计算机系《软件工程》大作业辅导

Session 2

计算机系《软件工程》助教团队

清华大学计算机科学与技术系

2024 年 3 月 5 日

课前准备

- 安装 Git：https://docs.net9.org/basic/git/
- 阅读课程小作业文档：https://lab.cs.tsinghua.edu.cn/software-engineering/handout/
- 完成 Django 小作业环境配置：https://lab.cs.tsinghua.edu.cn/software-engineering/handout/django/env/

将项目启动后，打开浏览器，访问项目本地网站时显示的含有"Congratulations! You have successfully installed the requirements. Go ahead!" 的网页截图，上传网络学堂。（不计分）

回忆过去做大作业的过程

日常开发时，我们常常会遇到：

- 代码不知道改了哪里不能运行了，想要回退但 Ctrl-Z 回退步数不够；
- 多个人同时想对代码进行修改（并行开发），开发完后不得不一行一行比对差异；
- ...

动机：版本控制

一个完整的版本控制软件应帮助使用者找出：

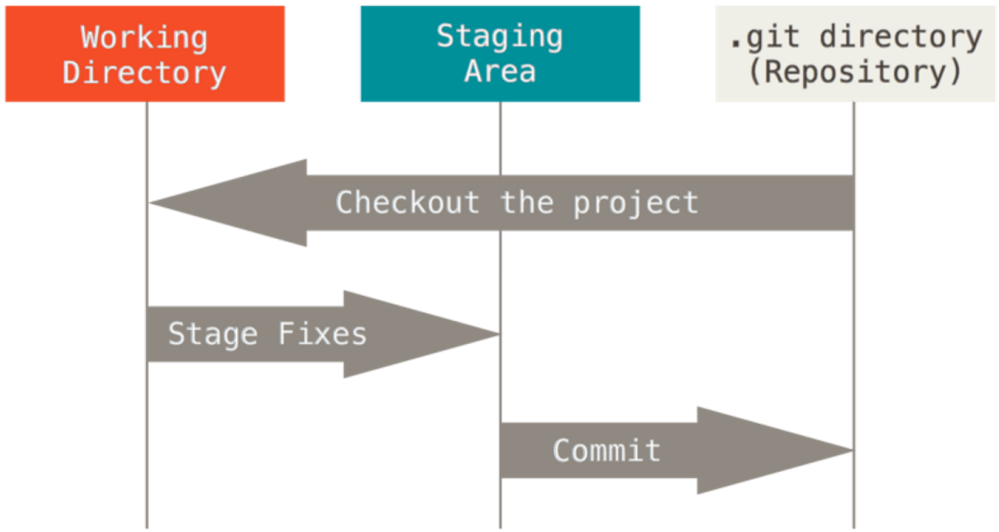
- 不同版本之间的差异
- 谁做出了这个修改
- 什么时候做出了这个修改
- 做出修改的人给出的修改理由

Git: 分布式版本控制器，允许我们将备份的代码以及记录完全独立在本地存储。

工作区、暂存区与分支

Git 的基本概念：

- 工作区：工作目录（你的代码所在的文件夹）；
- 暂存区：临时的缓存（做一个临时的标记）；
- 分支：将暂存区中的内容提交到分支（做一次完整的记录）。



基本操作

git init:

- 指定当前目录为工作区，git 初始化

git add <file_name>:

- 将文件添加到暂存区中
- git add *.txt: 将当前目录所有 txt 文件加入暂存区
- git add .: 将当前目录所有文件加入暂存区

git commit:

- 将暂存区中的内容提交到某个分支
- git commit -m <commit_message>: 提交的同时设置提交信息

分支的意义

当多人开发时，我们往往借助代码托管仓库（例如 Github）来与伙伴共享分支。
假如只有一条分支：

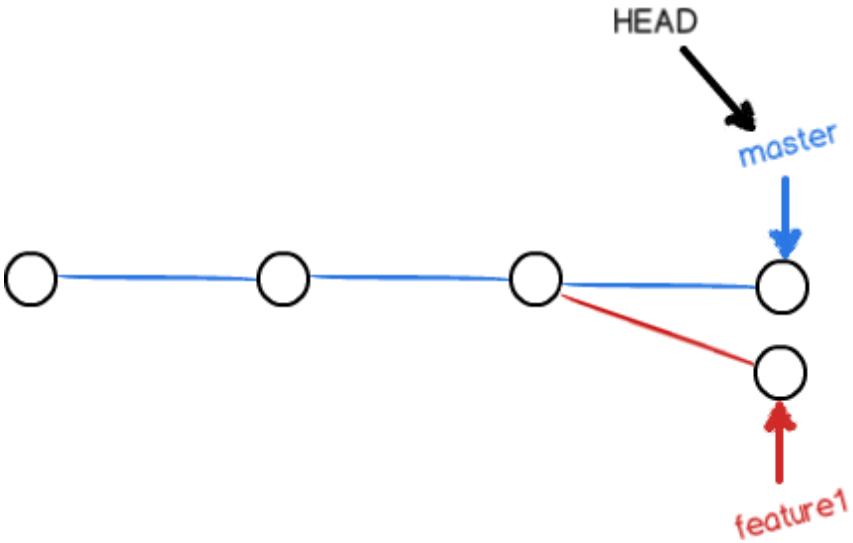
- 若一条新的功能开发需要 7 天时间完成，则
  - 全部写完后一次性提交到分支中，将面临代码丢失与版本无法追踪的风险；
  - 若分多次提交，则在完成这个功能前队友均无法工作。

