gist / jist/ *n* [AF , it lies, fr. *gisir* to lie, ultim. fr. L *jacere* — more at ADJACENT] (ca. 1711) **1** : the ground of a legal action **2** : the main point or part : ESSENCE <the ~ of an argument>

[1]**git** /'git/ *n* [var. of *get*, term of abuse, fr. [2]*get*] (1929) *Brit* : a foolish or worthless person

[2]**git** *dial* var of GET

git-go var of GET-GO

git·tern /'gi-tərn/ *n* [ME *giterne*, fr. MF *guiterne*, modif. of OSp *guitarra* guitar] (14c) : a medieval guitar

an introduction to git

Antonio Cervone and Luca Formaggia

2015 V1

Merriam-Webster's Collegiate Dictionary, p.529

What is a revision control system (RCS)

Software needs to be maintained, debugged, improved... while keeping track of the changes and being able to recover old revisions if needed.

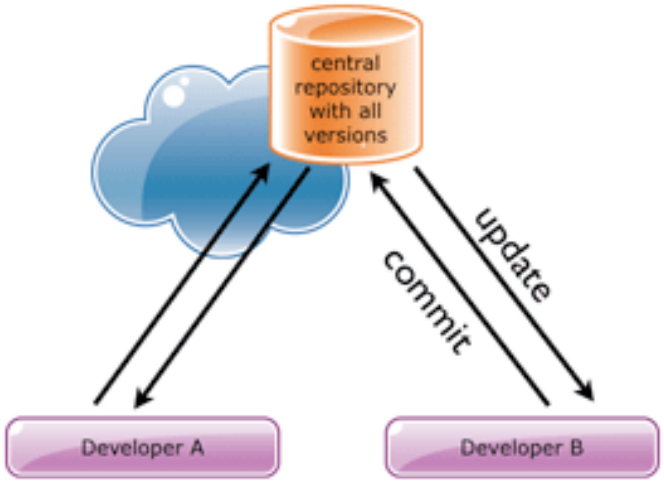
Those task can be done automatically using a Revision Control System.

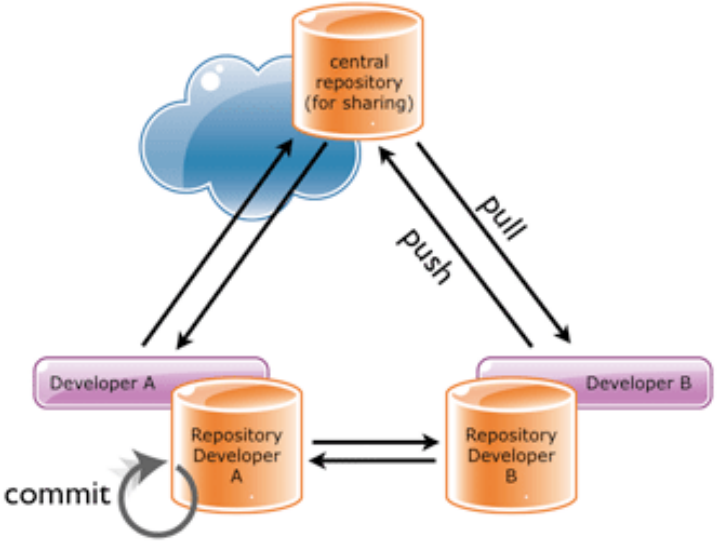
In addition, one may want to keep a copy of the code (and of all its changes) in a remote repository, for safety and for sharing with others.

Most common RCS: CVS, RCS, git, mercurial (Hg)....

git e mercurial sono compatibili
mercurial è git per gli umani

Centralised vs Distributed RCS



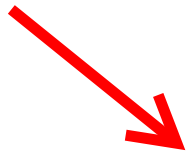


Centralised RCS compares with a distributed RCS as an hub compares with a peer–to–peer model.

The philosophy of a distributed RCS Come git

copia locale creata con clone

- Each user has a local copy of the repository: you have full control of all the history even if you are not connected to the web
- Most commands operate on the local copy (minimize communications).
- You may have multiple remote repositories: better distributed development.



su internet

A Distributed RCS: git

Git is a distributed revision control system with an emphasis on speed. Git was initially designed and developed by Linus Torvalds for Linux kernel development.

completamente sviluppata

Every Git working directory is a full-fledged repository with complete history and full revision tracking capabilities, not dependent on network access or a central server.

In git (almost) every command operated just on the local repository. Indeed you do not even need a remote repository to use git!.

add - commit

The only commands that communicate with a remote repository

Even if you do not yet know what they do, here there are the (only) three commands used by git to operate on a remote repository:

push, fetch, pull

First thing to do

If you have never used git on a computer you should first let git know who you are (so that people can blame you for your wrong deeds or cherish you for your programming skill!)

```
git config -global user.name “Luca Formaggia”
```

```
git config -global user.email “micky.mouse@gmail.com”
```

`git config` tells git to change some git configuration. `global` makes it to change some global parameter, i.e. a parameter that applies to all yours git working directories in that computer.

Need help?

Git has an integrated help. If you need help for the command `command` just do

```
git help command
```
help + nome del comando

Git uses a lot of terms that may confuse you at the beginning (and not only at the beginning...). A useful command is

```
git help glossary
```
mostra tutti i comandi di git

You find more on the Git Book (git-scm.com/book)

More help?

Other useful commands

```
git help tutorial
```

A tutorial

```
git help tutorial-2
```

The second part of the tutorial.

Create/clone a repo

A local repository. In the directory where you want to create it:

```
$> git init
```

Cloning from an existing remote repo

```
$> git clone \
> <username>@cmcsforge.epfl.ch:/gitroot/lifev/lifev
```

git clone + url del deposito (fork o main)

From a mox repository

```
$> git clone
gitolite@gitserver.mate.polimi.it:eni-lifev
```

Github

Github (github.com) is a site that allows you to create repositories and shere them with others. It provides also a nice web interface.
We will use github for the examples of the PACS course.

