NAME

perlrepository - Using the Perl source repository

SYNOPSIS

All of Perl's source code is kept centrally in a Git repository at *perl5.git.perl.org*. The repository contains many Perl revisions from Perl 1 onwards and all the revisions from Perforce, the version control system we were using previously. This repository is accessible in different ways.

The full repository takes up about 80MB of disk space. A check out of the blead branch (that is, the main development branch, which contains bleadperl, the development version of perl 5) takes up about 160MB of disk space (including the repository). A build of bleadperl takes up about 200MB (including the repository and the check out).

GETTING ACCESS TO THE REPOSITORY

READ ACCESS VIA THE WEB

You may access the repository over the web. This allows you to browse the tree, see recent commits, subscribe to RSS feeds for the changes, search for particular commits and more. You may access it at:

```
  http://perl5.git.perl.org/perl.git
```

A mirror of the repository is found at:

```
  http://github.com/github/perl
```

READ ACCESS VIA GIT

You will need a copy of Git for your computer. You can fetch a copy of the repository using the Git protocol (which uses port 9418):

```
  git clone git://perl5.git.perl.org/perl.git perl-git
```

This clones the repository and makes a local copy in the *perl-git* directory.

If your local network does not allow you to use port 9418, then you can fetch a copy of the repository over HTTP (this is slower):

```
  git clone http://perl5.git.perl.org/perl.git perl-http
```

This clones the repository and makes a local copy in the *perl-http* directory.

WRITE ACCESS TO THE REPOSITORY

If you are a committer, then you can fetch a copy of the repository that you can push back on with:

```
  git clone ssh://perl5.git.perl.org/gitroot/perl.git perl-ssh
```

This clones the repository and makes a local copy in the *perl-ssh* directory.

If you cloned using the git protocol, which is faster than ssh, then you will need to modify your config in order to enable pushing. Edit *.git/config* where you will see something like:

```
  [remote "origin"]
  url = git://perl5.git.perl.org/perl.git
```

change that to something like this:

```
  [remote "origin"]
  url = ssh://perl5.git.perl.org/gitroot/perl.git
```

NOTE: there are symlinks set up so that the /gitroot is optional and since SSH is the default protocol you can actually shorten the "url" to `perl5.git.perl.org:/perl.git`.

You can also set up your user name and e-mail address. For example

```
  % git config user.name "Leon Brocard"
  % git config user.email acme@astray.com
```

It is also possible to keep `origin` as a git remote, and add a new remote for ssh access:

```
  % git remote add camel perl5.git.perl.org:/perl.git
```

This allows you to update your local repository by pulling from `origin`, which is faster and doesn't require you to authenticate, and to push your changes back with the `camel` remote:

```
  % git fetch camel
  % git push camel
```

The `fetch` command just updates the `camel` refs, as the objects themselves should have been fetched when pulling from `origin`.

The committers have access to 2 servers that serve perl5.git.perl.org. One is camel.booking.com, which is the 'master' repository. The perl5.git.perl.org IP address also lives on this machine. The second one is dromedary.booking.com, which can be used for general testing and development. Dromedary syncs the git tree from camel every few minutes, you should not push there. Both machines also have a full CPAN mirror. To share files with the general public, dromedary serves your ~/public_html/ as http://users.perl5.git.perl.org/~yourlogin/

OVERVIEW OF THE REPOSITORY

Once you have changed into the repository directory, you can inspect it.

After a clone the repository will contain a single local branch, which will be the current branch as well, as indicated by the asterisk.

```
  % git branch
  * blead
```

Using the -a switch to `branch` will also show the remote tracking branches in the repository:

```
  % git branch -a
  * blead
    origin/HEAD
    origin/blead
  ...
```

The branches that begin with "origin" correspond to the "git remote" that you cloned from (which is named "origin"). Each branch on the remote will be exactly tracked by theses branches. You should NEVER do work on these remote tracking branches. You only ever do work in a local branch. Local branches can be configured to automerge (on pull) from a designated remote tracking branch. This is the case with the default branch `blead` which will be configured to merge from the remote tracking branch `origin/blead`.

