**Git进阶**

**基础命令**

**不妨尝试一下下面的实验:**

运行git add一定会让红色变为绿色吗?

|  |
| --- |
| Bash  cd /tmp  mkdir test-git1  cd test-git1    #create a new git repository, and commit 2 changes  git init  echo 1 > 1.txt  git add .  git commit -s -a -m "1st change"  echo 2 > 1.txt  git add .  git commit -s -a -m "2nd change"  git status  #On branch master  #nothing to commit, working tree clean    git log  # commit 4cf4037e6bb2016fe30d1af96788ef6f8499d516 (HEAD -> master)  # Author: Cheng Yang <[chengyang@xiaomi.com](http://chengyang@xiaomi.com)>  # Date: Sat Feb 12 20:04:22 2022 +0800  #  # 2nd change  #  # Signed-off-by: Cheng Yang <[chengyang@xiaomi.com](http://chengyang@xiaomi.com)>  #  # commit 889cc4292ce266e4607fd12aecb482cb89b1430d  # Author: Cheng Yang <[chengyang@xiaomi.com](http://chengyang@xiaomi.com)>  # Date: Sat Feb 12 20:04:22 2022 +0800  #  # 1st change  #  # Signed-off-by: Cheng Yang <[chengyang@xiaomi.com](http://chengyang@xiaomi.com)> |

上述命令到此为止, 应该很好理解, 创建了一个新的git仓库, 并且生成两笔change.

那么, 请问当你运行下面的命令, 显示结果里有几个红色, 几个绿色呢?

|  |
| --- |
| SQL  git reset HEAD~1 -- 1.txt  git status    # On branch master  # Changes to be committed:  # (use "git restore --staged <file>..." to unstage)  # modified: 1.txt  #  # Changes not staged for commit:  # (use "git add <file>..." to update what will be committed)  # (use "git restore <file>..." to discard changes in working directory)  # modified: 1.txt |

一个红色, 一个绿色? 也许到这一步你并不意外, 那么请问, 当你再运行git add的时候, 是几个红色几个绿色? 一个绿色? 一个红色?

|  |
| --- |
| Bash  git add .  git status    # On branch master  # nothing to commit, working tree clean |

为什么运行git add会让原本不干净的仓库变为clean?

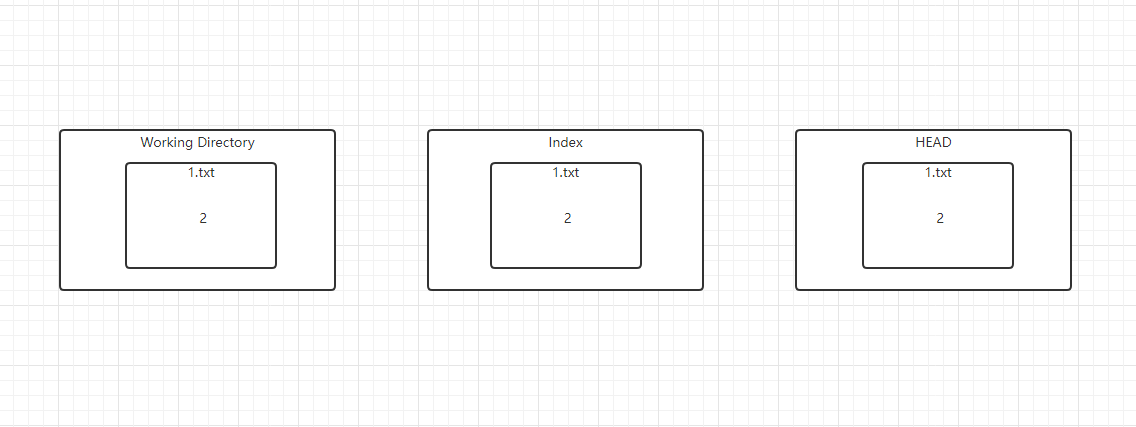
**WD, Index, HEAD:**

git的基础命令, 就是操作这三个区域. 相信大家已经或多或少看过. Working Directory(工作区), Index(索引区), HEAD(当前所在commit). 那么我现在要澄清两件事

1. git任何命令, 只会在这三个区域**复制.** 没错, 就是复制, 并不会计算差异
2. Index不是暂存, 不是暂存, 不是暂存. 重要的事情说三遍.

