微服务系统设计方案

# 微服务本质

微服务架构从本质上说其实就是分布式架构，与其说是一种新架构，不如说是一种**微服务架构风格。**

简单来说，微服务架构风格是要开发一种由**多个小服务组成**的应用。每个服务运**行于独立的进程**，并且采用**轻量级交互**。多数情况下是一个**HTTP的资源API**。这些服务**具备独立业务能力**并可以通过**自动化部署**方式**独立部署**。这种风格使**最小化集中管理**，从而可以使用多种**不同的编程语言和数据存储技术**。

对于微服务架构系统，由于其服务粒度小，模块化清晰，因此首先要做的是对系**统整体进行功能、服务规划**，优先考虑如何在交付过程中，从**工程实践出发，组织好代码结构、配置、测试、部署、运维、监控**的整个过程，从而有效体现微服务的独立性与可部署性。

本文将从微服务系统的设计阶段、开发阶段、测试阶段、部署阶段进行综合阐述。

**理解微服务架构和理念是核心。**

# 系统环境

|  |  |  |
| --- | --- | --- |
| 名称 | 版本 | 说明 |
| JDK | 1.8 |  |
| Spring Boot |  |  |
| Spring Framework |  |  |
| Ribbon |  |  |
| kafka |  |  |
| RabbitMQ |  |  |
|  |  |  |
|  |  |  |

# 微服务架构的挑战

* **可靠性：**

由于采用远程调用的方式，任何一个节点、网络出现问题，都将使得服务调用失败，随着微服务数量的增多，潜在故障点也将增多。

**也就是没有充分的保障机制，则单点故障会大量增加。**

* **运维要求高：**

系统监控、高可用性、自动化技术

* **分布式复杂性：**

网络延迟、系统容错、分布式事务

* **部署依赖性强：**

服务依赖、多版本问题

* **性能（服务间通讯成本高）：**

无状态性、进程间调用、跨网络调用

* **数据一致性：**

分布式事务管理需要跨越多个节点来保证数据的瞬时一致性，因此比起传统的单体架构的事务，成本要高得多。另外，在分布式系统中，通常会考虑通过数据的最终一致性来解决数据瞬时一致带来的系统不可用。

* **重复开发：**

微服务理念崇尚每个微服务作为一个产品看待，有自己的团队开发，甚至可以有自己完全不同的技术、框架，那么与其他微服务团队的技术共享就产生了矛盾，重复开发的工作即产生了。

