Git版本控制

注意：开始学习之前，确保自己的网络可以畅通的连接Github：https://github.com，这个是一个国外网站，连起来特别卡，至于用什么方式实现流畅访问，懂的都懂。

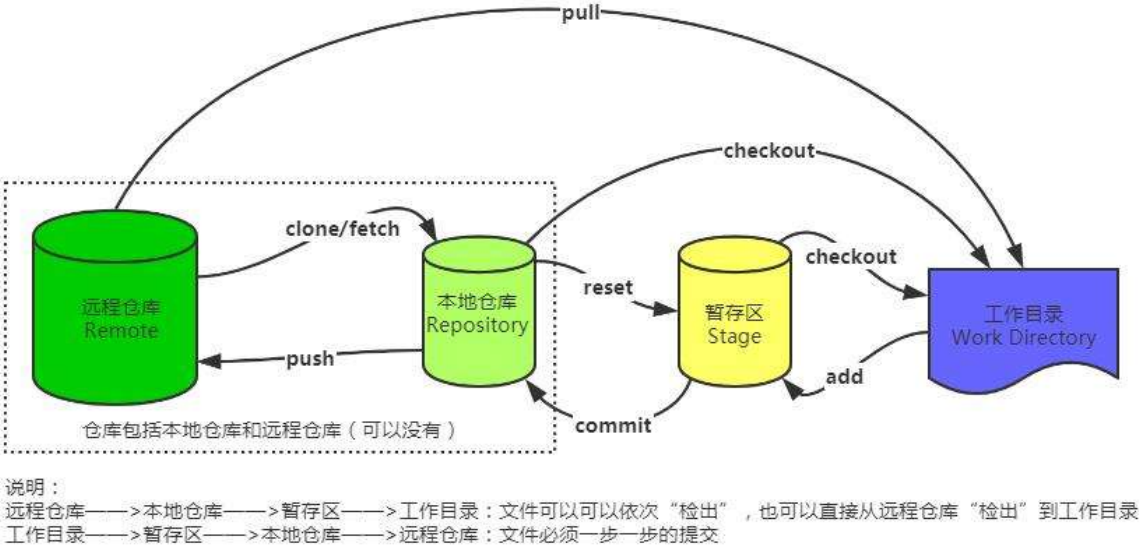
其实版本控制在我们的生活中无处不在，比如你的期末或是毕业答辩论文，由于你写得不规范或是老师不满意，你的老师可能会让你改了又改，于是就会出现下面这种情况：

我们手里的论文可能会经过多次版本迭代，最终我们会选取一个最好的版本作为最终提交的论文。使用版本控制不仅仅是为了去记录版本迭代历史，更是为了能够随时回退到之前的版本，实现时间回溯。同时，可能我们的论文是多个人一同完成，那么多个人如何去实现同步，如何保证每个人提交的更改都能够正常汇总，如何解决冲突，这些问题都需要一个优秀的版本控制系统来解决。

走进Git

我们开发的项目，也需要一个合适的版本控制系统来协助我们更好地管理版本迭代，而Git正是因此而诞生的（有关Git的历史，这里就不多做阐述了，感兴趣的小伙伴可以自行了解，是一位顶级大佬在一怒之下只花了2周时间用C语言开发的，之后的章节还会遇到他）

首先我们来了解一下Git是如何工作的：



可以看到，它大致分为4个板块：

- 工作目录：存放我们正在写的代码（当我们新版本开发完成之后，就可以进行新版本的提交）
- 暂存区：暂时保存待提交的内容（新版本提交后会存放到本地仓库）
- 本地仓库：位于我们电脑上的一个版本控制仓库（存放的就是当前项目各个版本代码的增删信息）
- 远程仓库：位于服务器上的版本控制仓库（服务器上的版本信息可以由本地仓库推送上去，也可以从服务器抓取到本地仓库）

它是一个分布式的控制系统，因此一般情况下我们每个人的电脑上都有一个本地仓库，由大家共同向远程仓库去推送版本迭代信息。

通过这一系列操作，我们就可以实现每开发完一个版本或是一个功能，就提交一次新版本，这样，我们就可以很好地控制项目的版本迭代，想回退到之前的版本随时都可以回退，想查看新版本添加或是删除了什么代码，随时都可以查看。

