Spring cloud是以Spring boot为基础的快速构建分布式系统的工具集

   服务提供者：是指服务的被调用方（即：为其它服务提供服务的服务）

服务消费者：是指服务的调用方（即：依赖其它服务的服务）

SpringBoot简化xml配置，快速整合框架

Springcloud解决什么问题？  
远程调用问题（rpc）、配置管理、注册中心、服务发现、服务注册、断路器、路由策略、负载均衡、分布式会话、客服端调用、网管接口（zuul）、全局锁、服务管理系统

1. eureka是一个高可用的组件，它没有后端缓存，每一个实例注册之后需要向注册中心发送信息（因此可以在内存中完成），默认情况下eureka server也是一个eureka client，必须在application.yml中进行说明。

这高可用下的eureka server 有多台eureka server同时进行相同的工作，在任意一个server注册的client都会显示在所有的server上。

server:

port: 8761

spring:

profiles: peer1

eureka:

instance:

hostname: peer1

client:

serviceUrl:

defaultZone: http://peer2:8769/eureka/

pee2 为另外一个eureka server的名字

单个eureka server进行注册

server:

port: 8761

eureka:

instance:

hostname: localhost

client:

registerWithEureka: false # 是否向 Eureka 注册服务。该应用为服务注册中心，不需要自注册，设置为 false  
fetchRegistry: false # 是否检索服务。该应用为服务注册中心，职责为注册和发现服务，无需检索服务，设置为 false

serviceUrl:

defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

spring:

application:

name: eurka-server

Springcloud两种服务调用方式 ribbon + restTemplate ：ribbon先向注册中心注册，然后通过restTemplate来调用相应的服务 feign：是一个声明式的伪Http客户端，通过创建接口并注解来消费服务。通过@ FeignClient（“服务名”），来指定调用哪个服务。

1. 负载均衡，英文名称为Load Balance，其含义就是指将负载（工作任务）进行平衡、分摊到多个操作单元上进行运行
2. 服务故障的“雪崩”效应：由于网络或者自身原因导致单个服务出现问题，调用这个服务就会出现线程阻塞，当有大量请求涌入，servlet容器的线程资源会被消耗完，导致服务瘫痪。服务与服务有依赖性，故障会传播对整个服务系统造成灾难性的后果。断路器就是解决这类问题。具体实现：ribbon中使用断路器：在调用服务的代码前加上 @HystrixCommand(fallbackMethod = “method”)method为具体的方法。例如public String method(String name){return “error”;}。

在feign中，在配置文件中打开它feign.hystrix.enable = true

1. 负载均衡方式：客户端的请求首先经过负载均衡（zuul、Ngnix），再到达服务网关（zuul集群），然后再到具体的服。，服务统一注册到高可用的服务注册中心集群，服务的所有的配置文件由配置服务管理（下一篇文章讲述），配置服务的配置文件放在git仓库，方便开发人员随时改配置。
2. Zuul的主要功能是路由转发和过滤器，路由的作用：根据路由配置中的path将请求转发给不同的服务。zuulFilter的几个常用函数的功能：
3. filterType：还回一个字符串代表过滤器的类型：pre 路由之前 routing 路由中 post 路由之后 error 发送错误调用
4. filterOrder 过滤的顺序
5. shouldFilter 判断逻辑，是否过滤，true表示永远过滤
6. run 过滤器的具体逻辑

6. 配置中心组件spring cloud config（config server和config client）,可以将配置服务放在配置服务的内存（本地），也可以放在Git仓库。服务获取配置信息的流程：Config serve从Git仓库中读取配置信息，config client 从config server获取配置信息。

7. spring cloud bus（消息总线）将分布式的节点用轻量级的消息代理连接起来。它可以用于广播配置文件的更改或者服务之间的通讯，也可以用于监控。（广播：通知给所有的订阅者），实现通知微服务框架的配置文件更改的流程：当存储在远程仓库或者本地的配置文件发生更改时，PC端用post向其中一个config client发送请求，config client会发送一个消息，由消息总线向其他服务传递，从而使整个服务集群都更新配置文件。

1. 服务链路追踪：用于查看服务之间的调用关系。

Span：基本工作单元，例如，在一个新建的span中发送一个RPC等同于发送一个回应请求给RPC，span通过一个64位ID唯一标识，trace以另一个64位ID表示，span还有其他数据信息，比如摘要、时间戳事件、关键值注释(tags)、span的ID、以及进度ID(通常是IP地址)

span在不断的启动和停止，同时记录了时间信息，当你创建了一个span，你必须在未来的某个时刻停止它。

Trace：一系列spans组成的一个树状结构，例如，如果你正在跑一个分布式大数据工程，你可能需要创建一个trace。

Annotation：用来及时记录一个事件的存在，一些核心annotations用来定义一个请求的开始和结束

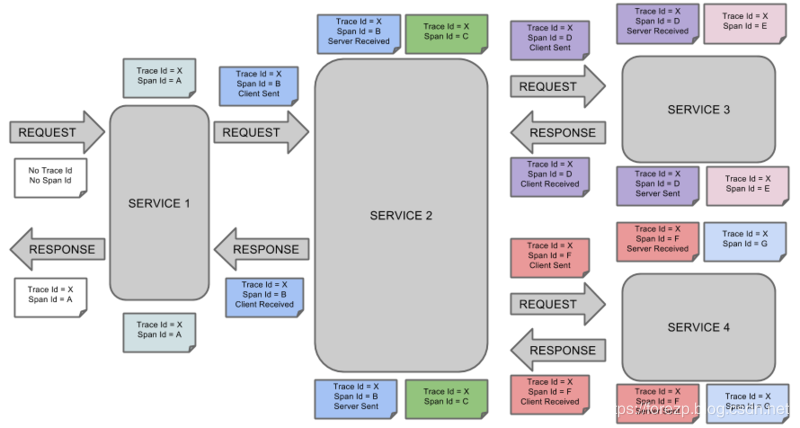
cs - Client Sent -客户端发起一个请求，这个annotion描述了这个span的开始

sr - Server Received -服务端获得请求并准备开始处理它，如果将其sr减去cs时间戳便可得到网络延迟

ss - Server Sent -注解表明请求处理的完成(当请求返回客户端)，如果ss减去sr时间戳便可得到服务端需要的处理请求时间

cr - Client Received -表明span的结束，客户端成功接收到服务端的回复，如果cr减去cs时间戳便可得到客户端从服务端获取回复的所有所需时间。

