Git 命令家底儿及 Git 数据通信原理详解

本文作者周立伟（目前就职于阿里巴巴 花名：华序）关注分布式、高并发和 java 中间件的研究

iteye 博客：http://mycolababy.iteye.com

新浪微博：http://weibo.com/alihuaxu

Git 是一款开源的分布式版本控制系统(VCS)，常用的 VCS 工具还包括 SVN、Mercurial 等，他们的使命是对资源变化的进行版本管理控制，对资源容灾备份，支持多域协同开发。这里的资源不仅仅是系统代码，还包括图片、文件、网页等。本篇文章结合流程图、详细的注解、实例操作针对 Git 的使用、Git 数据通信原理进行细致的讲解，利用半场足球赛的时间通读全文后相信你面对 Git 会自信满满、知其所以然，使用起来游刃有余，当然对其他工具的理解也就非常容易了。

Git 在各个操作系统的安装过程就不缀文了，步骤都是固定的，按照步骤一步一步安装就可以了。在开始讲解之前，我们先对 Git 进行资源版本管理有个整体的了解，如图 1 为 Git 资源状态流转过程，理解清楚这个流转图对 Git 命令的操作非常关键。

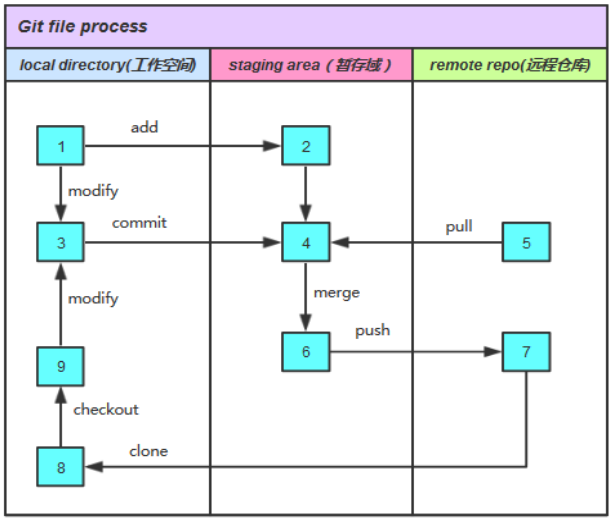

图1 Git资源状态流转

接下来，我们开始梦幻的 Git 命令之旅，了解下在项目协同工作过程中经常使用的 Git 命令。以下是我在工作过程中的总结(可以说是我的 Git 家底儿了)，为了更清晰的展示，将命令操作分成了有序的多块，大家在阅读过程中可能会感觉到一丝笔记的色彩，没错，这就是这篇文章的特点，我相信这种方式会更利于大家理解。

(PS：下文中【】的内容为对命令的注解)

git init 【初始化本地仓库，将当前项目目录加入 git 管理】

git add <filename||path> 【将新文件加入版本控制，Git 会对目标文件进行跟踪，纳入版本控制管理。(这是个多功能命令，根据目标文件的状态不同，此命令的效果也不同：可以用它开始跟踪新文件，或者把已跟踪的文件放到暂存区，还能用于合并时把有冲突的文件标记为已解决状态等)】

git add . 【将当前目录所有文件加入到版本控制】

git commit -m 'commit comment' 【提交变动，将修改的文件转移到暂存区】

git commit -a 'commit comment' 【将 add 和 commit 操作合并】

git commit --amend 【重新 commit，将之前 commit 合并为一个(add 上次 commit 漏掉的文件或者重写 comment)】

Example: git commit -m 'commit comment'
git add filename
git commit –amend

此例子只会产生一次 commit log，第二个 commit 会覆盖第一个 commit

```
zhouliwei1@BJXX-ZHOULIWEI MINGW64 /d/colababy
$ git init
Initialized empty Git repository in D:/colababy/.git/

zhouliwei1@BJXX-ZHOULIWEI MINGW64 /d/colababy (master)
$ git add mycolababy.txt

zhouliwei1@BJXX-ZHOULIWEI MINGW64 /d/colababy (master)
$ git commit -m 'the first file'
[master (root-commit) edba41e] the first file
 1 file changed, 3 insertions(+)
 create mode 100644 mycolababy.txt
```

