contribute

- Becoming a contributor
  - Overview
  - Step 0: Select a Google Account
  - Step 1: Contributor License Agreement
  - Step 2: Configure git authentication
  - Step 3: Create a Gerrit account
  - Step 4: Install the git-codereview command
- Before contributing code
  - Where to contribute
  - Check the issue tracker
  - Open an issue for any new problem
- Sending a change via Gerrit
  - Overview
  - Step 1: Clone the source code
  - Step 2: Prepare changes in a new branch
  - Step 3: Test your changes
    - In the main Go repository
    - In the golang.org/x/... repositories
  - Step 4: Send changes for review
  - Step 5: Revise changes after a review
- Sending a change via GitHub
- Good commit messages
  - First line
  - Main content
  - Referencing issues
- The review process
  - Common beginner mistakes
  - Trybots

    - Reviews
    - Voting conventions
    - Submitting an approved change
    - More information
- Miscellaneous topics
    - Gopls
    - Copyright headers
    - Troubleshooting mail errors
    - Quickly testing your changes
    - Specifying a reviewer / CCing others
    - Synchronize your client
    - Reviewing code by others
    - Set up git aliases
    - Sending multiple dependent changes
    - Minor Releases

The Go project welcomes all contributors.

This document is a guide to help you through the process of contributing to the Go project, which is a little different from that used by other open source projects. We assume you have a basic understanding of Git and Go.

In addition to the information here, the Go community maintains a CodeReview wiki page. Feel free to contribute to the wiki as you learn the review process.

Note that the `gccgo` front end lives elsewhere; see Contributing to gccgo.

Becoming a contributor

Overview

The first step is registering as a Go contributor and configuring your environment. Here is a checklist of the required steps to follow:

- **Step 0**: Decide on a single Google Account you will be using to contribute to Go. Use that account for all the following steps and make sure that `git` is configured to create

commits with that account's e-mail address.

- **Step 1**: Sign and submit a CLA (Contributor License Agreement).
- **Step 2**: Configure authentication credentials for the Go Git repository. Visit go.googlesource.com, click "Generate Password" in the page's top right menu bar, and follow the instructions.
- **Step 3**: Register for Gerrit, the code review tool used by the Go team, by visiting this page. The CLA and the registration need to be done only once for your account.
- **Step 4**: Install `git-codereview` by running `go install golang.org/x/review/git-codereview@latest`

If you prefer, there is an automated tool that walks through these steps. Just run:

```
$ go install golang.org/x/tools/cmd/go-contrib-init@latest
$ cd /code/to/edit
$ go-contrib-init\
```

The rest of this chapter elaborates on these instructions. If you have completed the steps above (either manually or through the tool), jump to Before contributing code.

Step 0: Select a Google Account

A contribution to Go is made through a Google account with a specific e-mail address. Make sure to use the same account throughout the process and for all your subsequent contributions. You may need to decide whether to use a personal address or a corporate address. The choice will depend on who will own the copyright for the code that you will be writing and submitting. You might want to discuss this topic with your employer before deciding which account to use.

Google accounts can either be Gmail e-mail accounts, G Suite organization accounts, or accounts associated with an external e-mail address. For instance, if you need to use an existing corporate e-mail that is not managed through G Suite, you can create an account associated with your existing e-mail address.

You also need to make sure that your Git tool is configured to create commits using your chosen e-mail address. You can either configure Git globally (as a default for all projects), or locally (for a single specific project). You can check the current configuration with this command:

```
$ git config --global user.email  # check current global config
$ git config user.email           # check current local config\
```

To change the configured address:

```
$ git config --global user.email name@example.com   # change global config
$ git config user.email name@example.com            # change local config\
```

Step 1: Contributor License Agreement

Before sending your first change to the Go project you must have completed one of the following two CLAs. Which CLA you should sign depends on who owns the copyright to your work.

- If you are the copyright holder, you will need to agree to the individual contributor license agreement, which can be completed online.
- If your organization is the copyright holder, the organization will need to agree to the corporate contributor license agreement.\

You can check your currently signed agreements and sign new ones at the Google Developers Contributor License Agreements website. If the copyright holder for your contribution has already completed the agreement in connection with another Google open source project, it does not need to be completed again.