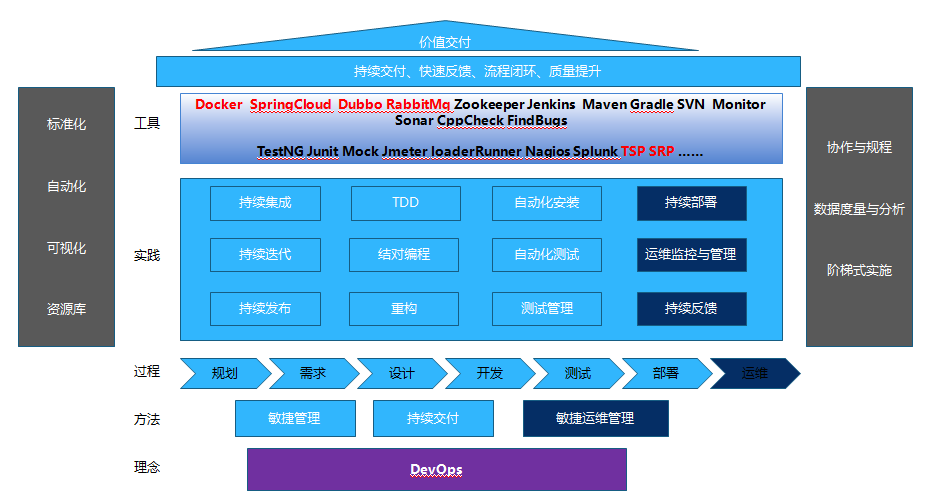


**没有最好的，只有最适合自己的。**

# 架构设计

## 思维设计

**微服务架构设计的根本目的是实现价值交付，微服务架构只有遵循DevOps理念方可进行的更顺畅，思维方式的转变是最重要的。**



实现微服务技术架构，现有产品需要进行技术上的改进以及相关配套服务的实现，采用**分阶段实施、以及试点产品优先实施的策略**，主要包括如下：

**一、技术上的改进：**

1、前后端分离，web前端通过Http/Https协议调用微服务的API网关，由API网关再经过路由服务调用相应的微服务

2、不同微服务之间通过REST方式互相调用

3、微服务之间通过消息中间件实现消息交互机制

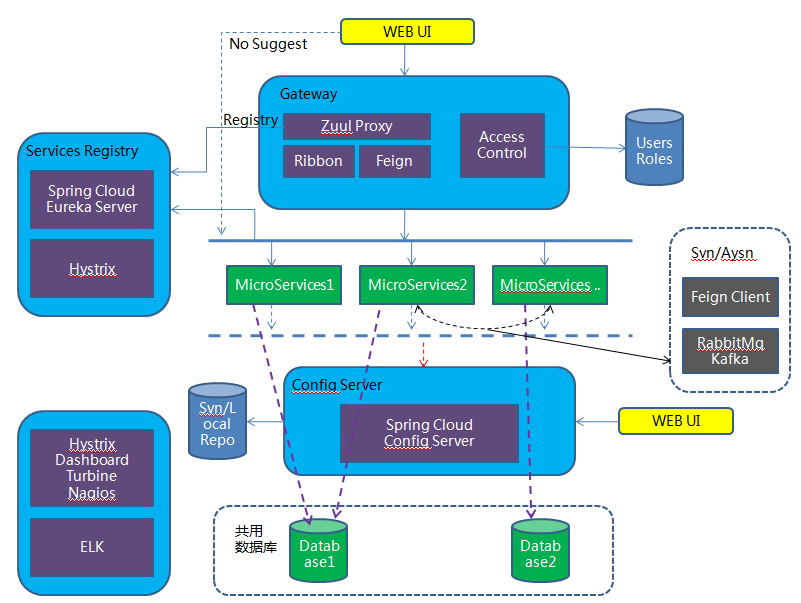
**二、配套服务与功能实现 ：**

1、需要进行相应的自动化服务实现，包括自动化构建、自动化安装部署、自动化测试、自动化平台发布（Docker实现）

2、管理服务，对于微服务架构，必须配套相应的监控与管理服务、日志管理服务等

3、协作服务，运用DevOps思想提升开发、测试、运维的高效沟通与协作，实现开发与运维的一体化

## 微服务架构设计



  1、我们把整个系统根据业务拆分成若干个子系统或微服务。

    2、每个子系统可以部署多个应用，多个应用之间使用负载均衡。

    3、需要一个服务注册中心Eureka，所有的服务都在注册中心注册，负载均衡也是通过在注册中心注册的服务来使用一定策略来实现。

**Eureka可部署多个，进行高可用保证。**

    4、所有的客户端都通过同一个网关地址访问后台的服务，通过路由配置ZUUL网关来判断一个URL请求由哪个服务处理。请求转发到服务上的时候使用负载均衡Ribbon。

    5、服务之间采用feign进行调用。

    6、使用断路器hystrix，及时处理服务调用时的超时和错误，防止由于其中一个服务的问题而导致整体系统的瘫痪。

    7、还需要一个监控功能，监控每个服务调用花费的时间等。

8、使用SpringCloud Config进行统一的配置管理，需要考虑与公司的配置管理平台如何配合使用。

9、Hystrix，监控和断路器。我们只需要在服务接口上添加Hystrix标签，就可以实现对这个接口的监控和断路器功能。

   10、Hystrix Dashboard，监控面板，他提供了一个界面，可以监控各个服务上的服务调用所消耗的时间等。

    11、Turbine，监控聚合，使用Hystrix监控，我们需要打开每一个服务实例的监控信息来查看。而Turbine可以帮助我们把所有的服务实例的监控信息聚合到一个地方统一查看。这样就不需要挨个打开一个个的页面一个个查看。

**架构的可靠性保证：**

**在关键节点做主备、集群部署，防止单点故障。**

**待后续确认问题：**

1、Access Control：Zuul网关提供了相关控制功能，与我司CAS如何结合使用

2、Config Server：Spring Cloud提供了远程配置中心，与我司的配置管理平台如何结合使用

# 设计阶段

## 总体设计

1、**功能规划**：对产品功能进行拆分，拆分为若干个微服务；一个功能可以创建多个微服务并部署在多个服务器节点上，以便进行负载均衡。

2、设计**原子服务层**，梳理和抽取核心应用、公共应用，作为独立的服务下沉到核心和公共能力层，逐渐形成稳定的服务中心，使应用能更快速的响应多变的客户需求。

3、为每个服务**设计API接口**（REST方式）

4、为不同的**服务进行分类**，不同类型的服务需要的资源不同，可以配置不同的资源，包括CPU、内存、存储等。

## 服务拆分原则

**1、粒度微小：**

根据业务功能划分服务粒度，总的原则是服务内部高内聚，服务之间低耦合。

**2、责任单一：**

每个服务只做一件事，即单一职责原则。

**3、隔离性原则：**

每个服务相互隔离，且不互相影响

**4、业务无关优先原则：**

基础服务，是一些基础组件，与具体的业务无关。比如：短信服务、邮件服务。这里的服务最容易划分出来做微服务，也是我们第一优先级分离出来的服务。

## 服务规划

为实现负载均衡，允许相同的服务在多个节点注册相同的服务名，不同的端口。如果没有前期的规划，不同的服务提供者可能会注册相同的服务名，导致消费者调用服务时产生调用混乱。

