1.Spring

1.1.说一下你对 Spring 的理解?

关于 Spring 的话，我们平时做项目一直都在用，不管是使用 ssh 还是使用 ssm，都可以整合。Spring 里面主要的就三点，也就是核心思想，IOC，DI,AOP。

其实 spring 这个框架也用到了 Java 里的反射机制.

DI 就是依赖注入，把我们需要的类啊，接口啥的注入到 spring 中去。

IOC 控制反转，像我们之前开发，如果想创建一个对象，就 new 一个，如果想这个对象中定义其他的变量或者对象，就在对象内部创建一个成员变量。但是现在的话，如果想用这个类的对象，咱们可以在 spring 的配置文件中配置一个 bean，指定对应的全路径名称。spring 通过配置文件用反射的方式，就可以直接帮我们获取到这个类的对象。还有 AOP，就是面向切面编程，它的原理的话，我看过它的底层代码，它实际上就是实现了 JDK 的动态代理，以前的话用这个做过事务的控制，现在的话我们都用注解来控制事务。

AOP 执行过程是一个纵向的过程，把每个方法当作一个点.基于这些点可以进行增强处理.形成了横向的切面,包含了原有方法和增强方法.不改变原有代码结构,还添加了额外的功能.

你了解的 AOP 的使用场景有哪些?**事务管理，日志打印,还有就是在老项目中也有可能用它来做权限管理.**

整体来说的话，Spring 在使用的时候非常方便，在配置文件中配置要依赖的对象，或者在配置文件中将对象及属性进行注入,当然现在基本都用注解的方式，更方便。

除了这些，我们之前的项目也用过 spring 的其他产品，像 spring boot（简化新 Spring 应用的初始搭建以及开发过程,用我的话来理解,就是 spring boot 其实不是什么新的框架,它默认配置了很多框架的使用方式，就像 maven 整合了所有的 jar 包，spring boot 整合了所有的框架），spring cloud 微服务框架。比 spring 更简单，快速，方便。

(然后就可以扯微服务，和 spring boot、cloud)。。。

1.2.Spring 常用的注解

我们开发的时候常用的注解也就@service 业务逻辑,@Transactionnal 事务的注解,有时候需要注入 DAO 的话还会用到@Repository 还有就是 springMVC 里的注解啦，比如说@Controller 对应表现层的注解在 spring 未来的版本中,@RestController 这个控制器返回的都是 json 格式,还有@RequestMapping 进入的 URL,@responsebody 也是定义该方法返回 JSON 格式,还有就是@RequestParam 获取参数,@RequestBody 获取前台的数据是 JSON 格式的数据,@PathVariable 从 URL 请求路径中获取数据,大概常用的也就这些.

1.3.简单说一下 SpringMVC 与 Spring 是如何整合的?

简单的说 springMVC 在 ssm 中整合 就是 在 web.xml 里边配置 springMVC 的核心控制器:DispatcherServlet; 它就是对指定后缀进行拦截;然后在 springMVC.xml 里边配置扫描器,可以扫描到带@controller 注解的这些类,现在用 springMVC 都是基与注解式开发， 像

@service,@Repository @Requestmapping,@responsebody 啦这些注解标签 等等 都是开发时用的,每个注解标签都有自己的作用;它还配置一个视图解析器,主要就是对处理之后的跳转进行统一配置 ,大致就是这些,如何使用 springMVC 获取表单里的数据? 通过形参和表单里的 name 值保持一致就能获取到

1.4.详细的说一下 springaop 的实现原理

它的底层是通过动态代理实现的面向切面编程，主要用在管理事物这一块，我们在配置文件里配置切入点，比如以 save/insert/update 等开头的方法我们开启事物，配置了一个 REQUIRED 开头的事物特性，还可以用在日志管理，还有权限方面也可以用 aop 来实现，但现在基本上没有人去这样控制权限，都用 shiro 框架来实现权限管理，springaop 大概就是这样，咱们公司是用什么来管理事物的，应该使用注解的方式来管理的吧。以下内容在于理解,能说出大概就行

spring AOP 事务的描述：（难点）
在 spring-common.xml 里通过里面先设定一个表达式，
设定对 service 里那些方法 如：对 add* ,delete*,update*等开头的方法
进行事务拦截。我们需要配置事务的传播（propagation="REQUIRED"）
特性,通常把增,删,改以外的操作需要配置成只读事务（read
only="true"）.只读事务可以提高性能。之后引入 tx:advice,在 tx:advice
引用 transactionManager（事务管理）,在事务管理里再引入
sessionFactory,sessionFactory 注入 dataSource，最后通过
引入 txAdvice。
Spring 实现 ioc 控制反转描述：
原来需要我们自己进行 bean 的创建以及注入，而现在交给 spring 容器去
完成 bean 的创建以及注入。所谓的“控制反转”就是 对象控制权的转
移，从程序代码本身转移到了外部容器。

1.5.Spring 中@Autowired 和@Resource 的区别?

@Autowired 默认的是按照类型进行注入， 如果没有类型会按照名称(红色字体)进行注入.
如果想直接按照名称注入需要加入@Qualifier("gatheringDao")

```
@Autowired
@Qualifier("gatheringDao")
private GatheringDao gatheringDao;
```

@Resource 默认的会按照名称注入,名称找不着会按照类型来找,如果这里写了名称,就直接按照名称找了不会按类型找@Resource(name = "aaa")

```
@Resource
private GatheringDao gatheringDao;
```

1.6.你给我说一下 SpringBoot 吧?

