1.5.0

GIT v1.5.0 Release Notes ==========

Old news --------

This section is for people who are upgrading from ancient versions of git. Although all of the changes in this section happened before the current v1.4.4 release, they are summarized here in the v1.5.0 release notes for people who skipped earlier versions.

As of git v1.5.0 there are some optional features that changes the repository to allow data to be stored and transferred more efficiently. These features are not enabled by default, as they will make the repository unusable with older versions of git. Specifically, the available options are:

- There is a configuration variable core.legacyheaders that changes the format of loose objects so that they are more efficient to pack and to send out of the repository over git native protocol, since v1.4.2. However, loose objects written in the new format cannot be read by git older than that version; people fetching from your repository using older clients over dumb transports (e.g. http) using older versions of git will also be affected.

```
To let git use the new loose object format, you have to
set core.legacyheaders to false.
```

- Since v1.4.3, configuration repack.usedeltabaseoffset allows packfile to be created in more space efficient format, which cannot be read by git older than that version.

```
To let git use the new format for packfiles, you have to
set repack.usedeltabaseoffset to true.
```

The above two new features are not enabled by default and you have to explicitly ask for them, because they make repositories unreadable by older versions of git, and in v1.5.0 we still do not enable them by default for the same reason. We will change this default probably 1 year after 1.4.2’s release, when it is reasonable to expect everybody to have new enough version of git.

- ’git pack-refs’ appeared in v1.4.4; this command allows tags to be accessed much more efficiently than the traditional ’one-file-per-tag’ format. Older git-native clients can still fetch from a repository that packed and pruned refs (the server side needs to run the up-to-date version of git), but older dumb transports cannot. Packing of refs is done by an explicit user action, either by use of "git pack-refs --prune" command or by use of "git gc" command.

- ’git -p’ to paginate anything — many commands do pagination by default on a tty. Introduced between v1.4.1 and v1.4.2; this may surprise old timers.
- ’git archive’ superseded ’git tar-tree’ in v1.4.3;
- ’git cvsserver’ was new invention in v1.3.0;
- ’git repo-config’, ’git grep’, ’git rebase’ and ’gitk’ were seriously enhanced during v1.4.0 timeperiod.
- ’gitweb’ became part of git.git during v1.4.0 timeperiod and seriously modified since then.
- reflog is an v1.4.0 invention. This allows you to name a revision that a branch used to be at (e.g. "git diff master@{yesterday} master" allows you to see changes since yesterday’s tip of the branch).

Updates in v1.5.0 since v1.4.4 series ----------

- Index manipulation
- git-add is to add contents to the index (aka "staging area" for the next commit), whether the file the contents happen to be is an existing one or a newly created one.
- git-add without any argument does not add everything anymore. Use ’git-add .’ instead. Also you can add otherwise ignored files with an -f option.
- git-add tries to be more friendly to users by offering an interactive mode ("git-add -i").
- git-commit <path> used to refuse to commit if <path> was different between HEAD and the index (i.e. update-index was used on it earlier). This check was removed.
- git-rm is much saner and safer. It is used to remove paths from both the index file and the working tree, and makes sure you are not losing any local modification before doing so.
- git-reset <tree> <paths>… can be used to revert index entries for selected paths.
- git-update-index is much less visible. Many suggestions to use the command in git output and documentation have now been replaced by simpler commands such as "git add" or "git rm".
- Repository layout and objects transfer
- The data for origin repository is stored in the configuration file $GIT_DIR/config, not in $GIT_DIR/remotes/, for newly created clones. The latter is still supported and there is no need to convert your existing repository if you are already comfortable with your workflow with the layout.

- git-clone always uses what is known as "separate remote" layout for a newly created repository with a working tree.

```
A repository with the separate remote layout starts with only
one default branch, 'master', to be used for your own
development.  Unlike the traditional layout that copied all
the upstream branches into your branch namespace (while
renaming their 'master' to your 'origin'), the new layout
puts upstream branches into local "remote-tracking branches"
with their own namespace. These can be referenced with names
such as "origin/$upstream_branch_name" and are stored in
.git/refs/remotes rather than .git/refs/heads where normal
branches are stored.

This layout keeps your own branch namespace less cluttered,
avoids name collision with your upstream, makes it possible
to automatically track new branches created at the remote
after you clone from it, and makes it easier to interact with
more than one remote repository (you can use "git remote" to
add other repositories to track).  There might be some
surprises:
```

- ’git branch’ does not show the remote tracking branches. It only lists your own branches. Use ’-r’ option to view the tracking branches.

