**14-git工具**

1. **git操作部分**

1.1 **普通操作**

1.1.1 **git用户建立**

用户配置

|  |
| --- |
| C++ git config --global user.name "用户名"  git config --global user.email "邮箱" 第一次配置 |

查看配置

|  |
| --- |
| C++ git config --global --list |

生成密钥

|  |
| --- |
| C++ ssh-keygen -t rsa -C '邮箱' 并按回车3下  windows就是C:\Users\Administrator)  linux 用户目录下.ssh/id\_rsa.pub id\_rsas |

1.1.2 **克隆仓库**

|  |
| --- |
| Bash git clone https://github.com/apache/brpc.git // 浅克隆，只克隆最近一次提交 git clone --depth 1 https://github.com/apache/brpc.git // 后续多次克隆 cd brpc git fetch --deepen=50 |

先生成链接，后续一层层克隆

|  |
| --- |
| Bash mkdir brpc && cd brpc git init git remote add origin https://github.com/apache/brpc.git git fetch --depth 1 # 或根据你的需要增加更多深度 git checkout -b master origin/master |

1.1.3 **分支切换**

|  |
| --- |
| C++ git branch -a 看分支  git checkout origin xxx 切分支 |

1.1.4 **协议更改**

更改https协议为ssh

|  |
| --- |
| C++  git remote set-url origin 地址 |

查看协议

1. 查看当前的远程 URL：运行以下命令来查看当前配置的远程仓库 URL：

|  |
| --- |
| Bash git remote -v |

通常看到的结果类似这样：

|  |
| --- |
| Plaintext origin https://github.com/username/repository.git (fetch) origin https://github.com/username/repository.git (push) |

1. 更改远程 URL 到 SSH：使用以下 git remote set-url 命令来更改远程 URL：

|  |
| --- |
| Bash git remote set-url origin git@github.com:username/repository.git |

替换 git@github.com:username/repository.git 为你的实际 SSH URL。

1. 验证更新是否成功：再次运行 git remote -v 确认远程 URL 是否更新成功。你应该会看到如下输出：

|  |
| --- |
| Plaintext origin git@github.com:username/repository.git (fetch) origin git@github.com:username/repository.git (push) |

1.1.5 **git提交**

|  |
| --- |
| C++ git add xxx(本地文件)  git commit -m '本次提交注释'  git push origin xxx（远程分支） |

1.1.6 **删除远程**

一定要带cached 不然会删除本地

|  |
| --- |
| C++ git rm -r --cached xxx |

1.1.7 **git 撤回**

撤销 commit -m

|  |
| --- |
| C++ git reset --soft HEAD^  git reset --soft HEAD~1 最近一次 |

撤销 git add.

|  |
| --- |
| C++ git reset HEAD  git reset HEAD <file> 对单独文件撤回git add |

撤销 commit，撤销git add

|  |
| --- |
| C++ git reset --mixed HEAD^ |

–mixed

意思是：不删除工作空间改动代码，撤销commit，并且撤销git add . 操作

这个为默认参数,git reset --mixed HEAD^ 和 git resest HEAD^ 效果是一样的。

–soft

不删除工作空间改动代码，撤销commit，不撤销git add .

–hard

删除工作空间改动代码，撤销commit，撤销git add .

拉取分支命令 git clone xxx(ssh或者https地址)

撤销 commit，撤销git add，撤回修改区

|  |
| --- |
| C++  git reset --hard HEAD~1 |

这个命令会彻底地撤销最近的一次提交，并且将工作目录和暂存区都恢复到指定提交的状态。

1.1.7.1 **逆转（保留历史记录）**

git revert（推荐用于公共分支）

新建一个分支，这个分支撤回原分支上次、或多次提交

如果你的提交已经推送或者你正在处理公共分支上，使用git revert创建一个新的提交来“逆转”之前的提交更为安全，这不会改变项目历史。

|  |
| --- |
| C++ git revert HEAD |

这将创建一个新提交，内容是上一个提交的逆。这样做的好处是，不会更改项目的历史记录，因此适用于团队环境以及已经推送到远程仓库的提交。

1.1.7.2 **撤销（不保留历史记录）**

git commit --amend（如果你刚刚误提交了）

取代前一个提交

如果你刚刚进行了一次提交，然后意识到需要对那次提交做出一点小更改，你可以使用git commit --amend来修改那次提交而非创建一个新提交。

|  |
| --- |
| C++ git add <file> 先将需要修改的文件添加到暂存区 git commit --amend 修改刚才的提交 |

这将会打开文本编辑器，让你编辑提交信息，或者你也可以直接在命令中添加新的提交信息，如：

git commit --amend -m "New commit message"

