如何构建亿级流量的网站系统-秒杀系统

配置环境：

服务器环境： 4 台服务器（阿里云服务器： 4cpu, 8GB 内存）

云原生迁移： 30 台服务器环境 ----- kubernetes 容器云

课程内容计划：

1、如何从架构的角度思考问题，如何构建一个高可用，高性能的架构的系统（架构设计，架构思路—如何选择一个合适的架构）

2、压力测试，及时发现系统的问题，系统瓶颈；根据压力测试结果，对系统进行性能的优化，问题修复，验证优化结果

3、服务端优化(tomcat 服务器优化，undertow 优化)，压力测试

4、JVM 优化（GC 日志分析，根据调优日志对 jvm 进行进一步的线上的环境调优--- jvm 调优原理）

5、数据库调优（数据库调优，连接池，缓存，表设计）

6、多级缓存（堆内缓存，分布式缓存，接入层缓存： openresty 内存字典 ， lua+redis 实现缓存）

7、秒杀下单（高并发下写操作）----- Lock 锁，Aop 锁，分布式锁（MySQL，redis，zookeeper）--- 对锁进行优化

8、写异步（队列对下单进行优化 ： BlockingQueue,RocketMQ）

9、数据一致性问题处理（RocketMQ 最终消息一致性）---- 考虑性能

10、架构进行重构 （单体架构重构为微服务架构，SpringCloud alibaba）

11、分布式环境下接口幂等性的问题

12、分布式环境下数据一致性的问题（强一致性）

13、防刷限流技术（防止后端服务被大流量冲垮）

14、kubernetes 云原生迁移（把微服务架构迁移到云原生模式下）

1 项目计划

1.1、课程前言

1、必须具有一定的开发基础，CRUD 不再关注

2、注重的是实际业务场景

3、注重的是问题的解决方案

4、注重的是架构的设计思路

1.2、课程特色

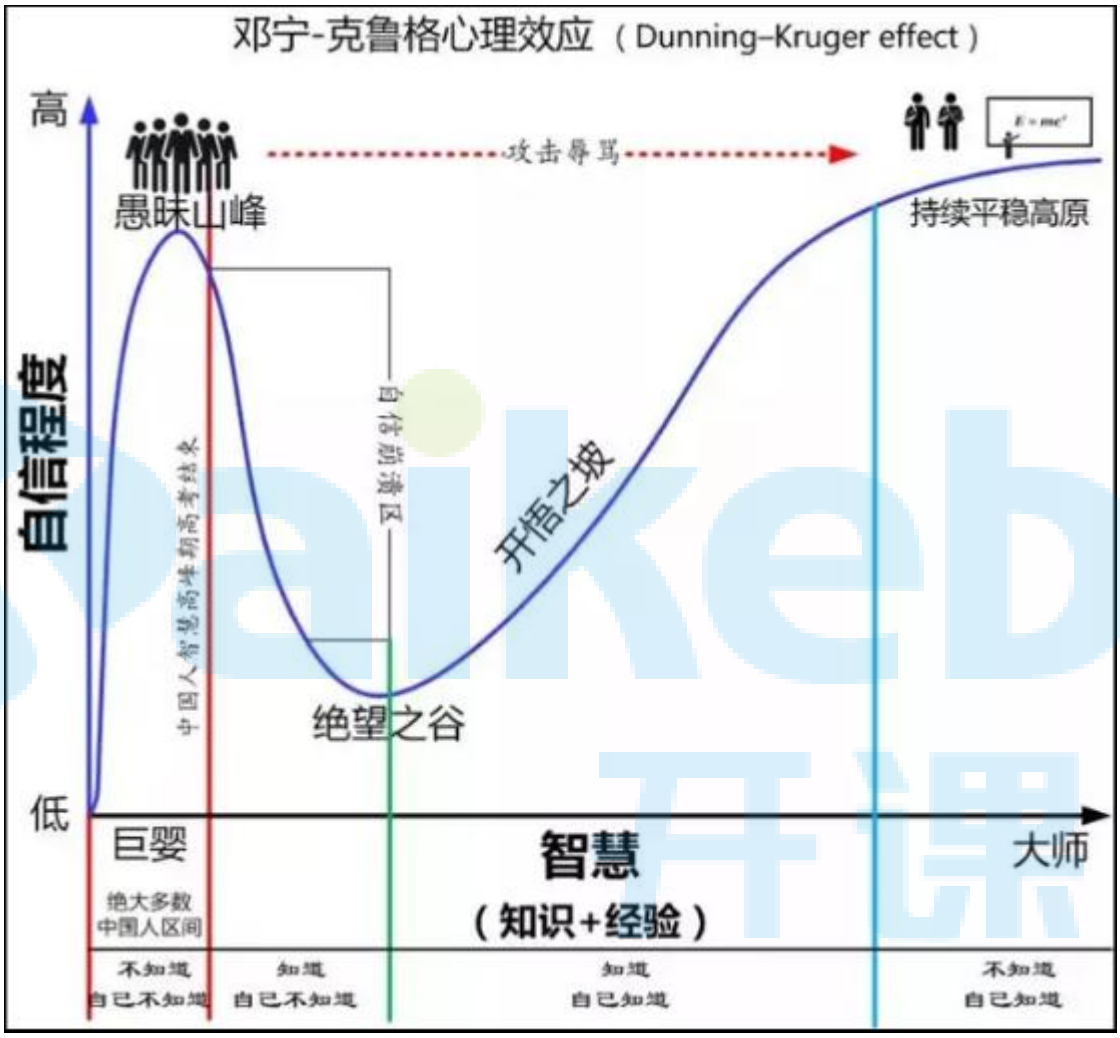
- 提升架构高度，仅仅寄希望于代码层级是远远不够的。
  - 代码解决的的执行力的问题，架构更多的是依赖 业务的洞察能力 和 技术视野
- 课程重点
  - 架构解决方案
    - 技术解决方案落地
  - 架构背后思考
  - 核心问题解决方案
- 全链路压力测试

