Self-Learning Points of Git

[1. What is Git 5](#_Toc446655230)

[1) Characteristics 5](#_Toc446655231)

[2) Advantage 6](#_Toc446655232)

[3) Disadvantage 6](#_Toc446655233)

[2. Key Points 6](#_Toc446655234)

[1) Stage and commit 6](#_Toc446655235)

[2) Branch is a named pointer to a commit, and HEAD is a pointer to a branch, i.e. Current branch. 7](#_Toc446655236)

[3) How to revert changes in git 7](#_Toc446655237)

[4) Git has 3 level config 8](#_Toc446655238)

[5) Git Merge 8](#_Toc446655239)

[6) Git Rebase 10](#_Toc446655240)

[7) Pull and Push 11](#_Toc446655241)

[8) Git Objects 12](#_Toc446655242)

[9) Git Refs 13](#_Toc446655243)

[10) Git Hooks 14](#_Toc446655244)

[11) Git 客户端和服务器交互协议：http, ssh, git协议 14](#_Toc446655245)

[3. Frequently Used Scenarios 16](#_Toc446655246)

[1) What’s SHA1? 16](#_Toc446655247)

[2) How to allow multiple people push code to a same repository? 16](#_Toc446655248)

[3) 如何由里程碑 A 对应的提交构造出一个根提交 16](#_Toc446655249)

[4) 如何在整个 git 仓库的提交历史中找寻内容并删除 16](#_Toc446655250)

[5) 如何生成块 blob 17](#_Toc446655251)

[6) 如何生成树tree 17](#_Toc446655252)

[7) 如何生成提交 commit 17](#_Toc446655253)

[8) 如何查看 tag 应用到哪些 object上 17](#_Toc446655254)

[9) 如何查看 SHA 值 18](#_Toc446655255)

[4. Frequently Used Commands 19](#_Toc446655256)

[1) git log 19](#_Toc446655257)

[2) git cat-file, git ls-tree, git write-tree 20](#_Toc446655258)

[3) git diff 22](#_Toc446655259)

[4) git stash 22](#_Toc446655260)

[5) git archive 22](#_Toc446655261)

[6) git dissect 二分查找 22](#_Toc446655262)

[7) git cherry-pick 22](#_Toc446655263)

[8) git rev-parse 获取当前check out 的 SHA1值 或者特定版本、分支或者refs的SHA1值 23](#_Toc446655264)

[9) git branch 24](#_Toc446655265)

[5. FAQ 25](#_Toc446655266)

[1) “detached from” 25](#_Toc446655267)

[2) “console font probably doesn't support Unicode” 27](#_Toc446655268)

[3) “missing \*\*\*” in git cat-file--batch / --batch-check output 28](#_Toc446655269)

[6. Training Materials 30](#_Toc446655270)

[1. Gerrit Concepts 31](#_Toc446655271)

[1) Push 31](#_Toc446655272)

[2) Push New Patch Set 32](#_Toc446655273)

[3) Gerrit Best Practices 32](#_Toc446655274)

[Table 1 Difference of using Soft, Mixed and Hard in git reset B 8](#_Toc446655275)

[Table 2 Examples of git log 20](#_Toc446655276)

[Table 3 Examples of git cat-file and git ls-tree 22](#_Toc446655277)

[Figure 1 Index in git repository: git add 6](#_Toc446398151)

[Figure 2 Folder objects in git repository: git commit 7](#_Toc446398152)

[Figure 3 Git has 3 level config 8](#_Toc446398153)

[Figure 4 Before Fast Forward merge 9](#_Toc446398154)

[Figure 5 After Fast Forward merge 9](#_Toc446398155)

[Figure 6 Before Cherry-Pick merge 9](#_Toc446398156)

[Figure 7 After Cherry-Pick merge 10](#_Toc446398157)

[Figure 8 Git before rebase 11](#_Toc446398158)

[Figure 9 Git after rebase: fast forward merge is possible 11](#_Toc446398159)

Git

# What is Git

## Characteristics

1. Git 是一个分布式版本管理系统
2. Git 是一个内容寻址系统 Content Tracking System
3. 有两个主要的数据结构： 对象库 object store， 索引 index。

(都位于 .git 下)

1. Index是一个临时的、动态的二进制文件。
2. 有4种类型的原子对象：

blob, tree, commit, tag.

|  |  |
| --- | --- |
| **Git 对象类型** | **描述** |
| blob | 二进制大对象 Binary Large Object  指某些可以保护任意数据的变量或者文件。  只保存数据，不包括元数据。 |
| tree | 代表一层目录信息。  包括：  blob标识符  路径名  在一个目录里所有文件的一些元数据  其他目录或者子树对象。 |
| commit | 提交。  每一个commit 对象指向一个 tree 对象。  保存版本库中每一次变化的元数据，  包括：  作者  提交者  提交日期  日志信息。 |
| tag | 一个标签对象分配一个任意且可读的名字给一个特定对象，通常是一个提交对象。  分为两种：轻量级 lightweight、带附注的 annotated。 |

1. Git 使用了一种叫做打包文件 pack file 的存储机制。

首先定位内容非常相似的全部文件，为它们之一存储整个内容。

之后计算相似文件之间的差异并且只存储差异。

1. Git 与传统数据库、Unix 文件系统的区别

|  |  |  |
| --- | --- | --- |
| **系统** | **索引机制** | **数据记录** |
| Git | .git/objects/hash、树对象内容 | blob对象、树对象 |
| 传统数据库 | 索引顺序存取方法 | 数据记录 |
| Unix 文件系统 | 目录/path/to/file | 数据块 |

## Advantage

1. A complete repository.

* No remote connection is necessary for:
  + Commit
  + History browsing
  + Switch to older revisions

1. Compared to Perforce, there is no per file checkout in Git.

⬄ Normally per file checkout in Git means revert.

