SpringCloudAlibaba 课程说明

SpringCloudAlibaba 中文社区地址:
https://github.com/alibaba/spring-cloud-alibaba/blob/master/README-zh.md

微服务架构演变过程

传统架构

传统的架构，也就是为单点应用，也就是大家在早期所学习的 JavaEE 知识 SSH 或者 SSM 架构模式，会采用分层架构模式：数据库访问层、业务逻辑层、控制层，从前端到后台所有的代码都是一个开发者去完成。
该架构模式没有对我们业务逻辑代码实现拆分，所有的代码都写入到同一个工程中里面，适合于小公司开发团队或者个人开发。
这种架构模式最大的缺点，如果该系统一个模块出现不可用、会导致整个系统无法使用。

分布式架构

分布式架构模式是基于传统的架构模式演变过来，将传统的单点项目根据业务模块实现拆分、会拆分为会员系统、订单系统、支付系统、秒杀系统等。 从而降低我们项目的耦合度，这种架构模式开始慢慢的适合于互联网公司开发团队。

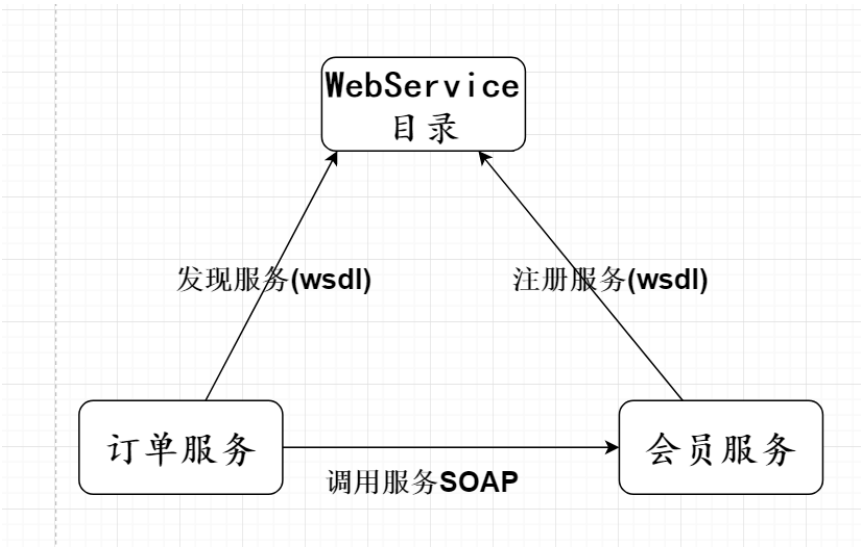
如果命名为系统的话：包含了视图层

SOA 面向服务架构

SOA 架构模式也称作为：面向服务架构模式、俗称面向与接口开发，将共同存在的业务逻辑抽取成一个共同的服务，提供给其他的服务接口实现调用、服务与服务之间通讯采用 rpc 远程调用技术。

SOA 架构模式特点：

1. SOA 架构通讯中，采用 XML 方式实现通讯、在高并发下通讯过程中协议存在非常大冗余性，所以在最后微服务架构模式中使用 JSON 格式替代了 XML。
2. SOA 架构模式实现方案为 WebService 或者是 ESB 企业服务总线 底层通讯协议 SOAP 协议（Http+XML）实现传输。



ESB 企业服务总线

解决多系统之间跨语言通讯，数据协议的转换，提供可靠消息传输。

基于 IDEA 快速构建 WebService

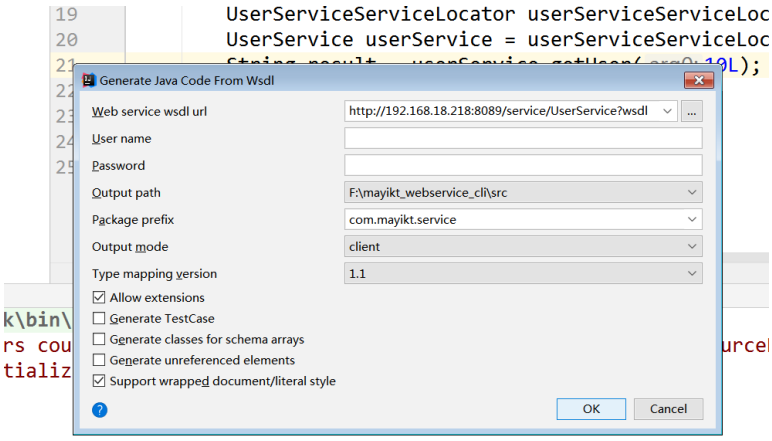
WebService 服务器端

```
@WebService
public class UserService {


    @WebMethod
    public String getUser(Long id) {
        return "mayikt 用户:" + id
                ;
    }


    public static void main(String[] args) {
        Endpoint.publish("http://192.168.18.218:8089/service/UserService", new UserService());
        System.out.println("服务发布成功");


    }
}
```

http://192.168.18.218:8089/service/UserService?wsdl　获取 wsdl

wsdl 文件描述接口的调用地址 服务的接口 方法 参数等。

WebService 客户端




```
public class WebServiceClient {

    public static void main(String[] args) throws ServiceException, RemoteException {

        UserServiceServiceLocator userServiceServiceLocator = new UserServiceServiceLocator();

        UserService userService = userServiceServiceLocator.getUserServicePort();

        String result = userService.getUser(10L);

        System.out.println("result:" + result);

    }
}
```

微服务架构

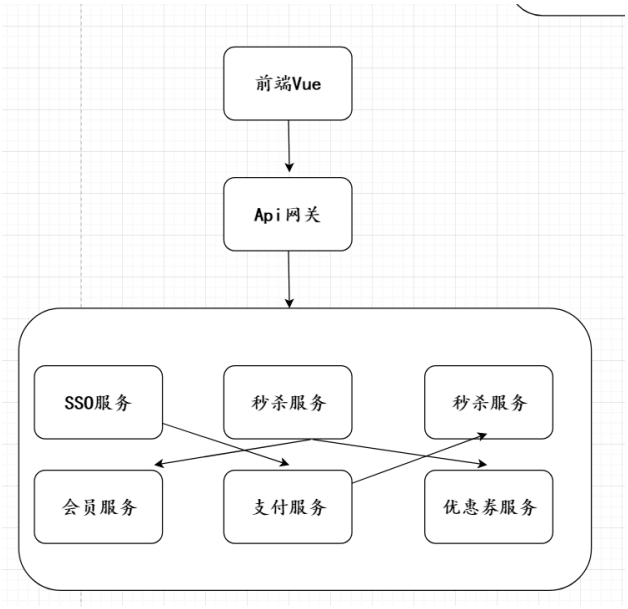
微服务架构产生的原因

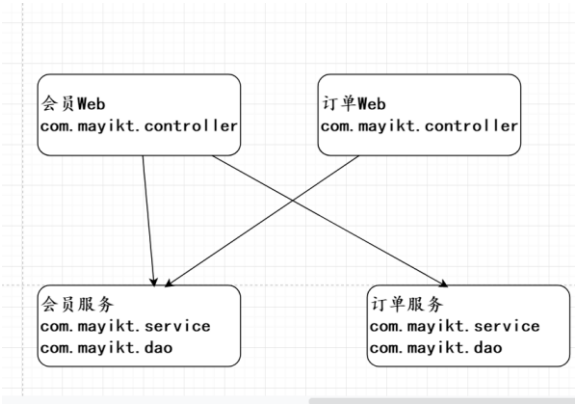
微服务架构基于 SOA 架构演变过来的
在传统的 WebService 架构中有如下问题:
1. 依赖中心化服务发现机制
2. 使用 Soap 通讯协议，通常使用 XML 格式来序列化通讯数据，xml 格式非常喜欢重，比较占宽带传输。
3. 服务化管理和治理设施不完善

微服务架构基本概念

微服务架构模式是从 SOA 架构模式演变过来， 比 SOA 架构模式粒度更加精细，让专业的人去做专业的事情（专注），目的是提高效率，每个服务与服务之间互不影响，微服务架构中每个服务必须独立部署、互不影响，微服务架构模式体现轻巧、轻量级、适合于互联网公司开发模式。

微服务架构倡导应用程序设计程多个独立、可配置、可运行和可微服务的子服务。
服务与服务通讯协议采用 Http 协议，使用 restful 风格 API 形式来进行通讯，数据交换格式轻量级 json 格式通讯，整个传输过程中，采用二进制，所以 http 协议可以跨语言平台，并且可以和其他不同的语言进行相互的通讯，所以很多开放平台都采用 http 协议接口。





微服务架构与 SOA 架构的不同

1.微服务架构基于 SOA 架构 演变过来，继承 SOA 架构的优点，在微服务架构中去除 SOA

架构中的 ESB 企业服务总线，采用 http+json（restful）进行传输。
2.微服务架构比 SOA 架构粒度会更加精细，让专业的人去做专业的事情（专注），目的提高效率，每个服务于服务之间互不影响，微服务架构中，每个服务必须独立部署，微服务架构更加轻巧，轻量级。
3.SOA 架构中可能数据库存储会发生共享，微服务强调独每个服务都是单独数据库，保证每个服务于服务之间互不影响。
4.项目体现特征微服务架构比 SOA 架构更加适合与互联网公司敏捷开发、快速迭代版本，因为粒度非常精细。

微服务架构会产生那些问题

分布式事务解决方案(rabbitmq/rocketmq/lcn(已经淘汰)/Seata)
分布式任务调度平台(XXL-Job、阿里 Scheduler)
分布式日志采集系统 ELJ+Kafka
分布式服务注册中心 eureka、Zookeeper、consule、nacos 等。
分布式服务追踪与调用链 Zipkin 等。

为什么我们要使用 SpringCloud

SpringCloud 并不是 rpc 远程调用框架，而是一套全家桶的微服务解决框架，理念就是解决我们在微服务架构中遇到的任何问题。

SpringCloud 第一代与第二代的区别

名称	SpringCloud第一代	SpringCloud第二代
网关	Spring Cloud Zuul	Spring Cloud Gateway
注册中心	Eureka(不再更新)，Consul,ZK	阿里Nacos，拍拍贷radar等可选
配置中心	SpringCloud Config	阿里Nacos，携程Apollo，随行付Config
客户端软负载均衡	Ribbon	Spring-cloud-loadbalancer
熔断器	Hystrix	spring-cloud-r4j(Resilience4J)，阿里Sentinel

SpringCloud 第一代：
SpringCloud Config 分布式配置中心
SpringCloud Netflix 核心组件
    Eureka:服务治理
    Hystrix:服务保护框架
    Ribbon:客户端负载均衡器

Feign：基于ribbon和hystrix的声明式服务调用组件
Zuul：网关组件,提供智能路由、访问过滤等功能。

SpringCloud第二代（自己研发）和优秀的组件组合：

Spring Cloud Gateway 网关
Spring Cloud Loadbalancer 客户端负载均衡器
Spring Cloud r4j(Resilience4J) 服务保护

