系统架构说明书

目录

[系统架构说明书 1](#_Toc480980785)

[前言 4](#_Toc480980786)

[当前系统总体架构 4](#_Toc480980787)

[系统设计的不足 5](#_Toc480980788)

[系统运行时的一些问题 6](#_Toc480980789)

[重新设计 6](#_Toc480980790)

[API网关 7](#_Toc480980791)

[总体架构图 10](#_Toc480980792)

[资源下载 11](#_Toc480980793)

[城市渲染 11](#_Toc480980794)

[场景服务 12](#_Toc480980795)

[电商支付时序图 14](#_Toc480980796)

[服务注册和发现 14](#_Toc480980797)

[熔断器 18](#_Toc480980798)

[日志收集ELK 18](#_Toc480980799)

[缓存 21](#_Toc480980800)

[消息中间件（MQ） 22](#_Toc480980801)

[服务组件 22](#_Toc480980802)

[部署 24](#_Toc480980803)

[存在的风险和问题 26](#_Toc480980804)

#### 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 |
| 2017-04-20 | V0.4 |  | 1. 日志收集说明 2. 部署说明 | 陈志国 |
| 2017-04-26 | V0.5 |  | 添加了城市渲染、资源、场景服务等说明 | 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

采用微服务设计思想设计总体架构，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网关部分没有负载均衡。
* 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，可以主要用于电商平台。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。

## 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。它还具有其它职责，如身份验证、监控、缓存。API网关负责服务请求、汇聚（组合微服务api接口）及协议转换。

1. API网关部署在Load Balancer之后，LB做负载均衡。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：



图二

1. 汇聚层取消，作为API网关的一部分。根据图三，统一入口，减少层级。
2. 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、资源下载网关、城市渲染网关、IM网关、第三方接口API网关（第三方接口API网关后期在实现）。
3. API网关实现数据格式转换和规整。
4. API网关为日志跟踪提供traceId。
5. 资源文件采用青云对象存储进行保存和下载，需要加深与青云合作沟通，要求青云定制一些API，做到资源打包下载，减少握手和建链次数和时间，提高用户体验。
6. API网关的流程



图三

## 总体架构图



图四

## 资源下载

1. 流程



图五

1. 资源数据由城市编辑器上传到青云对象存储服务器

## 城市渲染

1. 基本流程



图六

1. mongodb存储json格式的位置信息

json格式数据传输量比较大，后期可考虑改用Protocol Buffer来传输

## 场景服务

1. 基本流程



图七

基于需要对用户推送消息和关联用户的原因，场景服务采用TCP做传输协议。目前没有场景服务的TCP网关，用户先从SOA网关获取场景服务地址再与场景服务直连。

1. 数据传输协议

|  |  |  |
| --- | --- | --- |
| LEN 4字节 | Protobuf | |
| CMD 4字节 | Protobuf |

## 电商支付时序图

以支付为例的一个时序图：



图九

库存需要根据需求分两种处理：

1. 强一致性，就采用全局锁或者乐观锁的方式实现。
2. 弱一致性，采用MQ，推送消息进行库存增减等，提高效率。

## 服务注册和发现

服务请求方需要知道服务实例的网络地址（IP 和端口），在微服务体系中，由于服务实例会被动态地分配网络地址，而且自动伸缩、故障和升级，服务实例会动态地改变，网络地址是相对动态的，需要引入服务发现机制解决这个问题。

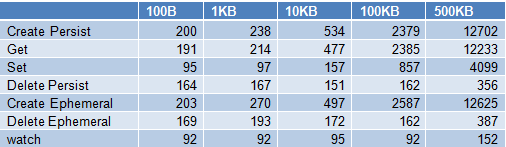
##### 当前系统采用Zookeeper做服务发现，ZooKeeper是按照CP原则构建的，它能保证每个节点的数据保持一致。

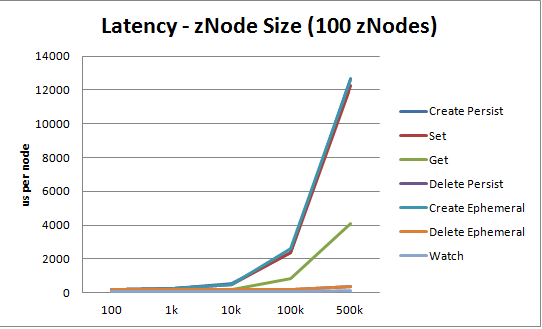
1. 后台有golang、Erlang等编写的服务，采用自定义的格式向zookeeper进行了注册，城市渲染服务采用zookeeper做服务发现。注册节点的格式如下：

serviceName：{url：“http://192.168.0.1:8080/ucenter”}

1. 青云Zookeeper性能：

异步并发操作100个 zNode 节点， 节点大小分别为100 B、 1 KB、 10 kB、 100 KB、 500 KB, 测试操作同测试一。测试结果如下，latency 单位为微秒。





