GIT Started with Community Software

Jerry Cooperstein

June 2010

Learn today. Design tomorrow.
ESC
Chicago • June 7 – 9, 2010
Donald E. Stephens Convention Center

GIT Started with Community Software

- Distributed Development
- GIT and Revision Control Systems
- GIT Design Features
- GIT Concepts
- Repositories
- Main Operations
- Main Commands
- Using GIT: an Example
- File Management
- Making Commitments

GIT Started with Community Software

- Cloning
- Branches
- Merging
- Rebasing
- Bisection
- Dealing with Patches
- Further Training

Distributed Development

- Many developers, working separately
- No structural authoritative central repository
- Every repository is as authoritative is any other
  - Each contains entire project history
- Peer-to-peer in nature
- The influence of the main project maintainer is social and political, not technical
- No preferred model for organization
  - Can be top down or very flat
  - Can be pyramidal or egalitarian
- GIT is a tool, not a rigid method

GIT and Revision Control Systems

- Long history of Revision/Source Control Systems
- RCS/SCCS:
  - Repository kept alongside working directory
  - Can't submit changes while files are checked out
- CVS/Subversion:
  - Central repository
  - Multiple users, network capabilities
  - Simultaneous commits
- Other open source projects:
  - Arch, Monotone, Mercurial, PRCS
- Bitkeeper
  - Used in Linux kernel until licensing problems developed
- Many commercial products

GIT Design Features

- Based on distributed development
- Scales to large numbers of users
- High speed and maximum efficiency
- Builds in strong trust and integrity
- All changes are documented and accountable
- Immutable data kept in the repository, such as history
- Transactions done atomically
- Branching and merging with parallel lines of development
- Repository independence
- Free, unencumbered license (GPL V2)

GIT Concepts

- A File is not an essential concept
- Two important data structures maintained:
  - Object Store: set of discrete binary objects containing the project guts
  - Index: binary file that contains overall project structure, changing with time
- Types of objects in the object store:
  - Blobs: Binary Large OBject
  - Trees: blob identifiers, pathnames, file metadata
  - Commits: metadata describing changes
  - Tags: human friendly names to describe stages
- Index file maintains changes until they are committed

Repositories

- Configuration information such as names, email of authors, which is not carried forth by ***clone*** operation
- All repositories are equal
- The **repository** is a database containing information to:
  - Manage revisions
  - Display history
  - Store a project
- Contains a copy of the repository itself

Main Operations

- Creating or Cloning
- Checkout
- Adding
- Committing
- Tagging
- Diffing
- Showing history
- Pulling and Pushing
- Patching
- Bisection
- Branching
- Merging
- Rebasing

Main Commands

```
usage: git [--version] [--exec-path[=GIT_EXEC_PATH]] [-p|--paginate|--no-pager] [--
bare] [--git-dir=GIT_DIR] [--work-tree=GIT_WORK_TREE] [--help] COMMAND [ARGS]

The most commonly used git commands are:
   add         Add file contents to the index
   bisect      Find the change that introduced a bug by binary search
   branch      List, create, or delete branches
   checkout    Checkout a branch or paths to the working tree
   clone       Clone a repository into a new directory
   commit      Record changes to the repository
   diff        Show changes between commits, commit and working tree, etc
   fetch       Download objects and refs from another repository
   grep        Print lines matching a pattern
   init        Create an empty git repository or reinitialize an existing one
   log         Show commit logs
   merge       Join two or more development histories together
   mv          Move or rename a file, a directory, or a symlink
   pull        Fetch from and merge with another repository or a local branch
   push        Update remote refs along with associated objects
   rebase      Forward-port local commits to the updated upstream head
   reset       Reset current HEAD to the specified state
   rm          Remove files from the working tree and from the index
   show        Show various types of objects
   status      Show the working tree status
   tag         Create, list, delete or verify a tag object signed with GPG
```

