**学习新技术方法**

1. 快速寻找新技术代码结构，通过查阅资料，查找出现频率较多的代码

2. 通过考试，将技术进行整合

3. 对于难点技术，根据类型的不同，分类掌握，

- 偏理论型知识，通过碎片化时间学习，最后整体消化

- 偏实践行知识，需要在时间前有一定的理论基础，通过实践进行验证

4. 通过技术分享，营造学习氛围，进行技术沉淀，增加影响力

**JVM提高内存使用率**

**软引用定义**

如果一个对象只有软引用，而当前虚拟机堆内存足够用，就不会被GC回收，反之，软件用指向的对象就会被回收；

**弱引用定义**

如果一个对象只有弱引用，不管当前内存够不够用，都会被回收；

**使用场景**

软引用实现在内存不够用时，自动释放一部分对象内存；

弱引用可以实现自动更新；

**TiDB架构**

1. 目标

- 分布式数据库，实现水平扩展

- 高可用，强一致性

- 事务 ACID 要求：原子性（Atomicity）、一致性（Consistency）、隔离性（[Isolation](https://baike.sogou.com/lemma/ShowInnerLink.htm?lemmaId=70806469&ss_c=ssc.citiao.link)）、持久性（Durability）

2. 架构迭代

支持sql、支持事务(内存中b-tree) -> 持久化数据库 -> 添加协议层，进行协议转换 -> sql<->kv 拆分，抽取kv接口 -> 优化sql层，添加支持类型 -> 分布式存储引擎（基于HBase） -> 优化引擎（coprocessor API Filter） -> 自己开发存储引擎TiKV（语言使用R，无gc | 单机引擎 RocksDB | 副本复制roft | 通信GRPC）



图1 TiKV

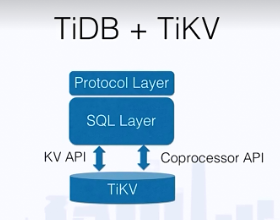


图2 TiDB修改数据引擎后的架构

-> sql层优化 -> kv引擎优化（通过PD实现负载均衡） TiSpark

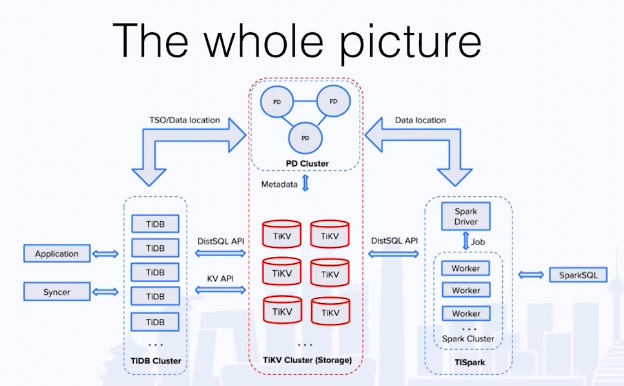


图3 TiDB 整体架构

3. 经验

- make it right, then make it fast

- test matters test for test test 最高优先级

- tools matters 实现自动化

- metrics system matters 监控指标 监控工具

4. 开源

- 开源后才能迅速提升价值

**如何设计低耦合、易复用的软件架构？**

**框架的特点：**

使用框架开发一个应用程序，开发者无需在程序中调用框架的代码

**依赖倒置原则：**

高层模块不应该依赖低层模块，二者都应该依赖抽象

抽象不应该依赖具体实现，具体实现依赖抽象

**举例**：JDBC

**依赖倒置设计原则：**

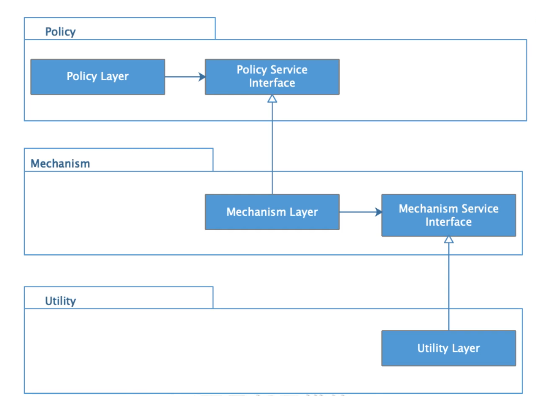


图4 依赖倒置原则

**注：高层模块拥有接口，低层模块实现接口，不是高层模块依赖低层模块的接口**

适用于一个一个类向另一个类发送消息

tomcat\spring都是用的这种方法来实现的。

依赖倒置也称为**好莱坞原则，don't call me, I'll call you**

**高可用、弹性动态的金融机移动架构在蚂蚁金服的演进之路**

大杂烩-》架构分层

向3个方向优化：团队协助、研发效率（接口实现分离、业务分治、微应用化）、性能与稳定(框架层面：统一开发规范、引用pipeline、AOP切面监控，向下突破：深入优化研究底层，基础指标跟踪优化：内存、存储、电量等)