取消已暂存到暂存域的文件或者修改未提交的文件，可以通过 git status 命令查看取消到命令方案。

<--clone start-->

如果我们想加入一个现有项目的协同开发，可以通过 clone 命令将远程 reposity 的项目克隆镜像到本地，此镜像（此处可以联想下操作系统的镜像）包含项目所有历史变更，所有历史版本，所有分支信息等等，是远程 reposity 的一个完整副本。Git 支持多种数据传输协议，本地传输、git 协议、ssh 协议、http 协议。

git clone source. zhouliwei.com/app/zhouliwei.git 【本地仓库 clone】
git clone git+ssh://gitusername@192.168.0.1/zhouliwei.git 【远程仓库 clone(ssh 协议)】
git clone http://gitusername@source.zhouliwei.com/app/ zhouliwei.git 【远程仓库 clone(http 协议)】
git clone -l /home/zhouliwei/test 【拷贝本地资源库到当前目录】
git clone -b 分支名 http://gitusername@source. zhouliwei.com/app/test.git 【clone 指定分支(类似 checkout)】
git clone -s 远程地址 【作为共享仓库】

<--clone end-->

git status 【查看当前版本状态。
该命令有几个信息块：
on branch branchname：本地资源库在 branchname 分支
changes not staged for commit：本地资源库做了哪些修改，还未 commit 到暂存域
new file：还没有加入版本控制的新文件
modified：有改动的文件
deleted：被执行删除的文件 git rm filename
unmerged：出现冲突的文件】

在协同开发的过程中，可能会对资源进行多次修改，多次提交，在一些场景下对提交历史

的回顾极为重要，我们可以借助 log 命令完成此工作。

git log 【显示所有历史提交日志，最近的在第一行】

git log -1 【显示最近一行】

git log --stat 【显示提交日志及相关变动文件，增改行统计】

git log -p -1 【详细显示每次提交的内容差异】

git log -p -m

```
$ git log
commit edba41e4636e1370b848070e462d2c3916c977a5
Author: zhouliwei <zhouliwei555@163.com>
Date:   Sun May 1 18:05:52 2016 +0800

    the first file

zhouliwei1@BJXX-ZHOULIWEI MINGW64 /d/colababy (master)
$ git log -1
commit edba41e4636e1370b848070e462d2c3916c977a5
Author: zhouliwei <zhouliwei555@163.com>
Date:   Sun May 1 18:05:52 2016 +0800

    the first file
```

```
$ git log --stat
commit edba41e4636e1370b848070e462d2c3916c977a5
Author: zhouliwei <zhouliwei555@163.com>
Date:   Sun May 1 18:05:52 2016 +0800

    the first file

 mycolababy.txt | 3 +++
 1 file changed, 3 insertions(+)
```

```
$ git log -p -m
commit edba41e4636e1370b848070e462d2c3916c977a5
Author: zhouliwei <zhouliwei555@163.com>
Date:   Sun May 1 18:05:52 2016 +0800

    the first file

diff --git a/mycolababy.txt b/mycolababy.txt
new file mode 100644
index 0000000..617a9a1
--- /dev/null
+++ b/mycolababy.txt
@@ -0,0 +1,3 @@
+
+
+<B4><F3><BC> ä<AC><CE><D2><CA><C7>colababy.
\ No newline at end of file
```

git clean --df 【是从工作目录中移除没有 track 的文件】

git rm –cached <filename||path> 【将文件或者路径从远程 reposity、本地暂存域中删除，在本地工作空间中保留，主要针对和项目本身无关的不小心提交到服务器的文件】

vim filename 【查看、编辑资源文件】

接下来我们了解下 Git branch，分支可以说是一个非常具有魅力的创造，他将协作的成员工作独立起来，互不影响，各自沿着自己的主线向前推进，他们以 master 分支作为共同的资源集散地，所有分支生成于 master，最终又回归到 master。图 2 为 Git 的分支模型

