Spring Cloud

Spring Cloud基础组件

> 注册与发现（Eureka,Zookeeper,Consul）
>
> 负载均衡（Ribbon）
>
> Api网关（Gateway, Zuul）
>
> REST调用（Feign）
>
> 容错处理（Hystrix）
>
> 统一管理配置（Config, Apollo,Nacos）
>
> 服务跟踪（Sleuth）

微服务组件——**网关**

一般情况下，**网关**一般都会提供请求转发、安全认证（身份/权限认证）、流量控制、负载均衡、容灾、日志、监控这些功能

网关实现核心，过滤器，路由

常用网关，Kong 基于 Openresty ，Zuul 基于 Java，SpringGateway

基于云平台的产生的概念Service Mesh（服务网格），相较传统部署来说，云平台优点弹性伸缩，相当于云平台的智能网关；

常用Service Mesh，linkerd基于主机部署， istio仅应用与Kubernetes

限流算法：固定窗口，滑动窗口，漏桶，令牌桶

微服务组件——**注册中心**

主要功能，服务注册和服务发现，监控，负载均衡

典型的分布式系统，需要支持高可用

CAP原则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)，不能同时成立

一致性和可用性不能同时满足。在注册中心的发展上面，一直有两个分支：一个就是 CP 系统，追求数据的强一致性。还有一个是 AP 系统，追求高可用与最终一致。我们接下来介绍的服务注册和发现组件中，Eureka满足了其中的AP，Consul和Zookeeper满足了其中的CP

服务注册 Register：

官方解释：当 `Eureka` 客户端向 `Eureka Server` 注册时，它提供自身的**元数据**，比如IP地址、端口，运行状况指示符URL，主页等

服务续约 Renew：

官方解释：`Eureka` **客户会每隔30秒(默认情况下)发送一次心跳来续约。** 通过续约来告知 `Eureka Server` 该 `Eureka` 客户仍然存在，没有出现问题。 正常情况下，如果 `Eureka Server` 在90秒没有收到 `Eureka` 客户的续约，它会将实例从其注册表中删除

获取注册列表信息 Fetch Registries：

官方解释：`Eureka` 客户端从服务器获取注册表信息，并将其缓存在本地。客户端会使用该信息查找其他服务，从而进行远程调用。该注册列表信息定期（每30秒钟）更新一次。每次返回注册列表信息可能与 `Eureka` 客户端的缓存信息不同，`Eureka` 客户端自动处理。如果由于某种原因导致注册列表信息不能及时匹配，`Eureka` 客户端则会重新获取整个注册表信息。 `Eureka` 服务器缓存注册列表信息，整个注册表以及每个应用程序的信息进行了压缩，压缩内容和没有压缩的内容完全相同。`Eureka` 客户端和 `Eureka` 服务器可以使用JSON / XML格式进行通讯。在默认的情况下 `Eureka` 客户端使用压缩 `JSON` 格式来获取注册列表的信息。

服务下线 Cancel：

官方解释：Eureka客户端在程序关闭时向Eureka服务器发送取消请求。 发送请求后，该客户端实例信息将从服务器的实例注册表中删除。该下线请求不会自动完成，它需要调用以下内容：
`DiscoveryManager.getInstance().shutdownComponent();`

服务剔除 Eviction：

官方解释：在默认的情况下，**当Eureka客户端连续90秒(3个续约周期)没有向Eureka服务器发送服务续约，即心跳，Eureka服务器会将该服务实例从服务注册列表删除**，即服务剔除。

zk扩展 https://blog.csdn.net/dwdyoung/article/details/80726883

微服务组件——**负载均衡**（Ribbon）

> `RestTemplate` **是** `Spring` **提供的一个访问Http服务的客户端类**
>
> 微服务之间的调用是使用的 `RestTemplate`
>
> 上面我们所讲的 `Eureka` 框架中的 **注册**、**续约** 等，底层都是使用的 `RestTemplate`

运行在消费者端，也就是调用方，其工作原理就是 `Consumer` 端获取到了所有的服务列表之后，在其**内部**使用**负载均衡算法**，进行对多个系统的调用

请注意 `Request` 的位置，在 `Nginx` 中请求是先进入负载均衡器，而在 `Ribbon` 中是先在客户端进行负载均衡才进行请求的。所以 `Ribbon` 是一种**集中式**的负载均衡器