安装Git

首先请前往Git官网去下载最新的安装包：https://git-scm.com/download/win

这手把手演示一下如何安装Git环境。

安装完成后，需要设定用户名和邮箱来区分不同的用户：

```
git config --global user.name "Your Name"
git config --global user.email "email@example.com"
```

基本命令介绍

创建本地仓库

我们可以将任意一个文件夹作为一个本地仓库，输入：

```
git init
```

输入后，会自动生成一个`.git`目录，注意这个目录是一个隐藏目录，而当前目录就是我们的工作目录。

创建成功后，我们可以查看一下当前的一个状态，输入：

```
git status
```

如果已经成功配置为Git本地仓库，那么输入后可以看到：

```
On branch master

No commits yet
```

这表示我们还没有向仓库中提交任何内容，也就是一个空的状态。

添加和提交

接着我们来看看，如何使用git来管理我们文档的版本，我们创建一个文本文档，随便写入一点内容，接着输入：

```
git status
```

我们会得到如下提示：

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    hello.txt

nothing added to commit but untracked files present (use "git add" to track)
```

其中Untracked files是未追踪文件的意思，也就是说，如果一个文件处于未追踪状态，那么git不会记录它的变化，始终将其当做一个新创建的文件，这里我们将其添加到暂存区，那么它会自动变为被追踪状态：

```
git add hello.txt #也可以 add . 一次性添加目录下所有的
```

再次查看当前状态：

```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   hello.txt
```

现在文件名称的颜色变成了绿色，并且是处于Changes to be committed下面，因此，我们的hello.txt现在已经被添加到暂存区了。

接着我们来尝试将其提交到Git本地仓库中，注意需要输入提交的描述以便后续查看，比如你这次提交修改了或是新增了哪些内容：

```
git commit -m 'Hello World'
```

接着我们可以查看我们的提交记录：

```
git log
git log --graph
```

我们还可以查看最近一次变更的详细内容：

```
git show [也可以加上commit ID查看指定的提交记录]
```

再次查看当前状态，已经是清空状态了：

```
On branch master
nothing to commit, working tree clean
```

接着我们可以尝试修改一下我们的文本文档，由于当前文件已经是被追踪状态，那么git会去跟踪它的变化，如果说文件发生了修改，那么我们再次查看状态会得到下面的结果：

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   hello.txt
```

也就是说现在此文件是处于已修改状态，我们如果修改好了，就可以提交我们的新版本到本地仓库中：

```
git add .
git commit -m 'Modify Text'
```

接着我们来查询一下提交记录，可以看到一共有两次提交记录。

我们可以创建一个`.gitignore`文件来确定一个文件忽略列表，如果忽略列表中的文件存在且不是被追踪状态，那么git不会对其进行任何检查：

```
# 这样就会匹配所有以txt结尾的文件
*.txt
# 虽然上面排除了所有txt结尾的文件，但是这个不排除
!666.txt
# 也可以直接指定一个文件夹，文件夹下的所有文件将全部忽略
test/
# 目录中所有以txt结尾的文件，但不包括子目录
xxx/*.txt
# 目录中所有以txt结尾的文件，包括子目录
xxx/**/*.txt
```

创建后，我们来看看是否还会检测到我们忽略的文件。

回滚

当我们想要回退到过去的版本时，就可以执行回滚操作，执行后，可以将工作空间的内容恢复到指定提交的状态：

```
git reset --hard commitID
```

执行后，会直接重置为那个时候的状态。再次查看提交日志，我们发现之后的日志全部消失了。

那么要是现在我又想回去呢？我们可以通过查看所有分支的所有操作记录：

```
git reflog
```

这样就能找到之前的commitID，再次重置即可。

分支

分支就像我们树上的一个树枝一样，它们可能一开始的时候是同一根树枝，但是长着长着就开始分道扬镳了，这就是分支。我们的代码也是这样，可能一开始写基础功能的时候使用的是单个分支，但是某一天我们希望基于这些基础的功能，把我们的项目做成两个不同方向的项目，比如一个方向做Web网站，另一个方向做游戏服务端。

