NAME

git-bisect - Find by binary search the change that introduced a bug

SYNOPSIS

git bisect <subcommand> <options>

DESCRIPTION

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

```
git bisect help
git bisect start [--no-checkout] [<bad> [<good>...]] [--] [<paths>...]
git bisect bad [<rev>]
git bisect good [<rev>...]
git bisect skip [(<rev>|<range>)...]
git bisect reset [<commit>]
git bisect visualize
git bisect replay <logfile>
git bisect log
git bisect run <cmd>...
```

This command uses *git rev-list --bisect* to help drive the binary search process to find which change introduced a bug, given an old good commit object name and a later bad commit object name.

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.

Basic bisect commands: start, bad, good

Using the Linux kernel tree as an example, basic use of the bisect command is as follows:

```
$ git bisect start
$ git bisect bad # Current version is bad
$ git bisect good v2.6.13-rc2 # v2.6.13-rc2 was the last version
# tested that was good
```

When you have specified at least one bad and one good version, the command bisects the revision tree and outputs something similar to the following:

```
Bisecting: 675 revisions left to test after this
```

The state in the middle of the set of revisions is then checked out. You would now compile that kernel and boot it. If the booted kernel works correctly, you would then issue the following command:

```
$ git bisect good # this one is good
```

The output of this command would be something similar to the following:

```
Bisecting: 337 revisions left to test after this
```

You keep repeating this process, compiling the tree, testing it, and depending on whether it is good or bad issuing the command git bisect good or git bisect bad to ask for the next bisection.

Eventually there will be no more revisions left to bisect, and you will have been left with the first bad kernel revision in refs/bisect/bad.

Bisect reset

After a bisect session, to clean up the bisection state and return to the original HEAD (i.e., to quit bisecting), 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 HEAD will leave you on the current bisection commit and avoid switching commits at all, while git bisect reset bisect/bad will check out the first bad revision.

Bisect visualize

To see the currently remaining suspects in *gitk*, issue the following command during the bisection process:

```
$ git bisect visualize
```

view may also be used as a synonym for visualize.

If the *DISPLAY* environment variable is not set, *git log* is used instead. You can also give command-line options such as -p and --stat.

```
$ git bisect view --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 next suggested revision is not a good one to test (e.g. the change the commit introduces is known not to work in your environment and you know it does not have anything to do with the bug you are chasing), you may want to find a nearby commit and try that instead.

For example:

```
$ git bisect good/bad # previous round was good or bad.
Bisecting: 337 revisions left to test after this
$ 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 by yourself a nearby commit, you can ask Git to do it for you by issuing the command:

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

But Git may eventually be unable to tell the first bad commit among a bad commit and one or more skipped commits.

You can even skip a range of commits, instead of just one commit, using the *<commit1>..<commit2>* 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 included and v2.6 included 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, and exit with a code between 1 and 127 (inclusive), except 125, if the current source code is bad.

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.

EXAMPLES

- Automatically 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
  ```

- Automatically 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
  ```

- Automatically 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.

- Automatically 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 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.)

- Automatically 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.

- Locate 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*.

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
   file:///usr/share/doc/git/html/git-bisect-lk2009.html