1.1.8 **分支合并**

1、避免覆盖本地，直接去dev merge自己分支，在dev合并冲突

2、在自己分支开发完，pull最新的dev分支相当于（merge dev），自己分支解决冲突。然后再去dev merge自己分支。

合并步骤：

1、进入要合并的分支（如开发分支合并到master，则进入master目录）

|  |
| --- |
| C++ git pull |

2、查看所有分支是否都pull下来了

|  |
| --- |
| C++ git branch -a |

3、使用merge合并开发分支

|  |
| --- |
| C++ git merge 分支名 |

4、查看合并之后的状态

|  |
| --- |
| C++ git status |

5、有冲突的话，通过IDE解决冲突（也就是文本编辑器手动修改成需要提交的）；

6、解决冲突之后，将冲突文件提交暂存区

|  |
| --- |
| C++ git add 冲突文件 |

7、提交merge之后的结果

|  |
| --- |
| C++ git commit |

如果不是使用git commit -m "备注" ，那么git会自动将合并的结果作为备注，提交本地仓库；

8、本地仓库代码提交远程仓库

|  |
| --- |
| C++ git push |

git将分支合并到分支，将master合并到分支的操作步骤是一样的；

可以使用idea git相关组件来解决冲突

合并例子

|  |
| --- |
| Plain Text // 切换到 master 分支 git checkout master git pull origin master // 将第一个补丁分支合并到 master git merge origin/master-patch-a7bf // 将第二个补丁分支合并到 master git merge origin/master-patch-a7bf-patch-10ab // 将更改推送到远程 master 分支 git push origin master |