You can see recent commits:

```
  % git log
```

And pull new changes from the repository, and update your local repository (must be clean first)

```
  % git pull
```

Assuming we are on the branch `blead` immediately after a pull, this command would be more or less equivalent to:

```
  % git fetch
  % git merge origin/blead
```

In fact if you want to update your local repository without touching your working directory you do:

```
  % git fetch
```

And if you want to update your remote-tracking branches for all defined remotes simultaneously you can do

```
  % git remote update
```

Neither of these last two commands will update your working directory, however both will update the remote-tracking branches in your repository.

To switch to another branch:

```
  % git checkout origin/maint-5.8-dor
```

To make a local branch of a remote branch:

```
  % git checkout -b maint-5.10 origin/maint-5.10
```

To switch back to blead:

```
  % git checkout blead
```

FINDING OUT YOUR STATUS

The most common git command you will use will probably be

```
  % git status
```

This command will produce as output a description of the current state of the repository, including modified files and unignored untracked files, and in addition it will show things like what files have been staged for the next commit, and usually some useful information about how to change things. For instance the following:

```
  $ git status
  # On branch blead
  # Your branch is ahead of 'origin/blead' by 1 commit.
  #
  # Changes to be committed:
  #   (use "git reset HEAD <file>..." to unstage)
  #
  #       modified:   pod/perlrepository.pod
  #
  # Changed but not updated:
  #   (use "git add <file>..." to update what will be committed)
  #
  #       modified:   pod/perlrepository.pod
  #
  # Untracked files:
  #   (use "git add <file>..." to include in what will be committed)
  #
  #       deliberate.untracked
```

This shows that there were changes to this document staged for commit, and that there were further changes in the working directory not yet staged. It also shows that there was an untracked file in the working directory, and as you can see shows how to change all of this. It also shows that there is one commit on the working branch `blead` which has not been pushed to the `origin` remote yet. **NOTE**: that this output is also what you see as a template if you do not provide a message to `git commit`.

Assuming we commit all the mentioned changes above:

```
  % git commit -a -m'explain git status and stuff about remotes'
  Created commit daf8e63: explain git status and stuff about remotes
   1 files changed, 83 insertions(+), 3 deletions(-)
```

We can re-run git status and see something like this:

```
  % git status
  # On branch blead
  # Your branch is ahead of 'origin/blead' by 2 commits.
  #
  # Untracked files:
  #   (use "git add <file>..." to include in what will be committed)
  #
  #       deliberate.untracked
  nothing added to commit but untracked files present (use "git add" to
track)
```

When in doubt, before you do anything else, check your status and read it carefully, many questions are answered directly by the git status output.

SUBMITTING A PATCH

If you have a patch in mind for Perl, you should first get a copy of the repository:

```
  % git clone git://perl5.git.perl.org/perl.git perl-git
```

Then change into the directory:

```
  % cd perl-git
```

Alternatively, if you already have a Perl repository, you should ensure that you're on the *blead* branch, and your repository is up to date:

```
  % git checkout blead
  % git pull
```

It's preferable to patch against the latest blead version, since this is where new development occurs for all changes other than critical bug fixes. Critical bug fix patches should be made against the relevant maint branches, or should be submitted with a note indicating all the branches where the fix should be applied.

Now that we have everything up to date, we need to create a temporary new branch for these changes and switch into it:

```
  % git checkout -b orange
```

which is the short form of

```
  % git branch orange
  % git checkout orange
```

Then make your changes. For example, if Leon Brocard changes his name to Orange Brocard, we should change his name in the AUTHORS file:

```
  % perl -pi -e 's{Leon Brocard}{Orange Brocard}' AUTHORS
```

You can see what files are changed:

```
  % git status
  # On branch orange
  # Changes to be committed:
  #   (use "git reset HEAD <file>..." to unstage)
  #
  # modified:   AUTHORS
  #
```

And you can see the changes:

```
  % git diff
  diff --git a/AUTHORS b/AUTHORS
  index 293dd70..722c93e 100644
  --- a/AUTHORS
  +++ b/AUTHORS
  @@ -541,7 +541,7 @@    Lars Hecking
<lhecking@nmrc.ucc.ie>
   Laszlo Molnar                  <laszlo.molnar@eth.ericsson.se>
   Leif Huhn                      <leif@hale.dkstat.com>
   Len Johnson                    <lenjay@ibm.net>
  -Leon Brocard                   <acme@astray.com>
  +Orange Brocard                 <acme@astray.com>
   Les Peters                     <lpeters@aol.net>
   Lesley Binks                   <lesley.binks@gmail.com>
   Lincoln D. Stein               <lstein@cshl.org>
```

