spring框架中的核心：IOC 可以帮助我们管理对象的依赖关系，极大减少对象的耦合性，而 AOP 的切面编程功能可以更方面的使用动态代理来实现各种动态方法功能（如事务、缓存、日志等）。

Spring Boot应用本质上就是一个基于Spring框架的应用，它是Spring对“约定优先于配置”理念的最佳实践产物，它能够帮助开发者更快速高效地构建基于Spring生态圈的应用。

Spring Boot是 Spring 开源组织下的一个子项目，也是 Spring 组件一站式解决方案，主要是为了简化使用 Spring 框架的难度，简省繁重的配置。

Spring Boot提供了各种组件的启动器（starters），开发者只要能配置好对应组件参数，Spring Boot 就会自动配置，让开发者能快速搭建依赖于 Spring 组件的 Java 项目。

什么是springboot

        用来简化spring应用的初始搭建以及开发过程 使用特定的方式来进行配置（properties或yml文件）

                创建独立的spring引用程序 main方法运行

                嵌入的Tomcat 无需部署war文件

                简化maven配置

                自动配置spring添加对应功能starter自动化配置

springboot常用的starter有哪些

        spring-boot-starter-web 嵌入tomcat和web开发需要servlet与jsp支持

        spring-boot-starter-data-jpa 数据库支持

        spring-boot-starter-data-redis redis数据库支持

        spring-boot-starter-data-solr solr支持

        mybatis-spring-boot-starter 第三方的mybatis集成starter

springboot自动配置的原理

        在spring程序main方法中 添加@SpringBootApplication和@EnableAutoConfiguration

        会自动去maven中读取每个starter中的spring.factories文件  该文件里配置了所有需要被创建spring容器中的bean

springboot读取配置文件的方式

        springboot默认读取配置文件为application.properties或者是application.yml

springboot集成mybatis的过程

        添加mybatis的starter maven依赖

                <dependency>

                        <groupId>org.mybatis.spring.boot</groupId>

                        <artifactId>mybatis-spring-boot-starter</artifactId>

                        <version>1.2.0</version>

                </dependency>

        在mybatis的接口中 添加@Mapper注解

        在application.yml配置数据源信息

springboot如何添加【修改代码】自动重启功能

        添加开发者工具集=====spring-boot-devtools

https://blog.csdn.net/u013605060/article/details/80255192

问：你觉得 Spring Boot 最大的优势是什么呢？

答：Spring Boot 的最大的优势是“约定优于配置“。“约定优于配置“是一种软件设计范式，开发人员按照约定的方式来进行编程，可以减少软件开发人员需做决定的数量，获得简单的好处，而又不失灵活性。

问：Spring Boot 中 “约定优于配置“的具体产品体现在哪里。

答：Spring Boot Starter、Spring Boot Jpa 都是“约定优于配置“的一种体现。都是通过“约定优于配置“的设计思路来设计的，Spring Boot Starter 在启动的过程中会根据约定的信息对资源进行初始化；Spring Boot Jpa 通过约定的方式来自动生成 Sql ，避免大量无效代码编写。具体详细可以参考：Spring Boot 为什么这么火？

问：Spring Boot Starter 的工作原理是什么？

答：Spring Boot  在启动的时候会干这几件事情：

Spring Boot 在启动时会去依赖的 Starter 包中寻找 resources/META-INF/spring.factories 文件，然后根据文件中配置的 Jar 包去扫描项目所依赖的 Jar 包。

根据 spring.factories 配置加载 AutoConfigure 类

根据 @Conditional 注解的条件，进行自动配置并将 Bean 注入 Spring Context

总结一下，其实就是 Spring Boot 在启动的时候，按照约定去读取 Spring Boot Starter 的配置信息，再根据配置信息对资源进行初始化，并注入到 Spring 容器中。这样 Spring Boot 启动完毕后，就已经准备好了一切资源，使用过程中直接注入对应 Bean 资源即可。

**微服务同时调用多个接口，怎么支持事务的啊？**

支持分布式事务，可以使用Spring Boot集成 Aatomikos来解决，但是我一般不建议这样使用，因为使用分布式事务会增加请求的响应时间，影响系统的TPS。一般在实际工作中，会利用消息的补偿机制来处理分布式的事务。