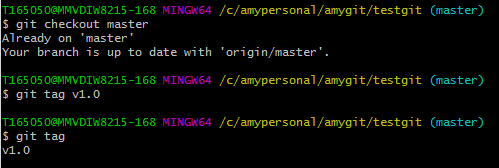
1.1.9 **git标签**

1、查看标签

|  |
| --- |
| C++ git tag |

2、默认上次commit绑在一起的标签

|  |
| --- |
| C++ git tag v1.0 |

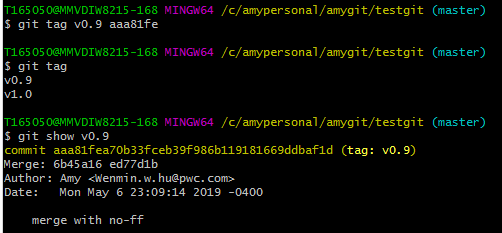


3、给往期版本打标签

git log找到历史提交的commit id，然后就可以打上了。

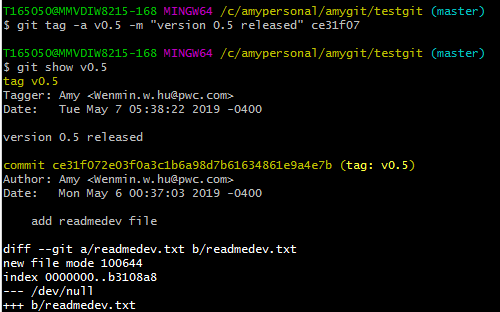
4、查看特定标签

|  |
| --- |
| C++ git show 标签名 |



5、还可以创建带有说明的标签，用-a指定标签名，-m指定说明文字：

|  |
| --- |
| C++  git tag -a v0.5 -m "version 0.5" |



注意：标签总是和某个commit挂钩。如果这个commit既出现在master分支，又出现在dev分支，那么在这两个分支上都可以看到这个标签。

6、删除标签

如果标签打错了，也可以删除：

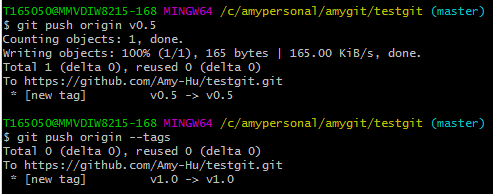
|  |
| --- |
| C++ git tag -d 标签名 |



7、标签推送到远程

|  |
| --- |
| C++ git push origin 标签名称 |

或者，一次性推送全部尚未推送到远程的本地标签：



如果标签已经推送到远程，要删除远程标签就麻烦一点，先从本地删除，然后，从远程删除。删除命令也是push，格式如下：



1.1.10 **解决冲突**

1、冲突，别人部分提交过

先add

再commit

最后pull 云端，再push，云端会merge

2、云端和本地历史记录不一样

(1)合并（Merge）: 如果你希望保留分叉的历史，可以执行以下命令：

bashCopy code

git pull origin main --no-rebase

或者你可以配置默认行为为合并：

bashCopy code

git config pull.rebase false

(2)变基（Rebase）: 如果你想让你的本地提交在远程提交之后，可以执行：

bashCopy code

git pull origin main --rebase

或者你可以配置默认行为为变基：

bashCopy code

git config pull.rebase true

(3)仅快进（Fast-forward only）: 如果你只想执行快进合并（fast-forward），可以执行：

bashCopy code

git pull origin main --ff-only

或者你可以配置默认行为为仅快进：

bashCopy code

git config pull.ff only

(4)以本地为准

git push origin main --force

git push origin main -f

以云端为准

git fetch --all

git reset --hard origin/main

1.1.11 **git fetch操作**

git fetch 是 Git 版本控制系统中的一个命令，它的作用是从远程仓库下载本地仓库中缺少的数据。这个命令会获取远程仓库中当前分支的更新（如新的或更新的分支、标签等），但它不会改变你的工作目录中的文件，也就是说，它不会自动合并或修改你当前的工作进度。为了整合这些新的更新，你需要使用 git merge 或 git rebase 命令。

以下是 git fetch 一些常用的用法：

|  |
| --- |
| C++ git fetch origin ：来更新远程仓库 origin 的所有分支信息。origin 是远程仓库默认名称的一个常见占位符。 git fetch origin master： 可以获取远程的 master 分支，但不会影响本地的 master 分支。 git fetch --all ：获取所有远程的更新（包括所有的远程仓库和分支） 。 |

git fetch 通常与 git pull 命令相比较。但不同于 git fetch，git pull 会自动获取远程分支的数据然后立即合并到当前分支。

git fetch 的主要优点是允许用户查看更新内容，然后在合并到本地分支之前进行手动检查。

在多人合作的项目中，定期运行 git fetch 是一个好习惯，这样可以保持本地仓库与远程仓库的同步，从而避免在提交自己的更改时发生冲突。在合并前，你还可以使用 git log 查看更新的提交记录，使用 git diff 等命令检查更新的内容。

1.2 **子模块**

1.2.1 **子模块和子树区别**

4.1 Git Submodule（子模块）:

* 子模块实际上是一个独立的Git仓库，它被嵌入到你的主仓库中作为引用。
* 子模块在主仓库中以特定的提交指针存在。当你克隆一个包含子模块的仓库时，子模块目录会存在，但是目录会空缺，除非你特别初始化和更新子模块。
* 子模块适用于更多的项目版本控制，允许你切换到子模块的特定提交。
* 当主仓库拉取更新时，子模块仍然保持在原来的提交上，除非你明确地更新子模块。
* 子模块对于多项目共享代码很有用，但它们可能会使仓库的克隆、拉取和更新过程变得稍微复杂一些。

4.2 Git Subtree（子树）:

* 子树允许你将一个外部仓库作为一个目录置于你的主仓库中，它包含外部仓库的所有历史和提交。
* 子树不依赖于Git仓库的引用或链接。这意味着克隆主仓库会自动包含子树的内容和历史，不需要额外的初始化步骤。
* 子树更容易管理，因为主仓库包含了子树的所有内容，而无需考虑子模块的指针和单独的仓库。
* 更新子树简单，因为你只需要从子树仓库拉取变更并合并到你的主仓库。
* 子树比较适合于那些需要包含并可能修改子仓库代码的场景。

