# 学习路径：

学习git一般知识√

<https://www.runoob.com/git/git-workspace-index-repo.html>

图示

描述已自动生成

# 配置文件及层级

Git 提供了一个叫做 git config 的工具，专门用来配置或读取相应的工作环境变量。

这些环境变量，决定了 Git 在各个环节的具体工作方式和行为。这些变量可以存放在以下三个不同级别：

## 系统级别system(优先级最低)

对所有用户都普遍适用的配置。

文件位置：程序安装路径\Git\etc\gitconfig

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

文本

描述已自动生成

## 用户级别global(优先级次之)

只适用于该用户

文件位置：用户目录/.gitconfig 文件。

若使用 git config 时用 --global 选项，读写的就是这个文件。

图形用户界面, 应用程序, 表格

描述已自动生成

## 仓库级别local(优先级最高)

仅仅针对当前项目有效

文件位置：工作目录/.git/config

注意：/.git/通常是**隐藏**文件夹

若使用 git config 时用 --local 选项，读写的就是这个文件。

每一个级别的配置都会覆盖上层的相同配置，所以 .git/config 里的配置会覆盖 /etc/gitconfig 中的同名变量。

在 Windows 系统上，Git 会找寻用户主目录下的 .gitconfig 文件。主目录即 $HOME 变量指定的目录，一般都是 C:\Documents and Settings\$USER。

此外，Git 还会尝试找寻 /etc/gitconfig 文件，只不过看当初 Git 装在什么目录，就以此作为根目录来定位。

# 配置: 增删查改配置参数

## 查询：所有配置的参数 --list 或 -l

显示所有配置，包括三层级别的配置

git config --list

文本

描述已自动生成

查看系统的所有配置：--system

git config --system –list

文本

描述已自动生成

最后(END)，可以按”q”退出。

查看用户的所有配置：--global

git config --global –list

文本

描述已自动生成

查看用户的所有配置：--local

git config --local –list

文本

描述已自动生成

## 查询：特定key的配置值 --get section.key或section.key

git config --global <section.key>

文本

描述已自动生成

git config --local --get <section.key>

文本

描述已自动生成

当没有找到key时，不会返回值：

list后只有app.name，没有app.names。所以不会返回值。

文本

描述已自动生成

## 添加：特定key的配置值 --add

git config --global --add <section.key> <value>

文本

描述已自动生成

git config --global <section.key> <value>

文本

描述已自动生成

## 修改：特定key的配置值 section.key value

git config --local <section.key> <value>

文本

描述已自动生成

## 删除：删除特定键值 --unset

git config --local --unset < section.key >

文本

描述已自动生成

如果没有section.key，则不会返回值。

## 编辑器编辑：--edit 或 -e

git config --local --edit

是调用编辑软件打开配置文件

图形用户界面, 文本, 应用程序

描述已自动生成

# 工作区操作

## 添加/删除/更改: 使用IDE

## 查看：使用命令 ls(powershell)/dir(CMD)

文本

描述已自动生成

已缓存文件但在工作区删除的文件，使用命令查看时不会返回文件。

文本

描述已自动生成

## 查看历史：工作区指定文件的修改历史git blame

命令

git blame <filename>

必须指定文件名

图形用户界面, 应用程序

描述已自动生成

文本

描述已自动生成

注意：查看的是工作区的指定文件

文本

描述已自动生成

## 恢复：从缓存区复制到工作区 git checkout

命令

git checkout **<filename>**

### <filename>：恢复指定缓存区指定文件到工作区

电脑屏幕的手机截图

描述已自动生成

文本

描述已自动生成

当没有带有指定文件名时，不会起作用。

文本

描述已自动生成

当指定文件没有修改时，会提示updated 0。

图形用户界面, 文本

中度可信度描述已自动生成

注意：是从缓存区恢复文件到工作区

以下：

1. 查询状态，MM表示工作区，缓存，仓库的文件都不相同；
2. 恢复文件，
3. 查询状态，M表示已缓存的修改。即checkout指定文件是从缓存区中恢复到工作区。

