Linux 命令 /Git Bash命令

- shell编程
- 潜规则：命令执行后没有任何输出，即表示执行成功

```
ls                  // 查看当前文件夹下的文件 (list单词缩写)
pwd                 // 查看当前所处的绝对路径
cd                  // 进入某一文件夹内
cd ..               //回到上一级目录
clear               // 清屏;也可以使用  ctrl+l

mkdir  文件夹名      // 创建文件夹
touch 文件名         // 创建文件

rm  文件名          //删除文件,无法找回
rm 文件夹名 -r          //删除文件夹 -r ; 强制删除 -f

mv 文件名 目标文件地址              //将指定文件移动到指定地址
mv 文件名 新的文件名                //更改指定文件的命名
mv 文件名 指定路径/新文件名         //将文件移动到指定路径并更名为新文件名

cat 文件名          //查看文件里面的内容
ctrl + c           //取消命令，当写错时 另起一行   //中止终端中的运行程序
tab键              //自动补全路径
上下方向键         //挑选曾经输入过的命令

q字母 / 输入exit/quit   //Linux 命令退出操作

cd -h / --help          //获取帮助
git -h / --help         //获取帮助
git remove -h           //获取remove的相关解释

exit 或 quit 或 ctrl+D        //退出终端
```

例子：

进入c盘 `cd /c`

进入文件 `cd 文件名`（tab键自动补全路径）

删库跑路(强制删除根目录)：`rm / -rf`

同时创建多个文件夹：`mkdir css html js imags`

新建文件/查看当前文件下的文件，可以不切换路径：在输入命令时后面跟个路径就可以了。

Vim文本编辑器

vim是一款命令行下的文本编辑器，编辑方式与图形化的编辑器不同。

因为:在非图形化的 Linux下无法使用VS code等图形化编辑器。

vim命令模式--底线命令　　vim命令模式--插入模式

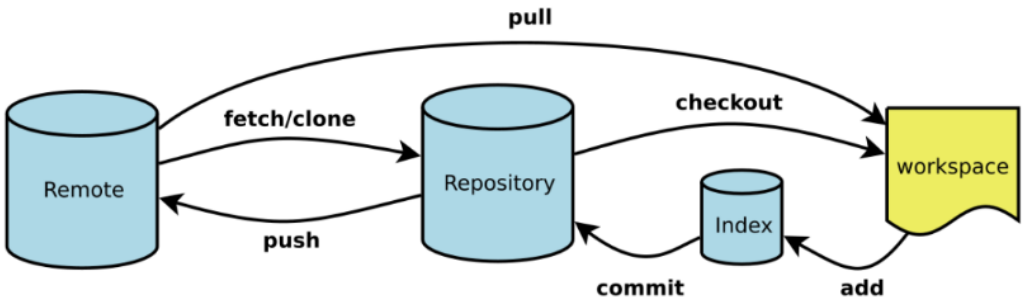
```
vim  文件名          //进入vim模式，编辑文件的内容

i / a / o           # 进入插入模式，才能开始写入内容
Esc键               # 退回到  vim命令模式
:w                  # 进入底线命令模式，保存文件
:q                  # 进入底线命令模式，退出vim模式,且不保存
: wq                # 进入底线命令模式，保存并退出

ctrl+r              # 重做，写错时删除上一行
ctrl+insert         # 复制
shift+insert        # 粘贴
#                   # vim git 中#号表示注释内容，后面可以跟任何内容。
```

注意：一定要在英文模式下输入命令!

Git

> 一款免费开源的分布式版本控制系统，是一个应用程序
>
> - 代码备份、版本退回、协作开发、权限控制
> - 在 Git Bash 中，使用的是 Linux 环境的命令。
> - 下载 Git：Git - Downloads (git-scm.com)
> - 常见平台：Git Hub、Gitee(码云)、GitLab
> - Git - Book官方教程

- Git 三区：
  - 工作区（代码编辑区）　　项目文件夹，写代码时就是在工作区修改内容
  - 暂存区（修改待提交区）　　`.git`的 index文件，修改的待提交区，　add
  - 仓库区（代码保存区）　　`objects`仓库，提交后的代码都放在里面。　commit

    Git是目前世界上最先进的分布式版本控制系统。
    工作原理 / 流程：
  - 

    - Workspace：工作区
    - Index / Stage：暂存区
    - Repository：仓库区（或本地仓库）
    - Remote：远程仓库
- Git分支
  - 分支是 Git 的重要功能特性之一，开发人员可以在主开发线的基础上分离出新的开发线。
  - 默认：`master`主分支 ；　其他分支的命名由创建分支时自定义。
    - 创建分支前需提交过代码，有主线存在，否则无法建立分支。