Edit!

modify the files in the repo

```
$> edit the sources ...
```

show status of the repo

```
$> git status
# On branch <branch_name>
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# modified:  file1.cpp
#
# Changed but not updated:
# (use "git add <file>..." to update what will be committed)
# (use "git checkout - <file>..." to discard changes in working directory)
#                                    eliminare
# modified:  file2.cpp
#
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
#
# file3.cpp
```

The possible state of a file in the working directory

- untracked The file is not under git control. Maybe is just a temporary file. Or you want to put it under control using `git add`. Il file non è mai stato added
- modified The file has been modified since the last commit in the repository. May be you want add it to the staging area. Il file è già stato added e committed ma ha subito una modifica dall'ultimo commit
- staged (in the index). Ready to be committed in the repository with a `git commit` Non ha subito modifiche dall'ultimo add ma non è ancora stato committed (la nuova versione)

Beware: when you run `git add filename` to a modified file you put a snapshot of that file in the staging area. If you modify it again before the commit you need to do `git add` again (unless you want to commit the previous version).

istantanea (foto)

Commit

show differences with the last commit

```
$> git diff <branch> <files>
```

stage files for a commit

```
$> git add <edited_files>
```

create a new commit in the repository

```
$> git commit
...
<vim editor will come up,
write a description for the commit>
```

```
$> git commit -m "message"
```

The last command writes the commit message directly.

What to put in a commit message

```
A first line with a brief description (it is the one
that is shown by some git commands)

Possibly followed by an empty line and a detailed
description (possibly wrap lines to 72 characters)

Paragraphs are separated by empty lines.

- you may use lists using the - character
```

Moving/removing files. Adding directories

```
$> git rm <files>
$> git mv filefrom fileto
```

Use git commands to move or remove files that are in the index (i.e. under git control). Git will automatically register the change for the next commit!
If you add a directory, put a file into it before adding it.

```
$> mkdir newdir; touch newdir/README.md
$> git add newdir
```

The file(s) in the directory is (are) automatically added.

What is a commit

A commit is a snapshot of your git working tree at the moment of the commit. cancelletto
It is identified by a SHA-1 hash key generated from the content of the files in the working tree and the hash key of the previous commit(s).
Any part of the hash key (as long as unique in the repo) can be used to address a commit. A commit may also have symbolic names called tags.

```
$> git show
```

shows the most recent commit (on the current branch).

Special symbolic names for some commits

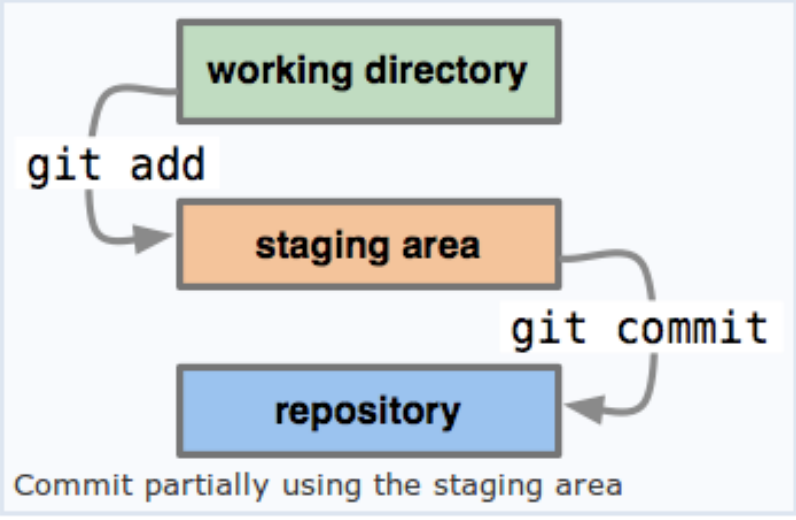
Shorthand	Definition
HEAD	Last commit
HEAD^	One commit ago
HEAD^^	Two commits ago
HEAD~1	One commit ago
HEAD~3	Three commits ago

You may use those short hands in all git command that may refer to a commit

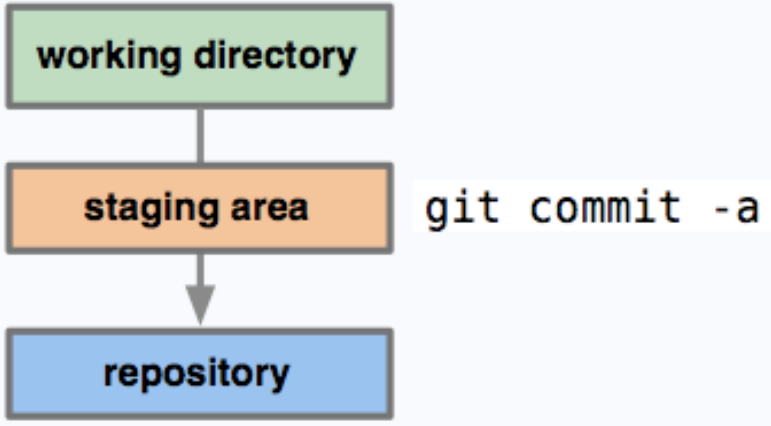
```
$> git log HEAD-3..HEAD
$> checkout HEADM
```

The first command prints a log briefly describing the last 4 commits. The second position your working area to two commits ago. HEAD may be thought as a pointer to the last commit.

A graphical view



Commit partially using the staging area



Commit everything directly with `git commit -a`

add, commit

So, the basic operations are

- git add file(s) Add a snapshot of the file(s) to the staging area ready for commit. On a new file, it adds also the file to the git history.
- git commit Stores the staging area into the git local repository creating a new “commit” (also called blob).

```
$> git commit -a
```

commits all modified files.

Some advice

- Do atomic commits: each commit should be related to a single “logical change” in your code: a bug fix, a new feature... Avoid monster commits with a lot of files.
- Write significant commit messages. Message like “this is a commit” are not allowed!
- Git won’t allow commits with empy messages.

Communicate with remote

To communicate with a remote repository we have three commands: push, fetch and pull.
We see now the simplest use of them. We assume we have a single remote repository (`origin`) and a single branch (`master`).

E' il deposito che abbiamo clonato
Non importa se sia il principale o il fork

E' il ramo costituito da tutte
le versioni dei file

Push

```
git push
```

Pushes all the commits for the current branch (`master` in this case) onto the remote repo.

