01-Features-Discovery Client

Sunday, 2 April 2023 4:18 PM

01-Q1: DiscoveryClient VS KubernetesClient

DiscoveryClient 是SpringCloud对服务发现的一个抽象接口。来自spring-cloud-commons模块。
KubernetesClient是K8S具体实现包里面对K8S java 客户端的一个抽象接口。来自io.fabric8:kubernetes-client包。【因为我用的是fabric8的实现:)】

二者间是接口层的包裹。DiscoveryClient其实是封装了KubernetesClient进行提供服务的。
例如自动配置类：KubernetesDiscoveryClientAutoConfiguration中的实现：

! 注意！
个人感觉关于K8S client其实SpringCloud体系里面实现的也是比较乱的： 例如下面两个包

org.springframework.cloud:spring-cloud-kubernetes-discovery中有这个KubernetesDiscoveryClient的定义。
在自动配置类KubernetesDiscoveryClientAutoConfiguration里面分别实现了servlet && reactive技术栈的KubernetesDiscoveryClient。
但是链接K8S都是依赖KubernetesDiscoveryClientProperties中的url拼接K8S api去获取集群信息数据的。

URL没有提供则抛出：DiscoveryServerUrlInvalidException。

org.springframework.cloud:spring-cloud-kubernetes-fabric8-discovery包中也有KubernetesDiscoveryClient的定义。
在自动配置类KubernetesDiscoveryClientAutoConfiguration里面是用KubernetesDiscoveryClient封装KubernetesClient来实现与K8S集群通信的。
那KubernetesClient更多是在Fabric8AutoConfiguration中进行配置初始化和Client实例化。所以KubernetesDiscoveryClientAutoConfiguration有这个注解：@AutoConfigureAfter({ Fabric8AutoConfiguration.class })
不过这里用到的更多是Endpoints这些K8S model的概念。

```java
@Override
public List<ServiceInstance> getInstances(String serviceId) {
    Assert.notNull(serviceId, message "[Assertion failed] - the object argument must not be null");

    List<EndpointSubsetNS> subsetsNS = this.getEndPointsList(serviceId).stream().map(this::getSubsetsFromEndpoints)
            .collect(Collectors.toList());

    List<ServiceInstance> instances = new ArrayList<>();
    if (!subsetsNS.isEmpty()) {
        for (EndpointSubsetNS es : subsetsNS) {
            instances.addAll(this.getNamespaceServiceInstances(es, serviceId));
        }
    }

    return instances;
}

public List<Endpoints> getEndPointsList(String serviceId) {
    return this.properties.isAllNamespaces()
            ? this.client.endpoints().inAnyNamespace().withField("metadata.name", serviceId)
                    .withLabels(properties.getServiceLabels()).list().getItems()
            : this.client.endpoints().withField("metadata.name", serviceId)
                    .withLabels(properties.getServiceLabels()).list().getItems();
}
```

02-两个包的KubernetsDiscoveryClient有什么问题，谁最终会被加载呢？

首先，如果都被加载，但是有个问题就是BeanDefinition一定是会冲突的。因为二者加载的BeanName一样，所以要有这个配置：**spring.main.allow-bean-definition-overriding: true**

其次我跑下来发现加载的是spring-cloud-kubernetes-fabric8-discovery包中的KubernetesDiscoveryClient。

03-如何识别出当前的application运行在K8S集群环境内呢？

其实主要还是依赖environment。这个AbstractKubernetesProfileEnvironmentPostProcessor的方法isInsideKubernetes(Environment environment)。因为是个抽象方法，委托给了实现类Fabric8ProfileEnvironmentPostProcessor。

```java
public class Fabric8ProfileEnvironmentPostProcessor extends AbstractKubernetesProfileEnvironmentPostProcessor {

    @Override
    protected boolean isInsideKubernetes(Environment environment) {
        try (DefaultKubernetesClient client = new DefaultKubernetesClient()) {
            Fabric8PodUtils podUtils = new Fabric8PodUtils(client);
            return environment.containsProperty(Fabric8PodUtils.KUBERNETES_SERVICE_HOST)
                    || podUtils.isInsideKubernetes();
        }
    }

}
```

【拓展：***EnvironmentPostProcessor***】

由SpringBoot提供。主要作用是：Allows for customization of the application's Environment prior to the application context being refreshed 。

04-如何KubernetesDiscoveryClient是如何与K8S集群通信的呢？

- 针对spring-cloud-kubernetes-discovery包中的KubernetesDiscoveryClient是通过配置装备KubernetesDiscoveryClientProperties这个类。然后从中加载到discoveryServerUrl后，通过内部的RestTemplate发起HTTP请求链接到K8S集群。

- 针对spring-cloud-kubernetes-fabric8-discovery包中的KubernetesDiscoveryClient是通过包装KubernetesClient实现与K8S集群通信。而KubernetesClient是通过Fabric8AutoConfiguration加载配置【kubeConfig文件或者其他纯配置】来进行KubernetesClient的初始化。

05-DiscoveryClient是如何初始化的呢？

首先是spring.factories文件的自动装配引入两个类：CompositeDiscoveryClientAutoConfiguration 和 SimpleDiscoveryClientAutoConfiguration。
DiscoveryClient本就是spring-cloud-commons包中对服务发现功能的抽象。自然，根据Spring一贯的特性就是提供了默认实现-->SimpleDiscoveryClient。
当然，这里Spring做了一个高度的统一管理：CompositeDiscoveryClient,这个类用来管理application中所有的DiscoveryClient的实现。
然后每当我们调用具体的功能的时候，都是通过CompositeDiscoveryClient委托给具体的DiscoveryClient实现类来处理。比如SimpleDiscoveryClient、KubernetesDiscoveryClient等等。

