NAME

git-bisect - Use binary search to find the commit that introduced a bug

SYNOPSIS

git bisect <subcommand> <options>

DESCRIPTION

The command takes various subcommands, and different options depending on the subcommand:

```
git bisect start [--term-{new,bad}=<term> --term-{old,good}=<term>]
                 [--no-checkout] [--first-parent] [<bad> [<good>...]] [--] [<paths>...]
git bisect (bad|new|<term-new>) [<rev>]
git bisect (good|old|<term-old>) [<rev>...]
git bisect terms [--term-good | --term-bad]
git bisect skip [(<rev>|<range>)...]
git bisect reset [<commit>]
git bisect (visualize|view)
git bisect replay <logfile>
git bisect log
git bisect run <cmd>...
git bisect help
```

This command uses a binary search algorithm to find which commit in your project's history introduced a bug. You use it by first telling it a "bad" commit that is known to contain the bug, and a "good" commit that is known to be before the bug was introduced. Then **git bisect** picks a commit between those two endpoints and asks you whether the selected commit is "good" or "bad". It continues narrowing down the range until it finds the exact commit that introduced the change.

In fact, **git bisect** can be used to find the commit that changed **any** property of your project; e.g., the commit that fixed a bug, or the commit that caused a benchmark's performance to improve. To support this more general usage, the terms "old" and "new" can be used in place of "good" and "bad", or you can choose your own terms. See section "Alternate terms" below for more information.

Basic bisect commands: start, bad, good

As an example, suppose you are trying to find the commit that broke a feature that was known to work in version **v2.6.13-rc2** of your project. You start a bisect session as follows:

```
$ git bisect start
$ git bisect bad                # Current version is bad
```

```
$ git bisect good v2.6.13-rc2    # v2.6.13-rc2 is known to be good
```

Once you have specified at least one bad and one good commit, **git bisect** selects a commit in the middle of that range of history, checks it out, and outputs something similar to the following:

```
Bisecting: 675 revisions left to test after this (roughly 10 steps)
```

You should now compile the checked-out version and test it. If that version works correctly, type

```
$ git bisect good
```

If that version is broken, type

```
$ git bisect bad
```

Then **git bisect** will respond with something like

```
Bisecting: 337 revisions left to test after this (roughly 9 steps)
```

Keep repeating the process: compile the tree, test it, and depending on whether it is good or bad run **git bisect good** or **git bisect bad** to ask for the next commit that needs testing.

Eventually there will be no more revisions left to inspect, and the command will print out a description of the first bad commit. The reference **refs/bisect/bad** will be left pointing at that commit.

Bisect reset

After a bisect session, to clean up the bisection state and return to the original HEAD, issue the following command:

```
$ git bisect reset
```

By default, this will return your tree to the commit that was checked out before **git bisect start**. (A new **git bisect start** will also do that, as it cleans up the old bisection state.)

With an optional argument, you can return to a different commit instead:

```
$ git bisect reset <commit>
```

For example, **git bisect reset bisect/bad** will check out the first bad revision, while **git bisect reset HEAD** will leave you on the current bisection commit and avoid switching commits at all.

Alternate terms

Sometimes you are not looking for the commit that introduced a breakage, but rather for a commit that caused a change between some other "old" state and "new" state. For example, you might be looking for the commit that introduced a particular fix. Or you might be looking for the first commit in which the source-code filenames were finally all converted to your company’s naming standard. Or whatever.

In such cases it can be very confusing to use the terms "good" and "bad" to refer to "the state before the change" and "the state after the change". So instead, you can use the terms "old" and "new", respectively, in place of "good" and "bad". (But note that you cannot mix "good" and "bad" with "old" and "new" in a single session.)

In this more general usage, you provide **git bisect** with a "new" commit that has some property and an "old" commit that doesn’t have that property. Each time **git bisect** checks out a commit, you test if that commit has the property. If it does, mark the commit as "new"; otherwise, mark it as "old". When the bisection is done, **git bisect** will report which commit introduced the property.

To use "old" and "new" instead of "good" and bad, you must run **git bisect start** without commits as argument and then run the following commands to add the commits:

```
git bisect old [<rev>]
```

to indicate that a commit was before the sought change, or

```
git bisect new [<rev>...]
```

to indicate that it was after.

To get a reminder of the currently used terms, use

```
git bisect terms
```

You can get just the old (respectively new) term with **git bisect terms --term-old** or **git bisect terms --term-good**.

