Git

This tutorial will address the source code management (SCM) tool named Git. By following these steps you should learn about the basic usage of Git, which is required for the whole practical course. Furthermore, Git is a great SCM tool, and it's good to know how to use it. During this tutorial, we will follow Alan Turing's thoughts towards developing the Turing Machine.

More in-depth Git documentation can be found on the official home page, which mentions books, videos, and links to other tutorials and references. Furthermore, the shell command `git help` lists the most commonly used Git commands, and `git help <command>` gives very detailed documentation for the specified Git command.

Contents

Creating Commits

Most steps of this tutorial are done by typing shell commands. The grey boxes contain the commands you should enter, preceded by a $ symbol, and followed by their output. While you may copy & paste these commands, some of them may require modifications to adapt them to your own projects. The output will be slightly different for many commands when you enter them, since it also depends on parameters such as the user name and time of execution.

1. Read the Git for Computer Scientists introduction (skip this if you are already familiar with Git).
2. For Linux, Git is available in its own package. Windows users can install msysGit. For Mac OSX, Git is available as part of Xcode; if you cannot install that, use Git for OSX.
3. Configure your name and email address (will be included in all commits you create):

   ```
   $ git config --global --add user.name "Your Name"
   $ git config --global --add user.email "<login>@informatik.uni-kiel.de"
   ```

4. Create a local repository for the "*Turing Project*":

   ```
   $ mkdir turing
   $ cd turing
   $ git init
   Initialized empty Git repository in ~/turing/.git/
   ```

   The `.git` subdirectory contains all history and metadata of the repository. You should not modify it. The `turing` directory contains the *working copy*, that is the currently checked-out snapshot. You work by modifying your working copy and committing the modifications to the repository (contained in `.git`).
5. Add and commit some content: copy `notes.txt` to your `turing` directory.

   ```
   $ git add notes.txt
   $ git commit -m "wrote some first notes"
   [master (root-commit) 2e73b34] wrote some first notes
    1 files changed, 5 insertions(+), 0 deletions(-)
    create mode 100644 notes.txt
   ```

   The file is now stored in the local history of your repository.
6. Edit `notes.txt`:
   a. Replace "fixed" with "infinite" in line 1.
   b. Replace "... (TODO)" with "a finite state machine" in line 4.
7. View the status of your current working copy:

```
$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   notes.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

8. Mark the modified file to include it in the next commit, then view the status again and compare with the previous output:

```
$ git add notes.txt
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   notes.txt
#
```

9. Commit the modified content to your local repository and view the status:

```
$ git commit -m "modified tape length, found a controller for tape head"
[master 52e2d49] modified tape length, found a controller for tape head
 1 files changed, 2 insertions(+), 2 deletions(-)
$ git status
# On branch master
nothing to commit (working directory clean)
```

After the preceding steps you have two commits in your local repository, each with one file in the index. You have different commands for viewing these commits:

```
$ git log
commit 52e2d4946791c2725015853e5e261ce143c6fe8a
Author: Miro Spoenemann <msp@informatik.uni-kiel.de>
Date:   Mon Oct 15 15:00:14 2012 +0200

    modified tape length, found a controller for tape head

commit 2e73b34ac44480773fc0e52875b7353a087d8c6d
Author: Miro Spoenemann <msp@informatik.uni-kiel.de>
Date:   Mon Oct 15 12:14:06 2012 +0200

    wrote some first notes

$ git show 52e2d49
commit 52e2d4946791c2725015853e5e261ce143c6fe8a
Author: Miro Spoenemann <msp@informatik.uni-kiel.de>
Date:   Mon Oct 15 15:00:14 2012 +0200