SpringBoot是我们最近的项目开始用的。我个人觉得SpringBoot比以前的SpringMVC更好用，

因为他的配置文件少了。原来SpringMVC的SSM整合的配置文件特别多，用了SpringBoot之后配置文件特别少了。就需要一个yml文件几乎全部搞定,我们用SpringBoot时结合的MyBatis去做得，SpringBoot基本上是一些YML文件，properties文件，MyBatis全程用的注解方式开发。SpringBoot和SpringMVC用法上大同小异，无非就是少了一些配置文件。启动SpringBoot服务器的时候是他自带的Tomca和Jetty服务器，可以通过main方法启动。启动的注解是@SpringBootApplication,我们也用过Spring的全家桶,SpringBoot+JPA+SpringCloud组件,然后把springcloud里的每一个组件说一下,下面有答案

SpringBoot的优点：

快速创建独立运行的 Spring 项目以及与主流框架集成 jpa mybatis
使用嵌入式的 Servlet 容器应用无需打成 WAR 包
Starters(场景启动器)自动依赖与版本控制
大量的自动配置，简化开发，也可修改默认值
无需配置大量的 XML，无代码生成，开箱即用

1.7.SpringCloud 的常用组件挨个介绍一下?

Eureka 组件(服务发现)音标(ju`ri:kə)： 相当于我们使用的 dubbox 的时候 zookeeper 注册中心.

Feign 组件：调用服务的时候用的组件

Hystrix(hist`riks)熔断器：我们在调用服务的时候,有可能涉及到服务的连锁调用,比如说 A 服务调用 B 服务,B 服务里还调用了 C 服务， 使用 A 服务的时候,B 服务和 C 服务都得正常运行才可以使用,B 调用 C 没有调通的时候， B 直接给 A 返回内容， 不至于像以前报错.

服务网关 zull：前后端进行调用的时候可以,可以走同一个 IP 地址,因为项目端口号太多,配置这个以后就可以直接走一个端口号,他自动会给你分配具体调用的哪一个端口

分布式配置 Spring Cloud Config:我们把多个项目的配置文件归置为一个,修改配置文件以后,不用再重新部署某一个项目啦.

消息总线:Spring Cloud Bus: 修改完配置文件以后不用重启项目.

1.8.SpringBoot 和 SpringMVC 与 springCloud 关系及运行原理?

1. springboot 是 springmvc 的升级版,其实就把 springmvc 里的配置文件,改为全注解的开发 2. SpringCloud 通过 Springboot 把其他的通信组件等等进行了封装,你如果使用 SpringCloud 的,那就必须得使用 SpringBoot,使用 SpringBoot 的话不一定非得使用 SpringCloud.

运行原理

1.Spring Boot 在启动时扫描项目所依赖的 JAR 包，寻找包含 spring.factories 文件的 JAR
2.根据 spring.factories 配置加载 AutoConfigure 类
3.根据@Conditional 注解的条件，进行自动配置并将 Bean 注入 Spring Context

1.9.springboot 如果要直接获取 form 数据的话要怎么做？

也是通过形参里添加参数就行啊,和 form 表单里的 name 值对应上就行了啊,在 form 表单的 action 中配置上请求改方法的路径不就行了么.

1.10.SpringCloud 的用什么版本

我们用的是2.0的版本,我那会学springcloud的时候也了解过,他们的版本是根据伦敦的地铁站进行命名的,我们用的是比较稳定版本就是2.0这个版本

1.11 单元测试的时候怎么启动 spring 容器的

这个可以通过读取配置文件的方式也行,用 applicationContext 读取 xml 配置文件,其实我更喜欢使用 Spring 配合 junit 用注解的方式去启动,下面代码不用背出来,如果问到能收出来就行

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"classpath*:spring-config-test.xml"})public
class TestProjectDao {