图5 分层架构

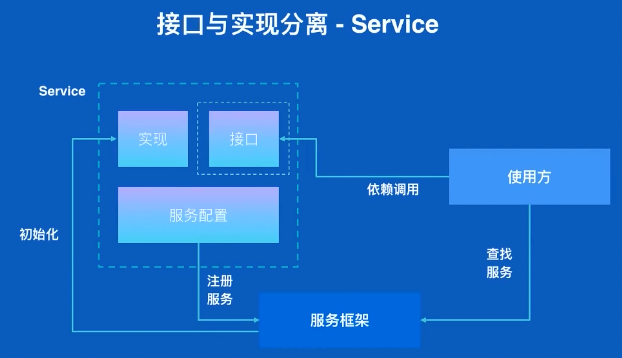


图6 service层实现接口实现分离

网络协议优化：

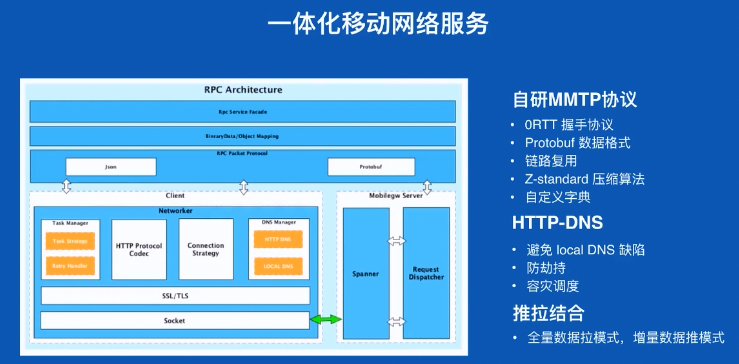


图7 网络优化

弹性、动态要求：研发模式发生改变

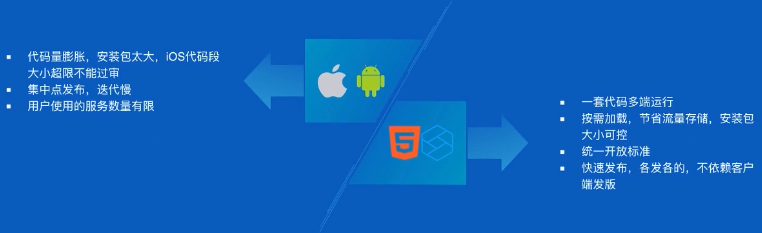


图8 动态研发模式

**WEB优化手段：**

前后端分离、差量更新、推拉结合、容错补偿、android独立浏览器内核、深度定制组件、全面监控

**线上高可用：**

1. 快速发布

- 智能灰度发布，多种升级策略

- 增量差分离线包更新能力

- 系统高性能保证

2. 实时监控

- 监控指标（闪退、流畅度、电量等）

- 上报策略

- 上报方式（自动上报、周期性检查上报、诊断指令驱动上报）

- 诊断分析（用户行为、APP日志）

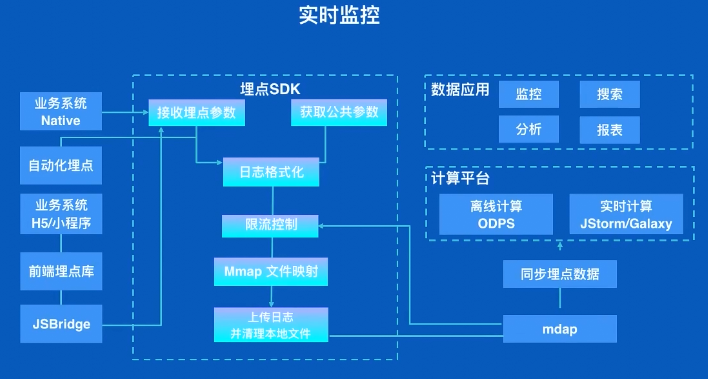


图9 监控架构

3. 容灾处理

- 故障隔离

- 流量熔断

- 自动恢复

- 动态能力修复

**传统金融银行手机APP监控较为空白，可借鉴蚂蚁mPaas**

**如何设计一个异步无阻塞的反应式编程框架**

针对各个用户请求，传统的 Web 应用会创建一个线程，而在用户的请求服务周期内，整个线程都是独占的。任何引起请求操作的阻塞，都会导致整个线程的阻塞。

由于线程阻塞导致的系统宕机，目前流行的解决方案是限流、降级、消息队列异步化处理、动态扩容web服务器规模