Fetch

Con fetch possiamo scaricare le nuove modifiche dal remoto (internet) e metterle nel deposito locale (creato con clone)
Con checkout possiamo leggere i file importati con fetch ma NON possiamo modificarli
Merge consente di unire le modifiche ai file che già abbiamo nel workspace (ora possiamo modificarli)

```
$> git fetch
```

Andare a prendere

Fetches all the commits for the current branch in the remote repository and store them locally in `remotes/origin/<branch>`
You can then examine them (read only!) by doing

```
$> git checkout remotes/origin/new-branch
```

You can finally merge the commits into your master

```
$> git checkout master
$> git merge remotes/origin/new_branch
```

Prima ci poniamo nel ramo a cui vogliamo unire le modifiche e dopo usiamo merge

pull

```
$> git pull <branch>
```

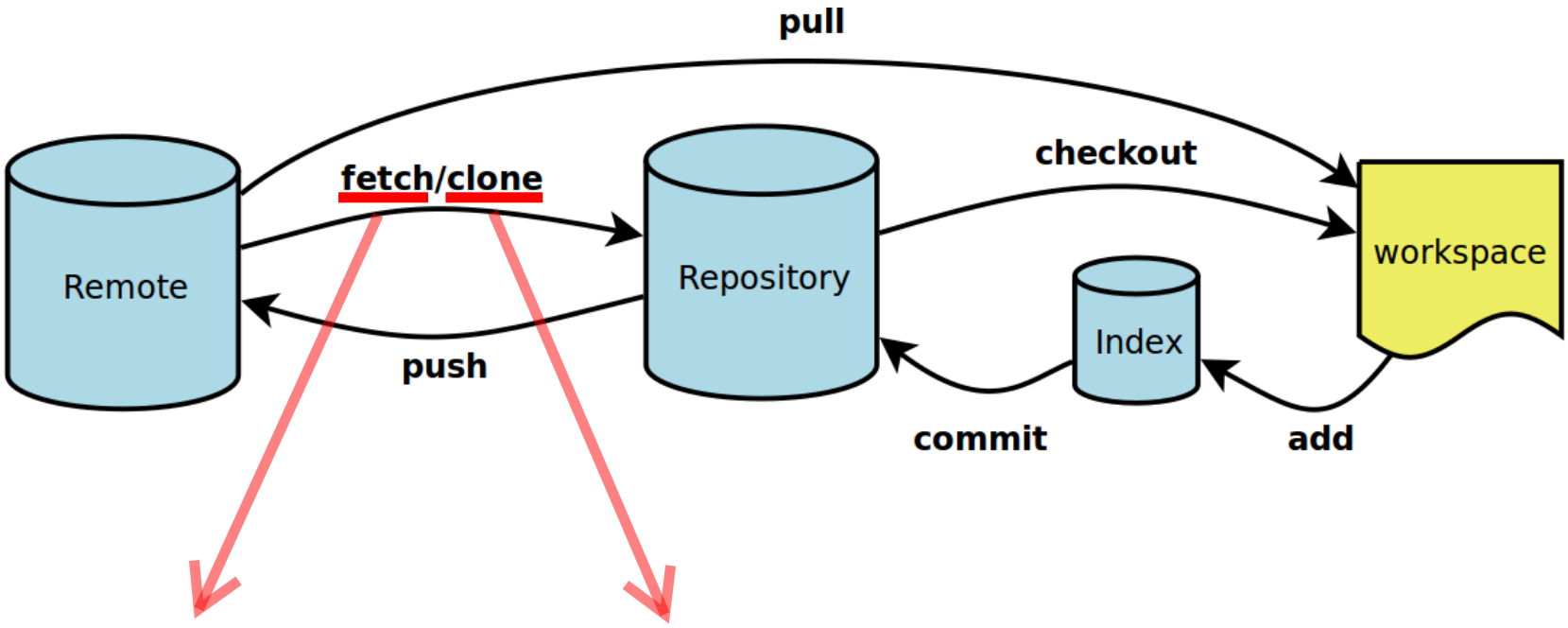
= git fetch + git merge (scarichiamo direttamente le modifiche nel workspace (read and write) senza passare dal locale)

Is equivalent to `git fetch` followed by `git merge`.

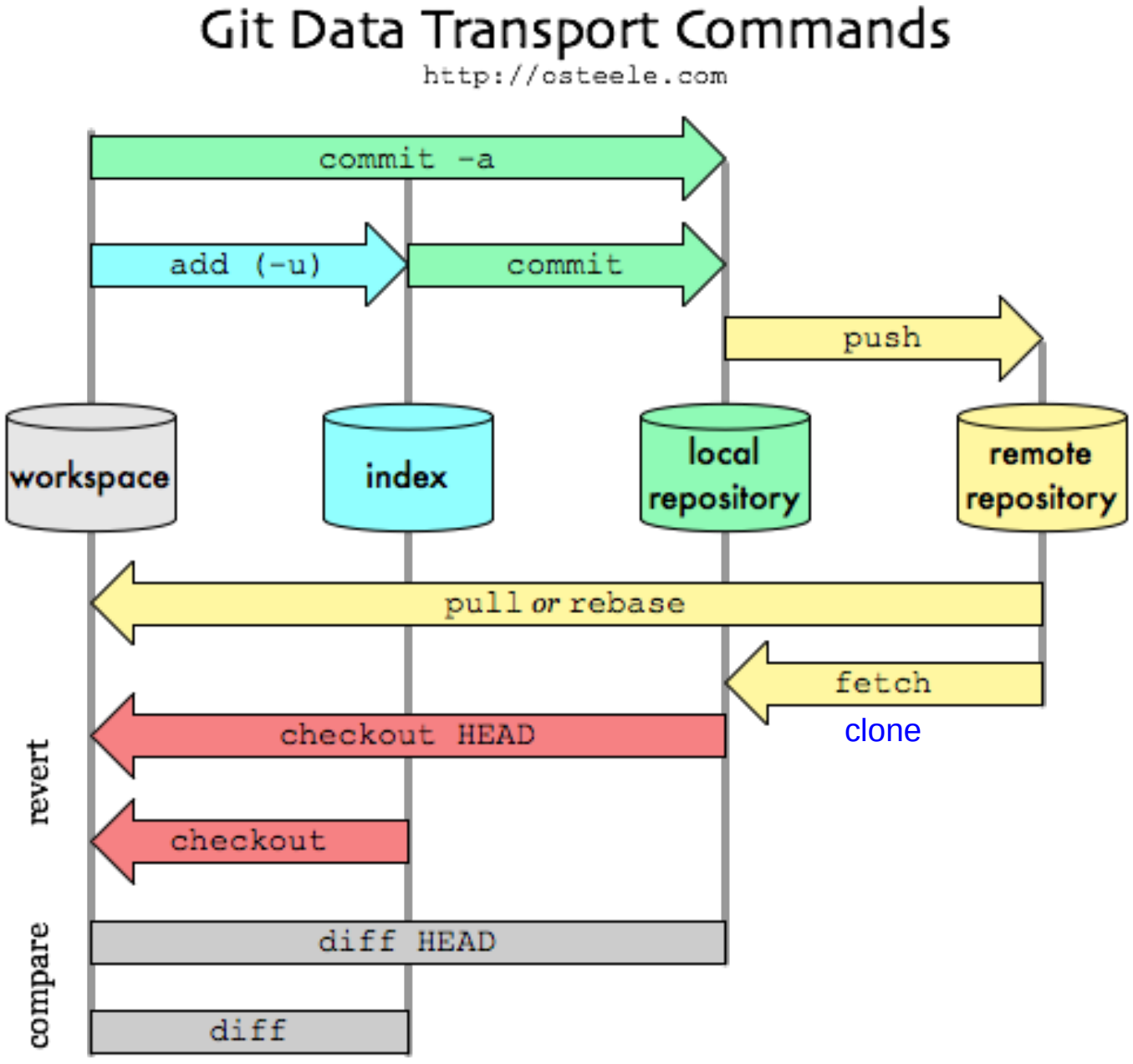
It is the most used command when you want to retrieve commits from a remote repository.

