第二章作业-2020211502-王小龙

思考题-1.

a.

Fielding 认为，一套理想的、完全满足 REST 风格的系统应该满足以下六大原则:

1. 服务端与客户端分离（Client-Server）
   将用户界面所关注的逻辑和数据存储所关注的逻辑分离开来，有助于提高用户界面的跨平台的可移植性，这一点正越来越受到广大开发者所认可，以前完全基于服务端控制和渲染（如 JSF 这类）框架实际用户已甚少，而在服务端进行界面控制（Controller），通过服务端或者客户端的模版渲染引擎来进行界面渲染的框架（如 Struts、SpringMVC 这类）也受到了颇大的冲击。这一点主要推动力量与 REST 可能关系并不大，前端技术（从 ES 规范，到语言实现，到前端框架等）的近年来的高速发展，使得前端表达能力大幅度加强才是真正的幕后推手。由于前端的日渐强势，现在还流行起由前端代码反过来驱动服务端进行渲染的 SSR（Server-Side Rendering）技术，在 Serverless、SEO 等场景中已经占领了一块领地。

2. 无状态（Stateless）
   无状态是 REST 的一条核心原则，部分开发者在做服务接口规划时，觉得 REST 风格的服务怎么设计都感觉别扭，很有可能的一种原因是在服务端持有着比较重的状态。REST 希望服务器不要去负责维护状态，每一次从客户端发送的请求中，应包括所有的必要的上下文信息，会话信息也由客户端负责保存维护，服务端依据客户端传递的状态来执行业务处理逻辑，驱动整个应用的状态变迁。客户端承担状态维护职责以后，会产生一些新的问题，譬如身份认证、授权等可信问题，它们都应有针对性的解决方案（这部分内容可参见“安全架构”的内容）。
   但必须承认的现状是，目前大多数的系统都达不到这个要求，往往越复杂、越大型的系统越是如此。服务端无状态可以在分布式计算中获得非常高价值的好处，但大型系统的上下文状态数量完全可能膨胀到让客户端在每次请求时提供变得不切实际的程度，在服务端的内存、会话、数据库或者缓存等地方持有一定的状态成为一种是事实上存在，并将长期存在、被广泛使用的主流的方案。

3. 可缓存（Cacheability）
   无状态服务虽然提升了系统的可见性、可靠性和可伸缩性，但降低了系统的网络性。“降低网络性”的通俗解释是某个功能如果使用有状态的设计只需要一次（或少量）请求就能完成，使用无状态的设计则可能会需要多次请求，或者在请求中带有额外冗余的信息。为了缓解这个矛盾，REST 希望软件系统能够如同万维网一样，允许客户端和中间的通讯传递者（譬如代理）将部分服务端的应答缓存起来。当然，为了缓存能够正确地运作，服务端的应答中必须明确地或者间接地表明本身是否可以进行缓存、可以缓存多长时间，以避免客户端在将来进行请求的时候得到过时的数据。运作良好的缓存机制可以减少客户端、服务器之间的交互，甚至有些场景中可以完全避免交互，这就进一步提高了性能。

4. 分层系统（Layered System）
   这里所指的并不是表示层、服务层、持久层这种意义上的分层。而是指客户端一般不需要知道是否直接连接到了最终的服务器，抑或连接到路径上的中间服务器。中间服务器可以通过负载均衡和共享缓存的机制提高系统的可扩展性，这样也便于缓存、伸缩和安全策略的部署。该原则的典型的应用是内容分发网络（Content Distribution Network，CDN）。如果你是通过网站浏览到这篇文章的话，你所发出的请求一般（假设你在中国国境内的话）并不是直接访问位于 GitHub Pages 的源服务器，而是访问了位于国内的 CDN 服务器，但作为用户，你完全不需要感知到这一点。我们将在“透明多级分流系统”中讨论如何构建自动的、可缓存的分层系统。
5. 统一接口（Uniform Interface）
   这是 REST 的另一条核心原则，REST 希望开发者面向资源编程，希望软件系统设计的重点放在抽象系统该有哪些资源上，而不是抽象系统该有哪些行为（服务）上。这条原则你可以类比计算机中对文件管理的操作来理解，管理文件可能会进行创建、修改、删除、移动等操作，这些操作数量是可数的，而且对所有文件都是固定的、统一的。如果面向资源来设计系统，同样会具有类似的操作特征，由于 REST 并没有设计新的协议，所以这些操作都借用了 HTTP 协议中固有的操作命令来完成。
   统一接口也是 REST 最容易陷入争论的地方，基于网络的软件系统，到底是面向资源更好，还是面向服务更合适，这事情哪怕是很长时间里都不会有个定论，也许永远都没有。但是，已经有一个基本清晰的结论是：面向资源编程的抽象程度通常更高。抽象程度高意味着坏处是往往距离人类的思维方式更远，而好处是往往通用程度会更好。用这样的语言去诠释 REST，大概本身就挺抽象的，笔者还是举个例子来说明：譬如，几乎每个系统都有的登录和注销功能，如果你理解成登录对应于 login()服务，注销对应于 logout()服务这样两个独立服务，这是“符合人类思维”的；如果你理解成登录是 PUT Session，注销是 DELETE Session，这样你只需要设计一种“Session 资源”即可满足需求，甚至以后对 Session 的其他需求，如查询登陆用户的信息，就是 GET Session 而已，其他操作如修改用户信息等都可以被这同一套设计囊括在内，这便是“抽象程度更高”带来的好处。想要在架构设计中合理恰当地利用统一接口，Fielding 建议系统应能做到每次请求中都包含资源的 ID，所有操作均通过资源 ID 来进行；建议每个资源都应该是自描述的消息；建议通过超文本来驱动应用状态的转移。
6. 按需代码（Code-On-Demand）
   按需代码被 Fielding 列为一条可选原则。它是指任何按照客户端（譬如浏览器）的请求，将可执行的软件程序从服务器发送到客户端的技术，按需代码赋予了客户端无需事先知道所有来自服务端的信息应该如何处理、如何运行的宽容度。举个具体例子，以前的Java Applet技术，今天的WebAssembly等都属于典型的按需代码，蕴含着具体执行逻辑的代码是存放在服务端，只有当客户端请求了某个 Java Applet 之后，代码才会被传输并在客户端机器中运行，结束后通常也会随即在客户端中被销毁掉。将按需代码列为可选原则的原因并非是它特别难以达到，而更多是出于必要性和性价比的实际考虑。