If the copyright holder for the code you are submitting changes—for example, if you start contributing code on behalf of a new company—please send mail to the `golang-dev` mailing list. This will let us know the situation so we can make sure an appropriate agreement is completed.

Step 2: Configure git authentication

The main Go repository is located at go.googlesource.com, a Git server hosted by Google. Authentication on the web server is made through your Google account, but you also need to configure `git` on your computer to access it. Follow these steps:

1. Visit go.googlesource.com and click on "Generate Password" in the page's top right menu bar. You will be redirected to accounts.google.com to sign in.
2. After signing in, you will be taken to a page with the heading "Configure Git". This page contains a personalized script that when run locally will configure Git to hold your unique authentication key. This key is paired with one that is generated and stored on the server, analogous to how SSH keys work.
3. Copy and run this script locally in your terminal to store your secret authentication token in a `.gitcookies` file. If you are using a Windows computer and running `cmd`, you should instead follow the instructions in the yellow box to run the command; otherwise run the regular script.

Step 3: Create a Gerrit account

Gerrit is an open-source tool used by Go maintainers to discuss and review code submissions.

To register your account, visit go-review.googlesource.com/login/ and sign in once using the same Google Account you used above.

Step 4: Install the git-codereview command

Changes to Go must be reviewed before they are accepted, no matter who makes the change. A custom `git` command called `git-codereview` simplifies sending changes to Gerrit.

Install the `git-codereview` command by running,

$ go install golang.org/x/review/git-codereview@latest\

Make sure `git-codereview` is installed in your shell path, so that the `git` command can find it. Check that

$ git codereview help\

prints help text, not an error. If it prints an error, make sure that `$GOPATH/bin` is in your `$PATH`.

On Windows, when using git-bash you must make sure that `git-codereview.exe` is in your `git` exec-path. Run `git --exec-path` to discover the right location then create a symbolic link or just copy the executable from `$GOPATH/bin` to this directory.

Before contributing code

The project welcomes code patches, but to make sure things are well coordinated you should discuss any significant change before starting the work. It's recommended that you signal your intention to contribute in the issue tracker, either by filing a new issue or by claiming an existing one.

Where to contribute

The Go project consists of the main go repository, which contains the source code for the Go language, as well as many golang.org/x/... repositories. These contain the various tools and infrastructure that support Go. For example, golang.org/x/pkgsite is for pkg.go.dev, golang.org/x/playground is for the Go playground, and golang.org/x/tools contains a variety of Go tools, including the Go language server, gopls. You can see a list of all the golang.org/x/... repositories on go.googlesource.com.

Check the issue tracker

Whether you already know what contribution to make, or you are searching for an idea, the issue tracker is always the first place to go. Issues are triaged to categorize them and manage the workflow.

The majority of the golang.org/x/... repos also use the main Go issue tracker. However, a few of these repositories manage their issues separately, so please be sure to check the right tracker for the repository to which you would like to contribute.

Most issues will be marked with one of the following workflow labels:

- **NeedsInvestigation**: The issue is not fully understood and requires analysis to understand the root cause.
- **NeedsDecision**: the issue is relatively well understood, but the Go team hasn't yet decided the best way to address it. It would be better to wait for a decision before writing code. If you are interested in working on an issue in this state, feel free to "ping" maintainers in the issue's comments if some time has passed without a decision.
- **NeedsFix**: the issue is fully understood and code can be written to fix it.

You can use GitHub's search functionality to find issues to help out with. Examples:

- Issues that need investigation: `is:issue is:open label:NeedsInvestigation`
- Issues that need a fix: `is:issue is:open label:NeedsFix`
- Issues that need a fix and have a suggested change: `is:issue is:open label:NeedsFix ("golang.org/cl" OR "go.dev/cl")`
- Issues that need a fix and do not have a suggested change: `is:issue is:open label:NeedsFix NOT "golang.org/cl" NOT "go.dev/cl"`

Open an issue for any new problem

Excluding very trivial changes, all contributions should be connected to an existing issue. Feel free to open one and discuss your plans. This process gives everyone a chance to validate the design, helps prevent duplication of effort, and ensures that the idea fits inside the goals for the language and tools. It also checks that the design is sound before code is written; the code review tool is not

the place for high-level discussions.