1.3、课程问题

- 项目实战 和 其他的 课程到底有什么区别？
- 课程中是否敲代码？（核心代码编写）

2 架构师认知

2.1、架构师成长之路



2.2、架构是什么？

1）对业务场景抽象后得出的支持骨架（网络拓扑结构）
老板：100w 日活量，10W QPS 微服务架构
2）架构为业务场景而生、被业务场景而弃
老板：10 天上线
3）架构没有最好、只有“最合适”（人员技术研发能力、业务复杂度、数据规模大小、时间成本、运维能力….）
4）“最合适”架构都是业务场景折中（Balance）的选择

总结：选择架构时候，必须选择最适合公司当下环境的架构。

2.3、架构目标是什么？

用户网站访问调查：reponse time : 3s ---- 60% 用户流失

RT 时间：ms 高性能 （前端：美观大气的上档次页面—非常简单，后端：一系列的优化 ms）
高可用： 任何时候项目都必须可用
可升缩： 大促，流量瞬间增大....
可扩展： 开发角度（新需求进行迭代），扩展
安全性： 网络安全，硬件安全，软件安全

敏捷开发： 可持续交付，可持续部署
架构师目标： 采用什么样方式，才能构建以上目标的项目？？

2.4、架构模式？-- 架构策略

分层：分层拆分（表现层，业务层，持久层）
分割：连接池分割，机房，进程（分布式）
分布式：分布式架构
集群： 高可用
缓存： 堆内存缓存，redis 缓存，lua 缓存
异步： 写异步
冗余： 数据库设计，读，写
安全： 数据安全（加密）、系统安全
自动： 运维，扩容，缩容；
敏捷： 可持续集成，交付，部署

2.5、高性能架构

以用户为中心，提供快速的网页访问体验。主要参数有较短的响应时间、较大的并发处理能力、较高的吞吐量与稳定的性能参数。

可分为前端优化、应用层优化、代码层优化与存储层优化。

- 前端优化：网站业务逻辑之前的部分；--- vue ,react +nodejs – 工程化
- 浏览器优化：减少 HTTP 请求数，使用浏览器缓存，启用压缩，CSS JS 位置，JS 异步，减少 Cookie 传输；CDN 加速，反向代理；

- 应用层优化：处理网站业务的服务器。使用缓存，异步，集群，架构优化

- 代码优化：合理的架构，多线程，资源复用（对象池，线程池等），良好的数据结构，JVM 调优，单例，Cache 等；

- 存储优化：缓存、固态硬盘、光纤传输、优化读写、磁盘冗余、分布式存储（HDFS）、NoSQL 等

总结：
1）服务尽量进行拆分（微服务）---- 提高项目吞吐能力
2）尽量将请求拦截在上游服务（多级缓存）--- 90% ----> 数据库压力非常小，闲庭信步，数据库架构（主从架构）
3）代理层（做限速，限流）
4）服务层：按照业务请求做队列的流量控制（流量削峰）

2.6、高可用架构

大型网站应该在任何时候都可以正常访问，正常提供对外服务。

因为大型网站的复杂性，分布式，廉价服务器，开源数据库，操作系统等特点，要保证高可用是很困难的，也就是说网站的故障是不可避免的。
以上问题和我们架构师保证服务高可用性是相互矛盾的；因此架构师要做的事情就是解决以上问题，保证服务高可用性；

服务问题：
1、业务问题 – 业务高可用性
2、系统的问题 – 系统高可用性

业务上也需要保证，网站高可用性。（bug,异常）

例如：
对输入有提示，数据有检查，防止数据异常。

系统健壮性强，应该能处理系统运行过程中出现的各种异常情况，
如：人为操作错误、输入非法数据、硬件设备失败等，系统应该能正确的处理，恰当的回避。

因软件系统的失效而造成不能完成业务的概率要小于 5‰。

要求系统 7x24 小时运行，全年持续运行故障停运时间累计不能超过 10 小时。

系统缺陷率每 1,000 小时最多发生 1 次故障。

在 1,000,000 次交易中，最多出现 1 次需要重新启动系统的情况。

业界：采用 N 个 9 评估系统的高可用性：
2 个 9： 系统可用性：99% ---------- 基本可用 87.6h / year
3 个 9 8.8h
4 个 9 53min
5 个 9 5 分钟
6 个 9

如何提高可用性，就是需要迫切解决的问题。首先，需要从架构级别考虑，在规划的时候，就考虑可用性。
不同层级使用的策略不同，一般采用冗余备份和失效转移解决高可用问题。

- 应用层：一般设计为无状态的，对于每次请求，使用哪一台服务器处理是没有影响的。一般使用负载均衡技术（需要解决 Session 同步问题）实现高可用。
- 服务层：负载均衡，分级管理，快速失败（超时设置），异步调用，服务降级，幂等设计等。
- 数据层：冗余备份（冷，热备[同步，异步]，温备），失效转移（确认，转移，恢复）。数据高可用方面著名的理论基础是 CAP 理论（持久性，可用性，数据一致性[强一致，用户一致，最终一致]）

总结：
1、负载均衡 （故障转移）
2、限流
3、降级
4、隔离（线程隔离，进程隔离，集群隔离，机房隔离，读写分离，动静分离，热点隔离…）
5、超时、重试
6、压测与预案
大促：演练

2.7、可伸缩架构

伸缩性是指在不改变原有架构设计的基础上，通过添加/减少硬件（服务器）的方式，提高/降低系统的处理能力。

- 应用层：对应用进行垂直或水平切分。然后针对单一功能进行负载均衡（DNS、HTTP[反向代理]、IP、链路层）

- 服务层：与应用层类似；
- 数据层：分库、分表、NoSQL 等；常用算法 Hash，一致性 Hash

云原生：项目运行云端，可以随时动态扩容—K8S

8 核心+16G : 2000QPS +- (此数字是估算结果，真实结果受到代码编写数据结构，业务逻辑，架构、rt,以现实测试结果)

2.8、可扩展架构

SOA,微服务 --- 根据业务拆分模块 ----- 新业务需求 ---- 根据新的业务需求创建一个新模块服务

可以方便地进行功能模块的新增/移除，提供代码/模块级别良好的可扩展性。

- 模块化，组件化：高内聚，低耦合，提高复用性，扩展性。
- 稳定接口：定义稳定的接口，在接口不变的情况下，内部结构可以“随意”变化。
- 设计模式：应用面向对象思想，原则，使用设计模式，进行代码层面的设计。
- 消息队列：模块化的系统，通过消息队列进行交互，使模块之间的依赖解耦。
- 分布式服务：公用模块服务化，提供其他系统使用，提高可重用性，扩展性。