Spring Cloud Alibaba Nacos 服务注册
Spring Cloud Alibaba Nacos 分布式配置中心
Spring Cloud Alibaba Sentinel 服务保护
SpringCloud Alibaba Seata 分布式事务解决框架
Alibaba Cloud OSS 阿里云存储
Alibaba Cloud SchedulerX 分布式任务调度平台
Alibaba Cloud SMS 分布式短信系统

为什么 Alibaba 要推出 SpringCloud 组件

目的就是为了对阿里云的产品实现扩展。

服务注册与发现 nacos

服务治理基本的概念

服务治理概念：
在RPC远程调用过程中，服务与服务之间依赖关系非常大，服务Url地址管理非常复杂，所以这时候需要对我们服务的url实现治理，通过服务治理可以实现服务注册与发现、负载均衡、容错等。

服务注册中心的概念

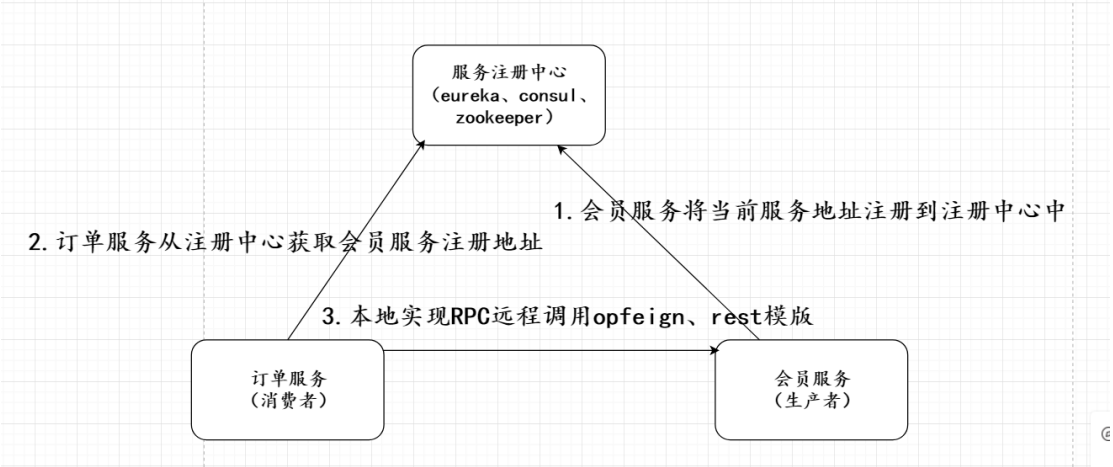
每次调用该服务如果地址直接写死的话，一旦接口发生变化的情况下，这时候需要重新发布版本才可以该接口调用地址，所以需要一个注册中心统一管理我们的服务注册与发现。

注册中心：我们的服务注册到我们注册中心，key为服务名称、value为该服务调用地址，该类型为集合类型。Eureka、consul、zookeeper、nacos等。
服务注册：我们生产者项目启动的时候，会将当前服务自己的信息地址注册到注册中心。

服务发现：消费者从我们的注册中心上获取生产者调用的地址（集合），在使用负载均衡的策略获取集群中某个地址实现本地 rpc 远程调用。

微服务调用接口常用名词

生产者：提供接口被其他服务调用
消费者：调用生产者接口实现消费
服务注册：将当前服务地址注册到
服务发现：



Nacos 的基本的介绍

Nacos 可以实现分布式服务注册与发现/分布式配置中心框架。
官网的介绍：https://nacos.io/zh-cn/docs/what-is-nacos.html

Nacos 的环境的准备

Nacos 可以在 linux/windows/Mac 版本上都可以安装
具体安装教程地址：https://nacos.io/zh-cn/docs/quick-start.html

手动实现服务注册与发现

1. 实现服务注册
发送 post 请求：
'http://127.0.0.1:8848/nacos/v1/ns/instance?serviceName=nacos.naming.serviceName&ip=20.18.7.10&port=8080'
2. 实现服务发现
http://127.0.0.1:8848/nacos/v1/ns/instance/list?serviceName=nacos.naming.serviceName
详细步骤操作：https://nacos.io/zh-cn/docs/quick-start.html

Nacos 整合 SpringCloud

Maven 依赖信息

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.RELEASE</version>
</parent>
<dependencies>
    <!--  springboot 整合web 组件-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
        <version>0.2.2.RELEASE</version>
    </dependency>
</dependencies>
```

会员服务(生产者)

服务接口

```
@RestController
public class MemberService {
    @Value("${server.port}")
    private String serverPort;

    /**
     * 会员服务提供的接口
     *
     * @param userId
     * @return
     */
    @RequestMapping("/getUser")
    public String getUser(Integer userId) {
        return "每特教育,端口号:" + serverPort;
    }
}
```

配置文件

application.yml 文件

```
spring:
  cloud:
    nacos:
      discovery:
        ###服务注册地址
        server-addr: 127.0.0.1:8848
  application:
    name: mayikt-member
server:
  port: 8081
```

订单服务(消费者)

订单调用会员服务

```
@RestController
public class OrderService {


    @Autowired
    private RestTemplate restTemplate;
    @Autowired
    private DiscoveryClient discoveryClient;
    @Autowired
    private LoadBalancer loadBalancer;


    /**
     * 订单调用会员服务
     *
     * @return
     */
//    @RequestMapping("/orderToMember")
//    public String orderToMember() {
//        // 从注册中心上获取该注册服务列表
//        List<ServiceInstance> serviceInstanceList = discoveryClient.getInstances("mayikt-
member");
//        ServiceInstance serviceInstance = serviceInstanceList.get(0);
//        URI rpcMemberUrl = serviceInstance.getUri();
//        // 使用本地 rest 形式实现 rpc 调用
//        String result = restTemplate.getForObject(rpcMemberUrl + "/getUser", String.class);
//        return "订单调用会员获取结果:" + result;
//    }
    @RequestMapping("/orderToMember")
    public String orderToMember() {
        // 从注册中心上获取该注册服务列表
        List<ServiceInstance> serviceInstanceList = discoveryClient.getInstances("mayikt-member");
        ServiceInstance serviceInstance = loadBalancer.getSingleAddres(serviceInstanceList);
        URI rpcMemberUrl = serviceInstance.getUri();
        // 使用本地 rest 形式实现 rpc 调用
        String result = restTemplate.getForObject(rpcMemberUrl + "/getUser", String.class);
        return "订单调用会员获取结果:" + result;
    }
}
```

负载均衡算法

```
public interface LoadBalancer {



    /**
     * 根据多个不同的地址 返回单个调用rpc 地址
     *
     * @param serviceInstances
     * @return
     */
    ServiceInstance getSingleAddres(List<ServiceInstance> serviceInstances);
}


@Component
public class RotationLoadBalancer implements LoadBalancer {
    private AtomicInteger atomicInteger = new AtomicInteger(0);


    @Override
    public ServiceInstance getSingleAddres(List<ServiceInstance> serviceInstances) {
        int index = atomicInteger.incrementAndGet() % 2;
        ServiceInstance serviceInstance = serviceInstances.get(index);
        return serviceInstance;
    }
}
```

Nacos 与其他注册对比分析

Nacos 与 Eureka 的区别

Nacos 与 Zookeeper 的区别

Nacos 的集群部署

Nacos 的数据持久化

Nacos 与 Eureka 区别

Eureka 与 Zookeeper 区别

客户端负载均衡器 Ribbon

SpringCloud 负载均衡器说明

在 SpringCloud 第一代中使用 Ribbon、SpringCloud 第二代中直接采用自研发 loadbalancer 即可，默认使用的 Ribbon。
使用方式非常简单：

```
    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
```

LoadBalancerClient 负载均衡器

```
@RequestMapping("/loadBalancerClient")
public Object loadBalancerClient() {
    return loadBalancerClient.choose("meitemayikt-member");
}
```

底层默认原理是调用 ribbon 的实现客户端负载均衡器。

本地负载均衡与 Nginx 的区别

本地负载均衡

本地负载均衡器基本的概念：我们的消费者服务从我们的注册中心获取到集群地址列表，缓存到本地，让后本地采用负载均衡策略（轮训、随机、权重等），实现本地的 rpc 远程的。

本地负载均衡器与 Nginx 的区别

Nginx 是客户端所有的请求统一都交给我们的 Nginx 处理，让后在由 Nginx 实现负载均衡转发，属于服务器端负载均衡器。
本地负载均衡器是从注册中心获取到集群地址列表，本地实现负载均衡算法，既本地负载均衡器。
应用场景的：
Nginx 属于服务器负载均衡，应用于 Tomcat/Jetty 服务器等，而我们的本地负载均衡器，应用于在微服务架构中 rpc 框架中，rest、openfeign、dubbo。

OpenFeign 客户端

OpenFeign 是一个 Web 声明式的 Http 客户端调用工具，提供接口和注解形式调用。

构建微服务项目

mayikt-opefeign-parent---父工程
---mayikt-service-api----微服务 Api 接口层
----mayikt-member-service-api
----mayikt-order-service-api
---mayikt-service-impl----微服务 Api 实现层
----mayikt-member-service-impl
----mayikt-order-service-api

Maven 依赖

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.RELEASE</version>
</parent>



<dependencies>
    <!--  springboot 整合 web 组件-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
        <version>0.2.2.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-openfeign</artifactId>
        <version>2.0.0.RELEASE</version>
    </dependency>
</dependencies>
```

会员服务接口

```java
public interface MemberService {


    /**
     * 提供会员接口
     *
     * @param userId
     * @return
     */
    @GetMapping("/getUser")
    String getUser(@RequestParam("userId") Long userId);
}
```

```java
@RestController
public class MemberServiceImpl implements MemberService {
    @Value("${server.port}")
    private String serverPort;

    @Override
    public String getUser(Long userId) {
        return "我是会员服务端口号为:" + serverPort;
    }
}
```

订单服务

```java
@RestController
public class OrderService {
    @Autowired
    private MemberServiceFeign memberServiceFeign;

    /**
     * 订单调用会员
     *
     * @return
     */
    @GetMapping("/orderToMember")
    public String orderToMember() {
        String result = memberServiceFeign.getUser(10L);
        return "我是订单服务，调用会员服务接口返回结果:" + result;
    }
}
```

```java
@FeignClient(name = "meitemayikt-member")
public interface MemberServiceFeign extends MemberService {
//    /**
//     * 提供会员接口
//     *
//     * @param userId
//     * @return
//     */
//    @GetMapping("/getUser")
```

```
//    String getUser(@RequestParam("userId") Long userId);
}
```