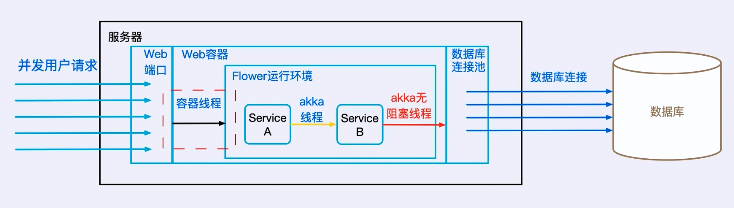


图10 反应式框架Flower

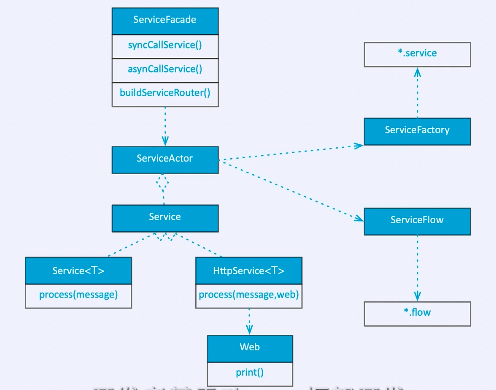


图11 Flower核心类图

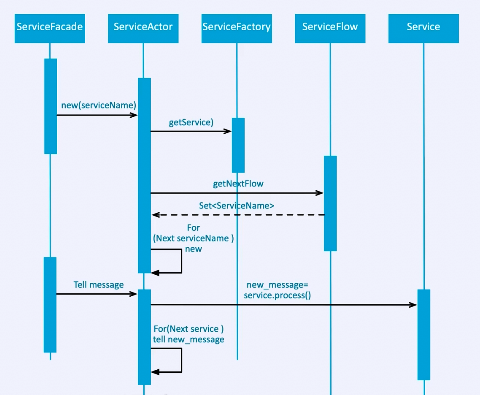


图12 Flower时序图

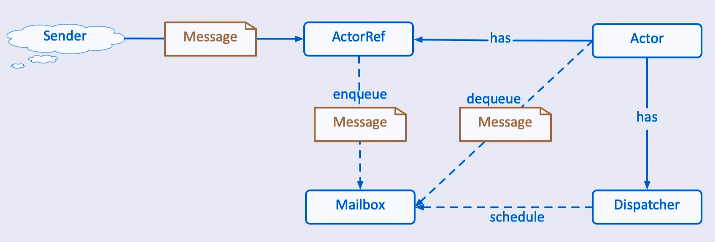


图13 AKKA actor 异步通讯图

反应式系统的特征：

1. 即时响应，应用非阻塞调用

2. 回弹性，应用实现容错和自动修复

3. 弹性，根据请求量调整资源使用

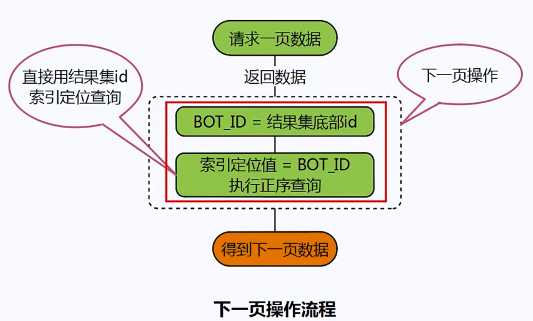
4. 消息驱动

**数据库遇到瓶颈属于数据库问题吗**

1. 数据的偏移量影响查询效率

2. 未利用上索引影响查询效率

3. 页面处理逻辑保留上一页和下一页的索引偏移





**关键：要使用索引定位数据位置**

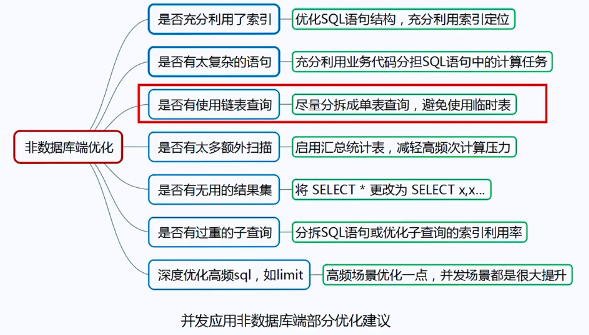
4. count/sum/groupby 使用不当会影响数据量查询效率

可以通过添加统计表来提高效率

5. 使用链表查询也会影响效率

可以通过拆分为单表查询，修改业务逻辑来优化

**优化建议：**



**JVM问题定位经典案例**

1. 类加载死锁

- 一般情况，使用jstack，查看dump

- 特殊情况执行 jstack -m pid （不推荐线上使用，可能导致进程挂起）