    @Autowired
    ProjectDao projectDao;
    @Test
    public void testCreateProjectCode(){
        long applyTime = System.currentTimeMillis();
        Timestamp ts = new Timestamp(applyTime);
        Map codeMap = projectDao.generateCode("5", "8",ts,"院内");
        String projectCode = (String)codeMap.get("_project_code");
```

1.12.Spring 是如何管理事务的?

我们用 spring 管理事务的时候用的是注解式的方式管理的,他底层实现也是 AOP 的方式来管理的事务.

1.13.Spring 里的声明式事务?

是 spring 里的那 7 中传播特性,用 AOP 配置切入点,开启或者关闭事务

1.14.事务的传播特性？

1.PROPAGATION_REQUIRED：支持当前事务，如果当前没有事务，就新建一个

事务。这是最常见的选择。
2.PROPAGATION_SUPPORTS：支持当前事务，如果当前没有事务，就以非事务方式执行。
3.PROPAGATION_MANDATORY：支持当前事务，如果当前没有事务，就抛出异常。
4.PROPAGATION_REQUIRES_NEW：新建事务，如果当前存在事务，把当前事务挂起。
5.PROPAGATION_NOT_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
6.PROPAGATION_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。
7.PROPAGATION_NESTED：支持当前事务，新增 Savepoint 点，与当前事务同步提交或回滚。

1.15.Spring 的三种注入方式?

我们项目中是配置扫描包,通过注解方式注入的,还有通过 seter 和构造方法的方式注入.

1.16.SpringMVC 的 controller 里可不可以写成员变量

可以写啊，注入的 service 不就是成员变量么，你是不想问 struts2 里的获取参数的方式啊？Struts2 早就不用了，他那种用成员变量获取参数的方式，在高并发下会造成线程不安全，springmvc 是使用的形参接收前台数据的，线程比较安全。

2.MyBatis 框架

2.1Mybatis 框架简介

Mybatis 框架也是一个持久层框架，我们目前做的这个项目就是用的这个框架，我觉他相对于以前的hibernate来说比较简单,把sql语句写在配置文件里,解除了代码和sql语句的耦合度,写一些复杂的查询比较灵活.

2.2.MyBatis 中# 和 $ 的区别?

这两个符号一般是在使用Mybatis编写底层SQL语句时使用，#就是一个占位符，具体的使用是#{id}，而$是一个原样输出的标识，是${value},我在项目里一直是使用#，因为这样可以防止Sql注入，安全性高。

2.3.Mybatis 和 Hibernate 的区别