If you would like to use your own terms instead of "bad"/"good" or "new"/"old", you can choose any names you like (except existing bisect subcommands like **reset**, **start**, ...) by starting the bisection using

```
git bisect start --term-old <term-old> --term-new <term-new>
```

For example, if you are looking for a commit that introduced a performance regression, you might use

```
git bisect start --term-old fast --term-new slow
```

Or if you are looking for the commit that fixed a bug, you might use

```
git bisect start --term-new fixed --term-old broken
```

Then, use **git bisect <term-old>** and **git bisect <term-new>** instead of **git bisect good** and **git bisect bad** to mark commits.

Bisect visualize/view

To see the currently remaining suspects in *gitk*, issue the following command during the bisection process (the subcommand **view** can be used as an alternative to **visualize**):

```
$ git bisect visualize
```

Git detects a graphical environment through various environment variables: **DISPLAY**, which is set in X Window System environments on Unix systems. **SESSIONNAME**, which is set under Cygwin in interactive desktop sessions. **MSYSTEM**, which is set under Msys2 and Git for Windows. **SECURITYSESSIONID**, which may be set on macOS in interactive desktop sessions.

If none of these environment variables is set, *git log* is used instead. You can also give command-line options such as **-p** and **--stat**.

```
$ git bisect visualize --stat
```

Bisect log and bisect replay

After having marked revisions as good or bad, issue the following command to show what has been done so far:

```
$ git bisect log
```

If you discover that you made a mistake in specifying the status of a revision, you can save the output of this command to a file, edit it to remove the incorrect entries, and then issue the following commands to return to a corrected state:

```
$ git bisect reset
$ git bisect replay that-file
```

Avoiding testing a commit

If, in the middle of a bisect session, you know that the suggested revision is not a good one to test (e.g. it fails to build and you know that the failure does not have anything to do with the bug you are chasing), you can manually select a nearby commit and test that one instead.

For example:

```
$ git bisect good/bad                # previous round was good or bad.
Bisecting: 337 revisions left to test after this (roughly 9 steps)
$ git bisect visualize               # oops, that is uninteresting.
$ git reset --hard HEAD~3            # try 3 revisions before what
                                     # was suggested
```

Then compile and test the chosen revision, and afterwards mark the revision as good or bad in the usual manner.

Bisect skip

Instead of choosing a nearby commit by yourself, you can ask Git to do it for you by issuing the command:

```
$ git bisect skip                # Current version cannot be tested
```

However, if you skip a commit adjacent to the one you are looking for, Git will be unable to tell

exactly which of those commits was the first bad one.

You can also skip a range of commits, instead of just one commit, using range notation. For example:

```
$ git bisect skip v2.5..v2.6
```

This tells the bisect process that no commit after **v2.5**, up to and including **v2.6**, should be tested.

Note that if you also want to skip the first commit of the range you would issue the command:

```
$ git bisect skip v2.5 v2.5..v2.6
```

This tells the bisect process that the commits between **v2.5** and **v2.6** (inclusive) should be skipped.

Cutting down bisection by giving more parameters to bisect start

You can further cut down the number of trials, if you know what part of the tree is involved in the problem you are tracking down, by specifying path parameters when issuing the **bisect start** command:

```
$ git bisect start -- arch/i386 include/asm-i386
```

If you know beforehand more than one good commit, you can narrow the bisect space down by specifying all of the good commits immediately after the bad commit when issuing the **bisect start** command:

```
$ git bisect start v2.6.20-rc6 v2.6.20-rc4 v2.6.20-rc1 --
        # v2.6.20-rc6 is bad
        # v2.6.20-rc4 and v2.6.20-rc1 are good
```

Bisect run

If you have a script that can tell if the current source code is good or bad, you can bisect by issuing the command:

```
$ git bisect run my_script arguments
```

Note that the script (**my_script** in the above example) should exit with code 0 if the current source code

is good/old, and exit with a code between 1 and 127 (inclusive), except 125, if the current source code is bad/new.

Any other exit code will abort the bisect process. It should be noted that a program that terminates via **exit(-1)** leaves $? = 255, (see the exit(3) manual page), as the value is chopped with **& 0377**.

The special exit code 125 should be used when the current source code cannot be tested. If the script exits with this code, the current revision will be skipped (see **git bisect skip** above). 125 was chosen as the highest sensible value to use for this purpose, because 126 and 127 are used by POSIX shells to signal specific error status (127 is for command not found, 126 is for command found but not executable--these details do not matter, as they are normal errors in the script, as far as **bisect run** is concerned).

You may often find that during a bisect session you want to have temporary modifications (e.g. s/#define DEBUG 0/#define DEBUG 1/ in a header file, or "revision that does not have this commit needs this patch applied to work around another problem this bisection is not interested in") applied to the revision being tested.

