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/mirrors/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 at least 4x 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/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 the URL for the origin remote to enable pushing. To do that edit *.git/config* with git-config(1) like this:

```
% git config remote.origin.url ssh://perl5.git.perl.org/perl.git
```

You can also set up your user name and e-mail address. Most people do this once globally in their *~/.gitconfig* by doing something like:

```
% git config --global user.name "Ã†var ArnfjÃ¶rÃ° Bjarmason"
```

```
% git config --global user.email avarab@gmail.com
```

However if you'd like to override that just for perl then execute then execute something like the following in *perl-git*:

```
% git config user.email avar@cpan.org
```

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`.

A note on camel and dromedary

The committers have SSH access to the two servers that serve `perl5.git.perl.org`. One is `perl5.git.perl.org` itself (*camel*), which is the 'master' repository. The second one is `users.perl5.git.perl.org` (*dromedary*), 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 in /srv/CPAN, please use this. To share files with the general public, dromedary serves your ~/public_html/ as `http://users.perl5.git.perl.org/~yourlogin/`

These hosts have fairly strict firewalls to the outside. Outgoing, only rsync, ssh and git are allowed. For http and ftp, you can use http://webproxy:3128 as proxy. Incoming, the firewall tries to detect attacks and blocks IP addresses with suspicious activity. This sometimes (but very rarely) has false positives and you might get blocked. The quickest way to get unblocked is to notify the admins.

These two boxes are owned, hosted, and operated by booking.com. You can reach the sysadmins in #p5p on irc.perl.org or via mail to `perl5-porters@perl.org`

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 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 that you'd like to commit all the changes you've just made as a a single atomic unit, run this command:

```
   % git commit -a
```

(That `-a` tells git to add every file you've changed to this commit. New files aren't automatically added to your commit when you use `commit -a` If you want to add files or to commit some, but not all of your changes, have a look at the documentation for `git add`.)

Git will start up your favorite text editor, so that you can craft a commit message for your change. See *Commit message* below for more information about what makes a good commit message.

Once you've finished writing your commit message and exited your editor, git will write your change to disk and tell you something like this:

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

If you re-run `git status`, you should 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
```

Creating a topic branch makes it easier for the maintainers to rebase or merge back into the master blead for a more linear history. If you don't work on a topic branch the maintainer has to manually cherry pick your changes onto blead before they can be applied.

That'll get you scolded on perl5-porters, so don't do that. Be Awesome.

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 -M origin..
0001-Rename-Leon-Brocard-to-Orange-Brocard.patch
```

You should now send an email to to *perlbug@perl.org* with a description of your changes, and include this patch file as an attachment. In addition to being tracked by RT, mail to perlbug will automatically be forwarded to perl5-porters. You should only send patches to *perl5-porters@perl.org* directly if the patch is not ready to be applied, but intended for discussion.

See the next section for how to configure and use git to send these emails for you.

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.
```

Using git to send patch emails

In your ~/git/perl repository, set the destination email to perl's bug tracker:

```
$ git config sendemail.to perlbug@perl.org
```

Or maybe perl5-porters (discussed above):

```
$ git config sendemail.to perl5-porters@perl.org
```

Then you can use git directly to send your patch emails:

```
$ git send-email 0001-Rename-Leon-Brocard-to-Orange-Brocard.patch
```

You may need to set some configuration variables for your particular email service provider. For example, to set your global git config to send email via a gmail account:

```
$ git config --global sendemail.smtpserver smtp.gmail.com
$ git config --global sendemail.smtpssl 1
$ git config --global sendemail.smtpuser YOURUSERNAME@gmail.com
```

With this configuration, you will be prompted for your gmail password when you run 'git send-email'. You can also configure `sendemail.smtppass` with your password if you don't care about having your password in the .gitconfig file.

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`.

As a special case, several files are regenerated by 'make regen' if your patch alters `embed.fnc`. These are needed for compilation, but are included in the distribution so that you can build perl without needing another perl to generate the files. You must test with these regenerated files, but it is preferred that you instead note that 'make regen is needed' in both the email and the commit message, and submit your patch without them. If you're submitting a series of patches, it might be best to submit the regenerated changes immediately after the source-changes that caused them, so as to have as little effect as possible on the bisectability of your patchset.

Getting your patch accepted

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

Commit message

As you craft each patch you intend to submit to the Perl core, it's important to write a good commit message.

The first line of the commit message should be a short description and should skip the full stop. It should be no longer than the subject line of an E-Mail, 50 characters being a good rule of thumb.

A lot of Git tools (Gitweb, GitHub, git log --pretty=oneline, ..) will only display the first line (cut off at 50 characters) when presenting commit summaries.

The commit message should include description of the problem that the patch corrects or new functionality that the patch adds.

As a general rule of thumb, your commit message should let a programmer with a reasonable familiarity with the Perl core quickly understand what you were trying to do, how you were trying to do it and why the change matters to Perl.

What

Your commit message should describe what part of the Perl core you're changing and what you expect your patch to do.

Why

Perhaps most importantly, your commit message should describe why the change you are making is important. When someone looks at your change in six months or six years, your intent should be clear. If you're deprecating a feature with the intent of later simplifying another bit of code, say so. If you're fixing a performance problem or adding a new feature to support some other bit of the core, mention that.

How

While it's not necessary for documentation changes, new tests or trivial patches, it's often worth explaining how your change works. Even if it's clear to you today, it may

not be clear to a porter next month or next year.

A commit message isn't intended to take the place of comments in your code. Commit messages should describe the change you made, while code comments should describe the current state of the code. If you've just implemented a new feature, complete with doc, tests and well-commented code, a brief commit message will often suffice. If, however, you've just changed a single character deep in the parser or lexer, you might need to write a small novel to ensure that future readers understand what you did and why you did it.

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

If your patch changes code (rather than just changing documentation) you should also include one or more test cases which illustrate the bug you're fixing or validate the new functionality you're adding. In general, you should update an existing test file rather than create a new one.

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. You 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
# if Encode is not needed for the test, you can speed up the bisect by
# excluding it from the runs with -Dnoextensions=Encode
```

```
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=$?
[ $ret -gt 127 ] && ret=127
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*.

You 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`. You 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/mirrors/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
```