Spring Cache 三种常用的缓存注解和意义？

@Cacheable ，用来声明方法是可缓存，将结果存储到缓存中以便后续使用相同参数调用时不需执行实际的方法，直接从缓存中取值。

@CachePut，使用 @CachePut 标注的方法在执行前，不会去检查缓存中是否存在之前执行过的结果，而是每次都会执行该方法，并将执行结果以键值对的形式存入指定的缓存中。

@CacheEvict，是用来标注在需要清除缓存元素的方法或类上的，当标记在一个类上时表示其中所有的方法的执行都会触发缓存的清除操作。

Spring Cloud 是一系列框架的有序集合。它利用 Spring Boot 的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用 Spring Boot 的开发风格做到一键启动和部署。

Spring Cloud 是为了解决微服务架构中服务治理而提供的一系列功能的开发框架，并且 Spring Cloud 是完全基于 Spring Boot 而开发，Spring Cloud 利用 Spring Boot 特性整合了开源行业中优秀的组件，整体对外提供了一套在微服务架构中服务治理的解决方案。

用一组不太合理的包含关系来表达它们之间的关系。

Spring ioc/aop > Spring > Spring Boot > Spring Cloud

https://blog.csdn.net/yuzongtao/article/details/84295732

sql中的#和$的区别：

1、用来传入参数，sql在解析的时候会加上” “,当成字符串来解析 ，如这里 role\_id = “roleid”;

2、#{}能够很大程度上防止sql注入;

延伸：

1、用{roleId,jdbcType=INTEGER},那么sql在解析的时候值为roleId = roleId，执行时会报错;

2、${}方式无法防止sql注入;

3、$一般用入传入数据库对象，比如数据库表名;

4、能用#{}时尽量用#{};

注意：

mybaties排序时使用order by 动态参数时需要注意，使用${}而不用#{};

默认情况下，使用#{}格式的语法会导致MyBatis创建预处理语句属性并以它为背景设置安全的值（比如?）。这样做很安全，很迅速也是首选做法，有时你只是想直接在SQL语句中插入一个不改变的字符串。比如，像ORDER BY，你可以这样来使用：

ORDER BY ${columnName};  
这里MyBatis不会修改或转义字符串。

重要：接受从用户输出的内容并提供给语句中不变的字符串，这样做是不安全的。这会导致潜在的SQL注入攻击，因此你不应该允许用户输入这些字段，或者通常自行转义并检查。

## JPA和Hibernate有哪些区别

         简而言之：

                   1、JPA是一个规范或者接口

                   2、Hibernate是JPA的一个实现

         当我们使用JPA的时候，我们使用javax.persistence包中的注释和接口时，不需要使用hibernate的导入包。

Spring Boot不但能创建传统的 war 包应用，还能创建独立的不依赖于任何外部容器（如：tomcat）的独立应用，使用 java -jar 命令就能启动。同时，Spring Boot也提供了一个命令行工具来执行 Spring 的脚本。

特点：

1.独立运行：Spring Boot内嵌了各种 Servlet 容器，Tomcat、Jetty等，现在不再需要打成 war 包部署到容器中，Spring Boot 只要打成一个可执行的jar包就能独立运行，所有的依赖包都在一个 jar 包内。

2.简化maven配置:现在只要依赖 spring-boot-starter-web 启动器包，它包含所有 web 开发所有的依赖，就能拥有 Spring Web 的能力，极大简少了 maven 对依赖的配置。

使用 Maven 命令：mvn dependency:tree 也可以看到完整的依赖树

3.自动配置

无代码生成和xml配置

应用监控

缺点：

1、需要非常了解 Spring Boot 的核心技术原理，不然一旦遇到问题就很棘手。因为什么东西都集成了，自动配置化。这样，需要对配置信息非常熟悉，要找起问题来不是很容易。

2、从原始 Spring 项目很难平滑迁移至 Spring Boot 框架上来，因为有些历史老旧的 XML 配置无法通过 Java 来配置，还需要额外的 XML 文件就不是很完美。再比如，之前是独立的 Tomcat，什么参数都在线上配置好了，你改为内置的 Tomcat 就会遇到很多问题。

