memory 3-way git tree merge. This is used rather than traditional git mechanisms to avoid needing to use the index file format, which can be slow to initialize for large repositories.

The INDEX file for my local mozilla-central checkout, for reference, is 35MB. While this isn't huge, it takes a perceptable amount of time to read the tree files and generate. This algorithm, on the other hand, avoids looking at unmodified trees and blobs when possible.

exception gitrevise.merge.**MergeConflict**

4.2 `odb` – Git object database interaction

Helper classes for reading cached objects from Git's Object Database.

class gitrevise.odb.**Blob**
In memory representation of a git `blob` object

class gitrevise.odb.**Commit**
In memory representation of a git `commit` object

author
Signature of this commit's author

committer
Signature of this commit's committer

message
Body of this commit's message

parent() → gitrevise.odb.Commit
Helper method to get the single parent of a commit. Raises `ValueError` if the incorrect number of parents are present.

parent_oids
List of *Oid* for this commit's parents

parents() → Sequence[gitrevise.odb.Commit]
List of parent commits

rebase(*parent: gitrevise.odb.Commit*) → gitrevise.odb.Commit
Create a new commit with the same changes, except with `parent` as it's parent.

summary() → str
The summary line (first line) of the commit message

tree() → gitrevise.odb.Tree
`tree` object corresponding to this commit

tree_oid
Oid of this commit's `tree` object

update(*tree: Optional[Tree] = None*, *parents: Optional[Sequence[Commit]] = None*, *message: Optional[bytes] = None*, *author: Optional[gitrevise.odb.Signature] = None*) → gitrevise.odb.Commit
Create a new commit with specific properties updated or replaced

class gitrevise.odb.**Entry**(*repo: gitrevise.odb.Repository*, *mode: gitrevise.odb.Mode*, *oid: gitrevise.odb.Oid*)
In memory representation of a single `tree` entry

blob() → gitrevise.odb.Blob
Get the data for this entry as a *Blob*

mode
Mode of the entry

oid
Oid of this entry's object

persist() → None
GitObj.persist() the git object referenced by this entry

repo
Repository this entry originates from

symlink() → bytes
Get the data for this entry as a symlink

tree() → gitrevise.odb.Tree
Get the data for this entry as a *Tree*

class gitrevise.odb.**GitObj**
In-memory representation of a git object. Instances of this object should be one of *Commit*, *Tree* or *Blob*

body
Raw body of object in bytes

oid
Oid of this git object

persist() → gitrevise.odb.Oid
If this object has not been persisted to disk yet, persist it

persisted
If `True`, the object has been persisted to disk

repo
Repository object is associated with

class gitrevise.odb.**Index**(*repo: gitrevise.odb.Repository*, *index_file: Optional[pathlib.Path] = None*)
Handle on an index file

commit(*message: bytes = b'<git index>'*, *parent: Optional[gitrevise.odb.Commit] = None*) → gitrevise.odb.Commit
Get a *Commit* for this index's state. If `parent` is `None`, use the current `HEAD`

git(**cmd*, *cwd: Optional[pathlib.Path] = None*, *stdin: Optional[bytes] = None*, *newline: bool = True*, *env: Optional[Mapping[str, str]] = None*, *nocapture: bool = False*) → bytes
Invoke git with the given index as active

index_file = None
Index file being referenced

repo = None

tree() → gitrevise.odb.Tree
Get a *Tree* object for this index's state

exception gitrevise.odb.**MissingObject**(*ref: str*)
Exception raised when a commit cannot be found in the ODB

class gitrevise.odb.**Mode**
Mode for an entry in a `tree`

DIR = b'40000'
directory entry

EXEC = b'100755'
executable entry

GITLINK = b'160000'
submodule entry

REGULAR = b'100644'
regular entry

SYMLINK = b'120000'
symlink entry

class gitrevise.odb.**Oid**
Git object identifier

classmethod for_object(*tag: str*, *body: bytes*)
Hash an object with the given type tag and body to determine its Oid

classmethod fromhex(*instr: str*) → gitrevise.odb.Oid
Parse an `Oid` from a hexadecimal string

classmethod null() → gitrevise.odb.Oid
An `Oid` consisting of entirely 0s

short() → str
A shortened version of the Oid's hexadecimal form

class gitrevise.odb.**Reference**(*obj_type: Type[GitObjT], repo: gitrevise.odb.Repository, name: str*)

A git reference

name = None
Resolved reference name, e.g. 'refs/tags/1.0.0' or 'refs/heads/master'

refresh()
Re-read the target of this reference from disk

repo = None
Repository reference is attached to

shortname = None
Short unresolved reference name, e.g. 'HEAD' or 'master'

target = None
Referenced git object

update(*new: GitObjT*, *reason: str*)
Update this refreence to point to a new object. An entry with the reason `reason` will be added to the reflog.

class gitrevise.odb.**Repository**(*cwd: Optional[pathlib.Path] = None*)