2.9、安全架构

对已知问题有有效的解决方案，对未知/潜在问题建立发现和防御机制。对于安全问题，首先要提高安全意识，建立一个安全的有效机制，从政策层面，组织层面进行保障，比如服务器密码不能泄露，密码每月更新，每周安全扫描等。以制度化的方式，加强安全体系的建设。同时，需要注意与安全有关的各个环节。安全问题不容忽视，包括基础设施安全，应用系统安全，数据保密安全等。

- 基础设施安全：硬件采购，操作系统，网络环境方面的安全。一般采用正规渠道购买高质量的产品，选择安全的操作系统，及时修补漏洞，安装杀毒软件防火墙。防范病毒，后门。设置防火墙策略，建立 DDOS 防御系统，使用攻击检测系统，进行子网隔离等手段。
- 应用系统安全：在程序开发时，对已知常用问题，使用正确的方式，在代码层面解决掉。防止跨站脚本攻击（XSS），注入攻击，跨站请求伪造（CSRF），错误信息，HTML 注释，

文件上传，路径遍历等。还可以使用 Web 应用防火墙（比如：ModSecurity），进行安全漏洞扫描等措施，加强应用级别的安全。

- 数据保密安全：存储安全（存储在可靠的设备，实时，定时备份），保存安全（重要的信息加密保存，选择合适的人员复杂保存和检测等），传输安全（防止数据窃取和数据篡改）；