SpringApplication只是将一个典型的Spring应用的启动流程进行了扩展。

SpringApplication.run(.class, args)；

Springcloud解决那些问题：

配置管理、（注册中心eureka、zk）、服务发现、服务注册、断路器、路由策略、全局锁、分布式会话、客户端调用、接口网关（zuul）、服务管理系统

SpringBoot与Springcloud：

1>、SpringBoot简化了xml配置，快速整合框架

2>、Springcloud是一套微服务解决方案—RPC远程调用

3>、关系Springcloud依赖与SpringBoot（web组件用的SpringMVC），为什么Springcloud会依赖与SpringBoot？因为Springcloud写接口就是SpringMVC接口

4>、SpringBootproperties和yml中可以使用${random}设置一些随机值

服务的调用：

rest、feign（均使用httpclient技术），负载均衡ribbon

服务调用的原理：

服务首先注册到注册中心eureka中(注册一个名字通过名字调用)

负载均衡

ribbon，先去注册中心取到对应的服务，然后交给我ribbon

配置详解：

1>、eureka.client.register-with-eureka：是否向注册中心注册自己，注册为true反之为false

2>、eureka.client.fetch-registry： 是否需要去检索服务，检索为true反之为false

3>、eureka.client.serviceUrl.defaultZone : 指定服务注册中心的地址

Eureka：

1>、eureka可分为三个角色：服务发现者、服务注册者、注册发现中心，但是这三个角色并不和实际部署的模型是一对一的关系

2>、所有的网络通信都是基于http(s)协议的

3>、Eureka和AWS是紧密结合的，无论是配置还是源码，比如Region、zone…，Region可以通过配置文件进行配置，如果不配置默认使用us-east-1。同样Zone也可以配置，若不配置默认使用defaultZone

高可用配置：

Eureka server 的高可用实际上就是将自己作为服务向其他服务注册中心注册自己，这样就可以形成一组互相注册的服务注册中心，以实现服务清单的互相同步，达到高可用效果。

**什么是微服务：**

以前所有的代码都放在同一个工程中、部署在同一个服务器、同一项目的不同模块不同功能互相抢占资源，微服务就是将工程根据不同的业务规则拆分成微服务，部署在不同的服务器上，服务之间相互调用，java中有的微服务有dubbo(只能用来做微服务)、springcloud( 提供了服务的发现、断路器等)。

微服务的特点：

按业务划分为一个独立运行的程序，即服务单元

服务之间通过HTTP协议相互通信

自动化部署

可以用不同的编程语言

可以用不同的存储技术

服务集中化管理

微服务是一个分布式系统

微服务的优势：

1、将一个复杂的业务拆分为若干小的业务，将复杂的业务简单化，新人只需要了解他所接管的服务的代码，减少了新人的学习成本。

2、由于微服务是分布式服务，服务于服务之间没有任何耦合。微服务系统的微服务单元具有很强的横向拓展能力。

3、服务于服务之间采用HTTP网络通信协议来通信，单个服务内部高度耦合，服务与服务之间完全独立，无耦合。这使得微服务可以采用任何的开发语言和技术来实现，提高开发效率、降低开发成本。

4、微服务是按照业务进行拆分的，并有坚实的服务边界，若要重写某一业务代码，不需了解所以业务，重写简单。

5、微服务的每个服务单元是独立部署的，即独立运行在某个进程中，微服务的修改和部署对其他服务没有影响。

6、微服务在CAP理论中采用的AP架构，具有高可用分区容错特点。高可用主要体现在系统7x24不间断服务，他要求系统有大量的服务器集群，从而提高系统的负载能力。分区容错也使得系统更加健壮。

微服务的不足：

1、微服务的复杂度：构建一个微服务比较复杂，服务与服务之间通过HTTP协议或其他消息传递机制通信，开发者要选出最佳的通信机制，并解决网络服务差时带来的风险。

2、分布式事物：将事物分成多阶段提交，如果一阶段某一节点失败仍会导致数据不正确。如果事物涉及的节点很多，某一节点的网络出现异常会导致整个事务处于阻塞状态，大大降低数据库的性能。