    - 在某个节点创建分支时，新创建的分支中包含该节点仓库中的所有代码。
    - 版本的切换、分支的切换前：一定要先对工作区的内容进行提交，否则切换后，原有未提交的代码文件会继续存在工作区 影响对切换后的代码。
- Git 初始配置
    - 第一次使用时，需要配置自己的用户名和邮箱，用于表示开发者的信息。
    - 安装后配置一次即可，如果不配置 `git commit` 无法生效

```
git config --global user.name "用户名"                  //配置用户名
git config --global user.email "1602453034@qq.com"     //配置邮箱
git config -l          //查看配置信息,l=list
```

Git 基本操作

- 借助vscode代码管理区域，手动选择更改提交的文件（不是自己修改的不要提交）
- 提交前先 pull 拉取代码
-

```
git init                          # 在当前路径下初始化仓库,初始化前的变化无法记录

git add 文件路径(index.html)    # 将指定文件加入暂存区；添加的是修改，而不是文件
git add -A                        # 将所有的修改都加入暂存区 A == all

git commit -m '提交注释'          # 将暂存区的内容提交到仓库，提交注释用来描述提交的内容。
git commit      #不写提交注释，会进入vim界面，需要填写并保存后退出才能正常存入暂存区。

git branch name             # 创建新的分支，name为分支的名称
git branch                  # 查看分支，列出本地已经存在的分支
git branch -r               # 查看远程版本库的分支列表
git branch -a               # 查看所有分支列表（包括本地和远程，remotes/开头的表示远程分支）
git branch -v               # 查看一个分支的最后一次提交
git branch --merged         # 查看哪些分支已经合并到当前分支
git branch --no-merged      # 查看所有未合并工作的分支

git checkout name           # 切换为name的分支仓库
git checkout -b name        # 创建并切换到新创建的分支
git branch -d name          # 删除指定的分支
git merge name              # 将指定分支合并到当前分支，不删除被合并的分支；合并后，提示分支中有merging，表示存在冲突。


git push -u origin master   # 将本地（master分支）推送到别名为origin的远程仓库
git push     # 进行过提交且进行-u分支关联的分支 再次提交时的简写
git push -f  # main分支名  -f强制覆盖仓库原有内容

git pull                    #将远程仓库更新到本地，且只拉取当前所在的仓库
git pull origin master      #将指定远程仓库的分支更新到本地  origin是远程仓库的别名；master是远程仓库的分支名
```

```
补充：
git log              #打印完整的仓库记录 查看HEAD日志
```

```
git log --oneline    #以简短的形式 显示该版本之前的显示仓库的记录
git reflog           #查看所有的版本操作记录，能够显示该版本之后的新版本编号。

git ls-files          # 查看暂存区的文件
git status         #查看已经做的修改，查看当前版本库的状态
  # 结果1：nothing to commint，working tree clean  表示所有的修改都已经存档
  # 结果2:红的文件名表示未放入暂存区，绿色表示已经放在暂存区，正常白色显示记录。

git diff              # 查看工作区与暂存区的差异(不显示新增文件)，显示做了那些修改。
git diff --cached     # 查看暂存区与仓库最新版本的差异。

git restore 文件名    # 还原文件，撤销对它的修改
git reset --hard 版本编号前七位    # 回滚到指定版本

git rm -cached 文件名              # 从版本库中删除该文件
```

- 代码回退
    1. reset是回退代码到某一版本，某一版本以后的代码都不保存
    2. revert是只回退某一版本代码，对齐它版本代码不影响(推荐)

```
1）回退到某一版本
git log  #查看HEAD日志
git reset --hard [目标版本号]    #目标版本号为HEAD编号，一般输前几位就可
git push -f  #将代码强制推送到远程仓库中

2）回退某一版本代码
git log  #查看HEAD日志
git revert [要回退的版本号]  #回退该版本代码并生成新的版本号
git status #查看本地变化的文件，是回退那个版本变化的文件，将其改回来了
git add .  #提交问价难道暂存区
git commit -m ''  #提交代码到本地仓库
git push  #上传到远程分支
```

- 本地有仓库
    - 获取远程仓库地址，本地配置远程仓库的地址
    - 本地提交后，再将本地仓库推送到远程仓库
    - readme.md上传到 github中会被以页面形式解析出来。
    - 注意事项
        - `GitHub` 仓库的存储器命名，不要使用中文，否则会出现一些错误。
        - 新构建的仓库不建议选择推荐的配置

```
git remote                 # 查看已经存在的远程仓库别名
git remote remove origin   # 移除origin这个别名 ，remove删除，rename重命名，add添加

# 创建远程仓库的别名  add添加  origin代表该地址仓库的别名
git remote add origin https://github.com/wuZhongtian/linshicangku121342.git
# 将本地仓库当前分支进行重命名为：main
git branch -M main
```

```
# 将本地的（main分支）推送到别名为origin的仓库；（需要输入github令牌、账号和密码）
# push推送    main分支名 -u分支关联，下次提交时可直接git push  -f会强制覆盖仓库原有内容
git push -u origin main -f

# 进行过提交且进行-u分支关联的分支 再次提交时的简写。
git push
```