常用的加解密算法（单项散列加密[MD5、SHA]，对称加密[DES、3DES、RC]），非对称加密[RSA]等。

3 互联网认识

3.1 互联网发展

3.2 Web 演进

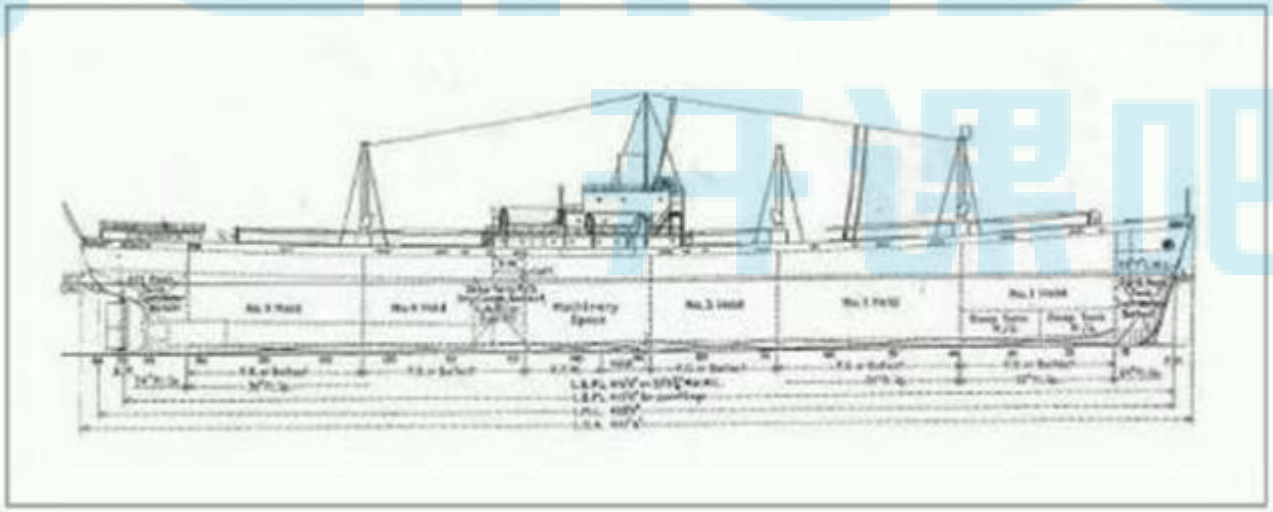
3.3 发展特点

3.4 上线全过程

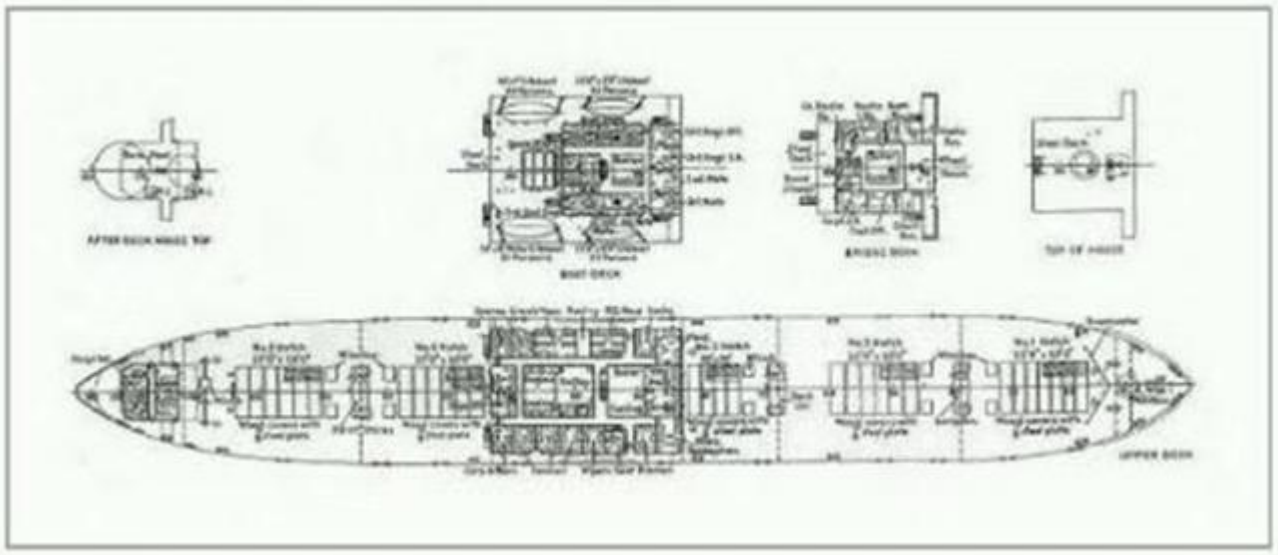
客户（老板）需求：造一条船，能过河就好

产品经理：我们可以提供这样的方案

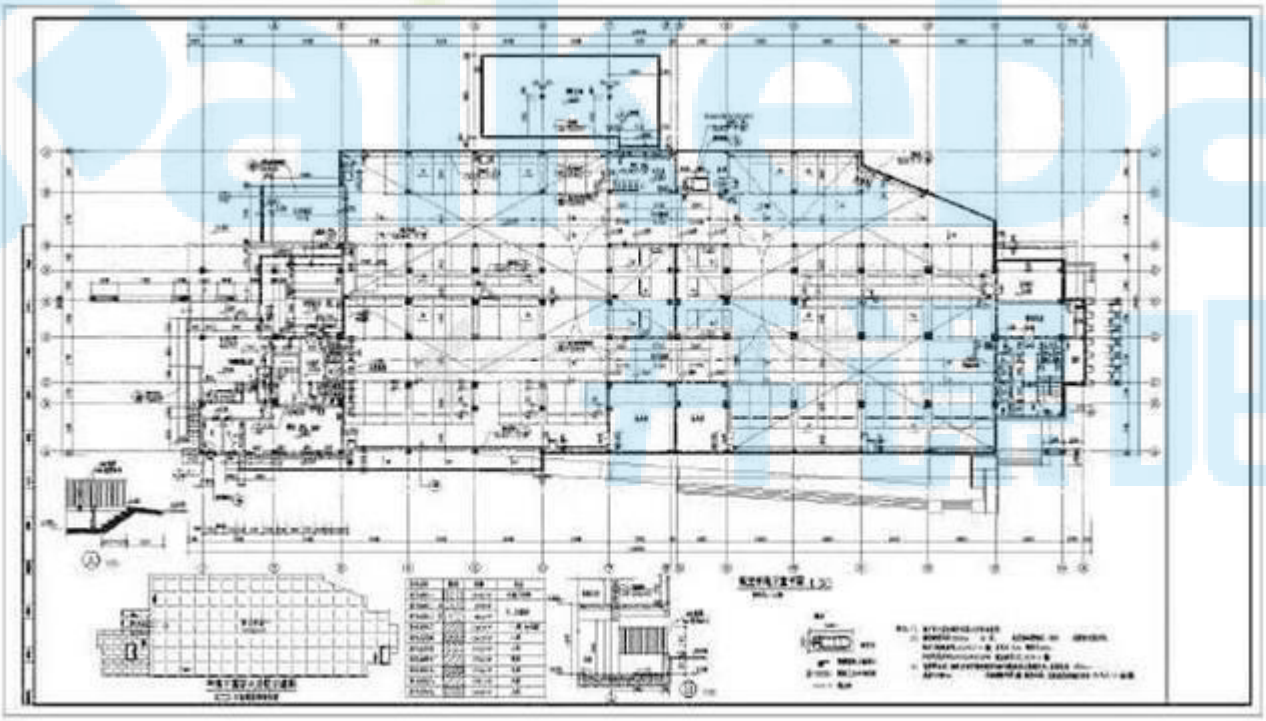
首席架构师：按照需求规划蓝图



高级研发经理：进行项目分解