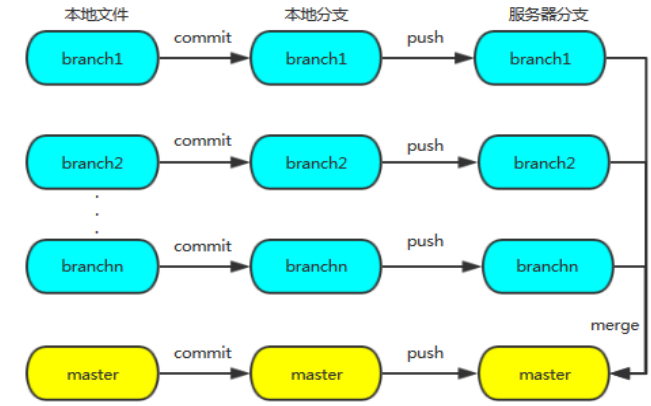

图2 Git分支模型

<--branch start-->

每次 commit 都会在暂存域中生成一个快照对象，生成一个新的版本，分支就是指向快照对象的可变指针。可以通过 HEAD 定位到当前在哪个分支工作，HEAD 是一个指向正在工作本地分支的特殊指针，可以通过 checkout 将 HEAD 切换成目标分支。HEAD 会随着当前分支的 commit 而移动，其他分支不受影响。

git branch 【列出本地所有分支(已检出)】
git branch -a 【列出本地+远程所有分支】
git branch -v 【可以看见每一个分支的最后一次提交】
git branch -av
git branch -r 【列出所有原创分支(origin/.)】
git branch branchname 【创建一个新分支】
git branch -d 分支名 【删除一个分支】
git branch -m oldbranch newbranch 【本地分支改名】
git branch --contains 字符串 【显示包含目标字符串的分支】
git branch --merged 【显示所有已合并到当前分支的分支】
git branch --no-merged 【显示所有未合并到当前分支的分支】
git branch --set-upstream 分支名 origin/分支名 【本地分支关联到远程路径】

<--branch end-->

<--checkout start-->

从远程 reposity checkout 的下来的本地分支称为跟踪分支，跟踪分支是一个和某个远程分支映射的本地分支。clone 之后本地会自动创建一个跟踪分支 master，映射到远程的分支 origin/master。

git checkout branchname 【切换到新分支】
git checkout -b branchname 【创建并切换到新的分支，如果本地已经有此分支则使用上个命令】
git checkout -b branchname origin/branchname 【在本地创建新分支，从远程拉取新分支代码】
git checkout filename 【替换本地改动，会从服务器下载最新的文件（HEAD 中最新的内容）覆盖工作目录中的文件(add、commit 的文件不受影响)，次这个操作是不可逆】

<--checkout end-->

<--merge start-->

在协同项目工作的过程中，如果多个人同时修改一个文件的相同地方，leader 在 master 上进行合并时难免会出现代码冲突的情况，此时的 merge 会合并失败，需要将冲突进行处理，我们可以采取下面方式进行处理。

git merge branchname || origin/branchname 【合并目标分支到当前分支，合并之后会生成一个新的快照对象】

如果出现冲突,通过 git status 查看冲突位置(标记为 unmerged 为重读文件)。我们可以通过手动修改成想要的代码，解决冲突的时候可以用到 git diff <source_branch> <target_branch>,处理完之后用 git add <filename||path>将文件标记为冲突已经被 resolved(即在暂存域生成一个快照对象).