分布式配置中心

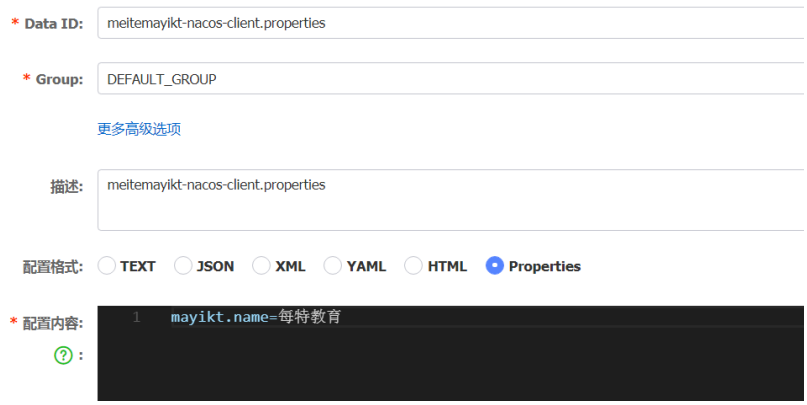
分布式配置中心的作用

分布式配置中心可以实现不需要重启我们的服务器，动态的修改我们的配置文件内容，常见的配置中心有携程的阿波罗、SpringCloud Config、Nacos 轻量级的配置中心等。

基于 Nacos 实现分布式配置中心

服务器端

在 Naocs 平台中创建配置文件　名称（默认为服务器名称）-版本.properties|yaml;



客户端

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
        <version>0.2.2.RELEASE</version>
```

</dependency>

创建一个 bootstrap.yml

```yaml
spring:
  application:
    ###服务的名称
    name: meitemayikt-nacos-client
  cloud:
    nacos:
      discovery:
        ###nacos 注册地址
        server-addr: 127.0.0.1:8848
        enabled: true
      config:
        ###配置中心连接地址
        server-addr: 127.0.0.1:8848
        ###分组
        group: DEFAULT_GROUP
        ###类型
        file-extension: yaml
```

```java
@RestController
@SpringBootApplication
@RefreshScope
public class NacosController {
    @Value("${mayikt.name}")
    private String userName;


    @RequestMapping("/getConfig")
    public String getConfig() {
        return userName;
    }


    public static void main(String[] args) {
        SpringApplication.run(NacosController.class);
    }
}
```

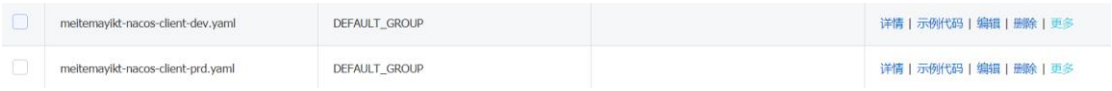
可以实现动态实现@RefreshScope

注意：连接 nacos 分布式配置中心一定采用 bootstrap 形式优先加载 否则可能会报错。

bootstrap.yml 用于应用程序上下文的引导阶段。application
.yml 由父 Spring ApplicationContext 加载。

多版本控制

分别在 nacos 服务器端创建
meitemayikt-nacos-client-dev.yaml
meitemayikt-nacos-client-prd.yaml



客户端指定读取版本

```yaml
spring:
  application:
    ###服务的名称
    name: meitemayikt-nacos-client
  cloud:
    nacos:
      discovery:
        ###nacos注册地址
        server-addr: 127.0.0.1:8848
        enabled: true
      config:
        ###配置中心连接地址
        server-addr: 127.0.0.1:8848
        ###分组
        group: DEFAULT_GROUP
        ###类型
        file-extension: yaml
  profiles:
    active: prd
#server:
```

数据持久化

默认的情况下，分布式配置中心的数据存放到本地 data 目录下，但是这种情况如果 nacos 集群的话无法保证数据的同步性。

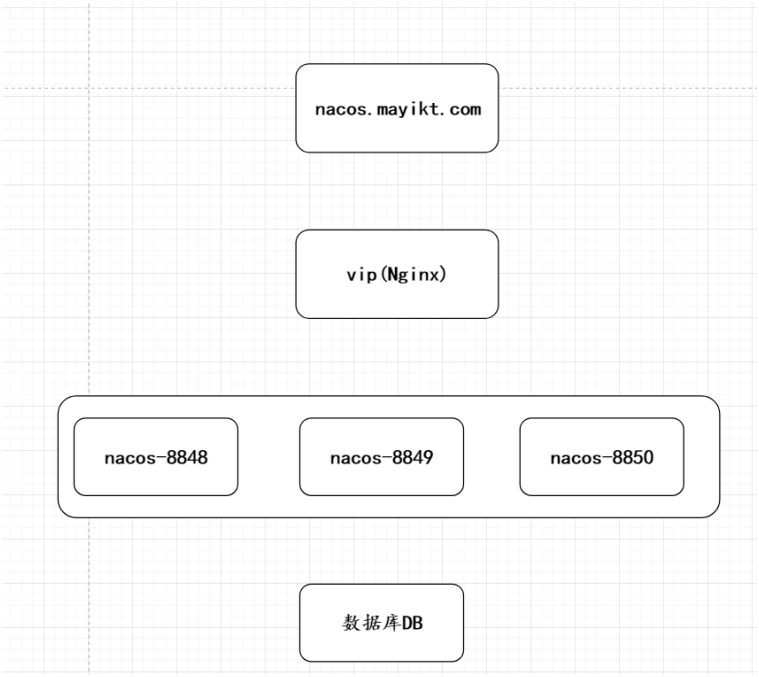
在 0.7 版本之前，在单机模式时 nacos 使用嵌入式数据库实现数据的存储，不方便观察数据存储的基本情况。0.7 版本增加了支持 mysql 数据源能力，具体的操作步骤：

1. 安装数据库，版本要求：5.6.5+
2. 初始化 mysql 数据库，数据库初始化文件：nacos-mysql.sql
3. 修改 conf/application.properties 文件，增加支持 mysql 数据源配置（目前只支持 mysql），添加 mysql 数据源的 url、用户名和密码。

```
spring.datasource.platform=mysql
db.num=1
db.url.0=jdbc:mysql://127.0.0.1:3306/nacos?characterEncoding=utf8&connectTim
eout=1000&socketTimeout=3000&autoReconnect=true
db.user=root
db.password=root
```

摘自官网: https://nacos.io/zh-cn/docs/deployment.html

基于 Nacos 集群部署方案



相关集群配置

创建 cluster 文件夹
---nacos-server-8848
---nacos-server-8849
---nacos-server-8850

cluster.conf
###ip 和端口号
127.0.0.1:8848
127.0.0.1:8849
127.0.0.1:8850

Nginx 相关配置
客户端连接

```yaml
spring:
  application:
    ###服务的名称
    name: meitemayikt-nacos-client
  cloud:
    nacos:
      discovery:
        ###nacos 注册地址
        server-addr: 127.0.0.1:8848,127.0.0.1:8849,127.0.0.1:8850
        enabled: true
      config:
        ###配置中心连接地址
        server-addr: 127.0.0.1:8848,127.0.0.1:8849,127.0.0.1:8850
        ###分组
        group: DEFAULT_GROUP
        ###类型
        file-extension: yaml
```

注意：

1. nacos 在 windows 版本下运行默认是单机版本 需要指定 startup.cmd -m cluster
2. nacos 在 linux 版本下运行默认是集群版本 如果想连接单机版本 startup.cmd -m standalone

节点Ip	节点状态	集群任期	Leader止时(ms)	心跳止时(ms)
192.168.18.218:8849	FOLLOWER	6	16674	3000
192.168.18.218:8850	FOLLOWER	0	14710	2500
192.168.18.218:8848	LEADER	6	12863	3000

Running in cluster mode,

Nacos 对比 Zookeeper、Eureka 之间的区别

CAP 定律

这个定理的内容是指的是在一个分布式系统中、Consistency（一致性）、 Availability（可用性）、Partition tolerance （分区容错性），三者不可得兼。

一致性(C)：在分布式系统中，如果服务器集群，每个节点在同时刻访问必须要保持数据的一致性。
可用性(A)：集群节点中，部分节点出现故障后任然可以使用 （高可用）
分区容错性(P)：在分布式系统中网络会存在脑裂的问题，部分 Server 与整个集群失去节点联系，无法组成一个群体。
只有在 CP 和 AP 选择一个平衡点

Eureka 与 Zookeeper 区别

分布式系统一致性算法

SpringCloud Gateway

什么是微服务网关

微服务网关是整个微服务 API 请求的入口，可以实现日志拦截、权限控制、解决跨域问题、限流、熔断、负载均衡、黑名单与白名单拦截、授权等。

过滤器与网关的区别

过滤器用于拦截单个服务
网关拦截整个的微服务

Zuul 与 Gateway 有那些区别

Zuul 网关属于 netfix 公司开源的产品属于第一代微服务网关
Gateway 属于 SpringCloud 自研发的第二代微服务网关
相比来说 SpringCloudGateway 性能比 Zuul 性能要好:
注意：Zuul 基于 Servlet 实现的，阻塞式的 Api， 不支持长连接。
SpringCloudGateway 基于 Spring5 构建，能够实现响应式非阻塞式的 Api，支持长连接，能

够更好的整合 Spring 体系的产品。

Gateway 环境快速搭建

Maven 依赖

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-gateway</artifactId>
        <version>2.0.0.RELEASE</version>
    </dependency>
</dependencies>
```

application 配置

```
server:
  port: 80
####服务网关名称
spring:
  application:
    name: mayikt-gateway
  cloud:
    gateway:
      discovery:
        locator:
          ####开启以服务 id 去注册中心上获取转发地址
          enabled: true
        ###路由策略
      routes:
        ###路由 id
        - id: mayikt
          ####转发 http://www.mayikt.com/
          uri: http://www.mayikt.com/
```

```
        ###匹配规则
        predicates:
          - Path=/mayikt/**
```

Gateway 整合 Nacos 实现服务转发

Maven 依赖

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-gateway</artifactId>
        <version>2.0.0.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
        <version>0.2.2.RELEASE</version>
    </dependency>
</dependencies>
```

application 配置

```
server:
  port: 80
####服务网关名称
spring:
```

```yaml
application:
  name: mayikt-gateway
cloud:
  gateway:
    discovery:
      locator:
        ####开启以服务id去注册中心上获取转发地址
        enabled: true
      ###路由策略
    routes:
      ###路由id
      - id: mayikt
        ####转发http://www.mayikt.com/
        uri: http://www.mayikt.com/
        ###匹配规则
        predicates:
          - Path=/mayikt/**
      ###路由id
      - id: member
        #### 基于lb负载均衡形式转发
        uri: lb://mayikt-member
        filters:
          - StripPrefix=1
        ###匹配规则
        predicates:
          - Path=/member/**
  nacos:
    discovery:
      server-addr: 127.0.0.1:8848
```

Nginx 与网关的区别

相同点：都是可以实现对 api 接口的拦截，负载均衡、反向代理、请求过滤等，可以实现和网关一样的效果。

不同点：
Nginx 采用 C 语言编写的

微服务都是自己语言编写的 比如 Gateway 就是 java 写的。

毕竟 Gateway 属于 Java 语言编写的， 能够更好对微服务实现扩展功能，相比 Nginx 如果想实现扩展功能需要结合 Nginx+Lua 语言等。

Nginx 实现负载均衡的原理：属于服务器端负载均衡器。
Gateway 实现负载均衡原理：采用本地负载均衡器的形式。

自定义 TokenFilter 实现参数拦截