技术平估：这个项目至少需要 1.5 年



老板发话：市场不等人，先上线再迭代，给技术团队 1 个月时间！

研发团队：重新更改设计，马上开始编码

测试团队：提前进入单元测试阶段

测试团队：突进入集成测试阶段

项目终于止式上线了

船动了，产品实际跑起来了

可伶了这帮苦逼的人肉运维

市场部：升始对外宣传成功案例

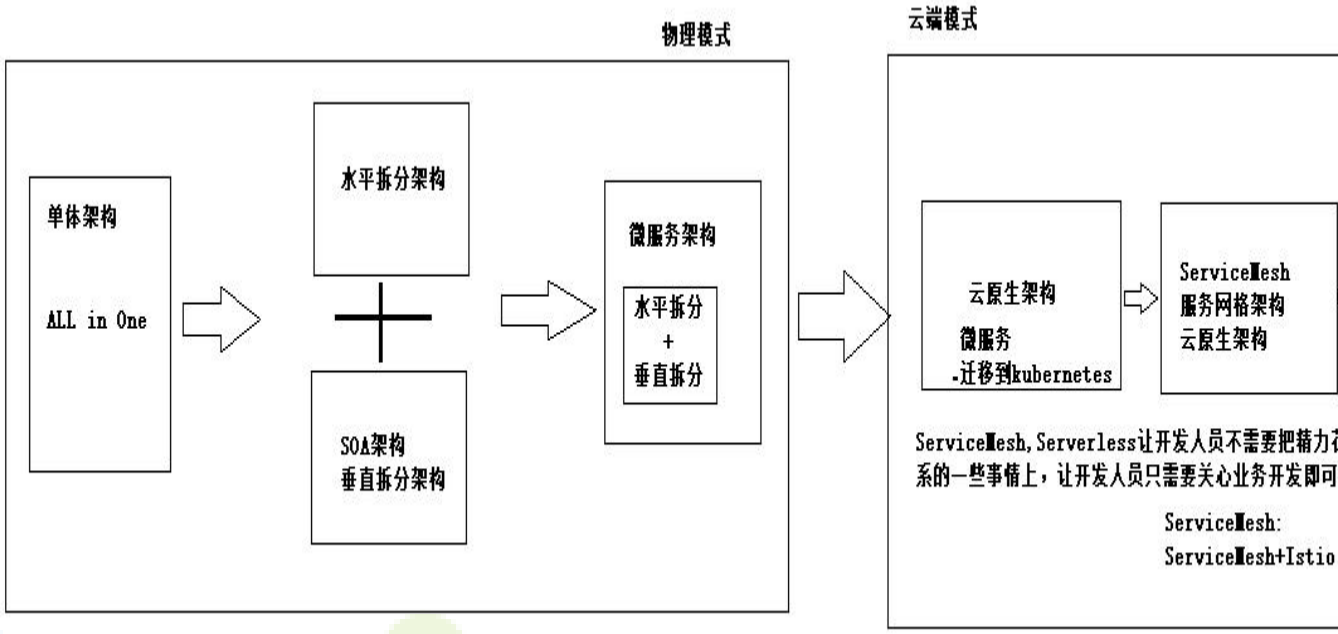
4 互联网架构演进思考

4.1 架构演进

单体架构（all in one） → 水平拆分/SOA 架构 → 微服务架构（云原生架构迁移） → ServiceMesh 服务网格（云原生架构）→ Serverless（无服务架构）