- If you are forking off of a branch obtained from the upstream, you would have done something like ’git branch my-next next’, because traditional layout dropped the tracking branch ’next’ into your own branch namespace. With the separate remote layout, you say ’git branch next origin/next’, which allows you to use the matching name ’next’ for your own branch. It also allows you to track a remote other than ’origin’ (i.e. where you initially cloned from) and fork off of a branch from there the same way (e.g. "git branch mingw j6t/master").

```
Repositories initialized with the traditional layout continue
to work.
```

- New branches that appear on the origin side after a clone is made are also tracked automatically. This is done with an wildcard refspec "refs/heads/**:refs/remotes/origin/**", which older git does not understand, so if you clone with 1.5.0, you would need to downgrade remote.*.fetch in the configuration file to specify each branch you are interested in individually if you plan to fetch into the repository with older versions of git (but why would you?).

- Similarly, wildcard refspec "refs/heads/**:refs/remotes/me/**" can be given to "git-push" command to update the tracking branches that is used to track the repository you are pushing from on the remote side.

- git-branch and git-show-branch know remote tracking branches (use the command line switch "-r" to list only tracked branches).

- git-push can now be used to delete a remote branch or a tag. This requires the updated git on the remote side (use "git push <remote> :refs/heads/<branch>" to delete "branch").

- git-push more aggressively keeps the transferred objects packed. Earlier we recommended to monitor amount of loose objects and repack regularly, but you should repack when you accumulated too many small packs this way as well. Updated git-count-objects helps you with this.

- git-fetch also more aggressively keeps the transferred objects packed. This behavior of git-push and git-fetch can be tweaked with a single configuration transfer.unpacklimit (but usually there should not be any need for a user to tweak it).

- A new command, git-remote, can help you manage your remote tracking branch definitions.

- You may need to specify explicit paths for upload-pack and/or receive-pack due to your ssh daemon configuration on the other end. This can now be done via remote. **.uploadpack and remote.**.receivepack configuration.

- Bare repositories

- Certain commands change their behavior in a bare repository (i.e. a repository without associated working tree). We use a fairly conservative heuristic (if $GIT_DIR is ".git", or ends with "/.git", the repository is not bare) to decide if a repository is bare, but "core.bare" configuration variable can be used to override the heuristic when it misidentifies your repository.

- git-fetch used to complain updating the current branch but this is now allowed for a bare repository. So is the use of ’git-branch -f’ to update the current branch.

- Porcelain-ish commands that require a working tree refuses to work in a bare repository.

- Reflog

- Reflog records the history from the view point of the local repository. In other words, regardless of the real history, the reflog shows the history as seen by one particular repository (this enables you to ask "what was the current revision in *this* repository, yesterday at 1pm?"). This facility is enabled by default for repositories with working trees, and can be accessed with the "branch@{time}" and "branch@{Nth}" notation.

- "git show-branch" learned showing the reflog data with the new -g option. "git log" has -g option to view reflog entries in a more verbose manner.

- git-branch knows how to rename branches and moves existing reflog data from the old branch to the new one.

- In addition to the reflog support in v1.4.4 series, HEAD reference maintains its own log. "HEAD@{5.minutes.ago}" means the commit you were at 5 minutes ago, which takes branch switching into account. If you want to know where the tip of your current branch was at 5 minutes ago, you need to explicitly say its name (e.g. "master@{5.minutes.ago}") or omit the refname altogether i.e. "@{5.minutes.ago}".

- The commits referred to by reflog entries are now protected against pruning. The new command "git reflog expire" can be used to truncate older reflog entries and entries that refer to commits that have been pruned away previously with older versions of git.

```
Existing repositories that have been using reflog may get
complaints from fsck-objects and may not be able to run
git-repack, if you had run git-prune from older git; please
run "git reflog expire --stale-fix --all" first to remove
reflog entries that refer to commits that are no longer in
the repository when that happens.
```

- Cruft removal

- We used to say "old commits are retrievable using reflog and ’master@{yesterday}’ syntax as long as you haven’t run git-prune". We no longer have to say the latter half of the above sentence, as git-prune does not remove things reachable from reflog entries.

- There is a toplevel garbage collector script, ’git-gc’, that runs periodic cleanup functions, including ’git-repack -a -d’, ’git-reflog expire’, ’git-pack-refs --prune’, and ’git-rerere gc’.

- The output from fsck ("fsck-objects" is called just "fsck" now, but the old name continues to work) was needlessly alarming in that it warned missing objects that are reachable only from dangling objects. This has been corrected and the output is much more useful.

- Detached HEAD

- You can use ’git-checkout’ to check out an arbitrary revision or a tag as well, instead of named branches. This will dissociate your HEAD from the branch you are currently on.

```
A typical use of this feature is to "look around".  E.g.


$ git checkout v2.6.16
... compile, test, etc.
```

```
$ git checkout v2.6.17
... compile, test, etc.
```

- After detaching your HEAD, you can go back to an existing branch with usual "git checkout $branch". Also you can start a new branch using "git checkout -b $newbranch" to start a new branch at that commit.
- You can even pull from other repositories, make merges and commits while your HEAD is detached. Also you can use "git reset" to jump to arbitrary commit, while still keeping your HEAD detached.

```
Remember that a detached state is volatile, i.e. it will be forgotten
as soon as you move away from it with the checkout or reset command,
unless a branch is created from it as mentioned above.  It is also
possible to rescue a lost detached state from the HEAD reflog.
```