When planning work, please note that the Go project follows a six-month development cycle for the main Go repository. The latter half of each cycle is a three-month feature freeze during which only bug fixes and documentation updates are accepted. New contributions can be sent during a feature freeze, but they will not be merged until the freeze is over. The freeze applies to the entire main repository as well as to the code in golang.org/x/... repositories that is needed to build the binaries included in the release. See the lists of packages vendored into the standard library and the `go` command.

Significant changes to the language, libraries, or tools (which includes API changes in the main repo and all golang.org/x repos, as well as command-line changes to the `go` command) must go through the change proposal process before they can be accepted.

Sensitive security-related issues (only!) should be reported to security@golang.org.

Sending a change via Gerrit

We encourage using Gerrit to send changes for review. While there is a bit of a learning curve for users who have only used the GitHub workflow, it is the primary workflow used by contributors to the Go project and is a smoother and more full-featured workflow.

The following sections give a brief walkthrough of submitting a change via Gerrit. For more details on interacting with Gerrit, refer to Gerrit's documentation. In particular, we recommend reading the Review UI Overview and Basic Gerrit Walkthrough — For GitHub Users pages.

Overview

This is an overview of the overall process:

- **Step 1:** Clone the source code from `go.googlesource.com` and make sure it's stable by compiling and testing it once.

  If you're making a change to the main Go repository:

  ```
  $ git clone https://go.googlesource.com/go
  $ cd go/src
  $ ./all.bash                    # compile and test\
  ```

  If you're making a change to one of the golang.org/x/... repositories (golang.org/x/tools, in this example):

$ git clone https://go.googlesource.com/tools
$ cd tools
$ go test ./... # compile and test\

- **Step 2:** Prepare changes in a new branch, created from the master branch. To commit the changes, use `git codereview change`; that will create or amend a single commit in the branch.
  $ git checkout -b mybranch
  $ [edit files...]
  $ git add [files...]
  $ git codereview change # create commit in the branch
  $ [edit again...]
  $ git add [files...]
  $ git codereview change # amend the existing commit with new changes
  $ [etc.]\
- **Step 3:** Test your changes, either by running the tests in the package you edited or by re-running `all.bash`.
  In the main Go repository:

  $ ./all.bash # recompile and test\
  In a golang.org/x/... repository:

  $ go test ./... # recompile and test\
- **Step 4:** Send the changes for review to Gerrit using `git codereview mail` (which doesn't use e-mail, despite the name).
  $ git codereview mail # send changes to Gerrit\
- **Step 5:** After a review, apply changes to the same single commit and mail them to Gerrit again:
  $ [edit files...]
  $ git add [files...]
  $ git codereview change # update same commit
  $ git codereview mail # send to Gerrit again\

The rest of this section describes these steps in more detail.

Step 1: Clone the source code

In addition to a recent Go installation, you need to have a local copy of the source checked out from the correct repository. You can check out the Go source repo onto your local file system anywhere you want as long as it's outside your `GOPATH` (defaults to the directory `go` in your home directory). Clone from `go.googlesource.com` (not GitHub):

Main Go repository:

```
$ git clone https://go.googlesource.com/go
$ cd go\
```

golang.org/x/... repository

(golang.org/x/tools in this example):

```
$ git clone https://go.googlesource.com/tools
$ cd tools\
```

Step 2: Prepare changes in a new branch

Each Go change must be made in a separate branch, created from the master branch. You can use the normal `git` commands to create a branch and add changes to the staging area:

```
$ git checkout -b mybranch
$ [edit files...]
$ git add [files...]\
```

To commit changes, instead of `git commit`, use `git codereview change`.

```
$ git codereview change
(open $EDITOR)\
```

You can edit the commit description in your favorite editor as usual. The `git codereview change` command will automatically add a unique Change-Id line near the bottom. That line is used by Gerrit to match successive uploads of the same change. Do not edit or delete it. A Change-Id looks like this:

```
Change-Id: I2fbdbffb3aab626c4b6f56348861b7909e3e8990\
```

The tool also checks that you've run `go fmt` over the source code, and that the commit message follows the suggested format.