recuperare

add, commit, push pull...



git transport mechanism



Merging gone wrong

```
$> git merge -abort
```

cancels the merge.

```
$> git diff <commit|branch>
```

show differences between working tree and the commit (default HEAD).

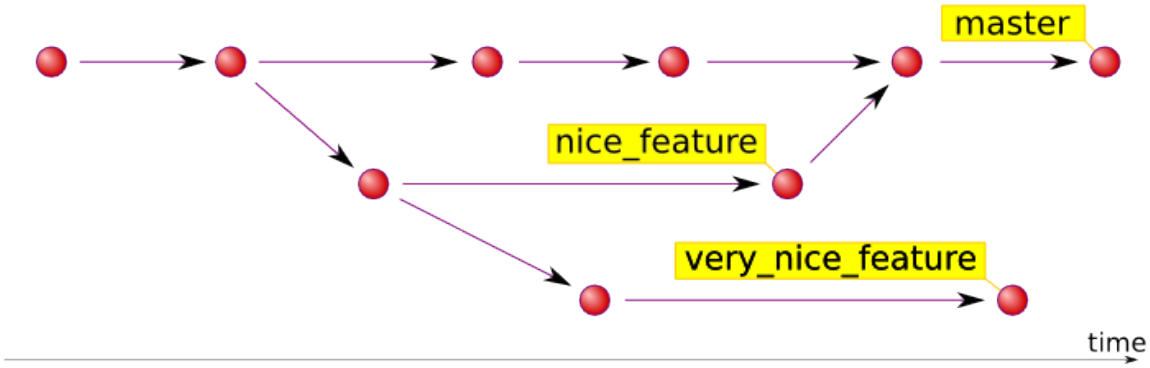
```
$> git mergetool <-tool=mytool> $> git mergetool
-tool-help
```

helps you in resolving conflict by using the tool you prefer. The second command gives you the list of tools.

```
$> git difftool <-tool=mytool>
```

Allows you to use a tool of choice for the diff.

Branches



Branches is one of the most important feature of git. When you create a repo you create also the `master` branch. But often you want to branch your tree to work on a specific topic without affecting the master branch.

Branching- create

create new branch

```
$> git branch <shiny_new_branch>
```

Go to that branch.

```
$> git checkout <shiny_new_branch>
```

Or create and position yourself in a new branch

```
$> git checkout -b <shiny_new_branch>
```

Get other branches

list local branches

```
$> git branch
  b1
* b2
  master
```

list remote branches (i.e. branches fetched from remote)

```
$> git branch -r
```

list all branches

```
$> git branch -a
```

LifeV branch name policy

```
YYYYMMDD_<meaningful_name>
```

Branching - delete

delete a branch

```
$> git checkout master
$> git branch -d <branch_to_delete>
```

will cause an error if not merged with master!
To delete without merging

```
$> git branch -D <branch_to_delete>
```

delete a remote branch

```
$> git push origin :<branch_to_delete>
```

the syntax comes from the generic generic push command

```
git push <remotename> <localbranch_name>:<remotebranch_name>
```

so we push an empty branch into the remote branch to be deleted

Branching - merge

merge a branch

```
$> git checkout master
$> git merge <branch_name>
Updating e0f73f9..cd928c7
Fast-forward
```

Branching - merge

merge a branch

```
$> git checkout master
$> git merge <branch_name>
Updating e0f73f9..cd928c7
Fast-forward
```

conflicts may arise!

```
CONFLICT (content):  Merge conflict in file.cpp
Automatic merge failed; fix conflicts and then commit
the result.
```

Branching - solve conflicts 1

manual

```
<<<<<<< HEAD
...
=======
...
>>>>>>> branch_name
```

Branching - solve conflicts 1

manual

```
<<<<<<< HEAD
...
=======
...
>>>>>>> branch_name
```

text editor/gui

```
$> git mergetool
merge tool candidates:  meld opendiff kdiff3 tkdiff
xxdiff tortoisemerge gvimdiff diffuse ecmerge p4merge
araxis emerge vimdiff
```

Branching - solve conflicts 2

after the diff are removed

```
$> git add file.cpp
```

a commit is needed when all the conflicts are solved!

```
$> git commit
```

Merge tools

If a merge has a commit, the command

```
$> git mergetool -tool=<tool>
```

opens a graphical interface to handle conflicts.

Branching - stash

checkout with hanging modifications is forbidden!

```
$> git checkout master
error:  Your local changes to the following files
would be overwritten by checkout:  ...
```