1.2.2 **子模块搭建**

1、在服务层工程中添加子模块：在每个服务层工程的仓库中，你可以通过以下命令将util库作为子模块添加到特定的文件夹：

|  |
| --- |
| C++ git submodule add <util库的仓库URL> <路径/到/子模块/文件夹> |

该命令会在服务层工程中创建一个包含util库的子文件夹，并且添加一个 .gitmodules 文件来跟踪子模块的信息。

克隆含有子模块的项目：当其他开发人员或者你在新环境中克隆含有子模块的服务层工程时，需要执行下面的命令以2、初始化和更新工程中的子模块：

|  |
| --- |
| C++ git clone <服务层工程的仓库URL> cd <服务层工程文件夹> git submodule update --init --recursive |

这些命令会拉取子模块的当前提交。

3、更新子模块：如果util库有更新，开发人员可以在服务层工程中通过以下命令来更新子模块到最新的提交：

|  |
| --- |
| C++ git submodule update --remote |

或者进入到具体的子模块目录中，手动拉取需要的版本。

工程中引用子模块：在服务层工程代码中，你可以通过包含子模块路径的方式来引用util库中的代码。

拉取代码后

|  |
| --- |
| C++ git submodule init 注册子模块，第一次 git submodule update 可以获取子模块代码 |

1.2.3  **子树搭建**

1、在本地克隆的父仓库中添加远程仓库引用: 这个步骤与添加一个常规的 Git 远程仓库相似。使用下面的命令：

|  |
| --- |
| C++ git remote add -f <subtree-remote-name> <subtree-repo-url> |

其中 <subtree-remote-name> 是你给这个远程仓库的本地名称， <subtree-repo-url> 是远程仓库的 URL。

2、使用 subtree add 将远程仓库作为子树添加到父项目中: 需要指定你希望将子仓库代码放置的本地子目录路径，以及要添加的子仓库分支。

|  |
| --- |
| C++ git subtree add --prefix <path/to/subtree> <subtree-remote-name> <branch> --squash |

* --prefix <path/to/subtree> 指明了你希望将子树内容放置在父仓库中的目录。
* <subtree-remote-name> 是你之前定义的远程仓库引用名。
* <branch> 表示子仓库中需要集成的分支名，通常是 main 或 master。
* --squash 参数意味着你将子树的历史压缩成一个单独的提交。

3、后续的 subtree 操作: 随着项目的进展，你可能需要从子树的远程仓库拉取更新，或是将本地对子树的修改推送回远程仓库。对于这些操作，你也可以使用 git subtree。

* 拉取子树的更新：

|  |
| --- |
| C++ git subtree pull --prefix <path/to/subtree> <subtree-remote-name> <branch> --squash |

* 推送本地更改到子树的远程仓库：

|  |
| --- |
| C++ git subtree push --prefix <path/to/subtree> <subtree-remote-name> <branch> |

1.3 **git log**

|  |  |
| --- | --- |
| git log: | 显示完整的提交历史。 |
| git log -n <limit>: | 显示最新的<limit>条提交。例如，git log -n 5显示最新的5个提交。 |
| git log --oneline: | 一行显示一个提交，只包括哈希和提交信息。 |
| git log --graph: | 显示一个ASCII图形表示的分支合并历史。 |
| git log --author="<pattern>": | 只显示指定作者的提交。 |
| git log --since="2 weeks ago": | 显示最近两周内的提交。 |
| git log --until="yesterday": | 显示直到昨天的提交。 |
| git log --grep="<pattern>": | 只显示提交信息中包含指定模式的提交。 |
| git log <path>: | 只显示涉及到特定文件或目录的提交。 |
| git log <branch1>..<branch2>: | 显示两个分支点之间的提交，不包括<branch1>而只包括<branch2>上的提交。 |
| git log --no-merges: | 显示不包含合并提交的日志。 |

1.4 **子模块和子树区别**

1.4.1 **Git Submodule（子模块）:**

* 子模块实际上是一个独立的Git仓库，它被嵌入到你的主仓库中作为引用。
* 子模块在主仓库中以特定的提交指针存在。当你克隆一个包含子模块的仓库时，子模块目录会存在，但是目录会空缺，除非你特别初始化和更新子模块。
* 子模块适用于更多的项目版本控制，允许你切换到子模块的特定提交。
* 当主仓库拉取更新时，子模块仍然保持在原来的提交上，除非你明确地更新子模块。
* 子模块对于多项目共享代码很有用，但它们可能会使仓库的克隆、拉取和更新过程变得稍微复杂一些。

