暑培第二讲 Git/GitHub

1. Git

为什么要用Git

在我们正式谈及“版本管理”之前，我们想象一个这样的场景：

1 月 1 日，我们要提交一份工作总结，我们将其命名为“工作总结.txt”。1 月 2 日，我们被要求修改工作总结。但是，当我们要修改这段文字的时候，我们要保留昨天的版本，防止我们改的更糟，还要该回去。但是为了防止命名冲突，我们不得不把之前的工作总结重命名为 “工作总结*01_01.txt*”，而本次的工作总结命名为“工作总结01_02.txt”。1 月 3 日……

然后就会出现很多文件，它们是同一个内容不同版本。这样做很不好管理，并且很不优雅。

为了解决这种问题，Linus老爷子写出了Git。Git 的开发工作于 2005 年 4 月开始，最初目的是为了作为 Linux 内核代码的版本管理工具，现在已经成为全世界程序员必备的工具之一。

Git简介

Git是一种分布式版本控制系统。

- 分布式：简单的说，分布式的版本控制就是每个人都可以创建一个独立的代码仓库用于管理，各种版本控制的操作都可以在本地完成。将备份的代码以及记录完全独立在本地存储，当你想将代码恢复到某一个版本的时候，本地版本控制，不需要依赖网络便可以完成此操作，因为本地版本控制器拥有完整独立的控制系统。
- 版本控制：对文件变更的管理。

好处：便于多文件多版本管理，协同开发，以防文件丢失。

Git安装与配置

在http://git-scm.com/downloads里下载，运行安装，逐步按照提示安装即可。

安装完成，在第一次使用之前配置自己的用户名和邮箱

```
$ git config --global user.name <username>
$ git config --global user.email <email>
```

Git中的相关概念

Git 本地数据管理，大概可以分为三个部分，工作区,暂存区和版本库。

- **工作区（Working Directory）**
  是我们直接编辑的地方，例如：记事本打开的文本等，肉眼可见，直接操作。
- **暂存区（Stage 或 Index）**
  数据暂时存放的区域，可在工作区和版本库之间进行数据的友好交流。
- **版本库（commit History）**
  存放已经提交的数据，“版本”。

Git本地操作

注意，回车之后如果没有消息

0. 最重要的操作，查询文档

```
$ git help <command>
```

1. 初始化仓库

```
$ git init
Initialized empty Git repository in D:/Fuyh/暑培/demo/.git/
```

这样就建好了一个空的本地仓库（生成了./.git目录）

2. 查看工作区、暂存区状态

```
$ git status
```

3. 添加到暂存区

```
$ git add <filename>
$ git rm --cached <filename> # 从暂存区里删除
```

4. 提交到本地库

```
$ git commit -m “commit message”
```

5. 查看提交历史

```
$ git log [options]
$ git reflog # 显示可引用的历史版本记录。
```

6. 删除文件

```
$ git rm <filename> # git rm 只是在暂存区删除了，并没有提交
$ git commit
```

7. 版本前进后退

本质：对HEAD指针进行操作

- `git reset <commit>` 将Git仓库的`HEAD`回退到某版本，清空暂存区，但是不修改目录中的文件。
- `git reset --hard <commit>` 前进、退后版本
- 使用^符号：只能往后退

  `git reset --hard HEAD^` 退后一步（几个^就回退几步）
- 使用~符号

  `git reset --hard HEAD~<步数>` 后退版本

> reset命令的三个参数对比：
>
> - --soft：仅仅在本地库移动指针
> - --hard：在本地库移动HAED指针、重置暂存区、重置工作区
> - --mixed：在本地库移动HAED指针、重置暂存区（默认参数）

8. 比较文件差异：（若不指定文件名则显示所有有差异的文件的差异）

- `git diff <filename>` 比较当前目录中文件与暂存区中文件的差异
- `git diff <commit> <filename>` 和本地库的某一版本进行比较
- `git diff <commit> <commit> <filename>` 比较两个历史提交版本中某文件的差异

8. 分支操作

```
$ git branch -v                     # 查看所有分支
$ git branch <branch>               # 创建分支
$ git branch -d <branch>            # 删除分支
$ git branch -m <oldname> <newname> # 改名
$ git checkout <branch>             # 切换分支
$ git checkout -b <branch>          # 创建并切换分支
```