we can stash modifications

```
$> git stash save
```

and bring them back

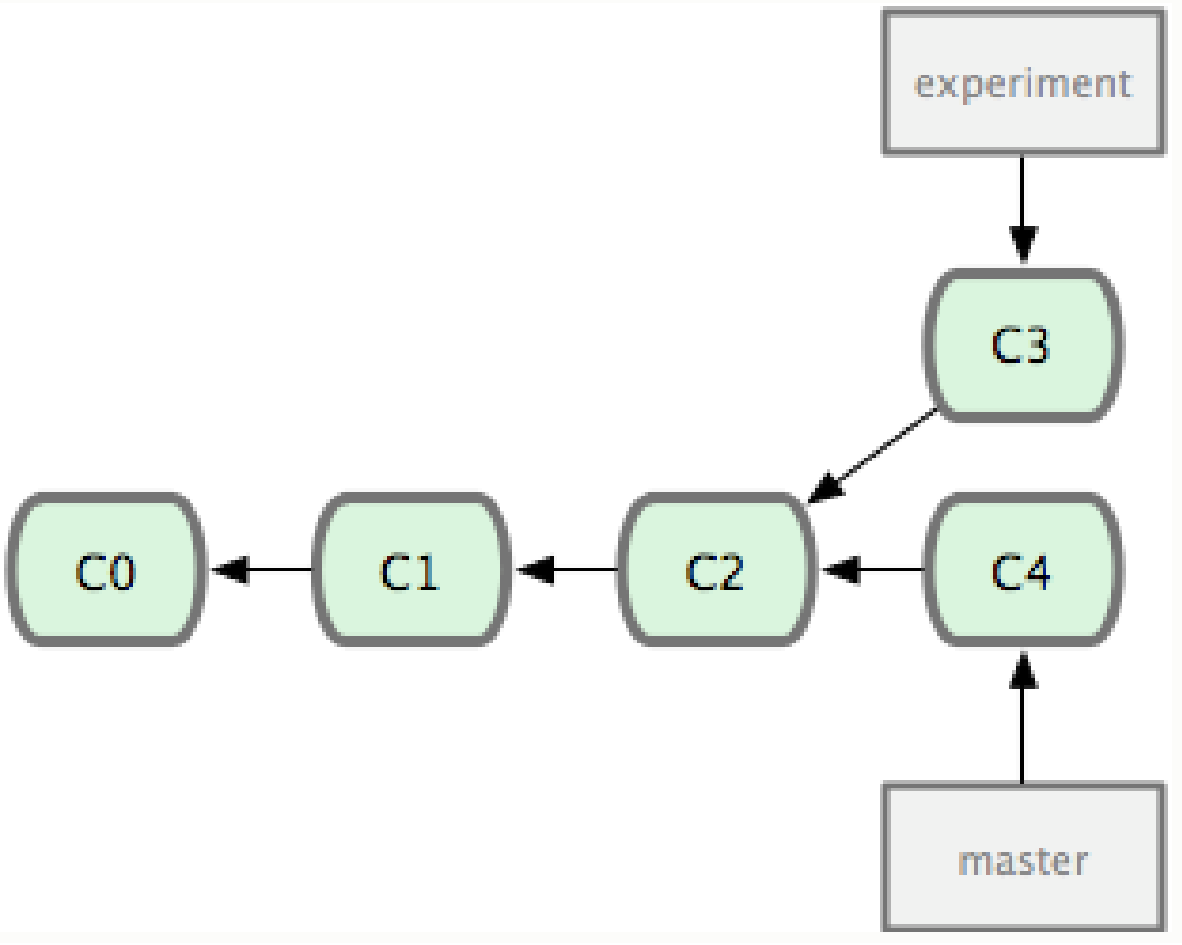
```
$> git stash { pop | apply }
```

remember to clear the stash (in particular if you use apply).

```
$> git stash list
$> git stash clear
```

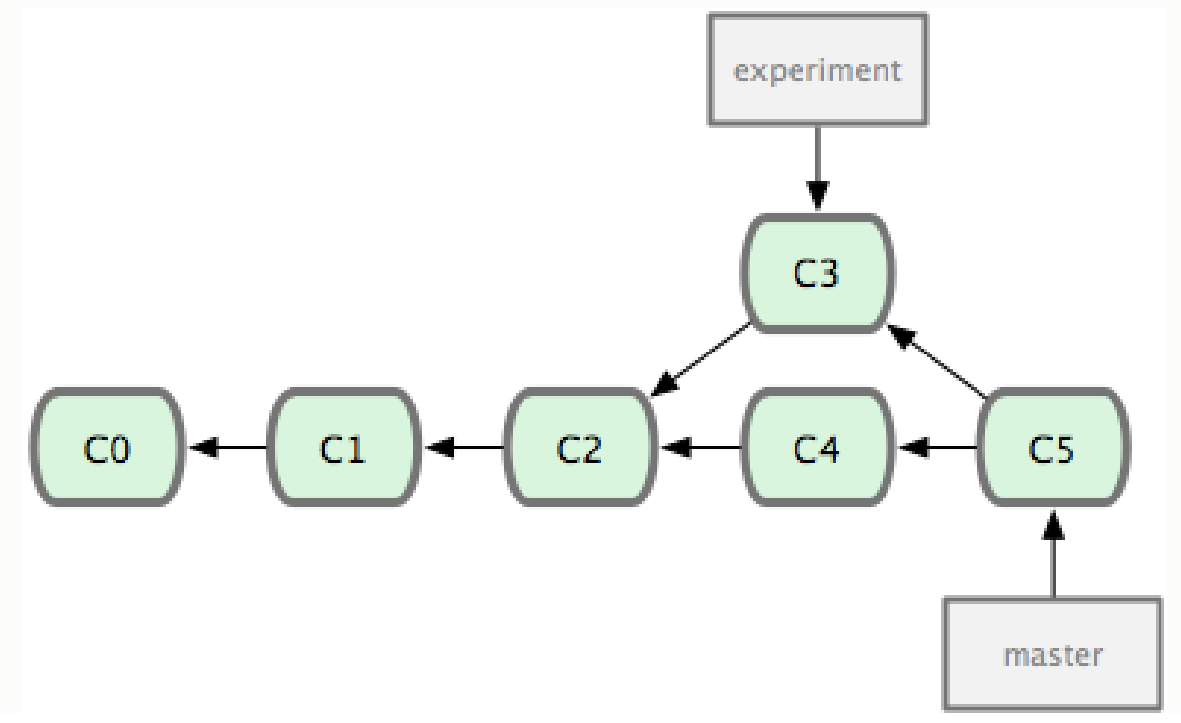
Merging and rebasing

There are two ways in git to integrate work from one branch to the other.