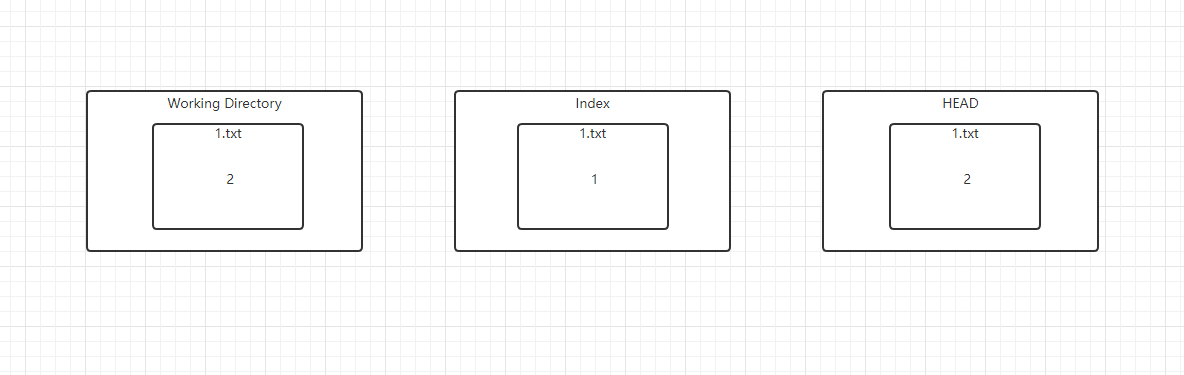
现在我们来看一看, 在刚才的实验中. 这三个区域的变化是什么. 当你的仓库运行完git commit之后, 应该是nothing to commit, working tree clean的状态. 这时候, 这三者的状态是这样的.

你会发现, 此时Index并不是空的, 他和WD还有HEAD里的内容是一样的. 他并不是什么"暂存"区域.

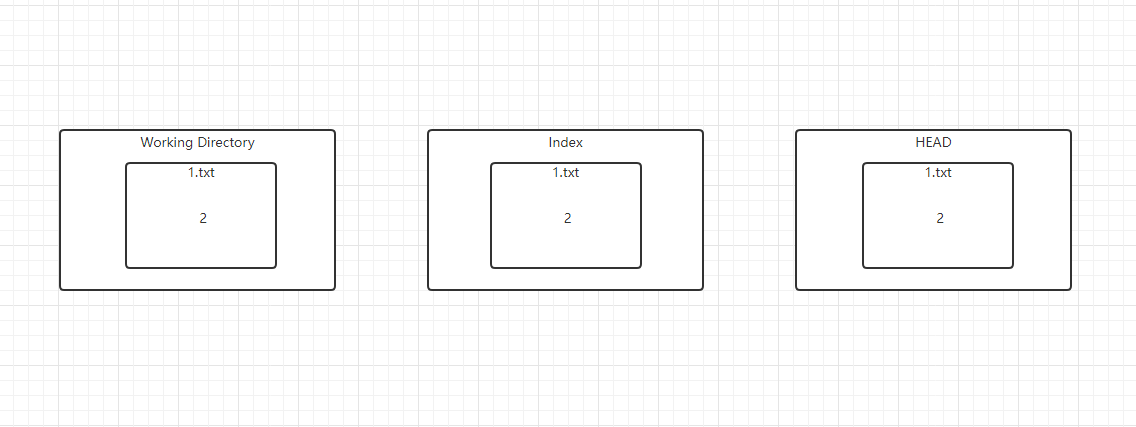


那么当你执行完git reset HEAD~1 -- 1.txt的结果是什么呢?