因此，我们可以在一个主干上分出N个分支，分别对多个分支的代码进行维护。

创建分支

我们可以通过以下命令来查看当前仓库中存在的分支：

```
git branch
```

我们发现，默认情况下是有一个master分支的，并且我们使用的也是master分支，一般情况下master分支都是正式版本的更新，而其他分支一般是开发中才频繁更新的。我们接着来基于当前分支创建一个新的分支：

```
git branch test
# 对应的删除分支是
git branch -d yyds
```

现在我们修改一下文件，提交，再查看一下提交日志：

```
git commit -a -m 'branch master commit'
```

通过添加-a来自动将未放入暂存区的已修改文件放入暂存区并执行提交操作。查看日志，我们发现现在我们的提交只生效于master分支，而新创建的分支并没有发生修改。

我们将分支切换到另一个分支：

```
git checkout test
```

我们会发现，文件变成了此分支创建的时的状态，也就是说，在不同分支下我们的文件内容是相互隔离的。

我们现在再来提交一次变更，会发现它只生效在yyds分支上。我们可以看看当前的分支状态：

```
git log --all --graph
```

合并分支

我们也可以将两个分支更新的内容最终合并到同一个分支上，我们先切换回主分支：

```
git checkout master
```

接着使用分支合并命令：

```
git merge test
```

会得到如下提示：

```
Auto-merging hello.txt
CONFLICT (content): Merge conflict in hello.txt
Automatic merge failed; fix conflicts and then commit the result.
```

在合并过程中产生了冲突，因为两个分支都对hello.txt文件进行了修改，那么现在要合并在一起，到底保留谁的hello文件呢？

我们可以查看一下是哪里发生了冲突：

```
git diff
```

因此，现在我们将master分支的版本回退到修改hello.txt之前或是直接修改为最新版本的内容，这样就不会有冲突了，接着再执行一次合并操作，现在两个分支成功合并为同一个分支。

变基分支

除了直接合并分支以外，我们还可以进行变基操作，它跟合并不同，合并是分支回到主干的过程，而变基是直接修改分支开始的位置，比如我们希望将yyds变基到master上，那么yyds会将分支起点移动到master最后一次提交位置：

```
git rebase master
```

变基后，yyds分支相当于同步了此前master分支的全部提交。

优选

我们还可以选择其将他分支上的提交作用于当前分支上，这种操作称为cherrypick：

```
git cherry-pick <commit id>:单独合并一个提交
```

这里我们在master分支上创建一个新的文件，提交此次更新，接着通过cherry-pick的方式将此次更新作用于test分支上。

使用IDEA版本控制

虽然前面我们基本讲解了git的命令行使用方法，但是没有一个图形化界面，始终会感觉到很抽象，所以这里我们使用IDEA来演示，IDEA内部集成了git模块，它可以让我们的git版本管理图形化显示，当然除了IDEA也有一些独立的软件比如：SourceTree（挺好用）

打开IDEA后，找到版本控模块，我们直接点击创建本地仓库，它会自动将当前项目的根目录作为我们的本地仓库，而我们编写的所有代码和项目目录下其他的文件都可以进行版本控制。

我们发现所有项目中正在编写的类文件全部变红了，也就是处于未追踪状态，接着我们进行第一次初始化提交，提交之后我们可以在下方看到所有的本地仓库提交记录。

接着我们来整合一下Web环境，创建新的类之后，IDEA会提示我们是否将文件添加到Git，也就是是否放入暂存区并开启追踪，我们可以直接对比两次代码的相同和不同之处。

接着我们来演示一下分支创建和分支管理。

远程仓库

远程仓库实际上就是位于服务器上的仓库，它能在远端保存我们的版本历史，并且可以实现多人同时合作编写项目，每个人都能够同步他人的版本，能够看到他人的版本提交，相当于将我们的代码放在服务器上进行托管。

远程仓库有公有和私有的，公有的远程仓库有GitHub、码云、Coding等，他们都是对外开放的，我们注册账号之后就可以使用远程仓库进行版本控制，其中最大的就是GitHub，但是它服务器在国外，我们国内连接可能会有一点卡。私有的一般是GitLab这种自主搭建的远程仓库私服，在公司中比较常用，它只对公司内部开放，不对外开放。