问题出现原因：jdk7之前 loadClass的时候会加锁，并发的时候可能导致死锁

2. FinalReference堆积

描述：

jmap -histo 看到 java.lang.ref.Finallizer对象，但是自己的代码中定没有用这个

执行完finalize对象执行完后的下一个GC周期才会回收

3. 堆外内存泄漏

描述：

heap利用率很低，但是出现了OOM或者是FullGC

定位：

如果是java溢出导致，可以使用btrace跟踪 DirectByteBuffer 的构造函数去定位

非java层面问题，可以使用google的perftools来分析具体哪里分配，比如压缩和解压缩

4. YGC不断拉长

描述： YGC时间不断变长

影响YGC的主要因素：

1. YGC三步骤： mark&copy&swap

2. mark和copy是同步进行的

3. 被mark的对象多少直接影响了耗时的多少

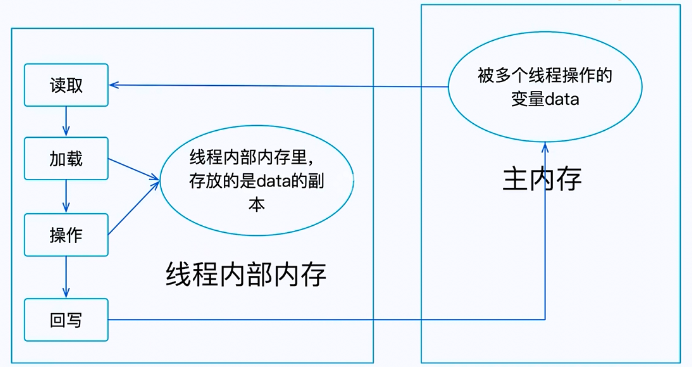
免费工具

1.xxfox jvm参数自动调优

2.免费jvm问答社区 JCafeBabe

、

**如何从Java线程内存模型角度分析线程是否安全**



“线程不安全”的对象在多线程环境下可能会出现“被抢占”的现象，所以叫不安全。

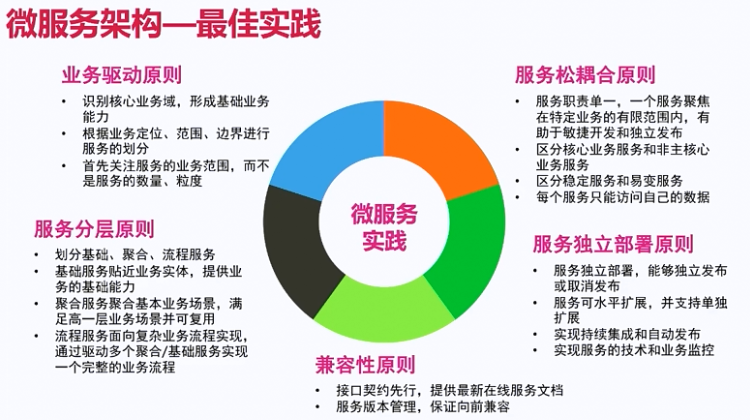
线程安全是需要代价的，大多数场景都是单线程运行，所以可以根据情况使用线程不安全的数据结构，性能更高。在多线程并发场景下且要保证数据正确性，需要使用线程安全对象