**因此，需进行服务名的统一规划：**

1、规划期统一制定每个服务提供者的服务名或者模块标示。

2、服务名的命名规则:ModuleName\_ServiceName，且**所有字符小写**，不同单词之间以**下划线分隔**。如用户管理模块提供了获取用户信息的服务，则命名为：user\_get\_info。

3、新增服务名时，需要提出申请，**审批通过后方可使用**，为减少审批复杂度，可只审批ModuleName，即在模块内部可以自由增加服务名，不需要进行审批。

## 开发策略

总体原则：不同的微服务需进行**物理隔离。**

1、SVN策略：SVN上创建**独立的分支**，不同微服务的代码提交不受相互影响；

**---由配置管理员统一控制。**

**问题：开发分支与集成分支，都将增加很多，维护工作量增加。**

2、编译策略：代码编译时，各个微服务独立编译、打包，**杜绝直接的依赖**；

3、工程构建：代码开发时，各微服务**创建独立的工程**，工程之间不能产生直接依赖

4、持续集成：每个微服务**独立执行持续集成**。

5、版本集成：由统一的集成工具，实现自动化的版本集成，将所有微服务集成到统一的版本发布包中。

## 版本策略

每个微服务可以独立制作版本，伴随着服务的增多，SVN分支增多，版本也将增多，版本管理的复杂度将成指数级增加。在服务之间依赖较多时，每个服务的升级或降级都将影响其他服务的正常运行。

因此需执行如下策略：

1、所有服务的版本制作交由专业的版本管理员执行。

2、采用自动化的版本制作策略，最大程度的减少人工操作。

3、每个服务的版本必须有详细的版本计划、版本说明，对于版本说明要制定模板，明确需要提交的内容、版本号、SVN标签等。

4、对项目经理的要求提升，需对整体的版本计划有严格的制定，尤其是版本之间的依赖关系要非常明确，版本升级、降级的**风险评估**需完全充分。

5、接口管理：严格执行接口管理制度，任何接口的变更必须进行审批、发公告等流程。

## 数据库挑战与策略

每个微服务都有自己独立的数据库，那么后台管理的联合查询怎么处理？这应该是大家会普遍遇到的一个问题，有三种处理方案。

1）严格按照微服务的划分来做，微服务相互独立，各微服务数据库也独立，后台需要展示数据时，调用各微服务的接口来获取对应的数据，再进行数据处理后展示出来，这是标准的用法，也是最麻烦的用法。

2) 将业务高度相关的表放到一个库中，将业务关系不是很紧密的表严格按照微服务模式来拆分，这样既可以使用微服务，也避免了数据库分散导致后台系统统计功能难以实现，是一个折中的方案。

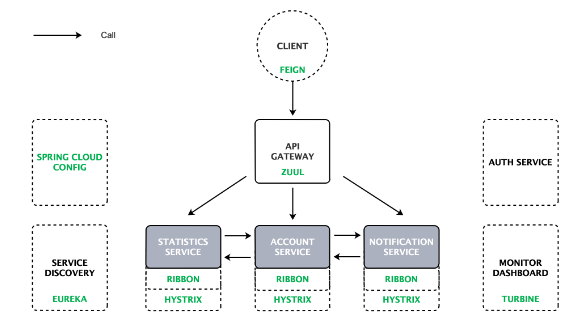
3）数据库严格按照微服务的要求来切分，以满足业务高并发，实时或者准实时将各微服务数据库数据同步到NoSQL数据库中，在同步的过程中进行数据清洗，用来满足后台业务系统的使用，推荐使用MongoDB、HBase等。

第一种方案适合业务较为简单的小公司；第二种方案，适合在原有系统之上，慢慢演化为微服务架构的公司；第三种适合大型高并发的互联网公司。

**建议，我们当前采用第二种方案。**

## 负载均衡

不再采用一般的增加负载均衡服务器的方式进行负载均衡，如F5、Nginx、LVS等，而是把负载均衡的功能**以库的方式集成到服务消费方的进程内**，这种方案称为**软负载均衡**（Soft Load Balancing）或者客户端负载均衡。在Spring Cloud中配合Eureka的服务注册功能，Ribbon子项目则为REST客户端实现了负载均衡。



使用Ribbon进行负载均衡，其工作原理可以概括为下面四个步骤：

1. Ribbon首先根据其所在Zone优先选择一个负载较少的Eureka Server;
2. 定期从Eureka Server更新并过滤服务实例列表;
3. 根据指定的负载均衡策略，从可用的服务器列表中选择一个服务实例的地址;
4. 然后通过RestClient进行服务调用。