根据git官方文档, git reset命令之后如果带有具体的文件名. 那么他会将指定HEAD里的文件复制到Index里, 但是不改变HEAD



那么git add又是什么意思呢? 根据官方文件它的功能实际上是: 将Working Directory里的内容, 复制到Index里. 那么很显然, 当你在上述状态下运行git add. 最后会让它回到初始状态.



理解了WD, Index, Head之后, 我们现在再来简单列一下git的常用命令的效果

1. git status: 比较WD和Index区域, 并将差异显示为红色, 然后比较Index和HEAD, 将差异显示为绿色
2. git reset. 不带任何参数, 是将指定commit里的所有内容, 复制到Index, 并且将HEAD指针移动到指定commit.

当加上--hard参数, 除了上述操作之外, 还会将所有内容也复制到WD里.

当不加--hard, 同时指定文件名的时候. 会将指定commit里这个文件的内容复制到Index, 但是不移动HEAD指针

1. git checkout在不加任何参数的情况下, 大多数行为和git reset相同.
2. git add.将任何指定内容, 从WD复制到Index中
3. git commit创建一个新的commit, 将Index内容复制到其中, 然后将HEAD指针指向这个commit(可以近似理解为将Index复制到HEAD中)

**Git的底层原理:**

**Blob 对象**

git在本质上就是一个键值对(key-value)的数据库. key是根据内容计算出来的hash值. value则是你需要存储的数据. 你可以使用git hash-object来确认一段数据的key.

在此之前, 请重新建一个新的git仓库

|  |
| --- |
| Bash  cd /tmp  mkdir test-git2  cd test-git2  git init |

|  |
| --- |
| Bash  echo 'test content' | git hash-object -w --stdin  # d670460b4b4aece5915caf5c68d12f560a9fe3e4 |

你这时候会发现你的.git/objects目录下面多了一个文件

|  |
| --- |
| Bash  ls -l .git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4  # -r--r--r-- 1 cy cy 32 Feb 11 13:19 .git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 |

在这个文件中, 存储的就是一些元信息和test content这个文件内容. 这个文件具体的内容, 可以查看[这里](https://git-scm.com/book/zh/v2/Git-%E5%86%85%E9%83%A8%E5%8E%9F%E7%90%86-Git-%E5%AF%B9%E8%B1%A1#_%E5%AF%B9%E8%B1%A1%E5%AD%98%E5%82%A8)

存储完了之后, 你也可以用git cat-file命令读取它

|  |
| --- |
| Apache  git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4  # test content |

你现在应该已经能初步理解git是如何存储信息的了. 任意数据你都可以存到git里. 它是最原始的信息, 我们把这种对象称之为blob object

**Tree 对象**

敏锐的同学应该已经发现了, 那文件名存在哪里? 答案是git会单独用另外一种object来存储

|  |
| --- |
| Bash  git update-index --add --cacheinfo 100644 d670460b4b4aece5915caf5c68d12f560a9fe3e4 test.txt |

此时你使用运行git status会发现Index区域变了

|  |
| --- |
| Bash  git status  # HEAD detached from f8f34a1  # Changes to be committed:  # (use "git restore --staged <file>..." to unstage)  # new file: test.txt  #  # Changes not staged for commit:  # (use "git add/rm <file>..." to update what will be committed)  # (use "git restore <file>..." to discard changes in working directory)  # deleted: test.txt  # |

git add实际上就是git hash-object和git update-index的组合. 实际上是把d670460b4b4aece5915caf5c68d12f560a9fe3e4 这个object复制到了Index区域

接下来就可以使用git write-tree命令.将Index区域的内容写入到一个Tree Object中.

|  |
| --- |
| Bash  git write-tree  # 80865964295ae2f11d27383e5f9c0b58a8ef21da  git cat-file -p 80865964295ae2f11d27383e5f9c0b58a8ef21da  # 100644 blob d670460b4b4aece5915caf5c68d12f560a9fe3e4 test.txt |