企业降本增效：企业数字化转型唯一路径

开发环境（链路） → 云原生架构转型



4.2 单体架构真的不能应对亿级流量吗？

单体架构： 很多企业（中小型企业，创业公司） 都在使用单体架构；

1、传统项目（并发量小，业务简单，需求固定），项目体量比较小（国企 ： war > 1G IBM 高性能机器，unix --- war）

2、小程序

3、追求极致性能体验的项目---必须使用单体架构（单体架构请求响应链路非常短，因此 RT 时间非常小）

4、互联网项目（创业型，中小型）

从流量的角度思考，单体架构是否可以抗的住亿级流量：

场景： 某电商网站，100w 订单 / day 订单产生时间段：(11:00 --- 2:00 , 5:00 – 12:00)

计算系统流量：

每下一单，发送了多少个请求？？？ ---- 平均下一单： 50 请求

流量： 100w * (50 x 3) = 1.5 亿

计算： 平均每一台服务器需要承担多大并发？？

1.5 亿 / 12h = 1250w QPS /h

1250QPS / 60 = 21w QPS /h

21w / 60s = 3400 QPS / s ----- 4cpu,8GB

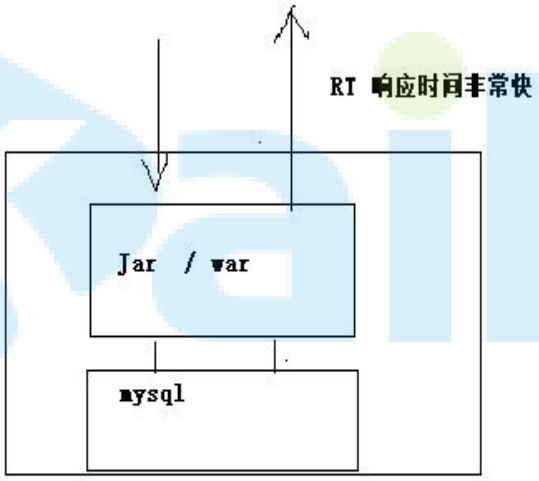
结论： 单体架构完全可以承受亿级流量；根据以上估算，最终落在每台服务器 3400 / s QPS, 因此对于单体架构来说，完全 ok；

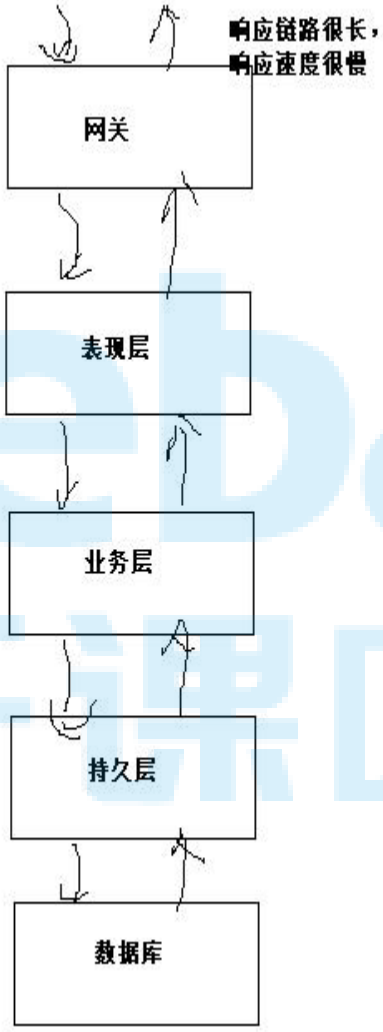
4.3 单体架构

单体架构优势：

1、部署简单
2、开发简单
3、测试简单
4、集群简单

最大优点： 响应时间非常快速，响应链路非常短





单体缺点：

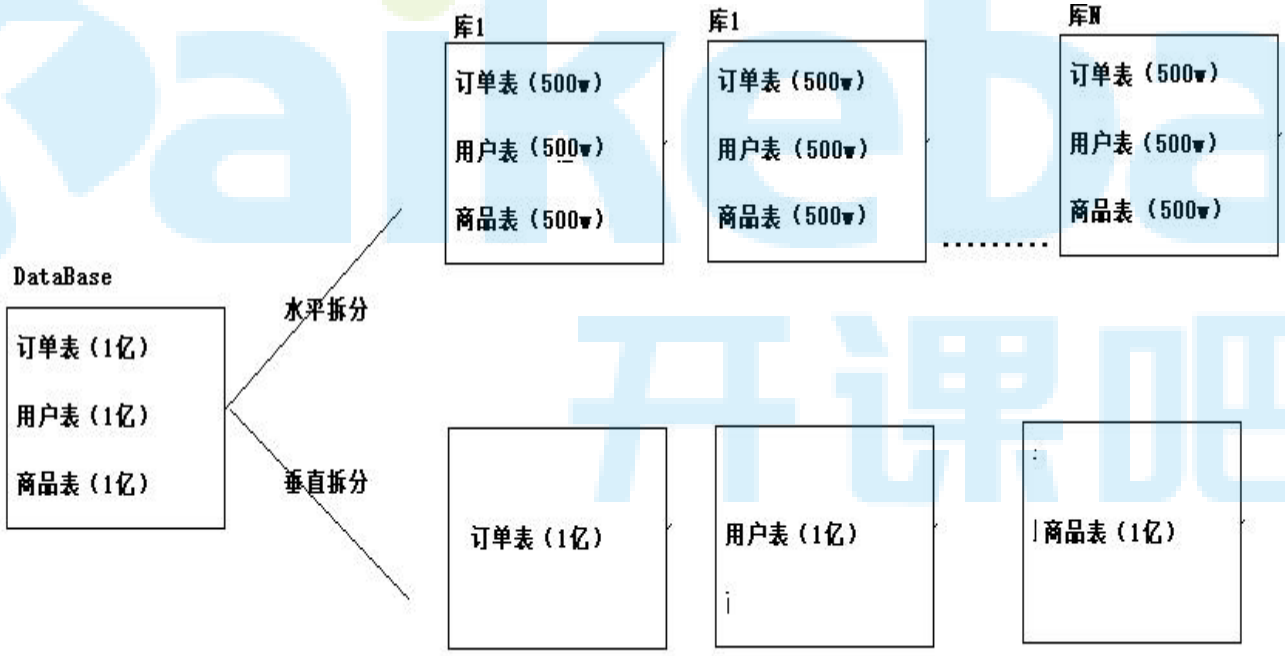
1、流量比较集中，单体架构无法应对
2、无法实现敏捷开发，业务增大，代码结构臃肿，维护变得困难
3、单体架构牵一发而动全身
4、扩展性差
5、稳定性差