TODO

## Disadvantage

TODO

# Key Points

## Stage and commit

Index = Stage Area

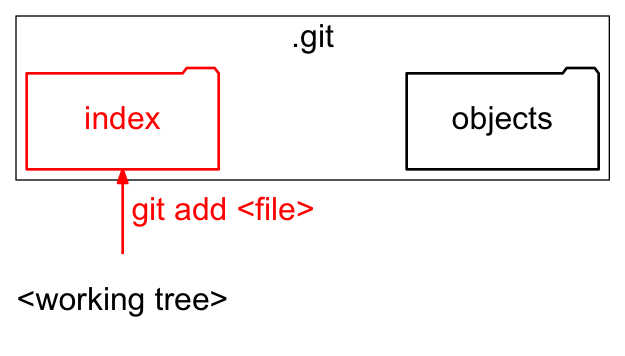


Figure 1 Index in git repository: git add

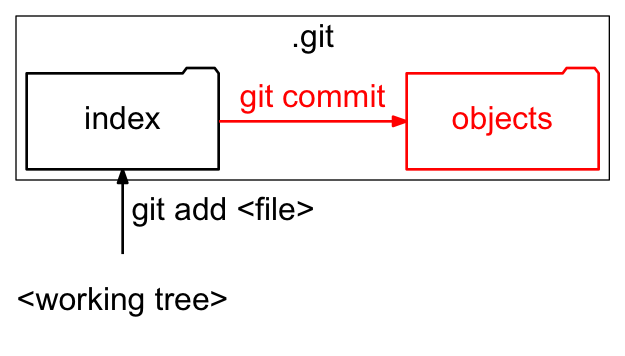


Figure 2 Folder objects in git repository: git commit

## Branch is a named pointer to a commit, and HEAD is a pointer to a branch, i.e. Current branch.

Ex. master. Its full name is refs/heads/master.

* <= This is why create/move/delete branch in git is quick.

Current branch is the branch pointed by HEAD

git checkout sets the current branch

git commit updates the current branch

git reset B reset current branch to commit B

git checkout - 切换到最后所在的分支

git branch --merged查看哪些分支已经合并进当前分支

git branch –merged | xargs git branch -d查看哪些分支已经合并进当前分支并删除

git branch –no-merged查看哪些分支没有合并进当前分支

git checkout (branch) -- (path/file)

= git stash $yourchanges, git checkout brach, git checkout path/file, git checkout -, git stash pop 从另一分支获取文件内容而不用切换分支

## How to revert changes in git

* When change is only in working directory, not staged yet.

TODO

* When change is staged, not committed yet.

TODO

* When change is committed.

TODO

* Difference of using Soft, Mixed and Hard in git reset B

|  |  |  |  |
| --- | --- | --- | --- |
| Reset | Working Directory | Stage (Index) | Remote Repo(Branch) |
| soft | No | No | Yes |
| mixed | No | Yes | Yes |
| hard | Yes | Yes | Yes |

Table 1 Difference of using Soft, Mixed and Hard in git reset B

## Git has 3 level config

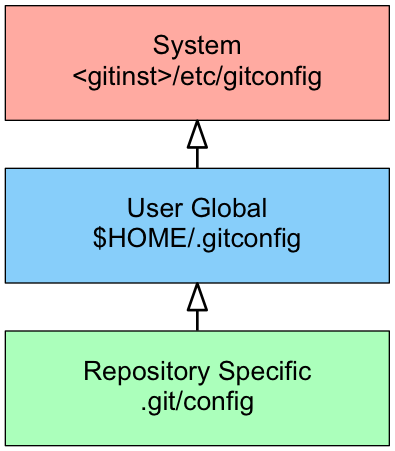


Figure 3 Git has 3 level config

## Git Merge

* Merge commit is a commit with more than one parent.
* git merge feature1 by default is Fast Forward merge, which just moves the pointer, no new merge commit is created.

**Fast Forward merge：当前分支的每一个提交都已经存在另一个分支里了。**

<= this is why merge in git is fast.

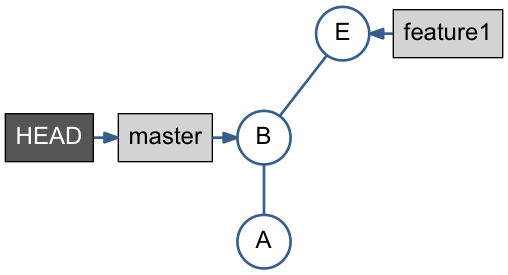


Figure 4 Before Fast Forward merge

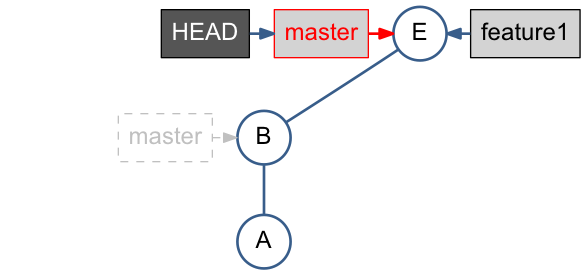


Figure 5 After Fast Forward merge

* git cherry-pick feature1 applies only changes done by F, means the diff-2, has no parent relation to F.