也可以用 git mergetool 来处理(需要进行特定的配置)，会调用一个可视化的合并工具引导冲突处理，处理起来相对直观些。

git reset --hard HEAD 【将当前版本重置为 HEAD（通常用于 merge 失败回退）】

丢弃所有的本地改动与提交：

git fetch origin 【1.从服务器拉取最新版本】

git reset --hard origin/master 【2.将你本地主分支指向到远程分支】

<--merge end-->

<--fetch start-->

git fetch --all【 从远处资源库拉取所有分支(merge 之后才会更新本地分支)，可以进行 diff、log

git fetch origin 【将从远程拉取上次克隆后的 master 分支所有变化，即获取 master 分支最新代码】

通过 fetch 命令合并代码过程：

git fetch origin branchname1 【1. <远程主机名> <分支名> 设置当前的 fetch_head 为分支 branchname(fetch_head 为每个分支在服务器上的最新状态)】

git fetch origin branchname1: branchname2 【2. 拉取远程 branchname1 到本地新分支 branchname2(branchname2 是一个临时分支) 】

git fetch diff branchname2 【3. 将当前分支和新建的临时分支 branchname2 进行比较】

git fetch merge branchname2 【 4. 将当前分支和新建的临时分支 branchname2 进行合并，此时 branchname1 为最新代码】

git fetch -d branchname2 【5.删除临时分支 branchname2】

git pull == git fetch + merge 【从远程拉取最新版本，合并】

git pull origin branchname1 【拉取并合并 branchname1】

使用 git fetch 操作性更好些(和 pull 对比)，我们可以进行 diff、log，再 merge，更利于

开发者根据当前情况进行针对性操作。

<--fetch end-->

<--push start-->

通过 push 命令将自己的分支资源和协同小组的其他人员进行共享，前提条件是 Git 账户必须拥有远程 reposity 的写权限。
git pull <远程主机名> <远程分支名>:<本地分支名>
git push <远程主机名> <本地分支名>:<远程分支名> 【将本地分支推送到远程分支】
1) git push origin <本地分支名> 【远程分支名为空，将本地分支推送到远程与其有对映关系的分支】
2) git push origin ：<远程分支名> 【本地分支名为空，将本地空分支推送到远程分支，即删除远程分支】
3) git push origin 【将本地当前分支推送到远程与其有对映关系的分支】

<--push end-->

<--remote start-->

参与项目的协作开发，本地资源来源于远程仓库，所以需要对远程仓库的管理，比如远程仓库的创建、查看、删除、client-server 资源映射等等。
git remote 【列出远程所有 alias 别名，自己权限范围内的远程 reposity】
git remote -v 【可以看见每一个别名对应的实际 url】
git remote add [alias] [url] 【给远程 url 添加别名||把 url 添加为远程仓库】
git remote add myRepo /home/zhouliwei/test.git 【添加本地仓库作为远程仓库，共享目录】
git remote rm [alias] 【删除一个别名】
git remote rename [old-alias] [new-alias] 【重命名】
git remote set-url [alias] [url] 【更新 url. 可以加上—push 和 fetch 参数,为同一个别名 set 不同的存取地址. 】
git remote add origin <server> 【将本地仓库连接到远程仓库 git remote add origin http://gitusername@source.zhouliwei.com/app/test.git 然后可以通过 git push origin branchname 将 branchname 推送到相应远程分支;创建远程仓库】
git remote show origin 【显示远程信息】

<--remote end-->

将本地工作空间上传到远程新建仓库操作：
首先在本地空间生成用于 ssh 加密传输的公钥和私钥，将公钥维护到远程仓库的 SSH key(后面会详细介绍如何操作)
git init
git add .
git commit –m ‘initial commit’
git remote add origin http://gitusername@source.zhouliwei.com/app/test.git

git push orgin master

<--rebase start-->

可以通过 rebase 命令(衍合)以补丁的方式将某个分支的改动在其他分支上再打一遍(合并到其他分支)，可以简化分支的历史操作记录，流程看起来更清晰(和 merge 对比)。
git checkout branchname
git rebase master
将 branchname 分支代码的改动衍合到 master，相当于是在 master 上复制了 branchname 分支的改动，只在 master 分支上生成操作历史。

<--rebase end-->

至此我们已经完成了常用 Git 命令的讲解，包括本地仓库的创建初始化、克隆远程资源，本地仓库资源的修改、提交、推送，分支的管理，远程仓库资源的检出，冲突处理，远程仓库管理、协同开发等等。每个命令模块讲解过程中的注解已经非常详细了，并且为了便于更好的理解列举了相应的示例，对于使用 Git 进行项目协同开发的人员来说，上面的内容可以说是绰绰有余了。