单体架构性能优化： 提升单体架构性能

1、服务集群 ---- 解决高可用问题，及性能问题
2、多级缓存（堆内缓存，分布式缓存，浏览器缓存，接入层缓存）
3、动静分离
4、线程池隔离，进程隔离，机房隔离
5、队列术（BlockingQueue,Disruptor,RocketMQ）
6、接入层限流
7、数据存储优化

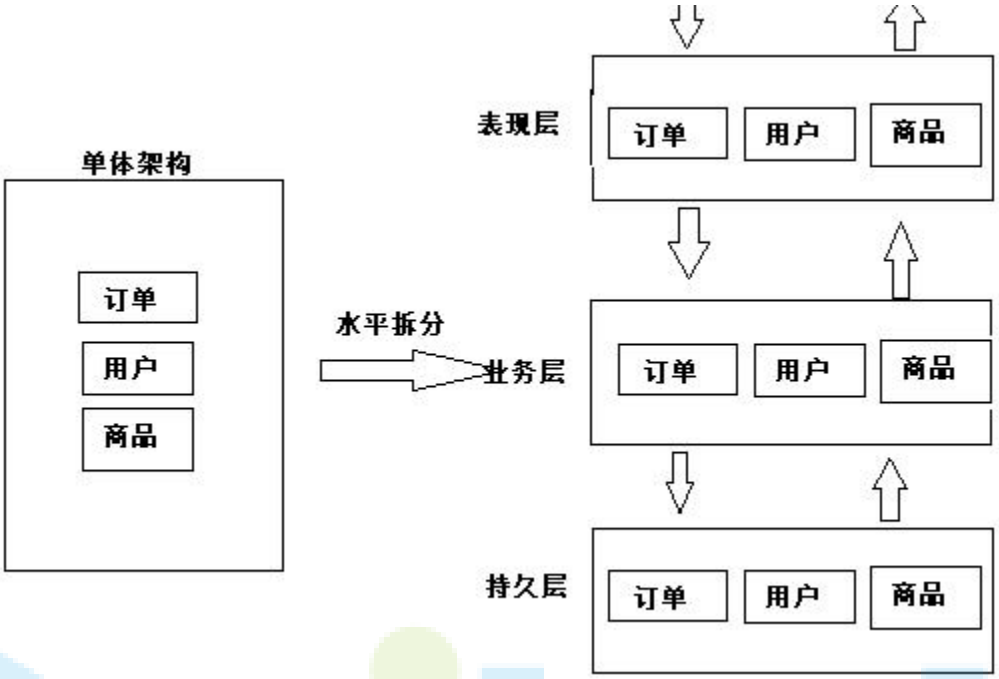
4.4 拆分架构

数据库表拆分模式：分库分表—拆分模式： 水平拆分模式，垂直拆分的模式；

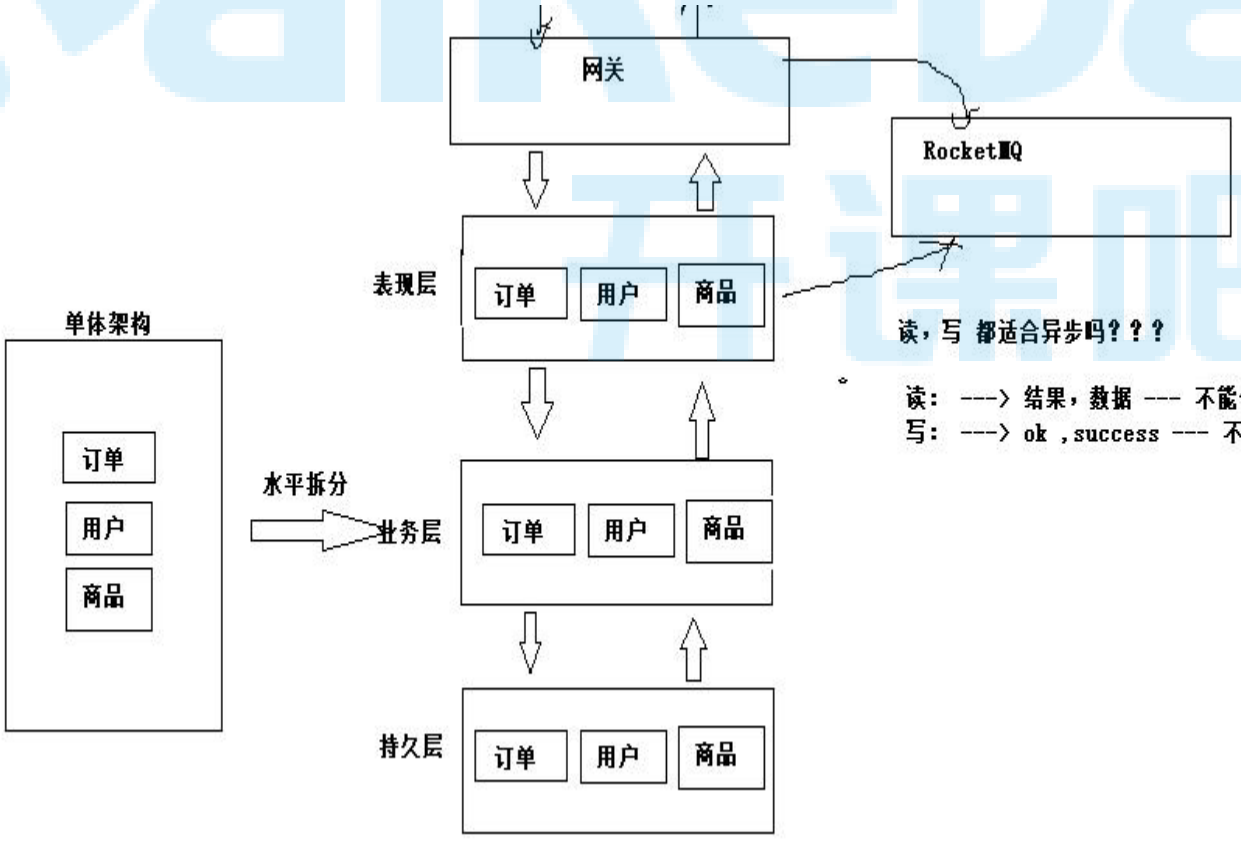


项目拆分模式： 其实和数据库的拆分是一样的拆分方式；因此项目拆分根据业务的划分，拆分如下：

1）水平拆分

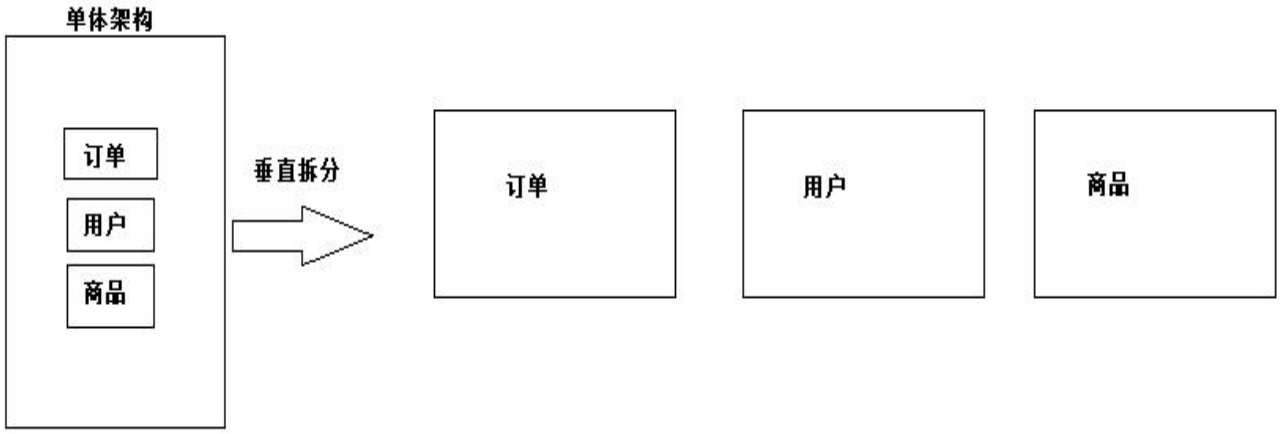

单体架构
订单
用户
商品
水平拆分
表现层
业务层
持久层
订单
用户
商品

水平拆分来说： 结构优化


网关
RocketMQ
表现层
单体架构
订单
用户
商品
水平拆分
业务层
持久层
读，写 都适合异步吗？？？
读： ---> 结果，数据 --- 不能使用异步
写： ---> ok ,success --- 不能使用异