If you need to edit the files again, you can stage the new changes and re-run `git codereview change`: each subsequent run will amend the existing commit while preserving the Change-Id.

Make sure that you always keep a single commit in each branch. If you add more commits by mistake, you can use `git rebase` to squash them together into a single one.

Step 3: Test your changes

You've written and tested your code, but before sending code out for review, run *all the tests for the whole tree* to make sure the changes don't break other packages or programs.

In the main Go repository

For standard library packages, all tests within the package must pass:

```
$ go test\
```

The short test suite for the entire tree can be run with `all.bash` (To build under Windows use `all.bat`):

```
$ cd go/src
$ ./all.bash\
```

After running for a while and printing a lot of testing output, the command should finish by printing,

```
ALL TESTS PASSED\
```

You can use `make.bash` instead of `all.bash` to just build the compiler and the standard library without running the test suite. Once the `go` tool is built, it will be installed as `bin/go` under the directory in which you cloned the Go repository, and you can run it directly from there. See also the section on how to test your changes quickly.

In the golang.org/x/... repositories

Run the tests for the entire repository (golang.org/x/tools, in this example):

```
$ cd tools
$ go test ./...\
```

If you're concerned about the build status, you can check the Build Dashboard. Test failures may also be caught by the TryBots in code review.

Some repositories, like golang.org/x/vscode-go will have different testing infrastructures, so always check the documentation for the repository in which you are working. The README file in the root of the repository will usually have this information.

Step 4: Send changes for review

Once the change is ready and tested over the whole tree, send it for review. This is done with the `mail` sub-command which, despite its name, doesn't directly mail anything; it just sends the change to Gerrit:

```
$ git codereview mail\
```

Gerrit assigns your change a number and URL, which `git` `codereview` `mail` will print, something like:

```
remote: New Changes:
remote:   https://go-
review.googlesource.com/99999 math: improved Sin, Cos and Tan precision for very large argumen
ts\
```

If you get an error instead, check the Troubleshooting mail errors section.

If your change relates to an open GitHub issue and you have followed the suggested commit message format, the issue will be updated in a few minutes by a bot, linking your Gerrit change to it in the comments.

Step 5: Revise changes after a review

Go maintainers will review your code on Gerrit, and you will get notifications via e-mail. You can see the review on Gerrit and comment on them there. You can also reply using e-mail if you prefer.

If you need to revise your change after the review, edit the files in the same branch you previously created, add them to the Git staging area, and then amend the commit with `git` `codereview` `change`:

```
$ git codereview change     # amend current commit
(open $EDITOR)
$ git codereview mail       # send new changes to Gerrit\
```

If you don't need to change the commit description, just save and exit from the editor. Remember not to touch the special Change-Id line.

Again, make sure that you always keep a single commit in each branch. If you add more commits by mistake, you can use `git rebase` to squash them together into a single one.

Sending a change via GitHub

Although the Gerrit flow is recommended and better supported, contributors that are already familiar with the GitHub flow are able to use the same process for Go contributions. Even though Go maintainers use Gerrit for code review, a tool called GerritBot has been created to sync GitHub pull requests to Gerrit.

Open a GitHub pull request as you normally would. GerritBot will create a corresponding Gerrit change list (a "CL") and post a link to it on your GitHub pull request; updates to the pull request will also get reflected in the Gerrit CL. When somebody comments on the CL, their comment will be also posted in your pull request, so you will get a notification.

Some things to keep in mind:

- You will need a Gerrit account to respond to your reviewers, including to mark feedback as 'Done' if implemented as suggested. It is a good idea to familiarize yourself with Gerrit, such as by skimming the open CLs, subscribing to updates on interesting CLs (via the star icon), or reviewing or giving a +1 to other people's CLs.
- To update the pull request with new code, just push it to the branch; you can either add more commits, or rebase and force-push (both styles are accepted).
- If the request is accepted, all commits will be squashed, and the final commit description will be composed by concatenating the pull request's title and description. The individual commits' descriptions will be discarded. See Writing good commit messages for some suggestions.
- See the GerritBot FAQ for more details.