1.4.2 **Git Subtree（子树）:**

* 子树允许你将一个外部仓库作为一个目录置于你的主仓库中，它包含外部仓库的所有历史和提交。
* 子树不依赖于Git仓库的引用或链接。这意味着克隆主仓库会自动包含子树的内容和历史，不需要额外的初始化步骤。
* 子树更容易管理，因为主仓库包含了子树的所有内容，而无需考虑子模块的指针和单独的仓库。
* 更新子树简单，因为你只需要从子树仓库拉取变更并合并到你的主仓库。
* 子树比较适合于那些需要包含并可能修改子仓库代码的场景。

1.4.3 **子模块搭建**

1、在服务层工程中添加子模块：在每个服务层工程的仓库中，你可以通过以下命令将util库作为子模块添加到特定的文件夹：

|  |
| --- |
| Plaintext git submodule add <util库的仓库URL> <路径/到/子模块/文件夹> |

该命令会在服务层工程中创建一个包含util库的子文件夹，并且添加一个 .gitmodules 文件来跟踪子模块的信息。

克隆含有子模块的项目：当其他开发人员或者你在新环境中克隆含有子模块的服务层工程时，需要执行下面的命令以2、初始化和更新工程中的子模块：

|  |
| --- |
| Plaintext git clone <服务层工程的仓库URL> cd <服务层工程文件夹> git submodule update --init --recursive |

这些命令会拉取子模块的当前提交。

3、更新子模块：如果util库有更新，开发人员可以在服务层工程中通过以下命令来更新子模块到最新的提交：

|  |
| --- |
| Plaintext git submodule update --remote |

或者进入到具体的子模块目录中，手动拉取需要的版本。

工程中引用子模块：在服务层工程代码中，你可以通过包含子模块路径的方式来引用util库中的代码。

拉取代码后

git submodule init 注册子模块，第一次

git submodule update 可以获取子模块代码

1.4.4 **子树搭建**

1、在本地克隆的父仓库中添加远程仓库引用: 这个步骤与添加一个常规的 Git 远程仓库相似。使用下面的命令：

|  |
| --- |
| Bash git remote add -f <subtree-remote-name> <subtree-repo-url> |

其中 <subtree-remote-name> 是你给这个远程仓库的本地名称， <subtree-repo-url> 是远程仓库的 URL。

2、使用 subtree add 将远程仓库作为子树添加到父项目中: 需要指定你希望将子仓库代码放置的本地子目录路径，以及要添加的子仓库分支。

|  |
| --- |
| Bash git subtree add --prefix <path/to/subtree> <subtree-remote-name> <branch> --squash |

3、后续的 subtree 操作: 随着项目的进展，你可能需要从子树的远程仓库拉取更新，或是将本地对子树的修改推送回远程仓库。对于这些操作，你也可以使用 git subtree。

* 拉取子树的更新：

|  |
| --- |
| Bash git subtree pull --prefix <path/to/subtree> <subtree-remote-name> <branch> --squash |

* 推送本地更改到子树的远程仓库：

|  |
| --- |
| Bash git subtree push --prefix <path/to/subtree> <subtree-remote-name> <branch> |

1.5 **github**

1、README格式

|  |
| --- |
| C++ #项目名称 一级标题   ##项目简介 二级标题 |

2. **gitlab搭建**

[docker部署GitLab-CSDN博客](https://blog.csdn.net/liyuchenii/article/details/136062204)

[Gitlab----使用Docker方式安装部署Gitlab-阿里云开发者社区](https://developer.aliyun.com/article/922952)

2.1 **镜像下载**

|  |
| --- |
| Bash docker pull gitlab/gitlab-ce:latest docker images |

版本区别

* latest: 最新稳定版本的 GitLab 社区版。
* nightly: 每夜构建版本，反映了最新的开发进度。
* rc: 发布候选版本，用于最终测试稳定性和功能。
* <version-tag>: 特定版本的 GitLab，用于精确控制使用的 GitLab 版本。