青云ZooKeeper 在此次测试中，性能优秀，并发查询100个节点，10K数据不到0.5秒。

##### Netflix Eureka是按照AP原则构建，java电商部分采用Eureka做服务发现。

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：



图十

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。

* API网关发现和调用服务，原则上网关之间不调用。
* 微服务向Zookeeper注册并发现服务，原则上微服务之间不相互调用，由于业务原因需要这样做，需抽象出下一级服务来提供调用或者在微服务网关来聚集，同级不调用。如下图：



图十一

## 熔断器

复杂分布式架构通常都具有很多依赖。如果一个应用不能对来自依赖的故障进行隔离，那该应用本身就处在被拖垮的风险中。

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

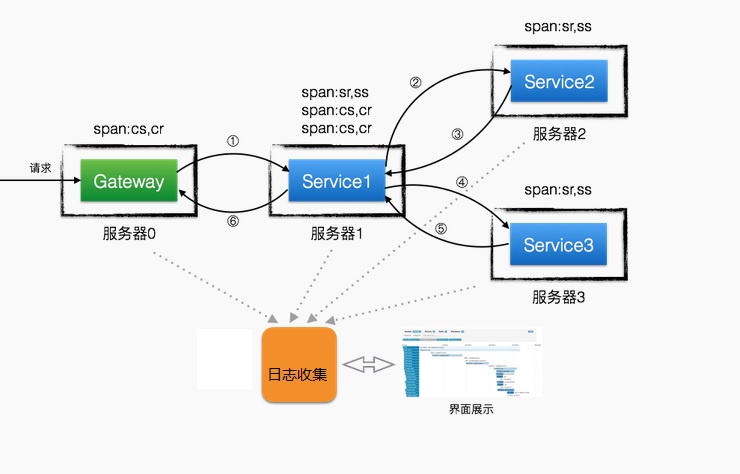
* Hystrix核心由RxJava驱动，是一个基于观察者模式的事件回调库；
* Hyxtrix的核心处理逻辑是将调用包装成Command，将对依赖的调用转换成Command API调用；
* Hystrix熔断器本质是一组状态机，是fast-fail设计思想的体现;

Hystrix基于Java语言开发，电商平台可以方便的引入。go语言的Hystrix在github上有个实现goHystrix，需要进一步研究。

## 日志收集ELK

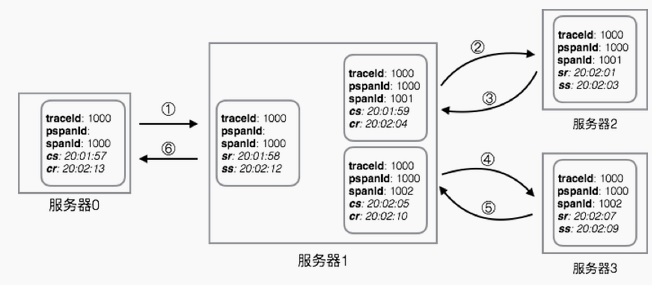
后台服务有php、java、erlang、golang等多种语言写的，服务是分布部署，互相有一些调用。假如有一个业务，需要把这些服务走完才完成，而这个业务出了问题，需要知道是哪个服务出了问题。现有后台服务的日志是分散的，运维挨个在服务器上找日志来排查问题，不方便、不及时，也不可靠。日志收集就是要把分散的日志收集在一起，对服务间的调用进行跟踪，方便排除问题以及发现性能瓶颈。

日志收集需要对日志进行跟踪，日志跟踪根据Zipkin的Span模型原理，API网关在用户发起请求时，生成traceId，spanId然后向下传递。



图十二

上图是服务间调用的示意图，gateway是第一级，service1是第二级，service2和service3是第三级。日志搜集会把有同一个traceId和 相关spanId的日志组装成最终完整的基本工作单元。图五是内部span的细节图：