    modified tape length, found a controller for tape head

diff --git a/notes.txt b/notes.txt
index 4ded2b3..bd422b3 100644
--- a/notes.txt
+++ b/notes.txt
@@ -1,5 +1,5 @@
- * A tape with fixed length
+ * A tape with infinite length
   * Tape head can read or write data
   * Tape head can move left or right
- * The head is controlled by ... (TODO)
+ * The head is controlled by a finite state machine
```

Note that each commit is identified by a looong hash value, but it is possible to use only a prefix when referencing them (if the prefix is not ambiguous): the example above uses `52e2d49` to identify the second commit. The commit hashes in your repository will be different from those seen in this tutorial, because the name of the author and the exact time of committing is also considered in the hash calculation. Also try the command `gitk` to get an overview of your commits (a better alternative available for Mac OSX is GitX).

Branching and Merging

In the previous section you have created two commits on the default branch, which is named `master`. Now you will create a new branch and commit there, thus adding complexity to the commit graph. In general, you may create as many local branches as you like, since they are simple to use and can be a great tool to structure your work.

1. Create a branch with name *sketches*:

   ```
   $ git branch sketches
   ```

2. View the list of branches:

   ```
   $ git branch
   * master
     sketches
   ```

   The star reveals that you are still on the old `master` branch.
3. Switch to the new branch:

   ```
   $ git checkout sketches
   Switched to branch 'sketches'
   $ git branch
     master
   * sketches
   ```

   It is also possible to create a branch and switch immediately to it using the option `-b` of `git checkout`.
4. Download and add the new file `examples.txt`:

   ```
   $ git add examples.txt
   $ git commit -m "wrote first examples"
   [sketches cd63135] wrote first examples
    1 files changed, 20 insertions(+), 0 deletions(-)
    create mode 100644 examples.txt
   ```

   Inspecting the commit graph with `gitk` (or another graphical viewer) you see that the `sketches` branch now has three commits, while `master` is still at the second commit.
5. Merging the `sketches` branch into `master` means that all changes that have been made in `sketches` are also applied to `master`. In order to perform this merge, we have to check out the `master` branch first:

   ```
   $ git checkout master
   Switched to branch 'master'
   $ git merge sketches
   Updating 52e2d49..cd63135
   Fast-forward
    examples.txt |   20 ++++++++++++++++++++
    1 files changed, 20 insertions(+), 0 deletions(-)
    create mode 100644 examples.txt
   ```

   This was a *fast-forward* merge: since the `master` branch was completely contained in the `sketches` branch, the merge could be done by simply changing the head pointer of `master` to be the same as the head of `sketches`.
6. Now add the line "`see some examples in 'examples.txt'`" to the file `notes.txt` and commit this change in the current branch:

   ```
   $ git add notes.txt
   $ git commit -m "added reference to the new examples"
   [master a5e244f] added reference to the new examples
    1 files changed, 2 insertions(+), 1 deletions(-)
   ```

7. Switch back to the `sketches` branch and modify it as shown below. Note that the `checkout` command modifies your working copy, hence you have to update your text editor's content if you opened one of the files.

```
$ git checkout sketches
Switched to branch 'sketches'
```

   Add the line "`Move one step left:`" followed by an accordingly updated version of the tape with tape head at the end of the file `examples.txt`, then commit.

```
$ git add examples.txt
$ git commit -m "added another example"
[sketches 55a9cb1] added another example
 1 files changed, 5 insertions(+), 0 deletions(-)
```

   Now your two branches have *diverged*, which means that they cannot be fast-forwarded anymore.
8. Merge the `master` branch into `sketches`:

```
$ git merge master
Merge made by recursive.
 notes.txt |    3 ++-
 1 files changed, 2 insertions(+), 1 deletions(-)
```

   Using `gitk` you can see that a new commit was created that has two parent commits. Such a commit is called *merge* commit and is done automatically when a non-fast-forward merge is applied. See how both the change to `notes.txt` done in the `master` branch and the change to `examples.txt` done in the `sketches` branch are now contained in the repository state that results from the merge.
9. Add a commit in each of the two branches using the commands you have already learned.
   a. Check out `master`.
   b. Insert the following line after line 4 of `notes.txt`:

      ```
      * The finite state machine has an initial state and one or more final states
      ```

   c. Commit the change of `notes.txt`.
   d. Check out `sketches` (make sure to refresh your text editor so that `notes.txt` is reset to its previous state, without the change made above).
   e. Insert the following line after line 4 of `notes.txt`:

      ```
      * Each state transition can trigger head movement and data read/write
      ```

   f. Commit the change of `notes.txt`.
10. Merge the `master` branch into the current branch (`sketches`):

```
$ git merge master
Auto-merging notes.txt
CONFLICT (content): Merge conflict in notes.txt
Automatic merge failed; fix conflicts and then commit the result.
```

    As expected, the branches could not be merged automatically, since both branches modified the same line in the same file.
11. Use the `status` command to see the list of affected files:

```
$ git status
# On branch sketches
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:      notes.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

12. The modified `notes.txt` should now contain the following text:

```
<<<<<<< HEAD
 * Each state transition can trigger head movement and data read/write
=======
 * The finite state machine has an initial state and one or more final states
>>>>>>> master
```

The upper line is the one committed to `sketches`, while the lower line was committed to `master`. You have to resolve the conflict by editing the file. In this case the conflict is resolved by keeping both lines in arbitrary order, that means you should just remove the conflict markers (lines 5, 7, and 9 in `notes.txt`).

13. Use the `add` command to mark `notes.txt` as resolved. Entering `git commit` without a message will open a text editor with an automatically created commit message. Just close the editor, and the merge commit is completed:

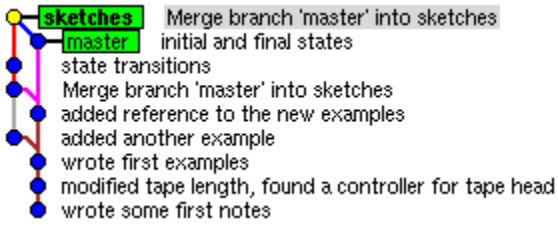
```
$ git commit
[sketches 21d5ddb] Merge branch 'master' into sketches
$ git show 21d5ddb
commit 21d5ddbbcba4e36464653a2a550dbf595ead921f
Merge: 17f75c7 8af2d50
Author: Miro Spoenemann <msp@informatik.uni-kiel.de>
Date:   Tue Oct 16 10:44:09 2012 +0200

    Merge branch 'master' into sketches

    Conflicts:
        notes.txt

diff --cc notes.txt
index 8f72873,bb81298..ba94a08
--- a/notes.txt
+++ b/notes.txt
@@@ -2,6 -2,6 +2,7 @@@
   * Tape head can read or write data
   * Tape head can move left or right
   * The head is controlled by a finite state machine
  + * Each state transition can trigger head movement and data read/write
 +  * The finite state machine has an initial state and one or more final states
    see some examples in 'examples.txt'
```

The `gitk` tool should now display this graph:



Remote Repositories

In the previous sections you have worked only with a local repository. The next step is to share this content with a remote repository, which we manage with Stash. You will first have to configure your Stash account:

1. Login to our Stash server with your Rtsys account information. If you haven't received your password yet, either wait until you have that password or register yourself in Stash (but don't use your Ifl login name – that one will be used later when we create your account).
2. Through the button in the top right corner, access your profile.
3. Switch to the *SSH keys* tab.
4. Click *Add Key* and upload a public SSH key that you want to use to access the repository.
    - If you don't have an SSH key: use the shell command `ssh-keygen`, confirm the default destination file `~/.ssh/id_rsa`, and choose whether to give a passphrase. If you have a passphrase, you need to enter it whenever you use your SSH key for the first time in a session. You can omit the passphrase, but that makes the key less secure. As result, the tool generates a private key `~/.ssh/id_rsa`, which has to be kept secret, and a public key `~/.ssh/id_rsa.pub`.