Main entry point for a git repository

default_author
author used by default for new commits

default_committer
committer used by default for new commits

get_blob(*ref: Union[gitrevise.odb.Oid, str]*) → gitrevise.odb.Blob
Like *get_obj()*, but returns a *Blob*

get_blob_ref(*ref: str*) → gitrevise.odb.Reference[gitrevise.odb.Blob]
Get a *Reference* to a *Blob*

get_commit(*ref: Union[gitrevise.odb.Oid, str]*) → gitrevise.odb.Commit
Like *get_obj()*, but returns a *Commit*

get_commit_ref(*ref: str*) → gitrevise.odb.Reference[gitrevise.odb.Commit]
Get a *Reference* to a *Commit*

get_obj(*ref: Union[gitrevise.odb.Oid, str]*) → gitrevise.odb.GitObj
Get the identified git object from this repository. If given an *Oid*, the cache will be checked before asking git.

get_obj_ref(*ref: str*) → gitrevise.odb.Reference[gitrevise.odb.GitObj]
Get a *Reference* to a *GitObj*

get_tempdir() → pathlib.Path
Return a temporary directory to use for modifications to this repository

get_tree(*ref: Union[gitrevise.odb.Oid, str]*) → gitrevise.odb.Tree
Like *get_obj()*, but returns a *Tree*

get_tree_ref(*ref: str*) → gitrevise.odb.Reference[gitrevise.odb.Tree]
Get a *Reference* to a *Tree*

git_path(*path: Union[str, pathlib.Path]*) → pathlib.Path
Get the path to a file in the .git directory, respecting the environment

gitdir
.git directory for this repository

index
current index state

new_commit(*tree: gitrevise.odb.Tree, parents: Sequence[Commit], message: bytes, author: Optional[gitrevise.odb.Signature] = None, committer: Optional[gitrevise.odb.Signature] = None*) → gitrevise.odb.Commit
Directly create an in-memory commit object, without persisting it. If a commit object with these properties already exists, it will be returned instead.

new_tree(*entries: Mapping[bytes, Entry]*) → gitrevise.odb.Tree
Directly create an in-memory tree object, without persisting it. If a tree object with these entries already exists, it will be returned instead.

workdir
working directory for this repository

class gitrevise.odb.**Signature**
Git user signature

email
user email

name
user name

offset
timezone offset from UTC

timestamp
unix timestamp

class gitrevise.odb.**Tree**
In memory representation of a git `tree` object

entries
mapping from entry names to entry objects in this tree

to_index(*path: pathlib.Path*, *skip_worktree: bool = False*) → gitrevise.odb.Index
Read tree into a temporary index. If skip_workdir is `True`, every entry in the index will have its "Skip Workdir" bit set.

4.3 `todo` – History edit sequences

class gitrevise.todo.**StepKind**
An enumeration.

gitrevise.todo.**validate_todos**(*old: List[gitrevise.todo.Step], new: List[gitrevise.todo.Step]*)
Raise an exception if the new todo list is malformed compared to the original todo list

4.4 `tui` – `git-revise` entry point

4.5 `utils` – Misc. helper methods

exception gitrevise.utils.**EditorError**

gitrevise.utils.**commit_range**(*base: gitrevise.odb.Commit*, *tip: gitrevise.odb.Commit*) → List[gitrevise.odb.Commit]

Oldest-first iterator over the given commit range, not including the commit `base`

gitrevise.utils.**cut_commit**(*commit: gitrevise.odb.Commit*) → gitrevise.odb.Commit

Perform a `cut` operation on the given commit, and return the modified commit.

gitrevise.utils.**edit_commit_message**(*commit: gitrevise.odb.Commit*) → gitrevise.odb.Commit

Launch an editor to edit the commit message of `commit`, returning a modified commit

gitrevise.utils.**local_commits**(*repo: gitrevise.odb.Repository*, *tip: gitrevise.odb.Commit*) → Tuple[gitrevise.odb.Commit, List[gitrevise.odb.Commit]]

Returns an oldest-first iterator over the local commits which are parents of the specified commit. May return an empty list. A commit is considered local if it is not present on any remote.

gitrevise.utils.**run_editor**(*repo: gitrevise.odb.Repository*, *filename: str*, *text: bytes*, *comments: Optional[str] = None*, *allow_empty: bool = False*) → bytes

Run the editor configured for git to edit the given text

gitrevise.utils.**run_sequence_editor**(*repo: gitrevise.odb.Repository*, *filename: str*, *text: bytes*, *comments: Optional[str] = None*, *allow_empty: bool = False*) → bytes

Run the editor configured for git to edit the given rebase/revise sequence

gitrevise.utils.**run_specific_editor**(*editor: str*, *repo: gitrevise.odb.Repository*, *filename: str*, *text: bytes*, *comments: Optional[str] = None*, *allow_empty: bool = False*, *allow_whitespace_before_comments: bool = False*) → bytes

Run the editor configured for git to edit the given text

CHAPTER 5

Contributing

5.1 Running Tests

tox is used to run tests. It will run **mypy** for type checking, **pylint** for linting, **pytest** for testing, and **black** for code formatting.

```
$ tox          # All python versions
$ tox -e py36  # Python 3.6
$ tox -e py37  # Python 3.7
```

5.2 Code Formatting

This project uses `black` for code formatting.

```
$ black .  # format all python code
```

5.3 Building Documentation

Documentation is built using **sphinx**.

```
$ cd docs/
$ make man  # Build manpage
```

5.4 Publishing

```
$ python3 setup.py sdist bdist_wheel
$ twine check dist/*
$ twine upload dist/*
```

Python Module Index

g

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W