Hibernate 一个是全封装，mybatis 是半封装，使用 hibernate 做单表查询操作的时候比较简单(因为 hibernate 是针对对象进行操作的)，但是多表查询起来就比较繁琐了，比如说 5 张表 10 张表做关联查询，就算是有 SQLquery 那后续的维护工作也比较麻烦，还有就是 Hibernate 在 Sql 优化上执行效率上会远低于 MyBatis(因为 hibernate 会把表中所有的字段查询出来，比较消耗性能)我们以前在做传统项目方面用过 hibernate，但是现在基本上都在用 mybatis.

2.4.Mybatis 缓存？

mybatis 一级缓存是 SqlSession 级别的缓存，默认支持一级缓存，不需要在配置文件去配置。
mybaits 的二级缓存是 mapper 范围级别，除了在 SqlMapConfig.xml 设置二级缓存的总开关<settingname='cacheEnabled'value='true'/>，还要在具体的 mapper.xml 中开启二级缓存：<mappernamespace='cn.hpu.mybatis.mapper.UserMapper'>

2.5.Mybatis 里多对多如何处理？举个多对多的例子?

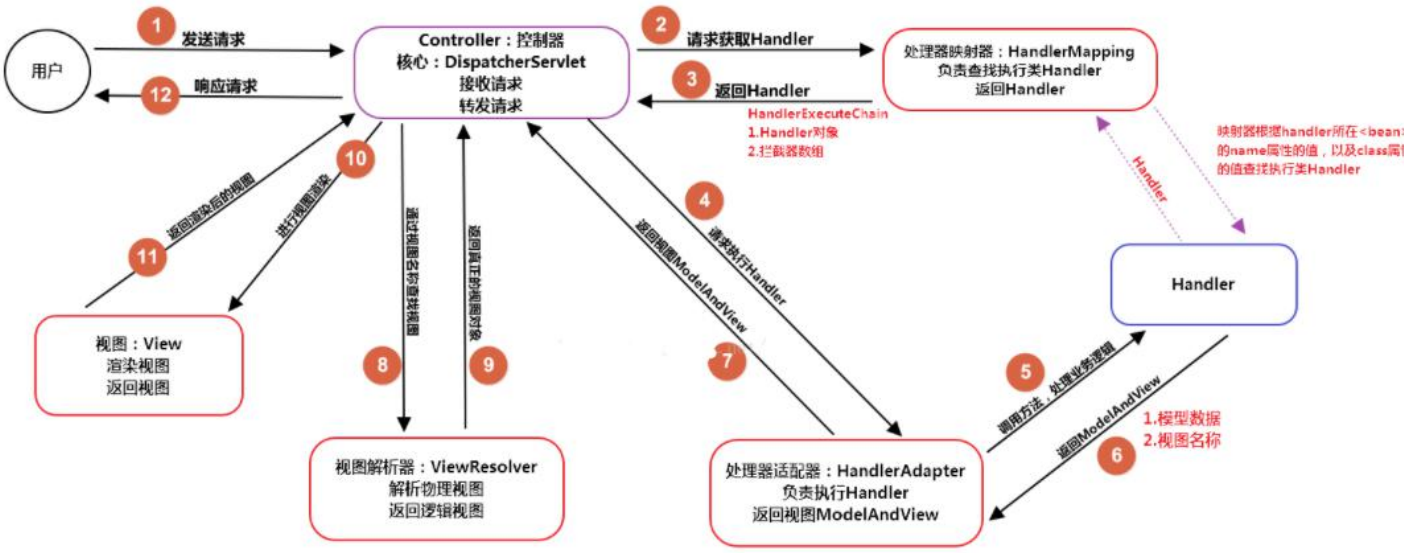
比如说学生表和课程表，一个学生可以选择多门课程，一门课程都能被多个学生选择，这样两张表的关系就是多对多的关系，怎么处理多对多的情况？遇到这种情况我们得创建一张中间的桥表，关联后就是 课程表对桥表就是一对多，学生表对桥表也是一对多，就可以了.

2.6.MyBatis 执行流程

3.SpringMVC 框架

3.1 SpringMVC 的工作原理

1. 用户向服务器发送请求，请求被 springMVC 前端控制器 DispatchServlet 捕获；
2. DispatcherServlet 对请求 URL 进行解析，得到请求资源标识符（URL），然后根据该 URL 调用 HandlerMapping 将请求映射到处理器 HandlerExecutionChain；
3. DispatchServlet 根据获得 Handler 选择一个合适的 HandlerAdapter 适配器处理；
4. Handler 对数据处理完成以后将返回一个 ModelAndView（）对象给 DisPatchServlet;
5. Handler 返回的 ModelAndView() 只是一个逻辑视图并不是一个正式的视图， DispatcherSevlet 通过 ViewResolver 试图解析器将逻辑视图转化为真正的视图 View;
6. DispatcherServlet 通过 model 解析出 ModelAndView()中的参数进行解析最终展现出完整的 view 并返回给客户端;



3.2 SpringMVC 常用注解都有哪些？

@RequestMapping 用于请求 url 映射。

@RequestBody 注解实现接收 http 请求的 json 数据，将 json 数据转换为 java 对象。

@ResponseBody 注解实现将 controller 方法返回对象转化为 json 响应给客户

3.3 如何开启注解处理器和适配器？

我们在项目中一般会在 springmvc.xml 中通过开启 <mvc:annotation-driven> 来实现注解处理器和适配器的开启。

3.4 如何解决 get 和 post 乱码问题？

解决 post 请求乱码:我们可以在 web.xml 里边配置一个 CharacterEncodingFilter 过滤器。