```
@Component
public class TokenFilter implements GlobalFilter {
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        String token = exchange.getRequest().getQueryParams().getFirst("token");
        if (token == null || token.isEmpty()) {
            ServerHttpResponse response = exchange.getResponse();
            response.setStatusCode(HttpStatus.BAD_REQUEST);
            String msg = "token not is null ";
            DataBuffer buffer = response.bufferFactory().wrap(msg.getBytes());
            return response.writeWith(Mono.just(buffer));
        }
        // 使用网关过滤
        return chain.filter(exchange);
    }
}
```

如何保证微服务接口的安全

接口分为内网和外网接口

外网接口 基于 OATUH2.0 构建开放平台 比如 appid、appsocet 获取 accesstoken 调用接口。
内网接口：都是当前内网中实现通讯，相对于来说比较安全的。

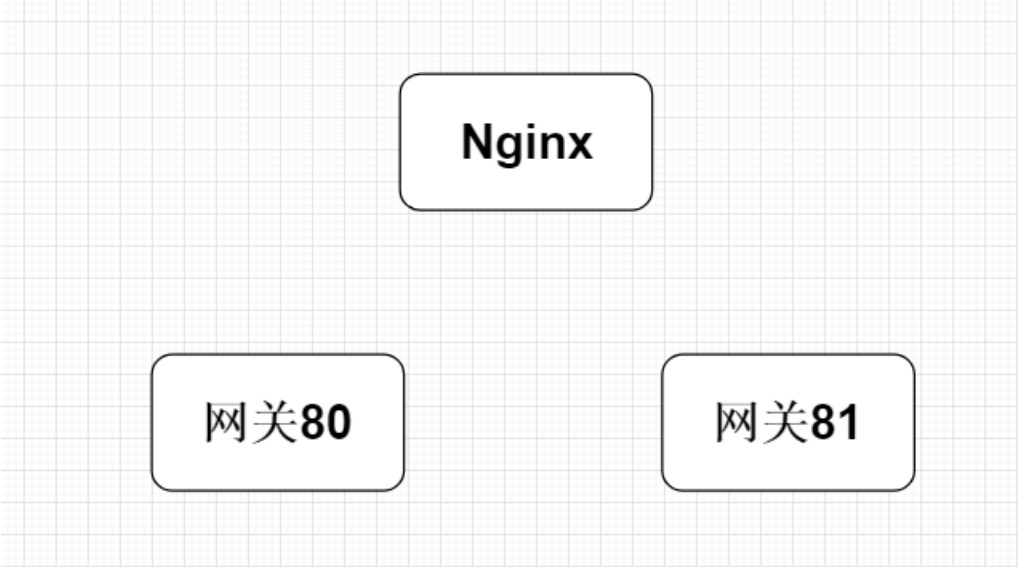
1. 需要保证接口幂等性问题（基于 Token）

2. 接口采用安全加密传输 https 协议
3. 防止数据被篡改 md5 验证签名
4. 使用微服务网关实现 Api 授权认证等、黑名单白名单。
5. 对我们的接口实现服务的保护 隔离、熔断、降级等等。

最后使用 apiswagger 管理我们的微服务接口。

GateWay 如何保证高可用和集群

使用 Nginx 或者 lvs 虚拟 vip 访问增加系统的高可用



网关过滤器相关配置

```
@Value("${server.port}")
private String serverPort;


@Override
public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
    //如何获取参数呢?
    String token = exchange.getRequest().getQueryParams().getFirst("token");
    if (StringUtils.isEmpty(token)) {
        ServerHttpResponse response = exchange.getResponse();
        response.setStatusCode(HttpStatus.INTERNAL_SERVER_ERROR);
        String msg = "token not is null ";
        DataBuffer buffer = response.bufferFactory().wrap(msg.getBytes());
        return response.writeWith(Mono.just(buffer));
    }
    // 在请求头中存放 serverPort serverPort
    ServerHttpRequest request = exchange.getRequest().mutate().header("serverPort", serverPort).build();
```

```
    return chain.filter(exchange.mutate().request(request).build());
}
```

Nginx 相关配置

```
    upstream mayiktgwadds {
    server 127.0.0.1:81;
    server 127.0.0.1:82;
}

server {
    listen 80;
    server_name    gw.mayikt.com;
    location / {
        proxy_pass http://mayiktgwadds/;
    }
}
```

动态请求参数网关

方案：
1. 基于数据库形式实现
2. 基于配置中心实现

注意：配置中心实现维护性比较差，建议采用数据库形式设计。

网关服务相关表

```
CREATE TABLE `mayikt_gateway` (
    `id` int(11) NOT NULL AUTO_INCREMENT,
    `route_id` varchar(11) DEFAULT NULL,
    `route_name` varchar(255) DEFAULT NULL,
    `route_pattern` varchar(255) DEFAULT NULL,
    `route_type` varchar(255) DEFAULT NULL,
    `route_url` varchar(255) DEFAULT NULL,
```

```
    PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=latin1;
```

```java
@Service
public class GatewayService implements ApplicationEventPublisherAware {
    private ApplicationEventPublisher publisher;
    @Autowired
    private RouteDefinitionWriter routeDefinitionWriter;
    @Autowired
    private MayiktGatewayMapper mayiktGateway;

    @Override
    public void setApplicationEventPublisher(ApplicationEventPublisher applicationEventPublisher) {
        this.publisher = applicationEventPublisher;
    }

    public void initAllRoute() {
        // 从数据库查询配置的网关配置
        List<GateWayEntity> gateWayEntities = mayiktGateway.gateWayAll();
        for (GateWayEntity gw :
                gateWayEntities) {
            loadRoute(gw);
        }

    }

    public String loadRoute(GateWayEntity gateWayEntity) {
        RouteDefinition definition = new RouteDefinition();
        Map<String, String> predicateParams = new HashMap<>(8);
        PredicateDefinition predicate = new PredicateDefinition();
        FilterDefinition filterDefinition = new FilterDefinition();
        Map<String, String> filterParams = new HashMap<>(8);
        // 如果配置路由type 为0 的话 则从注册中心获取服务
        URI uri = null;
        if (gateWayEntity.getRouteType().equals("0")) {
            uri = uri = UriComponentsBuilder.fromUriString("lb://" + gateWayEntity.getRouteUrl() +
"/").build().toUri();
        } else {
            uri = UriComponentsBuilder.fromHttpUrl(gateWayEntity.getRouteUrl()).build().toUri();
        }
        // 定义的路由唯一的id
        definition.setId(gateWayEntity.getRouteId());
```

```
        predicate.setName("Path");
        //路由转发地址
        predicateParams.put("pattern", gateWayEntity.getRoutePattern());
        predicate.setArgs(predicateParams);

        // 名称是固定的，路径去前缀
        filterDefinition.setName("StripPrefix");
        filterParams.put("_genkey_0", "1");
        filterDefinition.setArgs(filterParams);
        definition.setPredicates(Arrays.asList(predicate));
        definition.setFilters(Arrays.asList(filterDefinition));
        definition.setUri(uri);
        routeDefinitionWriter.save(Mono.just(definition)).subscribe();
        this.publisher.publishEvent(new RefreshRoutesEvent(this));
        return "success";
    }


}
```

实体类&访问层

```
public interface MayiktGatewayMapper {


    @Select("SELECT ID AS ID, route_id as routeid, route_name as routeName,route_pattern as routePattern\n" +
            ",route_type as routeType,route_url as routeUrl\n" +
            " FROM mayikt_gateway\n")
    public List<GateWayEntity> gateWayAll();


    @Update("update mayikt_gateway set route_url=#{routeUrl} where route_id=#{routeId};")
    public Integer updateGateWay(@Param("routeId") String routeId, @Param("routeUrl") String routeUrl);
}
```

Maven 依赖

```
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
```

```
    <version>1.1.1</version>
</dependency>
<!-- mysql 依赖 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
<!-- 阿里巴巴数据源 -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.0.14</version>
</dependency>
```

```
### 127.0.0.1/mayikt  转到到http://www.mayikt.com/
  datasource:
    url: jdbc:mysql://localhost:3306/meite_gateWay?useUnicode=true&characterEncoding=UTF-8
    username: root
    password: root
    driver-class-name: com.mysql.jdbc.Driver
```

GateWay 的词汇表有那些

路由：是网关基本的模块，分别为 id、目标 uri、一组谓词+过滤器一起组合而成，如果谓词匹配成功，则路由匹配成功。
谓词： 匹配 Http 请求参数
过滤器：对下游的服务器之前和之后实现处理。

1. 匹配时间之后

```
- id: mayikt
  uri: http://www.mayikt.com/
  ###匹配规则
  predicates:
    - After=2017-01-20T17:42:47.789-07:00[America/Denver]
```

此路由与 2017 年 1 月 20 日 17:42 MountainTime（Denver）之后的所有请求相匹配。

2. 匹配对应的 host

```
- id: meite
  uri: http://www.mayikt.com/
  ###匹配规则
  predicates:
    - Host=meite.mayikt.com
```

访问 mete.mayikt.com 转发到 http://www.mayikt.com/

3. 权重谓词

```
- id: weight_high
  uri: http://www.mayikt.com/yushengjun
  predicates:
    - Weight=group1, 2
- id: weight_low
  uri: http://www.mayikt.com
  predicates:
    - Weight=group1, 1
```

根据权重比例实现转发

```
- id: weight_order
  uri: lb://meitemayikt-order
  predicates:
    - Weight=group1,2
- id: weight_member
  uri: lb://mayikt-member
  predicates:
    - Weight=group1,1
```

详细参考：
https://cloud.spring.io/spring-cloud-gateway/reference/html/#gatewayfilter-factories

GateWay 解决跨域的问题

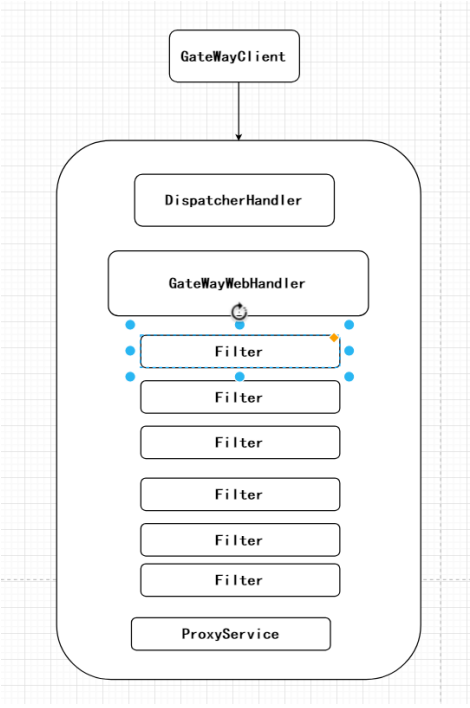
```
*/
@Component
public class CrossOriginFilter implements GlobalFilter {
    @Override
```