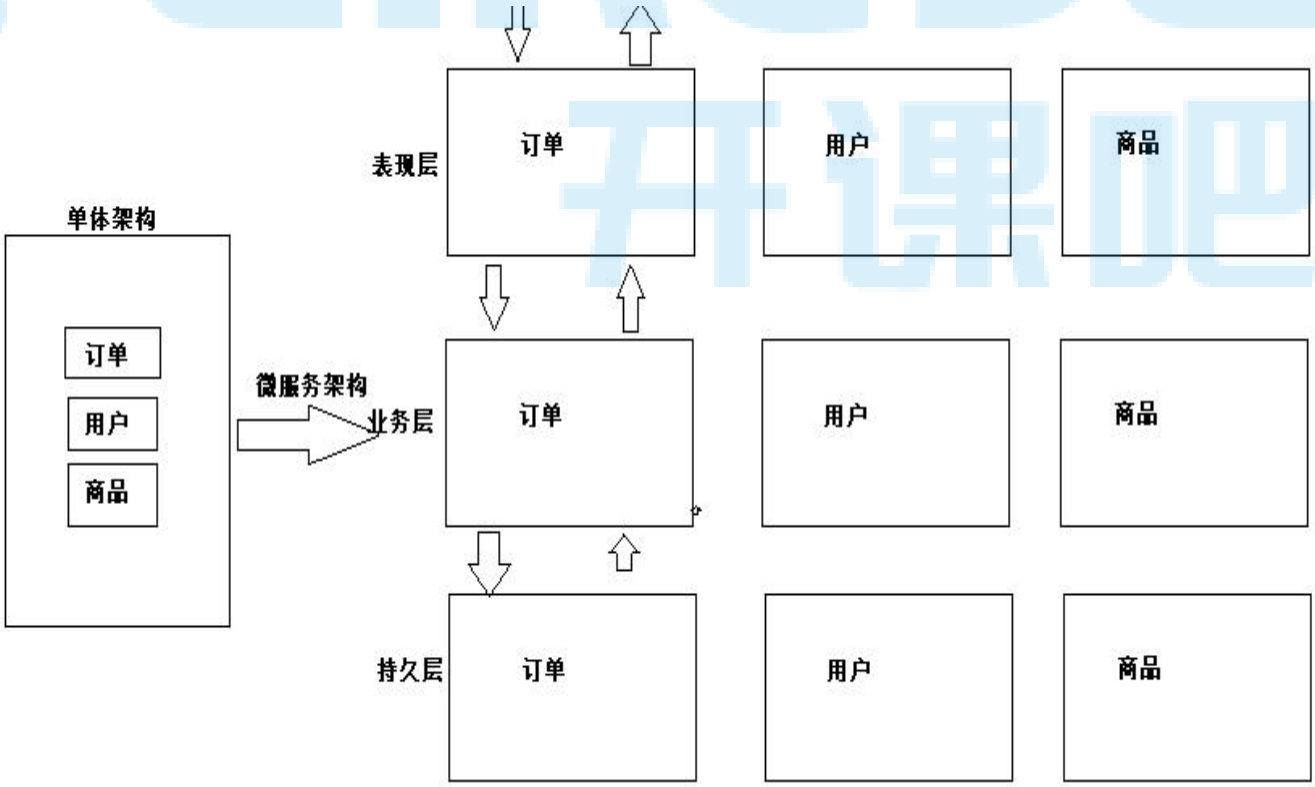
2）垂直拆分

项目中，根据业务进行拆分的；



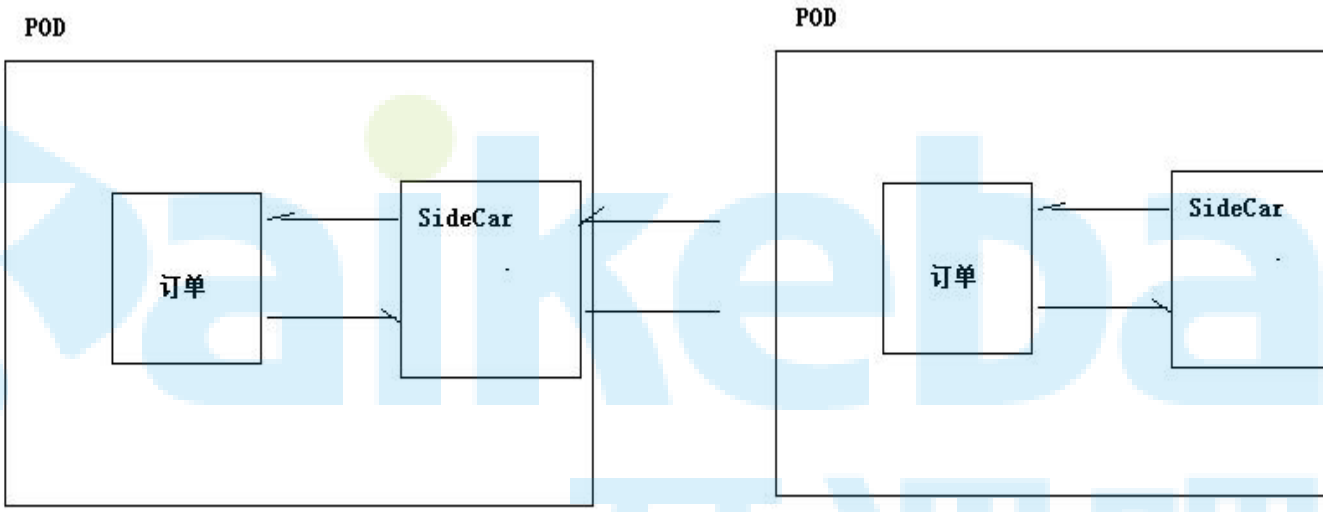
4.5 微服务架构

微服务架构： 水平拆分+垂直拆分的结合体



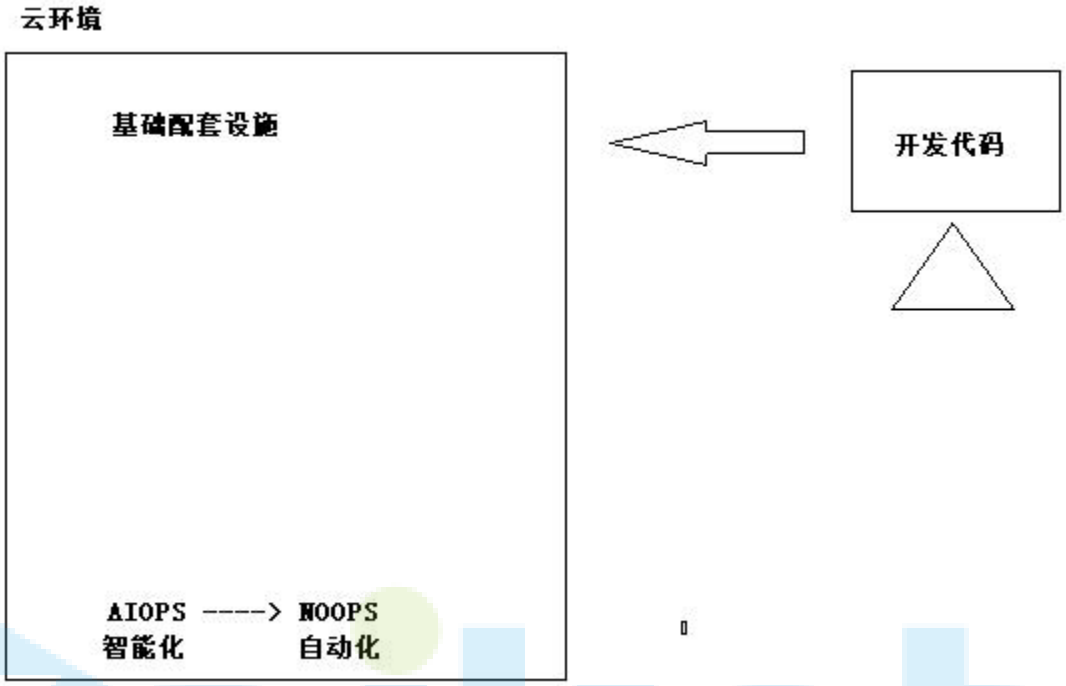
4.6 ServiceMesh

ServiceMesh 服务网格架构，云原生架构，CNCF 组织： 把服务网格架构也定义为云原生架构，ServiceMesh 服务网格架构落地实现框架是 istio,
思考： 为什么会存在 ServiceMesh 服务网格架构？？
答案： 微服务架构模式，服务调用链路，网状结构越来越复杂，导致对服务链路监控，服务限流，服务降级，服务熔断，日志监控，服务告警，负载均衡等等都会面临挑战；并且以上的这些东西和业务并没有太多的关系，但是在做开发的时候，还必须关注；架构，设计，开发这些配套设施；因此考虑这些问题，就会大大的降低的开发效率；
ServiceMesh 架构是什么样子的？？



4.7 Serverless

Serverless 架构： 面向未来的架构，无服务架构，从开发人员角度来看，不需要关心服务器，只需要关心开发即可；
例如：
把 css,js,image 部署 CDN 节点中，不需要关心 CDN 是如何存储，有多少节点，如何负载均衡的等等，这样的一种模式，就可以叫做 Serverless;


云环境
基础配套设施
开发代码
AIOPS -----> NOOPS
智能化 自动化