# 微服务搭建方案

微服务与面向服务架构SOA的区别

SOA架构是一种粗粒度，松耦合的服务架构，其更多的是强调异构系统之间的服务通信和解耦合，如WebService。而微服务架构强调的是系统按业务边界做细粒度的拆分和部署。

Dubbo与Spring Cloud都是框架和组件，SOA和微服务是架构思想，他们不是同一个层面的概念。可以说Doubbo很好地支持了SOA和微服务架构

## springcloud + k8s搭建微服务

平台集成Service Mesh功能，客户搭建起微服务来很简单

# 分布式相关问题及解决方案

## 分布式事物

分布式事物的存在，就是解决数据不一致的情况。分布式系统中有这么一个广为流传的理论：CAP 定理对于一个分布式计算系统来说，不可能同时满足以下三点：

•一致性（Consistency）

•可用性（Availability）

•分区容错性（Partition tolerance）

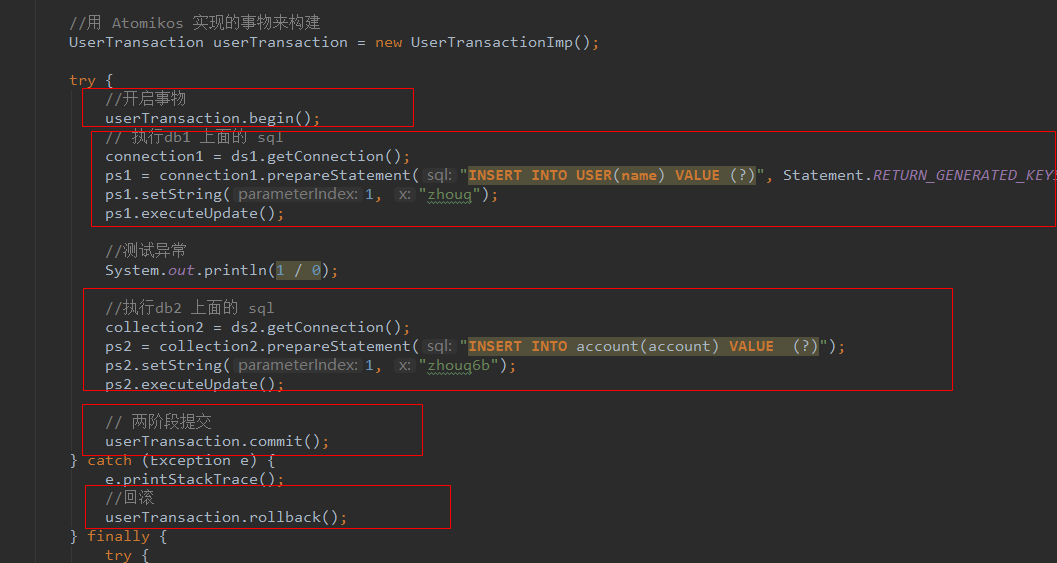
由于分区容错性（P）是必须保证的，那么我们分布式系统就更多是在一致性（CP） 和可用性（AP）上做平衡了，只能同时满足两个条件。其实，大家想想，ZK 是不是就是严格实现了 CP ，而 Eureka 则是保证了 AP。其实分布式事物强调的就是一致性。