3、服务划分：将一个完整的系统拆分成很多个服务，是一件非常困难的事，因为这涉及了具体的业务场景

4、服务部署：最佳部署容器Docker

微服务和SOA的关系：

微服务相对于和ESB联系在一起的SOA轻便敏捷的多，微服务将复杂的业务组件化，也是一种面向服务思想的体现。对于微服务来说，它是SOA的一种体现，但是它比ESB实现的SOA更加轻便、敏捷和简单。

springcloud如何实现服务注册与发现？

服务发布时指定对应的服务名(IP地址和端口号)，将服务注册到注册中心(eureka和zookeeper)，但是这一切是Springcloud自动实现的，只需要在SpringBoot的启动类上加上@EnableDisscoveryClient注解，同一服务修改端口就可以启动多个实例调用方法：传递服务名称通过注册中心获取所有的可用实例，通过负载均衡策略(Ribbon和Feign)调用对应的服务

**Ribbon和Feign的区别：**

Ribbon添加的maven依赖是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添加的maven依赖是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”)，

具体区别：

1、启动类使用的注解不同，Ribbon使用的是@RibbonClient，Feign使用的是@EnableFeignClients

2、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

3、调用方式不同，Ribbon需要自己构建http请求，模拟http请求然后使用RestTemplate发送给其他服务，步骤比较繁琐。Feign则是在Ribbon的基础上进行了一次改进，采用接口调用的方式，将需要调用的其他服务的方法定义成抽象方法即可，不需要自己构建http请求，不过要注意的是抽象方法的注解、方法签名要和提供方的完全一致。

{另一个版本的回答：

Ribbon和Feign都是用于调用其他服务，不过方式不同。

         1、启动类使用的注解不同，Ribbon使用的时@RibbonClient，Feign用的时@EnableFeignClients

         2、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使用@FeignClient声明。

         3、调用方式不同，Ribbon需要自己构建http请求，模拟http请求然后使用RestTemplate发送给其他服务，步骤相当繁琐。

                   Feign则是在Ribbon的基础进行了一次改进，采用接口的方式，将需要调用的其他服务的方法定义成抽象方法即可。

                   不需要自己构建http请求。不过要注意的是抽象方法的注解、方法签名要和提供服务的方法完全一致。

}

## springcloud如何实现服务的注册和发现

         服务在发布时 指定对应的服务名（服务名包括了IP地址和端口） 将服务注册到注册中心（eureka或者zookeeper）

         这一过程时springcloud自动实现  只需要在main方法添加@EnableDisscoveryClient 同一个服务修改端口就可以启动多个实例

         调用方法：传递服务名称通过注册中心获取所有的可用实例，通过负载均衡策略调用（ribbon和feign）对应的服务

**雪崩效应：**

分布式系统中的服务通信依赖于网络，网络不好，必然会对分布式系统带来很大的影响。在分布式系统中，服务之间相互依赖，如果一个服务之间出现了故障或者网络延迟，在高并发的情况下，会导致线程阻塞，在很短的时间内该服务的线程资源会消耗殆尽，最终使得该服务不可用。由于服务的相互依赖，可能会导致整个系统的不可用，这就是“雪崩效应”。为了防止此类事件的发生，分布式系统必然要采取相应的措施，如熔断机制（Springcloud采用的是Hystrix）

**熔断机制：**

1、当一个服务出现故障时，请求失败次数超过设定的阀值（默认50）之后，该服务就会开启熔断器，之后该服务就不进行任何业务逻辑操作，执行快速失败，直接返回请求失败的信息。其他依赖于该服务的服务就不会因为得不到响应而造成线程阻塞，这是除了该服务和依赖于该服务的部分功能不可用外，其他功能正常。

2、熔断器还有一个自我修复机制，当一个服务熔断后，经过一段时间（5s）半打开熔断器。半打开的熔断器会检查一部分请求（只能有一个请求）是否正常，其他请求执行快速失败，检查的请求如果响应成功，则可判断该服务正常了，就可关闭该服务的熔断器，反之则继续打开熔断器。这种自我熔断机制和自我修复机制可以使程序更加健壮、也可以为开发和运维减少很多不必要的工作。

