系统架构说明书

目录

[系统架构说明书 1](#_Toc480038036)

[前言 3](#_Toc480038037)

[当前系统总体架构 3](#_Toc480038038)

[系统设计的不足 4](#_Toc480038039)

[系统运行时的一些问题 5](#_Toc480038040)

[重新设计 5](#_Toc480038041)

[总体架构 6](#_Toc480038042)

[API网关 7](#_Toc480038043)

[服务注册和发现 8](#_Toc480038044)

[熔断器 9](#_Toc480038045)

[日志收集 9](#_Toc480038046)

[缓存 9](#_Toc480038047)

[消息中间件（MQ） 10](#_Toc480038048)

[服务组件 11](#_Toc480038049)

[存在的风险 12](#_Toc480038050)

#### 版本

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **日期** | **版本** | **状态** | **修订说明** | **修订人** |
| 2017-04-05 | v0.1 | 初稿 |  | 陈志国 |
| 2017-04-11 | V0.2 |  | 1. 定时任务移至到服务组件（微服务）那一层 2. 微服务间通信通过服务发现找到对应的服务，调用RESTful接口通信 | 陈志国 |
| 2017-04-11 | V0.3 |  | 修改api网关设计说明 | 陈志国 |

# 前言

当前后台服务，包括电商系统测试运行已有一段时间，出现了一些不稳定和性能上的问题，但有不能准确定位问题，需要对系统进行分析，找出原因，进行优化和改进。

# 当前系统总体架构

采用Spring Cloud微服务框架进行开发，Maven做工程管理，docker部署。图一为整理出的架构图。



图一

## 系统设计的不足

* API网关没有负载均衡。
* Zookeeper服务没有做负载。
* Java微服务没有熔断措施。
* 对分布式事物没有做处理。
* 可通过聚合层直接访问微服务，也可通过api网关访问微服务，调用方法不统一，访问控制不统一。
* 数据持久层采用JdbcTemplate，数据映射不太方便，开发效率低。安全性不好控制，容易出现sql注入的安全问题。
* 日志收集不统一，不能追溯问题。
* 对服务划分不够合理，如：支付服务和订单没能分开。

## 系统运行时的一些问题

* 系统不太稳定。
* 汇聚层采用PHP开发与Java服务端的沟通不够方便，同时日志收集也不统一。
* REST API调用没有统一的返回值，没有错误表示，汇聚层出错不能明确问题。
* 没有分库，一些服务之间耦合度增大，并不利于部署。
* 联营、众包和eboss通过电商后台登录，用户、商家通过SOA api网关登录。

# 重新设计

分析当前系统的不足和运行时的问题，首先重新评估了Spring Cloud和Dubbo/DubboX这两个微服务框架。认为Spring Cloud是比较适合的一个框架。为了解决目前的问题，重新设计了系统的架构。继续采用Maven做工程管理，docker部署。

## 总体架构



图二

## API网关

API网关是一个服务器，是系统的唯一入口。API网关封装了系统内部架构，为每个客户端提供一个定制的API。它可能还具有其它职责，如身份验证、监控、负载均衡、缓存。API网关负责服务请求路由、汇聚（组合微服务api接口）及协议转换。

1. 采用nginx访问API网关，API网关部署多个做集群，防止单点故障。
2. API网关在微服务架构中的位置：



图三

1. 调用soa开发的用户登录模块功能进行身份认证。
2. 采用Redis做缓存服务器。
3. 汇聚层作为API网关的一部分。
4. API网关根据不同的业务场景分别做定制。分如下几种：用户（APP）API网关、商家API网关、后台（eboss）API网关、联营API网关、众包API网关、用户（PC）网关、第三方接口API网关（用户（PC）网关、第三方接口API网关后期在实现）。
5. API网关实现数据格式转换和规整。

## 服务注册和发现

当前系统采用Zookeeper做服务发现，但是ZooKeeper是按照CP原则构建的，也就是说它能保证每个节点的数据保持一致，而一个Service发现服务应该从一开始就被设计成高可用的，所以它并不合适于做服务发现。Netflix Eureka是按照AP原则构建，符合架构的需要。

1. Eureka通过运行多个实例，并进行互相注册的方式可以比较方便的实现高可用的部署。如图：



图四

看到3个注册中心组成了集群，service服务通过Eureka1同步给了与之互相注册的Eureka2和Eureka3。

1. Eureka提供了server端和client。client是服务提供者，server端提供服务注册和发现，并提供了查看界面。
2. 后台有golang、Erlang等编写的服务，采用自定义的格式向zookeeper进行了注册，为了兼容这些服务，继续采用zookeeper做服务发现。