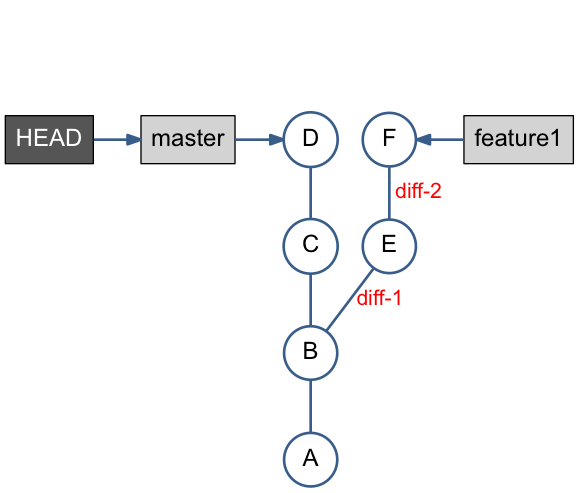
**

Figure 6 Before Cherry-Pick merge

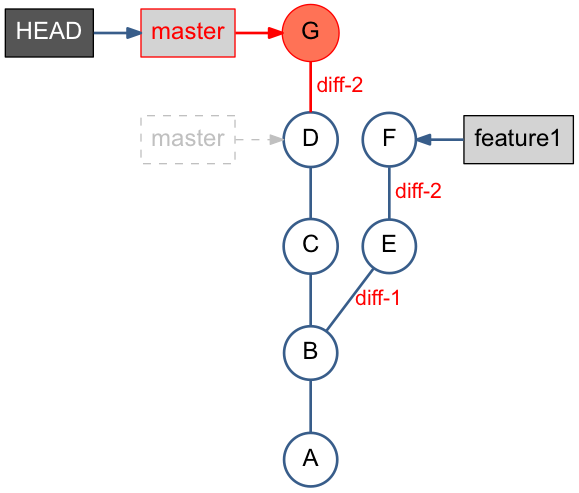
**

Figure 7 After Cherry-Pick merge

## Git Rebase

* **重新定义某个分支的参考基准.** 就好比移花接木那樣（稼接），把某個樹枝接到別的樹枝。

原理：*<since> …<till>* 所标识的提交范围写到一个临时文件中。然后逐一提交到变基后的分支上。

* git rebase master redo the work done in teature1 branch on top of the master*.*
* **Alternative to Merge** – Keeping history linear
* Fast Forward merge is possible after rebase.
* git rebase --onto *<new* base *commit> <current base commit>指定要從哪裡開始接枝*

[**https://blog.yorkxin.org/posts/2011/07/29/git-rebase/**](https://blog.yorkxin.org/posts/2011/07/29/git-rebase/)

* git rebase --onto *<new* base *commit> <since> <till>将指定范围内的提交嫁接到另外一个分支上*
* git rebase --i *interactive rebase. Could take one of the 6 actions on each commit.*

第一個欄位就是操作指令，指令的解釋在該檔案下方有：

pick = 要這條 commit ，什麼都不改

reword = 要這條 commit ，但要改 commit message

edit = 要這條 commit，但要改 commit 的內容

squash = 要這條 commit，但要跟前面那條合併，並保留這條的 messages

fixup = squash + 只使用前面那條 commit 的 message ，捨棄這條 message

exec = 執行一條指令（但我沒用過）

此外還可以調整 commits 的順序，直接剪剪貼貼，改行的順序就行了。

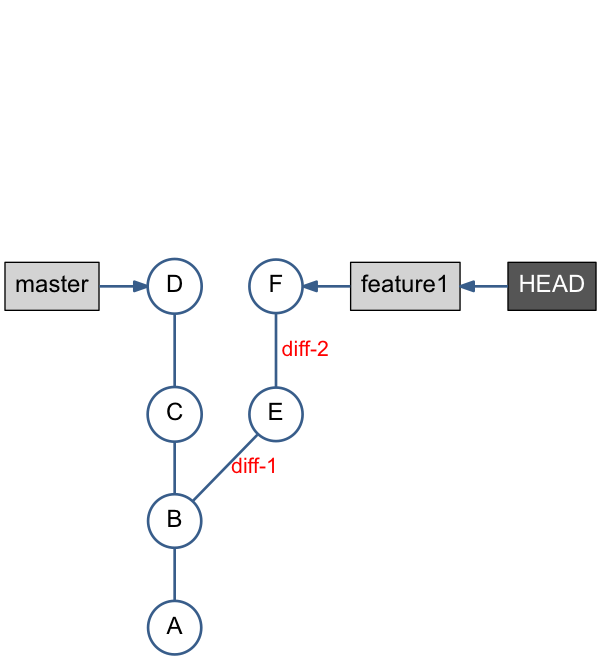
****

Figure 8 Git before rebase

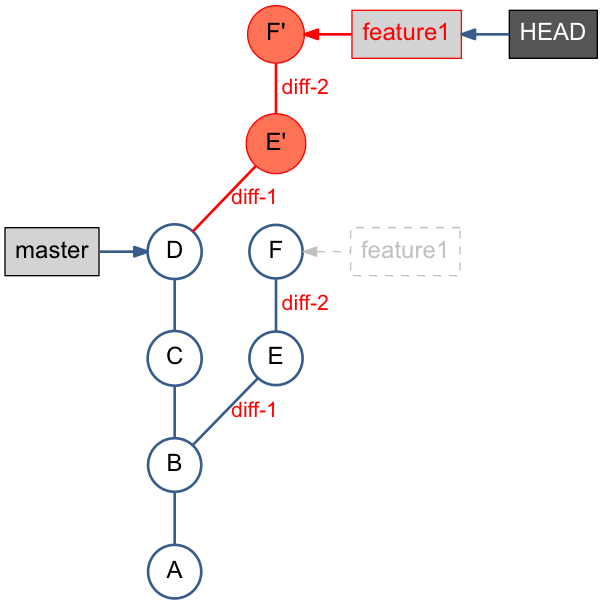
****

Figure 9 Git after rebase: fast forward merge is possible

## Pull and Push

Pull is equal to 3 possibilities:

* git pull = git fetch + git merge

or

* git pull = git fetch + git rebase

or

* git pull = git fetch

Push could mean to deal with 3 scenarios:

* push (when remote ‘origin’ repo is **NOT** changed)

or

* fetch, merge, push (when remote ‘origin’ repo is changed)

or

* fetch, rebase, push (when remote ‘origin’ repo is changed)