3、熔断组件往往会提供一系列的监控，如：服务可用与否、熔断器是否被打开、目前的吞吐量、网络延迟状态的监控等，从而可以让开发人员和运维人员的了解服务的状况。

{

## 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，断路器就是解决这一问题的。

         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，那么下次的请求不会请求到该服务。

         半开：

                   短时间内有回复迹象，断路器会将部分请求发送给服务，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

}

Eureka基础架构：

1>、服务注册中心：Eureka提供的服务端，提供服务注册与发现的功能

1>>、失效剔除：对于那些非正常下线的服务实例（内存溢出、网络故障导致的），服务注册中心不能收到“服务下线”的请求，为了将这些无法提供服务的实例从服务列表中剔除，Eureka Server在启动的时候会创建一个定时任务，默认每隔一段时间（默认60s）将当前清单中超时（默认90s）没有续约的服务剔除出去。

2>>、自我保护：Eureka Server 在运行期间，会统计心跳失败的比例在15分钟之内是否低于85%，如果出现低于的情况（生产环境由于网络不稳定会导致），Eureka Server会降当前的实例注册信息保护起来，让这些实例不过期，尽可能保护这些注册信息，但是在这保护期间内实例出现问题，那么客户端就很容易拿到实际上已经不存在的服务实例，会出现调用失败的情况，所以客户端必须有容错机制，比如可以使用请求重试、断路器等机制。

在本地进行开发时可以使用 eureka.server.enable-self-preseervation=false参数来关闭保护机制，以确保注册中心可以将不可用的实例剔除。

2>、服务提供者：提供服务的应用，可以是SpringBoot应用也可以是其他的技术平台且遵循Eureka通信机制的应用。他将自己提供的服务注册到Eureka，以供其他应用发现，（如：service层）

1>>、服务注册：服务提供者在启动的时候会通过发送Rest请求的方式将自己注册到Eureka Server（服务注册中心）中，同时带上自身服务的一些元数据，Eureka Server 接收到这个Rest请求后，将元数据存储在一个双层结构Map中，第一层的key是服务名，第二层key是具体服务的实例名

2>>、服务同步：若有两个或两个以上的Eureka Server（服务注册中心）时，他们之间是互相注册的，当服务提供者发送注册请求到一个服务注册中心时，它会将该请求转发到集群中相连的其他注册中心，从而实现注册中心间的服务同步，这样服务提供者的服务信息可以通过任意一台服务中心获取到

3>>、服务续约：在注册完服务之后，服务提供者会维护一个心跳来持续告诉Eureka Server：“我还活着”，以防止Eureka Server的“剔除任务”将该服务实例从服务列表中排除出去。配置：eureka.instance.lease-renewal-in-seconds=30(续约任务的调用间隔时间，默认30秒，也就是每隔30秒向服务端发送一次心跳，证明自己依然存活)，eureka.instance.lease-expiration-duration-in-seconds=90(服务失效时间，默认90秒，也就是告诉服务端，如果90秒之内没有给你发送心跳就证明我“死”了，将我剔除)

3>、服务消费者：消费者应用从服务注册中心获取服务列表，从而使消费者可以知道去何处调用其所需要的服务，如：Ribbon实现消费方式、Feign实现消费方式

1>>、获取服务：当启动服务消费者的时候，它会发送一个Rest请求给注册中心，获取上面注册的服务清单，Eureka Server会维护一份只读的服务清单来返回给客户端，并且每三十秒更新一次

2>>、服务调用：在服务消费者获取到服务清单后，通过服务名可以获得具体提供服务的实例名和该实例的元信息，采用Ribbon实现负载均衡

3>>、服务下线：当服务实例进行正常的关闭操作时，它会触发一个服务下线的Rest请求给Eureka Server，告诉服务注册中心“我要下线了”。服务端接收到请求之后，将该服务状态设置为下线，并把下线时间传播出去。

Eureka和zookeeper都可以提供服务注册与发现的功能，两者的区别：

Zookeeper保证了CP(C：一致性，P：分区容错性)，Eureka保证了AP(A：高可用，P：分区容错)

1、Zookeeper-----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信息，但不能容忍直接down掉不可用的。也就是说服务注册功能对高可用性要求比较高，但是zk会出现这样的一种情况，当master节点因为网络故障与其他节点失去联系时，剩余的节点会重新选leader。问题在于，选取leader的时间过长(30~120s)，且选取期间zk集群都不可用，这样就会导致选取期间注册服务瘫痪。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，虽然服务最终恢复，但是漫长的选择时间导致的注册长期不可用是不能容忍的

