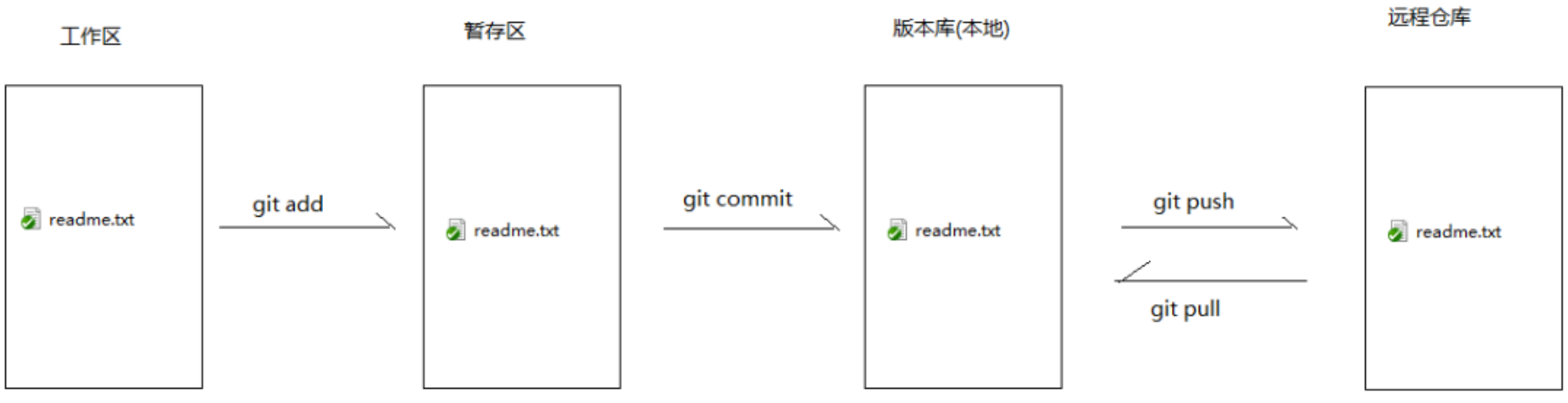
Git笔记

更多命令 ： [Git常用命令]

仓库结构详解在<*12.Git*仓库详解>



1.初始化设置

每台机器必须自报家门

```
// 用户名 和 邮箱
git config --global user.name "Luo"
git config --global user.email "1210225958@qq.com"
```

2.初始化仓库

```
// 初始化仓库
git init
```

执行后 出现隐藏文件夹 .git

3.add操作

把文件添加到暂存区

```
// '.' 表示当前路径的全部文件
git add .

// 指定某个文件 包括后缀
git add Test.txt

// 指定多个文件 包括后缀
git add file1.txt file2.txt
```

4.commit操作

把暂存区的所有文件提交到 -> 版本库(本地)

```
git commit -m "备注信息: xxx"
```

5.push/pull操作

把版本库(本地) 的文件*push*到本地仓库

```
// 配置远程仓库地址
git remote add origin https://gitee.com/xxx

// 将本地仓库master分支代码推送到远程仓库 ->
// 使用 -u 参数指定一个默认主机，这样后面就可以不加任何参数使用 git push
git push -u origin master

// 从远程仓库拉代码 origin远程主机 master哪个分支
git pull origin master

// 修改远程仓库 url
git remote set-url --push [Name] [newUrl]
```

详解 *push* 在 *< 11.Git push* 详解>

6.查询操作

```
// 查看工作区和暂存区的文件状态状态
git status

// 查看工作区文件内容相对于暂存区文件内容的差别
git diff

// 查看日志
git log

// 查看日志
git log --pretty=oneline

// 配合时间刺客使用
git reflog

// 所有分支查看 能看出当前所在分支
git branch

// 查看远程分支
git branch -r

// 查看远程库 一般有 origin
git remote show

// 查询远程仓库地址
git remote -v

// 查询目录下的所有文件夹和文件(包括隐藏文件)
ls -a
```

7.时间刺客

```
// 回上一个版本"HEAD^"，回上上个版本就是"HEAD^^"
git reset --hard HEAD^

// 回到上上个版本,或者当前版本的前某个版本
git reset --hard HEAD~2

// 回到指定版本 <commitId> 就是 git log 查询出来的某个版本的 id
git reset --hard <commitId>

// 将某个文件恢复到某个版本时的状态 fileName 文件名
git checkout <commitId> fileName

// 查询之前记录 得到大概这样的"9e8f60b" 使用上条命令回去
git reflog
```

git reset --hard commitId 命令，将工作区的提交穿越到你指定的 commitId 里这个时候你会发现 **git log** 根本没有记录这之后的提交记录，Git提供了一个命令**git reflog** 用来记录你的每一次改变目录树的命令，使用好他就可以很方便的恢复你的提交，那什么是目录树呢?

