An Introduction to (Easy) Git

Elijah Newren

Trivial changes for most developers

You may often hear that Git requires drastically different workflows.

Trivial changes for most developers

You may often hear that Git requires drastically different workflows.

While Git *allows* drastically different workflows, and some people promote them...

Trivial changes for most developers

You may often hear that Git requires drastically different workflows.

While Git *allows* drastically different workflows, and some people promote them...

The change from Subversion to (Easy) Git can be easier than the transition from CVS to Subversion.

Caching more data

Project checkouts contain

- CVS: data + metadata
- SVN: data + extra copy + metadata
  (faster diffs against last revision)
- GIT: data + all history + metadata
  (most operations faster, new operations possible)

Caching more data

Project checkouts contain

- CVS: data + metadata
- SVN: data + extra copy + metadata
  (faster diffs against last revision)
- GIT: data + all history + metadata
  (most operations faster, new operations possible)

Client disk data usage, relative to size of most recent revision:

- CVS: 1+ (typical is about 1.2)
- SVN: 2+ (typical is about 2.2)
- GIT: 1+ (typical is in the range 1.9-2.5)

Improved data integrity

Git has very strong safeguards against corruption, whether accidental (e.g. disk/memory/cpu failure) or malicious. It is not possible to change published revisions without being noticed.

Git achieves this through tracking cryptographic checksums of files, subtrees, trees, and commits.

The basics are essentially the same

Core commands

svn checkout URL	eg clone URL
svn status	eg status
svn update	eg update
svn diff	eg diff
svn add FILE	eg add FILE
svn commit	eg commit eg push

Other common commands

svn blame FILE	eg blame FILE
svn cat FILE	eg cat FILE
svn help [COMMAND]	eg help [COMMAND]
svn info	eg info
svn mv OLDNAME NEWNAME	eg mv OLDNAME NEWNAME
svn resolved PATH...	eg resolved PATH...
svn revert PATH...	eg revert PATH...
svn rm FILE...	eg rm FILE...

...unless you make policy decisions otherwise

Git has a *lot* of extra capabilities that you can use without changing the basic model.

If you want to adopt a linux-like development model, you can.

But most projects don't adopt such practices, at least not at first (and often not ever.)

...unless you make policy decisions otherwise

Git has a *lot* of extra capabilities that you can use without changing the basic model.

If you want to adopt a linux-like development model, you can.

But most projects don't adopt such practices, at least not at first (and often not ever.)

...unless you make policy decisions otherwise

Git has a *lot* of extra capabilities that you can use without changing the basic model.

If you want to adopt a linux-like development model, you can.

But most projects don't adopt such practices, at least not at first (and often not ever.)

Basic Commands

Clone

The clone command obtains a copy of a project for you (thus making it analagous to svn checkout.)

```
$ eg clone /home/newren/floss/gtk+-git
Initialized empty Git repository in /home/newren/devel/gtk+-git/.git
Checking out files:  100% (2691/2691), done.
```

Valid URLs are project directory names accessed by various protocols; examples:

- eg clone newren@work-machine:/home/coworker/project
- eg clone git://git.samba.org/samba.git
- eg clone http://git.gitorious.org/eg/mainline.git eg
- eg clone /PATH/TO/PROJECT NEWNAME

Commit

The commit command records changes locally.

```
$ eg commit -m "Random change, just for the fun of it"
Created commit 2ebb10a:  Random change, just for the fun of it
1 files changed, 1 insertions(+), 0 deletions(-)
```

To push this commit to the repository you cloned from, run

```
$ eg push
```

You can queue multiple commits before pushing, which allows checkpointing code that is not ready for everyone else in a more fine-grained fashion.

Status

The status commands shows which branch is active, and lists files according to their status (modified, unmerged (“has conflicts”), deleted, unknown, etc.)

```
$ eg status
(On branch master)
Changed but not updated ("unstaged"):
        modified:   doc/how-to-get-focus-right.txt
        modified:   src/core/window.c
        unmerged:   NEWS
        deleted:    src/tools/metacity-message.c
Unknown files:
        .gitignore
        build/
        green-chili-is-yucky
        notes.txt
```

Diff

The diff command shows changes in patch format.

```
$ eg diff
diff --git a/src/utils.py b/src/utils.py
index 2ad4d53..c76b540 100644
--- a/src/utils.py
+++ b/src/utils.py
@@ -454,4 +454,5 @@ class Task(object):
     return value

   def add(self, item, count):
-    self.container.insert(item, count)
+    if count:
+      self.container.insert(item, count)
```