**Ribbon本身提供了下面几种负载均衡策略：**

* **RoundRobinRule:** 轮询策略，Ribbon以轮询的方式选择服务器，这个是默认值。所以示例中所启动的两个服务会被循环访问;
* **RandomRule:** 随机选择，也就是说Ribbon会随机从服务器列表中选择一个进行访问;
* **BestAvailableRule:** 最大可用策略，即先过滤出故障服务器后，选择一个当前并发请求数最小的;
* **WeightedResponseTimeRule:** 带有加权的轮询策略，对各个服务器响应时间进行加权处理，然后在采用轮询的方式来获取相应的服务器;
* **AvailabilityFilteringRule:** 可用过滤策略，先过滤出故障的或并发请求大于阈值一部分服务实例，然后再以线性轮询的方式从过滤后的实例清单中选出一个;
* **ZoneAvoidanceRule:** 区域感知策略，先使用主过滤条件（区域负载器，选择最优区域）对所有实例过滤并返回过滤后的实例清单，依次使用次过滤条件列表中的过滤条件对主过滤条件的结果进行过滤，判断最小过滤数（默认1）和最小过滤百分比（默认0），最后对满足条件的服务器则使用RoundRobinRule(轮询方式)选择一个服务器实例。

## 性能策略

1、网络优化：优化组网结构，提升网络间通讯性能；

2、配置优化：优化Spring Cloud组件集以及其他组件的配置信息，使得性能最大化。

## 技术管理策略

微服务的架构理念中指出各微服务可以**独立建设，可以使用不同的技术、语言、框架等**，以便能更快速的使用新技术、新框架等响应特定客户需求，解决单体应用架构更新技术、更新框架时面临的困难或阻碍。

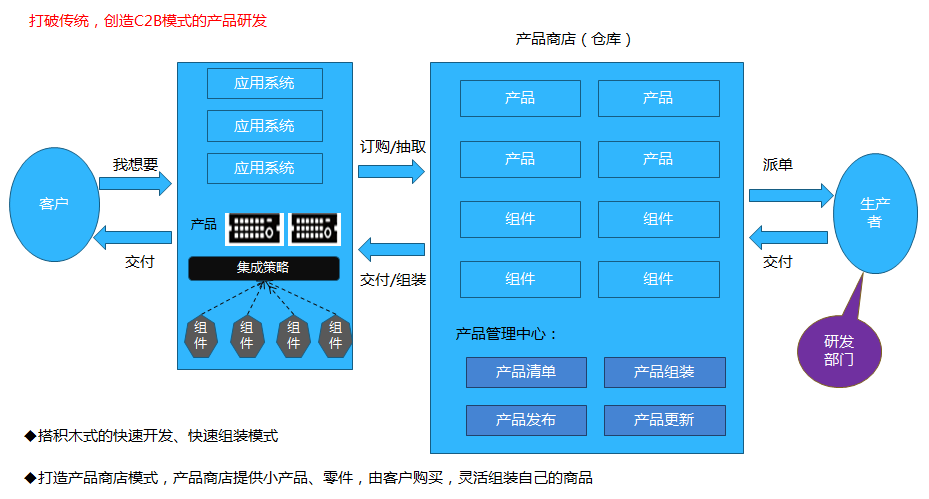
但这也同时带来了诸多问题，如下：

1、各服务是否可以任意使用自己的技术、自己的组件、框架呢？如果这样，势必带来更大的管理困难、维护困难、技术共享困难。

2、公共的方法如何实现共享？如格式化时间的一个简单方法需要共享，也需要封装为一个服务接口吗？

**管理策略：**

1、总体原则：仍然需要进行统筹考虑，所有组件统一管理，组件放置在产品仓库中，每个产品或服务需要共享组件时，从产品仓库获取。



2、特殊情况：特殊服务需要使用特殊的组件、框架，需提出申请，统筹规划后进行决策。

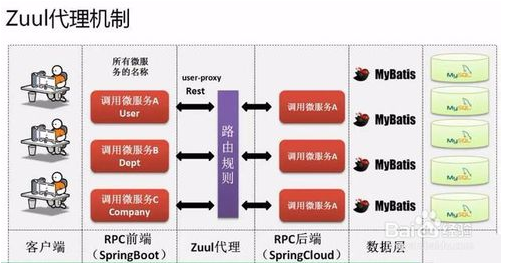
# 开发阶段

## 服务的调用

### AIP网关调用

所有服务通过Zuul网关进行调用，不允许直接调用微服务提供者。

Zuul可能会成为系统瓶颈，在项目复杂时可考虑为Zuul进行主备或负载均衡处理。



### 同步调用

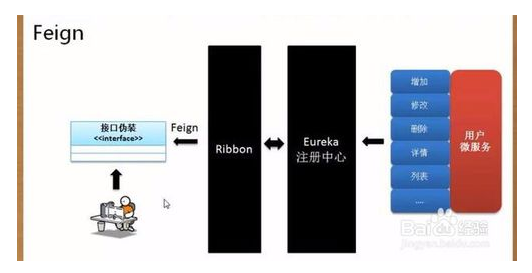
采用HTTP REST方式进行调用，针对业务需求可以进行负载均衡，负载均衡的调用方式有两种：