因此我们需要多条分支，在稳定的分支的某次提交上创建一个自己的分支，之后便在自己的分支上进行开发，等到完全开发结束后再合并到团队合作的分支上。

commit, master, head

分支管理的一些概念：

- commit: 一次提交，每次 git commit 指令都会产生一个提交，该提交用一个独一无二的哈希值代表，伴随提交信息；
- master(main): 默认的主分支（实际是一个指针，指向当前分支的最新的 commit）；
- head: 指向工作区当前位于的 commit 的指针，通常指向当前所在分支的最新的 commit。



开发合作模式

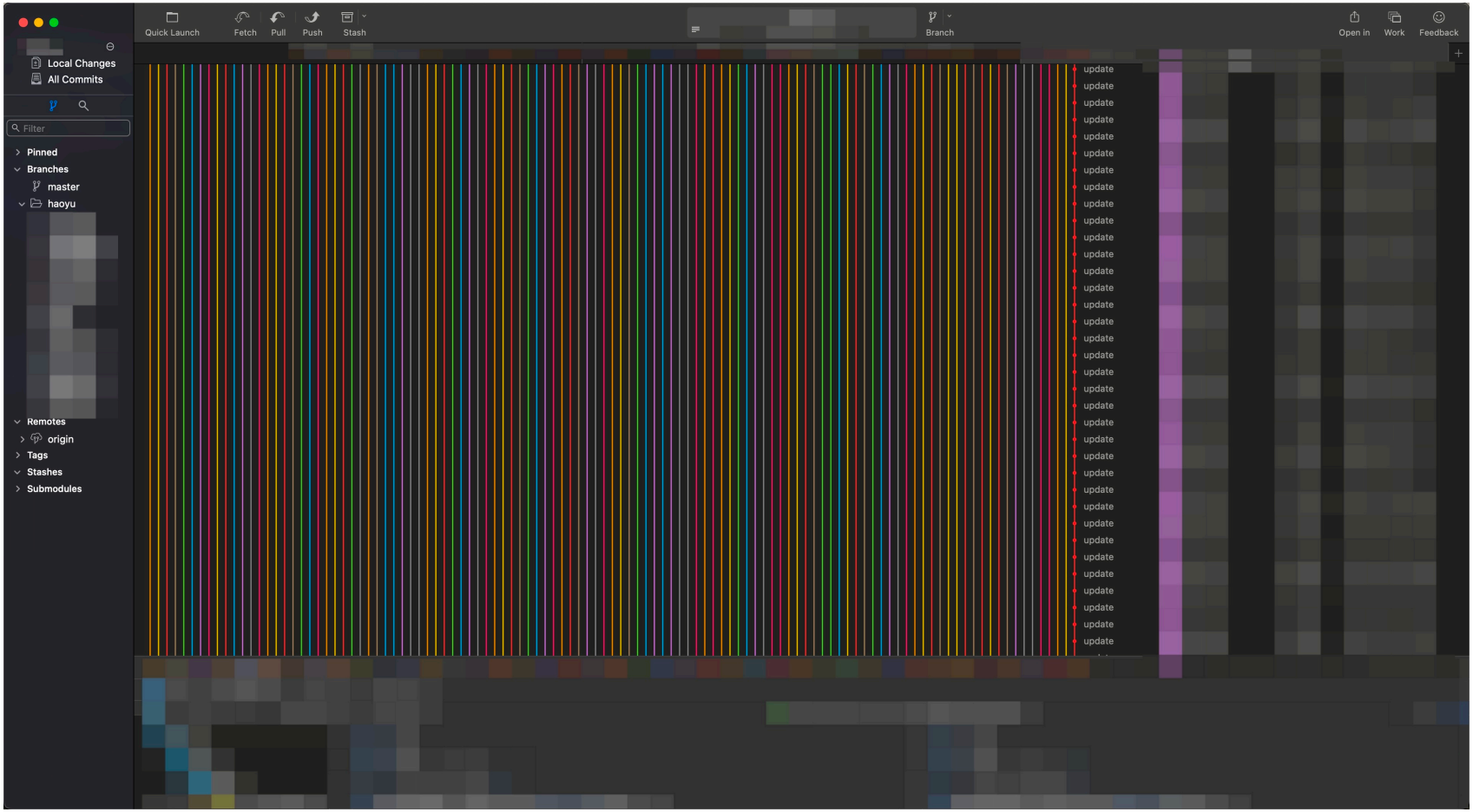
一种比较有效的开发合作模式举例：

- master 分支为稳定分支，可稳定部署，理论上不应存在 bug；
- dev 分支为开发分支，可稳定部署，可能存在 bug；
- 其他个人开发分支命名规则为：feat-xxx/fix-xxx，表示该分支应完成的功能；
- commit 信息应当简短有效，具体要求可参考 https://www.conventionalcommits.org/en/v1.0.0/。

