# 客户端

**在启动阶段**

1. 读取配置信息封装成响应的配置类

读取与 Eureka Server交互的配置信息,封装成 EurekaClientConfig，也就是配置eureka.Client下的配置

读取自身服务实例配置信息,封装成EurekalnstanceConfig；eureka.instance的配置信息

1. 从服务端发现服务，将服务拉取缓存到本地
2. 服务注册
3. 初始化发送心跳、缓存刷新和按需注册定时任务

刷新缓存：拉取注册表信息更新本地缓存

按需注册：监控服务实例信息变化,决定是否重新发起注册,更新注册表中的服务实例元数据

**应用执行阶段**

1. 定时发送心跳到Eureka Server中维持在注册表的租约
2. 定时从 Eureka Server中拉取注册表信息,更新本地注册表缓存
3. 监控应用自身信息变化,若发生变化,需要重新发起服务注册

**应用销毁阶段**

从 Eureka Server注销自身服务实例

## Starter包

Eureka-client的starter包

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

在META-INF/spring.factories中配置了springcloud 客户端相关的配置类，Eureka Client通过Starter的方式引人依赖, Spring Boot将会为项目使用以下的自动配置类：

**EurekaClientAutoConfiguration：**EurekeClient 自动配置类,负责Eureka关键Beans的配置和初始化,如AppplicationInfoManager和 EurekaClientConfig等。

**RibbonEurekaAutoConfiguration：**Ribbon负载均衡相关配置。

**EurekaDiscoveryClientConfiguration：**配置自动注册和应用的健康检查器。

## EurekaClientAutoConfiguration配置类

通过 **EurekaClientAutoConfiguration**配置类, Spring boot帮助 Eureka Client完成很多必要Bean的属性读取和配置

**EurekaServiceRegistry**：eureka service注册的bean ，负责注册的Bean

@Bean

public EurekaServiceRegistry eurekaServiceRegistry() {

return new EurekaServiceRegistry();}

**EurekaServiceRegistry**：自动注册调用的Bean

@Bean

public EurekaServiceRegistry eurekaServiceRegistry() {

return new EurekaServiceRegistry();

}

EurekaClientConfigBean：读取eureka-client的配置到

EurekaClientConfigBean中；封装 Eureka Client与 Eureka Server交

互所需要的配置信息。 Spring Cloud为其提供了一个默认配置类的

EurekaClientConfigBean,可以在配置文件中通过前缀 eureka.client属性名进行属性覆盖

EurekaInstanceConfigBean： 封装EurekaClient自身服务实例的配置信息,主要用于构建InstanceInfo通常这些信息在配置文件中的eureka.instance前缀下进行设置, SpringCloud通过EurekalnstanceConfigBean配置类提供了默认配置

ApplicationInfoManager

作为应用信息管理器,管理服务实例的信息类 InstanceInfo和服务实例的配置信息类 EurekaInstanceConfig

InstanceInfo

封装将被发送到 Eureka Server进行服务注册的服务实例元数据。它在Eurek Server的注册表中代表一个服务实例,其他服务实例可以通过 Instancelnfo了解该服的相关信息从而发起服务请求

DiscoveryClient（EurekaDiscoveryClient）：Spring Cloud中定义用来服务发现的客户端接口，实现列就是EurekaDiscoveryClient；

@Bean

public DiscoveryClient discoveryClient(EurekaInstanceConfig config, EurekaClient client) {

return new EurekaDiscoveryClient(config, client);

}

## EurekaDiscoveryClient实例的初始化

构造函数参数为EurekaInstanceConfig 和EurekaClient，CloudEurekaClient继承

DiscoveryClient，DiscoveryClient实现了接口EurekaClient，所以参数对象就是在

RefreshableEurekaClientConfiguration这个配置类中实现的实例化，方法就是eurekaClient

在DiscoveryClient的构造函数中

在com.netflix.discovery.DiscoveryClient#DiscoveryClient这个构造函数中实现服务拉取，服务注册，开启心跳，执行定时任务，健康检查相关的设置