1、FeignClient

2、RestTemplate

**建议使用FeignClient方式进行服务调用。**

不管是什么方式，他都是通过REST接口调用服务的http接口，参数和结果默认都是通过Jackson序列化和反序列化。因为Spring MVC的RestController定义的接口，返回的数据都是通过Jackson序列化成JSON数据。



### 异步调用

rabbitMq、kafka、Spring Cloud Stream均是可以选择的方案。

* Spring Cloud Stream，基于 Redis、Rabbit、Kafka 实现的消息微服务，简单声明模型用以在 Spring Cloud 应用中收发消息。

### 服务间调用的权限验证

一般我们的API接口都需要某种授权才能访问，登陆成功以后，然后通过token或者cookie等方式才能调用接口。

使用Spring Cloud Netfix框架的话，登录的时候，把登录请求转发到相应的用户服务上，登陆成功后，会设置cookie或header token等。然后客户端接下来的请求就会带着这些验证信息，从Zuul网关传到相应的服务上进行验证。

Zuul网关在把请求转发到后台的服务的时候，会默认把一些header传到服务端，如：Cookie、Set-Cookie、Authorization。这样，客户端请求的相关headers就可以传递到服务端，服务端设置的cookie也可以传到客户端。

但是，如果你想禁止某些header透传到服务端，可以在Zuul网关的application.yml配置里通过下面的方式禁用：

|  |
| --- |
| zuul: routes: users:  path: /users/\*\*  sensitiveHeaders: Cookie,Set-Cookie,Authorization  serviceId: user |

刚才说了我们的某个服务有时候需要调用另一个服务，这时候，这个请求不是客户端发起，他的请求的header里面也不会有任何验证信息。这时候，要么，通过防火墙等设置，保证服务间调用的接口，只能某几个地址访问；要么，就通过某种方式设置header。

同时，如果你想在某个服务里面获得这个请求的真是IP，（因为请求的通过网关转发而来，你直接通过request获得ip得到的是网关的IP），就可以从headerX-Forwarded-Host获得。如果想禁用这个header，也可以：

|  |
| --- |
| zuul.addProxyHeaders = false |

如果你使用RestTemplate的方式调用，可以在请求里面添加一个有header的Options。

也可以通过如下的拦截器的方式设置，它对RestTemplate方式和FeignClient的方式都可以起作用：

|  |
| --- |
| @Bean public RequestInterceptor requestInterceptor() { return new RequestInterceptor() {  @Override  public void apply(RequestTemplate template) {  String authToken = getToken();  template.header(AUTH\_TOKEN\_HEADER, authToken);  } }; } |

### 服务编排

主要的作用是减少项目中的相互依赖。比如现在有项目a调用项目b，项目b调用项目c...一直到h，是一个调用链，那么项目上线的时候需要先更新最底层的h再更新g...更新c更新b最后是更新项目a。这只是这一个调用链，在复杂的业务中有非常多的调用，如果要记住每一个调用链对开发运维人员来说就是灾难。

有这样一个好办法可以尽量的减少项目的相互依赖，就是服务编排，一个核心的业务处理项目，负责和各个微服务打交道。比如之前是a调用b，b掉用c，c调用d，现在统一在一个核心项目W中来处理，W服务使用a的时候去调用b，使用b的时候W去调用c。

其实可以理解为面向对象的设计，减少方法之间的一层层嵌套调用，而采取一个方法进行业务流程的串联，如方法W实现一个完整的业务处理，则采取下面方式：

function w（）

{

1、调用方法a；

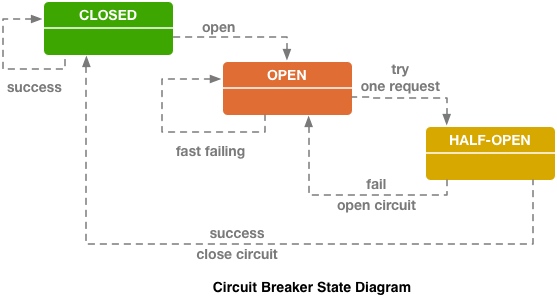
2、调用方法b；

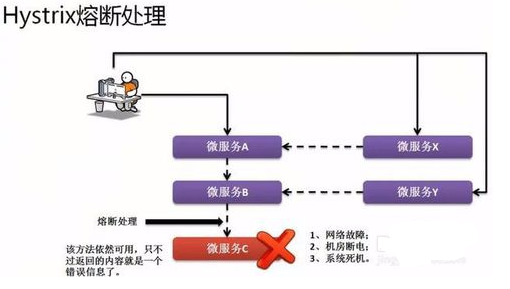
3、调用方法c；

}

## 服务的熔断处理

在服务之间进行调用时，由于各种原因会导致远程**服务不可用**或压力过载等异常导致的**故障蔓延**，此时需要有一种机制进行保护处理。Spring Cloud通过Netflix的Hystrix组件实现**熔断和降级**处理解决此问题。断路器(Cricuit Breaker)是一种能够在远程服务不可用时自动熔断(打开开关)，并在远程服务恢复时自动恢复(闭合开关)的设施，Spring Cloud通过Netflix的Hystrix组件提供断路器、资源隔离与自我修复功能。