**Commit 对象**

有了树对象, 我们就可以创建一个提交对象了

|  |
| --- |
| Bash  echo 'first commit' | git commit-tree 80865964295ae2f11d27383e5f9c0b58a8ef21da  e137f356afa88b2b8984c4105326d74826a8c5c1 |

|  |
| --- |
| Bash  git log e137f356afa88b2b8984c4105326d74826a8c5c1    # commit e137f356afa88b2b8984c4105326d74826a8c5c1  # Author: Cheng Yang <[chengyang@xiaomi.com](http://chengyang@xiaomi.com)>  # Date: Fri Feb 11 19:39:09 2022 +0800    # first commit |

至此, 我们在不使用git add和git commit的情况下, 创建了一个提交

**反过来看看Commit 对象**

git里, 万物皆对象.凡是有hash的东西一定是一个对象. 那么我们来从后往前看看, 一个提交里到底有什么?

|  |
| --- |
| Apache  git cat-file -p e137f356afa88b2b8984c4105326d74826a8c5c1  # tree 80865964295ae2f11d27383e5f9c0b58a8ef21da  # author Cheng Yang <[chengyang@xiaomi.com](http://chengyang@xiaomi.com)> 1644579549 +0800  # committer Cheng Yang <[chengyang@xiaomi.com](http://chengyang@xiaomi.com)> 1644579549 +0800    # first commit |

你会发现, Commit Object也不过是一个普通object, 只不过内容是存储了提交者, 作者, 时间戳, 以及这个提交所包含的Tree Object. 一如你运行git log所看到的内容

那么我们再次运行cat-file查看这个Tree Object

|  |
| --- |
| Bash  git cat-file -p 80865964295ae2f11d27383e5f9c0b58a8ef21da  # 100644 blob d670460b4b4aece5915caf5c68d12f560a9fe3e4 test.txt |

很显然, 和刚才一样, 这个Tree Object的意思是包含test.txt这个文件, 文件内容存储在d670460b4b4aece5915caf5c68d12f560a9fe3e4 这个对象中

那么再继续看看这个对象

|  |
| --- |
| Bash  git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4  # test content |

所以, 这个Commit Object包含了一个Tree Object, 而这个Tree Object包含了一个叫test.txt, 而test.txt文件里的内容是test content.

这里你会注意到, 我们的硬盘中甚至从头到尾都没有出现过一个叫test.txt的文件

**传输协议**

**Fetch:**

理解git的传输协议, 我们可以使用GIT\_TRACE\_PACKET宏, 让git能自动打印出所有发送的网络包

|  |
| --- |
| Bash  cd /tmp  mkdir fetch-test  cd fetch-test    GIT\_TRACE\_PACKET=1 git -c protocol.version=1 clone /tmp/test-git1 |

这个地方设置protocol.version=1的原因是因为git默认使用v2协议还是2020年的事, 相当数量的服务器并未支持v2协议, 而且v2协议协商过程会稍微复杂一些, 所以我们使用比较常见的v1先来解释git如何远程传输数据

|  |
| --- |
| Bash  Cloning into 'test-git1'...  20:09:07.319608 pkt-line.c:80 packet: upload-pack> version 1  20:09:07.319670 pkt-line.c:80 packet: clone< version 1  20:09:07.319761 pkt-line.c:80 packet: upload-pack> 4cf4037e6bb2016fe30d1af96788ef6f8499d516 HEAD\0multi\_ack thin-pack side-band side-band-64k ofs-delta shallow deepen-since deepen-not deepen-relative no-progress include-tag multi\_ack\_detailed allow-tip-sha1-in-want allow-reachable-sha1-in-want symref=HEAD:refs/heads/master object-format=sha1 agent=git/2.35.1  20:09:07.319874 pkt-line.c:80 packet: upload-pack> 4cf4037e6bb2016fe30d1af96788ef6f8499d516 refs/heads/master  20:09:07.319886 pkt-line.c:80 packet: upload-pack> 0000  20:09:07.319908 pkt-line.c:80 packet: clone< 4cf4037e6bb2016fe30d1af96788ef6f8499d516 HEAD\0multi\_ack thin-pack side-band side-band-64k ofs-delta shallow deepen-since deepen-not deepen-relative no-progress include-tag multi\_ack\_detailed allow-tip-sha1-in-want allow-reachable-sha1-in-want symref=HEAD:refs/heads/master object-format=sha1 agent=git/2.35.1  20:09:07.319925 pkt-line.c:80 packet: clone< 4cf4037e6bb2016fe30d1af96788ef6f8499d516 refs/heads/master  20:09:07.319929 pkt-line.c:80 packet: clone< 0000  done.  20:09:07.321498 pkt-line.c:80 packet: clone> 0000  20:09:07.321532 pkt-line.c:80 packet: upload-pack< 0000 |