文本

描述已自动生成

### -f：强制恢复缓存区所有文件

文本

描述已自动生成

# 缓存区操作

## 查看：缓存区中文件 git ls-files

命令

git ls-files

参数：

--cached (-c) 查看暂存区中文件，git ls-files 命令默认是此命令

--midified （-m）查看修改的文件

--delete （-d）查看删除过的文件

--other （-o）查看没有被git跟踪的文件

--stage (-s) 显示mode以及文件对应的Blob对象，进而我们可以获取暂存区中对应文件里面的内容。

文本

中度可信度描述已自动生成

## 添加/更新：从工作区添加 git add

### 添加/更新：工作区的指定文件到缓存区

git add <filename>

在VSC中，工作区中没有提交缓存区的文件会带有U标识。当提交到缓存区后，会带有A标识。

图形用户界面, 文本, 应用程序

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

提交缓存区后，可使用git ls-file命令查看缓存区的所有文件。

文本

描述已自动生成

如果没有指定文件时会显示did not match any files

图形用户界面, 文本, 网站

描述已自动生成

### 添加工作区的指定目录到缓存区

git add [dir]

### 添加/更新当前目录下的所有文件到缓存区

git add .

文本

描述已自动生成

注意：包括当前路径中的所有文件夹

文本

描述已自动生成

## 删除：缓存区中的文件 git rm --cached

删除文件。

命令

git rm --cached <filename>

缺少指定文件名：显示No pathspec was given

文本

描述已自动生成

### 文件名不在缓存区：则显示not match any files

注意，只对缓存区作用

文本

描述已自动生成

### 无参数：会提示使用参数

文本

描述已自动生成

### --cached：只删除缓存区中的文件，保留工作区文件

文本

描述已自动生成

### -f：强制删除缓存区并删除工作区的文件

以下的stage.py是在缓存区中，当使用参数-f后悔删除缓存区和工作区的文件。

文本

描述已自动生成

注意：是现在缓存区中寻找目标文件，找不到则显示not match；找到则删除缓存区文件，并删除工作区文件。

文本

描述已自动生成

### -r \*：递归当前路径下得所有文件名

查看当前状态，返现目前路径工作区中的文件都已经缓存。

然后使用-r \*，再加强制执行参数-f，将会递归当前路径的文件名并强制删除。

文本

描述已自动生成

注意：递归包括当前路径的子文件夹

文本

描述已自动生成

注意：当前路径工作区中文件有没有缓存区的文件时，由于-r \*是递归所有文件名，则当递归到没有缓存的文件名时，会不执行删除返回not match.

文本

描述已自动生成

注意：当存在AD状态的文件时，即已缓存的文件在工作区删除时，由于工作区递归没有出现AD状态文件名，所以缓存区会保留AD状态的文件。

文本

描述已自动生成

注意：对于AM文件，由于存在修改的差异，会被中止执行。

图形用户界面, 文本

描述已自动生成

# 本地仓库操作

## 创建：本地git仓库 git init

Git 使用 **git init** 命令来初始化一个Git仓库.

在执行完成 **git init** 命令后，Git 仓库会生成一个 .git 目录，该目录包含了资源的所有元数据，其他的项目目录保持不变（不像 SVN 会在每个子目录生成 .svn 目录，Git 只在仓库的根目录生成 .git 目录）。

### 在当前目录创建git仓库

git init

会在当前文件夹中创建.git隐藏文件夹。



图形用户界面, 文本, 应用程序

描述已自动生成

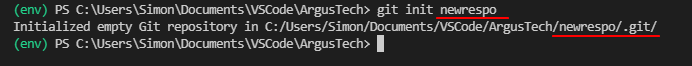
重复输入git init时会重新初始化本地仓库

图形用户界面, 文本, 电子邮件

描述已自动生成

### 在子目录中创建git仓库

git init <subfolder>



图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

### 通用命令

mkdir <new folder>

cd <new folder>

git init

mkdir：创建子目录

cd：进入子目录

git init：创建git仓库

## 克隆：现有的git仓库 git clone

### 克隆目标项目到当前路径

url: 目标项目的路径

git clone <repo>

repo: 目标git 仓库。

克隆github仓库时，repo参数时GitHub项目的url。

git clone git://github.com/schacon/grit.git

效果：

1. 在当前路径创建以github仓库命名的文件夹；
2. 在文件夹中创建.git隐藏文件夹用作本地git仓库；
3. 克隆所有远端仓库的文件；
4. 创建分支master；