Usually it is sufficient to have only one local copy of a Git repository. However, in this tutorial you will create a second copy in order to "simulate" what can happen if two users access the same remote repository: imagine the directories `turing` and `turing2` are each managed by a different user. You will simulate the resulting interference by switching your working directory between these two.

1. Go to Stash *Create Project* and call it "personal-<login>", replacing <login> with your own login name. Use your uppercase login name as project key, e.g. "MSP".
2. Go to the *Permissions* tab of the project page and add the user "msp" as observer.

3. On the project page, select *Create Repository* and name it "turing".
4. Copy the SSH URL shown in the top right and email it to msp@informatik.uni-kiel.de. This will serve as proof for your work on this tutorial.
5. Transfer your `master` branch to the new server-side repository. Replace the URL in the following command by the one copied from Stash:

```
$ git remote add stash ssh://git@git.rtsys.informatik.uni-kiel.de:7999/MSP/turing.git
$ git push stash master
Counting objects: 15, done.
Delta compression using up to 16 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (15/15), 1.54 KiB, done.
Total 15 (delta 3), reused 0 (delta 0)
To ssh://git@git.rtsys.informatik.uni-kiel.de:7999/MSP/turing.git
 * [new branch]      master -> master
```

   The first command adds a *remote* named "stash" to your local repository, which is just a bookmark for the long URL. The second command transfers the `master` branch to the server, which is called *pushing*. After that is done, reload the Stash page in your browser, and you see all changes that are transferred to the server-side repository.
6. Create a local clone of your remote repository (replace the URL accordingly):

```
$ cd ..
$ git clone ssh://git@git.rtsys.informatik.uni-kiel.de:7999/MSP/turing.git turing2
Initialized empty Git repository in /home/msp/tmp/turing2/.git/
remote: Counting objects: 15, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 15 (delta 3), reused 0 (delta 0)
Receiving objects: 100% (15/15), done.
Resolving deltas: 100% (3/3), done.
$ cd turing2
```

   The `clone` command automatically creates a remote named `origin` in the new local repository, which is set to the given URL. You will use this second clone to simulate another user with access to the repository.
7. Edit the file `examples.txt` in the new clone (`turing2`): replace `"a"` in line 6 by `"c"` and correct the tape representations in lines 9, 14, and 19 accordingly. Commit the change.
8. Push the new commit to the server:

```
$ git push
Counting objects: 5, done.
Delta compression using up to 16 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 362 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To ssh://git@git.rtsys.informatik.uni-kiel.de:7999/MSP/turing.git
   8af2d50..1d1577f  master -> master
```

   In this case the push command can be used without arguments, which means that it pushes all branches as configured in `.git/config`:

```
$ more .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[remote "origin"]
        fetch = +refs/heads/*:refs/remotes/origin/*
        url = ssh://git@git.rtsys.informatik.uni-kiel.de:7999/MSP/turing.git
[branch "master"]
        remote = origin
        merge = refs/heads/master
```

   Here the branch `master` is linked with the remote `origin`, hence `git push` does the same as `git push origin master`.
9. Go back to the original local repository and check out the `master` branch:

```
$ cd ../turing
$ git checkout master
Switched to branch 'master'
```

10. Merge the `sketches` branch into `master`:

```
$ git merge sketches
Updating 8af2d50..21d5ddb
Fast-forward
 examples.txt |    5 +++++
 notes.txt    |    1 +
 2 files changed, 6 insertions(+), 0 deletions(-)
```

Now your local `master` branch and the one on the server-side repository have diverged

11. Fetch the server-side changes:

```
$ git fetch stash
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From ssh://git@git.rtsys.informatik.uni-kiel.de:7999/MSP/turing.git
   8af2d50..1d1577f  master     -> stash/master
```

Now the change to `examples.txt` that was previously committed in the `turing2` repository is stored in a *remote tracking branch* named `stash/master`:

```
$ git branch -a
* master
  sketches
  remotes/stash/master
```

You can analyze the remote tracking branch using the `log` and `show` commands. However, you should never directly modify a remote tracking branch.