（com.netflix.discovery.DiscoveryClient）

1. 相关配置赋值
2. 备份注册中心的初始化，实际什么事都没做
3. 拉取Server注册表中的信息
4. 注册前的预处理
5. 向Server注册自身
6. 初始心跳定时任务，缓存刷新等定时任务

### DiscoveryClient构造函数的执行流程

**1.对AbstractDiscoveryClientOptionalArgs参数的处理，判断是否为空**

if (args != null) {

this.healthCheckHandlerProvider = args.healthCheckHandlerProvider;

this.healthCheckCallbackProvider = args.healthCheckCallbackProvider;

this.eventListeners.addAll(args.getEventListeners());

this.preRegistrationHandler = args.preRegistrationHandler;

} else {

this.healthCheckCallbackProvider = null;

this.healthCheckHandlerProvider = null;

this.preRegistrationHandler = null;

}

**2.获取应用管理器applicationInfoManager，从applicationInfoManager中获取实例信息**

this.applicationInfoManager = applicationInfoManager;

InstanceInfo myInfo = applicationInfoManager.getInfo();

**3.传入BackupRegistry（NotImplementedRegistryImpl）备份注册中心**

**4.从eureka server拉取注册的服务列表 eureka.client.fetch-register**

判断是否从eureka server来取注册的服务列表

if (config.shouldFetchRegistry()) {

this.registryStalenessMonitor = new ThresholdLevelsMetric(this, METRIC\_REGISTRY\_PREFIX + "lastUpdateSec\_", new long[]{15L, 30L, 60L, 120L, 240L, 480L});

} else {

this.registryStalenessMonitor = ThresholdLevelsMetric.NO\_OP\_METRIC;

}

**5.判断是否将服务注册到eureka； eureka.client.register-with-eureka**

if (config.shouldRegisterWithEureka()) {

this.heartbeatStalenessMonitor = new ThresholdLevelsMetric(this, METRIC\_REGISTRATION\_PREFIX + "lastHeartbeatSec\_", new long[]{15L, 30L, 60L, 120L, 240L, 480L});

} else {

this.heartbeatStalenessMonitor = ThresholdLevelsMetric.NO\_OP\_METRIC;

}

如果既不需要注册，也不需要拉去数据，直接返回,初始结束

**6.当前服务要注册**

创建一个容量为2的定时任务线程池， 1个线程用来处理心跳检查，一个用来处理缓存刷新

创建心跳线程池和缓存刷新线程池

**7.初始化client与server交互的jersey客户端（EurekaTransport，用来实现客户端和服务端交互）**

**8：判断是否要用服务端来取服务，**

要来取的话，就分为全量拉取和非全量拉取

在fetchRegistry方法中实现

if (clientConfig.shouldFetchRegistry() && !fetchRegistry(false)) {

fetchRegistryFromBackup();

}

拉取的对象就是eurekaTransport的queryClient对象：

全量拉取执行的方法

getApplications和getVip方法中的一个

获取到服务后，将去缓存到本地，也就是保存到原子引用类的实例中AtomicReference<Applications> **localRegionApps** = **new** AtomicReference<Applications>()

**localRegionApps**.set(**this**.filterAndShuffle(apps));

增量拉取执行的方法

getDelta

将要拉取的实例**remoteRegionsRef**发送到服务端，返回结果就是拉取的服务，如果没有获取到，就进行一次全量拉取，

如果获取到了，就将拉取到的更新到本地缓存中

在DiscoverClient中有Applications的本地对象（ThreadLocal），获取到的对象就放到这个Applications对象中

**9.将当前服务注册到服务器**

if (clientConfig.shouldRegisterWithEureka() && clientConfig.shouldEnforceRegistrationAtInit()) {

try {

if (!register() ) {

throw new IllegalStateException("Registration error at startup. Invalid server response.");

}

} catch (Throwable th) {

logger.error("Registration error at startup: {}", th.getMessage());

throw new IllegalStateException(th);

}

}

registry方法

eurekaTransport.registrationClient.register(instanceInfo);

**10，启动定时任务（）**

initScheduledTasks();