2.2 **启动镜像**

|  |
| --- |
| Bash docker run -d --name gitlab gitlab/gitlab-ce docker run -d --name gitlab --restart always gitlab/gitlab-ce  docker run -d --name gitlab --restart always \  -p 8443:443 \  -p 8080:80 \  -p 8022:22 \  -v /mydata/gitlab/data/etc:/etc/gitlab \  -v /mydata/gitlab/data/log:/var/log/gitlab \  -v /mydata/gitlab/data/opt:/var/opt/gitlab \  /gitlab/gitlab-ce:latest |

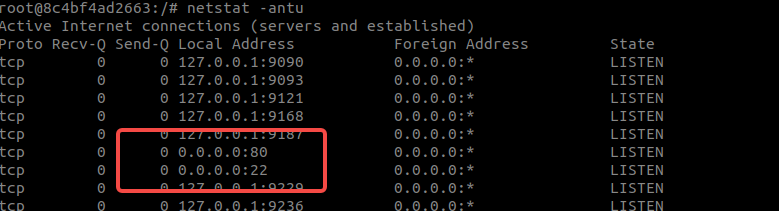
-restart always：不管退出状态码是什么，容器都会被重新启动

-p:端口映射：宿主机端口:容器端口

-v:文件挂载：宿主机文件:容器文件

除了 always，Docker 还提供了其他几种重启策略：

* no：容器不会在退出时自动重启。这是默认设置。
* on-failure[:max-retry]：仅在容器非正常退出时（退出状态非0）重启容器。可以选择性地附加最大重启尝试次数（max-retry）。
* unless-stopped：容器总是会自动重启，除非它被人工停止。如果在 Docker 重新启动时容器是停止状态，则该容器不会被自动启动。



看到容器里80和22端口监听，在run是将80端口映射到本地，否则http://local:80无法访问,单http://容器ip:80，千万别瞎映射，否则出一堆问题



要让docker容器运行一段时间，(health: starting)变为(healthy)

/etc/gitlab：这个卷是用来存储 GitLab 的配置文件的。 目录内通常包含 gitlab.rb 文件，以及其他的 GitLab 配置文件。

/var/log/gitlab：该卷用于存放 GitLab 服务的日志文件。GitLab 和它的组件（比如 GitLab Rails, GitLab Shell, GitLab Workhorse, NGINX, PostgreSQL等）的日志都会被写入到 目录下。

/var/opt/gitlab：这个卷是 GitLab 用来存储数据的地方，包括数据库文件、仓库数据（repositories）、上传的文件、LFS 对象、备份等。

2.3 **误区知识**

我们一直启动时使用 docker run -it 镜像 bash ，生成容器，但是it 产生伪终端就会把gitlab服务替换导致服务起不来。gitlab服务是一直在运行，只需docker -d 镜像服务在后台运行.

|  |
| --- |
| Plain Text docker run -it gitlab/gitlab-ce bash |

这个命令使用 -it 参数来运行容器。-i 或 --interactive 保持容器的标准输入开放，而 -t 或 --tty 分配一个伪终端。这个命令的主要目的是让你能够与运行中的容器进行交互。

当你在命令末尾指定 bash 时，你实际上是覆盖了容器的默认启动命令（即 GitLab 的启动命令）。这意味着，容器启动后会直接进入一个 bash shell 而不是执行其默认的启动流程（如启动 GitLab）。因此，你会发现 GitLab 没有自动启动，因为容器被用来提供一个交互式的 shell 环境而不是运行 GitLab。

|  |
| --- |
| Plain Text docker run -d gitlab/gitlab-ce |

-d 参数或者 --detach 表示在后台模式运行容器。这意味着 Docker 会按照 Dockerfile 中定义的默认命令来启动容器。对于 gitlab/gitlab-ce 镜像来说，这个默认命令涉及启动 GitLab 和所有相关的服务。因此，在这种模式下，GitLab 能够正常启动，因为容器没有被指定运行一个交互式的 shell，而是执行其默认的启动脚本。

2.4 **默认账户密码**

2.4.1 **账号**

|  |
| --- |
| Bash /etc/gitlab/initial\_root\_password |

账号root，密码查看目录下面

2.4.2 **代码下载**

|  |
| --- |
| Bash git@22425765a523:my\_group\_a/my\_code.git 将容器主机名改为ip git clone http://172.17.0.5:my\_group\_a/my\_code.git 注意现将ssh公钥填上去 |