**注： volatile在一定场景下能够提升性能，但是不能保证线程安全**

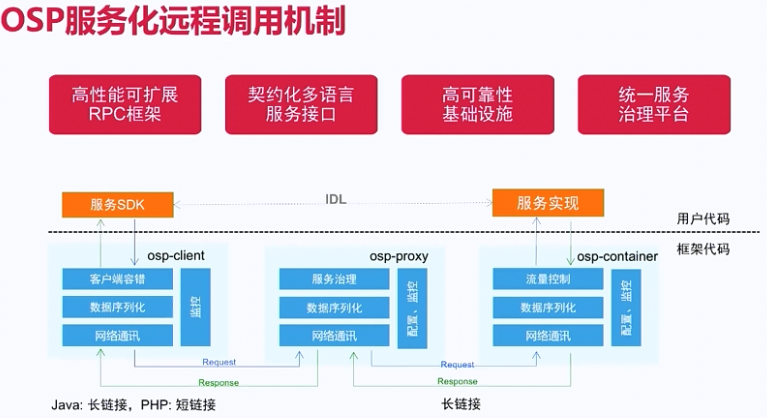
**唯品会微服务架构演进之路**

1. 微服务架构演进



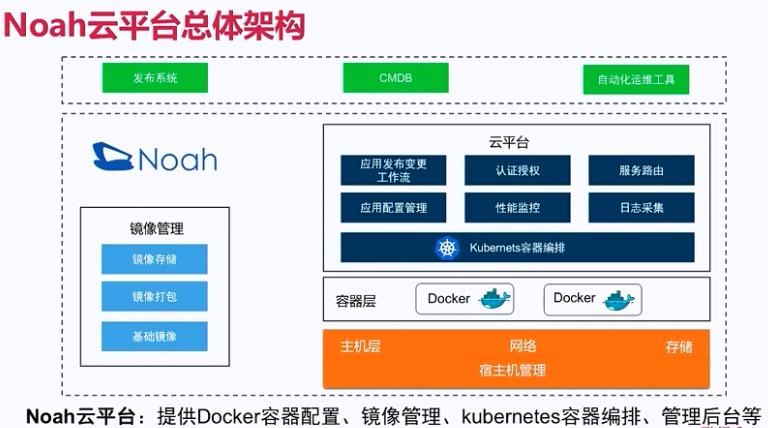


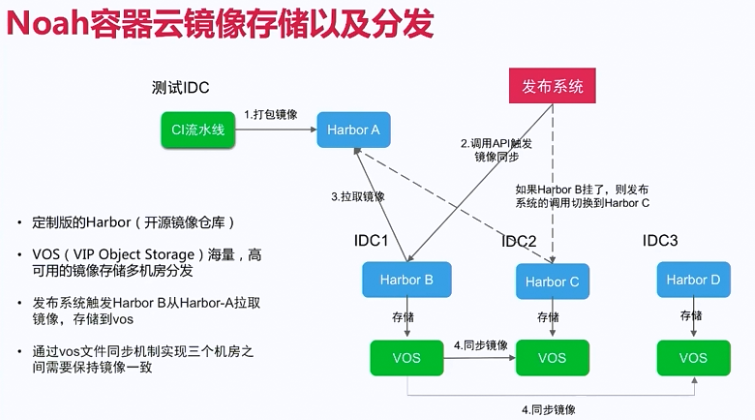
2. 微服务基础中台建设



多机房部署要点： 中央集群proxy

3. 基于kubernets\Docker打造云平台





**容器编排规则**：

节点选择：

剩余的CPU/内存

同域“尽量”跨机器，跨机架部署

高可用保证：

端口/health check url 定时探测

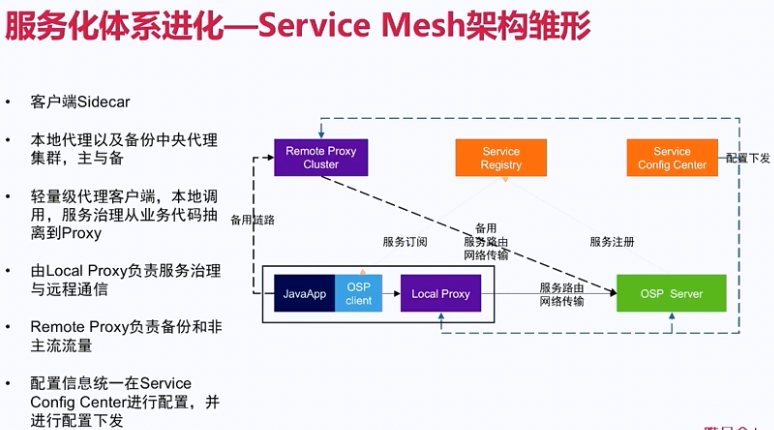
探测失败，同节点重建实例

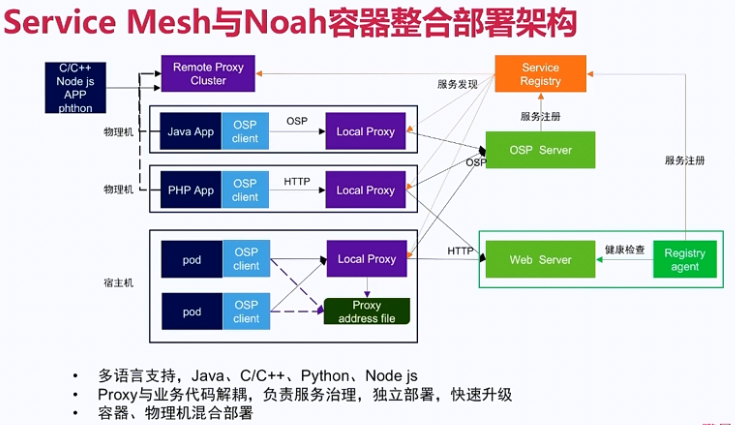
节点失效，新节点重建实例

自动扩容使用 **HPA算法**

容器发布使用**金丝雀**发布、**灰度**发布（设置不同的权重值，调整到进行灰度发布的机器的请求流量）

4. Service Mesh架构





**字节跳动线上性能监控体系的建设**

内存监控的功能点：

内存溢出OOM

大对象

内存触顶（使用率达到阈值）

内存泄漏

卡顿指标的建立：

基于FPS，建立卡顿指标，配合慢函数解决卡顿问题

缓存空间恶化原因：

人员：开发人员多，多团队协同，沟通成本

代码：历史代码，缓存失效bug，兼容问题

功能：需求迭代快，功能下线未处理对应缓存，业务场景复杂导致收敛困难

测试：不容复现，非真实用户环境

缓存监控：

大文件

大文件夹

过期文件

异常流量监控：

http 统一监控

webview 定义自研监控

socket 业务记录

视频 kafka下游数据

直播 kafka下游数据

单点数据问题：

日志库、日志流

**devops工具**

git 分布式版本控制工具

docker 容器

selenium 测试架构

jenkins 自动集成服务器

Ansible 配置工具

puppet 配置管理工具

nagios 监控工具

Chef 部署自动化工具

**为什么需要DDD**

业务微服务化，加剧了“散弹式修改”