- 本地没有仓库
  - 根据官网指示，配置此电脑的ssh公钥，加入安全组


```
# 克隆远端仓库，将项目代码同步到本地，本地会产生默认的远程地址的配置，别名为 origin
# git clone  将仓库的所有分支全部下载到本地
git clone https://github.com/wuZhongtian/linshicangku121342.git

# 本地修改代码并保存到本地仓库
# 将指定分支推送到指定的远程仓库
git push origin master

# 默认已经关联，可以直接进行 git push，只推送当前分支
# origin远端仓库地址的别名     master本地分支名
git push
```

注意：

- `git init` 仓库初始化时自动生成 `.git` 文件，若没有可以尝试打开 `显示隐藏文件`。
- 每`git init`一次就创建一个新的仓库。
- 只有在有`.git`的文件夹下才能执行`git命令`
- 版本切换为旧版本后，`git log --oneline`命令无法查看该版本之后的新版本。
- 切换版本前一定要进行当前代码的存档，否则工作区未保存的新代码会在切换版本后依然存在，影响切换后代码的状态。
- 对于不小心已经存入 git仓库的忽略文件需要做三步操作，具体看第5部分（配置忽略文件）。

常见问题

配置忽略文件

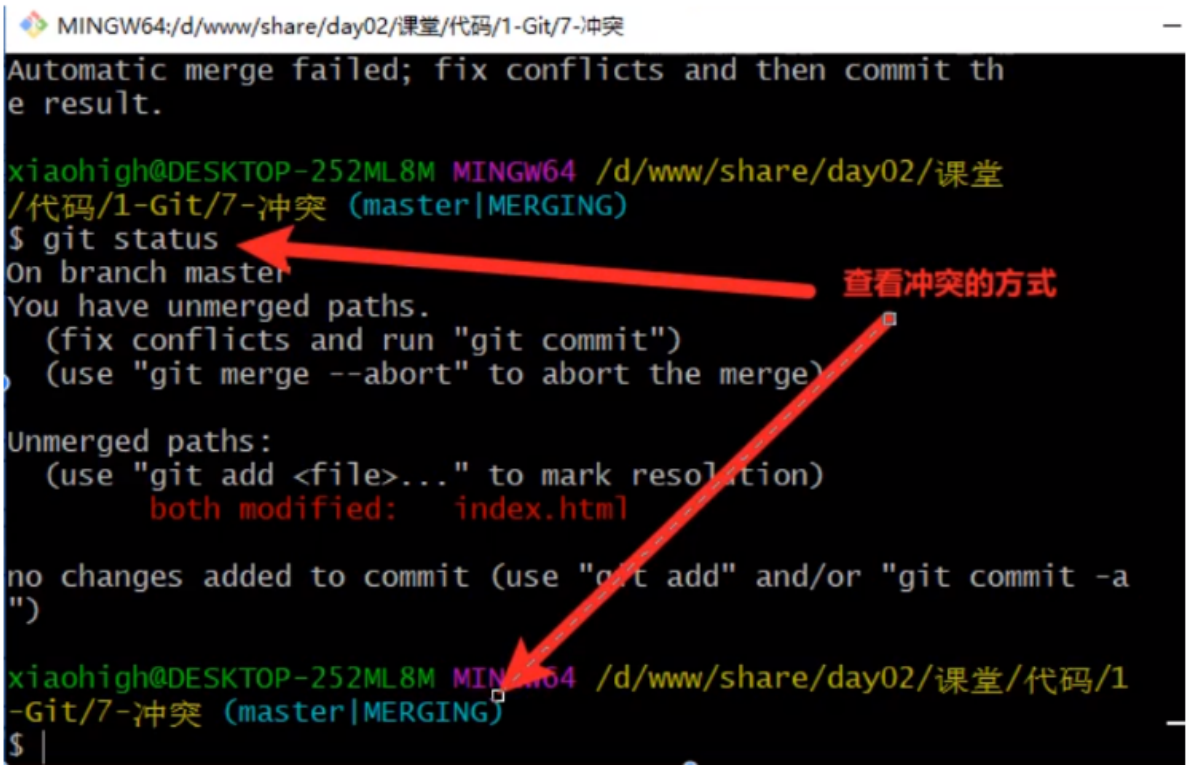
- 常见的忽略文件（项目中不应出现在版本库中的文件）
  - 临时文件
  - 多媒体文件，如音频、视频。
  - 编辑器生成的配置文件（.idea 不属于项目文件）
  - `npm`安装的第三方模块
- 误存入版本仓库的文件设置忽略