设置为 utf-8.

解决 get 请求的乱码:有两种方法。对于 get 请求中文参数出现乱码解决方法有两个:

1. 修改 tomcat 配置文件添加编码与工程编码一致。

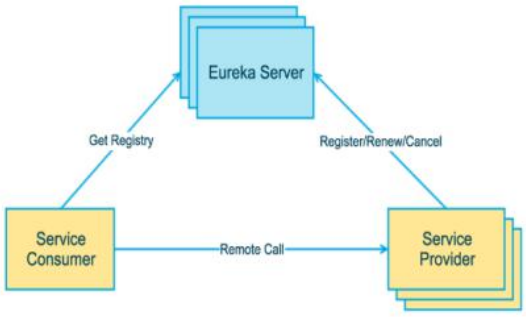
2.另外一种方法对参数进行重新编码

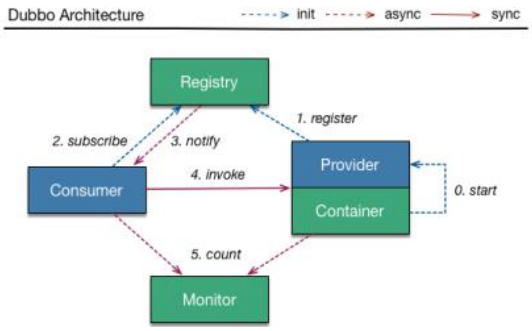
String userName=NewString(Request.getParameter(“userName”).getBytes(“ISO8859-1”),“utf-8”);

四、SpringCloud

4.1Eureka 介绍

Eureka 是 Netflix 出品的用于实现服务注册和发现的工具。 Spring Cloud 集成了 Eureka，并提供了开箱即用
的支持。其中， Eureka 又可细分为 Eureka Server 和 Eureka Client。
Spring Cloud 封装了 Netflix 公司开发的 Eureka 模块来实现服务注册和发现(请对比 Zookeeper)。
Eureka 采用了 C-S 的设计架构。Eureka Server 作为服务注册功能的服务器，它是服务注册中心。
而系统中的其他微服务，使用 Eureka 的客户端连接到 Eureka Server 并维持心跳连接。这样系统的维护人员就
可以通过 Eureka Server 来监控系统中各个微服务是否正常运行。SpringCloud 的一些其他模块（比如 Zuul）就可
以通过 Eureka Server 来发现系统中的其他微服务，并执行相关的逻辑。

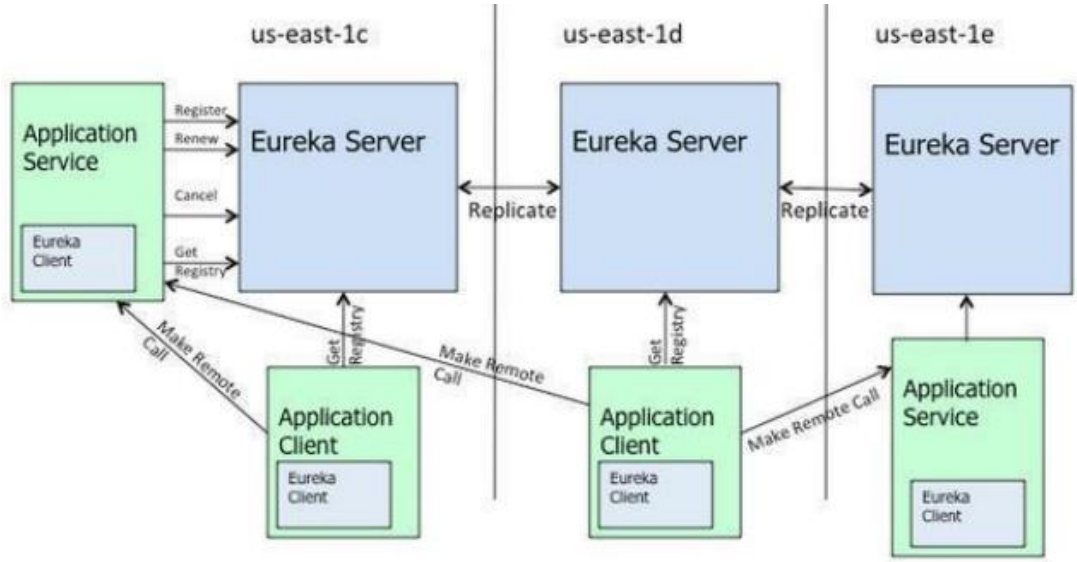




Eureka 包含两个组件：Eureka Server 和 Eureka Client。

- Eureka Server 提供服务注册服务,各个节点启动后会在 EurekaServer 中进行注册，这样 EurekaServer 中的服务注
册表中将会存储所有可用服务节点的信息，服务节点的信息可以在界面中直观的看到
- EurekaClient 是一个 Java 客户端，用于简化 Eureka Server 的交互，客户端同时也具备一个内置的、使用轮询
(round-robin)负载算法的负载均衡器。在应用启动后，将会向 Eureka Server 发送心跳(默认周期为 30 秒)。如果
Eureka Server 在多个心跳周期内没有接收到某个节点的心跳，EurekaServer 将会从服务注册表中把这个服务节
点移除（默认 90 秒）

4.2 集群基本原理



上图是来自 eureka 的官方架构图，这是基于集群配置的 eureka；
- 处于不同节点的 eureka 通过 Replicate 进行数据同步
- Application Service 为服务提供者
- Application Client 为服务消费者
- Make Remote Call 完成一次服务调用

服务启动后向 Eureka 注册，Eureka Server 会将注册信息向其他 Eureka Server 进行同步，当服务消费者要调用
服务提供者，则向服务注册中心获取服务提供者地址，然后会将服务提供者地址缓存在本地，下次再调用时，则直
接从本地缓存中取，完成一次调用。