**Spring cloud Hystrix 熔断器  
  
[](http://photo.blog.sina.com.cn/showpic.html#blogid=493a84550102wkp2&url=http://album.sina.com.cn/pic/001l8XD7zy76r07juaXb7)**

****

## 统一日志管理

不同微服务部署在不同节点上，登录每个节点查看日志是比较麻烦的，同时对于需要关联多个微服务日志联合查看分析的情况将更加麻烦。伴随节点数量的增加，如果没有合适的管理机制与工具，定位问题、发现问题的复杂性将越来越大，将成指数级增长，因此需要进行统一日志管理。

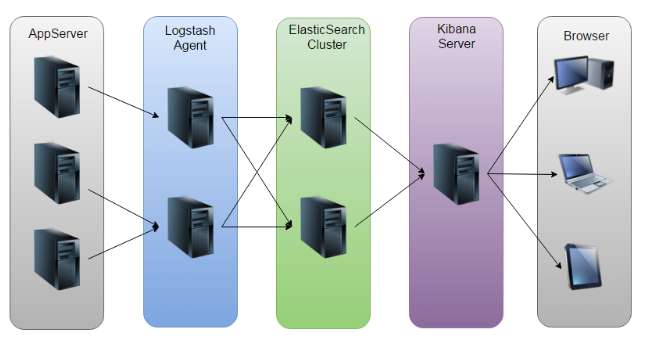
1、建立统一的日志管理规范；

2、开发并使用统一的日志组件，为所有微服务提供统一的日志服务，由log4j或Blitz4j封装；

3、在每个服务节点上部署日志采集Agent组件，由此Agent进行日志的采集与转发；

4、建立统一的日志中心，所有日志写入日志中心。

说明：上述日志的实现由公司的“日志管理平台”进行实现，采用的是ELK集合框架。



## 统一监控管理

使用Hystrix组件进行服务的监控，使用Nagios进行服务器等资源的监控。

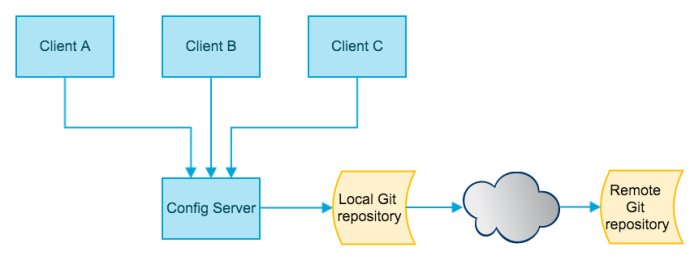
1、Hystrix，监控和断路器。我们只需要在服务接口上添加Hystrix标签，就可以实现对这个接口的监控和断路器功能。

  2、Hystrix Dashboard，监控面板，他提供了一个界面，可以监控各个服务上的服务调用所消耗的时间等。

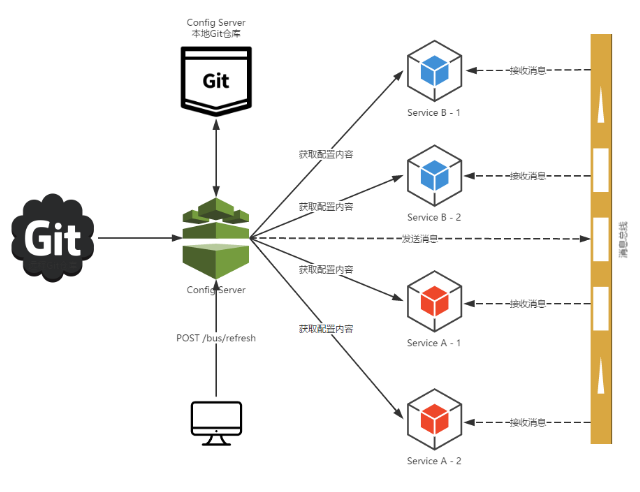
 3、Turbine，监控聚合，使用Hystrix监控，我们需要打开每一个服务实例的监控信息来查看。而Turbine可以帮助我们把所有的服务实例的监控信息聚合到一个地方统一查看。这样就不需要挨个打开一个个的页面一个个查看。

## 统一配置管理

实现各微服务的**统一参数配置以及版本管理**，可采用公司的配置管理平台或者Spring Cloud Config配置中心。

**Spring Cloud Config配置中心**  
  
[](http://photo.blog.sina.com.cn/showpic.html#blogid=493a84550102wkp2&url=http://album.sina.com.cn/pic/001l8XD7zy76qZZNxXA1a)  
  
