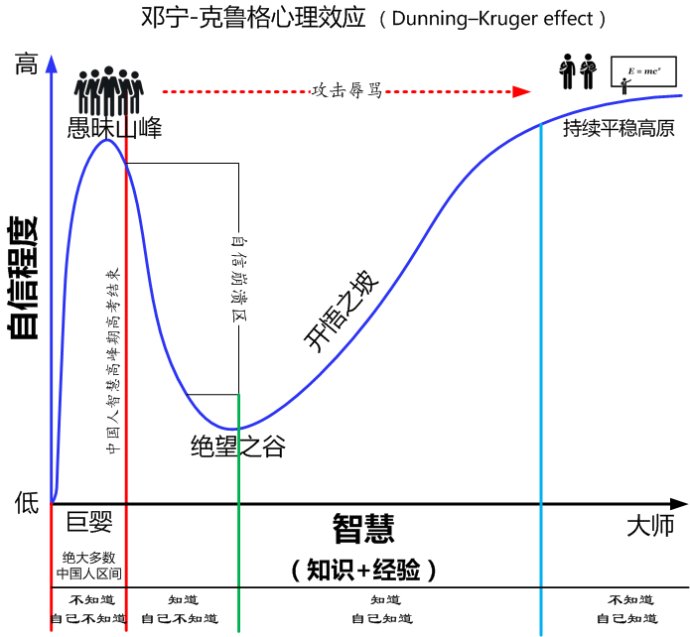
1. 邓宁克鲁格图心理效应



1. 架构完整解决方案
   1. 具体业务场景
   2. 架构如何选型
   3. 架构如何设计
   4. 架构如何折中
   5. 架构线上问题如何解决
      1. FEED系统 微信朋友圈和微博，朋友圈有5000条限制，所以可以用push,而微博没有，只能用PULL
2. 架构背后哲学思考
   1. 为什么要这样设计
   2. 其它方案为什么不优雅
      1. 用redis实现分布式锁极端情况：主从模式。两个线程同时访问，一个线程先拿到锁，这一瞬间主挂了，从还没同步，并没有这个锁。根据哨兵模式，从会提升为主。这样，另一个线程也能拿到锁，这样就会有问题
      2. 为什么？分布式锁是一个CP模式。REDIS是AP模式。用AP去实现CP，这是不可能的。
3. 提升架构高度，仅寄希望于代码层级是远远不够的。
   1. 代码解决执行力的问题，架构高度更多依赖于战略（业务洞察力）和战术问题（技术视野）
   2. 百万年薪需要的是架构高度（即架构战略和架构战术，也就是架构格局）
4. 互联网架构演进
   1. monoliths->horizontal layered和SOA->microservices->service mesh

单体架构->水平分层架构和面向服务架构->微服务架构->服务网格架构

* + 1. service mesh参考(https://www.hi-linux.com/posts/41166.html)
  1. 单体架构
     1. 应用场景
        1. 业务场景简单，功能不复杂，研发人员较少
        2. 创业公司初期
        3. 对性能（延时）要求及其苛刻。如股票中的高频交易。
     2. 缺点：耦合度高，技术选型单一，开发效率低下
     3. 如何破局
        1. 数据库存储量大破局思路
           1. 拆分

垂直拆分（分库）

水平拆分（分表），垂直虽然拆好了，但比如商品，他还是一个单表。我现在商品有一百亿了，会对性能产生很大影响，对商品表进行水平拆分，比如根据商品ID模1024来分表

* + - 1. 架构同理
         1. 垂直方向拆分（业务维度）
         2. 水平方向拆分（功能维度），垂直虽然拆好了，但比如用户模块，他还是一个单体，那我就水平再拆，分为网关层，业务逻辑层，数据访问层这样。
  1. 水平访问层架构设计与实践
     1. 网关层（与展示服务分离，和业务没有任何关系。比如日期只会传long数字，具体的format交给前端去处理）
        1. 请求鉴权：发布商品，登录鉴权

鉴权的规则一般都不会放在数据库中，而是放在配置中心。

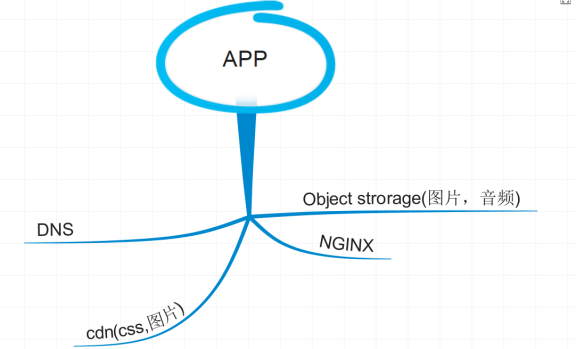
* + - 1. 数据完整性检查：数据包：定长header+变长body

不会去检查价格属性有没有填，但是会去检查uuid,sessionid,cmd,bodylength有没有填。即只会去检查通用逻辑，不会去检查具体语义。

* + - 1. 协议转换：json->hashmap(string,object),这里就可以用pb把hashmap做序列化。

一旦APP到了网关层，就不需要HTTP短连接了。传输协议应该是TCP的（比如RPC就是over tcp的），数据协议用二进制的，往往我们用PB（Google Protocol Buffer 序列化协议，序列化出的二进制远比JAVA序列化要短，各语言支持）