## Git Objects

* Git 对象 ID的类型：4种

Blob

Commit

Tree

Tag

可通过 git cat-file -t [SHA1] 查看。

* Git 对象以两种形式存储：

松散对象 loose object

打包对象 packed object

都是以 SHA 值为索引用 gzip 格式压缩的。

* 查看松散对象在对象库中的实际位置：

For id in [若干SHA1]; do ls .git/ojbects/${id:0:2}/${id:2}\*; done

* 查看对象类型、内容等：

git cat-file –t $SHA

git cat-file $type $SHA ($type = commit / blob / tree / tag)

git ls-tree $SHA

git ls-files –stage //列出索引即 index 中的内容

* 打包文件 packfile 和打包文件索引 packfile index：
* 打包文件 packfile 格式：

Header（包括 4bytes 的“PACK”,

4bytes 的版本号，

4bytes 的此文件中入口的个数）

一系列打包过的对象（每个都有自己的 header 和 body。

header：1bit是否还有后续字节，

3bits是 type，

4bits和可能的后续 7bits的字节，是数据展开后的长度。

）

trailer (即校验尾部)

* 打包文件索引 packfile index 基本上是一系列指向打包文件内位置的书签。格式：

header

fanout table 展开

sha1 listing

crc checksum

packfile offset

crc checksum of packfile and idxfile

## Git Refs

* branch, remote-tracking branch, tag etc. 都是对 commit 的 Reference

全写和简写的格式：

refs/heads/test <-> test

refs/tags/v1.0.0 <-> v1.0.0

refs/remotes/origin/master <-> origin/master

Tip: 仓库的名字可以代替该仓库的HEAD

例如：origin 是访问 origin 中HEAD 分支的一个捷径。

* Remote-tracking branch is read-only. Such as origin/master.

⬄ If you check it out, HEAD gets detached.

* 查看、更新refs：

git show-ref

git rev-parse HEAD

git pack-refs

git update-ref

* 查看分支 master 上特有的提交

gitk --branches master --not refs/heads/rel-5.25 refs/heads/rel-5.7

gitk --branches --not $( git show-ref --heads | cut -d' ' -f2 | grep -v '^refs/heads/master' ) // works in git bash.

git log master --not --remotes=\*/master

git log 1421525 --not --remotes=\*/1421525 --oneline

Shows all commits that are in local master but not in any remote repository master branches.

--branches 指定 local branches。

--not 对后面列出的 branch 取反，直到下一个 not。

--branches[=<pattern>]

Pretend as if all the refs in refs/heads are listed on the command line as <commit>. If <pattern> is given, limit branches to ones matching given shell glob. If pattern lacks ?, \*, or [, /\* at the end is implied.

--not

Reverses the meaning of the ^ prefix (or lack thereof) for all following revision specifiers, up to the next --not.

## Git Hooks

Git hooks need to be installed in 2 places:

* $Git\_installation\_path\share\git-core\templates\hooks\
* $Git\_repository\_parentpath\.git\hooks\

## Git 客户端和服务器交互协议：http, ssh, git协议

|  |  |  |  |
| --- | --- | --- | --- |
| **协议** | **前提** | **客户端** | **服务器** |
| **http** | Git 服务器的仓库里安装了 post-receive hook: 它会在仓库更新后启动 git update-server-info,来更新 objects/info/pack, info/refs.  http的优点：  所有的逻辑计算都在客户端进行。服务器上只要把 git 目录放到一个可访问的web 目录即可。 | 发出 git http-fetch …  （如果加上 –recover 会强制 git 反复检查我们是否拥有所有的对象。  更详细参见 git http-fetch. | 抓取流程：  Commit  -> tree of the commit  -> next parent of the commit  -> tree of the next parent of the commit  ->… |
| **Ssh / git** |  | 抓取流程：  socket connection  -> git fetch-pack  ->客户端回应：它所 需要和所拥有的 SHA 值。  上推流程：  git push 调用git send-pack    ->客户端生成 packfile， 传给服务器。 | 抓取流程：  服务器调用 link git: git update-pack :  <- 服务器告诉客户端它每个引用所拥有的 SHA 值。  <- 服务器把客户端需要的所有对象打包 packfile，传给客户端。  上推流程：  服务器调用 link git: git-receive-pack  <- 返回 all ref head SHAs |

# Frequently Used Scenarios

## What’s SHA1?

SHA1 is a globally unique commit ID

SHA1 is a function of the commit object content.

SHA1 is a 40-digit hexadecimal number, seen in git log output, git history view etc. ex. e168254… 23400e6…

## How to allow multiple people push code to a same repository?

Using core.sharedRepository to control access.

Ref: <http://criticallog.thornet.net/2010/01/07/sharing-your-git-repository/>

Steps:

* Create a group git, put multiple people into it.

useradd git

usermod -g git steven

useradd -g git nicholas

chmod 770 /home/git

chmod g+w objects/

* *Update .git/config, adding sharedRepository*

sharedRepository=1

加这个参数的目的是git在objects目录下创建的目录的属性由

drwxr-xr-x 变成 drwxrwsr-x

* Done.

## 如何由里程碑 A 对应的提交构造出一个根提交

方法1：

echo “Commit from tree of tag A.” | git commit-tree A^{tree}

方法2：

git cat-file commit A^0 | sed -e ‘/^parent/ d’ > tmpfile

git hash-object –t commit –w -- tmpfile

## 如何在整个 git 仓库的提交历史中找寻内容并删除

git rev-list --all | xargs git grep --F ‘strings\_to\_be\_found’ #搜寻所有的字符串

git filter-branch --index-filter ‘git rm --cached --ignore-unmatch (filename)’ --prune-empty --tag-name-filter cat -- --all #重新分支历史，移除敏感信息

echo (filename) >> .gitignore

git add .gitignore

git commit --m “Add sensitive (filename) file to gitignore”

