# Spring部分

# Spring Boot部分

1、 springboot自动配置的原理？

使用SpringBoot需要在spring boot的启动类上添加@SpringBootApplication注解，这个注解是一个组合注解，它的核心功能是由@EnableAutoConfiguration注解提供的。

EnableAutoConfigurationImportSelector使用SpringFactoriesLoader.loadFactoryNames来扫描具有META-INF/spring.factories

的jar包，比如spring-boot-autoconfigure-1.3.0.jar中就有一个spring.facories文件，此文件中声明了有哪些自动配置类，比如

org.springframework.boot.autoconfigure.aop.AopAutoConfiguration，并对这些自动配置类进行注册。

这些自动配置类，一般都有一些条件注解，比如@ConditionalOnBean，@ConditionalOnWebApplication(org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration中就含有此注解)，正是由这些条件注解来决定这些自动配置要不要起作用。

# Spring Cloud部分

1. Eureka client可以从服务列表中获取可用的服务提供者信息，但是要使用服务名称进行服务的调用还需要使用到ribbon或者feign，直接使用普通的restTemplate是无法解析服务名称对应的具体服务提供者的。比如下面这种方式就不行：



1. Eureka服务注册的实现原理？

以Eureka Client为例，使用Eureka Client的相关功能之前要在SpringBoot的启动类上加上@EnableDiscoveryClient注解，这个注解会引入一些自动化配置类，其完成的核心功能就是创建了一个DiscoveryClient的Bean，正是通过这个Bean完成了Eureka Client的一系列功能(服务注册、服务续约、可用服务列表获取等功能)。

1. 什么是Eureka的自我保护机制？其作用是什么？

当EurekaServer短时间内丢失了过多的实例连接时(比如由于网络故障或频繁的启动关闭客户端)，那么这个EurekaServer节点就会进入自我保护模式，一旦进入到该模式，EurekaServer就会保护服务注册表中的信息，不会删除任何服务注册表中的数据。

Eureka进入自我保护模式通常是因为多个服务实例与此Eureka之间的网络连接都出现了问题，此时问题可能出在网络连接，而不是在于注册到Eureka上的多个微服务，这是一种应对网络异常的安全保护措施。**\*\*它的架构哲学是宁可同时保留所有微服务（健康的微服务和不健康的微服务都会保留），也不盲目注销任何健康的微服务。\*\***使用自我保护模式，可以让Eureka集群更加的健壮、稳定。

1. Ribbon实现原理？

因为要使用Ribbon提供的功能，必须使用@LoadBalanced注解的RestTemplate，

这个注解完成的核心功能就是为我们的RestTemplate Bean添加LoadBalancerInterceptor拦截器。

当一个被@LoadBalanced注解修饰的RestTemplate对象向外发起Http请求时，就会被LoadBalancerInterceptor.intercept方法拦截，

之后会调用LoadBalancerClient的实现类RibbonLoadBalancerClient代理发起请求。

更具体的来说，DynamicServerListLoadBalancer使用EurekaClient动态更新Server List(默认每10s向EurekaClient发送一次ping，进而检查是否需要更新服务的注册列表信息)，

RibbonLoadBalancerClient类通过DynamicServerListLoadBalancer获取ServerList，

之后使用ServerListFilter选出一部分符合条件的Server List,然后使用IRule选择出最终要访问的服务提供者地址。

我们可以通过配置users.ribbon.NFLoadBalancerRuleClassName= com.netflix.loadbalancer.WeightedResponseTimeRule来自定义负载均衡规则。

1. 什么是微服务架构中的雪崩效应？

在微服务架构中往往存在许多个服务单元，若一个服务单元出现故障，其它服务调用这个故障服务时就可能被阻塞，这样就很容易因为依赖关系引发故障的蔓延，最终导致整个系统的瘫痪，也就是常说的雪崩效应，为了解决这个问题，产生了断路器等一系列的服务保护机制。

1. 什么是服务降级？什么是服务熔断？其实现原理是怎样的？

服务降级是在Hystrix命令执行失败时的一种备用方案，在HystrixCommand中可以通过重载getFallback（）方法来实现服务降级逻辑处理，Hystrix会在方法执行中出现错误，超时，线程池拒绝，信号量获取不到、断路器熔断等情况时，执行降级逻辑。

一个服务在一段时间内被降级的次数太多，超过了我们设置的阈值，就会被熔断，之后到来的请求直接调用fallback函数。一段时间之后(有一个窗口的概念)这个服务又会处于半打开状态，尝试让一部分请求调用真正的服务，如果发现服务已经恢复的话让服务切换到正常状态，否则继续处于熔断状态。

其核心实现原理是使用线程池隔离与信号隔离。

线程池隔离将客户端调用线程与实际发起请求的线程分离。默认每个类下的所有方法调用共享一个线程池，默认线程池corePoolSize与maximumSize都为10.线程池底层默认实现方式是ThreadPoolExecutor。

信号量隔离默认类中的每个方法共享一个信号量，默认限制数量为10，信号隔离中的信号计数Hystrix底层是通过AtomicInteger实现的。信号量隔离不支持客户端调用线程请求超时自动调用fallBack函数返回的机制。

其中线程池隔离猜测是使用SpringAOP将被@HystrixCommand的类生成了代理类，当用户调用这个代理类中的方法时，判断是否有必要进行线程隔离或者信号量隔离。

1. Spring Cloud Feign的作用及其工作原理？

Feign对Ribbon与Hystrix进行了进一步的封装，使得服务之间的依赖调用非常优雅(伪RPC)。      

实际的请求调用是使用Ribbon实现的。

默认情况下，Feign会将所有Feign客户端的方法都封装到Hystrix命令中进行服务保护，当然也可以通过feign.hystrix.enabled=false来关闭Hystrix功能。

Feign的实现原理也是动态代理，通过扫描@FeignClient注解标注的接口，通过spring的IOC方式注入到我们的应用中。之后生成接口对应的代理RequestTemplate模板对象，通过代理类实现对依赖服务的伪RPC调用以及上述Feign的核心功能。

1. Zuul的作用及工作原理？

作为微服务系统的网关组件，用于构建边界服务，可以在基础服务进行聚合提供用户所需要的功能，在此基础之上还提供了动态路由、过滤、监控、弹性伸缩和安全等能力。     
  
Zuul中自身就包含了对于hystrix和ribbon模块的依赖，所以Zuul天生就有线程隔离和断路器的自我保护功能，以及对服务调用的客户端负载均衡功能。

Zuul的工作原理：

引入了ZuulServlet，是HttpServlet的子类，其内部实现类似于DispatcherServlet，当用户的请求url匹配我们在application.properties中要交给zuul处理的url时，就将用户的请求交由ZuulServlet处理，否则还是交由DispatcherServlet处理。

ZuulServlet中引入了一系列核心ZuulFilter对用户的请求进行处理，正是通过这些核心过滤器实现了Zuul的请求路由功能。我们还可以通过自定义过滤器实现鉴权等功能。