Good commit messages

Commit messages in Go follow a specific set of conventions, which we discuss in this section.

Here is an example of a good one:

math: improve Sin, Cos and Tan precision for very large arguments

The existing implementation has poor numerical properties for
large arguments, so use the McGillicutty algorithm to improve
accuracy above 1e10.

The algorithm is described at https://wikipedia.org/wiki/McGillicutty_Algorithm

Fixes #159\

First line

The first line of the change description is conventionally a short one-line summary of the change, prefixed by the primary affected package.

A rule of thumb is that it should be written so to complete the sentence "This change modifies Go to _____." That means it does not start with a capital letter, is not a complete sentence, and actually summarizes the result of the change.

Follow the first line by a blank line.

Main content

The rest of the description elaborates and should provide context for the change and explain what it does. Write in complete sentences with correct punctuation, just like for your comments in Go. Don't use HTML, Markdown, or any other markup language. The text should be wrapped at around 72 columns. See Commit messages for additional details.

Add any relevant information, such as benchmark data if the change affects performance. The benchstat tool is conventionally used to format benchmark data for change descriptions.

Referencing issues

The special notation "Fixes #12345" associates the change with issue 12345 in the Go issue tracker. When this change is eventually applied, the issue tracker will automatically mark the issue as fixed.

If the change is a partial step towards the resolution of the issue, write "Updates #12345" instead. This will leave a comment in the issue linking back to the change in Gerrit, but it will not close the issue when the change is applied.

If you are sending a change against a golang.org/x/... repository, you must use the fully-qualified syntax supported by GitHub to make sure the change is linked to the issue in the main repository,

not the x/ repository. Most issues are tracked in the main repository's issue tracker. The correct form is "Fixes golang/go#159".

The review process

This section explains the review process in detail and how to approach reviews after a change has been mailed.

Common beginner mistakes

When a change is sent to Gerrit, it is usually triaged within a few days. A maintainer will have a look and provide some initial review that for first-time contributors usually focuses on basic cosmetics and common mistakes. These include things like:

- Commit message not following the suggested format.
- The lack of a linked GitHub issue. The vast majority of changes require a linked issue that describes the bug or the feature that the change fixes or implements, and consensus should have been reached on the tracker before proceeding with it. Gerrit reviews do not discuss the merit of the change, just its implementation.
  Only trivial or cosmetic changes will be accepted without an associated issue.
- Change sent during the freeze phase of the development cycle, when the tree is closed for general changes. In this case, a maintainer might review the code with a line such as `R=go1.12`, which means that it will be reviewed later when the tree opens for a new development window. You can add `R=go1.XX` as a comment yourself if you know that it's not the correct time frame for the change.

Trybots

After an initial reading of your change, maintainers will trigger trybots, a cluster of servers that will run the full test suite on several different architectures. Most trybots complete in a few minutes, at which point a link will be posted in Gerrit where you can see the results.

If the trybot run fails, follow the link and check the full logs of the platforms on which the tests failed. Try to understand what broke, update your patch to fix it, and upload again. Maintainers will trigger a new trybot run to see if the problem was fixed.

Sometimes, the tree can be broken on some platforms for a few hours; if the failure reported by the trybot doesn't seem related to your patch, go to the Build Dashboard and check if the same failure appears in other recent commits on the same platform. In this case, feel free to write a comment in Gerrit to mention that the failure is unrelated to your change, to help maintainers understand the

situation. You can also search GitHub issues for the failing error message or browse recently updated `watchflakes` issues. If your change is based on an older commit or it looks like someone else might have already fixed the problem, try rebasing to the latest master commit via `git rebase` .

Reviews

The Go community values very thorough reviews. Think of each review comment like a ticket: you are expected to somehow "close" it by acting on it, either by implementing the suggestion or convincing the reviewer otherwise.

After you update the change, go through the review comments and make sure to reply to every one. You can click the "Done" button to reply indicating that you've implemented the reviewer's suggestion; otherwise, click on "Reply" and explain why you have not, or what you have done instead.