Log

The log command shows the history of changes.

```
$ eg log

commit 23dbb9a7643186c1402709e535622595e9b857a1 (master)
Author: Elijah Newren <newren@gmail.com>
Date:   Fri Oct 3 20:35:26 2008 -0600

    Mark the current version of eg as .93; it's time to release

    If you look at .93 upside down, it kind of looks like E-G.  :-)

commit 3f767870a7c70ba518217dac4e5ed6738176c783 (master~1)
Author: Elijah Newren <newren@gmail.com>
Date:   Thu Oct 2  21:56:20 2008 -0600

    Fix weird bug when sh != bash: caret needs to be quoted on Sun machines

    The command
      git branch | sed -e s/^..//
    was failing which caused some nasty messages and warnings on Sun
    machines.  I don't know why sh doesn't like this command, but quoting
    the substitution command avoids the problem.
```

It has been said that few users make use of more than 20% of the features in a word processor.

It has been said that few users make use of more than 20% of the features in a word processor.

I probably use less than 1%.

It has been said that few users make use of more than 20% of the features in a word processor.

I probably use less than 1%.

But it is nice to know the other features are there.

Git has a lot of features.

Git has a lot of features.

You don’t need to use or even understand 1% of them.

Git has a lot of features.

You don’t need to use or even understand 1% of them.

But here’s a brief introduction to what’s possible...

Time savers

Stashing

The stash command saves any uncommitted changes in your project, and returns you to a clean slate. The stash command can also be used to reapply previously stashed away changes.

```
$ eg stash
<Do a bunch of other stuff, even including making commits>
$ eg stash apply
```

You can have multiple stashes and name them, if you like.

```
$ eg stash save Stuff I was doing before customer called
$ eg stash save Crazy idea
$ eg stash apply Stuff I was doing before customer called
```

Staging

In git, you can explicitly mark a subset of your changes as being ready for commit. You can also make additional changes, and then just commit the changes that are ready.

```
$ echo hi > there
$ eg stage there
$ echo "hi again" >> there
$ eg commit --staged -m "New single-line file called there"
Created commit b7e2002:  New single-line file called there
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 there
$ eg status
(On branch testing)
Changed but not updated ("unstaged"):
        modified:   there
```

Bisecting

Git provides a bisect command for finding the commit that introduced some bug through a binary search of history.

```
$ eg bisect start BAD_REVISION GOOD_REVISION
<Repeat until done:>
  <Compile, link, test, see if given revision is good or bad>
  $ eg bisect bad    ⇔    $ eg bisect good
```

You can automate that looping step with a script that compiles, links, tests, and returns whether the given revision is good:

```
$ eg bisect run NAME_OF_SCRIPT
```

Branching

Git makes both branching and merging easy, fast, and robust.

```
$ eg branch newbranch
$ eg switch newbranch
<Work on newbranch>
$ eg switch original-branch
<Work on original-branch>
```

Then merging another branch into the current one is as simple as

```
$ eg merge newbranch
```

If you want to merge changes back the other way:

```
$ eg switch newbranch
$ eg merge original-branch
```

Setting up repositories

Git makes it really easy to just stick data under version control.

```
$ eg init
$ eg add .
$ eg commit
<Make more changes>
$ eg commit

...
```

Collaboration

Pulling changes

One can pull updates or changes from the original repository you cloned from

```
$ eg pull
```

But it's just as easy to pull changes directly from another repository

```
$ eg pull --branch some-branch URL-OR-PATH-TO-PROJECT
```

If you pull from the same person often, you can add a nickname for their repository

```
$ eg remote add jim URL-OR-PATH-TO-PROJECT
$ eg pull --branch some-branch jim
```

Fetching multiple branches simultaneously

You can pull the branches in bob's repository, and stick them in branches in your repository with the name `bob/<branch>`

```
$ eg fetch bob
```

This assumes you've already set up bob as a nickname for his repository, by running:

```
$ eg remote add bob URL
```

Patch review

Git makes it easy to submit patches for review via email.

```
$ eg format-patch --numbered master..working-branch
$ eg send-email --compose --to maintainer@project.org
```

It is also easy to apply patches received in email (even some emails not formatted by git), **if** you can figure out how to get the relevant emails saved off into a separate mbox file:

```
$ eg am mbox-file
```

('am' stands for 'apply [a series of patches from] mail'.)