我们的主线往往是一根直线，多一个分支相当于多一个分叉，无数分支纵横交错就像一颗树状的结构，所以我们称之为目录树。**commit rebase reset merge** 这些都是改变目录树的操作。这个恢复丢失代码方法也只有存在改变目录树的操作才恢复的回来....

git log 也可以看到所有分支的历史提交，不一样的是看不到已经被删除的commit 记录和 reset rebase merge 的操作

8.管理修改

注意*:* 每次 *commit* 前检查文件是否 有 *add*

git checkout -- filename`可以丢弃工作区的修改：**-- 后面是一个空格**

命令 `git checkout -- readme.txt` 意思就是，把 `readme.txt` 文件在工作区的修改全部撤销，这里有两种情况：
一：`readme.txt` 自修改后还没有被放到暂存区(`git add`)，现在，撤销修改就回到和版本库一模一样的状态；
二：`readme.txt` 已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。

总之，就是让这个文件回到最近一次 `git commit` 或 `git add` 时的状态。

注意:
`git checkout -- file` 命令中的 `--` 很重要，没有 `--` ，就变成了**“切换到另一个分支”**的命令，我们在后面的分支管理中会再次遇到 `git checkout` 命令

```
// 撤销整个工作区的修改 fileName 包括文件后缀名
git checkout -- fileName

// 恢复整个工作区回到暂存区或者本地仓库的状态 就近原则 不包括远程仓库
git checkout -- .

// 撤销 add 后 放入暂存区的文件 不动现工作区文件
git restore --staged FileName

// 撤销 add 后 放入暂存区的文件 工作区对应文件恢复到这个撤销文件当初 add 时的样子
git restore FileName
```

9.删除操作

rm 理解为 remove -rf 理解: -r 递归的处理文件, -f 无提示强制删除=> 合写 -rf

```
// 删除文件 fileName 包括文件后缀名
git rm fileName

// 删除工作区和暂存区文件删除后
// 注意: 暂存区会有未提交的commit操作, 不提交则不会记录 版本库(本地)
git rm -f fileName

// 删除暂存区文件 不删除工作区文件
git rm --cached fileName

// 删除分支
git branch -d Name

//删除远程分支
git push origin --delete <BranchName>

// 删除关联的远程仓库链接
 git remote rm origin

// 删除目录或者文件(删除.git目录)
rm -rf .git

// 查看远程仓库所有文件 -n 文件列表预览 不会删除文件
git rm -r -n --cached .

// 删除远程仓库所有文件 相当于清空本地版本库再push到远程
git rm -r --cached .
再接着
git commit -m "清空仓库"
git push origin master
```

这个时候，`Git`知道你删除了文件，因此，工作区和版本库就不一致了，`git status`命令会立刻告诉你哪些文件被删除了：

删除完成后需要`commit`

如果删除了想恢复,可以使用`reset`版本恢复

10.分支管理

***Name* 表示 分支名称**

```
// 所有分支查看 能看出当前所在分支
git branch

// 创建分支
git branch Name

//创建远程分支(本地分支push到远程)
git push origin Name

// 创建并切换创建的分支
git branch checkout -b Name

// 重命名分区
 git branch -m Name

// 删除分支
git branch -d Name

//删除远程分支
git push origin --delete <BranchName>

// 切换分支
```

```
git checkout Name

// 合并分支  执行后->  "Already up to date." 意思是已经更新 有注意事项 >>>>>
git merge Name

合并分区注意事项:  idea不受影响 只限于 Git 操作

**当前分支合并其它分支后, 其它分支不曾改变,当前分支就算改变文件后再
    怎么合并哪个其它分区都没有任何效果,只是控制台提示数据已更新**

**分支文件旧的合并新的   ->      被新的覆盖       然后需要 commit  才能存入本地版本库**

**分支文件新的合并旧的   ->      新旧混一起       需要修改合并后重新 add 再 commit 存入 本地版本库**
```

11.Git push 详解

git push的一般形式为 git push <远程主机名> <本地分支名> <远程分支名>，例如 git push origin master：refs/for/master，即是将本地的master分支推送到远程主机origin上的对应master分支， origin 是远程主机名，