- 合并分支：站在被合并的分支上；

```
$ git checkout <branch>
$ git merge <branch>
```

- 解决冲突：

  删掉git的标记提醒，然后手动修改文件到满意的情况，然后git add，再git commit

gitignore

我们经常遇到这样的事情：有一些文件，或者一些更改，我们是不想进行提交的。

例如，我们在写一个 C 语言的程序，我们一般情况下并不需要提交它编译之后的文件。首先，编译之后的二进制文件体积较大，会增大我们的 Git 仓库；第二，我们仅通过源代码和指定的编译环境，便可以将该二进制文件进行复现，因此该文件是没有必要提交到 Git 仓库中的；第三，我们在看提交历史的时候，也会看到这个文件的更改信息，但程序员直接更改的是源代码文件，而这个二进制文件并不是程序员直接更改的文件，并不能代表更改历史。我们大部分情况下只需要提交能够复现此项目所需的最少文件即可。

再比如，一些编辑器会在目录中创建一些文件来记录编辑器的配置，而这可能只是程序员的个人偏好设置，或是面向该电脑本身的一种配置（例如该电脑安装的第三方库的路径），这些也是没必要提交到 Git 仓库的。

也鉴于此，我们不能直接愉快地使用 `git add .` 来添加当前目录下的所有更改了，而且，我们还很有可能因为误操作而将不想提交的文件也提交了上去。但 gitignore 便解决了这一问题。

Gitignore 的作用是设置文件的忽略规则，让 Git 根据该规则有选择地忽略文件的更改。使用 gitignore 的方式便是在目录中创建名为 `.gitignore` 的文本文件，在该文本文件里编写忽略规则。最简单的规则便是在该文件里写上某个文件的文件名，那么 Git 就会忽略该具有该文件名的文件：

```
main.out
```

那么，`.gitignore` 文件所在的目录，及其所有子目录下的 `main.out` 文件的更改均会被忽略。

如果要指定某一个文件被忽略，则需要指定它所在的路径，例如：

```
./main.out
./bin/main.out
```

即忽略当前目录下的 `main.out` 文件和当前目录的 `bin` 子目录下的 `main.out` 文件。

有时，我们还想忽略一个文件夹内的所有文件，方法是类似的（gitignore 使用 `#` 作为行注释）：

```
# 忽略当前目录及所有子目录下的 build 文件夹内的所有文件
build/
# 忽略当前目录下的 bin 文件夹内的所有文件
./bin/
# 一个例外——虽然 build 文件夹已经被忽略，但是 build 文件夹内的 publish.sh 文件并不忽略
!build/publish.sh
```

除此之外，还有很多其他的语法，例如使用 `*` 表示通配符（e.g. `*.bin` 忽略所有后缀名为 .bin 的文件），等等，在此不一一赘述。

当然，很多编辑器、集成开发环境，都具有很多的配置文件、缓存文件、代码生成中间文件，等等，这些都是我们不需要进行提交的。而对于一款工具来说，每个项目需要忽略的文件都是相同的，如果我们写项目还要花心思钻研我们使用的工具生成的文件中哪些需要忽略而那些不需要忽略，那么就太麻烦了！我们很自然地想到，很多工具都会有它自己的 gitignore 模板，这样我们就可以直接使用模板了！

下面这个网址是一个非常可靠的 gitignore 模板集合，我们可以把它们给我们自己的项目使用：

https://github.com/github/gitignore

例如，Visual Studio 和 Visual Studio Code、CMake、Unity、UE、Qt、JetBrains 等的模板均在里面可以找到。

gitattributes

Gitattributes 用于设置文件的属性。和 gitignore 类似，gitattributes 是在目录内创建 `.gitattributes` 文本文件，用于设置文件的属性，一般用于设置行尾转换、字符编码等等一系列属性。正常情况下，Git 默认的设置都能够使我们达到要求，也能够自动识别文件，不需要单独使用 gitattributes 进行指定。但是一些特殊情况下还是有必要的。在这里，我们仅对 gitattributes 做一个简短的介绍，想要深入了解的读者可以自行查阅相关资料进行学习，读者也可以跳过本部分。