微服务划分，需要注意功能维度的划分。需要使用ddd协助。

微服务和ddd是互补关系

微服务关注应用程序行为的分割

**总结**：DDD和微服务协同知道业务进行合理的拆分

**C端服务器的渲染和性能提升--React和SSR**

为什么需要SSR？ SEO C端新能 统一技术栈 （Umi SSR/Rax SSR/Servless SSR方向）

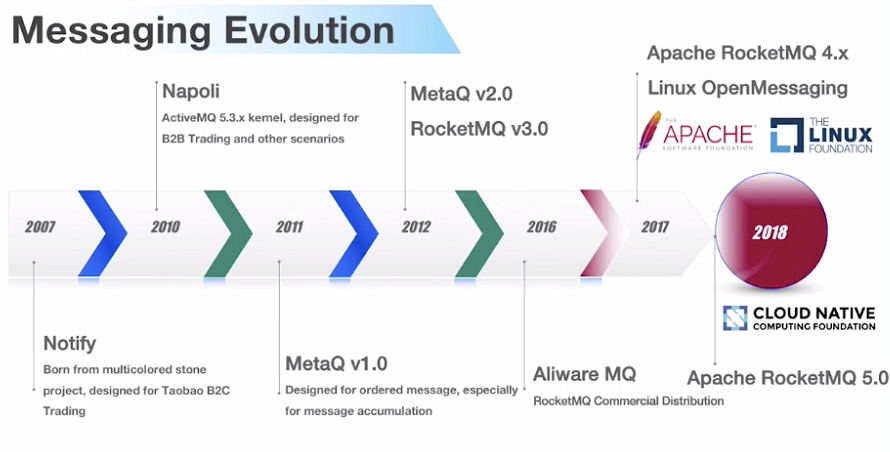
从CSR到SSR的演进之路

bigpipe-> webpack \ API和模块拆分 预渲染 -> React SSR

API和模块拆分

优化实践 编辑&缓存 压测