Creating and Using Bundles

Create a bundle in the file bundle.file that contains the whole repository

```
$ eg bundle create bundle.file
```

$\Rightarrow$

The collaborator, after somehow receiving this file, can run

```
$ eg clone \
  /path/to/bundle.file project
```

After a while, create a file to send them updates

```
$ mv bundle.file old-repo
$ eg bundle create-update \
  bundle.file old-repo
```

$\Rightarrow$

After sending them the new repo.bundle file, they stick the file in the same place and run

```
$ eg pull
```

They can also send you bundles, and you treat the filename as a repository URL to pull from.

We've just scratched the surface

Revert

You can revert the uncommitted changes to a set of files or directories

```
$ eg revert foo.txt bar.c
$ eg revert src
```

You can also revert to a prior revision

```
$ eg revert --since REVISION src
```

or revert the changes made in a prior revision

```
$ eg revert --in REVISION src
```

High level change statistics

You can get high-level statistics about the number of changes per file.

```
$ eg diff --stat
```

Or the percentage of line changes by directory

```
$ eg diff --dirstat
```

or the type of changes to each file (equivalent to cvs update output)

```
$ eg diff --name-status
```

or just the names of the files that have changed

```
$ eg diff --name-only
```

Logs with additional information

You can combine patches (diffs) with logs

```
$ eg log -p
```

Or, high level patch statistics

```
$ eg log --stat
$ eg log --shortstat
$ eg log --dirstat
$ eg log --name-status
$ eg log --name-only
```

Shorter change logs

You can get just the summary of each change message

```
$ eg log --pretty=oneline
```

Or the one-line summaries grouped by author

```
$ eg shortlog
```

You can also easily specify a range for either of these

```
$ eg shortlog gnome-2-24..master
```

Searching for changes

You can look for string or regex matches in currently checked out files

```
$ eg grep PATTERN
```

or in files of a previous revision

```
$ eg grep PATTERN REVISION
```

or in the files under a specific directory of a previous revision

```
$ eg grep PATTERN REVISION -- DIRECTORY
```

Searching for changes

You can search for when some text was introduced

```
$ eg log -S'FIXME'
```

Or when some text matching a regular expression was introduced

```
$ eg log -S'\bHACK #[0-9]+:'  --pickaxe-regex
```

Searching for changes

You can search for when some text was introduced

```
$ eg log -S'FIXME'
```

Or when some text matching a regular expression was introduced

```
$ eg log -S'\bHACK #[0-9]+:'  --pickaxe-regex
```

(It's a joke, not a regex that's supposed to be useful.)

Cleaning up tags and branches

You can push all the tags in your repository to another

```
$ eg push --all-tags
```

Then delete all the tags in your repository

```
$ eg tag -d v3.14
$ eg tag -d v3.141
$ eg tag -d v3.1415
...
```

And get all the tags back if you like

```
$ eg pull --all-tags
```

You can similarly move branches between repositories.

More advantages of staging

You can view just the changes that are staged (i.e. were explictly marked as ready to be committed)

```
$ eg diff --staged
```

or the changes that aren't marked as such

```
$ eg diff --unstaged
```

More advantages of staging

In addition to the ability to stage individual files, you can also selectively stage individual changes *within* files (by patch hunk)

```
$ eg stage -p
```

You can also split patch hunks and even edit patch hunks interactively before staging them.

Forward porting a series of patches

Say you have two branches, side-branch and main-branch, with side-branch being based on some old revision of main-branch. You'd like the patches in side-branch to be rewritten as if they were based on the current revision of main-branch. You can do so easily:

```
$ eg switch side-branch
$ eg rebase --against main-branch
```

This can be useful to keep experimental (or rejected) commits together in history, while also testing against "mainline" changes. It also makes it easier to present a completed feature for review.

Cleaning up history

You can amend the previous commit to include additional changes or to modify the log message

```
$ eg commit --amend
```

(Though this is typically a bad idea once the commit has been made public.)

Cleaning up history

You can also reorder, combine, split, or drop prior commits, or insert new ones

```
$ eg rebase --interactive --since REVISION
```

(This is typically a bad idea to do on commits that have been made public, but can be really nice for changes that are still private.)

Cleaning up history

You can undo rebases, merges, or commits by setting the tip of the branch to a different revision. (You can also redo them the same way.)

```
$ eg reset --working-copy REVISION
```

Merging independent repositories

You can merge changes from another repository that has no common history

```
$ eg pull --branch BRANCH REPOSITORY_URL
```

This preserves the exact (and separate) history of both projects, while also reflecting that the two merged at some point.