An Example: I

- **Creating a local project**

```
$ mkdir git-test
$ cd git-test
$ git init


$ ls -l .git
total 40
drwxrwxr-x 7 coop coop 4096 Dec 30 13:59 ./
drwxrwxr-x 3 coop coop 4096 Dec 30 13:59 ../
drwxrwxr-x 2 coop coop 4096 Dec 30 13:59 branches/
-rw-rw-r-- 1 coop coop   92 Dec 30 13:59 config
-rw-rw-r-- 1 coop coop   58 Dec 30 13:59 description
-rw-rw-r-- 1 coop coop   23 Dec 30 13:59 HEAD
drwxrwxr-x 2 coop coop 4096 Dec 30 13:59 hooks/
drwxrwxr-x 2 coop coop 4096 Dec 30 13:59 info/
drwxrwxr-x 4 coop coop 4096 Dec 30 13:59 objects/
drwxrwxr-x 4 coop coop 4096 Dec 30 13:59 refs/
```

An Example: II

- **Adding a file to the project**

```
$ echo some junk > somejunkfile
$ git add somejunkfile

$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file: somejunkfile
#
```

- **Make a modification**

```
$ echo another line >> somejunkfile
$ git diff
diff --git a/somejunkfile b/somejunkfile
index 9638122..6023331 100644
--- a/somejunkfile
+++ b/somejunkfile
@@ -1 +1,2 @@
 some junk
+another line
```

An Example: III

- **Commit the changes**

```
$  git commit -m "My initial commit" --author="A Genius <a_genius@linux.com>"
Created initial commit eafad66: My initial commit
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 somejunkfile
```

- **Storing author information**

```
$ git config user.name "Another Genius"
$ git config user.email "b_genius@linux.com"
```

- **Showing history**

```
$ git log
commit eafad66304ebbcd6acfe69843d246de3d8f6b9cc
Author: A Genius <a_genius@linux.com>
Date:   Wed Dec 30 11:07:19 2009 -0600

    My initial commit
```

File Management

- **File Categories:**
  - **Tracked:** in repository
  - **Ignored:** mentioned in `.gitignore` file
  - **Untracked:** not added yet, temporary, etc
- **Basic File Commands**

```
$ git add myfile
$ git rm myfile                 (removes from repository, not directory)
$ git rm myfile –cached         (removes a file not yet committed)
$ git mv oldfile newfile        (renames the actual file as well)
$ git ls-files
```

Making Commitments I

- You can commit changes as often as you want
- Examples:

```
$ git commit file1 file2 file3
$ git commit ./
$ git commit -a
$ git commit -m "This is a commitment I am making"
```

- To see uncommitted changes:

```
$ git diff
```

- Commits are fast; unchanged objects are reused

Making Commitments II

- **Viewing the Commit history:**

```
$ git log
commit 4b4bf2c5aa95b6746f56f9dfce0e4ec6bddad407
Author: A Smart Guy <asmartguy@linux.com>
Date:   Thu Dec 31 13:50:15 2009 -0600
    This is the fourth commit
commit 55eceacc9ab2b4fc1c806b26e79eca4429d8b52a
Author: A Smart Guy <asmartguy@linux.com>
Date:   Thu Dec 31 13:50:15 2009 -0600
    This is the third commit
commit f60c0c21764676beca75b7edc2f5f5e51b5dd404
Author: A Smart Guy <asmartguy@linux.com>
Date:   Thu Dec 31 13:50:15 2009 -0600
.............
$ git log --pretty=oneline
4b4bf2c5aa95b6746f56f9dfce0e4ec6bddad407 This is the fourth commit
55eceacc9ab2b4fc1c806b26e79eca4429d8b52a This is the third commit
f60c0c21764676beca75b7edc2f5f5e51b5dd404 This is the second commit
712cbafa7ee0aaef03861b049ddc7865220b4e2c This is the first commit
```

Making Commitments III

- **Identifiers and Tags:**

```
$ git log | grep commit | head -5
  commit 08d869aa8683703c4a60fdc574dd0809f9b073cd
  commit 1201b2a9bec0413188ada1443ece1a52da6dbff4
  commit d7f0eea9e431e1b8b0742a74db1a9490730b2a25
  commit 05a625486efc3209ae4d98e253dafa6ce0124385
  commit 1f11abc966b82b9fd0c834707486ef301b2f398d
```