b.

达到第三级的好处有：

- 更低的耦合度和更好的维护性，因为客户端不需要事先知道服务器提供了哪些服务和接口。
- 更高的可扩展性和灵活性，因为客户端可以根据服务器返回的超媒体动态地发现和调用相关的服务。

c.

不足与争议：

<1>面向资源的编程思想只适合做 CRUD，面向过程、面向对象编程才能处理真正复杂的业务逻辑

<2>REST 与 HTTP 完全绑定，不适合应用于要求高性能传输的场景中

<3>REST 不利于事务支持

<4>REST 没有传输可靠性支持

<5>REST 缺乏对资源进行“部分”和“批量”的处理能力

以上不足与争议有些是应该认同的，有些是可以争论的，但需要承认的是REST风格的接口是一种有价值的设计思想，但并不适用于所有场景，按需使用合适的方式才是正确的做法。

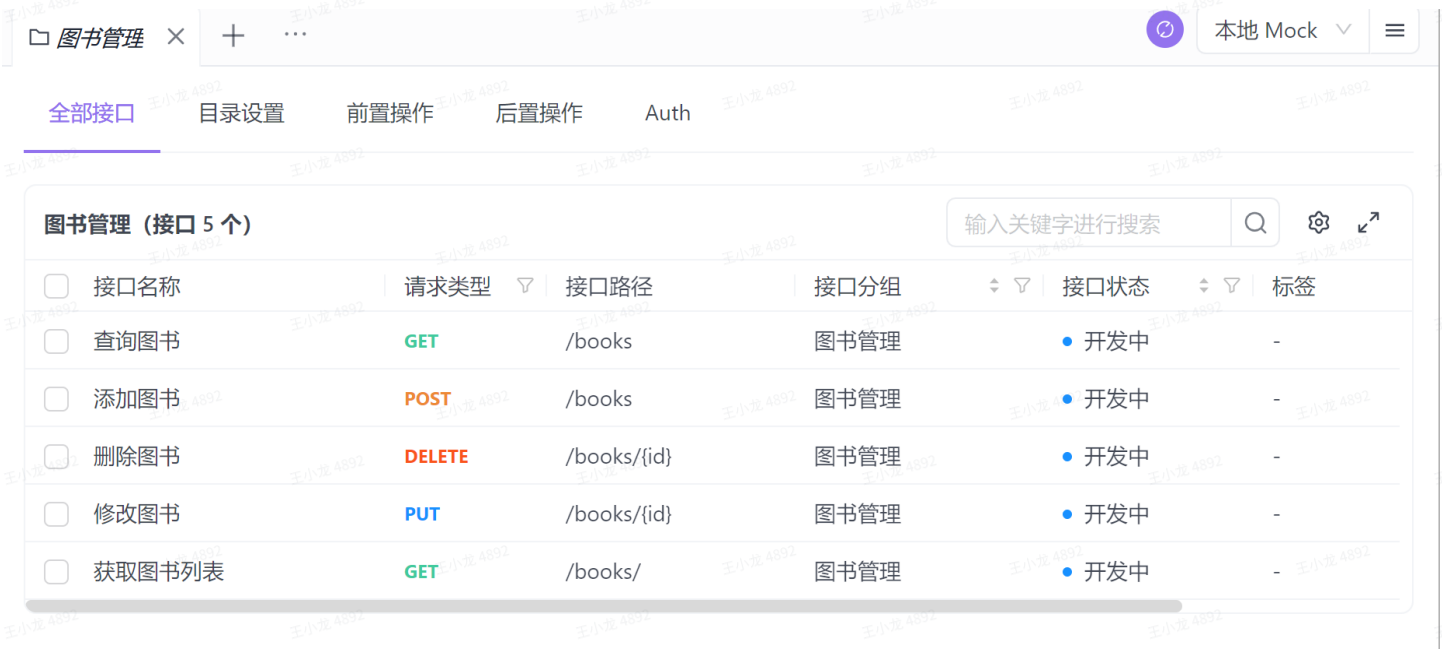
实验题

代码仓库地址（已添加成员）：

https://gitee.com/lhfhlhfl/web-api-demo.git

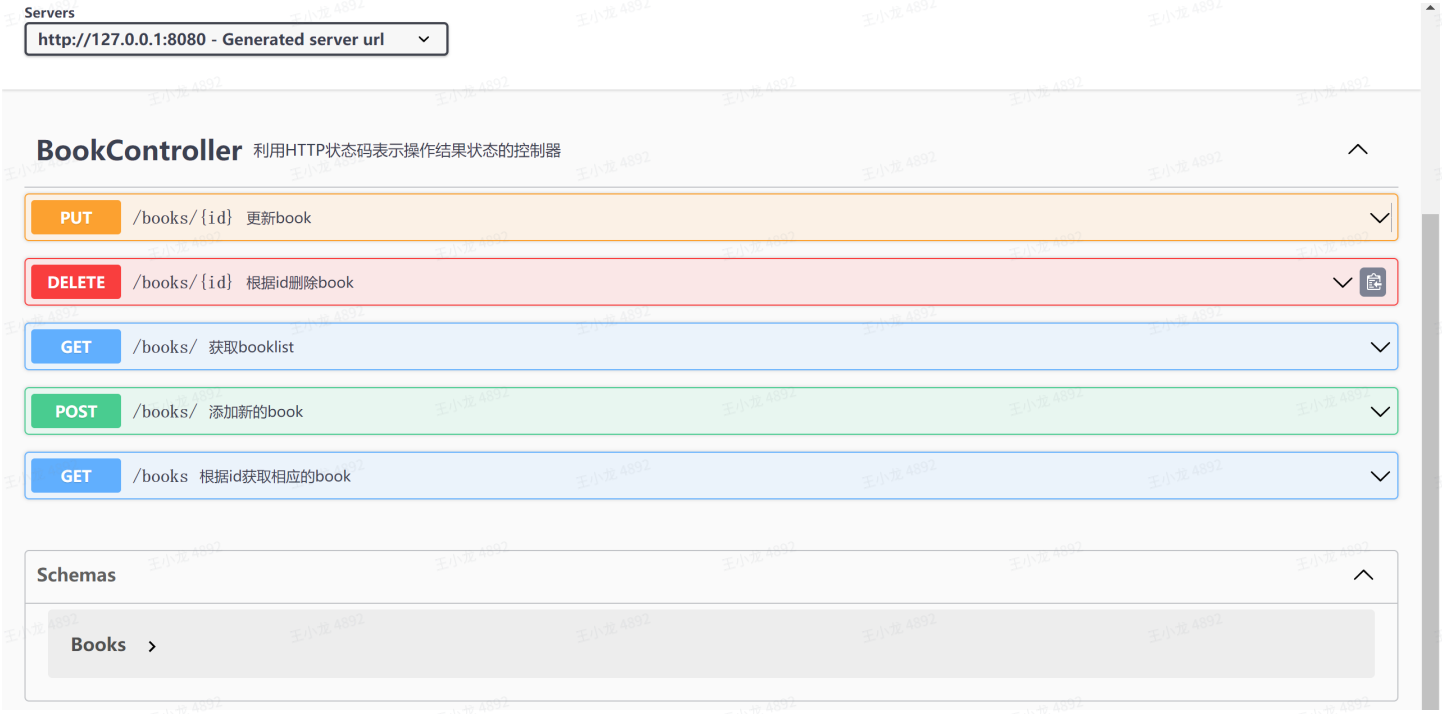
a.

使用Apifox进行接口设计如下：



b.

创建springboot项目，引入springdoc用来生成API接口文档如下：



c.

使用Apifox对接口进行测试如下：

只测试了获取图书列表的接口，其他的类似：

可以看到测试正确响应：

GET 获取图书列表

本地 Mock

文档 修改文档 运行 高级 Mock

GET http://127.0.0.1:4523/m1/2440764-0-default/books/

发送 暂存 保存为用例

Params Body Headers Cookies Auth 前置操作 后置操作 设置

Query 参数

参数名	参数值	类型	说明
添加参数			

Body Cookie Header 7 控制台 实际请求

Pretty Raw Preview Visualize JSON utf8 提取


```
[
    {
        "author": "eu officia veniam",
        "book_id": 74,
        "bookname": "理老现长"
    },
    {
        "book_id": 3,
        "bookname": "革进国产并称",
        "author": "nisi consequat id"
    },
    {
        "author": "sit occaecat in irure",
        "book_id": 74,
        "bookname": "厂金习的料团"
    },
```


校验响应 成功 (200)

200 29 ms 375 B

返回数据结构校验通过