系统架构设计模式

1.DDD

各种边界上下文，最终就是聚合。过度关注名词--领域对象，相反，我们应该更多关注动词，即事件。DDD寻找聚合就是寻找各种事件+状态构成的因果链。

2.响应式设计

两个阶段：响应式编程和分布式的响应式系统，提高对资源的时间和空间调度能力。将无状态行为从有状态的实体中分离出来，解耦行为和结构（将数据模块和计算模块分离）。

响应式编程：基于事件的异步响应模式，提高单机多线程资源调度能力。

响应式系统：基于异步消息传递，提高机器间资源调度能力。

3.基于事件的持久化

日志应该基于数据+命令（事件），数据库是日志的缓存子集。

分布式事务：

使用最终一致性带来的伸缩性远高于两阶段提交（两阶段提交是违反伸缩性和高可用性的）。

大型网站后台架构

1.水平+垂直方向业务拆分

a) 水平拆分，即分层，如接入层，逻辑层，持久层等。

b) 垂直拆分，即将业务拆分成多个独立模块，进行独立部署和伸缩，每个模块有自己的业务数据库。

2.集群

一致性hash解决相同业务模块多台机器的增删问题，也包括缓存集群机器。

3.缓存

a) 本地缓存，即内存。

b) 分布式缓存，如redis。

c) 反向代理，如nginx。

d) CDN。

4.异步

a) 消息队列，解耦模块间的强耦合（调用耦合，开发耦合），实现最终一致性，队列缓冲与限流。

5.数据库集群

a)分库+分表

b)读写分离

安全认证与鉴权

1.HTTP基本认证

a) client 发送request到server。

b) server返回401“未授权”。

c) client把用户名和密码用base64加密后，放在authorization header中发给server。

d) server认证通过，发送数据给client。

2.Session认证

用户登陆后，将信息存储在业务服务器，返回sessionid给client。相比web server的session复制，更好的选择是session分布式存储，方便相关业务模块扩展。

3.Token

a) client向认证服务发送登录信息，如果验证通过，认证服务返回token给client。

b) client存储token，后续client请求都携带token，业务服务器验证token并返回数据。

4.JWT

client登录到server，server将用户名和密码发送到认证服务器，认证通过server生成jwt给client，并加入到后续请求header中。Server校验jwt并返回数据。Jwt其实是server端的私钥加密数据。

5.Oauth2.0

client向res owner请求grant，client携带grant到res auth服务器生成token，client携带token到res server请求并返回资源。

Oauth适合第三方接入场景的权限管理，JWT适合端到端之间的访问鉴权。

微服务架构设计

微服务：一些协同工作的小而自治的服务。
特点：
a)很小：专注于做好一件事。
b)自治性：服务间独立部署和更新。
好处：
a)技术异构性
b)弹性（处理服务不可用和功能降级）
c)扩展
d)简化部署
e)与组织结构相匹配
f)可组合，可替代

好的架构是一个持续迭代的过程，不是一蹴而就的。

微服务架构设计

数据库扩展：

读扩展：

a)缓存

b)主从结构，主写，多副本读。（过去流行，现在更高效的是缓存）

写扩展：

a)sharding分片

将要写入的数据用key进行hash，写入到具体的分片。比如key可以是记录id取模，可以是省市区域编号。分片写的一个问题是集合查询，需要从每个分片异步读取满足条件的记录，最后进行合并。分片写另一个问题是增加额外节点时的数据复制，解决这个问题需要采用一致性hash算法。

sharding分片只能扩展写，但并不解决弹性问题。某个节点宕机，该节点上的数据就不可用，解决办法是在一致性hash的基础上增加虚拟节点，保证每个节点的数据都有多个备份。

总之，你要么在自己的应用程序中管理一致性hash问题，要么加一个中间层，要么使用已经提供了一致性hash功能的数据库，如mongo（sharding），cassandra（一致性hash+节点副本）。

微服务架构设计

CAP定理：一致性，可用性，分区容错性最多只能保证其中两个。

a)AP系统：牺牲一致性，如一般的web系统，网络支付系统（最终一致性）。

b)CP系统：牺牲可用性，如必须保证强一致性的银行系统。

服务降级：某个服务故障不会影响整个服务的正常使用。比如购物网站购物车模块不可用，不应该影响其他的如商品列表展示，直接支付购买等其他功能。故障的购物车模块可以做一些服务降级处理，如显示“马上回来”或其他处理等。无论哪个模块故障，都不应该导致整个服务的不可用。

秒杀系统架构设计

典型的读多写少系统，所以正确的处理该特点即可。

思路：

a)将请求尽量拦截在前端。请求都压在数据库上，肯定不行。

b)充分利用缓存。典型的读多写少系统，适合缓存。

具体实现：

a)web层请求拦截。如页面限制用户点击次数，js与后台的刷新频率，添加验证码防止刷票等。可以拦截大部分无效请求。

b)接入层请求拦截和缓存。同一个id限制访问频率，同一个商品item缓存等，应对刷票软件。可以拦截大部分该层请求。

c)服务层，对数据库写请求进行排队，每次只进行少量请求的执行；对数据库的读请求进行Redis缓存。

d)数据库层。保证高可用即可，到达这一层的请求很少了。