```
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        ServerHttpRequest request = exchange.getRequest();
        ServerHttpResponse response = exchange.getResponse();
        HttpHeaders headers = response.getHeaders();
        headers.add(HttpHeaders.ACCESS_CONTROL_ALLOW_ORIGIN, "*");
        headers.add(HttpHeaders.ACCESS_CONTROL_ALLOW_METHODS, "POST, GET, PUT, OPTIONS, DELETE, PATCH");
        headers.add(HttpHeaders.ACCESS_CONTROL_ALLOW_CREDENTIALS, "true");
        headers.add(HttpHeaders.ACCESS_CONTROL_ALLOW_HEADERS, "*");
        headers.add(HttpHeaders.ACCESS_CONTROL_EXPOSE_HEADERS, "*");
        return chain.filter(exchange);

    }
}
```

网关 GateWay 源码分析

1. 客户端向网关发送 Http 请求，会到达 DispatcherHandler 接受请求，匹配到 RoutePredicateHandlerMapping。

1. 根据 RoutePredicateHandlerMapping 匹配到具体的路由策略。
2. FilteringWebHandler 获取的路由的 GatewayFilter 数组，创建 GatewayFilterChain 处理过滤请求
3. 执行我们的代理业务逻辑访问。



常用配置类说明:

1. GatewayClassPathWarningAutoConfiguration 检查是否有正确的配置 webflux
2. GatewayAutoConfiguration 核心配置类
3. GatewayLoadBalancerClientAutoConfiguration 负载均衡策略处理
4. GatewayRedisAutoConfiguration Redis+lua 整合限流

常见错误

Parameter 0 of method modifyRequestBodyGatewayFilterFactory in org.springframework.cloud.gateway.config.GatewayAutoConfiguration required a bean of type 'org.springframework.http.codec.ServerCodecConfigurer' that could not be found.

原因就是
SpringCloud gateway 基于 webflux 实现的，不是基于 SpringBoot-web，所以应该删除 Springboot-web 依赖组件。

SpringCloud gateway 常用名词

Route 路由 路由的 id （唯一）、 转发 uri （真实服务地址）、过滤器、谓词组成。
谓词 匹配的规则

源码分析:

SpringBoot 项目源码的入口

1. GatewayClassPathWarningAutoConfiguration 作用检查是否配置我们 webfux 依赖。
2. GatewayAutoConfiguration 加载了我们 Gateway 需要的注入的类。
3. GatewayLoadBalancerClientAutoConfiguration 网关需要使用的负载均衡 Lb//mayikt-member// 根据服务名称查找真实地址
4. GatewayRedisAutoConfiguration 网关整合 Redis 整合 Lua 实现限流
5. GatewayDiscoveryClientAutoConfiguration 服务注册与发现功能

微服务中跨域的问题 不属于前端解决 jsonp 只能支持 get 请求。

核心点就是在我们后端。

解决跨域的问题
1. HttpClient 转发
2. 使用过滤器允许接口可以跨域 响应头设置
3. Jsonp 不支持我们的 post 属于前端解决
4. Nginx 解决跨域的问题保持我们域名和端口一致性
5. Nginx 也是通过配置文件解决跨域的问题
6. 基于微服务网关解决跨域问题，需要保持域名和端口一致性
7. 使用网关代码允许所有的服务可以跨域的问题
8. 使用 SpringBoot 注解形式@CrossOrigin

SpringCloud Sentinel

服务保护的基本概念

服务限流/熔断

服务限流目的是为了更好的保护我们的服务，在高并发的情况下，如果客户端请求的数量达到一定极限（后台可以配置阈值），请求的数量超出了设置的阈值，开启自我的保护，直接调用我们的服务降级的方法，不会执行业务逻辑操作，直接走本地 falback 的方法，返回一个友好的提示。

服务降级

在高并发的情况下， 防止用户一直等待，采用限流/熔断方法，使用服务降级的方式返回一个友好的提示给客户端，不会执行业务逻辑请求，直接走本地的 falback 的方法。
提示语：当前排队人数过多，稍后重试~

服务的雪崩效应

默认的情况下，Tomcat 或者是 Jetty 服务器只有一个线程池去处理客户端的请求，
这样的话就是在高并发的情况下，如果客户端所有的请求都堆积到同一个服务接口上，
那么就会产生 tomcat 服务器所有的线程都在处理该接口，可能会导致其他的接口无法访问。

假设我们的 tomcat 线程最大的线程数量是为 20，这时候客户端如果同时发送 100 个请求会导致有 80 个请求暂时无法访问，就会转圈。

服务的隔离的机制

服务的隔离机制分为信号量和线程池隔离模式
服务的线程池隔离机制：每个服务接口都有自己独立的线程池，互不影响，缺点就是占用 cpu 资源非常大。
服务的信号量隔离机制：最多只有一定的阈值线程数处理我们的请求，超过该阈值会拒绝请求。

Sentinel 与 hytrix 区别

前哨以流量为切入点，从流量控制，熔断降级，系统负载保护等多个维度保护服务的稳定性。

前哨具有以下特征：

1. 丰富的应用场景：前哨兵承接了阿里巴巴近 10 年的双十一大促流的核心场景，例如秒杀（即突然流量控制在系统容量可以承受的范围），消息削峰填谷，传递流量控制，实时熔断下游不可用应用等。
2. 完备的实时监控：Sentinel 同时提供实时的监控功能。您可以在控制台中看到接收应用的单台机器秒级数据，甚至 500 台以下规模的整合的汇总运行情况。
广泛的开源生态：Sentinel 提供开箱即用的与其他开源框架/库的集成模块，例如与 Spring Cloud，Dubbo，gRPC 的整合。您只需要另外的依赖并进行简单的配置即可快速地接入 Sentinel。
3. 完善的 SPI 扩展点：Sentinel 提供简单易用，完善的 SPI 扩展接口。您可以通过实现扩展接口来快速地定制逻辑。例如定制规则管理，适应动态数据源等。

Sentinel 中文文档介绍：
https://github.com/alibaba/Sentinel/wiki/%E4%BB%8B%E7%BB%8D

	Sentinel	Hystrix
隔离策略	信号量隔离	线程池隔离/信号量隔离
熔断降级策略	基于响应时间或失败比率	基于失败比率
实时指标实现	滑动窗口	滑动窗口（基于 RxJava）
规则配置	支持多种数据源	支持多种数据源
扩展性	多个扩展点	插件的形式
基于注解的支持	支持	支持
限流	基于 QPS，支持基于调用关系的限流	不支持
流量整形	支持慢启动、匀速器模式	不支持
系统负载保护	支持	不支持
控制台	开箱即用，可配置规则、查看秒级监控、机器发现等	不完善
常见框架的适配	Servlet、Spring Cloud、Dubbo、gRPC 等	Servlet、Spring Cloud Netflix

Sentinel 实现对 Api 动态限流

SpringBoot 项目整合 Sentinel

Maven 依赖的配置

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-alibaba-sentinel</artifactId>
    <version>0.2.2.RELEASE</version>

</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

手动配置管理 Api 限流接口

```
private static final String GETORDER_KEY = "getOrder";


@RequestMapping("/initFlowQpsRule")
public String initFlowQpsRule() {
    List<FlowRule> rules = new ArrayList<FlowRule>();
    FlowRule rule1 = new FlowRule();
    rule1.setResource(GETORDER_KEY);
    // QPS 控制在 2 以内
    rule1.setCount(1);
    // QPS 限流
    rule1.setGrade(RuleConstant.FLOW_GRADE_QPS);
    rule1.setLimitApp("default");
    rules.add(rule1);
    FlowRuleManager.loadRules(rules);
    return "....限流配置初始化成功..";
}


@RequestMapping("/getOrder")
public String getOrders() {
    Entry entry = null;
    try {
        entry = SphU.entry(GETORDER_KEY);
        // 执行我们服务需要保护的业务逻辑
        return "getOrder 接口";
    } catch (Exception e) {
        e.printStackTrace();
        return "该服务接口已经达到上线!";
    } finally {
        // SphU.entry(xxx) 需要与 entry.exit() 成对出现,否则会导
致调用链记录异常
        if (entry != null) {
            entry.exit();
```

```
        }
    }

}
```

手动放入到项目启动自动加载

```
@Component
@Slf4j
public class SentinelApplicationRunner implements ApplicationRunner {
    private static final String GETORDER_KEY = "getOrder";

    @Override
    public void run(ApplicationArguments args) throws Exception {
        List<FlowRule> rules = new ArrayList<FlowRule>();
        FlowRule rule1 = new FlowRule();
        rule1.setResource(GETORDER_KEY);
        // QPS 控制在 2 以内
        rule1.setCount(1);
        // QPS 限流
        rule1.setGrade(RuleConstant.FLOW_GRADE_QPS);
        rule1.setLimitApp("default");
        rules.add(rule1);
        FlowRuleManager.loadRules(rules);
        log.info(">>>限流服务接口配置加载成功>>>");
    }
}
```

注解形式配置管理 Api 限流

@SentinelResource value 参数：流量规则资源名称、
blockHandler 限流/熔断出现异常执行的方法
Fallback 服务的降级执行的方法

```
@SentinelResource(value = GETORDER_KEY, blockHandler =
"getOrderQpsException")
@RequestMapping("/getOrderAnnotation")
```

```java
public String getOrderAnnotation() {
    return "getOrder 接口";
}

/**
 * 被限流后返回的提示
 *
 * @param e
 * @return
 */
public String getOrderQpsException(BlockException e) {
    e.printStackTrace();
    return "该接口已经被限流啦!";
}
```

控制台形式管理限流接口

Sentinel dashboard 控制台选择创建流量规则，设置资源名称（服务接口地址）、设置 QPS 为 1 表示每 s 最多能够访问 1 次接口。

Sentinel 环境快速搭建

下载对应 Sentinel-Dashboard
https://github.com/alibaba/Sentinel/releases/tag/1.7.1 运行即可。
运行执行命令

```
java -Dserver.port=8718 -Dcsp.sentinel.dashboard.server=localhost:8718 -
```

Dproject.name=sentinel-dashboard -Dcsp.sentinel.api.port=8719 -jar
8718 属于 界面端口号 8719 属于 api 通讯的端口号

