211220154 俞锐基

My Perspective on Software Architecture

软件体系结构这门课讲了什么？三大主线：一是以 Spring MVC 架构为基础的三层模型架构；二是强调分布式、可扩展的架构，如典中典的微服务架构；三是以消息驱动思想为核心的响应式架构。有两个词几乎贯穿了一学期的课程，一个是降耦合，一个是高可用。

一开始讲到 spring 中的 IoC 容器，为什么需要反转控制权，让组件「被动式」注入到系统中，强调的就是降低系统的耦合度，尽量减少写死在代码中的内容，将来需要更换组件，或者做扩展的时候，只需要修改少量的代码就可以做到。之后讲到微服务体系，更是降耦合思想的集中体现，把不同的功能抽象出来，作为一个单独的服务，不同的服务之间只需要约定好接口，也不需要关心别的服务怎么实现的，只需要实现自己这个服务即可。这样就做到了不同模块之间的相互独立，大大降低了依赖度，非常便于大型系统的开发。

如果面试官问我设计一个模块或者系统时需要注意什么，我第一个想到的点就是：备份、冗余。因为在设计的时候，首先要考虑的就是这个系统的可用性，任何模块都可能 crash，怎么确保某个点挂掉不会影响整个系统？最简单的就是做个备份。在讲 Redis 时就在强调主从节点，甚至要部署成一个集群；在大数据实验课上讲 MapReduce 时也讲到 Map 节点要备份、数据也要备份 · · · · · · 这种「多」的思想还体现在微服务架构中：某个服务要多开几个实例，避免单一节点出问题或者压力过大成为瓶颈，要做负载均衡，做水平向扩展。总结下来，「多」的思想体现的就是高可用的观点，如果把系统的各个模块看成一个链条，那么我们要保证链条上的每个节点都不那么容易坏、都不那么慢。

在学习的过程中，我体会到：当提出一个新的架构时，虽然能够解决一些问题，但同时也会引入新的问题，从而使得系统的复杂度提高。例如微服务虽然在降低耦合度这点上很有优势，但是由于服务间通过网络来传输，不仅对网络提出了要求，而且网络传输是不可靠的，存在调用雪崩现象，因此又引入了断路器。把商品模块和订单模块拆分后，怎么保证并发修改的一致性呢？怎么在两个服务中同时开启事务呢？这又要引入分布式事务的组件了。响应式架构虽然通过异步的方式提高了性能，但是也提高了代码的复杂度，异步执行也为 debug 带来了不小的挑战。从一开始的 web 版 pos 机，到最后的电商平台，复杂度在一路提高，带来的后果就是系统维护的难度就不断变高了。

因此，我认为在学习软件体系结构的过程中，应带着「发现问题-解决问题」的思路去学习新的架构。每一种体系结构的出现，并不是为了设计而设计，而是现有的架构出现了问题，例如某个点成为了性能瓶颈，为了解决这个瓶颈，才发明了一种新的架构；或者原来的架构中存在一些设计得不好的地方，在某个场景表现不佳，才引入新的架构。当我们在面对一个系统设计需求时，不应该上来就是微服务 + 高可用这一套组合拳，而应先分析需求、场景，再来考虑使用什么架构。当规模变大了，才考虑变成分布式系统；当数据量上来了，才考虑加入缓存；当并发量大了，才考虑引入消息队列来做削峰限流。结合课内所学知识点举例：为什么需要注册服务？微服务是动态变化的，如扩展、升级、死机，注册服务就可以跟踪并提供最新信息；负载均衡，因为可以知道每个服务的情况；帮助微服务之间互相发现；监控、管理，提供整个系统的大纲。为什么需要网关层？路由，修改微服务层不影响客户端；负载均衡（网关服务也是要注册到 eureka 的，从 eureka 那里获得其他服

务的 IP)；缓存一部分 requested data；安全性，提供验证和授权；监控所有 API 请求，便于诊断问题。

在学习和写作业的过程中，我遇到了很多坑，也发现了一些有意思的、值得思考的地方。

在做 aw07 的时候，需要批处理一个巨大的数据集，然后把数据插入到数据库中。当面对这样的需求时，首先需要考虑怎么建表可以提高效率呢？通常对于海量数据会采取分库分表的做法，虽然作为一次作业并不需要实现多高的性能，但保持这样的思考习惯总是有益的。在这次作业中我选择把数据插入到云服务器的数据库中，我希望处理速度尽可能快一些，但是存在网络、服务器配置、Mysql 性能这些因素的影响，且我暂时没有能力控制这些因素，所以我希望能提高本地批处理的速度。这就需要考虑怎么设计 job 可以提高性能，例如 split flow 并发执行，step 内部串行；或者 split flow 和 step 都并发。设置核心线程数也是一个关键的点，理论上设置为 cpu 核心数是最优的，经过不同参数下的实践，确实验证了这个结论。但写着写着发现，文件读取才是最大的瓶颈。一开始我用普通的 reader，可以观察到数据库最后插入了很多重复项，有些项没有被插入。然后看文档说，普通的 FileReader 是线程不安全的，多线程情况下需要用 SynchronizedItemStreamReader 进行封装。那这不就回到了串行执行的方式吗？为了解决这个问题，需要对大文件进行切分。我的第一想法是找现成的 jsonl 转 json 的工具，还真有，但是用着又出现一个问题：原本数据是 utf-8 格式的，但里面又包含一些 utf-16 的转义字符（用户的 emoji 表情），转换结果会变成 utf-16 格式，这时如果强转回 utf-8 格式，转义字符就会出问题，导致读取错误。工具不行，只能尝试求助 gpt 给我生成一段 python 脚本来处理了，这下处理没问题了，把大文件分割之后，批处理速度得到了很大的提升：数据集大小为 meta 70537，review 130434；原来的耗时要 45min，分割之后只需要 17min。

在做 aw09 的时候，看一些讲 Webflux 视频的评论区看到，Webflux 发展得并不好，主要问题有异步代码不好 debug，支持不够完善等等。我在尝试引入分布式事务的时候也发现，Seata 是不支持 Webflux 架构的，就只能把这个加入 TODO list 中了。然后我看到有人说 jdk21 引入了虚拟线程，可以用于替换 Webflux，想想发现也挺合理的，在 Go 中有协程这个概念，虚拟线程就类似于 Go 的协程，那它在处理 IO 密集型任务时是有很好的性能的，例如课上讲到的 SpringMVC 应用休眠若干毫秒后返回的例子，如果引入协程的话，是不是就可以达到 Webflux 的效果，甚至更好呢，感觉对课程的理解又加深了。

再次引用《Scalable Web Architecture and Distributed Systems》这篇文章软件体系结构做个总结：服务拆分、冗余、分区、缓存、代理、索引、负载均衡、队列，再加上后面的学习的消息驱动，以后谈到软件设计就打这几张牌了。