<表示收到, >表示发送. upload-pack>的意思就是upload-pack发送了数据. 而upload-pack是服务端上的进程, 其实意思也就是服务端发送了数据.

在协商协议版本后, 远程服务器发送的第一项数据是一些元数据和能力, 主要是告诉客户端, 服务端支持哪些功能. 在此处我们不深入此处. 而第二项是最重要的

4cf4037e6bb2016fe30d1af96788ef6f8499d516 refs/heads/master

这个实际上是在告诉客户端, 服务端所拥有的所有refs.在此处我们的服务器只有master这么一个分支, 它指向4cf4037e6bb2016fe30d1af96788ef6f8499d516 这个Commit Object. 那么后面的传输信息也很好理解了. 就是客户端在告诉服务器, 我需要的是refs/heads/master这个ref上的所有内容.

clone的过程比较简单, 那么增量fetch呢? 我们不妨给test-git1添加一个change

|  |
| --- |
| Bash  cd /tmp/test-git1  echo 3 > 1.txt  git commit -a -s -m "3rd change" |

此时我们再执行fetch

|  |
| --- |
| Bash  cd /tmp/fetch-test/test-git1  GIT\_TRACE\_PACKET=1 git -c protocol.version=1 fetch |

|  |
| --- |
| Bash  20:13:55.779809 pkt-line.c:80 packet: upload-pack> version 1  20:13:55.779937 pkt-line.c:80 packet: fetch< version 1  20:13:55.780221 pkt-line.c:80 packet: upload-pack> 3fb2316cbe6a7594b981c9bf8ef2fd08159a4b90 HEAD\0multi\_ack thin-pack side-band side-band-64k ofs-delta shallow deepen-since deepen-not deepen-relative no-progress include-tag multi\_ack\_detailed allow-tip-sha1-in-want allow-reachable-sha1-in-want symref=HEAD:refs/heads/master object-format=sha1 agent=git/2.35.1  20:13:55.780500 pkt-line.c:80 packet: upload-pack> 3fb2316cbe6a7594b981c9bf8ef2fd08159a4b90 refs/heads/master  20:13:55.780531 pkt-line.c:80 packet: upload-pack> 0000  20:13:55.780549 pkt-line.c:80 packet: fetch< 3fb2316cbe6a7594b981c9bf8ef2fd08159a4b90 HEAD\0multi\_ack thin-pack side-band side-band-64k ofs-delta shallow deepen-since deepen-not deepen-relative no-progress include-tag multi\_ack\_detailed allow-tip-sha1-in-want allow-reachable-sha1-in-want symref=HEAD:refs/heads/master object-format=sha1 agent=git/2.35.1  20:13:55.780596 pkt-line.c:80 packet: fetch< 3fb2316cbe6a7594b981c9bf8ef2fd08159a4b90 refs/heads/master  20:13:55.780616 pkt-line.c:80 packet: fetch< 0000  20:13:55.781535 pkt-line.c:80 packet: fetch> want 3fb2316cbe6a7594b981c9bf8ef2fd08159a4b90 multi\_ack\_detailed side-band-64k thin-pack include-tag ofs-delta deepen-since deepen-not agent=git/2.35.1  20:13:55.781553 pkt-line.c:80 packet: fetch> 0000  20:13:55.781617 pkt-line.c:80 packet: fetch> have 4cf4037e6bb2016fe30d1af96788ef6f8499d516  20:13:55.781612 pkt-line.c:80 packet: upload-pack< want 3fb2316cbe6a7594b981c9bf8ef2fd08159a4b90 multi\_ack\_detailed side-band-64k thin-pack include-tag ofs-delta deepen-since deepen-not agent=git/2.35.1  20:13:55.781628 pkt-line.c:80 packet: fetch> have 889cc4292ce266e4607fd12aecb482cb89b1430d  20:13:55.781635 pkt-line.c:80 packet: fetch> done  20:13:55.781800 pkt-line.c:80 packet: upload-pack< 