> 举一个比较常见的例子。Windows 下文本文件的行尾是 CRLF（即一个字节 13 加上一个字节 10），而 Linux 下文本文件的行尾是 LF。如果无论什么系统均是逐个字节提交到 Git 仓库，则该 Git 仓库很难达到跨操作系统的目的。因此，默认情况下，Windows 平台上的 Git 会进行行尾转换，把文本文件的 CRLF 转换为 LF 提交到 Git 仓库，而把 Git 仓库调入到当前目录时则把 LF 转换回 CRLF；而 Linux 平台下则不进行行尾转换。这样以保证 Git 仓库内的文本文件文件均为 LF 行尾。但是存在一种情况，我们在一些意外情况下使得 Linux 系统上也出现了 CRLF 结尾的文件（例如 Linux 远程共享了 Windows 系统的文件夹），这样由于 Linux 系统的 Git 不进行行尾转换，便使得 Git 仓库中出现了 CRLF 行尾的文件，污染了 Git 仓库。这可以通过 `git config`，让 Linux 系统上的 Git 开启行尾转换而解决。但是，我们在多人合作开发的时候，很难要求所有人都进行设置。因此，使用 gitattributes 便可以让所有人的 Git 忽略自己的默认设置，而使用 `.gitattributes` 则可以让 Git 自动识别文本文件并进行行尾转换。
>
> Gitattributes 一般的语法为：`<filename> <attributes>`。要达到上述目的，只需要在 `.gitattributes` 中写一行：
>
> ```
> 1 * text=auto
> ```
>
> 其中，`*` 代表所有文件，`text=auto` 是让 Git 自动判断文件是二进制文件还是文本文件并进行行尾转换。

2. Git远程操作与GitHub

远程仓库

很多时候，我们的开发工作并不是一个人，而是很多人在一起协同开发；或者，我们将代码在网上公布，让社区大众一起来维护我们的代码。这个时候，我们便需要一个云端的平台，来帮助我们达到这一目的。

Git 作为一个流行的版本管理工具，很多云平台都是基于Git 来管理代码的。

例如：GitHub（https://github.com/）：世界公认的代码托管平台，也是我们本章的主角

GitLab（https://about.gitlab.com/）：GitHub 有力的竞争对手

Gitee（码云，https://gitee.com/）：中国大陆流行的代码托管平台

清华 GitLab（https://git.tsinghua.edu.cn/）：清华大学的代码托管平台，基于 GitLab 打造下面，我们以 GitHub 为例进行介绍，其他的平台是类似的。

GitHub是一个基于git的代码托管平台，可以注册一个账号。

Git本地仓库和GitHub仓库之间的传输是通过SSH加密的。

首先需要创建SSH key

```
ssh-keygen -t rsa -C "email"
```

一路回车即可

用户家目录下找到`.ssh/id_rsa.pub`，复制之后粘贴到GitHub->setting->sshkeys中，便可以将本地的仓库推送到GitHub上

Git远程操作

1. 将本地仓库与远程仓库绑定

```
$ git remote -v                # 查看当前是否有远程地址
$ git remote add <name> <url> # origin为是常用的别名，也可以用其他的
$ git remote rm <name>         # 删除远程仓库
```

2. 推送到远程仓库

```
$ git push origin <branch>  # 第一次push可以加参数-u，建立upstream，之后可以直接git push
# 如果不是基于GitHub远程库的最新版，不能push，必须先pull
```

3. clone远程仓库

```
$ git clone <url> # 可以是HTTPS，也可以是SSH；将仓库完整地下载到本地
```

4. 拉取

```
$ git pull <url> <branch> # fetch merge的结合
# git fetch origin main
# get merge
```

直接把别人的远程仓库clone下来之后，修改完之后不能直接push，需要让远程仓库的拥有则把你加成collaborators，否则只能提pr

GitHub 跨团队操作

- fork：在参与一项多人项目的开发时，我们通常要把该仓库复制一份到我们自己的 GitHub 账号的仓库中。我们的`git clone`通常也是从我们自己的远程仓库中 clone，并且所做的任何修改优先 push 到我们自己的仓库内。将该仓库复制到我们自己的账号下的过程即为“fork”
- clone：使用`git clone`将我们自己的仓库下载到本地
- 将公共的仓库中最新的更改拉取到本地（`git pull <url> <branch>`）

  习惯上我们把该仓库的链接命名为`upstream`。在命名后，便可以使用`git pull upstream`。如果我们是从默认分支pull的则可以省略分支名
- pull request：当自己的 GitHub 仓库已经更改了一个阶段性成果之后，可以从自己的分支向原来的仓库的某个分支提出一个“Pull Request（PR）”，即向该仓库发起一个请求，请求将自己的更改合并到该仓库

小结：不需要加入团队，fork + pull request。

clone本地的仓库之后就可以在工作区进行修改，然后commit，push到自己fork的远程仓库

在自己fork的远程仓库网页中去提PR

3. CI/CD

CI 持续集成（Continuous Integration）自动构建，测试和集成共享存储库中的代码更改

CD 持续交付（Continuous Delivery）自动将代码更改交付到生产就绪环境以供批准

CD 持续部署（Continuous Deployment）自动将代码更改直接部署到客

GitHub Actions

Github Actions是由Github创建的 CI/CD服务。 它的目的是使所有软件开发工作流程的自动化变得容易。 直接从GitHub构建，测试和部署代码。CI（持续集成）由很多操作组成，比如代码合并、运行测试、登录远程服务器，发布到第三方服务等等。GitHub 把这些操作就称为 actions。