定时拉取服务

当配置了

**eureka**: **client**:  
 **fetch-registry**: **false**

**fetch-registry为true时，就定时从eureka server中拉取服务到本地，在线程**CacheRefreshThread中实现

默认30秒

**if** (**clientConfig**.shouldFetchRegistry()) {  
 *// registry cache refresh timer* **int** registryFetchIntervalSeconds = **clientConfig**.getRegistryFetchIntervalSeconds();  
 **int** expBackOffBound = **clientConfig**.getCacheRefreshExecutorExponentialBackOffBound();  
 **scheduler**.schedule(  
 **new** TimedSupervisorTask(  
 **"cacheRefresh"**,  
 **scheduler**,  
 **cacheRefreshExecutor**,  
 registryFetchIntervalSeconds,  
 TimeUnit.***SECONDS***,  
 expBackOffBound,  
 **new** CacheRefreshThread()  
 ),  
 registryFetchIntervalSeconds, TimeUnit.***SECONDS***);  
}

**eureka**: **client**:**register-with-eureka**: **false** *#服务端不需要注册*

需要配置注册服务

主动向eureka server发送当前的服务实例，实现续租；HeartbeatThread线程具体实现

默认30秒

**if** (**clientConfig**.shouldRegisterWithEureka()) {  
 **int** renewalIntervalInSecs = **instanceInfo**.getLeaseInfo().getRenewalIntervalInSecs();  
 **int** expBackOffBound = **clientConfig**.getHeartbeatExecutorExponentialBackOffBound();  
 ***logger***.info(**"Starting heartbeat executor: "** + **"renew interval is: {}"**, renewalIntervalInSecs);  
  
 *// Heartbeat timer* **scheduler**.schedule(  
 **new** TimedSupervisorTask(  
 **"heartbeat"**,  
 **scheduler**,  
 **heartbeatExecutor**,  
 renewalIntervalInSecs,  
 TimeUnit.***SECONDS***,  
 expBackOffBound,  
 **new** HeartbeatThread()  
 ),  
 renewalIntervalInSecs, TimeUnit.***SECONDS***);  
  
 *// InstanceInfo replicator* **instanceInfoReplicator** = **new** InstanceInfoReplicator(  
 **this**,  
 **instanceInfo**,  
 **clientConfig**.getInstanceInfoReplicationIntervalSeconds(),  
 2); *// burstSize* **statusChangeListener** = **new** ApplicationInfoManager.StatusChangeListener() {  
 @Override  
 **public** String getId() {  
 **return "statusChangeListener"**;  
 }  
  
 @Override  
 **public void** notify(StatusChangeEvent statusChangeEvent) {  
 **if** (InstanceStatus.***DOWN*** == statusChangeEvent.getStatus() ||  
 InstanceStatus.***DOWN*** == statusChangeEvent.getPreviousStatus()) {  
 *// log at warn level if DOWN was involved* ***logger***.warn(**"Saw local status change event {}"**, statusChangeEvent);  
 } **else** {  
 ***logger***.info(**"Saw local status change event {}"**, statusChangeEvent);  
 }  
 **instanceInfoReplicator**.onDemandUpdate();  
 }  
 };  
  
 **if** (**clientConfig**.shouldOnDemandUpdateStatusChange()) {  
 **applicationInfoManager**.registerStatusChangeListener(**statusChangeListener**);  
 }  
  
 **instanceInfoReplicator**.start(**clientConfig**.getInitialInstanceInfoReplicationIntervalSeconds());  
}

## 服务下线

关闭客户端时执行的方法，通知eureka server当前服务下线了

定义了@PreDestroy注解的方法，在bean销毁时执行

注销状态监听器

取消定时任务

关闭与server连接的客户端

关闭相关监控

**public synchronized void** shutdown() {  
 **if** (**isShutdown**.compareAndSet(**false**, **true**)) {  
 ***logger***.info(**"Shutting down DiscoveryClient ..."**);  
  
 **if** (**statusChangeListener** != **null** && **applicationInfoManager** != **null**) {  
 **applicationInfoManager**.unregisterStatusChangeListener(**statusChangeListener**.getId());  
 }  
  
 cancelScheduledTasks();  
  