开发合作模式


master
dev
:twisted_rightwards_arrows: merge: merge branch 'dev' into 'master'
:twisted_rightwards_arrows: merge: merge branch 'feat-img_download' into dev asyNc-web#95
:see_no_evil: docs: avoid tracking images/ asyNc-web#95
:bug: fix: keep trying to reconnect db when disconnected asyNc-web#99
:sparkles: feat: download images crawled for frontend use asyNc-web#95
:twisted_rightwards_arrows: merge: merge branch 'feat-homenews_table' into dev asyNc-web#93
:rocket: chore: change execution script to expand crawl scope asyNc-web#93
:bug: fix: ignore full crawls that don't meet date specification asyNc-web#94
:sparkles: feat: insert item into different table by spider asyNc-web#93
:seedling: feat: dynamically allocate ports to enable multiple crawlers asyNc-web#93
Merge branch 'dev' into 'master'
:twisted_rightwards_arrows: merge: merge branch 'fix-es' into dev
:sparkles: fix: add tags in articletype asyNc-web#78
:twisted_rightwards_arrows: merge: merge branch 'dev' into 'master'
:twisted_rightwards_arrows: merge: merge branch 'fix-es' into dev
:sparkles: fix: remove useless member in articletype asyNc-web#78

大型项目


Quick Launch
Fetch
Pull
Push
Stash
Branch
Open in
Work
Feedback
Local Changes
All Commits
Filter
Pinned
Branches
master
haoyu
Remotes
origin
Tags
Stashes
Submodules
update

基本操作

创建与切换分支：

- git checkout [-b] <branch_name>

合并分支：

- git merge <target_branch>
- merge mode:
  - Fast-Forward
  - Three-Way Merge
  - Squash Merge

操作远程仓库：

- git push
- git pull
- git fetch

了解更多

如果你希望了解更多知识：

- Git 官方文档：https://git-scm.com/
- 廖雪峰教程：https://www.liaoxuefeng.com/wiki/896043488029600
- learngit 在练习中学习 git：https://learngitbranching.js.org/?locale=zh_CN

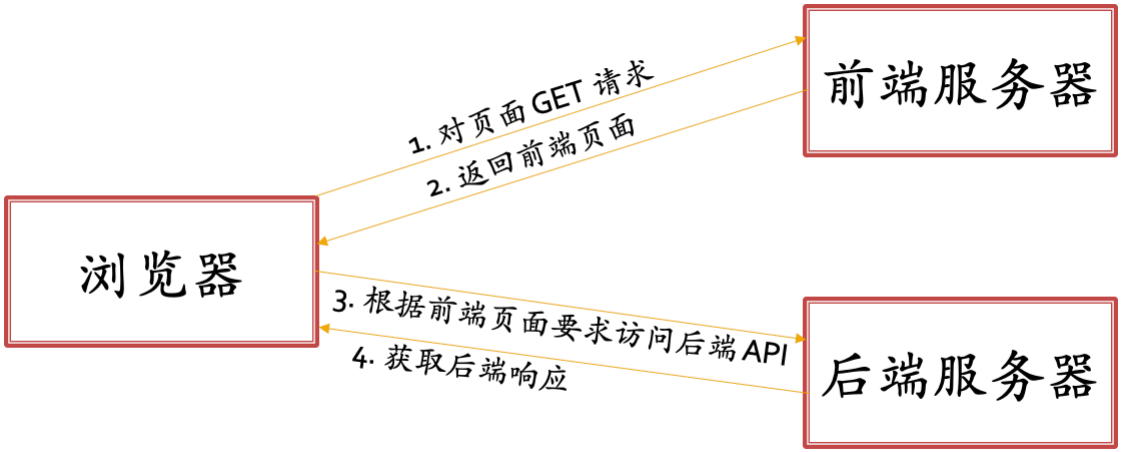
一些名词

前端三件套

- HTML。超文本标记语言，Hyper Text Markup Language。定义了网页上元素的框架与结构
- CSS。层叠样式表，Cascading Style Sheet。定义了网页上各个元素的样式
- JS。JavaScript。定义了网页如何响应用户操作，现在该语言不仅可以运行在客户端，也可以运行在服务器端