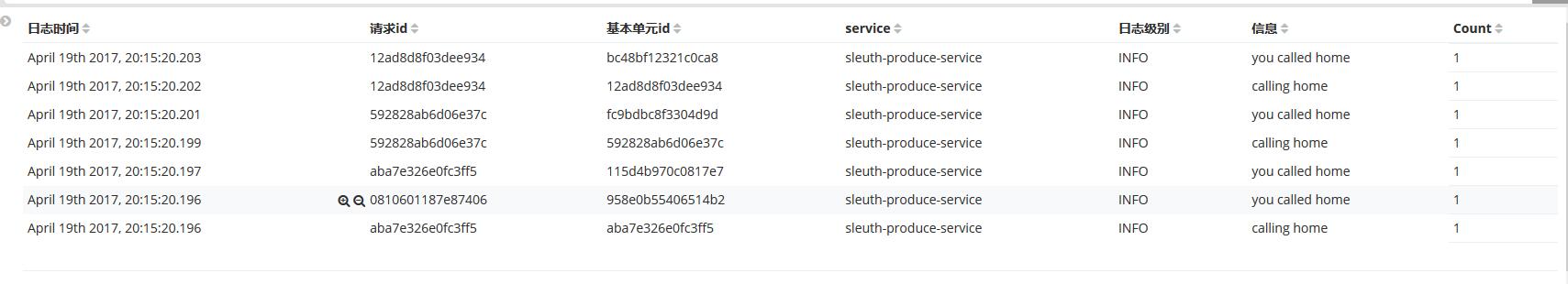
图十三

traceId要唯一标示每一次调用，它需要保证全局唯一。将int64的除第一位外的其他63位分成三段，前面41位为时间戳、后面10位为工作机器(进程)ID，也称为WorkerID ，最后12位为递增序列号。App网关在启动的时候，从Zookeeper服务器中查找一个可用的WorkerID， 如果找到一个可用的WorkerID，即创建一个临时子节点，当程序关闭时，该WorkerID就自动释放了，以达到了WorkerID的重用。在进程启动时，访问一次Redis来获取WorkeID，运行时不需要Redis持续提供服务，性能也不会有损失。

实现日志跟踪，要求程序在写日志的时候，把traceId和spanId写到日志里面。例如：2017-04-19 19:53:28.680  INFO [bootstrap,6e35af5cfbfbfef3,5fa483e79e066101,true] [sleuth-produce-service,6e35af5cfbfbfef3,5fa483e79e066101,true]  8932 --- [o-8001-exec-130] com.test.controller.TestController       : you called home

[sleuth-produce-service,6e35af5cfbfbfef3,5fa483e79e066101,true]中的6e35af5cfbfbfef3是traceId，5fa483e79e066101是spanId，其它字段是帮助查询的一些辅助字段。

日志收集采用开源组件ELK来实现， logstash做日志收集， elasticsearch对traceId和spanId做索引，kibana对traceId做分组，就把一个业务链全部串起了，同时还可利用日志中的时间戳可做性能分析。下图是一个日志收集的展现，已经可以把一次请求经过的全部服务串起来：



图十四

日志格式，各种不同语言的服务都按照统一格式，把日志发送到logstash的监听地点，实现日志收集。如上面的日志示例，日志中要包括时间、traceId，spanId，应用名，日志内容。Java服务可采用logback，通过配置帮助实现日志格式的定义和输出。golang用标准log库可实现，erlang有第三方Log4Erl，php可采用log4php来实现日志格式化。

由于数据中心的日志目前只是采集用户端的日志，只做了用户行为分析，做服务器日志分析还需要一段开发的时间。服务器日志跟踪可分2个阶段来实施：

第一阶段：采用ELK来做，可快速的实施和运行。

第二阶段：等数据中心实现日志分析后，只需简单的将日志输出到数据中心，数据中心就可以做到统一日志管理和分析。

## 缓存

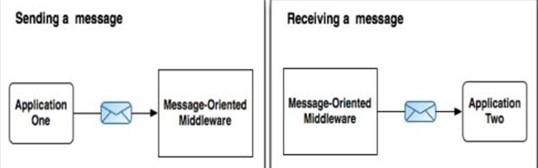
采用Redis作为缓存服务器，Redis有如下特性：

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。
3. 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：



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。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. 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）。有如下测试数据：