git push origin master --force = git push origin +master

#把敏感信息记录到 .gitignore 文件并提交

rm --rf .git/refs/original/

git reflog expire --expire=now --all

git gc --prune=now

git gc --aggressive --prune=now

#从本地仓库中删除敏感信息

## 如何生成块 blob

git hash-object –w $filename

–w 即生成一个 blob 对象到 object database 里。

## 如何生成树tree

export GIT\_INDEX\_FILE=/tmp/index

git read-tree --prefix=copy1/ 8ba240929c40075a52c820493b3c68942adf4c89

git read-tree --prefix=copy2/ 8ba240929c40075a52c820493b3c68942adf4c89

$ git write-tree

571c685cb1947f78ef3aa44ff00ea269026598fb

查看树的信息和内容：

git ls-tree 571c685cb1947f78ef3aa44ff00ea269026598fb

git cat-file tree 571c685cb1947f78ef3aa44ff00ea269026598fb

## 如何生成提交 commit

TODO

## 如何查看 tag 应用到哪些 object上

git show --stat 5.26.0

git show --name-only 5.9.4

git show --name-status 5.9.4

查看一个commit上有哪些tag：

git tag --contains 5dd014bc72b1df4eb5db08f7d7a71a904ee94970

*查看所有的 tag：*

git show-ref –tags

git tag –n

git tag -l

git tag -l 5.26\*

*查看一个 tag 的详细信息：*

git cat-file tag 5.27.0

git show 5.27.0 --quiet

git show 5.27.0

# ref doc: <http://stackoverflow.com/questions/4185888/how-do-i-read-tagger-information-from-a-git-tag>

## 如何查看 SHA 值

Method 1. git hash-object $filename

Method 2. git cat-file --batch， git cat-file --batch-check命令

echo HEAD:pom.xml | git cat-file --batch | head

echo HEAD:pom.xml | git cat-file --batch-check

echo 'HEAD@{5 minutes ago}':pom.xml | git cat-file --batch-check

Ref: <https://github.com/git/git/blob/master/Documentation/git-cat-file.txt>

Method3. git rev-parse

ex.

git rev-parse HEAD

git rev-parse "FETCH\_HEAD^{commit}"

git rev-parse "refs/heads/master^{commit} "

git rev-parse "refs/heads/master^{object} "

## 如何查看一个 commit 中修改的文件列表和修改内容

Method 1. git diff --name-status HEAD~8..HEAD^

( git diff --name-status START\_COMMIT..END\_COMMIT )

git diff --name-status HEAD~8..HEAD^ | cut -f2 | xargs git diff HEAD~8..HEAD^ > log.diff

Method 2. git show --pretty="format:" --name-only 73f85e6

git show --name-only --oneline HEAD~8..HEAD^

*git reflog | grep -i Linux | awk '{print $1}' | xargs -l echo | git show --pretty="format:" --name-only*

*git reflog | grep -i Linux | awk '{print $1}' | xargs git show --pretty="format:" --name-only*

*git reflog | grep -i Linux | awk '{print $1}' | xargs git show --pretty="format:" --name-only | grep -v '^$' | sort -u*

Method 3. git diff-tree --no-commit-id --name-only -r 73f85e6

Ref doc: http://stackoverflow.com/questions/424071/how-to-list-all-the-files-in-a-commit

# Frequently Used Commands

## git log

Ref: <http://www.cnblogs.com/beginman/p/3577553.html>

|  |  |
| --- | --- |
| Command | Description |
| *git log --follow pom.xml* | *Shows commits that changed pom.xml, including those that occurred before the file was given its present name.* |
| *git log --name-only pom.xml* | *Shows the commits for file pom.xml, listing only name*  *--name-only 仅在提交信息后显示已修改的文件清单。* |
| *git log --name-status pom.xml* | *Shows the commits for file pom.xml, listing name and status*  *--name-status 显示新增、修改、删除的文件清单。* |
| *git log --stat pom.xml* | *Generates a diffsta.*  *显示简要的增改行数统计,每次提交文件的变更统计.* |
| *git log -p pom.xml* | *Similar to--stat, with more info.* |
| *git log -1 pom.xml* | *Limits the number of commits to show to 1* |
| *git shortlog pom.xml* | *Shows commit with only name and short description.* |
| *git log --pretty=oneline pom.xml* | *Set the output to oneline format*  *一行显示，只显示哈希值和提交说明。* |
| *git log --since=200.days -- pom.xml* | *Shows the changes during the last 200 days to the file pom.xml. “-- is necessary to avoid confusion with a branch named pom.xml.*  *--since, --after 仅显示指定时间之后的提交。* |
| *git log --until=200.days pom.xml* | *--until, --before 仅显示指定时间之前的提交。* |
| *git log rel-5.10 extensions/* | *Shows all commits since version rel-5.10 that changed any file in extensions/ subdirectories* |
| *git log --graph pom.xml* | *--graph 显示 ASCII 图形表示的分支合并历史。* |
| *git log master --not --remotes=\*/master* | *Shows all commits that are in local master but not in any remote repository master branches.* |
| *git log --decorate* | *参数 decorate 表示： 显示提交所属的里程碑tag* |
| *git log --until=200.days pom.xml* | *--until, --before 仅显示指定时间之前的提交。* |
| *git log rel-5.10 extensions/* | *Shows all commits since version rel-5.10 that changed any file in extensions/ subdirectories* |
| *git log --graph pom.xml* | *--graph 显示 ASCII 图形表示的分支合并历史。* |
| *git log --branches --not --remotes=origin* | *Shows all commits that are in any of local branches but not in any of remote-tracking branches for*origin*(what you have that origin doesn’t).* |
| *git log master --not --remotes=\*/master* | *Shows all commits that are in local master but not in any remote repository master branches.* |

Table 2 Examples of git log

## git cat-file, git ls-tree, git write-tree

*git cat-file* provide content or type and size information for repository objects.