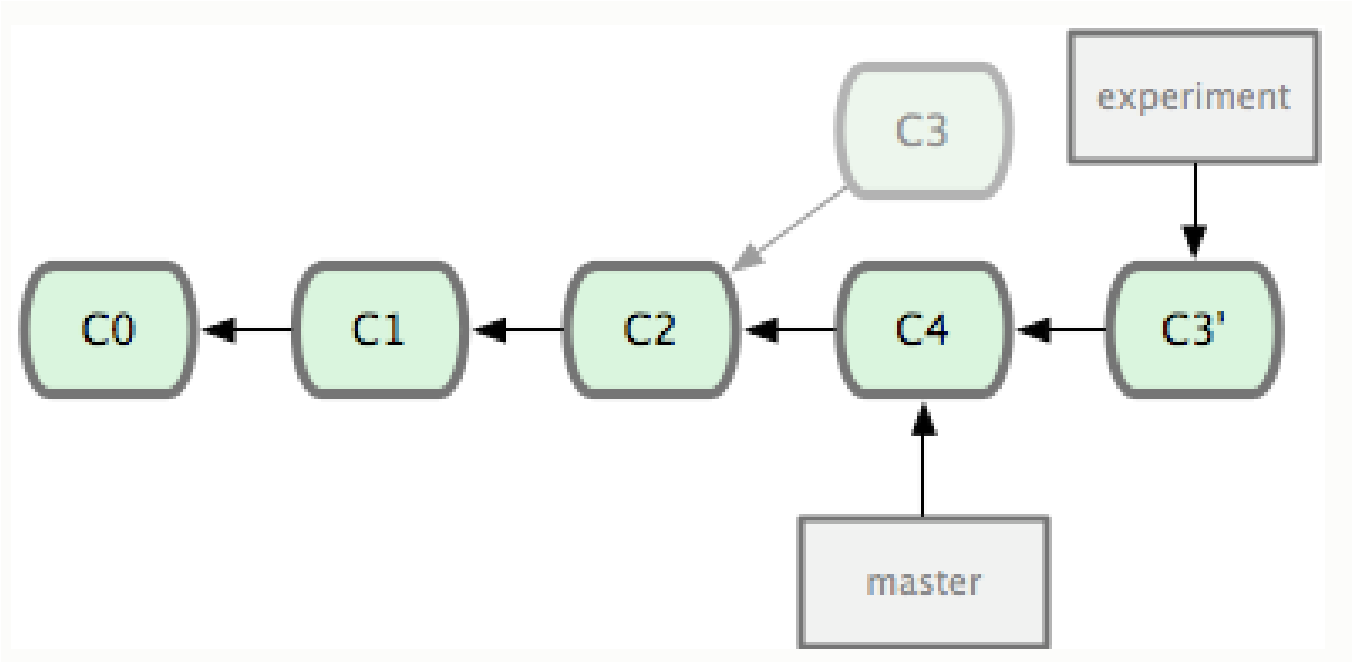
Merge

```
$> git checkout master
$> git merge experiment
```



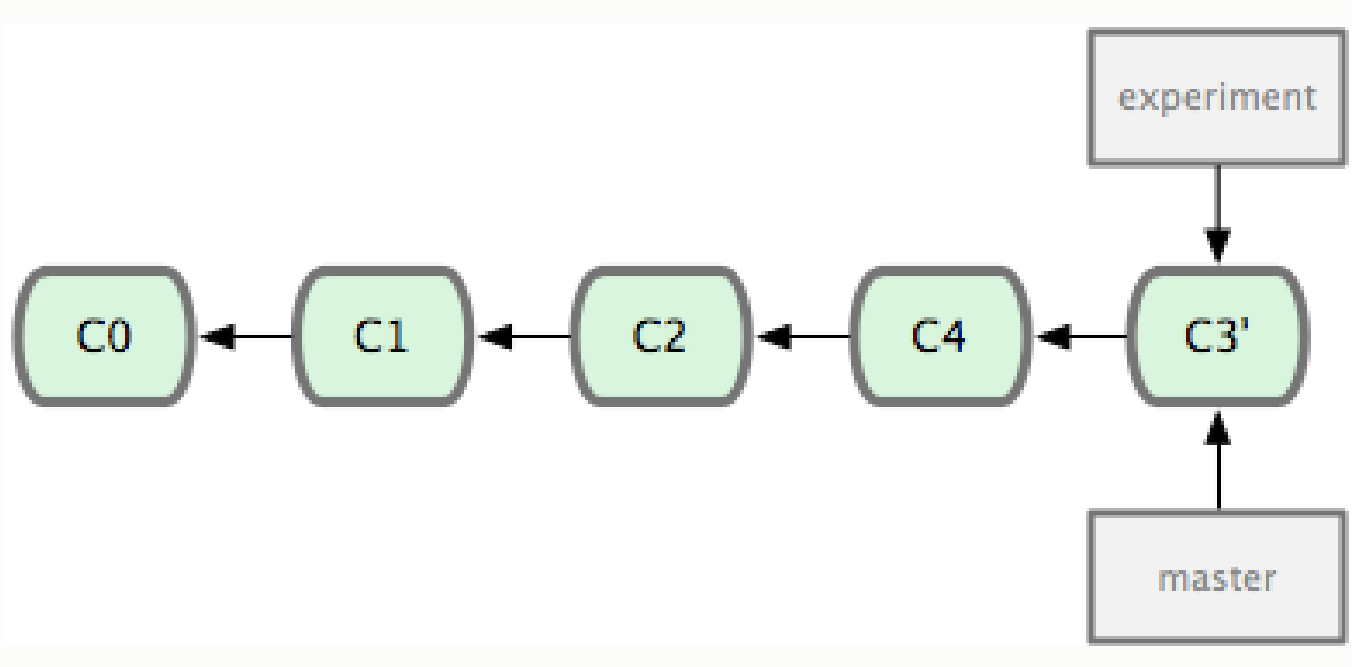
Rebase

```
$> git checkout experiment
$> git rebase master
```