Help

Help is readily available

```
$ eg help
Creating repositories
  eg clone      Clone a repository into a new directory
  eg init       Create a new repository

Obtaining information about changes, history, & state
  eg diff       Show changes to file contents
  eg log        Show history of recorded changes
  eg status     Summarize current changes
...
```

You can also get help on a specific command

```
$ eg help bundle
bundle: Pack repository updates (or whole repository) into a file

Usage:
  eg bundle create FILENAME [REFERENCES]
  eg bundle create-update NEWFILENAME OLDFILENAME [REFERENCES]
  eg bundle verify FILENAME

Description:
  Bundle creates a file which contains a repository, or a subset thereof.
  This is useful when two machines cannot be directly connected (thus
...
```

Resources

Easy Git

- Website: `http://www.gnome.org/~newren/eg`
- Detailed comparison to subversion:
  `http://www.gnome.org/~newren/eg/git-for-svn-users.html`

Core Git

- Website: `http://git.or.cz`

That's not all that git can do...

Cherry picking a change off another branch

Sometimes, it may not make sense to merge branches. But you may want to cherry pick a change off another branch and apply it to the current branch

```
$ eg cherry-pick REVISION
```

By default, it'll make a commit with the same commit message too.

Finding where certain lines came from

You can find out which revision each line of the Foo constructor of foo.cc was introduced in

```
$ eg blame -L '/^Foo::Foo/,/^}$/' foo.cc
```

(The starting and ending lines to report on were specified by a comma-separated and slash-enclosed pair of regular expressions, i.e. `/regex1/,/regex2/` — though the whole thing was also quoted in order to avoid shell weirdness.)

You can also find out where lines 115 through 120 of bar.c came from...including which revision they were introduced in *and* which file they were in at that time. You can also ignore whitespace changes when comparing lines to see where they came from.

```
$ eg blame -L 115,120 -C -C -w bar.c
```

Which files have not changed?

You can find out things like which files have not changed in the last 100 commits

```
$ comm -13 <(git diff --name-only HEAD~100 | sort) <(git ls-files | sort)
```

(This command assumes you are using bash and the command 'comm', part of GNU coreutils, is available.)

Pull in lots of changes at once

Pull in all the branches from bob's repository. And from jim's, and from alice's, and...

```
$ eg remote update
```

(This assumes you've setup remote nicknames for each developer with `eg remote add nickname URL-for-repository`)

You can also combine remotes into groups and get updates from a specific group of developers.

Extra bisect options

When bisecting history to find the commit that introduced a bug, you can get a history of commits you've marked as good/bad

```
$ eg bisect log
```

and replay parts of this history if you realize you made a mistake

```
$ eg bisect replay file-with-edited-output-of-bisect-log
```

or see the remaining commits that need to be checked

```
$ eg bisect visualize
```

or skip unbuildable commits

```
$ eg bisect skip
```

Rewriting lots of history

Git provides a filter-branch command for when you need fine-grained control of rewriting lots of history. While the result is incompatible with the original repository, this can be handy when fixing up imports of history from other version control systems. Example types of operations that can be performed:

- Change author and commiter names or email addresses.
- Remove any trace of a certain file having been in the repository.
- Change the "beginning" of history by grafting the "initial" commit on top of another commit(s). Or adding another "beginning" of history.
- Remove all commits made by a certain author
- Make the history of a coworker's repository cyclic when they forget to lock their screen. Teach them a really tough lesson.
- Make a certain subdirectory become the toplevel directory of the repository, throwing away all files and directories that were not originally underneath it.
- For each commit, run a script that will rewrite multiple files in that commit.

And that's not all either...

The 2008 Git user's survey included 65 commands and variations thereof, which featured high-level functionality of Git. The survey asked whether (and how often) users used the various commands.

And that's not all either...

The 2008 Git user's survey included 65 commands and variations thereof, which featured high-level functionality of Git. The survey asked whether (and how often) users used the various commands.

If you used everything in this presentation, you could check about half the boxes.

And that's not all either...

The 2008 Git user's survey included 65 commands and variations thereof, which featured high-level functionality of Git. The survey asked whether (and how often) users used the various commands.

If you used everything in this presentation, you could check about half the boxes.

...but those boxes don't cover all existing capabilities either.

And that's not all either...

The 2008 Git user's survey included 65 commands and variations thereof, which featured high-level functionality of Git. The survey asked whether (and how often) users used the various commands.

If you used everything in this presentation, you could check about half the boxes.

...but those boxes don't cover all existing capabilities either.

...let alone the features being developed.