Span和Trace在一个系统中使用Zipkin注解的过程图形化：



spring cloud gateway ：Hystrix是 spring cloud gateway中是以filter的形式使用的，最重要的两个概念：

Predicates：有一个重要的点就是讲请求和路由进行匹配，这时候就需要用到predicate，它是决定了一个请求走哪一个路由。

filters

网关的作用：

* 协议转换，路由转发
* 流量聚合，对流量进行监控，日志输出
* 作为整个系统的前端工程，对流量进行控制，有限流的作用
* 作为系统的前端边界，外部流量只能通过网关才能访问系统
* 可以在网关层做权限的判断
* 可以在网关层做缓存

1. IMG_256常见的限流方式，比如Hystrix适用线程池隔离，超过线程池的负载，走熔断的逻辑。在一般应用服务器中，比如tomcat容器也是通过限制它的线程数来控制并发的；也有通过时间窗口的平均速度来控制流量。常见的限流纬度有比如通过Ip来限流、通过uri来限流、通过用户访问频次来限流。

限流算法：计数器算法

计数器算法采用计数器实现限流有点简单粗暴，一般我们会限制一秒钟的能够通过的请求数，比如访问量限流：限流qps为100，算法的实现思路就是从第一个请求进来开始计时，在接下去的1s内，每来一个请求，就把计数加1，如果累加的数字达到了100，那么后续的请求就会被全部拒绝。等到1s结束后，把计数恢复成0，重新开始计数。具体的实现可以是这样的：对于每次服务调用，可以通过AtomicLong#incrementAndGet()方法来给计数器加1并返回最新值，通过这个最新值和阈值进行比较。这种实现方式，相信大家都知道有一个弊端：如果我在单位时间1s内的前10ms，已经通过了100个请求，那后面的990ms，只能眼巴巴的把请求拒绝，我们把这种现象称为“突刺现象“

并发量限流：确定方法的最大并发量MAX，每次进入方法前计数器+1，将结果和最大并发量MAX比较，如果大于等于MAX，则直接返回；如果小于MAX，则继续执行；退出方法后计数器-1。比如限制服务的并发访问数是100，而服务处理的平均耗时是10毫秒，那么1分钟内，该服务平均能提供( 1000 / 10 ) \* 60 \* 100 = 6000 次

优缺点：并发量限流一般用于对于服务资源有严格限制的场景，但是某个服务在业务高峰期和低峰期的并发量很难评估，这给并发阈值的设置带来了困难，但我们可以通过线上业务的监控数据来逐步对并发阈值进行调优，只要肯花时间，我们总能找到一个即能保证一定服务质量又能保证一定服务吞吐量的合理并发阈值，从表面上看并发量限流似乎很有用，但也不可否认，它仍然可以造成流量尖刺，即每台服务器上该服务的并发量从0上升到阈值是没有任何“阻力”的，这是因为并发量考虑的只是服务能力边界的问题。

漏桶算法

漏桶算法为了消除"突刺现象"，可以采用漏桶算法实现限流，漏桶算法这个名字就很形象，算法内部有一个容器，类似生活用到的漏斗，当请求进来时，相当于水倒入漏斗，然后从下端小口慢慢匀速的流出。不管上面流量多大，下面流出的速度始终保持不变。不管服务调用方多么不稳定，通过漏桶算法进行限流，每10毫秒处理一次请求。因为处理的速度是固定的，请求进来的速度是未知的，可能突然进来很多请求，没来得及处理的请求就先放在桶里，既然是个桶，肯定是有容量上限，如果桶满了，那么新进来的请求就丢弃。在算法实现方面，可以准备一个队列，用来保存请求，另外通过一个线程池（ScheduledExecutorService）来定期从队列中获取请求并执行，可以一次性获取多个并发执行。这种算法，在使用过后也存在弊端：无法应对短时间的突发流量。

令牌桶算法

可以解决突发传输

从某种意义上讲，令牌桶算法是对漏桶算法的一种改进，桶算法能够限制请求调用的速率，而令牌桶算法能够在限制调用的平均速率的同时还允许一定程度的突发调用。在令牌桶算法中，存在一个桶，用来存放固定数量的令牌。算法中存在一种机制，以一定的速率往桶中放令牌。每次请求调用需要先获取令牌，只有拿到令牌，才有机会继续执行，否则选择选择等待可用的令牌、或者直接拒绝。放令牌这个动作是持续不断的进行，如果桶中令牌数达到上限，就丢弃令牌，所以就存在这种情况，桶中一直有大量的可用令牌，这时进来的请求就可以直接拿到令牌执行，比如设置qps为100，那么限流器初始化完成一秒后，桶中就已经有100个令牌了，这时服务还没完全启动好，等启动完成对外提供服务时，该限流器可以抵挡瞬时的100个请求。所以，只有桶中没有令牌时，请求才会进行等待，最后相当于以一定的速率执行。实现思路：可以准备一个队列，用来保存令牌，另外通过一个线程池定期生成令牌放到队列中，每来一个请求，就从队列中获取一个令牌，并继续执行。