0000  20:13:55.781819 pkt-line.c:80 packet: upload-pack< have 4cf4037e6bb2016fe30d1af96788ef6f8499d516  20:13:55.781944 pkt-line.c:80 packet: upload-pack> ACK 4cf4037e6bb2016fe30d1af96788ef6f8499d516 common  20:13:55.781971 pkt-line.c:80 packet: upload-pack< have 889cc4292ce266e4607fd12aecb482cb89b1430d  20:13:55.782004 pkt-line.c:80 packet: fetch< ACK 4cf4037e6bb2016fe30d1af96788ef6f8499d516 common  20:13:55.782061 pkt-line.c:80 packet: upload-pack> ACK 889cc4292ce266e4607fd12aecb482cb89b1430d common  20:13:55.782094 pkt-line.c:80 packet: upload-pack< done  20:13:55.782109 pkt-line.c:80 packet: upload-pack> ACK 889cc4292ce266e4607fd12aecb482cb89b1430d  20:13:55.782122 pkt-line.c:80 packet: fetch< ACK 889cc4292ce266e4607fd12aecb482cb89b1430d common  20:13:55.782151 pkt-line.c:80 packet: fetch< ACK 889cc4292ce266e4607fd12aecb482cb89b1430d  20:13:55.784146 pkt-line.c:80 packet: sideband< \2Enumerating objects: 5, done.  remote: Enumerating objects: 5, done.  20:13:55.784171 pkt-line.c:80 packet: sideband< \2Counting objects: 20% (1/5)\15  20:13:55.784180 pkt-line.c:80 packet: sideband< \2Counting objects: 40% (2/5)\15  20:13:55.784193 pkt-line.c:80 packet: sideband< \2Counting objects: 60% (3/5)\15  20:13:55.784200 pkt-line.c:80 packet: sideband< \2Counting objects: 80% (4/5)\15  20:13:55.784208 pkt-line.c:80 packet: sideband< \2Counting objects: 100% (5/5)\15Counting objects: 100% (5/5), done.  remote: Counting objects: 100% (5/5), done.  20:13:55.784401 pkt-line.c:80 packet: sideband< \2Total 3 (delta 0), reused 0 (delta 0), pack-reused 0  remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0  20:13:55.784421 pkt-line.c:80 packet: sideband< PACK ...  20:13:55.784526 pkt-line.c:80 packet: upload-pack> 0000  20:13:55.784575 pkt-line.c:80 packet: sideband< 0000  Unpacking objects: 100% (3/3), 360 bytes | 360.00 KiB/s, done.  From /tmp/test-git1  4cf4037..3fb2316 master -> origin/master |

最开始, 仍旧是服务器发送了他拥有的refs

upload-pack> 3fb2316cbe6a7594b981c9bf8ef2fd08159a4b90 refs/heads/master

此时, 客户端收到了之后, 就和自己的refs/heads/master做对比. 它告诉了服务器, 我需要哪个commit, 已经有了哪个commit

|  |
| --- |
| Bash  20:13:55.781535 pkt-line.c:80 packet: fetch> want 3fb2316cbe6a7594b981c9bf8ef2fd08159a4b90 multi\_ack\_detailed  20:13:55.781553 pkt-line.c:80 packet: fetch> 0000  20:13:55.781617 pkt-line.c:80 packet: fetch> have 4cf4037e6bb2016fe30d1af96788ef6f8499d516  20:13:55.781628 pkt-line.c:80 packet: fetch> have 889cc4292ce266e4607fd12aecb482cb89b1430d |

这时候, git服务器的最优做法, 是指发送客户端没有的东西, 而不是发送整个的refs/heads/master

接下来的Enumerating Objects和Couting Objects都是在整理"服务端有, 客户端没有的东西". 并且发送回来.