当服务注册中心 Eureka Server 检测到服务提供者因为宕机、网络原因不可用时，则在服务注册中心将服务置为
DOWN 状态，并把当前服务提供者状态向订阅者发布，订阅过的服务消费者更新本地缓存。
服务提供者在启动后，周期性（默认 30 秒）向 Eureka Server 发送心跳，以证明当前服务是可用状态。Eureka Server
在一定的时间（默认 90 秒）未收到客户端的心跳，则认为服务宕机，注销该实例。

4.3 Eureka 的自我保护机制

在默认配置中，Eureka Server 在默认 90s 没有得到客户端的心跳，则注销该实例，但是往往因为微服务跨进程调用，网络通信往往会面临着各种问题，比如微服务状态正常，但是因为网络分区故障时，Eureka Server 注销服务实例则会让大部分微服务不可用，这很危险，因为服务明明没有问题。

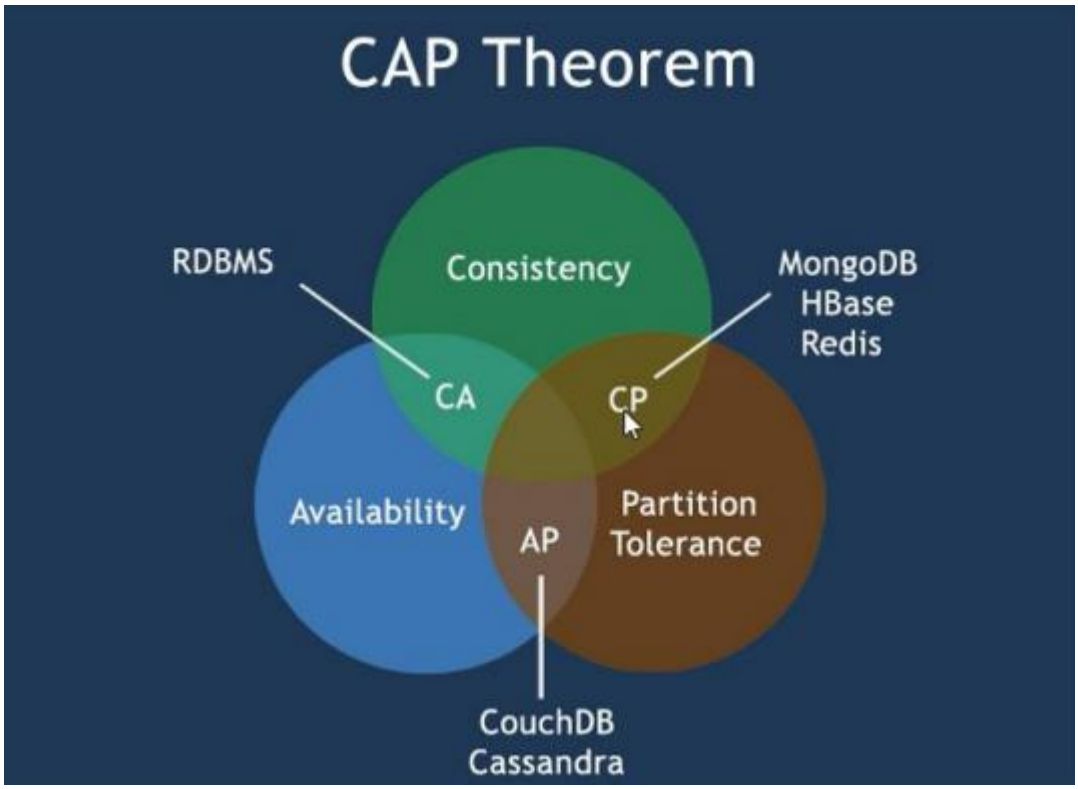
为了解决这个问题，Eureka 有自我保护机制，通过在 Eureka Server 配置如下参数，可启动保护机制 `eureka.server.enable-self-preservation=true`

它的原理是，当 Eureka Server 节点在短时间内丢失过多的客户端时（可能发送了网络故障），那么这个节点将进入自我保护模式，不再注销任何微服务，当网络故障回复后，该

节点会自动退出自我保护模式。自我保护模式的架构哲学是宁可放过一个，决不可错杀一千，好死不如赖活着

4.4 Eureka 和 Zookeeper 比较

CAP 理论的核心：一个分布式系统不可能同时很好地满足一致性、可用性和分区容错性这三个需求。因此，根据
CAP 原理将 NOSQL 数据分成了 CA 原则、CP 原则、AP 原则三大类：
CA：单点集群，满足一致性，可用性的系统，通常在可扩展性上不强
CP：满足一致性，分区容错性的系统，通常性能不够高
AP：满足可用用，分区容错性的系统，通常可能对一致性的要求低一些



CAP 原则又称 CAP 定理，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），三者不可兼得。由于分区容错性在是分布式系统中必须要保证的，因此我们只能在 A 和 C
之间进行权衡。在此 Zookeeper 保证的是 CP, 而 Eureka 则是 AP。

4.5 Zookeeper 保证 CP

当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接 down 掉不可用。也就是说，服务注册功能对可用性的要求要高于一致性。但是 zk 会出现这样一种情况，当 master 节点因为网络故障与其他节点失去联系时，剩余节点会重新进行 leader 选举。问题在于，选举 leader 的时间太长，

30～120s, 且选举期间整个 zk 集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络 问题使得 zk 集群失去 master 节点是较大概率会发生的事，虽然服务能够最终恢复，但是漫长的选举时间导致的注 册长期不可用是不能容忍的。

4.6 Eureka 保证 AP