The same patch as above, using github might look like this:

```
% cd perl-github
% git remote add upstream git://perl5.git.perl.org/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 (see *Submitting a patch*) 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 dandv
```

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
```

Topic branches and rewriting history

Individual committers should create topic branches under **yourname/some_descriptive_name**. Other committers should check with a topic branch's creator before making any change to it.

The simplest way to create a remote topic branch that works on all versions of git is to push the current head as a new branch on the remote, then check it out locally:

```
$ branch="$yourname/$some_descriptive_name"
$ git push origin HEAD:$branch
$ git checkout -b $branch origin/$branch
```

Users of git 1.7 or newer can do it in a more obvious manner:

```
$ branch="$yourname/$some_descriptive_name"
$ git checkout -b $branch
$ git push origin -u $branch
```

If you are not the creator of **yourname/some_descriptive_name**, you might sometimes find that the original author has edited the branch's history. There are lots of good reasons for this. Sometimes, an author might simply be rebasing the branch onto a newer source point. Sometimes, an author might have found an error in an early commit which they wanted to fix before merging the branch to blead.

Currently the master repository is configured to forbid non-fast-forward merges. This means that the branches within can not be rebased and pushed as a single step.

The only way you will ever be allowed to rebase or modify the history of a pushed branch is to delete it and push it as a new branch under the same name. Please think carefully about doing this. It may be better to sequentially rename your branches so that it is easier for others working with you to cherry-pick their local changes onto the new version. (XXX: needs explanation).

If you want to rebase a personal topic branch, you will have to delete your existing topic branch and push as a new version of it. You can do this via the following formula (see the explanation about `refspec`'s in the git push documentation for details) after you have rebased your branch:

```
# first rebase
$ git checkout $user/$topic
$ git fetch
$ git rebase origin/blead

# then "delete-and-push"
$ git push origin :$user/$topic
$ git push origin $user/$topic
```

NOTE: it is forbidden at the repository level to delete any of the "primary" branches. That is any branch matching `m!^(blead|maint|perl)!`. Any attempt to do so will result in git producing an error like this:

```
$ git push origin :blead
*** It is forbidden to delete blead/maint branches in this repository
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/blead
To ssh://perl5.git.perl.org/perl
 ! [remote rejected] blead (hook declined)
 error: failed to push some refs to 'ssh://perl5.git.perl.org/perl'
```

As a matter of policy we do **not** edit the history of the blead and maint-* branches. If a typo (or worse) sneaks into a commit to blead or maint-*, we'll fix it in another commit. The only types of updates

allowed on these branches are "fast-forward's", where all history is preserved.

Annotated tags in the canonical perl.git repository will never be deleted or modified. Think long and hard about whether you want to push a local tag to perl.git before doing so. (Pushing unannotated tags is not allowed.)

Committing to maintenance versions

Maintenance versions should only be altered to add critical bug fixes, see *perlpolicy*.

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.

Grafts

The perl history contains one mistake which was not caught in the conversion: a merge was recorded in the history between blead and maint-5.10 where no merge actually occurred. Due to the nature of git, this is now impossible to fix in the public repository. You can remove this mis-merge locally by adding the following line to your `.git/info/grafts` file:

```
296f12bbbbaa06de9be9d09d3dcf8f4528898a49
434946e0cb7a32589ed92d18008aaa1d88515930
```

It is particularly important to have this graft line if any bisecting is done in the area of the "merge" in question.

SEE ALSO

- The git documentation, accessible via the `git help` command
- *perlpolicy* - Perl core development policy