前后端分离

现今常见的前后端分离结构如下所示：



前端服务器所返回的是一个 HTML 模板，一般会缺失重要数据，浏览器从后端服务器获取实际数据后才能组装出完整的网页。

HTTP 协议

HTTP 请求的结构为：

- 请求行：包含 HTTP 方法（GET、POST 等）、协议版本、请求路径
- 请求头：若干键值对，一般会携带身份信息等
- 请求体：请求所携带的数据

HTTP 应答的结构为：

- 状态行：包含 HTTP 状态码（200、404 等）、协议版本
- 应答头：若干键值对，描述应答的基本信息
- 应答体：应答的数据本体

HTTP 协议

打开浏览器的控制台中“网络”部分，可以看到浏览器发出的所有 HTTP 请求，以下是 Safari 浏览器上的一例：

请求

```
GET /adsid/google/ui
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/15.5 Safari/605.1.15
Accept: image/webp,image/png,image/svg+xml,image/*;q=0.8,video/*;q=0.8,*/*;q=0.5
Accept-Language: zh-CN,zh-Hans;q=0.9
Referer: https://www.google.com.hk/
Accept-Encoding: gzip, deflate, br
```

重定向响应

```
302
Cross-Origin-Resource-Policy: cross-origin
Timing-Allow-Origin: *
Cache-Control: private, max-age=900
Location: https://adservice.google.com.hk/adsid/google/si?gadsid=AORoGNRZqDgMRKotiRQAQtFrTS2FOQmOyRlCpWcEt2XqOwvkU8jXVrpEldH5rA
Date: Sat, 18 Feb 2023 17:51:20 GMT
```

API 约定

具体按照何种数据结构发送数据，后端应当如何响应指定请求，如何设定状态码等约定构成了前后端之间交流的 API 文档。

URL `/boards`

该 API 用于操作整体的游戏记录列表，包括获取全部游戏记录以及新增游戏记录。

该 API 仅接受以 GET 与 POST 方法请求。以其他方法请求均应当设置状态码为 405 Method Not Allowed，错误响应格式为：

```
{
    "code": -3,
    "info": "Bad method"
}
```

GET

使用 GET 方法请求该 API 即表示请求当前存储的全部游戏记录，以创建时间为基准倒序排列，越晚创建的记录出现在数组索引越小的位置。

请求体 成功响应 错误响应

所有错误响应的格式均为：

```
{
    "code": *,
    "info": "[Some message]"
}
```

- 若读取数据中途抛出错误，错误响应的状态码为 500 Internal Server Error，`code` 字段为 `-4`，`info` 字段尽量携带错误信息（CI 不评测该错误响应）

前端不可信任原则

如果前端服务器返回的 HTML 中嵌入了恶意代码，那么很有可能产生严重后果，如窃取个人信息或冒充用户发送请求。

跨域请求限制

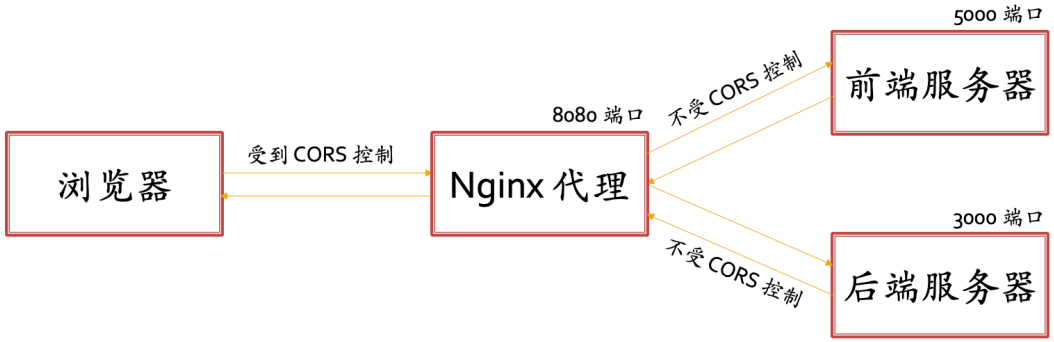
为了防止这类问题出现，现代浏览器会限制所有跨域请求（同域名同端口的称为同域，从一个域发往另外一个域的请求就是跨域请求），例如 CORS(Cross Origin Resource Sharing) 机制。
CORS 机制要求，浏览器在发送跨域请求前都会先发送预检请求，通过该预检请求后浏览器才会实际发送跨域请求。

前端不可信任原则

应对方式

面对浏览器对跨域请求的限制，有以下几种方案：

- 老实遵守 CORS 机制，开发后端的时候也支持 CORS 机制
- 使用静态资源请求方法回避跨域检查，如 JSONP 方法
- 使用反向代理使前后端服务器暴露相同端口（大作业项目中推荐）