Ref: <http://gitbook.liuhui998.com/7_2.html>

|  |  |
| --- | --- |
| Command | Description |
|  |  |
| *git cat-file -t 24aa5cbbe229*  显示类型 | Output:  *commit* |
| *git cat-file commit 24aa5cbbe229*  显示内容 | Output:  *tree 4fd4fbf360df0fc5d24c6fe5872f2592466887b6*  *parent f23fed367902342cb639d706869eb8dbb0904c3f*  *author Felix Xi <felix.xi@sap.com> 1438591623 +0800*  *committer Gerrit Code Review <gerrit@git.wdf.sap.corp> 1438594911 +0200*  *Summary: add demo page for legend text wrap*  *Change-Id: I4fa3021da2e1a30605ea1b233c0e1590f77a93ec*  *JIRA: BITSDC1-7210*  *Action: add* |
| *git ls-tree 24aa5cbbe229*  显示树的内容 | Output:  *100644 blob 05000bc34e374d540116fa83662108a6e96a1da0 .gitignore*  *100644 blob 326a99ef49bf772ea1d39b57eed3aa6a62287fe2 .gitreview*  *040000 tree 8a5a68e4982038d2ea5ce04a26ed6c7f88e1dc91 container*  *100644 blob b97b4bc1a3129e4f56d2435f42ce9d106ecea1b9 cvom.html\_context\_Visual Intelligence.xml*  *040000 tree a2ac5ce35313d2f9902151d1eb5634843b987d27 documentation*  *040000 tree a6475fe4436f68037507f42ce7da4bcbc7b56faa extension*  *040000 tree 3cd5614b2a8d7a89651f34d76f43c5f99d71a4ce fortify*  *040000 tree 4dcdbad168902aa45561c1947e9577ab22b960c4 info*  *100644 blob 55c0e6df5087017a5783d4ed13035e2a1925c13e pom.xml*  *040000 tree 60df7c4ecef2192d5aa9086a6040b33cff324337 testPnR*  *040000 tree 4182968183fad31ffc389b3e80e8862783610ee7 viz*  *040000 tree 2ea8a93bd3e2758031d0f61c37a5d1d056a5fe2d vizPacker* |
| *git cat-file -p master^{tree}* | Output:  100644 blob 05000bc34e374d540116fa83662108a6e96a1da0 .gitignore  100644 blob 5ebfd75fb43b68f670c5148e7cfa15834325eadb .gitreview  040000 tree e88c475df6e75cbaae601b866a7770948d233c3d container  100644 blob b97b4bc1a3129e4f56d2435f42ce9d106ecea1b9 cvom.html\_context\_Visual Intelligence.xml  040000 tree a2ac5ce35313d2f9902151d1eb5634843b987d27 documentation  040000 tree 3733cb7468f0aff8a21f91caee9b266e0d07bf5a extension  040000 tree 3cd5614b2a8d7a89651f34d76f43c5f99d71a4ce fortify  040000 tree d47fd2b8eb9eaeeb82f89d8a037c0eeaeae71d03 info  100644 blob 55c0e6df5087017a5783d4ed13035e2a1925c13e pom.xml  040000 tree 60df7c4ecef2192d5aa9086a6040b33cff324337 testPnR  040000 tree 93d31d1d7f2aaf4cac21735a858ee45c567a7cbe viz  040000 tree 2ea8a93bd3e2758031d0f61c37a5d1d056a5fe2d vizPacker |
| *git cat-file blob 05000bc34e374d5*  显示内容 | Output:  *\_src*  *.project*  *.classpath*  *.settings*  *\*.swp*  *target/*  */container/node\_modules*  *.DS\_Store*  *.idea/\**  *info/src/test/js/example/geo/demo\_map\_ex/jspm\_packages*  *info/src/test/js/example/geo/demo\_map\_ex/node\_modules* |
| *git cat-file blob 3f2e86e826c5bb5d2c2458b9d8ac8864cef8838a* | 等同于 git cat-file -p 3f2e86e826c5bb5d2c2458b9d8ac8864cef8838a |
| *git config --get-all user.name* | Get value of a given key. |
| *git config --global gc.auto 0* | Disable this behavior permanently without further considerations |
| *git gc* | Clean up unnecessary files and optimize the local repository |
| *git write-tree捕获索引状态并把它保存到一个树对象里* | *Output:*  *84d77628f1866d11cea523b04aece1f40ff9476c* |

Table 3 Examples of git cat-file and git ls-tree

## git diff

|  |  |
| --- | --- |
| **Command** | **Description** |
| *git diff* | *显示当前工作目录 和 上次提交*  *与本地索引之间的差异。*  *必须是修改。* |
| *git diff –cached* | *显示下次要提交的内容。* |

## git stash

保存进度。实际上会将进度保存在 refs/stash 所指向的提交中；

而 refs/stash 引用的变化由 reflog .git/logs/refs/stash 所记录下来。

## git archive

git archive -o latest.tar head基于最新提交建立归档文件

git archive -o partial.tar head src doc

git archive –format=tar –prefix=1.0/ v1.0 | grip > foo-1.0.tar.gz

## git dissect 二分查找

## git cherry-pick

git cherry-pick [commit]

git cherry-pick -C C 重用C提交的提交说明

## git rev-parse 获取当前check out 的 SHA1值 或者特定版本、分支或者refs的SHA1值

**git-rev-parse - Pick out and massage parameters**

git rev-parse "$VAR^{commit}" will make sure $VAR names an existing object that is a commit-ish (i.e. a commit, or an annotated tag that points at a commit).

To make sure that $VAR names an existing object of any type, git rev-parse "$VAR^{object}" can be used.

*<rev>^{<type>}*, e.g. *v0.99.8^{commit}*

A suffix *^* followed by an object type name enclosed in brace pair means dereference the object at *<rev>* recursively until an object of type *<type>* is found or the object cannot be dereferenced anymore (in which case, barf).

*<rev>:<path>*, e.g. *HEAD:README*, *:README*, *master:./README*

A suffix *:* followed by a path names the blob or tree at the given path in the tree-ish object named by the part before the colon. *:path* (with an empty part before the colon) is a special case of the syntax described next: content recorded in the index at the given path. A path starting with *./* or *../* is relative to the current working directory. The given path will be converted to be relative to the working tree’s root directory. This is most useful to address a blob or tree from a commit or tree that has the same tree structure as the working tree.