图形用户界面, 文本

描述已自动生成

git clone 时，可以所用不同的协议，包括 ssh, git, https 等，其中最常用的是 ssh，因为速度较快，还可以配置公钥免输入密码。各种写法如下：

git clone git@github.com:fsliurujie/test.git --SSH协议

git clone git://github.com/fsliurujie/test.git --GIT协议

git clone https://github.com/fsliurujie/test.git --HTTPS协议

实践：从github克隆仓库到本机

文本

描述已自动生成

### 克隆目标仓库到指定路径

git clone <repo> <directory>

repo: 目标git 仓库。

directory：文件夹名/路径

git clone git://github.com/schacon/grit.git mygrit

效果：会在当前路径中创建指定文件夹名的子文件夹。

文本

描述已自动生成

当directory参数可以是路径。

文本

描述已自动生成

以上的效果是在父目录中创建mygrit文件夹

图形用户界面, 文本, 应用程序

描述已自动生成

## 查看变化：上一次提交后的文件变化git status

显示所有文件变化，包括工作区与缓存区的对比，缓存区与本地仓库的对比

命令

git status

注意：自上一次提交后没有改变时，返回：

文本

描述已自动生成

参数

### 无参数：显示所有改动的文件

changes to be committed：在缓存区中被改动的文件，包含新增的文件

changes not staged for commit：已经在缓存区但在工作区中被改动的文件

untracked files：在工作区中没有提交

文本

描述已自动生成

### -s：简短输出结果

* ??表示在工作区但没有缓存
* A表示已经缓存，无提交
* M表示已缓存但工作区中文件被修改
* D表示已缓存但工作区中文件被删除

文本

描述已自动生成

## 查看历史：提交历史 git log

命令

git log

### 无参数：显示详细历史

文本

描述已自动生成

### --oneline：查看历史记录的简洁的版本

文本

描述已自动生成

可以与--reverse共用。

文本

描述已自动生成

### --graph：以拓扑图显示历史。

文本

描述已自动生成

可以与--oneline共用。

文本

描述已自动生成

### --author：指定用户的历史

文本

描述已自动生成

## 提交/删除/修改：将缓存区中的文件提交到仓库git commit

命令

git commit

如果自上一次提交后没有修改，则会返回nothing to commit

文本

描述已自动生成

参数：

### 无参数：无参数时，VSC会打开一个文件。

如果没有输入信息，则会中止提交；

如果输入信息以“#”开头，则视作无输入信息；

输入信息后关闭文件，相当于参数-m ‘message’。

### -m ‘message’：message是提交时的信息。

以下是

1. 查询状态，显示有未提交缓存的修改文件stage1.py;
2. 缓存文件，并查询状态，显示修改文件已经缓存；
3. 提交缓存区中文件并留下信息‘commit1’;
4. 查询状态，显示无更新。

文本

描述已自动生成

### -a：提交缓存区最新修改。

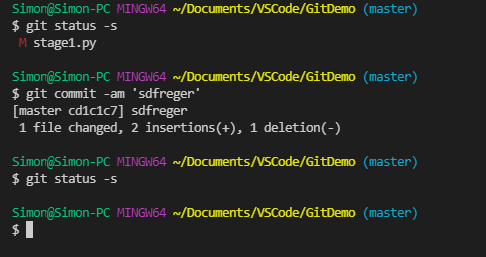
一般使用参数-am ‘message’。

注意1：提交的是已经缓存的文件。对于工作区中没有缓存的文件无法直接提交。

Git的提交只是针对缓存区中的文件。

注意2：提交的是工作区中最新的修改。即免去git add的步骤。

以下红色标识“M”标识工作区有修改但没有缓存。



# 恢复版本：从已提交的仓库恢复到缓存区git reset

## 基本参数

命令

git reset <head>

### 无参数：缺少参数。不会起作用。

文本

描述已自动生成

### 参数<head>：从指定的head仓库恢复到暂存区，并将当前head恢复到指定head。

参数形式1：

* HEAD^ 上一个版本
* HEAD^^ 上上一个版本
* HEAD^^^ 上上上一个版本

参数形式2：

可以使用 **～数字**表示

* HEAD~0 表示当前版本
* HEAD~1 上一个版本
* HEAD^2 上上一个版本
* HEAD^3 上上上一个版本

以此类推...

