第十周总结

首先是第十周的课程笔记

10.1 微服务：服务本身的设计、维护以及治理

1. 微服务架构的核心是模块的分解

2. 巨无霸系统带来的问题：

    1. 编译，部署困难

    2. 代码分支管理困难

    3. 数据库连接耗尽

    4. 新增业务困难

3. 解决方案就是拆分，将模块独立部署，降低系统的耦合性

    1. 纵向拆分：将一个大应用拆分成多个小应用，如果新增业务比较独立，那么就直接将其设计部署为一个独立的web应用系统

    2. 横向拆分：将复用的业务拆出来，独立部署成微服务，新增业务只需要调用这些微服务即可快速搭建一个应用系统。

4. WebService的缺点（选框架要注意）

    1. 臃肿的注册与发现机制

    2. 低效的xml序列化手段（这点不是很好理解，感觉dubbo的xml机制还好，因为本身要注册的其实也不多）

    3. 开销相对较高的HTTP远程通信

    4. 复杂的部署与维护手段

5. 微服务框架需求：

    1. 服务的注册与发现，服务调用等标准功能

    2. 负载均衡：服务请求者可以使用加权轮询等手段进行访问，使服务提供者集群实现负载均衡

    3. 失效转移：如果目标服务有服务器宕机，应该能把后续请求转移到没问题的机器上，使得请求能够成功进行调用

    4. 高效远程通信：核心服务每天的调用次数数以亿计，如果没有高效的远程通信手段，服务调用很可能成为系统瓶颈

    5. 对应用最少侵入：微服务的使用者尽量不感知微服务框架的使用（类比于dubbo，其实调用一个远程服务是比较方便的，和一个本地的bean区别不大）

6. 版本管理：服务端多版本实例同时运行，可供不同的客户端进行调用（感觉这个见仁见智吧，大部分case其实是业务上做了兼容，另外，不同实例运行不同版本，其实也可以用换接口的方法来代替）

10.2 微服务：落地实践的策略与思路

1. 微服务架构落地

    1. 业务先行，先理顺业务边界和依赖，技术是手段而不是目的

    2. 先有独立的模块，后有分布式的服务

    3. 业务耦合严重，逻辑复杂多变的系统进行微服务重构要谨慎

    4. 要搞清楚实施微服务的目的是什么，业务复用？开发边界清晰？分布式集群提升性能?

2. 读写分开部署：可分别进行优化，保护数据（个人感觉更多的是带来不必要的复杂性，比如说vo重新复制一次。。。）

3. 事件溯源：以日志方式记录所有的操作，用于监控和复盘，支持分布式事务等

4. 断路器：可用于服务降级

5. 服务调用及超时：上游应大于下游

6. 最重要的是需求：需求 -> 价值 -> 原则 -> 实践 -> 工具

10.3 微服务网关的技术架构

1. 基于网关的微服务架构：以网关作为统一入口

2. 网关作用：

    1. 统一接入：高性能，高并发，高可靠，负载均衡

    2. 安全防护：防刷控制，黑白名单

    3. 流量管控与容器：限流、降级、熔断

    4. 协议适配：http，dubbo，jsf…

3. 网关管道模式：本质是一连串连续调用的filter

4. 异步网关：从描述看起来，从客户端发起请求到客户端收到结果，都是异步的，也就是说，客户端需要有一个CallBack异步处理服务端返回的结果，实现全流程非阻塞以后，整体的性能会更高

5. 开放平台网关：需要有计费，审计等模块，可以使用OAuth授权

10.4 领域驱动设计DDD

1. 为什么要DDD？ -> 主要是解决代码一坨的问题（互联网企业反而这种情况特别严重）

2. 领域驱动设计需要基于现实中的业务关系去进行设计，架构师需要考虑产品的界面是否合理，有没有更好的

3. 子域：领域驱动设计第一步就是子域拆分：商品、订单、发票、用户、物流、库存等

4. 司机提现场景：界面驱动设计的话，收入合法性，订单检查等会放在司机子域，单实际上应该放在财务以及其他的一些子域

5. 限界上下文：在一个字域中，会创建一个概念上的领域边界，在这个边界中，任何对象都只表示特定于该边界内部的确切含义。这样的边界被称为限界上下文。限界上下文和子域有一对一的关系，用来控制子域的边界，保证字域内的概念统一性。通常限界上下文表示一个模块，微服务或者子系统