To cope with such a situation, after the inner *git bisect* finds the next revision to test, the script can apply the patch before compiling, run the real test, and afterwards decide if the revision (possibly with the needed patch) passed the test and then rewind the tree to the pristine state. Finally the script should exit with the status of the real test to let the **git bisect run** command loop determine the eventual outcome of the bisect session.

OPTIONS

--no-checkout

Do not checkout the new working tree at each iteration of the bisection process. Instead just update a special reference named **BISECT_HEAD** to make it point to the commit that should be tested.

This option may be useful when the test you would perform in each step does not require a checked out tree.

If the repository is bare, **--no-checkout** is assumed.

--first-parent

Follow only the first parent commit upon seeing a merge commit.

In detecting regressions introduced through the merging of a branch, the merge commit will be identified as introduction of the bug and its ancestors will be ignored.

This option is particularly useful in avoiding false positives when a merged branch contained broken or non-buildable commits, but the merge itself was OK.

EXAMPLES

- bisect a broken build between v1.2 and HEAD:

```
$ git bisect start HEAD v1.2 --      # HEAD is bad, v1.2 is good
$ git bisect run make               # "make" builds the app
$ git bisect reset                  # quit the bisect session
```

- bisect a test failure between origin and HEAD:

```
$ git bisect start HEAD origin --    # HEAD is bad, origin is good
$ git bisect run make test          # "make test" builds and tests
$ git bisect reset                  # quit the bisect session
```

- bisect a broken test case:

```
$ cat ~/test.sh
#!/bin/sh
make || exit 125                    # this skips broken builds
~/check_test_case.sh                # does the test case pass?
$ git bisect start HEAD HEAD~10 --   # culprit is among the last 10
$ git bisect run ~/test.sh
$ git bisect reset                  # quit the bisect session
```

Here we use a **test.sh** custom script. In this script, if **make** fails, we skip the current commit. **check_test_case.sh** should **exit 0** if the test case passes, and **exit 1** otherwise.

It is safer if both **test.sh** and **check_test_case.sh** are outside the repository to prevent interactions between the bisect, make and test processes and the scripts.

- bisect with temporary modifications (hot-fix):

```
$ cat ~/test.sh
#!/bin/sh

# tweak the working tree by merging the hot-fix branch
# and then attempt a build
if      git merge --no-commit --no-ff hot-fix &&
        make
then
        # run project specific test and report its status
        ~/check_test_case.sh
        status=$?
else
        # tell the caller this is untestable
        status=125
fi

# undo the tweak to allow clean flipping to the next commit
git reset --hard

# return control
exit $status
```

This applies modifications from a hot-fix branch before each test run, e.g. in case your build or test environment changed so that older revisions may need a fix which newer ones have already. (Make sure the hot-fix branch is based off a commit which is contained in all revisions which you are bisecting, so that the merge does not pull in too much, or use **git cherry-pick** instead of **git merge**.)

- bisect a broken test case:

```
$ git bisect start HEAD HEAD~10 --   # culprit is among the last 10
$ git bisect run sh -c "make || exit 125; ~/check_test_case.sh"
$ git bisect reset                   # quit the bisect session
```

This shows that you can do without a run script if you write the test on a single line.

- a good region of the object graph in a damaged repository

```
$ git bisect start HEAD <known-good-commit> [ <boundary-commit> ... ] --no-checkout
$ git bisect run sh -c '
    GOOD=$(git for-each-ref "--format=%(objectname)" refs/bisect/good-*) &&
    git rev-list --objects BISECT_HEAD --not $GOOD >tmp.$$ &&
    git pack-objects --stdout >/dev/null <tmp.$$
    rc=$?
    rm -f tmp.$$
    test $rc = 0'

$ git bisect reset              # quit the bisect session
```

In this case, when *git bisect run* finishes, bisect/bad will refer to a commit that has at least one parent whose reachable graph is fully traversable in the sense required by *git pack objects*.

ტ

for a fix instead of a regression in the code

```
$ git bisect start
$ git bisect new HEAD    # current commit is marked as new
$ git bisect old HEAD~10 # the tenth commit from now is marked as old
```

or:

```
$ git bisect start --term-old broken --term-new fixed
$ git bisect fixed
$ git bisect broken HEAD~10
```

Getting help

Use **git bisect** to get a short usage description, and **git bisect help** or **git bisect -h** to get a long usage description.

SEE ALSO

Fighting regressions with git bisect[1], **git-blame**(1).

GIT

Part of the **git**(1) suite

NOTES

1. Fighting regressions with git bisect

git-htmldocs/git-bisect-lk2009.html