It is perfectly normal for changes to go through several round of reviews, with one or more reviewers making new comments every time and then waiting for an updated change before reviewing again. This cycle happens even for experienced contributors, so don't be discouraged by it.

Voting conventions

As they near a decision, reviewers will apply a Code-Review “vote” to your change. There are two possible votes:

- **+2** The change is approved for being merged. Only Go maintainers (also referred to as “approvers”) can cast a +2 vote.
- **+1** The change looks good, but either the reviewer is requesting minor changes before approving it, or they are not a maintainer and cannot approve it, but would like to encourage an approval.

To be submitted, a change must have a Code-Review +2 from a maintainer.

Maintainers can also apply a Hold +1 vote to the change, to mark a change that should not be submitted now (for example, because the proposal review for new API in the change has not completed).

To be submitted, a change must not have any Hold +1 votes from a maintainer.

Finally, to be submitted, a change must have the involvement of two Google employees, either as the uploader of the change or as a reviewer voting at least Code-Review +1. This requirement is for compliance and supply chain security reasons.

Submitting an approved change

When a change is ready, a maintainer will submit the change, which adds it as a commit to the Gerrit repository.

The two steps (approving and submitting) are separate because in some cases maintainers may want to approve it but not to submit it right away (for instance, the tree could be temporarily frozen).

Submitting a change checks it into the repository. The change description will include a link to the code review, which will be updated with a link to the change in the repository. Since the method used to integrate the changes is Git's "Cherry Pick", the commit hashes in the repository will be changed by the submit operation.

If your change has been approved for a few days without being submitted, feel free to write a comment in Gerrit requesting submission.

More information

In addition to the information here, the Go community maintains a CodeReview wiki page. Feel free to contribute to this page as you learn more about the review process.

Miscellaneous topics

This section collects a number of other comments that are outside the issue/edit/code review/submit process itself.

Gopls

When working on the main Go repository and using `gopls` with your editor, the `go` command invoked by `gopls` must correspond to the version of the source code you are working on. The `go` command can be built with `make.bash` and the `bin` directory should be added to your `PATH`. See Gopls: Advanced topics for additional details.

Complete documentation for Gopls can be found at https://go.dev/gopls.

Copyright headers

Files in the Go repository don't list author names, both to avoid clutter and to avoid having to keep the lists up to date. Instead, your name will appear in the change log.

New files that you contribute should use the standard copyright header:

```
// Copyright {{now.UTC.Year}} The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.\
```

Files in the repository are copyrighted the year they are added. Do not update the copyright year on files that you change.

Troubleshooting mail errors

The most common way that the `git codereview mail` command fails is because the e-mail address in the commit does not match the one that you used during the registration process.
If you see something like...

```
remote: Processing changes: refs: 1, done
remote:
remote: ERROR:  In commit ab13517fa29487dcf8b0d48916c51639426c5ee9
remote: ERROR:  author email address XXXXXXXXXXXXXXXXXX
remote: ERROR:  does not match your user account.\
```

you need to configure Git for this repository to use the e-mail address that you registered with. To change the e-mail address to ensure this doesn't happen again, run:

```
$ git config user.email email@address.com\
```

Then change the commit to use this alternative e-mail address with this command:

```
$ git commit --amend --author="Author Name <email@address.com>"\
```

Then retry by running:

```
$ git codereview mail\
```

Quickly testing your changes

Running `all.bash` for every single change to the code tree is burdensome. Even though it is strongly suggested to run it before sending a change, during the normal development cycle you may want to compile and test only the package you are developing.