前后端交互中的鉴权问题

Cookie 与基于令牌的身份认证

最初的 Web 应用是用户无关的，即无论是谁，只要在相同时间访问了相同的 Web 应用，其向后端服务器请求到的数据都应当一致。然而当下更多的 Web 应用是用户相关的，即使是新闻网站，也需要根据用户的偏好推送相应的新闻条目。这一类 Web 应用就面临一个重要问题，即后端服务器如何分辨当前是哪位用户在请求数据。

- Cookie 是存储在浏览器之中的一段信息
- 我们小作业采用的令牌——JWT 令牌

什么是 JWT？

- JWT（JSON Web Token）是一种开放标准（RFC 7519），用于安全地在两方之间传递信息作为 JSON 对象。
- 它可以用于身份验证和信息交换，提供了一种简洁的、自包含的方法用于在各方之间传递信息。

JWT 的结构

- JWT 由三部分组成，使用‘.‘分割：头部（Header）、载荷（Payload）和签名（Signature）。
- 结构示例：header.payload.signature

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKx
wRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

头部（Header）

- 头部通常由两部分组成：令牌类型（‘typ‘）和所使用的加密算法（‘alg‘）。
- 例如，使用 HMAC SHA256 算法的 JWT 头部：

```
{
    "alg": "HS256",
    "typ": "JWT"
}
```

- 编码后的头部示例：
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

载荷（Payload）

- 载荷包含 JWT 的主要信息，比如用户的标识、令牌的签发时间等。
- 它可以包含多个预定义的字段，也可以包含自定义的字段。

```
{
    "sub": "1234567890",
    "name": "John Doe",
    "iat": 1516239022
}
```