**Push:**

push的过程其实相对来说就是fetch的逆向操作

|  |
| --- |
| Apache  cd /tmp/fetch-test/test-git1  git checkout origin/master  echo 4 > 1.txt  git commit -a -s -m "4th change"    GIT\_TRACE\_PACKET=1 git -c protocol.version=1 push origin HEAD:master |

这条命令的意思是, 把本地HEAD指向的内容, push到远程refs/heads/master上.这里可以注意下, 如果你想, 你甚至可以把HEAD替换为其他任意hash值

输出会有一些报错, 只是因为test-git1不是bare仓库. 这不影响我们分析

|  |
| --- |
| Bash  20:30:46.011723 pkt-line.c:80 packet: receive-pack> version 1  20:30:46.011807 pkt-line.c:80 packet: push< version 1  20:30:46.011846 pkt-line.c:80 packet: receive-pack> 3fb2316cbe6a7594b981c9bf8ef2fd08159a4b90 refs/heads/master\0report-status report-status-v2 delete-refs side-band-64k quiet atomic ofs-delta object-format=sha1 agent=git/2.35.1  20:30:46.011865 pkt-line.c:80 packet: receive-pack> 0000  20:30:46.011867 pkt-line.c:80 packet: push< 3fb2316cbe6a7594b981c9bf8ef2fd08159a4b90 refs/heads/master\0report-status report-status-v2 delete-refs side-band-64k quiet atomic ofs-delta object-format=sha1 agent=git/2.35.1  20:30:46.011878 pkt-line.c:80 packet: push< 0000  20:30:46.012115 pkt-line.c:80 packet: push> 3fb2316cbe6a7594b981c9bf8ef2fd08159a4b90 37607550824f6769a97caa25d06a0335970abdb5 refs/heads/master\0 report-status-v2 side-band-64k object-format=sha1 agent=git/2.35.1  20:30:46.012125 pkt-line.c:80 packet: push> 0000  20:30:46.012145 pkt-line.c:80 packet: receive-pack< 3fb2316cbe6a7594b981c9bf8ef2fd08159a4b90 37607550824f6769a97caa25d06a0335970abdb5 refs/heads/master\0 report-status-v2 side-band-64k object-format=sha1 agent=git/2.35.1  20:30:46.012163 pkt-line.c:80 packet: receive-pack< 0000  Enumerating objects: 5, done.  Counting objects: 100% (5/5), done.  Writing objects: 100% (3/3), 380 bytes | 380.00 KiB/s, done.  Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 |

当push收到了服务端refs/heads/master的内容之后

|  |
| --- |
| Bash  push< 3fb2316cbe6a7594b981c9bf8ef2fd08159a4b90 refs/heads/master\0report-status report-status-v2 delete-refs side-band-64k quiet atomic ofs-delta object-format=sha1 agent=git/2.35.1 |

然后对比自己所拥有的commit. 决定发送内容

|  |
| --- |
| Bash  push> 3fb2316cbe6a7594b981c9bf8ef2fd08159a4b90 37607550824f6769a97caa25d06a0335970abdb5 refs/heads/master\0 report-status-v2 side-band-64k object-format=sha1 agent=git/2.35.1 |

这个packet的意思是, 客户端会发送

3fb2316cbe6a7594b981c9bf8ef2fd08159a4b90和37607550824f6769a97caa25d06a0335970abdb5之间的内容.

接下来和fetch如出一辙, 客户端开始计算这两笔commit之前所需要传送的object, 打包之后发送给服务器

**引用:**

1. [ProGit](https://git-scm.com/book/zh/v2): Git开源社区官方维护的书籍
2. [Git Protocol v2](https://git-scm.com/docs/protocol-v2): git传输协议的官方文档