**Apache RocketMQ**



1. 用处

EAI 企业应用级集成

系统解耦

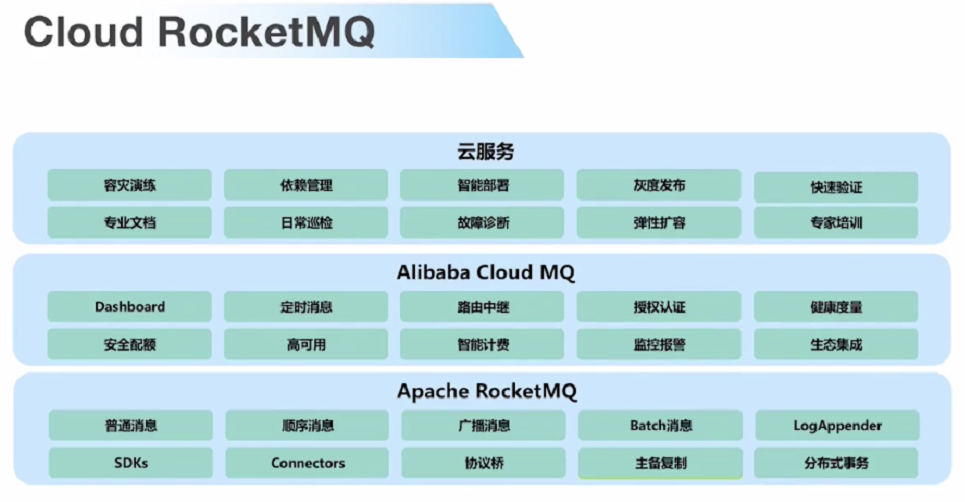
serverless

数据复制

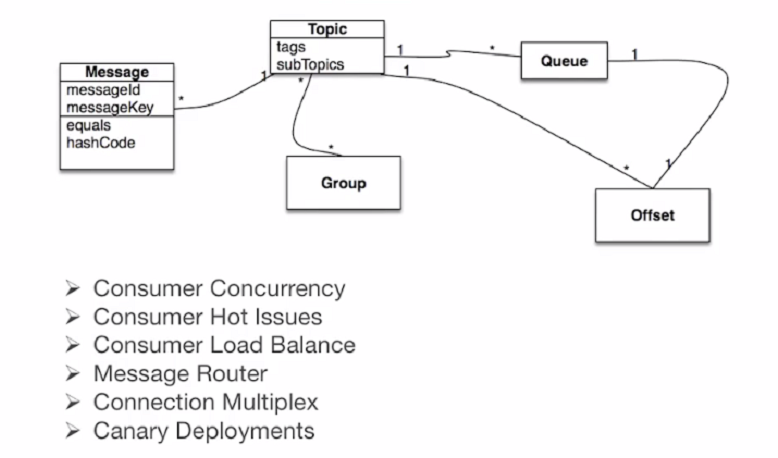
reactive streaming

事件驱动

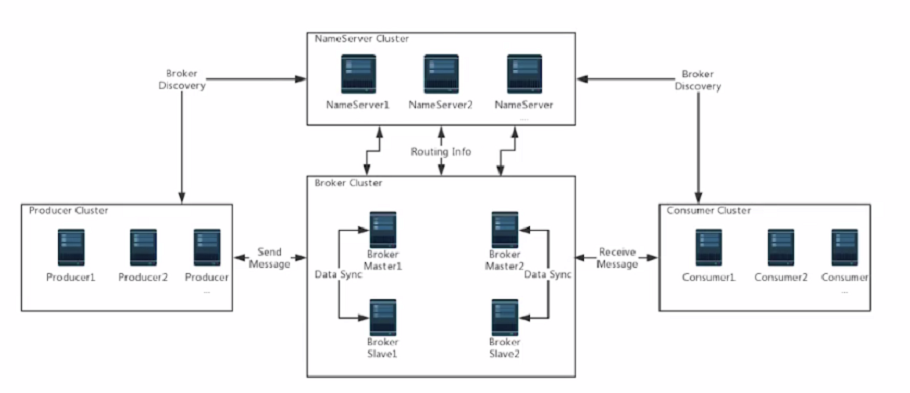
2. 架构



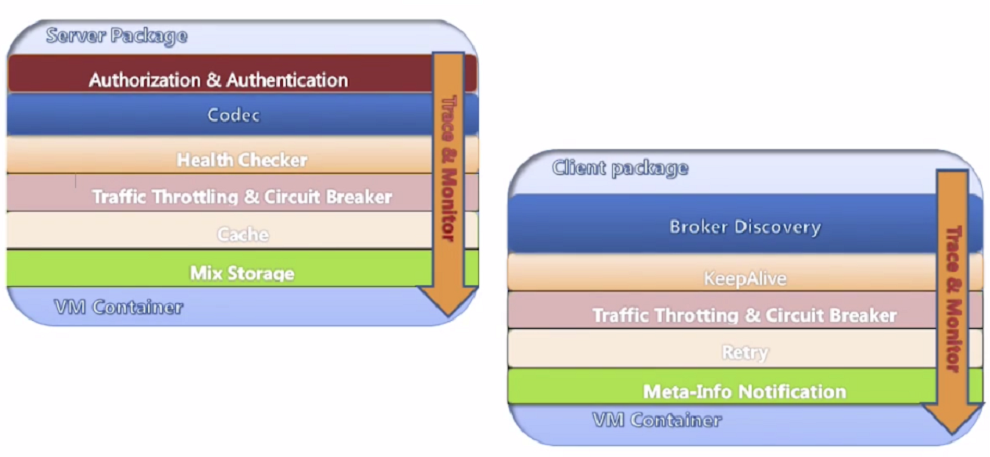
领域模型



拓补图