## 熔断器

Hystrix基于Spring Cloud和Netflix打造。引入Hystrix可以避免服务出现瘫痪的问题。

* Hystrix核心由RxJava驱动，是一个基于观察者模式的事件回调库；
* Hyxtrix的核心处理逻辑是将调用包装成Command，将对依赖的调用转换成Command API调用；
* Hystrix熔断器本质是一组状态机，是fast-fail设计思想的体现;
* 使用非常简单，在主类上加入@EnableCircuitBreaker即开启断路器功能；

## 日志收集

采用Spring Cloud Sleuth做日志是Spring Cloud的组成部分之一，为SpringCloud应用实现了一种分布式追踪解决方案，其兼容了Zipkin, HTrace和log-based追踪。Sleuth兼容log-based（例如：ELK）还可以继续使用ELK做日志分析。

## 缓存

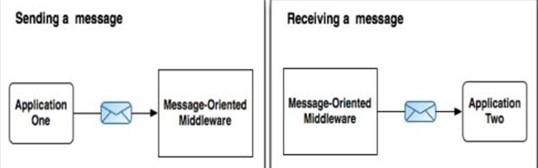
采用Redis作为缓存服务器，Redis有如下特性：

1. Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，zset，hash等数据结构的存储。
2. Redis支持数据的备份，即master-slave模式的数据备份。
3. Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。
4. 可采用Redis做分布式锁，解决部分分布式事物的问题。

由于采用微服务架构，服务和服务直接的调用可能出现比较复杂的情况，可以根据业务，把对应的数据存储到Redis里面，从而简化服务的调用关系，同时提高了运行效率。

## 消息中间件（MQ）

微服务之间采用REST API或者RPC调用会导致服务之间的耦合度相对变高，如果被调用者处理过程复杂，将会严重影响调用者的执行效率和资源使用率。此外，如果调用失败，整个架构即失败。采用MQ可以实现解耦，如下图：



图五

应用程序1向MQ发送一条消息，很可能一段时间之后，应用程序2调用MQ来收取消息。任何一个应用程序都不知道对方是否存在也不需要阻塞等待。这种通信方式大大缩减了维护开销，因为对于一个应用程序的修改，会对其他应用程序影响极小。采用MQ这样异步的通信和调用方式也有缺点，不是全部业务都适用。MQ相对效率低，异步模式的开发相对复杂。电商系统中，订单服务和支付服务比较适合采用MQ来做通信。

MQ框架非常之多，比较流行的有RabbitMQ、ActiveMQ、ZeroMQ、kafka，ZeroMQ使用比较复杂，kafka数据可能重复。考虑在RabbitMQ和ActiveMQ中选择一种。Spring Cloud Stream提供对了RabbitMQ和kafka使用的实现。

## 服务组件

微服务的实现，对外提供REST API实现服务调用接口。

1. Spring Could框架下微服务间的调用是REST API（通过一下设置，也可以改成RPC，目前还没有采用RPC调用必要）。
2. 服务拆分

* 将购物车从之前的用户服务中分离出来，将服务的主体由用户改成购物车（作为用户的一个属性）
* 将支付从订单服务中分离，支付作为一个独立的服务，支持多种支付手段。
* 把库存管理从商品服务中分离出来。
* 采用一个服务一个数据库的方式，对数据库进行拆分，有利于服务的部署和扩展。
* 服务内部采用本地事物保证一致性。服务间事物强一致性采用Redis做分布式锁，采用MQ或计划任务做最终一致性。
* 持久层由JdbcTemplate改成MyBatis。

开发约定选择XML文件，将接口和 Java 的 POJOs，映射成数据库中的记录，禁止使用拼接方式传参，避免sql注入。

## 存在的风险和问题

1. API网关成为服务的唯一入口，稳定性和性能要求非常高，开发难度大。
2. Hystrix虽然使用不难，但是熔断器的概念不容易理解，断路器何时打开和关闭断路,并在失败的情况下做什么，不好把握。
3. Sleuth日志跟踪后怎样有效的分析，还需要进一步研究。
4. MQ异步操作开发比较麻烦。
5. 微服务对业务拆分比较细，服务比较多，对运维的要求相对比较高。
6. 联营、众包和eboss的用户和soa的api网关分别管理。

以上风险虽然存在，但是如果对关键代码、关键环节做充分的设计和讨论，加强代码审核，加强单元测试，并建立对应机制，再加上与DBA和运维多沟通和讨论，就应该可以规避和减少以上风险。