负载均衡器Ribbon调度算法

- `RoundRobinRule`：轮询策略。`Ribbon` 默认采用的策略。若经过一轮轮询没有找到可用的 `provider`，其最多轮询 10 轮。若最终还没有找到，则返回 `null`。
    - 具体实现是一个负载均衡算法：第N次请求 % 服务器集群的总数 = 实际调用服务器位置的下标
      那么怎么保证线程安全问题呢？因为N次请求次数会自增，怎么保证不会多次请求都拿到同一个N进行自增？答案就是简单的CAS
- `RandomRule`: 随机策略，从所有可用的 `provider` 中随机选择一个。
- `RetryRule`: 重试策略。先按照 `RoundRobinRule` 策略获取 `provider`，若获取失败，则在指定的时限内重试。默认的时限为 500 毫秒。
- WeightedResponseTimeRule响应速度决定权重
- BestAvailableRule最优可用（底层也有RoundRobinRule），判断最优其实用的是并发连接数。选择并发连接数较小的server发送请求

- AvailabilityFilteringRule可用性过滤规则
- ZoneAvoidanceRule区域内可用性能最优

```
@EnableFeignClients
@RibbonClient(name="UUC", configuration=SelfRibbonRule.class) //支持多个

@FeignClient(value="UUC")
```

微服务组件——**REST调用**（Open Feign）

理论上有了 `Eureka` ， `RestTemplate` ， `Ribbon` ， 我们就可以愉快地进行服务间的调用；但不方便；故利用映射，注解简化服务间调用

`OpenFeign` 也是运行在消费者端的，使用 `Ribbon` 进行负载均衡，所以 `OpenFeign` 直接内置了 `Ribbon`

微服务组件——**容错处理**（Hystrix之熔断和降级）

> 在分布式环境中，不可避免地会有许多服务依赖项中的某些失败。Hystrix是一个库，可通过添加等待时间容限和容错逻辑来帮助您控制这些分布式服务之间的交互。Hystrix通过隔离服务之间的访问点，停止服务之间的级联故障并提供后备选项来实现此目的，所有这些都可以提高系统的整体弹性。

A --> B --> C C程序有问题，导致大量请求阻塞；过一会之后，B会因为C的阻塞而变得阻塞，同理，A也会。这种现象叫做**服务雪崩**

熔断 就是指的 `Hystrix` 中的 **断路器模式** ，你可以使用简单的 `@HystrixCommand` 注解来标注某个方法

```
@HystrixCommand(
    commandProperties = {@HystrixProperty(name =
"execution.isolation.thread.timeoutInMilliseconds",value = "1200")}
)
public List<Xxx> getXxxx() {
    // ...省略代码逻辑
}
```

降级，是一种比断路模式更友好的一种模式，可指定后背方法进行安抚用户

```
// 指定了后备方法调用
@HystrixCommand(fallbackMethod = "getHystrixNews")
```

Hystrix隔离方式

支持**信号量**隔离和**线程池**隔离

- 信号量隔离适应非网络请求，因为是同步的请求，无法支持超时，只能依靠协议本身
- 线程池隔离，即，每个实例都增加个线程池进行隔离

> 信号量隔离

> 每次调用线程，当前请求通过计数信号量进行限制，当信号大于了最大请求数（maxConcurrentRequests）时，进行限制，调用fallback接口快速返回
>
> 最重要的是，**信号量的调用是同步的**，也就是说，每次调用都得阻塞调用方的线程，直到结果返回。这样就导致了无法对访问做超时（只能依靠调用协议超时，无法主动释放）

> 线程池隔离
>
> 通过每次都开启一个单独线程运行。它的隔离是通过线程池，即每个隔离粒度都是个线程池，互相不干扰
>
> 线程池隔离方式，等于多了一层的保护措施，可以通过hytrix直接设置超时，超时后直接返回

微服务组件——**统一管理配置，配置中心**（Spring Cloud Config）

简单来说，`Spring Cloud Config` 就是能将各个 应用/系统/模块 的配置文件存放到 **统一的地方然后进行管理**(Git 或者 SVN)。

你想一下，我们的应用是不是只有启动的时候才会进行配置文件的加载，那么我们的 `Spring Cloud Config` 就暴露出一个接口给启动应用来获取它所想要的配置文件，应用获取到配置文件然后再进行它的初始化工作。

可以*使用 Spring Cloud Bus （需要借助消息Kafka）来自动刷新多个端*