Ex.

git rev-parse HEAD

git rev-parse "FETCH\_HEAD^{commit}"

git rev-parse "refs/heads/master^{commit} "

git rev-parse "refs/heads/master^{object} "

git rev-parse "4eeef26185eda85751528f008190fff7d7dd9d68^{commit}"

git rev-parse --symbolic HEAD

git rev-parse --symbolic-full-name HEAD

git rev-parse @ # @ alone is a shortcut for HEAD.

git rev-parse "HEAD^1"

git rev-parse "HEAD^1^1^1" = git rev-parse "HEAD~3"

> git rev-parse "HEAD^{tree}"

e6964248ecf2f98f0d13ecc884f1ec82c8df32a5

> git rev-parse "HEAD^{commit}"

ceb705e72ae72354ecd4dd165341691aa7a8817d

> git cat-file -t e6964248ecf2f98f0d13ecc884f1ec82c8df32a5

tree

> git cat-file -t ceb705e72ae72354ecd4dd165341691aa7a8817d

commit

git rev-parse e6964248ecf2f98f0d13ecc884f1ec82c8df32a5:vizframe/src/main

# pre-requisite: path vizframe/src/main must exists in object e6964248ecf2f98f0d13ecc884f1ec82c8df32a5.

## git branch

**to create a new branch:**

git branch $branch\_name

git checkout = git checkout –b

**to set up a local branch to track remote branch from origin repository:**

Method 1. git branch –u

git branch -u origin/1421525 1421525

Method 2. git branch --set-upstream-to

git branch --set-upstream-to origin/1421525 1421525

Method 3. git push –u

At the first time to run git push:

git push -u origin my\_branch

Method 4. Set up a configuration item branch.autosetupmerge for future branches

git config --lobal branch.autosetupmerge always

which will link the upstream branch each time you create or checkout a new branch

ref doc:

<http://stackoverflow.com/questions/6089294/why-do-i-need-to-do-set-upstream-all-the-time/>

<https://felipec.wordpress.com/2013/09/01/advanced-git-concepts-the-upstream-tracking-branch/>

# FAQ

## “detached from”

### Phenomenon:

$ git branch

trace: built-in: git 'branch'

\* (detached from df5c9a7)

### Root cause & solution

<http://stackoverflow.com/questions/5772192/how-can-i-reconcile-detached-head-with-master-origin>

|  |  |
| --- | --- |
|  | First, let’s clarify what HEAD is and what it means when it is detached.  HEAD is the symbolic name for the currently checked out commit. When HEAD is not detached (the “normal”1 situation: you have a branch checked out), HEAD actually points to a branch’s “ref” and the branch points to the commit. HEAD is thus “attached” to a branch. When you make a new commit, the branch that HEAD points to is updated to point to the new commit. HEAD follows automatically since it just points to the branch.   * git symbolic-ref HEAD yields refs/heads/master The branch named “master” is checked out. * git rev-parse refs/heads/master yield 17a02998078923f2d62811326d130de991d1a95a That commit is the current tip or “head” of the master branch. * git rev-parse HEAD also yields 17a02998078923f2d62811326d130de991d1a95a This is what is means to be a “symbolic ref”. It points to an object through some other reference. (Symbolic refs were originally implemented as symbolic links, but later changed to plain files with extra interpretation so that they could be used on platforms that do not have symlinks.)   We have HEAD → refs/heads/master → 17a02998078923f2d62811326d130de991d1a95a  When HEAD is detached, it points directly to a commit—instead of indirectly pointing to one through a branch. You can think of a detached HEAD as being on an unnamed branch.   * git symbolic-ref HEAD fails with fatal: ref HEAD is not a symbolic ref * git rev-parse HEAD yields 17a02998078923f2d62811326d130de991d1a95a Since it is not a symbolic ref, it must point directly to the commit itself.   We have HEAD → 17a02998078923f2d62811326d130de991d1a95a  The important thing to remember with a detached HEAD is that if the commit it points to is otherwise unreferenced (no other ref can reach it), then it will become “dangling” when you checkout some other commit. Eventually, such dangling commits will be pruned through the garbage collection process (by default, they are kept for at least 2 weeks and may be kept longer by being referenced by HEAD’s reflog).  1 It is perfectly fine to do “normal” work with a detached HEAD, you just have to keep track of what you are doing to avoid having to fish dropped history out of the reflog.  The intermediate steps of an interactive rebase are done with a detached HEAD (partially to avoid polluting the active branch’s reflog). If you finish the full rebase operation, it will update your original branch with the cumulative result of the rebase operation and reattach HEAD to the original branch. My guess is that you never fully completed the rebase process; this will leave you with a detached HEAD pointing to the commit that was most recently processed by the rebase operation.  To recover from your situation, you should create a branch that points to the commit currently pointed to by your detached HEAD:  git branch temp  git checkout temp  (these two commands can be abbreviated as git checkout -b temp)  This will reattach your HEAD to the new temp branch.  Next, you should compare the current commit (and its history) with the normal branch on which you expected to be working:  git log --graph --decorate --pretty=oneline --abbrev-commit master origin/master temp  git diff master temp  git diff origin/master temp  (You will probably want to experiment with the log options: add -p, leave off --pretty=… to see the whole log message, etc.)  If your new temp branch looks good, you may want to update (e.g.) master to point to it:  git branch -f master temp  git checkout master  (these two commands can be abbreviated as git checkout -B master temp)  You can then delete the temporary branch:  git branch -d temp  Finally, you will probably want to push the reestablished history:  git push origin master  You many need to use --force to push if the remote branch can not be “fast-forwarded” to the new commit (i.e. you dropped, or rewrote some existing commit, or otherwise rewrote some bit of history).  If you were in the middle of a rebase operation you should probably clean it up. You can check whether a rebase was in process by looking for the directory .git/rebase-merge/. You can manually clean up the in-progress rebase by just deleting that directory (e.g. if you no longer remember the purpose and context of the active rebase operation). Usually you would use git rebase --abort, but that does some extra resetting that you probably want to avoid (it moves HEAD back to the original branch and resets it back to the original commit, which will undo some of the work we did above). |