```
/**
 * 被限流后返回的提示
 *
 * @param e
 * @return
 */
public String getOrderQpsException(BlockException e) {
    e.printStackTrace();
    return "该接口已经被限流啦!";
}



@SentinelResource(value = "getOrderDashboard", blockHandler = "getOrderQpsException")
@RequestMapping("/getOrderDashboard")
public String getOrderDashboard() {
    return "getOrderDashboard";
}
```

SpringBoot 整合 Sentinel 仪表盘 配置

```
spring:
  application:
    ###服务的名称
    name: meitemayikt-order

  cloud:
    nacos:
      discovery:
        ###nacos 注册地址
        server-addr: 127.0.0.1:8848
    sentinel:
      transport:
        dashboard: 127.0.0.1:8718
      eager: true
```

基于并发数量处理限流

每次最多只会有一个线程处理该业务逻辑，超出该阈值的情况下，直接拒绝访问。

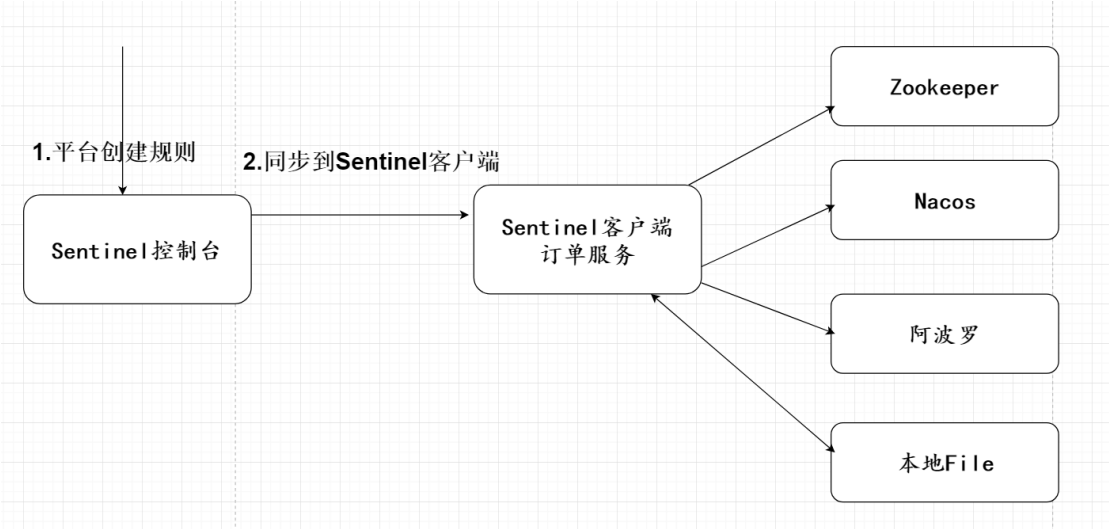
```
@SentinelResource(value = "getOrderThrad", blockHandler = "getOrderQpsException")
@RequestMapping("/getOrderThrad")
public String getOrderThrad() {
    System.out.println(Thread.currentThread().getName());
    try {
        Thread.sleep(1000);
    } catch (Exception e) {

    }
    return "getOrderThrad";
}
```

Sentinel 如何保证规则的持久化

默认的情况下 Sentinel 的规则是存放在内存中，如果 Sentinel 客户端重启后，Sentinel 数据规则可能会丢失。
解决方案：
Sentinel 持久化机制支持四种持久化的机制。

1. 本地文件
2. 携程阿波罗
3. Nacos
4. Zookeeper

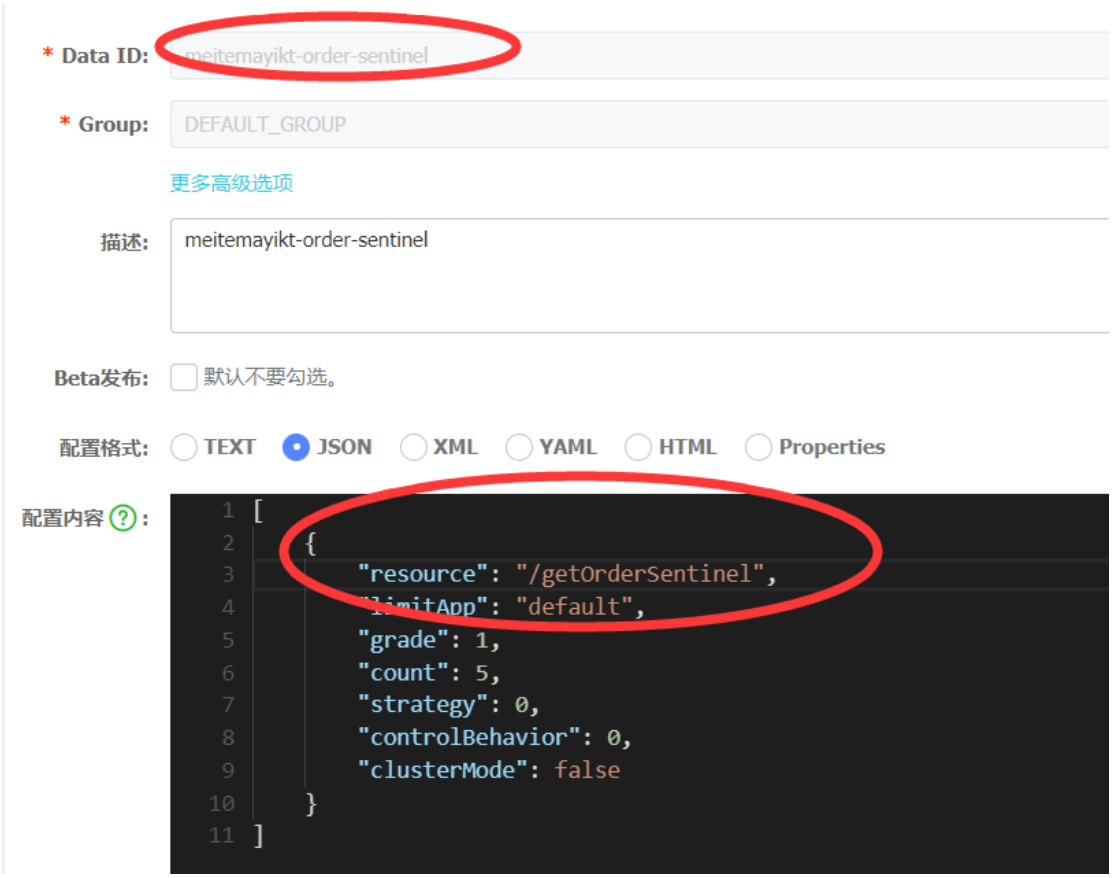
```
g.FileDataSourceProperties file;
g.NacosDataSourceProperties nacos;
g.ZookeeperDataSourceProperties zk;
g.ApolloDataSourceProperties apollo;
```



基于 Nacos 持久化我们的数据规则

Nacos 平台中创建我们的流控规则

meitemayikt-order-sentinel




```
[
    {
        "resource": "/ getOrderSentinel",
        "limitApp": "default",
        "grade": 1,
        "count": 5,
        "strategy": 0,
        "controlBehavior": 0,
        "clusterMode": false
    }
]
```

resource：资源名，即限流规则的作用对象
limitApp：流控针对的调用来源，若为 default 则不区分调用来源
grade：限流阈值类型（QPS 或并发线程数）；0 代表根据并发数量来限流，1 代表根据 QPS 来进行流量控制
count：限流阈值
strategy：调用关系限流策略

controlBehavior：流量控制效果（直接拒绝、Warm Up、匀速排队）
clusterMode：是否为集群模式

```
@SentinelResource(value = "getOrderSentinel", blockHandler = "getOrderQpsException")
@RequestMapping("/getOrderSentinel")
public String getOrderSentinel() {
    return "getOrderSentinel";
}
```

SpringBoot 客户端整合

```
<!--sentinel 整合nacos -->
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-datasource-nacos</artifactId>
    <version>1.5.2</version>
</dependency>
```

相关配置

```
sentinel:
  transport:
    dashboard: 127.0.0.1:8718
  eager: true
  datasource:
    ds:
      nacos:
        ### nacos 连接地址
        server-addr: localhost:8848
        ## nacos 连接的分组
        group-id: DEFAULT_GROUP
        ###路由存储规则
        rule-type: flow
        ### 读取配置文件的 data-id
        data-id: meitemayikt-order-sentinel
        ###  读取培训文件类型为json
        data-type: json
```

SpringCloud 网关如何整合 sentinel 实现限流

查看到 sentinel 中文社区文档
https://github.com/alibaba/Sentinel/wiki/%E4%BB%8B%E7%BB%8D
https://github.com/alibaba/Sentinel/wiki/%E7%BD%91%E5%85%B3%E9%99%90%E6%B5%81

相关核心配置

Maven 依赖配置

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-gateway</artifactId>
        <version>2.0.0.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba.csp</groupId>
        <artifactId>sentinel-spring-cloud-gateway-adapter</artifactId>
        <version>1.6.0</version>
    </dependency>
```

```
gateway:
  routes:
    - id: my-member
      uri: lb://meitemayikt-member
      predicates:
        - Path=/meitemayikt-member/**
    - id: mayikt
      uri: http://www.mayikt.com
      predicates:
        - Path=/mayikt/**
```

```
import com.alibaba.csp.sentinel.adapter.gateway.sc.SentinelGatewayFilter;
import com.alibaba.csp.sentinel.adapter.gateway.sc.exception.SentinelGatewayBlockExceptionHandler;
import org.springframework.beans.factory.ObjectProvider;
import org.springframework.cloud.gateway.filter.GlobalFilter;
```

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.Ordered;
import org.springframework.core.annotation.Order;
import org.springframework.http.codec.ServerCodecConfigurer;
import org.springframework.web.reactive.result.view.ViewResolver;

import java.util.Collections;
import java.util.List;

@Configuration
public class GatewayConfiguration {

    private final List<ViewResolver> viewResolvers;
    private final ServerCodecConfigurer serverCodecConfigurer;

    public GatewayConfiguration(ObjectProvider<List<ViewResolver>> viewResolversProvider,
                                ServerCodecConfigurer serverCodecConfigurer) {
        this.viewResolvers = viewResolversProvider.getIfAvailable(Collections::emptyList);
        this.serverCodecConfigurer = serverCodecConfigurer;
    }

    @Bean
    @Order(Ordered.HIGHEST_PRECEDENCE)
    public SentinelGatewayBlockExceptionHandler sentinelGatewayBlockExceptionHandler() {
        // Register the block exception handler for Spring Cloud Gateway.
        return new SentinelGatewayBlockExceptionHandler(viewResolvers, serverCodecConfigurer);
    }