参数是指定的head。

以下：

1. 查询状态：工作区、缓存区、仓库文件都一致。
2. ls-files显示当前缓存区只有demo1.py;
3. reset后，恢复此前的demo1.py和demo2.py;
4. ls-files显示当前缓存区恢复了demo1.py和demo2.py。
5. 查询状态：工作区与缓存区不一致。即reset恢复缓存区，保留工作区。

文本

描述已自动生成

注意：恢复后，当前head会变成指定的head。

文本

描述已自动生成

## --mixed：保留修改在工作区(默认)。

当没有参数时，适用--mixed.

以下：

1. 查询log，显示当前HEAD是modified demo1.
2. 恢复上一版本到缓存区，并保留当前工作区修改。所以显示unstaged changes.
3. 查询log，显示当前HEAD是create demo1.
4. 查询当前状态，显示当前demo1是M，即工作区修改了已缓存的demo1. 证明mixed参数是保留工作区修改，但缓存区是指定版本。

文本

描述已自动生成

## --soft：保留修改在缓存区

以下：

1. 查询log，当前HEAD是modified demo1;
2. 恢复--soft；
3. 查询log，当前HEAD是create demo1；
4. 查询状态，显示标识M，表示当前工作区的修改保留在缓存区。相当于工作区与指定版本的修改被git add。

文本

描述已自动生成

## --hard：不保留修改。工作区与缓存区都被恢复到指定版本。

以下：

1. 查询log，显示当前HEAD是modified demo；
2. 恢复版本；
3. 查询log，显示当前HEAD是create demo;
4. 查询状态，显示当前工作区，缓存区和指定版本相同。

所以--hard参数的效果是不保留修改，工作区与缓存区都被恢复到指定版本。

文本

描述已自动生成

## 总结：

--hard：不保留新修改。工作区=缓存区=指定仓库。

--soft：保留新修改在缓存区。工作区=缓存区 > 指定仓库。

--mixed/无参数：保留新修改在工作区。工作区 > 缓存区=指定仓库。

# 比较差异：

命令

git diff

## 基本：

参数：

### 指定文件名：查看指定缓存区中指定文件的差别

文本

描述已自动生成

### --stat：显示摘要而非整个

文本

描述已自动生成

## 无参数： 比较工作区与缓存区差异。

文本

描述已自动生成

## --cached：比较缓存区与HEAD的差异

以下：

1. 查询状态，显示当前工作区有修改，但未缓存；
2. 使用cached，返回为空。即缓存区=HEAD；
3. Add缓存到缓存区；
4. 查询状态，显示修改已缓存；
5. 使用cached，显示缓存区与HEAD的差异。

文本

描述已自动生成

## HEAD：比较工作区与指定HEAD的差异。

注意，HEAD可以以往版本。

以下：

1. 查询状态，当前工作区有修改；
2. 查询log，显示当前的HEAD；
3. 适用参数HEAD，显示工作区与当前HEAD的差异。
4. 最后适用cached，返回为空。即当前工作区>缓存区=HEAD。

文本

描述已自动生成

## HEAD1 HEAD2：比较两个版本

注意：比较是以HEAD1为参照系。

减号-：表示HEAD2比HEAD1少的内容；

加号+：表示HEAD2比HEAD1多的内容；

以下比较上一个版本和上上一个版本。

减号表示上上一个版本比最新的版本少。即最新版本比上上版本多。

文本

描述已自动生成

将参数调换，则显示为加号；

文本

描述已自动生成

# END