**几种分布式事物解决方案**

**2PC**

在说 2PC 之前，我们先了解一下 XA规范 是个什么东西？

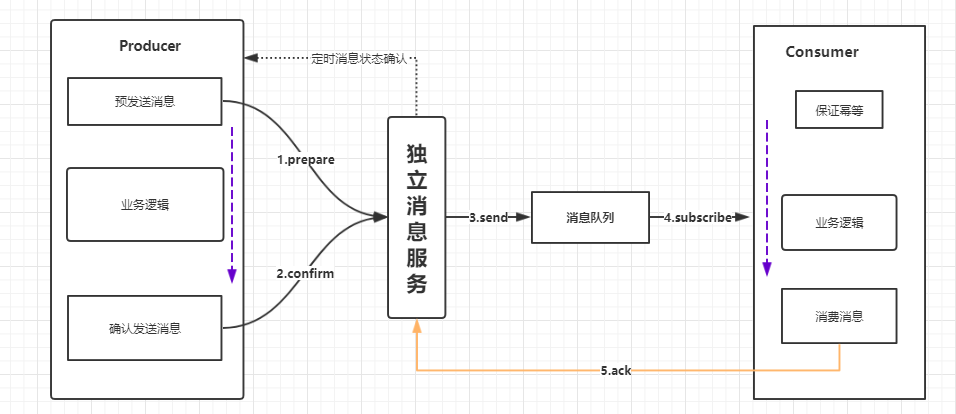
XA规范描述了全局的事务管理器与局部的资源管理器之间的接口。XA规范的目的是允许多个资源（如数据库，应用服务器，消息队列，等等）在同一事务中访问，这样可以使ACID属性跨越应用程序而保持有效。



XA 使用 两阶段提交（2PC） 来保证所有资源同时提交或回滚任何特定的事务。（2PC即两个事物分别预提交，再确认提交，如果报错则回滚。）

**可靠消息最终一致性方案来解决分布式事物的问题**

这个方案，就有消息服务中间件角色参与进来了。



后面的系统，一定要处理 幂等，重试，日志这几个东西。

**TCC 方案**

TCC 的全程分为三个阶段，分别是 Try、Confirm、Cancel：

1.Try阶段：这个阶段说的是对各个服务的资源做检测以及对资源进行锁定或者预留

2.Confirm阶段：这个阶段说的是在各个服务中执行实际的操作

3 .Cancel阶段：如果任何一个服务的业务方法执行出错，那么这里就需要进行补偿，就是执行已经执行成功的业务逻辑的回滚操作

还是以转账的例子为例大概思路就是这样的：

1.Try 阶段：先把A 银行账户先冻结 1 块，B银行账户中的资金给预加 1 块。

2.Confirm 阶段：执行实际的转账操作，A银行账户的资金扣减 1块，B 银行账户的资金增加 1 块。

3.Cancel 阶段：如果任何一个银行的操作执行失败，那么就需要回滚进行补偿，就是比如A银行账户如果已经扣减了，但是B银行账户资金增加失败了，那么就得把A银行账户资金给加回去。

## 分布式服务隔离

**按种类隔离**

按照服务种类隔离要从两个纬度来说:即服务提供方和服务调用方！假设我们一个系统有三个服务:订单服务，库存服务，支付服务！服务提供方这个角度而言，每一个服务乃至其对应的数据库，给一个服务器部署。服务调用方这个角度而言，如果不针对被调服务做服务隔离，一个被调服务出问题，就将导致调用方服务不可用！目前业内有信号量隔离和线程池隔离两种隔离方式。

线程池隔离方式，即给每个微服务都初始化出一个线程池，如下图所示，给订单服务和库存服务都初始化出一个线程池，不使用Tomcat线程池中的线程直接调用，而是用相应线程池中的线程去调用！如果此时库存服务不可用了呢？库存服务线程池会被迅速塞满，此时后面进来的新请求发现库存服务线程池满啦，于是乎就不去调库存服务，直接返回！实现方式可以了解一下Hytrix、Sentinel、以及Resilience4j