Eureka 看明白了这一点，因此在设计时就优先保证可用性。Eureka 各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而 Eureka 的客户端在向某个 Eureka 注册或时如果发现连接失败，则会自动切换至其它节点，只要有一台 Eureka 还在，就能保证注册服务可用(保证可用性)，只不过查到的信息可能不是最新的(不保证强一致性)。除此之外，Eureka 还有一种自我保护机制，如果在 15 分钟内超过 85%的节点都没有正常的心跳，那么 Eureka 就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况：1. Eureka 不再从注册列表中移除因为长时间没收到心跳而应该过期的服务 2. Eureka 仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(即保证当前节点依然可用)3. 当网络稳定时，当前实例新的注册信息会被同步到其它节点中因此， Eureka 可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像 zookeeper 那样使整个注册服务瘫痪。

4.7 总结

Zookeeper 保证的是 CP, 而 Eureka 则是 AP
Eureka 作为单纯的服务注册中心来说要比 zookeeper 更加“专业”，因为注册服务更重要的是可用性，我们可以 接受短期内达不到一致性的状况。不过 Eureka 目前 1.X 版本的实现是基于 servlet 的 Javaweb 应用，它的极限性能肯定会受到影响。期待正在开发之中的 2.X 版本能够从 servlet 中独立出来成为单独可部署执行的服务。

4.8 Ribbon 和 Feign 的区别

Ribbon 和 Feign 都是用于调用其他服务的，不过方式不同。

- Ribbon 添加 maven 依赖 spring-starter-ribbon 使用@RibbonClient(value="服务名称") 使用 RestTemplate 调用远程服务对应的方法
- feign 添加 maven 依赖 spring-starter-feign 服务提供方提供对外接口 调用方使用在接口上使用@FeignClient("指定服务名")
- 服务的指定位置不同，Ribbon 是在@RibbonClient 注解上声明，Feign 则是在定义抽象方法的接口中使用@FeignClient 声明。

4.9 springcloud 断路器的作用

当一个服务调用另一个服务由于网络原因或者自身原因出现问题时 调用者就会等待被调用者的响应 当更多的服务请求到这些资源时
导致更多的请求等待 这样就会发生连锁效应（雪崩效应） 断路器就是解决这一问题

断路器有完全打开状态
一定时间内 达到一定的次数无法调用 并且多次检测没有恢复的迹象 断路器完全打开，那么下次
请求就不会请求到该服务

半开

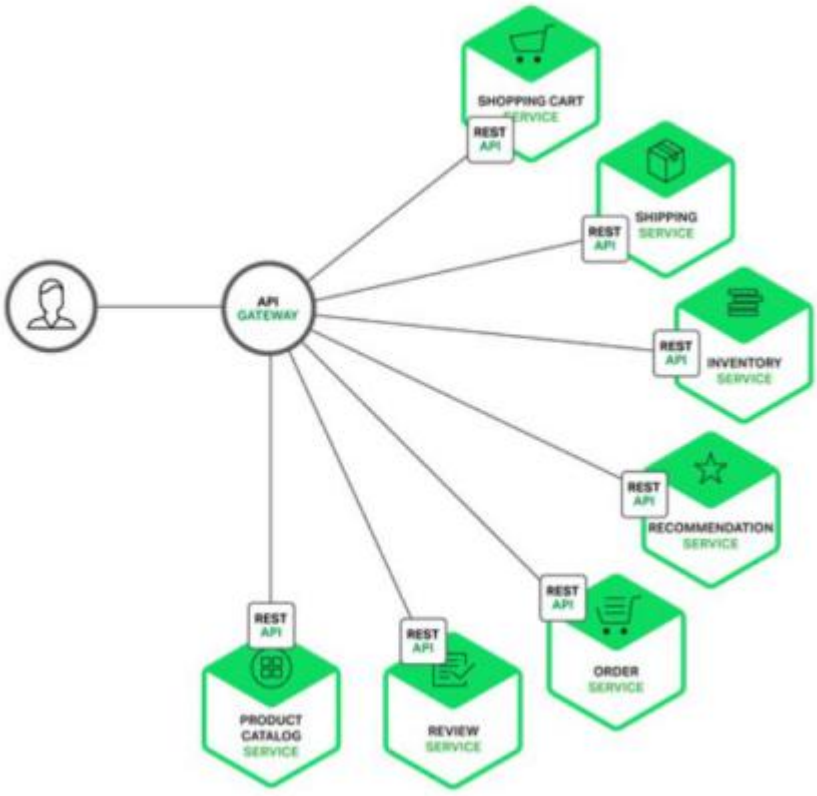
短时间内 有恢复迹象 断路器会将部分请求发给该服务 当能正常调用时 断路器关闭

关闭

当服务一直处于正常状态 能正常调用 断路器关闭

5.0 API 网关

API Gateway 是一个服务器，也可以说是进入系统的唯一节点。这跟面向对象设计模式中的 Facade 模式【外观模式】很像。API Gateway 封装内部系统的架构，并且提供 API 给各个客户端。它还可能有 其他功能，如授权、监控、负载均衡、缓存、请求分片和管理、静态响应处理等。下图展示了一个适应当前架构的 API Gateway。