接下来，我们顺着 Git 的梦幻之旅继续往下走。项目协作的各个成员是如何与 Git 服务端进行数据通信的呢？之前我们有提到 Git 支持四种数据传输协议，那么我们来深入了解下这四种传输协议各自的优势和不足。

传输协议	优势	缺点
本地传输	1、远程仓库部署在本地目录，Git client-server 之间的数据通信类似本地文件的复制剪切，数据的通信速度较快； 2、资源的权限沿用本地操作系统的文件权限和网络访问权限，不需要单独配置	1、由于远程仓库在本地目录，资源毁灭性丢失的危险性增大
ssh 协议(安全外壳传输协议 ssh://)	1、服务搭建相对较简单 2、基于公钥私钥对的方式进行加密授权数据传输， 3、同时支持数据的读和写操作	1、不支持匿名访问，必须通过 ssh 访问主机才能读写仓库
Git 协议(git://)	1、自身携带的传输协议，传输速度最快的协议 2、使用类似 ssh 相同的数据传输机制，但取消了加密解密的开销	1、没有授权机制，要么所有客户端都可读，要么所有客户端都可写，不能根据情况选择性配置读写权限 2、服务搭建相对较复杂

http/https 协议(超文本传输协议)	1、服务搭建相对较简单，基于 Apache 等 web 容器就可以实现 2、授权机制简单，能够访问 Git 仓库所在服务器的 web 服务的人都可以获取远程仓库资源	1、数据通信网络开销较大 2、执行写操作需要基于 ssh 协议

通过上面的对比分析，我们发现 http/https 是最简单最流行的一种协议方式，ssh 是最安全的一种协议方式，特别是在互联网领域这一点尤为重要，并且 http/https 的写操作也是基于 ssh 协议完成的，那么我们继续深入的了解下 ssh 通信协议。

ssh 数据通信协议也称作安全外壳协议，从他的名字就可以看出他使命就是确保安全数据传输，并且传输的数据会进行压缩，降低网络传输消耗，提高数据传输速度。ssh 协议是基于公钥私钥对的方式进行加密授权数据传输的，下面我们通过两个加密算法来理解公钥私钥——对称加密算法和非对称加密算法。

图 3 为对称加密算法流程图，数据的发送方 sender 和接收方 receiver 通过相同的密钥 key 对数据进行加密解密操作。秘钥用于确保数据在公共通道传输过程中的安全性，即使密文数据在传输过程中被外部窃取，如果没有密钥也不能获取其中的内容。

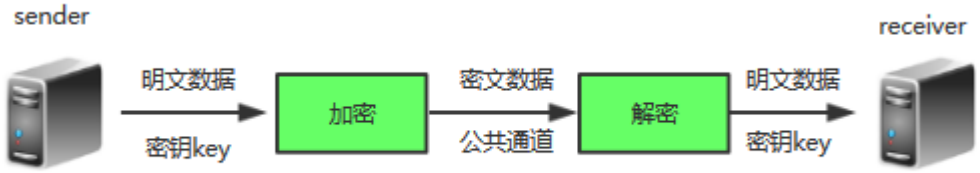

图3 对称加密算法

图 4 为非对称加密算法流程图，数据的发送方 sender 和接收方 receiver 通过不同的密钥 key 对数据进行加密解密操作。sender 通过公钥 key1 对明文数据进行加密，receiver 通过私钥 key2 对密文数据进行解密。公钥和私钥一定是成对出现的，如果一个文件用公钥进行加密，则可以通过私钥进行解密；如果一个文件用私钥进行加密，则可以通过公钥进行解密。比如，我们在互联网环境中和其他合作方进行数据通信，我们的私钥是保密的，只有自己知道，公钥可以分发给合作方 1，合作方 2 等等，这些合作方可以通过公钥对数据进行加密传送给我们，然后我们通过自己的私钥进行解密，并且在这个过程中合作方之间是独立的数据安全的，不会看到其他其他合作方的数据，这也是非对称加密的一个优势。