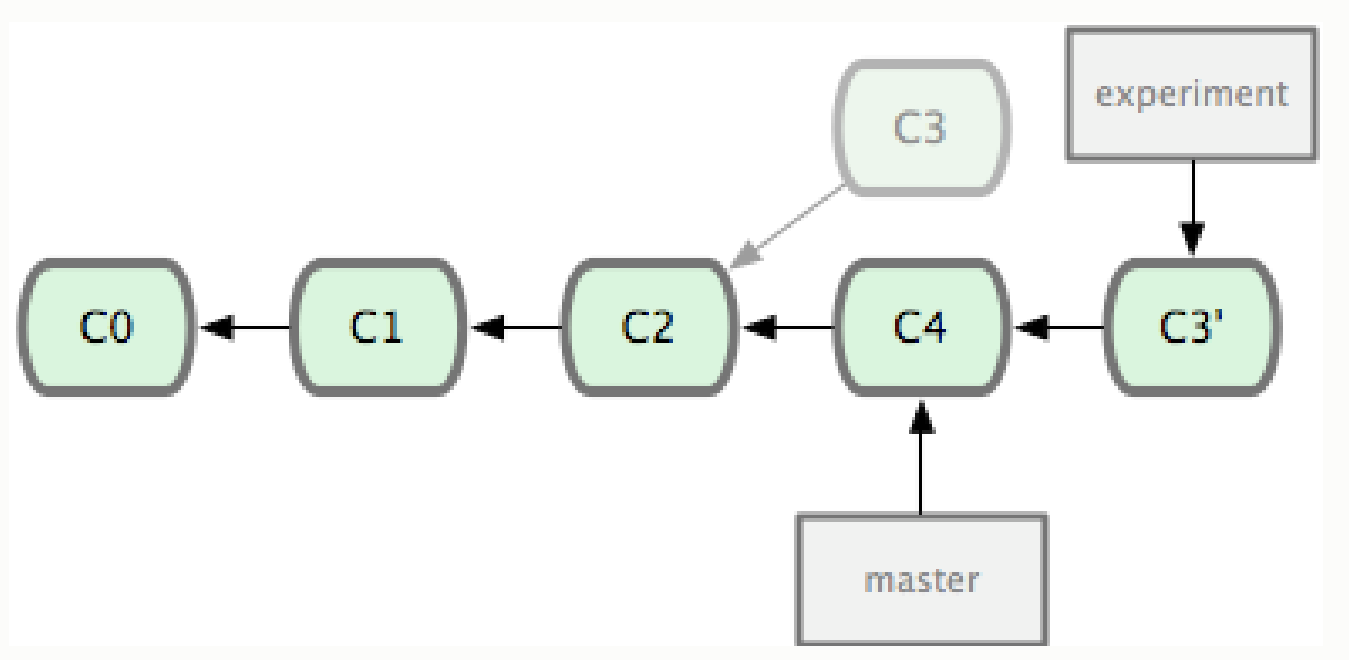
A merge to realign master with experiment

```
$> git checkout master
$> git merge experiment
```



Rebase (alternative form)

```
$> git rebase master experiment
```



The differences

Merge does a three way merge. The history after the merge reflects what has happened. You create a commit with 2 parents.

Rebase “replays” commits on the base on top of another branch. It will produce a linear history, easier to read.
Rebasing is also more general: you can select the commits to rebase (advanced topic).

When merging? When rebasing?

Rebase is useful if one wants to keep a linear history. It could be the method of choice to merge on an important branch work done on a only local topic branch.
Rebase however rewrites history changing commits. So it should never rebase using shared branches, where several developers are working.

Collaboration - remote

share a branch with a remote repo

```
$> git push <repo_name> <branch_name>
$> git pull <repo_name> <branch_name>
```

setup remote link

```
$> git config branch.<branch_name>.remote \
> <repo_name>
$> git config branch.<branch_name>.merge \
> refs/heads/<branch_name>
```

the others can access the branch with

```
$>  git checkout <branch_name> $>  git checkout
--track -b <branch_name> \
> <repo_name>/<branch_name>
```

Collaboration - local

share a branch locally

```
$>  git remote add ant_repo \
> /path/to/ant/git/repo
$>  git fetch ant_repo
$>  git checkout --track -b shared_branch \
> ant_repo/my_super_secret_branch
```

push modifications to the original branch

```
$>  git checkout shared_branch
$>  git push ant_repo shared_branch
```

or pull them from the original repo

```
$> git remote add nur_repo /path/to/nur/git/repo
$> git fetch nur_repo /path/to/nur/git/repo
$> git checkout my_super_secret_branch
$> git pull nur_repo/shared_branch
```

check for read/write permissions!

Fixing commits: amend reset and revert

Sometimes you want to correct your last commit

```
$> <do your additional work/add/rm>
$> git commit -amend
```

Never amend a commit that has been already pushed: use git revert

Reset

`git reset` sets yours HEAD to a specified commit. So it can be used to “delete commits”

```
$> git reset <file>
```

Removes the specified file from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes. It is the opposite of `git add`.

```
$> git reset
```

This unstages all files without overwriting any changes, giving you the opportunity to re-build the staged snapshot from scratch.

```
$> git reset <commit>
```

Move the current branch tip backward to <commit>, reset the staging area to match, but leave the working directory alone. All changes made since <commit> will reside in the working directory.

Hard Reset

```
$> git reset -hard
```

In addition to unstaging changes, the –hard flag tells Git to overwrite all changes in the working directory too. Put another way: this obliterates all uncommitted changes, so make sure you really want to throw away your local developments.

```
$> git reset -hard <commit>
```

Move the current branch tip backward to <commit> and reset both the staging area and the working directory to match. This obliterates not only the uncommitted changes, but all commits after <commit>, as well.

```
$> git reset [-|<commit>] [-hard] <paths>
```

resets all index entries of <paths> to their state at commit.

A warning

Reset may change the history. Do not use it on commits to “delete” that have already been pushed to the repository.

Revert

`git revert` reverts commits by playing them backwards. So it does not “delete” them. It is history safe.

```
$> git revert HEAD~3
```

reverts the changes specified by the fourth last commit in HEAD and create a new commit with the reverted changes.

It requires your working tree to be clean.

Get info

```
$> git log
```

Shows commit logs.

```
$> git shortlog
```

Summarizes log output, showing commit description from each contributor

```
$> git show <object>
```

Show some properties of the `object` (blob, commit...).

Use a graphical interface like `gitk`...

Common issues - I

Q edited on master branch by mistake, I wanted to do in new_branch instead!

Common issues - I

Q edited on master branch by mistake, I wanted to do in new_branch instead!

A a simple checkout will do it

```
$> git checkout -b new_branch
```

Common issues - I

Q edited on master branch by mistake, I wanted to do in new_branch instead!

A a simple checkout will do it

```
$> git checkout -b new_branch
```

Q already committed to master branch

Common issues - I

Q edited on master branch by mistake, I wanted to do in new_branch instead!

A a simple checkout will do it

```
$> git checkout -b new_branch
```

Q already committed to master branch

A create a new branch and delete the commit on master

```
$> git branch new_branch
$> git reset --hard HEAD~1
$> git checkout new_branch
```

Common issues - II

Q I want to discard my edit and get the latest version of the file!

Common issues - II

Q I want to discard my edit and get the latest version of the file!

A Simple

```
$> git checkout - file
```

Common issues - II

Q I want to discard my edit and get the latest version of the file!

A Simple

```
$> git checkout - file
```

Q I want to get the file from another branch or another commit!

Common issues - II

Q I want to discard my edit and get the latest version of the file!

A Simple

```
$> git checkout - file
```

Q I want to get the file from another branch or another commit!