第一个master是本地分支名，第二个master是远程分支名。

1.1 git push origin master

如果远程分支被省略，如上则表示将本地分支推送到与之存在追踪关系的远程分支（通常两者同名），如果该远程分支不存在，则会被新建

1.2 git push origin： refs/for/master

如果省略本地分支名，则表示删除指定的远程分支，因为这等同于推送一个空的本地分支到远程分支，等同于 git push origin --delete master

1.3 git push origin

如果当前分支与远程分支存在追踪关系，则本地分支和远程分支都可以省略，将当前分支推送到origin主机的对应分支

1.4 git push

如果当前分支只有一个远程分支，那么主机名都可以省略，形如 git push，可以使用git branch -r ，查看远程的分支名

1.5 git push 的其他命令

这几个常见的用法已足以满足我们日常开发的使用了，还有几个扩展的用法，如下：

（1） **git push -u origin master** 如果当前分支与多个主机存在追踪关系，则可以使用 -u 参数指定一个默认主机，这样后面就可以不加任何参数使用git push，

不带任何参数的git push，默认只推送当前分支，这叫做simple方式，还有一种matching方式，会推送所有有对应的远程分支的本地分支， Git 2.0之前默认使用matching，现在改为simple方式

如果想更改设置，可以使用git config命令。git config --global push.default matching OR git config --global push.default simple；可以使用git config -l 查看配置

（2） git push --all origin 当遇到这种情况就是不管是否存在对应的远程分支，将本地的所有分支都推送到远程主机，这时需要 -all 选项

（3） git push --force origin git push的时候需要本地先git pull更新到跟服务器版本一致，如果本地版本库比远程服务器上的低，那么一般会提示你git pull更新，如果一定要提交，那么可以使用这个命令。

（4） git push origin --tags //git push 的时候不会推送分支，如果一定要推送标签的话那么可以使用这个命令

1.6 关于 refs/for

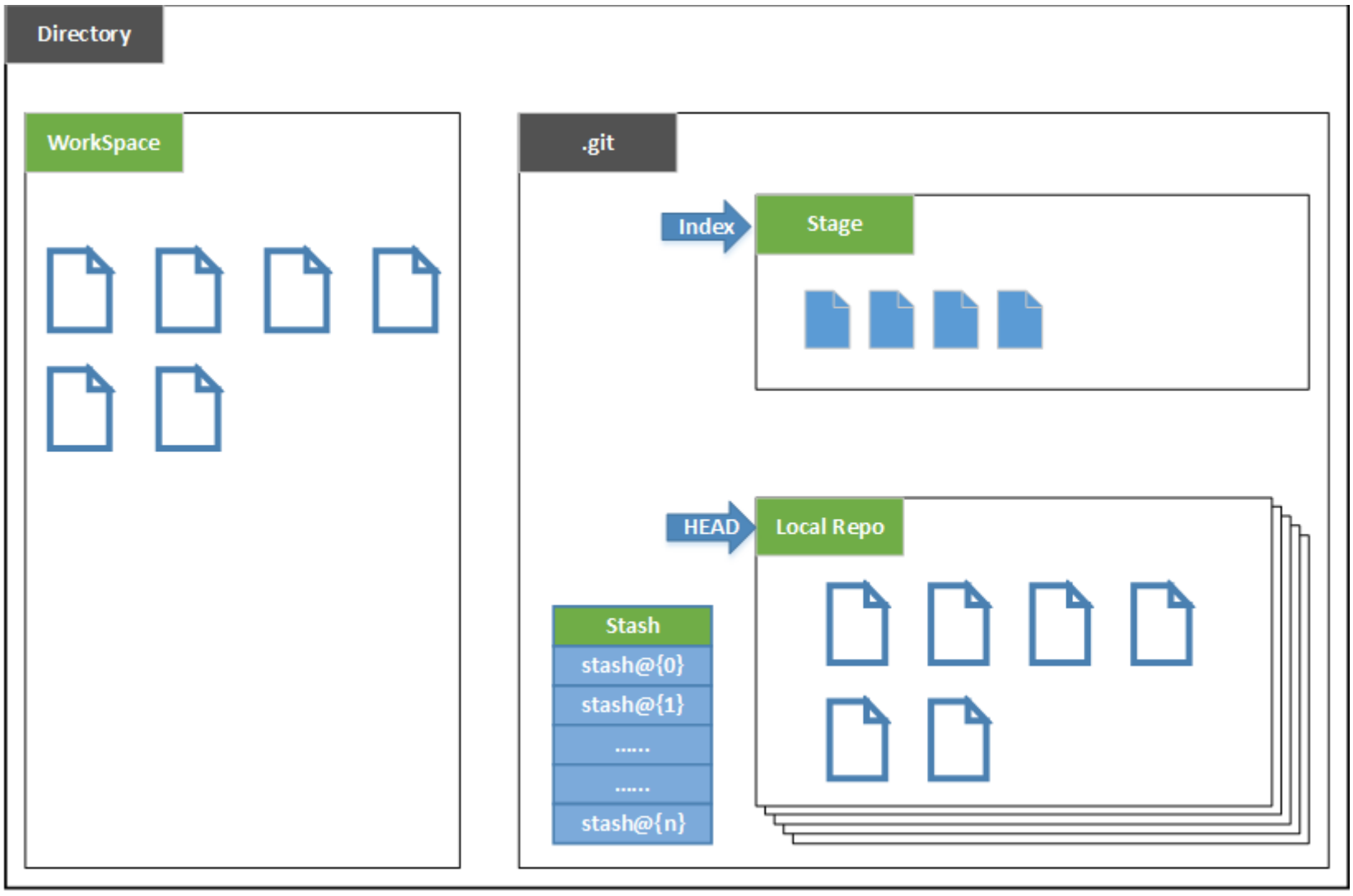
refs/for 的意义在于我们提交代码到服务器之后是需要经过code review 之后才能进行merge的**(意思是需要被他人审核通过的代码才能合并)**，而refs/heads 不需要