- 编码后的载荷示例：
  eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ

签名（Signature）

- 签名用于验证消息的发送者身份并确保消息在传输过程中未被篡改。
- 签名是通过对头部和载荷进行加密算法处理，并使用秘钥来生成。

```
signature = HS256(
    "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9" +
    "." +
    "eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyf(
    "your-256-bit-secret",
)  # SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

- 最终的 JWT 示例：`header.payload.signature`

JWT 的使用场景

- 身份验证：JWT 是进行身份验证的一种常见方法。当用户登录后，每个后续请求将包括 JWT，允许用户访问该令牌允许的路由、服务和资源。
- 信息交换：JWT 是安全地在各方之间传输信息的好方法，因为可以验证每一部分的签名。

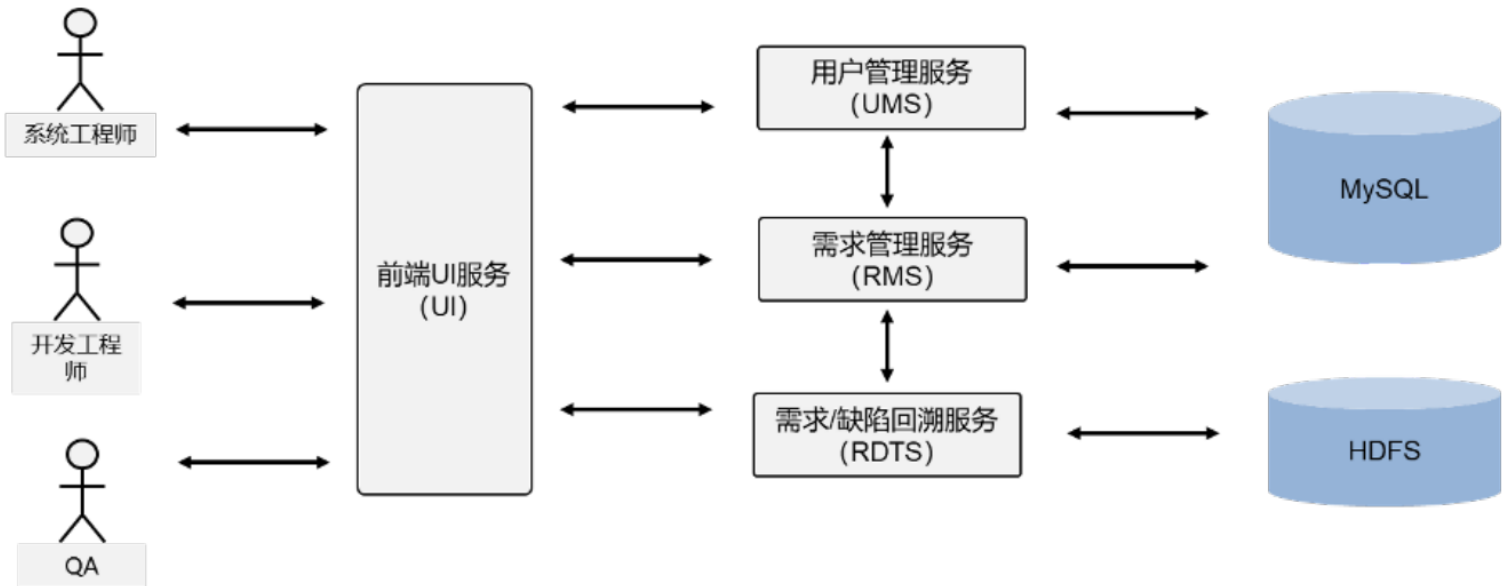
总结

- JWT 提供了一种简洁的方式来安全地在客户端和服务器之间传递信息。
- 它由头部、载荷和签名组成，可以用于身份验证和信息交换。

后端的作用

总述

- 与前端进行数据交互
- 操作与管理持久性存储



后端的作用

与前端进行数据交互：请求类型

- 面对前端的查询请求（**GET** 方法）要返回相应的数据供前端渲染，比如用户点开网页后首页要进行视频推荐，用户可以查看自己的个人信息
- 面对前端的修改请求（POST 方法、PUT 方法、DELETE 方法，分别对应创建、修改、删除）要在检查后做出响应，比如用户可以上传视频、发表评论、修改个人信息、删除视频等等

后端的作用

与前端进行数据交互：请求体格式

- 后端与前端的交互通过网络请求的格式进行，而最常用的数据规范便是 JSON 格式（JavaScript Object Notation）
- JSON 将数据组织成键值对的形式，其值涵盖的类型包括：object, array, number, string, Boolean (true / false) 和 null
- 很多高级语言都支持 JSON 格式与自身自带的 HashMap 容器的高效转换，比如在 Python 中，json 库可以将 JSON 字符串与 dict 进行转换

后端的作用

操作与管理持久化存储

- 操作与管理持久化存储是由于我们需要将用户的数据高效地组织起来，以供后续查询与修改
- 这里的“持久化存储”是一个相对广泛的概念
  - 对于简单的开发需求，文件系统也可以满足这个“数据库”的要求，用户的数据可以直接存储成文件的形式
  - 面对更大体量的数据，面对更快的查询与修改需求，各种各样的数据库软件是不可或缺的
  - 在小作业中我们采用本地存储 SQLite3，在大作业中你可能会用到 MySQL, MongoDB, PostgreSQL 等等

后端的作用

操作与管理持久化存储：ORM 机制

- 关系型数据库是数据库的一种，一个数据库应用中可以创建若干个数据库，每个数据库又可以包含若干个数据表