API Gateway 负责请求转发、合成和协议转换。所有来自客户端的请求都要先经过 API Gateway，然后路由这些
请求到对应的微服务。
API Gateway 将经常通过调用多个微服务来处理一个请求以及聚合多个服务的结果。它可以在 web 协议与内部
使用的非 Web 友好型协议间进行转换，如 HTTP 协议、WebSocket 协议。

- 请求转发 ：服务转发主要是对客户端的请求安装微服务的负载转发到不同的服务上

- 响应合并：把业务上需要调用多个服务接口才能完成的工作合并成一次调用对外统一提供服务。
- 协议转换：重点是支持 SOAP，JMS，Rest 间的协议转换。
- 数据转换：重点是支持 XML 和 Json 之间的报文格式转换能力（可选）
- 安全认证：

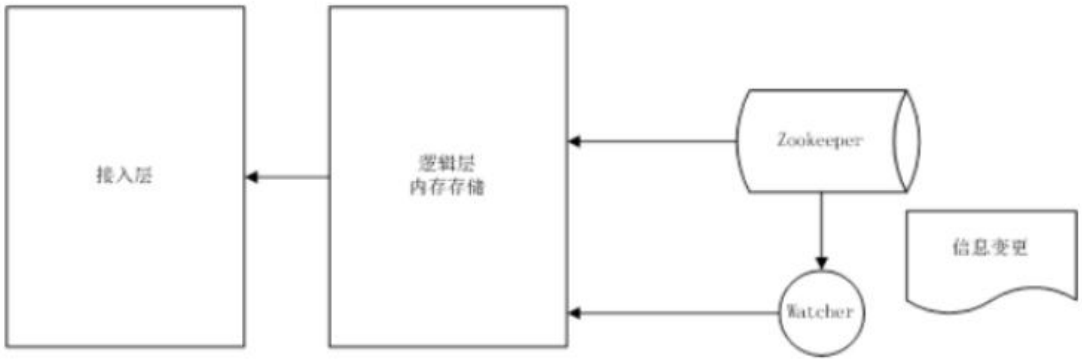
1. 基于 Token 的客户端访问控制和安全策略
2. 传输数据和报文加密，到服务端解密，需要在客户端有独立的 SDK 代理包
3. 基于 Https 的传输加密，客户端和服务端数字证书支持
4. 基于 OAuth2.0 的服务安全认证(授权码，客户端，密码模式等）

5.1 配置中心

配置中心一般用作系统的参数配置，它需要满足如下几个要求：高效获取、实时感知、分布式访问。

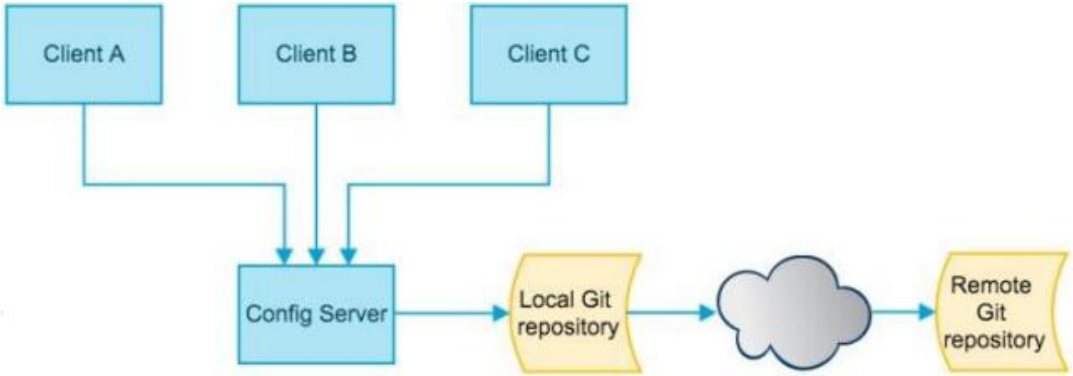
5.1.1 zookeeper 配置中心

实现的架构图如下所示，采取数据加载到内存方式解决高效获取的问题，借助 zookeeper 的节点 监听机制来实现实时感知。



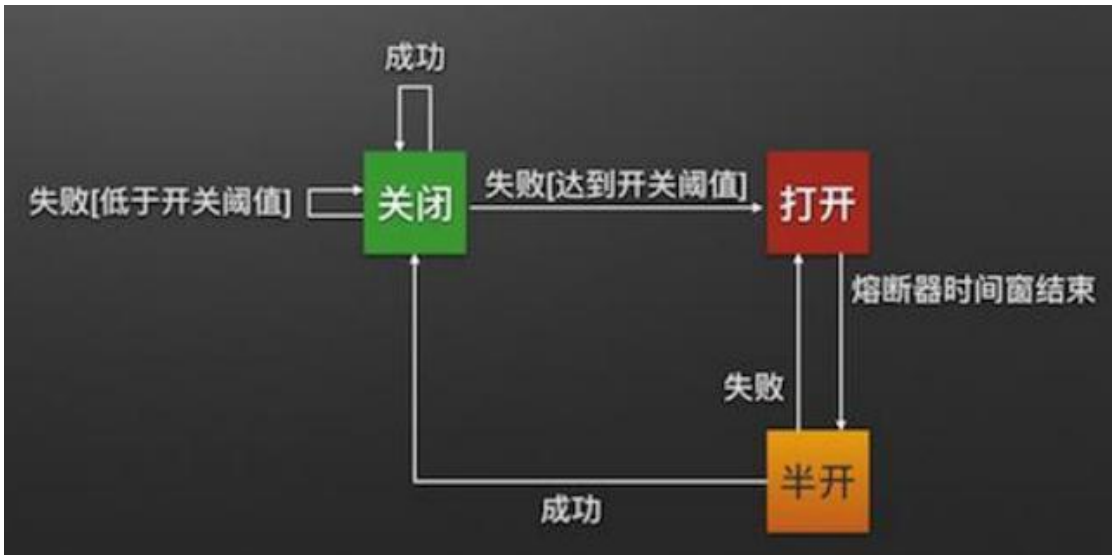
5.1.2 SpringCloud Config

SpringCloud Config 为微服务架构中的微服务提供集中化的外部配置支持，配置服务器为各个不同微服务应用的所有环境提供了一个中心化的外部配置。



5.2 服务熔断（Hystrix）

在微服务架构中通常会有多个服务层调用，基础服务的故障可能会导致级联故障，进而造成整个 系统不可用的
情况，这种现象被称为服务雪崩效应。服务雪崩效应是一种因“服务提供者”的不 可用导致“服务消费者”的不可用,并将不可用逐渐放大的过程。
熔断器的原理很简单，如同电力过载保护器。它可以实现快速失败，如果它在一段时间内侦测到 许多类似的错误，会强迫其以后的多个调用快速失败，不再访问远程服务器，从而防止应用程序 不断地尝试执行可能会失败的操作，使得应用程序继续执行而不用等待修正错误，或者浪费 CPU 时间去等到长时间的超时产生。熔断器 也可以使应用程序能够诊断错误是否已经修正，如果已经修正，应用程序会再次尝试调用操作。



5.2.1Hystrix 断路器机制

断路器很好理解，当 Hystrix Command 请求后端服务失败数量超过一定比例(默认 50%), 断路器会 切换到开 路状态(Open). 这时所有请求会直接失败而不会发送到后端服务. 断路器保持在开路状态 一段时间后(默认 5 秒), 自动切换到半开路状态(HALF-OPEN). 这时会判断下一次请求的返回情况, 如果请求成功, 断路器切回闭路状态(CLOSED), 否则重新切换到开路状态(OPEN). Hystrix 的断路器 就像我们家庭电路中的保险丝, 一旦后端服务不可用, 断路器会直接切断请求链, 避免发送大量无效 请求影响系统吞吐量, 并且断路器有自我检测并恢复的能力。