6. 上下文映射图：描述限界上下文之间的交互

7. 实体：领域模型对象也被称为实体，实体本身可变，但其唯一识别ID不变，实体设计，是DDD的核心所在，关键是要把握业务中实体所承担的职责。

8. 值对象：领域内不变的对象，有点像在说enum

9. 聚合：一些关联对象的集合，一个微服务的最小单位，随着产品的迭代壮大，可以以聚合为单位进行拆分，以此维护服务的边界和底线。

10. 六边形架构：适配器 -> 应用程序 -> 领域模型

10.5 软件组件设计原则

1. 软件的复杂度和它的规模成指数关系。一个复杂的度100的系统如果能拆分成两个互不相关的子系统，子系统各自的复杂度只有25，也就是说，软件需要模块化，组件化的设计。

2. 组件内聚原则：讨论哪些类应该聚合在同一个组件中，以便既能提供相对完整的功能，又不至于太过庞大。

    1. 复用发布等同原则：希望别人以怎样的维度复用软件，就应该以怎样的维度发布软件。

    2. 共同封闭原则：将那些会同时修改，并且为了相同目的而修改的类放到同一个组件中。

    3. 共同复用原则：不要强迫用户依赖他们不需要的东西

3. 组件耦合原则：讨论组件之间的耦合关系如何设计

    1. 无循环以来原则：组件之间不应该进行循环依赖。循环以来很容易导致问题难以分析和定位。循环依赖往往发生于无意识的界面驱动编程里面，有意识的设计中往往会很容易意识到这个问题。

    2. 稳定依赖原则：一个组件不应该依赖比自己更不稳定的组件，被很多组件依赖的组件应该是相对稳定的，变更很少的组件。

3. 稳定抽象原则：一个组件的稳定程度应该和其抽象化程度一致。即一个稳定的组件应该是抽象的。

10.6 案例：用领域驱动设计驱动系统重构

1. 系统的重构是不可避免的：软件开发过程中，大家对系统的认知会不断发生变化，当系统和大家的认知相差越来越大以后，系统就会变得难以维护。

2. 如何随着业务的发展保持高内聚低耦合?

1. 低复杂性系统采用CRUD成本相对小一些，地复杂系统使用DDD也叫overengineering

2. 搞复杂系统采用CRUD设计代码复杂度指数级上升，这时候需要改成DDD

3. 出行app重构总结：

1. 当前系统设计与问题汇总讨论：代码与架构混乱，需求迭代困难，部署麻烦，bug率升高

2. 针对问题分析具体原因：微服务A太庞大，微服务B和C职责不清晰，团队内业务理解不一致，内部代码设计不良，硬编码和耦合太多

3. 重新梳理业务流程，明确业务术语，进行DDD战略设计：（泳道）活动图、子域分解，限界上下文设计

4. 针对当前系统设计和DDD不匹配的地方设计微服务重构方案

5. DDD战术设计与技术验证：聚合、实体、值对象设计，打样代码开发

6. 任务分解与持续重构：在不影响业务开发的前提下，按照战略和战术设计，将重构开发和业务迭代有机融合。

个人总结

这周主要讲了微服务框架，领域驱动设计，以及组件设计原则。就像作业题2中所总结的那样，微服务，领域驱动设计，以及组件设计其实都是在通过各种方式，把一个巨无霸型的复杂业务不断拆分成小的业务，使得整体的复杂度可以指数级下降。其中，微服务和组件设计主要是站在一个大的业务的维度，进行服务级别的拆分，而领域驱动设计则更关注服务内部实体的抽象和相互之间的关系，以降低代码的整体复杂度，两者对于做好一个大的应用都非常重要。