* + - 1. 路由转发：根据CMD转发到不同业务逻辑层
      2. 服务治理：降级，熔断等
      3. 常见的网关有zuul,springcloud gateway,nginx,kong,nodejs,自研
    1. 业务逻辑层：业务逻辑判断
    2. 数据访问层
       1. CRUD
       2. ORM，如mybatis
       3. sharding(分库分表)最复杂的，有时候还分不了，那我们可以用newsql来替换这个功能，把这个功能推到DB层
       4. 屏蔽底层存储差异性
    3. 异步架构。目的：提升吞吐量。手段：消息队列
       1. 可以在两层之间加上MQ，就可以达到异步
       2. 为什么MQ写起来比直接写DB快?MQ 99.9%是通过append顺序写,db99.9%是随机写
       3. MQ加到网关层和业务逻辑层
       4. 适用场景：读绝对不用MQ,写如果是弱一致性，需要进MQ
    4. 适中的架构（MVC已经OUT了）
       1. 同步：四层：网关层--业务逻辑层--数据访问层--数据存储层
       2. 异步：五层：网关层--异步消息队列层--业务逻辑层--数据访问层--数据存储层
    5. 架构全貌



访问css,先去找cdn缓存,如果不存在,那就直接去nginx获取。

访问图片,先去找cdn缓存,如果不存在,那就直接去object storage（对象存储）获取。

（对象存储：规模比较小的fastdfs就可以了，规模比较大的用ceph.如果用ceph,运维团队需要比较强大）

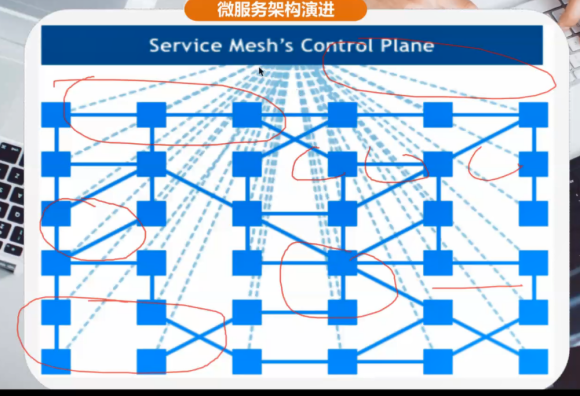
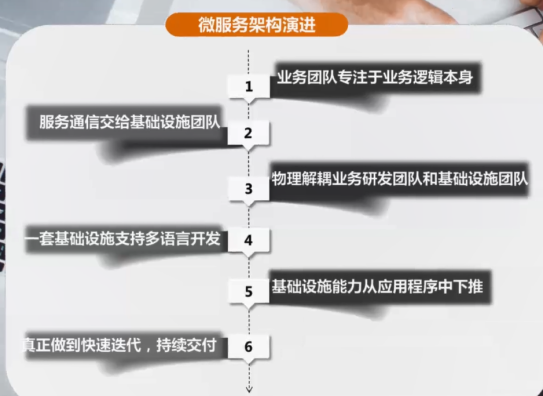
* 1. 面向服务架构:就是垂直拆分
  2. 微服务架构
     1. 本质
        1. 既按照水平拆，也按照垂直拆，最验证的就是垂直怎么拆
        2. 是一个业务架构
        3. 也是一个组织架构
     2. 配置中心-- 已经水平和垂直拆好的一大堆微服务--注册中心

配置中心用apollo,注册中心用zookeeper是不行的，eureka可以

* + 1. 适用场景
       1. 需求层面：变更频繁（内部OA,ERP等内部使用的系统需求往往没那么频繁，一个季度+变一次的话意义就不大）
       2. 性能层面：不需要实时性太高
       3. 数据一致性层面：最终一致性
    2. 目的：项目快速迭代和项目持续交付
    3. 案例



* + 1. 微服务架构不是银弹
       1. 基础设施组件升级困难，影响基础设施团队的交付能力和交付速度。业务团队不帮你升级基础设施（如降级V1->降级V2）,你也没办法
       2. 多语言编程之间通信问题，业务每种语言一套基础设施，成本太大



* 1. service mesh架构