    @Bean
    @Order(Ordered.HIGHEST_PRECEDENCE)
    public GlobalFilter sentinelGatewayFilter() {
        return new SentinelGatewayFilter();
    }
}
```

加载网关流控规则

```java
@Slf4j
@Component
public class SentinelApplicationRunner implements ApplicationRunner {

    @Override
```

```
    public void run(ApplicationArguments args) throws Exception {
        initGatewayRules();

    }

    /**
     * 配置限流规则
     */
    private void initGatewayRules() {
        Set<GatewayFlowRule> rules = new HashSet<>();
        rules.add(new GatewayFlowRule("mayikt")
                // 限流阈值
                .setCount(1)
                // 统计时间窗口，单位是秒，默认是 1 秒
                .setIntervalSec(1)
        );
        GatewayRuleManager.loadRules(rules);
    }
}
```

如何修改限流错误提示

```
public class JsonSentinelGatewayBlockExceptionHandler implements
WebExceptionHandler {
    public JsonSentinelGatewayBlockExceptionHandler(List<ViewResolver>
viewResolvers, ServerCodecConfigurer serverCodecConfigurer) {
    }

    @Override
    public Mono<Void> handle(ServerWebExchange exchange, Throwable ex) {
        ServerHttpResponse serverHttpResponse = exchange.getResponse();
        serverHttpResponse.getHeaders().add("Content-Type",
"application/json;charset=UTF-8");
        byte[] datas = "{\"code\":403,\"msg\":\"API 接口被限流
\"}".getBytes(StandardCharsets.UTF_8);
        DataBuffer buffer = serverHttpResponse.bufferFactory().wrap(datas);
        return serverHttpResponse.writeWith(Mono.just(buffer));
    }
}
```

```java
@Bean
@Order(Ordered.HIGHEST_PRECEDENCE)
public JsonSentinelGatewayBlockExceptionHandler sentinelGatewayBlockExceptionHandler() {
    // Register the block exception handler for Spring Cloud Gateway.
    return new JsonSentinelGatewayBlockExceptionHandler(viewResolvers, serverCodecConfigurer);
}
```

sentinel 实现熔断降级

https://github.com/alibaba/Sentinel/wiki/%E7%86%94%E6%96%AD%E9%99%8D%E7%BA%A7

除了流量控制以外，对调用链路中不稳定的资源进行熔断降级也是保障高可用的重要措施之一。由于调用关系的复杂性，如果调用链路中的某个资源不稳定，最终会导致请求发生堆积。Sentinel 熔断降级会在调用链路中某个资源出现不稳定状态时（例如调用超时或异常比例升高），对这个资源的调用进行限制，让请求快速失败，避免影响到其它的资源而导致级联错误。当资源被降级后，在接下来的降级时间窗口之内，对该资源的调用都自动熔断（默认行为是抛出 DegradeException）。

降级的策略

1.平均响应时间 (DEGRADE_GRADE_RT)：当 1s 内持续进入 5 个请求，对应时刻的平均响应时间（秒级）均超过阈值（count，以 ms 为单位），那么在接下的时间窗口（DegradeRule 中的 timeWindow，以 s 为单位）之内，对这个方法的调用都会自动地熔断（抛出 DegradeException）。注意 Sentinel 默认统计的 RT 上限是 4900 ms，超出此阈值的都会算作 4900 ms，若需要变更此上限可以通过启动配置项 -Dcsp.sentinel.statistic.max.rt=xxx 来配置。
2.异常比例 (DEGRADE_GRADE_EXCEPTION_RATIO)：当资源的每秒请求量 >= 5，并且每秒异常总数占通过量的比值超过阈值（DegradeRule 中的 count）之后，资源进入降级状态，即在接下的时间窗口（DegradeRule 中的 timeWindow，以 s 为单位）之内，对这个方法的调用都会自动地返回。异常比率的阈值范围是 [0.0, 1.0]，代表 0% - 100%。
3.异常数 (DEGRADE_GRADE_EXCEPTION_COUNT)：当资源近 1 分钟的异常数目超过阈值之后会进行熔断。注意由于统计时间窗口是分钟级别的，若 timeWindow 小于 60s，则结束熔断状态后仍可能再进入熔断状态。

平均的响应时间

如果在 1s 秒，平均有 5 个请求的响应时间大于配置的 10rt 毫秒时间 阈值，则会执行一定时间窗口的熔断和降级。

```
@SentinelResource(value = "getOrderDowngradeRtType", fallback = "getOrderDowngradeRtTypeFallback")
@RequestMapping("/getOrderDowngradeRtType")
public String getOrderDowngradeRtType() {
    try {
        Thread.sleep(300);
    } catch (Exception e) {
    }
    return "getOrderDowngradeRtType";
}

public String getOrderDowngradeRtTypeFallback() {
    return "服务降级啦，当前服务器请求次数过多，请稍后重试!";
}
```

异常的比例

当我们每秒的请求大于 5 的时候，会根据一定比例执行我们的熔断降级的策略。

```
@SentinelResource(value = "getOrderDowngradeErrorType", fallback =
"getOrderDowngradeErrorTypeFallback")
@RequestMapping("/getOrderDowngradeErrorType")
public String getOrderDowngradeErrorType(int age) {
    int j = 1 / age;
    return "正常执行我们的业务逻辑";
}

public String getOrderDowngradeErrorTypeFallback(int age) {
    return "服务降级啦，当前服务器请求次数过多，请稍后重试!";
}
```

异常的次数

当资源近 1 分钟的异常数目超过阈值之后会进行熔断。注意由于统计时间窗口是分钟级别的，若 timeWindow 小于 60s，则结束熔断状态后仍可能再进入熔断状态。

Sentinel 实现热点词限流

https://github.com/alibaba/Sentinel/wiki/%E7%83%AD%E7%82%B9%E5%8F%82%E6%95%B0%E9%99%90%E6%B5%81
可以根据访问频繁的参数实现限流。

热点参数限流

fallback 与 blockHandler 的区别

fallback 是服务熔断或者业务逻辑出现异常执行的方法（1.6 版本以上）
blockHandler 限流出现错误执行的方法。

手动形式创建限流

```
@RestController
@Slf4j
public class SeckillServiceImpl {


    public SeckillServiceImpl() {
        initSeckillRule();
    }


    /**
     * 秒杀路由资源
     */
    private static final String SEKILL_RULE = "seckill";


    /**
     * 秒杀抢购
     *
```

```
     * @return
     */
    @RequestMapping("/seckill")
    public String seckill(Long userId, Long orderId) {
        try {
            Entry entry = SphU.entry(SEKILL_RULE, EntryType.IN, 1, userId);
            return "秒杀成功";
        } catch (Exception e) {
            return "当前用户访问过度频繁，请稍后重试!";
        }
    }
    // seckill?userId=123456&orderId=644064779
    // seckill?userId=123456&orderId=644064779

    private void initSeckillRule() {
        ParamFlowRule rule = new ParamFlowRule(SEKILL_RULE)
                // 对我们秒杀接口第 0 个参数实现限流
                .setParamIdx(0)
                .setGrade(RuleConstant.FLOW_GRADE_QPS)
                // 每秒 QPS 最多只有 1s
                .setCount(1);
        ParamFlowRuleManager.loadRules(Collections.singletonList(rule));
        log.info(">>>秒杀接口限流策略配置成功<<<");
    }
}
```

控制台自定义形式

```
@RequestMapping("/seckill")
@SentinelResource(value = SEKILL_RULE, fallback = "seckillFallback", blockHandler = "seckillBlockHandler")
public String seckill(Long userId, Long orderId) {
    return "秒杀成功";
}
```

参数索引表示我们方法传递的第一个参数

使用全局捕获异常捕获修改限流出现错误

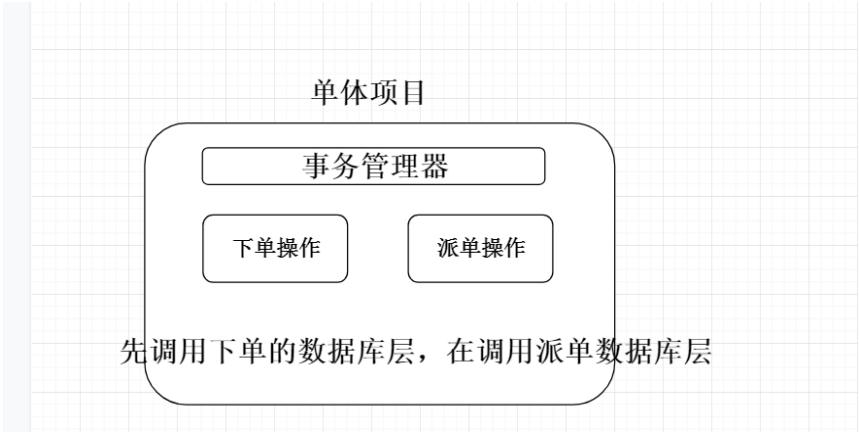
```
@RestControllerAdvice
public class InterfaceExceptionHandler {
    @ResponseBody
    @ExceptionHandler(ParamFlowException.class)
    public String businessInterfaceException(ParamFlowException e) {
        return "您当前访问的频率过高，请稍后重试!";
    }
}
```

SpringCloud 解决分布式事务

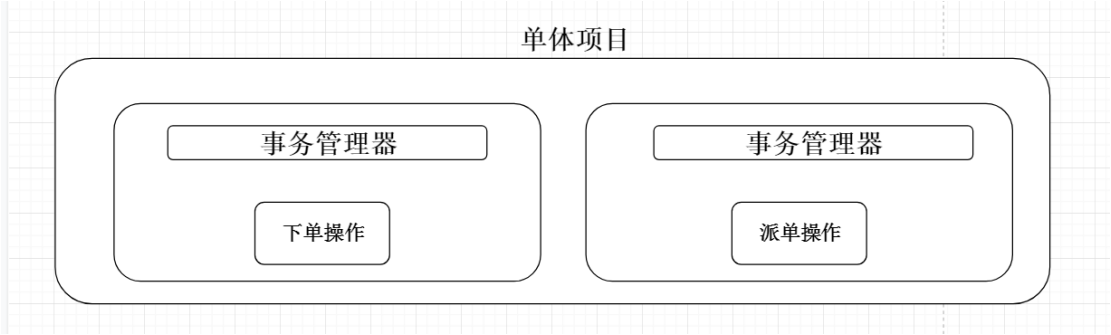
分布式事务产生的背景

分情况来定。

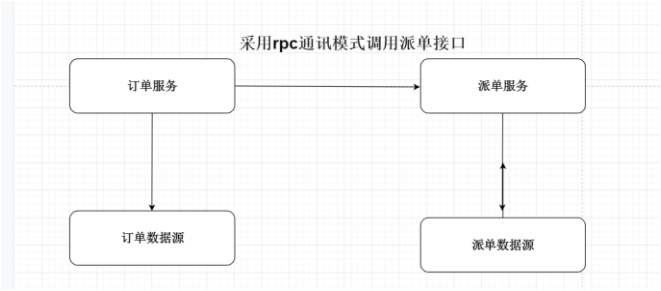
1. 在单体的项目中，多个不同业务逻辑都是在同一个数据源中实现事务管理，是不存在分布式事务的问题，因为同一数据源的情况下都是采用事务管理器，相当于每个事务管理器对应一个数据源。



2. 在单体的项目中，有多个不同的数据源，每个数据源中都有自己独立的事务管理器，互不影响，那么这时候也会存在多数据源事务管理：解决方案 jta+ Atomikos



3. 在分布式/微服务架构中，每个服务都有自己的本地的事务，每个服务本地事务互不影响，那么这时候也会存在分布式事务的问题。



事务的定义

对我们的业务逻辑可以实现提交或者回滚，保证数据的一致性的情况。
所以要么提交，要么回滚。

原子性 a　要么提交 要么回滚
一致性 c
隔离性 i 多个事务在一起执行的时候，互不影响；
持久性 d 事务一旦提交或者回滚后，不会在对该结果有任何影响

Base 与 CAP 理论

这个定理的内容是指的是在一个分布式系统中、Consistency（一致性）、 Availability（可用性）、Partition tolerance （分区容错性），三者不可得兼。

一致性 C：在分布式系统中，同一时刻所有的节点的数据都是相同的；
可用性 A： 集群中部分节点出现了故障，集群的整体也能够给响应；
分区容错性 P：分区容错性是指系统能够容忍节点之间的网络通信的故障，意味着发生了分区的情况，必须就当前操作在 C 和 A 之间做出选择；

BASE 是 Basically Available（基本可用）、Soft state（软状态）和 Eventually consistent（最终一致性）三个短语的缩写

目前主流分布式解决框架

1. 单体项目多数据源 可以 jta+ Atomikos
2. 基于 rabbitmq 的形式解决 最终一致性的思想
3. 基于 rocketmq 解决分布式事务 采用事务消息
4. LCN 采用 lcn 模式 假关闭连接 （目前已经被淘汰）
5. Alibaba 的 Seata 未来可能是主流 背景非常强大

两阶段提交协议基本概念

两阶段提交协议可以理解为 2pc，也就是分为参与者和协调者，协调者会通过两次阶段实现数据最终的一致性的。

SIT 环境 测试环境
PRD 环境

2PC 和 3pc 的区别就是解决参与者超时的问题和多加了一层询问，保证数据的传输可靠性。

简单的回顾一下 LCN 解决分布式事务

LCN 官网基本介绍

http://www.txlcn.org/zh-cn/ LCN 并不生产事务，LCN 只是本地事务的协调工

LCN 基本实现原理

1. 发起方与参与方都与我们的 LCN 管理器一直保持长连接；
2. 发起方在调用接口之前，先向 LCN 管理器申请一个全局的事务分组 id；
3. 发起方调用接口的时候在请求头中传递事务分组 id；
4. 参与方获取到请求头中有事务分组的 id 的，则当前业务逻辑执行完实现假关闭，不会提交或者回滚当前的事务。
5. 发起方调用完接口后，如果出现异常的情况下，在通知给事务协调者回滚事务，这时候事务协调则告诉给参与方回滚当前的事务。

SpringBoot 整合 lcn5.0

Maven 依赖