Now commit your change locally:

```
  % git commit -a -m 'Rename Leon Brocard to Orange Brocard'
  Created commit 6196c1d: Rename Leon Brocard to Orange Brocard
   1 files changed, 1 insertions(+), 1 deletions(-)
```

You can examine your last commit with:

```
  % git show HEAD
```

and if you are not happy with either the description or the patch itself you can fix it up by editing the files once more and then issue:

```
  % git commit -a --amend
```

Now you should create a patch file for all your local changes:

```
  % git format-patch origin
  0001-Rename-Leon-Brocard-to-Orange-Brocard.patch
```

You should now send an email to perl5-porters@perl.org with a description of your changes, and include this patch file as an attachment.

If you want to delete your temporary branch, you may do so with:

```
  % git checkout blead
  % git branch -d orange
  error: The branch 'orange' is not an ancestor of your current HEAD.
  If you are sure you want to delete it, run 'git branch -D orange'.
  % git branch -D orange
  Deleted branch orange.
```

A note on derived files

Be aware that many files in the distribution are derivative--avoid patching them, because git won't see the changes to them, and the build process will overwrite them. Patch the originals instead. Most utilities (like perldoc) are in this category, i.e. patch utils/perldoc.PL rather than utils/perldoc. Similarly, don't create patches for files under $src_root/ext from their copies found in $install_root/lib. If you are unsure about the proper location of a file that may have gotten copied while building the source distribution, consult the `MANIFEST`.

A note on binary files

Since the patch(1) utility cannot deal with binary files, it's important that you either avoid the use of binary files in your patch, generate the files dynamically, or that you encode any binary files using the *uupacktool.pl* utility.

Assuming you needed to include a gzip-encoded file for a module's test suite, you might do this as follows using the *uupacktool.pl* utility:

```
    $ perl uupacktool.pl -v -p -D lib/Some/Module/t/src/t.gz
    Writing lib/Some/Module/t/src/t.gz into
lib/Some/Module/t/src/t.gz.packed
```

This will replace the `t.gz` file with an encoded counterpart. During `make test`, before any tests are run, perl's Makefile will restore all the `.packed` files mentioned in the MANIFEST to their original name. This means that the test suite does not need to be aware of this packing scheme and will not need to be altered.

Getting your patch accepted

The first thing you should include with your patch is a description of the problem that the patch corrects. If it is a code patch (rather than a documentation patch) you should also include a small test case that illustrates the bug (a patch to an existing test file is preferred).

If you are submitting a code patch there are several other things that you need to do.

Comments, Comments, Comments

Be sure to adequately comment your code. While commenting every line is unnecessary, anything that takes advantage of side effects of operators, that creates changes that will be felt outside of the function being patched, or that others may find confusing should be documented. If you are going to err, it is better to err on the side of adding too many comments than too few.

Style

In general, please follow the particular style of the code you are patching.

In particular, follow these general guidelines for patching Perl sources:

```
    8-wide tabs (no exceptions!)
    4-wide indents for code, 2-wide indents for nested CPP #defines
    try hard not to exceed 79-columns
    ANSI C prototypes
    uncuddled elses and "K&R" style for indenting control constructs
    no C++ style (//) comments
    mark places that need to be revisited with XXX (and revisit
often!)
    opening brace lines up with "if" when conditional spans multiple
        lines; should be at end-of-line otherwise
    in function definitions, name starts in column 0 (return value is
 on
        previous line)
    single space after keywords that are followed by parens, no space
        between function name and following paren
    avoid assignments in conditionals, but if they're unavoidable,
use
        extra paren, e.g. "if (a && (b = c)) ..."
    "return foo;" rather than "return(foo);"
    "if (!foo) ..." rather than "if (foo == FALSE) ..." etc.
```