 Spring Cloud Config就是我们通常意义上的配置中心。Spring Cloud Config-把应用原本放在本地文件的配置抽取出来放在中心服务器，**本质是配置信息从本地迁移到云端**。从而能够提供更好的管理、发布能力。  
 Spring Cloud Config分服务端和客户端，**服务端负责将git（svn）中存储的配置文件发布成REST接口**，客户端可以从服务端REST接口获取配置。但**客户端并不能主动感知到配置的变化**，从而主动去获取新的配置，这需要**每个客户端通过POST方法触发各自的/refresh。**

为解决配置信息能及时通知到各服务，同时减少每个微服务处理配置信息更新的复杂度，为此我们通过消息总线来解决此问题，方案如下：



1. Git仓库、Config Server、以及微服务“Service A”、 “Service B”的实例中都引入了Spring Cloud Bus，所以他们都连接到了RabbitMQ的消息总线上。
2. 从Git仓库中配置的修改到发起/bus/refresh的POST请求这一步可以通过Git仓库的Web Hook来自动触发。
3. /bus/refresh请求不再发送到具体服务实例上，而是发送给Config Server，并通过destination参数来指定需要更新配置的服务或实例。
4. 由于所有连接到消息总线上的应用都会接受到更新请求，所以在Web Hook中就不需要维护所有节点内容来进行更新，从而解决了通过Web Hook来逐个进行刷新的问题。

## 分布式session

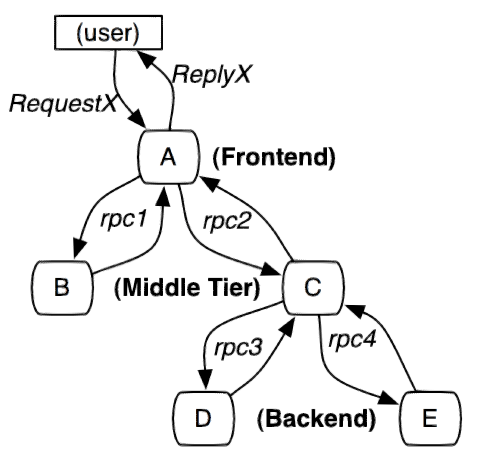
采用Redis作为缓存组件以及session的共享组件。

## REST资源响应结构

制定规范和解析方法。

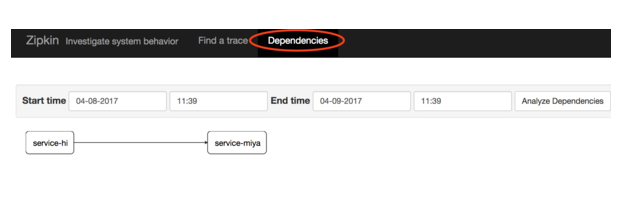
## API调用链追踪

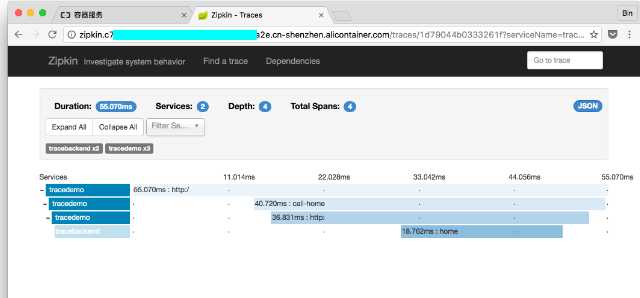
微服务架构上通过业务来划分服务的，通过REST调用，对外暴露的一个接口，可能需要很多个服务协同才能完成这个接口功能，如果链路上任何一个服务出现问题或者网络超时，都会形成导致接口调用失败。随着业务的不断扩张，服务之间互相调用会越来越复杂。



Spring Cloud Sleuth 主要功能就是在分布式系统中提供追踪解决方案，并且兼容支持了 zipkin，你只需要在pom文件中引入相应的依赖即可。







## 单元测试

做微服务架构，进行系统测试的复杂度较大，为保证产品质量与开发、测试效率，单元测试是必不可少的。

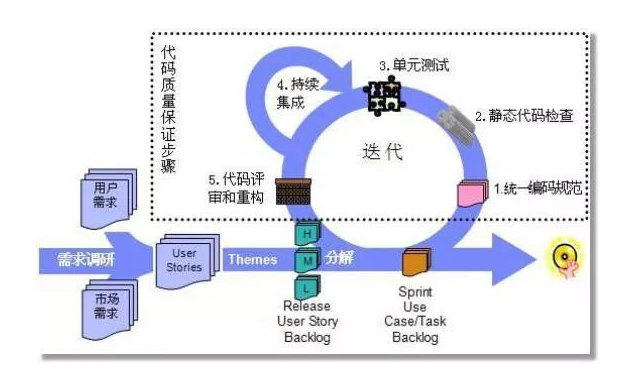
可采用Mock方式进行测试模拟，由持续集成进行自动化单元测试的执行以及结果输出。