- Packed refs
- Repositories with hundreds of tags have been paying large overhead, both in storage and in runtime, due to the traditional one-ref-per-file format. A new command, git-pack-refs, can be used to "pack" them in more efficient representation (you can let git-gc do this for you).
- Clones and fetches over dumb transports are now aware of packed refs and can download from repositories that use them.
- Configuration
- configuration related to color setting are consolidated under color.* namespace (older diff.color.**, status.color.** are still supported).
- ’git-repo-config’ command is accessible as ’git-config’ now.
- Updated features
- git-describe uses better criteria to pick a base ref. It used to pick the one with the newest timestamp, but now it picks the one that is topologically the closest (that is, among ancestors of commit C, the ref T that has the shortest output from "git-rev-list T..C" is chosen).
- git-describe gives the number of commits since the base ref between the refname and the hash suffix. E.g. the commit one before v2.6.20-rc6 in the kernel repository is:

```
v2.6.20-rc5-306-ga21b069
```

```
which tells you that its object name begins with a21b069,
v2.6.20-rc5 is an ancestor of it (meaning, the commit
contains everything -rc5 has), and there are 306 commits
since v2.6.20-rc5.
```

- git-describe with --abbrev=0 can be used to show only the name of the base ref.
- git-blame learned a new option, --incremental, that tells it to output the blames as they are assigned. A sample script to use it is also included as contrib/blameview.
- git-blame starts annotating from the working tree by default.
- Less external dependency
- We no longer require the "merge" program from the RCS suite. All 3-way file-level merges are now done internally.
- The original implementation of git-merge-recursive which was in Python has been removed; we have a C implementation of it now.
- git-shortlog is no longer a Perl script. It no longer requires output piped from git-log; it can accept revision parameters directly on the command line.
- I18n
- We have always encouraged the commit message to be encoded in UTF-8, but the users are allowed to use legacy encoding as appropriate for their projects. This will continue to be the case. However, a non UTF-8 commit encoding *must* be explicitly set with i18n.commitencoding in the repository where a commit is made; otherwise git-commit-tree will complain if the log message does not look like a valid UTF-8 string.
- The value of i18n.commitencoding in the originating repository is recorded in the commit object on the "encoding" header, if it is not UTF-8. git-log and friends notice this, and re-encodes the message to the log output encoding when displaying, if they are different. The log output encoding is determined by "git log --encoding=<encoding>", i18n.logoutputencoding configuration, or i18n.commitencoding configuration, in the decreasing order of preference, and defaults to UTF-8.
- Tools for e-mailed patch application now default to -u behavior; i.e. it always re-codes from the e-mailed encoding to the encoding specified with i18n.commitencoding. This unfortunately forces projects that have happily been using a legacy encoding without setting i18n.commitencoding to set the configuration, but taken with other improvement, please excuse us for this very minor one-time inconvenience.
- e-mailed patches

- See the above I18n section.
- git-format-patch now enables --binary without being asked. git-am does *not* default to it, as sending binary patch via e-mail is unusual and is harder to review than textual patches and it is prudent to require the person who is applying the patch to explicitly ask for it.
- The default suffix for git-format-patch output is now ".patch", not ".txt". This can be changed with --suffix=.txt option, or setting the config variable "format.suffix" to ".txt".
- Foreign SCM interfaces
- git-svn now requires the Perl SVN
  libraries, the command-line backend was too slow and limited.
- the ’commit’ subcommand of git-svn has been renamed to ’set-tree’, and ’dcommit’ is the recommended replacement for day-to-day work.
- git fast-import backend.
- User support
- Quite a lot of documentation updates.
- Bash completion scripts have been updated heavily.
- Better error messages for often used Porcelainish commands.
- Git GUI. This is a simple Tk based graphical interface for common Git operations.
- Sliding mmap
- We used to assume that we can mmap the whole packfile while in use, but with a large project this consumes huge virtual memory space and truly huge ones would not fit in the userland address space on 32-bit platforms. We now mmap huge packfile in pieces to avoid this problem.
- Shallow clones
- There is a partial support for ’shallow’ repositories that keeps only recent history. A ’shallow clone’ is created by specifying how deep that truncated history should be (e.g. "git clone --depth 5 git://some.where/repo.git").

```
Currently a shallow repository has number of limitations:
```

- Cloning and fetching *from* a shallow clone are not supported (nor tested — so they might work by accident but they are not expected to).
- Pushing from nor into a shallow clone are not expected to work.

- Merging inside a shallow repository would work as long as a merge base is found in the recent history, but otherwise it will be like merging unrelated histories and may result in huge conflicts.

```
but this would be more than adequate for people who want to
look at near the tip of a big project with a deep history and
send patches in e-mail format.
```

□□□ #2
□ □□□ □□□ 2026-10-31 19:45:17 UTC
□ □□□ □□□ 2026-10-31 19:45:19 UTC