Testsuite

When submitting a patch you should make every effort to also include an addition to perl's regression tests to properly exercise your patch. Your testsuite additions should generally follow these guidelines (courtesy of Gurusamy Sarathy <gsar@activestate.com>):

```
    Know what you're testing.  Read the docs, and the source.
    Tend to fail, not succeed.
    Interpret results strictly.
    Use unrelated features (this will flush out bizarre
interactions).
    Use non-standard idioms (otherwise you are not testing
TIMTOWTDI).
    Avoid using hardcoded test numbers whenever possible (the
      EXPECTED/GOT found in t/op/tie.t is much more maintainable,
      and gives better failure reports).
    Give meaningful error messages when a test fails.
    Avoid using qx// and system() unless you are testing for them.
If you
      do use them, make sure that you cover _all_ perl platforms.
    Unlink any temporary files you create.
    Promote unforeseen warnings to errors with $SIG{__WARN__}.
    Be sure to use the libraries and modules shipped with the version
      being tested, not those that were already installed.
    Add comments to the code explaining what you are testing for.
    Make updating the '1..42' string unnecessary.  Or make sure that
      you update it.
    Test _all_ behaviors of a given operator, library, or function:
      - All optional arguments
      - Return values in various contexts (boolean, scalar, list,
lvalue)
      - Use both global and lexical variables
      - Don't forget the exceptional, pathological cases.
```

ACCEPTING A PATCH

If you have received a patch file generated using the above section, you should try out the patch.

First we need to create a temporary new branch for these changes and switch into it:

```
  % git checkout -b experimental
```

Patches that were formatted by `git format-patch` are applied with `git am`:

```
  % git am 0001-Rename-Leon-Brocard-to-Orange-Brocard.patch
  Applying Rename Leon Brocard to Orange Brocard
```

If just a raw diff is provided, it is also possible use this two-step process:

```
  % git apply bugfix.diff
  % git commit -a -m "Some fixing" --author="That Guy
<that.guy@internets.com>"
```

Now we can inspect the change:

```
  % git show HEAD
  commit b1b3dab48344cff6de4087efca3dbd63548ab5e2
  Author: Leon Brocard <acme@astray.com>
  Date:   Fri Dec 19 17:02:59 2008 +0000

    Rename Leon Brocard to Orange Brocard

  diff --git a/AUTHORS b/AUTHORS
  index 293dd70..722c93e 100644
  --- a/AUTHORS
  +++ b/AUTHORS
  @@ -541,7 +541,7 @@ Lars Hecking
<lhecking@nmrc.ucc.ie>
   Laszlo Molnar                  <laszlo.molnar@eth.ericsson.se>
   Leif Huhn                      <leif@hale.dkstat.com>
   Len Johnson                    <lenjay@ibm.net>
  -Leon Brocard                   <acme@astray.com>
  +Orange Brocard                 <acme@astray.com>
   Les Peters                     <lpeters@aol.net>
   Lesley Binks                   <lesley.binks@gmail.com>
   Lincoln D. Stein               <lstein@cshl.org>
```

If you are a committer to Perl and you think the patch is good, you can then merge it into blead then push it out to the main repository:

```
  % git checkout blead
  % git merge experimental
  % git push
```

If you want to delete your temporary branch, you may do so with:

```
  % git checkout blead
  % git branch -d experimental
  error: The branch 'experimental' is not an ancestor of your current HEAD.
  If you are sure you want to delete it, run 'git branch -D experimental'.
  % git branch -D experimental
  Deleted branch experimental.
```

CLEANING A WORKING DIRECTORY

The command `git clean` can with varying arguments be used as a replacement for `make clean`.

To reset your working directory to a pristine condition you can do:

```
  git clean -dxf
```

However, be aware this will delete ALL untracked content. You can use

```
  git clean -Xf
```

to remove all ignored untracked files, such as build and test byproduct, but leave any manually created files alone.

If you only want to cancel some uncommitted edits, you can use `git checkout` and give it a list of files to be reverted, or `git checkout -f` to revert them all.

If you want to cancel one or several commits, you can use `git reset`.

BISECTING

`git` provides a built-in way to determine, with a binary search in the history, which commit should be blamed for introducing a given bug.

Suppose that we have a script *~/testcase.pl* that exits with `0` when some behaviour is correct, and with `1` when it's faulty. We need an helper script that automates building `perl` and running the testcase:

```
  % cat ~/run
  #!/bin/sh
  git clean -dxf
  # If you can use ccache, add -Dcc=ccache\ gcc -Dld=gcc to the Configure
line
  sh Configure -des -Dusedevel -Doptimize="-g"
  test -f config.sh || exit 125
  # Correct makefile for newer GNU gcc
  perl -ni -we 'print unless /<(?:built-in|command)/' makefile x2p/makefile
  # if you just need miniperl, replace test_prep with miniperl
  make -j4 test_prep
  -x ./perl || exit 125
  ./perl -Ilib ~/testcase.pl
  ret=$?
  git clean -dxf
  exit $ret
```

This script may return `125` to indicate that the corresponding commit should be skipped. Otherwise, it returns the status of *~/testcase.pl*.