12. You can merge the remote changes into your local `master` branch with the following command:

```
$ git merge stash/master
Auto-merging examples.txt
Merge made by recursive.
 examples.txt |    8 ++++----
 1 files changed, 4 insertions(+), 4 deletions(-)
```

Since this combination of `fetch` and `merge` is used very often, Git offers a shortcut for it, namely the `pull` command. In this case the according command would have been `git pull stash master`.

13. Push the merged branch to the server, and then push the `sketches` branch, which is not on the server yet:

```
$ git push stash master
Counting objects: 23, done.
Delta compression using up to 16 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (14/14), 1.65 KiB, done.
Total 14 (delta 4), reused 0 (delta 0)
To ssh://git@git.rtsys.informatik.uni-kiel.de:7999/MSP/turing.git
   1d1577f..957f686  master -> master
$ git push stash sketches
Total 0 (delta 0), reused 0 (delta 0)
To ssh://git@git.rtsys.informatik.uni-kiel.de:7999/MSP/turing.git
 * [new branch]      sketches -> sketches
```

14. As next step change your working directory to the second local repository `turing2`, add the following line to the end of `notes.txt` in the `turing2` directory, and commit the change:

```
TODO: formal definition
```

15. Trying to push this commit to the server results in the following error message:

```
$ git push
To ssh://git@git.rtsys.informatik.uni-kiel.de:7999/MSP/turing.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'ssh://git@git.rtsys.informatik.uni-kiel.de:7999/MSP/turing.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes before pushing again.  See the 'Note about
fast-forwards' section of 'git push --help' for details.
```

    This is because you have modified the branch while working in the original `turing` repository, and these changes have to be merged with the new commit you have just made for `notes.txt`.
16. The solution is to apply the `pull` command followed by the `push` command:

```
$ git pull
remote: Counting objects: 23, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 14 (delta 4), reused 0 (delta 0)
Unpacking objects: 100% (14/14), done.
From ssh://git@git.rtsys.informatik.uni-kiel.de:7999/MSP/turing.git
   1d1577f..957f686  master     -> origin/master
 * [new branch]      sketches   -> origin/sketches
Auto-merging notes.txt
Merge made by recursive.
 examples.txt |    5 +++++
 notes.txt    |    1 +
 2 files changed, 6 insertions(+), 0 deletions(-)
$ git push
Counting objects: 10, done.
Delta compression using up to 16 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 673 bytes, done.
Total 6 (delta 2), reused 0 (delta 0)
To ssh://git@git.rtsys.informatik.uni-kiel.de:7999/MSP/turing.git
   957f686..b58ded7  master -> master
```

    While `pull` performs a `fetch` and a `merge`, `push` transfers the new merged branch to the server. Note that during the merge operation conflicts can occur. In that case you have to resolve them and commit the changes before you can push. When used without parameters like shown above, `pull` lookes in `.git/config` to determine which branches to pull from which remotes.
17. In order to check out the `sketches` branch locally, which was previously pushed to the server, simply type the following command:

```
$ git checkout sketches
Branch sketches set up to track remote branch sketches from origin.
Switched to a new branch 'sketches'
```

    This branch can be pushed and pulled with the server in the same way as you did for the `master` branch. Never check out `origin/sketches`, since that is a remote tracking branch!

The `master` branch should look like this:

Other Useful Commands

This section contains optional steps that you don't need to push online, but can be useful for you to learn.