- In general, you can run `make.bash` instead of `all.bash` to only rebuild the Go tool chain without running the whole test suite. Or you can run `run.bash` to only run the whole test suite without rebuilding the tool chain. You can think of `all.bash` as `make.bash` followed by `run.bash`.
- In this section, we'll call the directory into which you cloned the Go repository `$GOROOT`. The `go` tool built by `$GOROOT/src/make.bash` will be installed in `$GOROOT/bin/go` and you can invoke it to test your code. For instance, if you have modified the compiler and you want to test how it affects the test suite of your own project, just run `go test` using it:
  $ cd <MYPROJECTDIR>
  $ $GOROOT/bin/go test\
- If you're changing the standard library, you probably don't need to rebuild the compiler: you can just run the tests for the package you've changed. You can do that either with the Go version you normally use, or with the Go compiler built from your clone (which is sometimes required because the standard library code you're modifying might require a newer version than the stable one you have installed).
  $ cd $GOROOT/src/crypto/sha1
  $ [make changes...]
  $ $GOROOT/bin/go test .\
- If you're modifying the compiler itself, you can just recompile the `compile` tool (which is the internal binary invoked by `go build` to compile each single package). After that, you will want to test it by compiling or running something.
  $ cd $GOROOT/src
  $ [make changes...]
  $ $GOROOT/bin/go install cmd/compile
  $ $GOROOT/bin/go build [something...] # test the new compiler
  $ $GOROOT/bin/go run [something...] # test the new compiler
  $ $GOROOT/bin/go test [something...] # test the new compiler
  The same applies to other internal tools of the Go tool chain, such as `asm`, `cover`, `link`, and so on. Just recompile and install the tool using `go install cmd/<TOOL>` and then use the built Go binary to test it.
- In addition to the standard per-package tests, there is a top-level test suite in `$GOROOT/test` that contains several black-box and regression tests. The test suite is run by `all.bash` but you can also run it manually:
  $ $GOROOT/bin/go test cmd/internal/testdir\

Specifying a reviewer / CCing others

Unless explicitly told otherwise, such as in the discussion leading up to sending in the change, it's better not to specify a reviewer. All changes are automatically CC'ed to the golang-codereviews@googlegroups.com mailing list. If this is your first ever change, there may be a

moderation delay before it appears on the mailing list, to prevent spam.

You can specify a reviewer or CC interested parties using the `-r` or `-cc` options. Both accept a comma-separated list of e-mail addresses:

```
$ git codereview mail -r joe@golang.org -cc mabel@example.com,math-nuts@swtch.com\
```

Synchronize your client

While you were working, others might have submitted changes to the repository. To update your local branch, run

```
$ git codereview sync\
```

(Under the covers this runs `git pull -r`.)

Reviewing code by others

As part of the review process reviewers can propose changes directly (in the GitHub workflow this would be someone else attaching commits to a pull request). Gerrit provides access to commands that will help you import changes proposed by another developer so you can review/test them locally. From the Gerrit page for the CL you want to import, open the " ⋮ " menu, click the "Download patch" link. Depending on your preferred git workflow, choose the appropriate command. The options will look something like this:

```
$ git fetch https://go.googlesource.com/review refs/changes/21/13245/1 && git checkout FETCH_HEAD\
```

To revert, change back to the branch you were working in.

Set up git aliases

The `git-codereview` command can be run directly from the shell by typing, for instance,

```
$ git codereview sync\
```

but it is more convenient to set up aliases for `git-codereview`'s own subcommands, so that the above becomes,

$ git sync\

The `git-codereview` subcommands have been chosen to be distinct from Git's own, so it's safe to define these aliases. To install them, copy this text into your Git configuration file (usually `.gitconfig` in your home directory):

```
[alias]
	change = codereview change
	gofmt = codereview gofmt
	mail = codereview mail
	pending = codereview pending
	submit = codereview submit
	sync = codereview sync\
```

Sending multiple dependent changes

Advanced users may want to stack up related commits in a single branch. Gerrit allows for changes to be dependent on each other, forming such a dependency chain. Each change will need to be approved and submitted separately but the dependency will be visible to reviewers.

To send out a group of dependent changes, keep each change as a different commit under the same branch, and then run:

$ git codereview mail HEAD\

Make sure to explicitly specify `HEAD`, which is usually not required when sending single changes.

More details can be found in the git-codereview documentation.

Minor Releases

If you want to make a change to a release branch for backporting, see Minor Releases.

□□□ #2
□ □□□ □□□ 2026-10-31 13:51:26 UTC
□ □□□ □□□ 2026-10-31 13:51:28 UTC