- **Setting a tag and checking it out:**

```
$ git tag ver_10 08d869aa8683703c4a60fdc574dd0809f9b073cd
$ git tag ver_10 08d869
$ git checkout ver_10
```

Cloning

- To obtain a copy of a remote repository:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

- Returns entire project including .git directory with all objects, indexes etc.
- Protocol choices:

```
git://path/to/repo.git (Fastest)
file:///path/to/repo.git
ssh://user@remotesite.org[:port]/path/to/repo.git
user@remotesite.org:/path/to/repo.git
http://remotesite.org/path/to/repo.git
https://remotesite.org/path/to/repo.git
rsync://remotesite.org/path/to/repo.git
```

- Synchronize with changes at remote site:

```
$ git pull
```

Branches I

- Independent branches are useful:
  - Major release branch
  - Development branch
  - Subsystem branch(es)
- Tags are not branches:
  - A branch contains many tags
  - Two branches with common ancestor share tags
- Branches can be merged and rebased
- List branches with:

```
$ git branch
```

- Detailed history with:

```
$ git show-branch
```

Branches II

- Create a branch with:

```
$ git branch branch_name [starting_point]
$ git branch devel
```

- Delete a branch with:

```
$ git branch -d devel
```

- Checkout a branch with:

```
$ git checkout master
```

- Create and checkout a branch with:

```
$ git checkout -b newbranch [startpoint]
```

Merging

- **Merging the development branch:**

```
$ git checkout master
$ git merge devel
   Auto-merged file1
   CONFLICT (content): Merge conflict in file1
   Automatic merge failed; fix conflicts and then commit the result.
```

- **Then work out the conflicts**

Rebasing

- Rebasing the development branch:

```
$ git checkout devel
$ git rebase master devel
```

- Fix any conflicts and then do:

```
$ git rebase continue
```

- The development branch is now based on the updated master branch
- Problems can result:
  - Commit history is changed
  - Subtle problems because changes may not have been tested fully
  - Developers working off your branch have to rebase

Bisection

Suppose current version is bad, and there are many revisions since an earlier good version

- **Bisection can rapidly find the last good version**
- **Start with:**

```
$ git bisect start
$ git bisect bad
$ git bisect good V_10
```

- **Now test the code; it bug is still there do:**

```
$ git bisect bad
```

- **Otherwise, if bug is gone, do:**

```
$ git bisect good
```

- **Can automate with:**

```
$ git bisect run ./myscript.sh
```

Dealing With Patches: I

- **Patches are convenient:**
  - **Can be emailed instead of pushed and pulled**
  - **Large change sets can be broken into bite size pieces**
- **Producing a patch without GIT:**

```
$ diff -Nur stable_tree modified_tree > path-to/my_patch
```

- **Applying the patch:**

```
$ cd stable_tree ; patch -p1 < path-to/my_patch
```

- **Be careful with format, plain ascii text, no line-wrapping, etc. when emailing**

Dealing With Patches: II

- **Producing a patch with GIT:**

```
$ git format-patch -2
```

  - **Produces a patch file for each of last 2 commits:**

```
0001-first-commit.patch
0002-second-commit.patch
$ git format-patch master
```

  - **Produces patches for all changes from the master branch**
- **Signing off on patches with `-signoff` or `-s` adds**

```
Signed-off-by: A Smart Guy <asmartguy@linux.com>
```

- **Email patches with**

```
$ git send-email -to linux-kernel@vger.kernel.org 0001-first-commit-patch
```

Further Training

- **LF411: Embedded Linux Development**
  - **5-day class**
  - **Lab-based with real devices**
- **http://training.linuxfoundation.org**

Free Linux Training Webinar Series

Webinar #3: ***"An Introduction to Git"***
with James Bottomley

Sign up at **training.linuxfoundation.org**

Brought to you by