## 代码调试

对于单体架构系统，可直接本地化调试，但对于微服务架构，接口间的调用需采用远程通讯的方式，也就是说被调用的服务必须启动后方可被调用，因此当微服务增多时，你可能需要启动大量的微服务或者web服务器，这给本地化调用与调试带来了困难。

**解决方案待研究。**

# 测试



## 自动化测试

* **单元测试：**

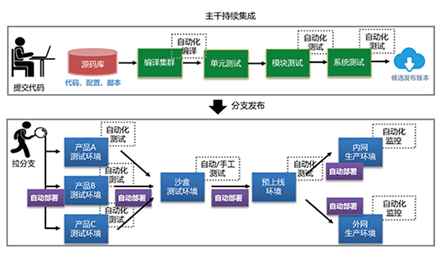
由开发人员实现。

采用Mock方式进行测试模拟，由持续集成进行自动化单元测试的执行以及结果输出。

* **业务测试：**

开发进行实现，测试也需考虑如何实现。

将多个服务或业务单元进行串联，测试一个完整的业务，甚至是不同业务之间组成的系统测试，需要采用相关的自动化测试框架执行，如RobotFramework自动化测试框架。



## 依赖测试

也可以称为接口测试或者契约测试，在微服务逐渐增多的情况下，如何有效保证服务之间能够按照接口的约定正常工作，即符合契约，成为微服务实施过程中，测试面临的主要挑战。

一、**开发自动化的接口测试工具，**

1、检测接口是否满足约定

2、检测接口是否发生变化

3、检测接口是否可以正常被调用。

**二、测试方法：**

**采取基于消费者驱动的契约测试，测试架构如下：**

**其优势如下：**

* **从价值实现的角度定义契约**

从消费者使用契约的角度出发，首先保证消费者基于此契约是可以实现价值的，有了这个前提，再使用契约来验证提供者，如果提供者提供的契约同定义的契约一致，则证明提供者提供的契约是能够实现服务消费者的。通过这种方式，使得更聚焦于如何从价值实现出发。

* **隔离消费者和提供者的测试**

对于契约的消费者和提供者可以分开独立测试，有效解决传统集成测试服务架构的弊端，将微服务的接口测试成本降到最低。

**三、测试工具：**

**Pact、Janus、Pacto等。**

## 系统测试

## 熔断测试

1、通过停止微服务的方式测试服务路由的正确性

2、通过压力测试，将某个微服务产生过载等异常，测试服务熔断或降级

3、通过压力测试，测试负载均衡策略的正确性

## 性能测试

原有本地化的api调用将会变成REST的远程调用，调用速度势必受到影响，因此需要对系统性能进行考虑以及性能测试，主要影响因素如下：

1、网络：远程调用时受到网络通讯速度的影响，这涉及到网络速度、网络部署以及系统架构，有相互依赖的服务应采取**就近部署原则**。

2、服务器：受到远程服务所在服务器性能的影响。

3、数据量：数据量这里指的是数据大小以及数据传输的次数以及频率，此时REST调用方式会产生瓶颈，当然，**最好的方式是避免此种情况发生**，此种场景采取消息中间件的方式异步通讯。

# 持续集成

1、持续集成：每个微服务独立执行持续集成。

2、版本集成：由统一的集成工具，实现自动化的版本集成，将所有微服务集成到统一的版本发布包中。

3、持续集成可制作多种场景的版本，包括测试环境、开发环境、生产环境。

4、统计测试覆盖率等指标数据。

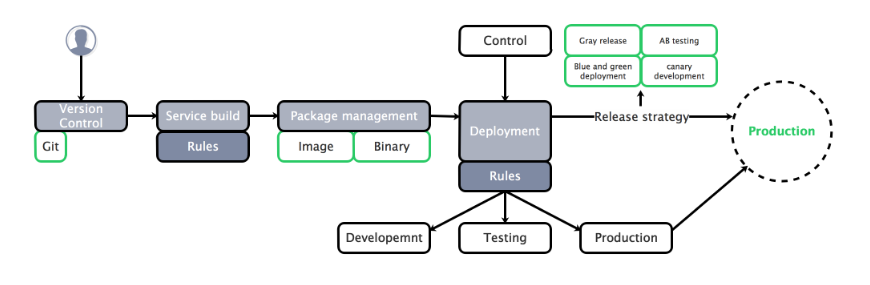
5、工具：Jenkins、Sonar等。



# 持续部署

1、通过持续集成自动制作发布版本的Docker镜像；

2、将docker镜像自动上传到docker容器中。



# 运维阶段

## 远程升级

微服务不断增加后，意味着部署容器也在同步增加，对于后续升级维护的工作量将会逐渐增加，开发统一管理中心，支持远程维护与升级将可减少运维的复杂度。

## 统一配置中心

使用Spring Cloud Config或者配置管理平台进行统一的配置管理。

## 统一日志中心

使用日志管理平台进行统一的日志采集、日志分析。