12.Git仓库 详解

在Git中，我们将需要进行版本控制的文件目录叫做一个**仓库（repository）**，每个仓库可以简单理解成一个目录，这个目录里面的所有文件都通过Git来实现版本管理，Git都能跟踪并记录在该目录中发生的所有更新。

现在我们已经知道什么是repository（缩写repo）了，假如我们现在建立一个仓库（repo），那么在建立仓库的这个目录中有一个“.git”的文件夹。这个文件夹非常重要，所有的版本信息，更新记录，以及Git进行仓库管理的相关信息

全部保存在这个文件夹里面。所以，不要修改/删除其中的文件，以免造成数据的丢失。



根据上面的图片，下面给出了每个部分的简要说明：

- **Directory**：使用Git管理的一个目录，也就是一个仓库，包含我们的工作空间和Git的管理空间。
- **WorkSpace**：需要通过Git进行版本控制的目录和文件，这些目录和文件组成了工作空间，除了.git之外的都属于工作区。
- **.git**：存放Git管理信息的目录，初始化仓库的时候自动创建。
- **Index/Stage**：暂存区，或者叫待提交更新区，在提交进入repo之前，我们可以把所有的更新放在暂存区。
- **Local Repo**：本地仓库，一个存放在本地的版本库；HEAD会只是当前的开发分支（branch）。
- **Stash**：是一个工作状态保存栈，用于保存/恢复WorkSpace中的临时状态。

  有了上面概念的了解，下面简单介绍仓库的文件结构。

名称	修改日期	类型	大小
hooks	2021/9/5 18:44	文件夹	
info	2021/9/5 18:44	文件夹	
logs	2021/9/5 18:48	文件夹	
objects	2021/9/5 18:52	文件夹	
refs	2021/9/5 18:49	文件夹	
COMMIT_EDITMSG	2021/9/5 18:52	文件	1 KB
config	2021/9/5 18:49	文件	1 KB
description	2021/9/5 18:44	文件	1 KB
HEAD	2021/9/5 18:44	文件	1 KB
index	2021/9/5 18:52	文件	1 KB

新版本的 *Git* 不再使用 ***branches* 目录**，*description* 文件仅供 *GitWeb* 程序使用，所以不用关心这些内容。config 文件包含了项目特有的配置选项，info 目录保存了一份不希望在 .gitignore 文件中管理的忽略模式 (ignored patterns) 的全局可执行文件。hooks 目录保存了客户端或服务端钩子脚本。

另外还有四个重要的文件或目录：HEAD 及 index 文件，objects 及 refs 目录。这些是 Git 的核心部分。

- **objects** 目录存储所有数据内容
- **refs** 目录存储指向数据 (分支) 的提交对象的指针,里面即有stash栈指针以及tag等
- **HEAD** 文件指向当前分支
- **index** 文件保存了暂存区域信息