**按用户隔离**

租户和用户：

用户: 一个环境/系统的一个使用者即该环境/系统的一个用户。

租户:用户从某种粒度上被分到若干组内，每组成为一个租户。

组可以这么理解:用户根据一定的特征去做分组，比如是VIP的一组，不是VIP的一组。

那么所谓的用户隔离，就是按照不同的分组形成不同的服务实例。

有如下三种方式！

方式一:每个租户有独立的服务和独立的数据库

方式二:每个租户有共享的服务和独立的数据库

方式三:每个租户有共享的服务和共享的数据库

## 内存泄露排查思路

1、使用top命令，查看占用内存较高的进程ID。

发现PID为3331的进程占用内存 90.6%。而且是一个Java进程，基本断定是程序问题。

2、使用jmap查看内存情况，并分析是否存在内存泄露。

jmap -heap 3331：查看java 堆（heap）使用情况

jmap -histo 3331：查看堆内存(histogram)中的对象数量及大小

jmap -histo:live 3331：JVM会先触发gc，然后再统计信息

jmap -dump:format=b,file=heapDump 3331：将内存使用的详细情况输出到文件

得到堆dump文件后，可以进行对象分析。如果有大量对象在持续被引用，并没有被释放掉，那就产生了内存泄露，就要结合代码，把不用的对象释放掉。

## 设计一个消息中间件

让你从整体架构，核心流程，数据结构，等各个层面来考虑，你会如何完成这个设计？

首先第一个点，如果有人生产了消息，你作为一个消息中间件，应该如何存储这个数据？你会存储在内存里呢？还是存储在磁盘文件里呢？或者两者都同时共存？你的消费模型是什么样的？比如说一个queue里的数据，是会均匀分配给消费者的各个实例呢？还是会怎么做呢？

接着你应该考虑第二个大的问题，就是你的消息中间件肯定会遇到每天TB级海量数据高并发高吞吐写入的场景。你的数据是不是要分布式的存储？此外你还需要考虑，你的数据分片是不是要可以支撑扩容？

再者，数据宕机场景下的高可用架构。一般分布式系统实现高可用架构，都是采用多副本冗余机制。

还有就是支持数据不丢失的ack机制。

## 抢红包逻辑

方法1：将剩余的红包金额平均再乘二，取这个区间的值。

方法2：取一线段进行切割，依此给抢红包的人。按平均切割，然后将一边的数字随机立减到另一边，再打乱顺序。

## 怎样处理高并发的

高并发就是指网站在同一个时间可以同时处理多个用户请求。要处理高并发，最基本的就是：使用高性能服务器、高性能数据库、高效率编程语言、还有高性能的Web容器。其次就是(1)HTML静态化(2)图片服务器分离(3)数据库集群、库表散列(3)缓存机制(4)镜像(5)负载均衡

## 分布式集群下如何做到唯一序列号

1. 数据库自增长序列或字段:每个Master库设置的起始数字不一样，步长一样，可以是Master的个数。比如：Master1 生成的是 1，4，7，10，Master2生成的是2,5,8,11 Master3生成的是 3,6,9,12。

2. UUID

3. UUID的变种:

1）为了解决UUID不可读，可以使用UUID to Int64的方法。

2）为了解决UUID无序的问题，NHibernate在其主键生成方式中提供了Comb算法（combined guid/timestamp）。保留GUID的10个字节，用另6个字节表示GUID生成的时间（DateTime）。

4. Redis生成ID

5. Twitter的snowflake算法

6. 利用zookeeper生成唯一ID

分布式锁目前比较常用的几种方案：

基于数据库实现分布式锁

基于缓存（redis , memcached）实现分布式锁

基于zookeeper实现分布式锁

## 分布式系统优化

**分布式系统优化原则：**

减少跨机网络交互

尽可能带sharding key

分页优化

减少数据读写热点

切分颗粒度尽可能细

减少锁开销

尽可能规避分布式事务

尽一切可能利用单机资源

单机事务

单机join

好的存储模型，就是尽可能多多做到以下几点：

尽可能走内存

尽可能将一次要查询到到数据物理到放在一起

通过合理到数据冗余，减少走网络到次数

合理并行提升响应时间

读取数据瓶颈，可以通过加slave节点解决

写入瓶颈，用规则sharding和扩容来解决

全部信息索引：

http://weibo.com/1765738567/C72pRgl83

一致性选择

无主机方案

有主机方案

分布式存储

key-value多机扩展

CAP和分布式系统多一致性

## 服务治理

服务治理的核心是：服务性能KPI统计、告警、服务健康管理、灵活的弹性伸缩策略、故障自动迁移、服务限流和服务降级等多种治理手段，保障服务高质量运行

常见的服务治理：服务降级、服务流控、服务动态扩展、超时控制、优先级调度、负载均衡策略调整、分组调整、等

## 高可用系统常用解决手段

1. 系统拆分，读写分离
2. 降级：当故障发生后，怎么减小故障所造成的损失，让系统服务能力不直接降到0

2.1限流

2.2异步调用：弱依赖调用通过消息中间件的方式来实现。

2.3同步调用设置超时时间

2.4失败重试