```
<dependency>
    <groupId>com.codingapi.txlcn</groupId>
    <artifactId>txlcn-tc</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>com.codingapi.txlcn</groupId>
    <artifactId>txlcn-txmsg-netty</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>
```

相关配置

```
spring:
  application:
    ###服务的名称
    name: meitemayikt-order
  datasource:
```

```
    url: jdbc:mysql://localhost:3306/order?useUnicode=true&characterEncoding=UTF-8
    username: root
    password: root
    driver-class-name: com.mysql.jdbc.Driver
  cloud:
    nacos:
      discovery:
        ###nacos 注册地址
        server-addr: 127.0.0.1:8848
    refresh:
      refreshable: none



tx-lcn:
  client:
    manager-address: 127.0.0.1:8070
  logger:
    enabled: true
```

用法

参与方与发起方都要加上该注解
@LcnTransaction
@Transactional

源码核心入口

重新 feign 客户端拦截器 RequestInterceptor
Aop 的重学的入口 TransactionAspect
实现该接口可以在请求之前处理参数 SpringTracingApplier
http://127.0.0.1:8090/insertOrder?age=1

1. Lcn 如何判断自己是发起方还是参与方？
根据当前的线程 threadlocal 中获取事务分组 id，如果能够成功获取到则是为参与方，没有能够获取到就是为发起方。
2. A 调用 B，B 调用 C 到底会生产几次事务 id？

A 调用 B 调用 C 调用 D 只有全局的分组的 id 都是有一个局部的事务 id

3.参与方如何从请求头中获取事务 id?如何加入事务组中？
4.LCN 如何实现数据源代理实现假关闭？

学习 LCN 源码分析的话 入口 @LcnTransaction 必须有 AOP 才能够对我们注解生效。

TransactionAspectAop 的入口类。

深入了解 seata 解决分布式事务

Seata 简单介绍

https://github.com/seata/seata
https://seata.io/zh-cn/index.html

https://github.com/seata/seata/releases/

Seata 的实现原理

Seata 有 3 个基本组成部分：

事务协调器（TC）：维护全局事务和分支事务的状态，驱动全局提交或回滚。
事务管理器 TM：定义全局事务的范围：开始全局事务，提交或回滚全局事务。
资源管理器（RM）：管理分支事务正在处理的资源，与 TC 进行对话以注册分支事务并报告分支事务的状态，并驱动分支事务的提交或回滚。

笔记

分布式事务产生的背景

假设我们现在做案例 顺丰上门取件 点外卖 设计几张表

先下单--- 订单表
派单—派单表 orderid 对应 userId
思考点：
传统的项目情况下

```
String orderId = System.currentTimeMillis() + "";
OrderEntity newOrder = createOrder(orderId);
// 1.向订单数据库表插入数据
```

```
int result = orderMapper.addOrder(newOrder);
if (result < 0) {
    return "插入订单失败";
}
// 2.调用派单服务，实现对该笔订单派单
String resultDistribute = distributeservice.distributeOrder(orderId);


distributeservice.distributeOrder(orderId);
distributeMapper. distributeAddOrder(ordserId);
```

分布式事务产生的背景

1. 如果是在传统项目中，使用同一个数据源，在数据用同用一个事务管理器的情况下，不存在分别事务事务问题，因为有事务的传播行为帮助我们实现。每个数据源都自己独立的事务事务管理，每个数据源中的事务管理都互不影响。
2. 如果是在单体项目中， 存在多个不同的数据源，每个事务源都有自己独立的事务管理器，每个事务管理器互不影响，也会存在分布式事务的问题。Jta+atominc 将每个独立的事务管理器统一交给我们的 atominc 全局事务管理。
3. 在分布式系统中采用 rpc 远程通讯也会存在分布式事务问题

分布式rpc 通讯中为什么会存在分布式事务？
消费者(调用方)调用完接口成功之后，调用方突然抛出异常

调用者(订单服务) 生产者(派单服务)

1.生产者调用接口如果失败的情况下，一定要抛出异常或者是手动的回滚，不然会产生脏读数据。

Rpc 通讯中产生的分布式事务的问题原因

1.调用方（订单服务）调用完 rpc 接口之后，突然程序抛出异常，调用方的事务回滚了，但是被调用方接口没有回滚。

订单服务回滚了，派单成功，在每个 jvm 中都有自己的本地事务，每个事务都互不影响。

2.被调用方（派单服务）的接口失败的话，调用方可以根据返回的结果，手动回滚调用方本地事务

解决分布式事务的最大核心是什么？

1. 最终一致性 在分布式系统中， 因为 rpc 通讯是需要时间的，短暂的数据一致这是允许的，但是最终数据一定要保持一致性；
2. 全局协调者

有很多的机构的老师反应，有学员报名钱已经付款了，但是在后台中查询该订单还是没有支付状态。

Base 理论和 CAP 理论

CAP 总结：三者无法兼顾，在分布式系统当中可以容忍网络之间出现的通讯故障；
要么是 CP 或者 AP
CP：当你网络出现故障之后，只能保证数据一致性，但是不能保证可用性； zk
AP：当你网络出现故障之后，不能保证数据一致性，但是能够保证可用性 eureka

在分布式系统中，可能存在强一致性的问题

回顾下传统事务
Acid
A 原子性
C 一致性
I 隔离性
D 持久性

强一致性 弱一致性

Begin----对该数据上行锁，其他的线程不能够对该行数据做操作的
---调用数据库层
Commit/rollback

注意：在分布式系统中无法保证强一致性，因为数据短暂不一致这是运行的，但是最终数据一定要保证一致性的问题。

分布式事务解决框架有那些？
1. 传统多数据源的情况下 采用 jta
2. 基于 MQ 保证数据一致性 最终一致性
3. 基于 rocketmq 解决分布式事务 核心采用自带事务消息
4. 基于 LCN 模式解决分布式事务 tcc/2pc/lcn 模式
5. 基于阿里巴巴 seata 解决分布事务 （背景非常强大）

核心思想 tcc、2pc、最终一致性

zk 集群中 所有的写的请求统一交给我们的领导实现，领导将数据写完后，在同步给每个从节点。

每个节点之间的数据同步需要保持数据一致性的问题

为什么所有的写请求都必须要交个领导实现？在同步给每个从节点？

Zk 将数据写入完毕之后，zk 是如何将数据同步给每从节点保持数据一致性呢

第一节点 准备节点 zk 领导会给每个从节点发一个通知，是否可以同步该数据
所有的从节点回复给 zk 领导同步数据。
第二节点 zk 领导收到 zk 从节点回复的可以同步数据的话，在直接将数据同步给每个从节点。

LCN 实现原理：

1. 发起方和参与方项目启动的时候必须和全局协调者一直保持长连接;
2. 发起方向我们的LCN的协调者申请一个事务分组id
3. 发起方在调用参与方的接口的时候，重写 feign 客户端 在请求头中传递该事务分组的id
4. 参与方获取到请求头中有传递对应的事务分组 id，当前业务执行完毕之后不会提交事务，采用 jdbc 假关闭。
5. 发起方如果产生了回滚或者是提交的话，都会将该结果告诉给协调者，协调者在将该结果群发给所有的参与者

LCN A调用B　B调用C　C调用D?　全局的id

访问秒杀接口的时候 对用户的频率实现限流 qps2　redis

基于我们 sentinel 对我们热词实现限流

热点参数限流：对我们接口热词实现限流

Seckill?userId=123456&orderId=644064779
Seckill?userId=123456&orderId=644064779
Seckill?userId=7888&orderId=644064779