A It’s the same command as before:

```
$> git checkout <branch|commit> - file
```

Common issues - II

Q Get a from another branch

Common issues - II

Q Get a from another branch

A cherry-pick it to the correct one

```
$> git checkout correct_branch
$> git cherry-pick commit_hash
```

Common issues - II

Q Get a from another branch

A cherry-pick it to the correct one

```
$> git checkout correct_branch
$> git cherry-pick commit_hash
```

Q cannot get the branch of the local remote

Common issues - II

Q Get a from another branch

A cherry-pick it to the correct one

```
$> git checkout correct_branch
$> git cherry-pick commit_hash
```

Q cannot get the branch of the local remote

A missing references to the other branches

```
$> git fetch remote_name
$> git remote show remote_name
```

Common issues - III

Q the merge has gone bananas!

Common issues - III

Q the merge has gone bananas!

A give up the merge and try again

```
$> git merge <branch_to_be_merged>
$> #$!&#%$&^
$> git reset --hard <original_branch>
```

Common issues - III

Q the merge has gone bananas!

A give up the merge and try again

```
$> git merge <branch_to_be_merged>
$> #$!&#%$&^
$> git reset --hard <original_branch>
```

or, more simply

```
git merge -abort
```

Q i want a file from that branch!

Common issues - III

Q the merge has gone bananas!

A give up the merge and try again

```
$> git merge <branch_to_be_merged>
$> #$!&#%$&^
$> git reset --hard <original_branch>
```

or, more simply

```
git merge -abort
```

Q i want a file from that branch!

A get it with

```
$> git checkout <branch> <files>
```

Common issues - IV

Q git push refuses to work

```
remote:  error:  refusing to update checked out branch:  refs/heads/<branch_name>
remote:  error:  By default, updating the current branch in a non-bare repository
remote:  error:  is denied, because it will make the index and work tree
inconsistent
remote:  error:  with what you pushed, and will require 'git reset -hard' to match
remote:  error:  the work tree to HEAD.
remote:  error:
remote:  error:  You can set 'receive.denyCurrentBranch' configuration variable to
remote:  error:  'ignore' or 'warn' in the remote repository to allow pushing into
remote:  error:  its current branch; however, this is not recommended unless you
remote:  error:  arranged to update its work tree to match what you pushed in some
remote:  error:  other way.
remote:  error:
remote:  error:  To squelch this message and still keep the default behaviour, set
remote:  error:  'receive.denyCurrentBranch' configuration variable to 'refuse'.
To <repo>
!  [remote rejected] <branch_name> -> <branch_name> (branch is currently checked
out)
```

Common issues - IV

Q git push refuses to work

```
remote:  error:  refusing to update checked out branch:  refs/heads/<branch_name>
remote:  error:  By default, updating the current branch in a non-bare repository
remote:  error:  is denied, because it will make the index and work tree
inconsistent
remote:  error:  with what you pushed, and will require 'git reset -hard' to match
remote:  error:  the work tree to HEAD.
remote:  error:
remote:  error:  You can set 'receive.denyCurrentBranch' configuration variable to
remote:  error:  'ignore' or 'warn' in the remote repository to allow pushing into
remote:  error:  its current branch; however, this is not recommended unless you
remote:  error:  arranged to update its work tree to match what you pushed in some
remote:  error:  other way.
remote:  error:
remote:  error:  To squelch this message and still keep the default behaviour, set
remote:  error:  'receive.denyCurrentBranch' configuration variable to 'refuse'.
To <repo>
!  [remote rejected] <branch_name> -> <branch_name> (branch is currently checked
out)
```

A use a bare repo

```
$> git clone -bare <repo> <repo>.git
```

http://stackoverflow.com/questions/2199897/git-convert-normal-to-bare-repository

Tips & tricks - I

- `gitk` is your friend

Tips & tricks - I

- `gitk` is your friend
- do small, atomic commits

Tips & tricks - I

- `gitk` is your friend
- do small, atomic commits
- read git output

Tips & tricks - I

- `gitk` is your friend
- do small, atomic commits
- read git output
- branch in `PS1`: add to `.bashrc`

```
function GitBranch {
  _branch="$(git branch 2>/dev/null | sed -e "/^\s/d"
  -e "s/^\*\s//")"
  test -n "$_branch" && echo -e "@$_branch"
}
PS1='\u@\h:\w$(GitBranch)> '
```

Tips & tricks - I

- `gitk` is your friend
- do small, atomic commits
- read git output
- branch in `PS1`: add to `.bashrc`

```
function GitBranch {
  _branch="$(git branch 2>/dev/null | sed -e "/^\s/d"
  -e "s/^\*\s//")"
  test -n "$_branch" && echo -e "@$_branch"
}
PS1='\u@\h:\w$(GitBranch)> '
```

- watch out for non history-safe commands after pushing! (`cherry-pick`, `rebase`,...)

Tips & tricks - I

- `gitk` is your friend
- do small, atomic commits
- read git output
- branch in `PS1`: add to `.bashrc`

```
function GitBranch {
  _branch="$(git branch 2>/dev/null | sed -e "/^\s/d"
  -e "s/^\*\s//")"
  test -n "$_branch" && echo -e "@$_branch"
}
PS1='\u@\h:\w$(GitBranch)> '
```

- watch out for non history-safe commands after pushing! (`cherry-pick`, `rebase`,...)
- everything else (and more...) on git

  `http://book.git-scm.com/index.html`

Tips & tricks - II

- bash autocompletion

  `http://git.kernel.org/?p=git/git.git;a=blob_plain;f=contrib/completion/git-completion.bash;hb=HEAD`

Tips & tricks - II

- bash autocompletion

  `http://git.kernel.org/?p=git/git.git;a=blob_plain;f=contrib/completion/git-completion.bash;hb=HEAD`

- colored output

```
$> git config --global color.ui true
```

Tips & tricks - II

- bash autocompletion

  `http://git.kernel.org/?p=git/git.git;a=blob_plain;f=contrib/completion/git-completion.bash;hb=HEAD`

- colored output

```
$> git config --global color.ui true
```

- no empty push

```
$> git config --global push.default nothing
```

Tips & tricks - II

- bash autocompletion

  `http://git.kernel.org/?p=git/git.git;a=blob_plain;f=contrib/completion/git-completion.bash;hb=HEAD`

- colored output

```
$> git config --global color.ui true
```

- no empty push

```
$> git config --global push.default nothing
```

- creating archives

```
$> git archive <commit_or_branch> | gzip > \
> <archive_name>.tgz
```