测试机器 cpu i5-4460 3.2GHz 内存8GB

feign rest ：

调用 1万次 有对象与json相互转换 在5100毫秒左右；无对象与json相互转换简单字符串在2400毫秒左右

调用10万次 有对象转换 54000毫秒左右，无对象转换简单字符串24500毫秒

dubbo rpc ：

调用 1万次 对象序列化反序列化 在2400毫秒 ；无对象序列化反序列化，简单字符串2000毫秒左右

调用 10万次 有序列化 23000毫秒；无序列化 简单字符串20000毫秒

纯本地调用平均每次调用不到1毫秒

加上数据库这样的io操作后 调用1万次 大约需要耗时10万毫秒，也就是平均一次调用需要10毫秒，已经是一个数量级的差距，所以通信协议上的差距不会造成太大影响。

1. 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，后期）
* 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(后期)。
* 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。

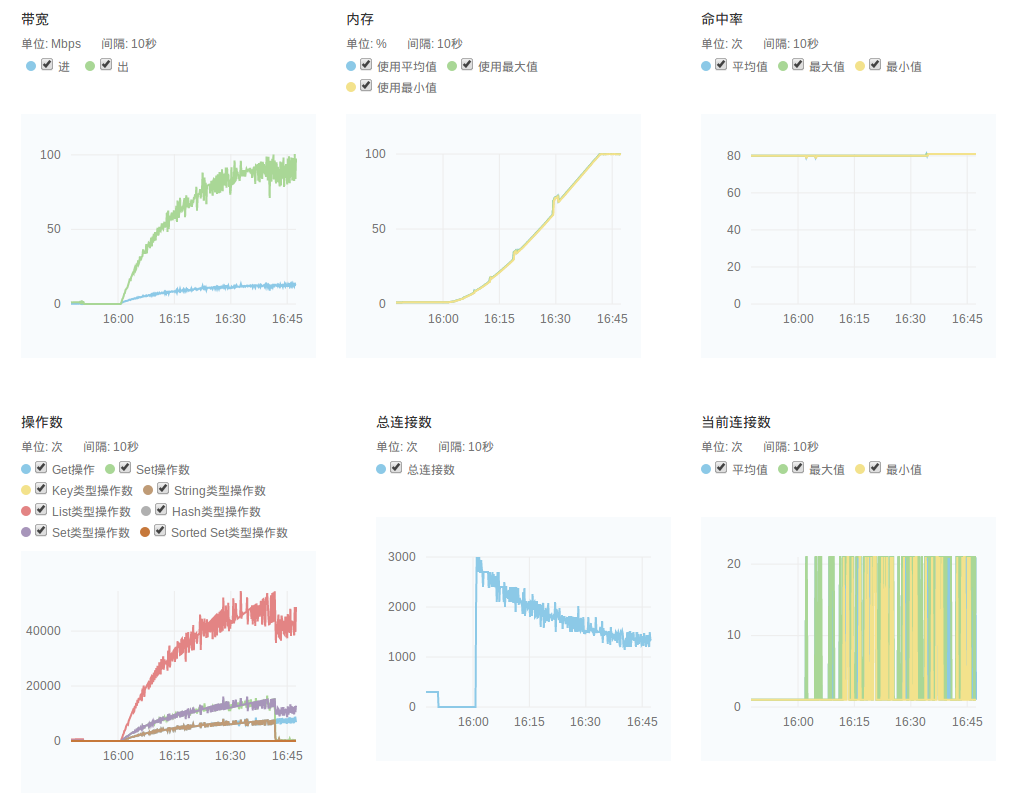
## 部署

在整个系统中，我们的服务除了soa\_ejabberd（聊天服务）、soa\_scene（场景服务）和pc\_vcity\_erlang（Vcity服务）之外，其余的服务全部采用docker进行部署，这些服务做成docker镜像，插入docker容器，可方便的扩容和降级。上面3个服务暂时由运维手动部署；相应的开发人员问题进行分析，最终达到采用docker部署。

架构体系中需要用到的负载均衡、Redis集群、消息队列集群Kafka、Zookeeper集群、docker容器、虚拟主机等，青云全都提供了相应的产品，而且都管理、监控、告警等功能，将平台部署在云之上，极大的降低了运维的部署、维护成本，同时也降低了开发的成本（例如：Zookeeper的集群不用自己配置）。研发和运维需要和青云密切合作，充分利用云资源，把我们的平台搭建好。

为了防止单点故障，至少在正式环境要把API网关和服务子系统部署2个。

青云服务器的监控图例：



告警设置：



配置后可送短信和邮件通知给用户。

## 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对于java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。还没有Hystrix对php、erlang、golang的实现。
3. MQ异步操作开发比较麻烦。
4. 事物一致性的问题虽然可以采用分布式锁和最终一致性来处理，但在细节上还需要根据具体的需求来仔细处理。
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。

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，就可以规避和减少以上风险。