- 可以将数据表与一个实体类联系起来，将每一列看做是这个实体类的一个具体的成员变量，每一行看成是这个类的一个实例
- 将每张数据表看做是相同类的实体构成的集合，这种观点就是 ORM 机制

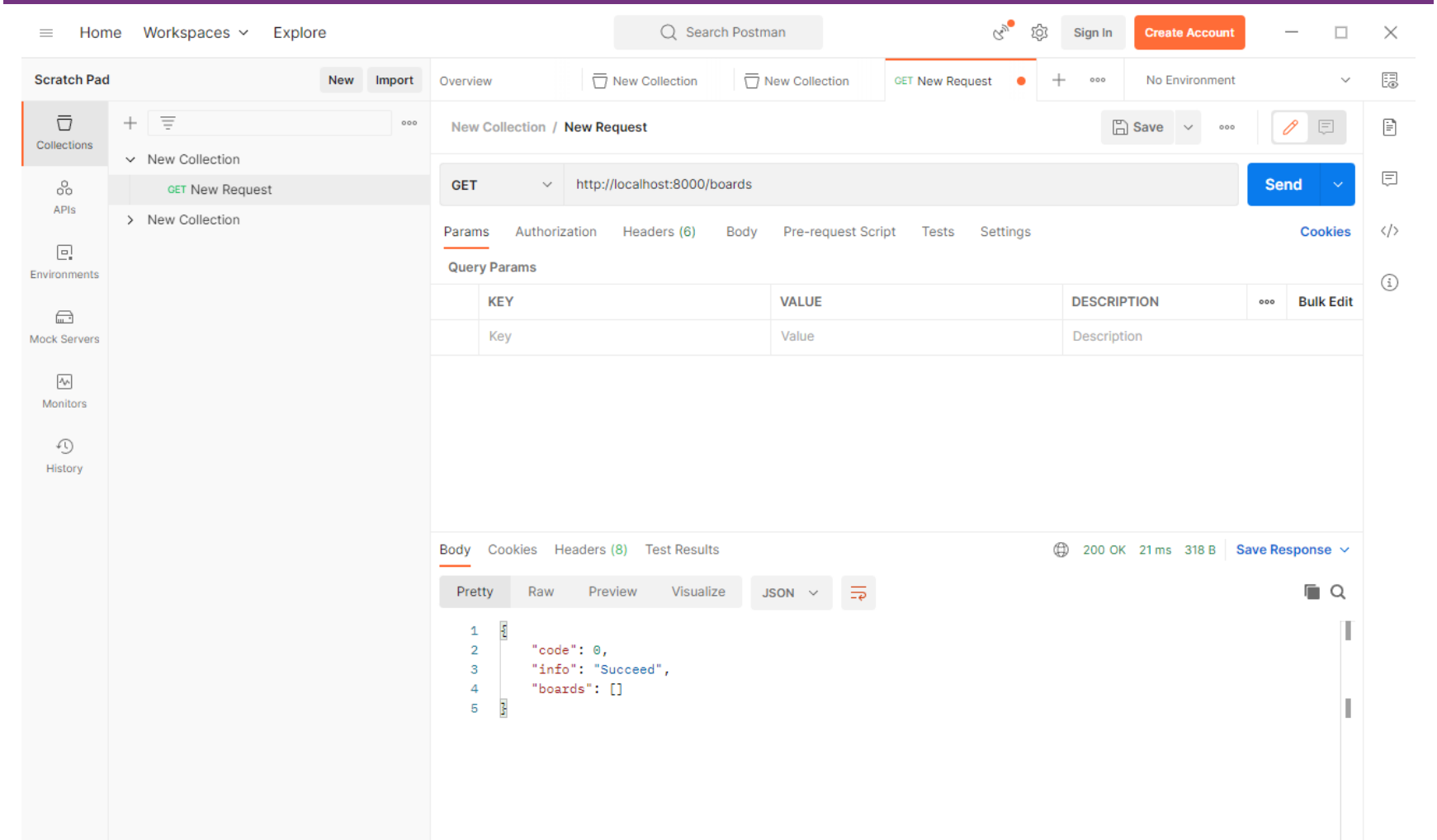
后端的调试

人工调试：Postman

- Postman 是一种常用的 API 开发工具，它可以帮助开发人员快速创建、测试、调试和文档化 API
- Postman 提供了一个直观的用户界面，使得开发人员可以轻松地构建 HTTP 请求，并查看和分析响应
- 它支持各种 HTTP 方法，包括 GET、POST、PUT、DELETE 等
- 它支持各种数据格式，例如 JSON、XML 等
- 链接：`https://www.postman.com/`

后端的调试

人工调试：Postman



后端的调试

自动化测试：单元测试

- 测试工程师撰写相应的测试代码，涉及到一般情况与尽可能多的 Corner Cases
- 一方面，这能帮助开发工程师有效开发；另一方面，在一个大型项目中可能会出现改动一部分代码导致另一部分代码的原有功能失效的情况，而通过固定的测试程序与测试用例则可以保证只有稳定的版本才会最终呈现给用户

后端的调试

自动化测试：测试流程

- 测试工程师将开发工程师撰写的后端逻辑视为黑盒
- 测试工程师可以直接模拟前端请求，读取与修改数据库中的内容
- 测试工程师可以通过一系列断言验证程序的正确性

测试工程师就像是在设计 OJ 测例，断言后端系统在面对输入（用户请求）时，应该产生什么样的输出（对用户请求的响应、对数据库的修改）。若断言失败，则说明开发工程师提交的本次修改存在功能性问题，需要进行修复。

后端框架

后端框架的作用

一个成熟的后端框架往往具备以下解耦合的功能：

- 路由：将前端请求的不同路径映射到不同的处理函数的方法
- 模型：与数据库进行交互的方法，可以使用 ORM 机制撰写成面向对象风格
- 视图：前端请求的处理函数，接受用户请求并返回响应
- 单元测试：如何为测试工程师提供模拟请求、读写数据库、断言的操作

这里的“路由”、“模型”、“视图”都是 Django 中的概念，在其他的后端框架中它们不一定被如此称呼，但你可以在学习其他后端框架时找到这些概念的影子。

我们不是专业开发人员...

Docs: https://docs.djangoproject.com/en/4.1/

开发环境配置

https://lab.cs.tsinghua.edu.cn/software-engineering/handout/django/env/

开发环境配置

小作业中我们采用以下环境：

- Python==3.9（你可以使用 conda 管理虚拟环境）
- django==4.1.3

简易指令说明

如果你可以理解下面每条指令在做什么，你可以直接运行以下指令：

- conda create -n django_hw python=3.9 -y
- conda activate django_hw
- pip install -i https://pypi.tuna.tsinghua.edu.cn/simple -r requirements.txt

项目与应用

项目与应用

- 项目文件夹是一个 Django 项目的顶层模块，其中含有 settings.py
- 一个 Django 项目中可以同时存在多个应用，一个应用是具有完成某个独立功能作用的模块