## “console font probably doesn't support Unicode”

### Phenomenon:

C:\git\lumina-desktop\src\features\com.sap.hilo.desktop.composite.repository>git cat-file commit 4a0ac63

tree 6a89e043a1ef89139c7d2f5f15243d117bd7199b

parent 68b8e44f124852047cee8dccfcdd6d2f6c87334e

author Tintillier, Vivien <vivien.tintillier@sap.com> 1384424139 +0000

committer Tintillier, Vivien <vivien.tintillier@sap.com> 1384424139 +0000

Summary: duplicate CVOM code to try to workaround optional bundles dependencies when running tests from Tycho

Reviewed by: Tintillier, Vivien

What and how:

- NavteqLevels012cRepositoryFactory has optional dependencies on NavteqGeometriesLevels012cFactory and NavteqNamesLevels012cFactory

- since we changed the way we generate the p2 repository our tycho build and tests are based on, despite the optional bundles are here, they are not getting used at runtime in the tests

- one difference I could catch is that in the p2 metadata, those two optional bundles dependence are marked as optional=true and greedy=false

- it seems with p2 from eclipse 3.7.2, greedy was true

- I don't understand how it can impact the runtime, but let's try by not relying on those optional dependencies for now, and see if tests pass

Pending changeid: new

[git-p4: depot-paths = "//depot3/hilo.desktop/trunk/PI/": change = 5688910]

Former-commit-id: 5964fc496c15e75676d9b75631885648ce68b48c

Warning: Your console font probably doesn't support Unicode. If you experience strange characters in the output, consider switching to a TrueType font such as Lucida Console!

### Root cause & solution

Change font to ex. Lucida Console.

Context menu -> Defaults -> Font -> Lucida Console

Context menu -> Properties -> Font -> Lucida Console

### Reference

<http://www.xuebuyuan.com/202940.html>

## “missing \*\*\*” in git cat-file--batch / --batch-check output

### Phenomenon:

‘git cat-file --batch / --batch-check’ does not work on Windows.

* echo ffb99ee5db9d1a7bdec03bac70b454b4b5829b57 | git cat-file --batch

missingdb9d1a7bdec03bac70b454b4b5829b57

* echo HEAD:test/pom.xml | git cat-file --batch

missingt/pom.xml

while ‘git hash-object’ works on Windows and ‘git cat-file--batch’ works on Linux / Unix.

* git hash-object test\pom.xml

ffb99ee5db9d1a7bdec03bac70b454b4b5829b57

$ echo ffb99ee5db9d1a7bdec03bac70b454b4b5829b57 | git cat-file --batch-check

ffb99ee5db9d1a7bdec03bac70b454b4b5829b57 blob 4036

$ echo ffb99ee5db9d1a7bdec03bac70b454b4b5829b57 | git cat-file --batch | head -1

ffb99ee5db9d1a7bdec03bac70b454b4b5829b57 blob 4036

### Root cause & solution

Before git 2.8, March 2016, msysgit does NOT support ‘git cat-file --batch / --batch-check’ due to the newline issue.

On Windows in batch command EOL is ended with CRLF while LF is expected in git commands.

On Git Bash (the bash msysgit shell), EOL uses LF, so ‘git cat-file --batch / --batch-check’ works.

### Reference

<http://stackoverflow.com/questions/4688751/git-cat-file-output>

# Training Materials

<http://git.wdf.sap.corp:50002/presentations/git-gerrit-workshop/git-gerrit-workshop.html>

Gerrit

# Gerrit Concepts

## Push

* Push to Gerrit is the same like push to Git, with one Gerrit speciality “refs/for” in the target branch name.
* Gerrit creates internally a new branch for the commit and creates a new open change containing the pushed commit
* **Change** consists of
  + **Change ID** (important)
  + Metadata (owner, project, etc.)
  + One or more patch-sets
  + Comments
  + Votes (Build & Test results / Code Review)

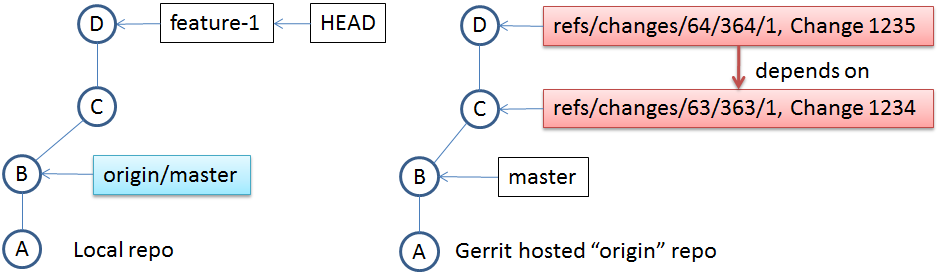
Patch set represents a Git commit.

|  |  |  |
| --- | --- | --- |
|  | Push to Git | Push to Gerrit |
| Command | *git push origin HEAD:master* | *git push origin HEAD:****refs/for/****master* |
|  |  |  |

* What if feature branch contained 2 commits?

Same for Git and Gerrit,

* + 2 commits in Git / ( 2 changes or 2 patch sets ) in Gerrit.



=> in Gerrit, a Change is bigger than a Patch Set.

a Patch Set represents a Git Commit.

## Push New Patch Set

* A Common Mistake
  + 1. author of the Patch Set 1 is not available and somebody else needs to continue and provide Patch Set 2
    2. use git pull to get the Patch Set 1 into a local branch
    3. Fix issues in commit
    4. Push (including the same Change-Id)
    5. Gerrit rejects!
    6. git pull origin refs/changes/66/366/1
    7. D is successor of C and can not be Patch Set 2
* The right way:
  + 1. fetch (don’t pull)
    2. create a new branch out of the fetched Patch Set 1
    3. fix the issue
    4. commit –amend
    5. push
    6. commit D is not successor of C
    7. D can become Patch Set 2

## Gerrit Best Practices

* One proposal feature, one feature branch.
* Push only finished features to branches with build and test automatics.
* Push complete feature as one commit ex. by squash.
* Prefer small yet complete changes to big one before pushing.
* Write good commit messages.