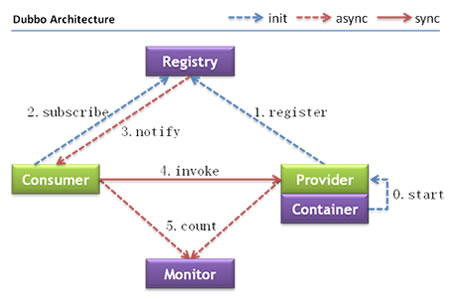
Dubbo基本原理和请求过程

结构



Dubbo序列化：

hessian json

和springcloud区别：

dubbo主要用dubbo协议 springcloud用http协议(rest风格)

可以在监控中心配置负载均衡策略

**中台**

大中台（技术中台->业务中台），轻前台

**微服务系统设计的要点**

服务拆分

注册中心

服务发现

服务消费

负载均衡

路由管理

扩容缩容

熔断机制

**微服务拆分**

业务服务：订单，结算，分销，酒店价格库存操作…

基础服务：酒店静态数据查询，价格查询…

聚合服务（提供给用户页面调用）：酒店详情页面，同时调用多个业务服务接口..

**RPC与REST区别**

|  |  |  |
| --- | --- | --- |
|  | RPC | REST |
| 耦合性 | 强耦合 | 松耦合 |
| 消息协议 | 二进制(protobuf IDL) | 文本 Json/XML |
| 通讯协议 | TCP | HTTP |
| 性能 | 高 | 低于RPC |
| 对外开放 | 需要转成文本的协议 | 可直接对外 |
| 客户端 | 需要单独引入客户端 | HTTP客户端 |

**分布式事务**

产生场景：

库存和订单两个独立的微服务，每个微服务维护了自己的数据库。

先扣减库存，然后生成订单

结果库存扣成功 订单生成失败

解决方案

1. 两段式提交

引入事务协调者，分为准备阶段和提交阶段

准备阶段：参与者执行事务，但不提交，参与者返回协调者是否执行成功

提交阶段：若所有参与者都提交成功，则协调者发起命令提交事务。若有参与者执行失败，则都回滚(Undo Log)

问题：

1. 性能不高，各节点都占用数据库资源，需要等所有节点都成功，才能提交事务，释放资源
2. 协调者单点故障，会导致参与者卡住，无法继续下一步（可以引入超时时间，超过超时事件自动提交，即三段式）
3. 数据不一致。 如第二阶段，网络故障导致某个参与者没有收到提交的指令。
4. TCC事务(通过业务代码实现)

Try Confirm Cancel

Try阶段

锁定资源，系统监测

Confirm阶段

对业务系统做确认提交，默认如果try成功，则confirm一定成功

Cancel阶段

2如果失败了，通过对应的Cancel操作回滚补偿

优点 流程简单 缺点 一致性差，要写很多补偿代码

1. 本地消息表

服务a在处理业务逻辑的时候，同时发送一条消息，两者位于一个事务中。

然后服务b收到消息，开始处理业务，如果处理失败，则返回失败，服务a回滚

优点：避免了前面分布式事务的架构，实现最终一致性

缺点：发消息耦合到业务逻辑中

1. 事务MQ

RocketMQ支持

业务方先发一条消息，状态为prepare。然后执行本地事务。成功后修改消息状态

消息中心根据消息状态，决定是否转发到下一个业务方进行处理

**秒杀系统注意问题**

1. 库存问题 放入redis操作,通过自减命令扣减
2. 秒杀系统及页面尽量独立部署
3. 使用CDN
4. 限流措施到位
5. MQ削峰，排队
6. 异步处理
7. 可扩展
8. 良好的熔断机制