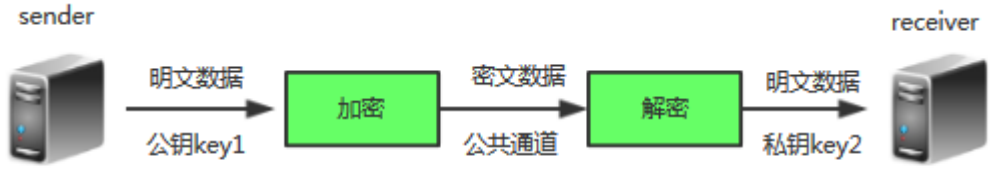

图4 非对称加密算法

接着，我们来看下如果生成属于自己的公钥和私钥。

在 Git 窗口输入 ssh-keygen –t rsa –C “zhouliwei555@163.com”命令,回车，如图 5。

```
$ ssh-keygen -t rsa -C "zhouliwei555@163.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/zhouliwei1/.ssh/id_rsa):
Created directory '/c/Users/zhouliwei1/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/zhouliwei1/.ssh/id_rsa.
Your public key has been saved in /c/Users/zhouliwei1/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:nBfHMP2NOHzvgj17FPsX9LXUKRE3AUGJkz+U1S2JCbI zhouliwei555@163.com
The key's randomart image is:
+---[RSA 2048]----+
|          . +o+BB*+|
|           o B+=+ +|
|          E ..*o.+o|
|         . . o+++=+|
|          S .   o=.*|
|            .     ++|
|                o oo|
|               . + =|
|                 .=.|
+----[SHA256]-----+
```

图 5 生成公钥私钥

在提示信息的目录中，我们看到生成两个文件,id_rsa.pub 为公钥文件，id_rsa 为私钥文件，如图 6。

（在 Mac 中，切换到.ssh 目录(cd .ssh)，执行 ssh-keygen –t rsa –C “zhouliwei555@163.com”，生成私钥和公钥）

图 6 公钥私钥文件

生成的公钥和私钥怎么使用呢？现在我们使用 SourceTree 基于 ssh 协议从 github 上 clone 一个项目，会发现图 7 ssh 认证失败提示。

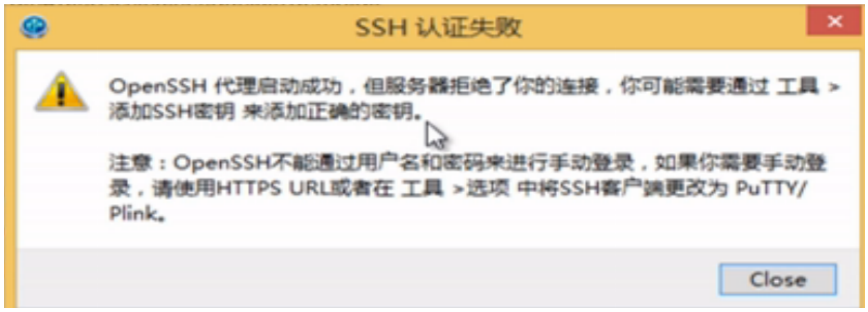

图 7 ssh 认证失败

为什么会 ssh 认证失败呢？提示信息描述的很清楚，我们需要将自己生成的公钥加入到 github 维护的该项目中(这个操作由该项目管理员完成，一个项目可能会被加入多个公钥)，加入之后配合本地的私钥就可以进行安全的数据通信了，此时客户端就拥有了该项目的写权限，然后重新尝试 clone，克隆项目成功。

回过头来，思考下 Git 基于 http/https 通信协议的写权限是不是也是通过这种方式实现的呢？答案是肯定的。

现在到了可以庆祝的时刻了，  你不但可以熟练使用 Git 命令进行协同工作，还

透彻的了解了 Git 数据通信的内部原理。知其然并知其所以然，将知识运用到实践中，才是研究技术的最高境界。（送人玫瑰，手留余香）

心灵鸡汤

如果你爱他，就送他去创业公司，因为那里是天堂；如果你恨他，就送他去创业公司，因为那里是地狱。