所以大致的初始化流程就是：

第一步：

CompositeDiscoveryClientAutoConfiguration --->CompositeDiscoveryClient

第二步：

SimpleDiscoveryClientAutoConfiguration--->SimpleDiscoveryClient
KubernetesDiscoveryClientAutoConfiguration--->KubernetesDiscoveryClient
…
XxxDiscoveryClientAutoConfiguration--->XxxDiscoveryClient

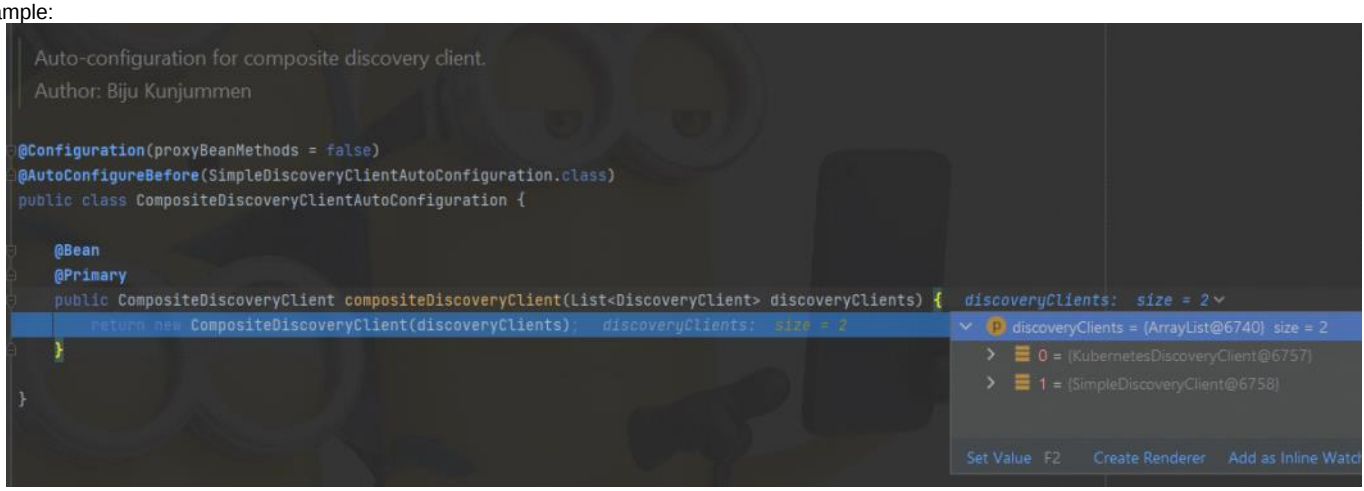
第三步：

SimpleDiscoveryClient && KubernetesDiscoveryClient && … && XXXDiscoveryClient--->CompositeDiscoveryClient

【拓展: Spring自动类型注入】

Spring的自动注入我们知道可以在构造器上参数上不用加注解，会默认从Spring的容器里面去拿符合的bean进行注入。但是有一点比较高级的就是他可以自动进行类型的整合。比如注入一个List<T>,那么Spring会自动帮我们收集好已经存在的符合条件的T类型bean，包装成一个List进行自动注入。

Example:



06-DiscoveryClient提供了服务监听变化的功能，是如何实现的？

其主要实现类是KubernetesCatalogWatch，当然这个类也是自动配置类KubernetesCatalogWatchAutoConfiguration引入的。

它持有一个对Endpoints的原子引用和一个事件发布器publisher以及KubernetesClient，每隔固定时间用KubernetesClient发送一次请求，获取Endpoints信息，并处理后用原子引用包裹，比较前后两次的原子引用对象是否相等来标记服务是否变化。如果有变化就利用publisher发布事件。

当然，这个心跳请求发送的时间间隔我们可以自己配置。

源码参考如下：

```
@Scheduled(fixedDelayString = "${spring.cloud.kubernetes.discovery.catalogServicesWatchDelay:30000}")
public void catalogServicesWatch() {
    try {
        List<String> previousState = this.catalogEndpointsState.get();

        // not all pods participate in the service discovery. only those that have
        // endpoints.
        List<Endpoints> endpoints = this.properties.isAllNamespaces()
                ? this.kubernetesClient.endpoints().inAnyNamespace().withLabels(properties.getServiceLabels())
                        .list().getItems()
                : this.kubernetesClient.endpoints().withLabels(properties.getServiceLabels()).list().getItems();
        List<String> endpointsPodNames = endpoints.stream().map(Endpoints::getSubsets).filter(Objects::nonNull) Stream<List<EndpointSubset>>
                .flatMap(Collection::stream).map(EndpointSubset::getAddresses).filter(Objects::nonNull) Stream<List<EndpointAddress>>
                .flatMap(Collection::stream).map(EndpointAddress::getTargetRef).filter(Objects::nonNull) Stream<ObjectReference>
                .map(ObjectReference::getName) // pod name
                                               // unique in
                                               // namespace
                .sorted(String::compareTo).collect(Collectors.toList());

        this.catalogEndpointsState.set(endpointsPodNames);

        if (!endpointsPodNames.equals(previousState)) {
            logger.trace("Received endpoints update from kubernetesClient: {}", endpointsPodNames);
            this.publisher.publishEvent(new HeartbeatEvent( source: this, endpointsPodNames));
        }
    }
    catch (Exception e) {
        logger.error("Error watching Kubernetes Services", e);
    }
}
```

【拓展：原子引用AtomicReference是如何比较相等的】

并发编程里面的知识。参考如下即可： https://www.baeldung.com/java-atomic-variables

主要我们思考这里为什么要用，它的并发场景在哪里？