We first enter in bisect mode with:

```
  % git bisect start
```

For example, if the bug is present on `HEAD` but wasn't in 5.10.0, `git` will learn about this when you enter:

```
  % git bisect bad
  % git bisect good perl-5.10.0
  Bisecting: 853 revisions left to test after this
```

This results in checking out the median commit between `HEAD` and `perl-5.10.0`. We can then run the bisecting process with:

```
  % git bisect run ~/run
```

When the first bad commit is isolated, `git bisect` will tell you so:

```
  ca4cfd28534303b82a216cfe83a1c80cbc3b9dc5 is first bad commit
  commit ca4cfd28534303b82a216cfe83a1c80cbc3b9dc5
  Author: Dave Mitchell <davem@fdisolutions.com>
  Date:   Sat Feb 9 14:56:23 2008 +0000

      [perl #49472] Attributes + Unknown Error
      ...

  bisect run success
```

You can peek into the bisecting process with `git bisect log` and `git bisect visualize`. `git bisect reset` will get you out of bisect mode.

Please note that the first `good` state must be an ancestor of the first `bad` state. If you want to search for the commit that *solved* some bug, you have to negate your test case (i.e. exit with `1` if OK and `0` if not) and still mark the lower bound as `good` and the upper as `bad`. The "first bad commit" has then to be understood as the "first commit where the bug is solved".

`git help bisect` has much more information on how you can tweak your binary searches.

SUBMITTING A PATCH VIA GITHUB

GitHub is a website that makes it easy to fork and publish projects with Git. First you should set up a GitHub account and log in.

Perl's git repository is mirrored on GitHub at this page:

```
  http://github.com/github/perl/tree/blead
```

Visit the page and click the "fork" button. This clones the Perl git repository for you and provides you with "Your Clone URL" from which you should clone:

```
  % git clone git@github.com:USERNAME/perl.git perl-github
```

We shall make the same patch as above, creating a new branch:

```
  % cd perl-github
  % git remote add upstream git://github.com/github/perl.git
  % git pull upstream blead
  % git checkout -b orange
  % perl -pi -e 's{Leon Brocard}{Orange Brocard}' AUTHORS
  % git commit -a -m 'Rename Leon Brocard to Orange Brocard'
  % git push origin orange
```

The orange branch has been pushed to GitHub, so you should now send an email to perl5-porters@perl.org with a description of your changes and the following information:

```
  http://github.com/USERNAME/perl/tree/orange
  git@github.com:USERNAME/perl.git branch orange
```

MERGING FROM A BRANCH VIA GITHUB

If someone has provided a branch via GitHub and you are a committer, you should use the following in your perl-ssh directory:

```
  % git remote add dandv git://github.com/dandv/perl.git
  % git fetch
```

Now you can see the differences between the branch and blead:

```
  % git diff dandv/blead
```

And you can see the commits:

```
  % git log dandv/blead
```

If you approve of a specific commit, you can cherry pick it:

```
  % git cherry-pick 3adac458cb1c1d41af47fc66e67b49c8dec2323f
```

Or you could just merge the whole branch if you like it all:

```
  % git merge dandv/blead
```

And then push back to the repository:

```
  % git push
```

COMMITTING TO MAINTENANCE VERSIONS

Maintenance versions should only be altered to add critical bug fixes.

To commit to a maintenance version of perl, you need to create a local tracking branch:

```
  % git checkout --track -b maint-5.005 origin/maint-5.005
```

This creates a local branch named `maint-5.005`, which tracks the remote branch `origin/maint-5.005`. Then you can pull, commit, merge and push as before.

You can also cherry-pick commits from blead and another branch, by using the `git cherry-pick` command. It is recommended to use the **-x** option to `git cherry-pick` in order to record the SHA1 of the original commit in the new commit message.

SEE ALSO

The git documentation, accessible via `git help command`.