这里我们以GitHub做讲解，官网：https://github.com，首先完成用户注册。

远程账户认证和推送

接着我们就可以创建一个自定义的远程仓库了。

创建仓库后，我们可以通过推送来将本地仓库中的内容推送到远程仓库。

```
git remote add 名称 远程仓库地址
git push 远程仓库名称 本地分支名称[:远端分支名称]
```

注意`push`后面两个参数，一个是远端名称，还有一个就是本地分支名称，但是如果本地分支名称和远端分支名称一致，那么不用指定远端分支名称，但是如果我们希望推送的分支在远端没有同名的，那么需要额外指定。推送前需要登陆账户，GitHub现在不允许使用用户名密码验证，只允许使用个人AccessToken来验证身份，所以我们需要先去生成一个Token才可以。

推送后，我们发现远程仓库中的内容已经与我们本地仓库中的内容保持一致了，注意，远程仓库也可以有很多个分支。

但是这样比较麻烦，我们每次都需要去输入用户名和密码，有没有一劳永逸的方法呢？当然，我们也可以使用SSH来实现一次性校验，我们可以在本地生成一个rsa公钥：

```
ssh-keygen -t rsa
cat ~/.ssh/github.pub
```

接着我们需要在GitHub上上传我们的公钥，当我们再次去访问GitHub时，会自动验证，就无需进行登录了，之后在Linux部分我们会详细讲解SSH的原理。

接着我们修改一下工作区的内容，提交到本地仓库后，再推送到远程仓库，提交的过程中我们注意观察提交记录：

```
git commit -a -m 'Modify files'
git log --all --oneline --graph
git push origin master
git log --all --oneline --graph
```

我们可以将远端和本地的分支进行绑定，绑定后就不需要指定分支名称了：

```
git push --set-upstream origin master:master
git push origin
```

在一个本地仓库对应一个远程仓库的情况下，远程仓库基本上就是纯粹的代码托管了（云盘那种感觉，就纯粹是存你代码的）

克隆项目

如果我们已经存在一个远程仓库的情况下，我们需要在远程仓库的代码上继续编写代码，这个时候怎么办呢?

我们可以使用克隆操作来将远端仓库的内容全部复制到本地：

```
git clone 远程仓库地址
```

这样本地就能够直接与远程保持同步。

抓取、拉取和冲突解决

我们接着来看，如果这个时候，出现多个本地仓库对应一个远程仓库的情况下，比如一个团队里面，N个人都在使用同一个远程仓库，但是他们各自只负责编写和推送自己业务部分的代码，也就是我们常说的协同工作，那么这个时候，我们就需要协调。

比如程序员A完成了他的模块，那么他就可以提交代码并推送到远程仓库，这时程序员B也要开始写代码了，由于远程仓库有其他程序员的提交记录，因此程序员B的本地仓库和远程仓库不一致，这时就需要有先进行pull操作，获取远程仓库中最新的提交：

```
git fetch 远程仓库 #抓取：只获取但不合并远端分支，后面需要我们手动合并才能提交
git pull 远程仓库 #拉取：获取+合并
```

在程序员B拉取了最新的版本后，再编写自己的代码然后提交就可以实现多人合作编写项目了，并且在拉取过程中就能将别人提交的内容同步到本地，开发效率大大提升。

如果工作中存在不协调的地方，比如现在我们本地有两个仓库，一个仓库去修改hello.txt并直接提交，另一个仓库也修改hello.txt并直接提交，会得到如下错误：

```
To https://github.com/xx/xxx.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/xx/xxx.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

一旦一个本地仓库推送了代码，那么另一个本地仓库的推送会被拒绝，原因是当前文件已经被其他的推送给修改了，我们这边相当于是另一个版本，和之前两个分支合并一样，产生了冲突，因此我们只能去解决冲突问题。

如果远程仓库中的提交和本地仓库中的提交没有去编写同一个文件，那么就可以直接拉取：

```
git pull 远程仓库
```

拉取后会自动进行合并，合并完成之后我们再提交即可。

但是如果两次提交都修改了同一个文件，那么就会遇到和多分支合并一样的情况，在合并时会产生冲突，这时就需要我们自己去解决冲突了。

我们可以在IDEA中演示一下，实际开发场景下可能会遇到的问题。

至此，Git版本控制就讲解到这里，下一章我们会继续认识一个全新的数据库：Redis。