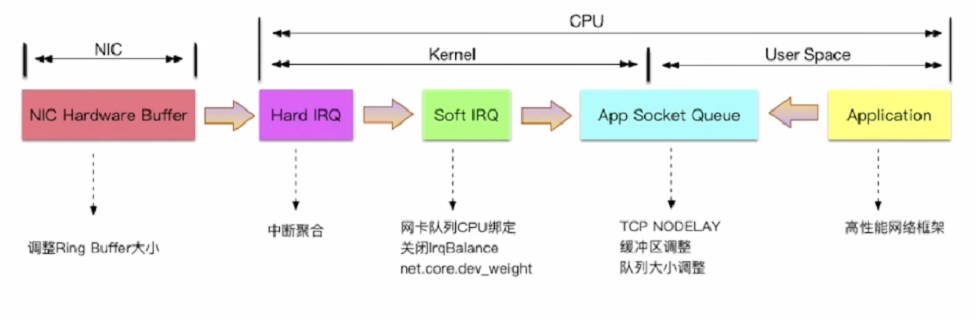
分布式架构



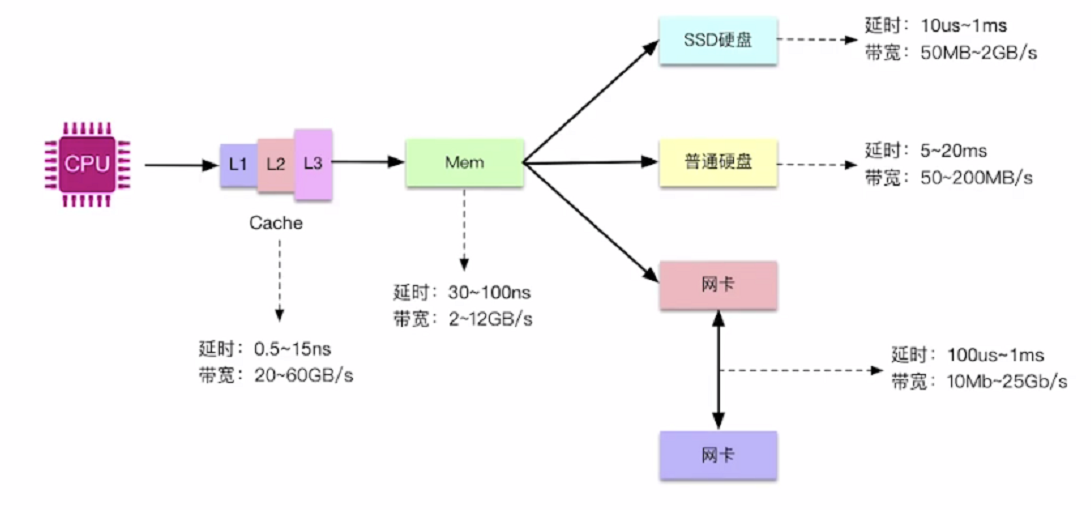
devops 平台

3. 高性能和低延时

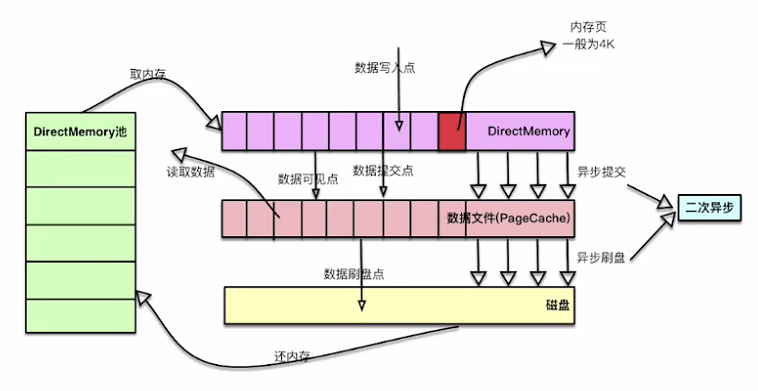
请求链



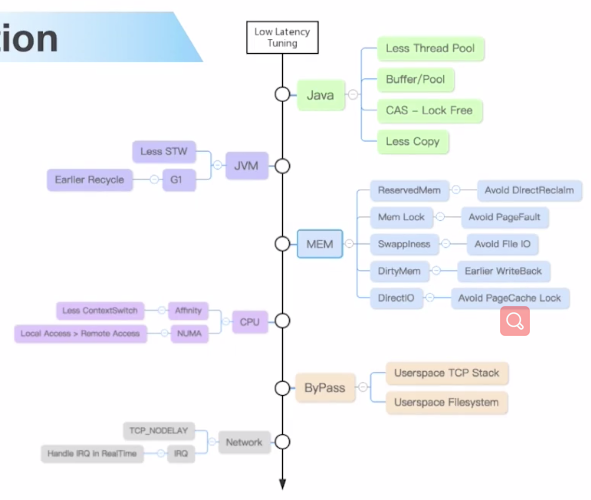
低延迟



page Cache 低延迟优化

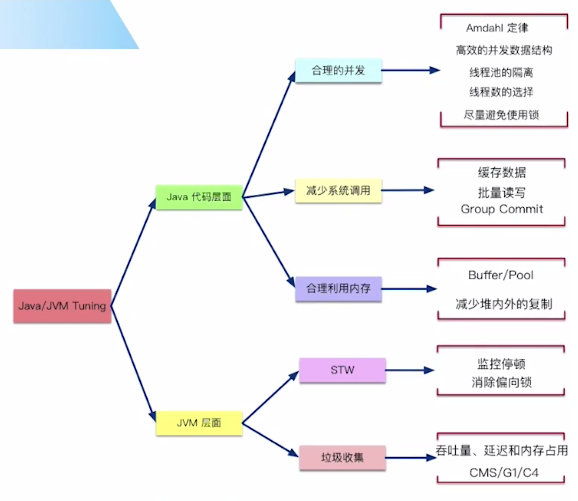


解决方案



优化方案参考：

1、jvm方面



2、linux内核方面