- 从版本库中 删除该文件 `git rm -cached 文件名`
- 在`.gitigonre 中配置忽略规则`
- 进行 `add`和 `commit`再次提交到仓库即可

```
# 创建 .gitignore配置文件，一般与 .git目录同级
#  .gitignore中写入 忽略规则

/app.swp          # 忽略当前文件夹下的 app.swp
text.html         # 直接写文件名，忽略当前项目下所有名为 text.html 的文件
*.mp4             # *+后缀名，忽略当前项目中所有的 .mp4 后缀的文件
.文件夹名         # 忽略所有指定的文件夹
/node_modules     # 忽略当前文件夹下的 node_modules 文件和文件夹
```

冲突合并

- 合并分支前，需要先切换到合成的分支中，再合并需要合并的分支.
- 当多个分支修改同一个文件后，合并分支时就会产生冲突。
- 解决思路：将内容修改为最想要的结果，然后继续执行`git add`与`git commit`保存到即可。
- 冲突的文件合并后会将两者的内容都合并到一个文件中。
  - `git status`定位冲突的文件
  - 修改冲突的文件内容 为最终正确的内容



.git 目录

- hooks 目录：包含客户端或服务器的钩子脚本，在特定操作下自动执行。类似于 JS事件，自动执行。
- info 信息文件夹：包含一个全局性排除文件、可配置文件忽略。不属于项目代码的临时性文件。
- logs 日志：记录我们的提交历史记录。
- objects 仓库目录：每次版本的更改都会保存再里面。
- refs 分支信息：
- index 暂存文件：存放暂存区文件的文件。

多人协作

Github 团队协作开发管理比较容易，可以创建一个组织。

- 首页 — 右上角+号 — new Organization
- 免费计划
- 填写组织名称和联系方式（不能使用中文名称）
- 邀请其他开发者进入组织（会发送邮件邀请）
- 点击仓库的settings 设置
- 将用户权限由'Read'设置为'Write'
- 创建多人协作仓库时，一定要在组织中创建仓库
- 每次提交前都要 将远端仓库更新到本地。

```
git pull                    #将远程仓库更新到本地，且只拉取当前所在的仓库
git pull origin master      #将指定远程仓库的分支更新到本地  origin是远程仓库的别名；
master是远程仓库的分支名
```

第一次：

- 得到`Github`远程仓库的地址
- 将代码克隆clone 到本地
- 切换分支（不要在主分支做修改，另建分支用来存放提交自己的代码）
- 合并分支（在确定代码没有问题后，**切换到主分支，进行合并，并处理冲突）**
- 在指定分支上更新本地代码`git pull`
- 在指定分支上提交代码`git push`

第n次：

- 更新代码（更新主分支的代码即可）

  ```
  git checkout master
  git pull
  ```

- 切换并合并分支（将最新的主分支合并到自己的开发分支中，再进行开发）

```
git checkout name
#切换到自己开发使用的分支
git merge master
#将 master 分支合并到自己的开发分支上
```

- 开发功能
- 提交
- 合并分支（将自己确认无误的分支合并到主分支）
- 更新代码（更新本地主分支的内容）
- 推送代码

注意：

- 所有的工作不要直接在主分支上进行操作
  - push操作后，原有的的代码会被直接覆盖掉
  - 在自己创建的分支上进行开发
- **每次提交代码到远端仓库时都必须先将远端仓库更新pull 到本地，保证不存在差异代码。**
- 提交时，相当于将原代码与现存代码进行合并，在弹出的vim文本编辑器中直接保存退出即可。`:wq`
- 在进行仓库的操作时，不能进行仓库套仓库。不能在仓库里再克隆一个仓库。否则出错！
- 很多情况下企业并不使用`Github`作为企业仓库。
- 好习惯：**每次修改代码后，及时提交到本地仓库**

Git Flow 最佳实践

- Master 主分支：只保存正式发布的版本。
- Hotfix 线上代码 bug修复分支：修复BUG后需要合并回Master和Develop分支，同时在Master上打一个tag
- Release 分支：待发布分支，居于Develop分支单独创建，在这个分支上进行发布前的测试和修复BUG。
- Develop 开发分支：开发者都在该分支上进行开发。
- Feature 功能分支：当各个开发者进行功能开发时，都新建自己的分支进行开发，完成后合并到Develop分支。

GitFlow 是团队开发的一种最佳实践，将代码划分为以下几个分支