Ignoring Files

While working on his Machine, Alan Turing has produced a temporary file `experiments.tmp`, which he does not want to commit in the repository:

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       experiments.tmp
nothing added to commit but untracked files present (use "git add" to track)
```

Since the extra mention of that file can make Git's status reports unnecessarily cluttered, Alan wants to ignore it permanently. Help him by adding a `.gitignore` file to the repository:

```
$ echo "*.tmp" > .gitignore
$ git add .gitignore
$ git commit -m "added ignore file"
[master 738ce4c] added ignore file
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 .gitignore
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
nothing to commit (working directory clean)
```

Now the `experiments.tmp` file is not considered when viewing the status. You can add arbitrary file name patterns to the `.gitignore` file; for example it is a good idea to ignore `*.class`, which are binary files generated for Java projects.

Discarding Changes

While working on his Machine, Alan Turing has made some changes to notes.txt that he later found out to be nonsense:

```
$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   notes.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Help Alan by restoring the last committed state of that file:

```
$ git checkout HEAD notes.txt
$ git status
# On branch master
nothing to commit (working directory clean)
```

Instead of HEAD, which is the last commit on the current branch, you can also name any other branch or commit hash. In that case you would have to commit the change to make it permanent. While resolving conflicts it is possible to use `--theirs` or `--ours` instead of HEAD, which replaces the whole content of the respective file by their version (the one on the remote branch) or our version (the one on the current branch).

A more brute-force option is using the `reset` command:

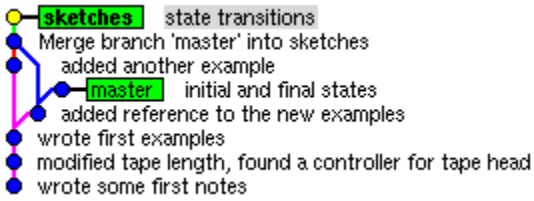
```
$ git reset --hard
HEAD is now at b58ded7 Merge branch 'master' of git.rtsys.informatik.uni-kiel.de:7999/MSP/turing
```

This resets *all* changes to the working copy to the head of the current branch, so use it with caution! However, `reset` does not remove unstaged files. In order to do that in one command, use `clean`:

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       test1.tmp
#       test2.tmp
nothing added to commit but untracked files present (use "git add" to track)
$ git clean -f
Removing test1.tmp
Removing test2.tmp
```

Rebasing

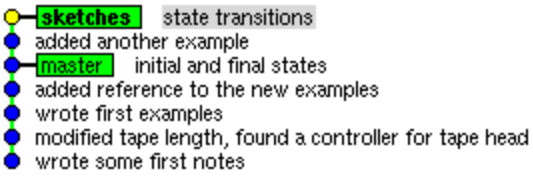
Consider the following situation:



If you want to merge the changes made on the `master` branch into the `sketches` branch, the normal way is to use the `merge` command and create a merge commit. However, the `rebase` command gives an interesting alternative to that: it reapplies all commits done in the current branch starting from a given reference.

```
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added another example
Applying: state transitions
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging notes.txt
```

Afterwards the commit graph looks like this:



The two commits made in `sketches` are reapplied starting from the head of the `master` branch. The resulting structure of commits is much cleaner than before. `rebase` even allows to squeeze multiple commits into one. Note that in this example a merge conflict had to be resolved in the same way as it was done in Section "Branching and Merging"; instead of committing the resolved file, the rebase command is resumed with `git rebase --continue`.

Never rebase a branch that is already pushed online! Due to the structural change the rebased branch is no longer compatible with the previous one, and pushing it will fail, since fast-forward merge is not possible.

Tagging

Finally Alan Turing has made a great success in the development of his Machine, and he would like to fix that stage as "Milestone 1". Help him by tagging the current state of the project:

```
$ git tag milestone1
```

Then the head of the current branch is stored under the name `milestone1`, so it can be found very easily at later stages of the project:

```
$ git tag
milestone1
$ git checkout milestone1
Note: checking out 'milestone1'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 957f686... Merge remote branch 'stash/master'
```

Tags can also be loaded to the server using the `push` command.