```
.
├── DjangoHW  # 我们的项目名为 DjangoHW
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── README.md
├── board  # 我们新建了一个应用叫做 board
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   │   ├── 0001_initial.py
│   │   ├── 0002_remove_board_deleted_remove_user_deleted.py
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   ├── urls.py
│   └── views.py
├── db.sqlite3  # 本地数据库存储
├── manage.py
├── requirements.txt
└── utils  # 撰写的功能函数，可以放在应用中也可以放在根目录下，保证引用
    ├── utils_request.py
    ├── utils_require.py
    └── utils_time.py
```

路由

- 首先，我们来解决后端收到请求时，后端会将请求交给哪个应用的哪个视图函数处理的问题。和这个功能有关的文件主要为项目文件夹和应用文件夹下的 urls.py。
- 假如我们的后端部署在 my-backend.com，我们在访问 my-backend.com/board/restart 时，后端会首先在项目文件夹下的 urls.py 中以 board/restart 开始搜索。假设其配置为：

```
1  from django.urls import path, include
2
3  urlpatterns = [
4      path('board/', include("board.urls")),
5  ]
```

- 我们会匹配掉字符串 board/，然后将剩下的请求 restart 交给 board/urls.py 处理

路由

- 假设子应用的 urls.py 配置为：

```python
from django.urls import path, include
import board.views as views

urlpatterns = [
    path('restart', views.restart_board),
]
```

- 这时剩余请求 restart 匹配到第一条规则后，交由 board/views.py 中的 restart_board 函数进行处理，即后端会帮助我们调用这个函数，并把请求体（和请求有关的信息，包括请求方法、请求数据等等）作为参数传给这个函数。

模型

- 在 Django 中，模型用于数据库中数据表的结构设计以及数据表的元数据（如主键、外键、索引等）管理。我们使用 Django 提供的 ORM 机制来进行对数据表和数据表列属性的管理。具体来说，我们只需要在 <app>/models.py 中定义一个类继承 django.db.models.Model 即可。

```python
from django.db import models

class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField()

class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

- 在你修改完应用的 models.py 之后，你应该使用如下命令去生成修改数据表结构与属性的语句：
- python3 manage.py makemigrations <app_name>

视图函数

- 视图函数是后端逻辑的主入口，其接受经过路由之后的 HttpRequest 类型的请求作为参数，并返回一个 HttpResponse 类型的对象作为响应。我们可以在 <app>/views.py 中定义一个应用所具有的视图函数。在这里举一个留言板应用“获取与创建留言”的视图函数作为例子。

```python
def message(request):  # 这里 request 是 HttpRequest 类型的对象
    # 功能函数，快速创建具有特定状态码的响应
    def gen_response(code: int, data: str):
        return JsonResponse({
            'code': code,
            'data': data
        }, status=code)  # JsonResponse 是 HttpResponse 的子类
                         # 可以传入一个 dict 转换成 JSON 响应

    if request.method == 'GET':
        limit = request.GET.get('limit', default='100')
        offset = request.GET.get('offset', default='0')
        if not limit.isdigit():
            return gen_response(400, f'{limit} is not a number')
        if not offset.isdigit():
            return gen_response(400, f'{offset} is not a number')

        return gen_response(200, [
                {
                    'title': msg.title,
                    'content': msg.content,
                    'user': msg.user.name,
                    'timestamp': int(msg.pub_date.timestamp())
                }
                for msg in Message.objects.all()\
                            .order_by('-pub_date')\
                            [int(offset):int(offset)+int(limit)]
            ])

    elif request.method == 'POST':
        # 从 cookie 中获得 user 的名字，如果 user 不存在则新建一个
        # 如果 cookie 中没有 user 则使用 "Unknown" 作为默认用户名
```

单元测试

- 在 Django 中，测试工程师会将开发工程师所编写的路由与视图视为黑盒，通过 django.test.TestCase 类与 django.test.Client 类来模拟前端与开发工程师所撰写的后端交互，并通过其提供的断言函数来断言响应所应该具有的属性或是数据库应被如何修改。

```python
from django.test import TestCase, Client
from .models import Message, User  # Defined in models.py

class MessageModelTests(TestCase):
    def setUp(self):  # Preparation
        alice = User.objects.create(name="Alice")
        bob = User.objects.create(name="Bob")
        Message.objects.create(user=alice,
                               title="Hi",
                               content="Hello World!")
        Message.objects.create(user=bob,
                               title="This is a title",
                               content="This is my content")


    # 测试 POST 方法
    def test_add_new_message(self):
        # 模拟前端请求
        title, content = "Title", "Message"
        user = "student"
        payload = {
            'title': title,
            'content': content,
        }
        self.client.cookies['user'] = user

        response = self.client.post('/api/message',
                                    data=payload,
                                    content_type="application/json")

        # 断言响应的属性
        self.assertJSONEqual(response.content,
                             {
                                 'code': 201,
                                 'data': "Message received successfully"}
                             )
        # 断言数据库的属性
        self.assertTrue(User.objects.filter(name=user).exists())
        self.assertTrue(Message.objects.filter(
            title=title, content=content).exists())

```

Thanks!

Questions?