2、Eureka则看明白这一点，因此再设计的优先保证了高可用性。Eureka各个节点都是平等的，几个节点挂掉不会影响到正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端再向某个Eureka注册时如果发现连接失败，则会自动切换至其他节点，只要有一台Eureka还在，就能保证注册服务的可用(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，此时就会出现以下几种情况：

1>、Eureka不再从注册列表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(保证当前节点可用)

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

Eureka还有客户端缓存功能(Eureka分为客户端程序和服务器端程序两个部分，客户端程序负责向外提供注册与发现服务接口)。所以即便Eureka集群中所有节点都失效，或者发生网络分隔故障导致客户端不能访问任何一台Eureka服务器；Eureka服务的消费者任然可以通过Eureka客户端缓存来获取所有的服务注册信息。甚至最极端的环境下，所有正常的Eureka节点都不对请求产生响应也没有更好的服务器解决方案来解决这种问题时；得益于Eureka的客户端缓存技术，消费者服务仍然可以通过Eureka客户端查询与获取注册服务信息，这点很重要，因此Eureka可以很好的应对网络故障导致部分节点失去联系的情况，而不像Zookeeper那样使整个注册服务瘫痪。

CAP理论：

1、Consistency：指数据的强一致性。如果写入某个数据成功，之后读取，读到的都是新写入的数据；如果写入失败，读到的都不是写入失败的数据。

2、Availability：指服务的可用性

3、Partition-tolerance：指分区容错

Ribbon和Nginx的区别：

Nginx性能好，但Ribbon可以剔除不健康节点，Nginx剔除比较麻烦，Ribbon是客户端负载均衡，Nginx是服务端负载均衡

服务注册与发现：

服务注册就是向服务注册中心注册一个服务实例，服务提供者将自己的服务信息（服务名、IP地址等）告知注册中心。服务发现是服务消费另一个服务时，注册中心将服务的实例返回给服务消费者，一个服务既是服务提供者又是服务消费者。

服务注册中心健康检查机制，当一个服务实例注册成功以后，会定时向注册中心发送一个心跳证明自己可用，若停止发送心跳证明服务不可用将会别剔除。若过段时间继续想注册中心提供心跳，将会重新加入服务注册中心列表中。

服务的负载均衡：

为什么要用：微服务是将业务代码拆分为很多小的服务单元，服务之间的相互调用通过HTTP协议来调用，为了保证服务的高可用，服务单元往往都是集群化部署的，那么消费者该调用那个服务提供者的实例呢？

介绍：服务消费者集成负载均衡组件，该组件会向服务消费者获取服务注册列表信息，并隔一段时间重新刷新获取列表。当服务消费者消费服务时，负载均衡组件获取服务提供者所有实例的注册信息，并通过一定的负载均衡策略（可以自己配置）选择一个服务提供者实例，向该实例进行服务消费，这样就实现了负载均衡。

————————————————

原文链接：<https://blog.csdn.net/qq_40117549/article/details/84944840>

## Zookeeper是什么框架

         分布式的、开源的分布式应用程序协调服务，原本是Hadoop、Hbase的一个重要组件，它为分布式应用提供一致性服务的软件，

         包括：配置维护、域名服务、分布式同步、组服务等。

应用场景：

         Zookeeper的功能很强大，应用场景很多，Zookeeper主要是做注册中心用。基于Dubbo框架开发的提供者、消费者都向Zookeeper

         注册自己的URL，消费者还能拿到并订阅提供者的注册URL，一边在后续程序的执行中去调用提供者。而提供者发生了变动，

         也会通过Zookeeper向订阅的消费者发送通知。

Zookeeper有哪几种节点类型：

         持久：创建之后一直存在，除非有删除操作，创建节点的客户端会话失效也不影响此节点。

         持久顺序：跟持久一样，就是父节点在创建下一级节点的时候，记录每个子节点创建的先后顺序。会给每个子节点加上一个数字后缀。

         临时：创建客户端会话失效（注意是会话失效，而不是断了），节点也就没了，不能创建子节点。

         临时顺序：。。。。

<https://m.imooc.com/article/23679>

都是图，说是原理，但是看不很懂