很多操作在不同项目里面是类似的，完全可以共享。GitHub 允许开发者把每个操作写成独立的脚本文件，存放到代码仓库，使得其他开发者可以引用。

如果你需要某个 action，不必自己写复杂的脚本，直接引用他人写好的 action 即可，整个持续集成过程，就变成了一个 actions 的组合。这就是 GitHub Actions 最特别的地方。

GitHub 做了一个GitHub Marketplace，可以搜索到他人提交的 actions。另外，还有一个Awesome Actions的仓库，也可以找到不少 action。

基本概念

- workflow （工作流程）：持续集成一次运行的过程。
- job （任务）：一个 workflow 由一个或多个 job 构成，含义是一次持续集成的运行，可以完成多个任务。
- step（步骤）：每个 job 由多个 step 构成，一步步完成。
- action （动作）：每个 step 可以依次执行一个或多个命令（action）。

workflow

GitHub Actions 的配置文件叫做 workflow 文件，存放在代码仓库的`.github/workflows`目录中。

workflow文件采用的是YAML格式，后缀名为`.yml`

yaml语法简介

yaml是专门用来写配置文件的，十分简洁。

要点：

- 使用缩进表示层级关系
- 缩进不能使用tab，只能用空格
- 注释用#
- 多行字符串可以使用`|`保留换行符，也可以使用`>`折叠换行。

三种数据结构：

- 对象：键值对的集合，也叫映射mapping
- 数组：一组值
- 纯量scalars：单个不可分的值

```
# 对象
```

```yaml
key: value

key:
  child-key1: value1
  child-key2: value2

# 数组 -开头
- cat
- dog
- sheep # ['cat', 'dog', 'sheep'] json格式

-
 - A
 - B
 - C  # [['A', 'B', 'C']] json格式

# 纯量
# 字符串、布尔值、整数、浮点数、Null、时间、日期
```

workflow

- name：name 字段是 workflow 的名称。若忽略此字段，则默认会设置为 workflow 文件名。
- on：on 字段指定 workflow 的触发条件，通常是某些事件，比如示例中的触发事件是 push，即在代码 push 到仓库后被触发。on 字段也可以是事件的数组，多种事件触发，比如在 push 或 pull_request 时触发，也可指定运行的分支。
- jobs：jobs 表示要执行的一项或多项任务。每一项任务必须关联一个 ID (job_id)，比如示例中的 check-bats-version。job_id 里面的 name 字段是任务的名称。job_id 不能有空格，只能使用数字、英文字母和 - 或_符号，而 name 可以随意，若忽略 name 字段，则默认会设置为 job_id。当有多个任务时，可以指定任务的依赖关系，即运行顺序，否则是同时运行。
- runs-on：runs-on 字段指定任务运行所需要的虚拟服务器环境，是必填字段
- steps：steps 字段指定每个任务的运行步骤，可以包含一个或多个步骤。步骤开头使用 - 符号。每个步骤可以指定以下字段:
    - name：步骤名称。
    - uses：该步骤引用的action或 Docker 镜像。
    - run：该步骤运行的命令（例如bash命令）。
    - env：该步骤所需的环境变量。

  其中 uses 和 run 是必填字段，每个步骤只能有其一。同样名称也是可以忽略的。

示例

```yaml
name: learn-github-actions
on: [push]
jobs:
  check-bats-version:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '14'
      - run: npm install -g bats
      - run: bats -v
```

示例：clang-format

clang-format是 LLVM 开发的用于格式化 C/C++/Java/JavaScript/Objective-C/Objective-C++/Protobuf 等多种语言代码的工具，借助 clang-format 可以实现代码仓库的风格统一，提升开发效率。

clgg的仓库

---

4. 总结

在本节课程中，我们学习了版本管理系统的相关知识，掌握了 Git 和 GitHub 的基本用法。但是由于课时限制，很多有用的知识没有提及，例如：Git 更高级的用法，例如合并提交、submodule、定制 Git、Conventional commits……GitHub 仓库的权限管理、Issue、Project、tag、review、……略去上述内容不会对我们后续内容的学习产生太大影响，感兴趣的读者可以自行了解更多相关内容。

git详细教程参见Git - Book (git-scm.com)

5. 作业

- fork作业仓库https://github.com/fuyh20/Git-Homework到自己的仓库
- 在dev分支上按提示更改hw.txt，然后将更改后的文件push到自己仓库的dev分支
- 尝试提pr将自己仓库dev分支合并到作业仓库的main分支（注：一般不会这么做，但是为了作业www）