 *// If APPINFO was registered* **if** (**applicationInfoManager** != **null** && **clientConfig**.shouldRegisterWithEureka()  
 && **clientConfig**.shouldUnregisterOnShutdown()) {  
 **applicationInfoManager**.setInstanceStatus(InstanceStatus.***DOWN***);  
 unregister();  
 }  
  
 **if** (**eurekaTransport** != **null**) {  
 **eurekaTransport**.shutdown();  
 }  
  
 **heartbeatStalenessMonitor**.shutdown();  
 **registryStalenessMonitor**.shutdown();  
  
 ***logger***.info(**"Completed shut down of DiscoveryClient"**);  
 }  
}

# 服务端

EurekaServer 是服务的注册中心，负责Eureka Client的相关信息注册，主要职责

* 服务注册
* 接受心跳服务
* 服务剔除
* 服务下线
* 集群同步

## Starter包

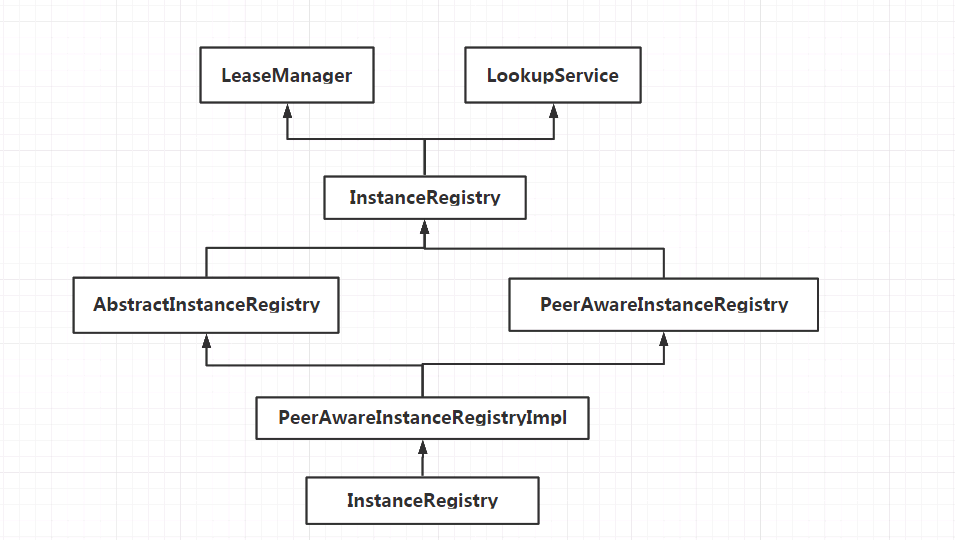
<dependency>

<groupId>org.springframework.cloud</groupId>

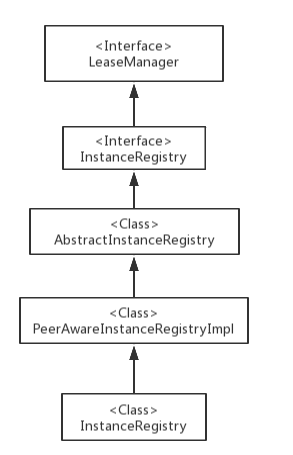
<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

在META-INF/spring.factories中配置了springcloud的配置类EurekaServerAutoConfiguration，在这个类中配置InstanceRegistry这个对象的bean,在这个类中实现注册，下线，续租，剔除，同步等



## 继承体系



LookupService接口提供用来查找存活服务实例的方法

LeaseManager接口：提供注册，下线，续租，剔除这几个方法

public interface LeaseManager<T> {

注册方法

void register(T var1, int var2, boolean var3);

下线方法，客户端下线通知服务端删除服务

boolean cancel(String var1, String var2, boolean var3);

续租

boolean renew(String var1, String var2, boolean var3);

剔除：主动剔除没有定时续租的服务

void evict();

}

AbstractInstanceRegistry抽象类中实现了LeaseManager，LookupService的方法

租约lease

private long evictionTimestamp;剔除时间

private long registrationTimestamp;注册时间

private long serviceUpTimestamp;

private volatile long lastUpdateTimestamp;最后更新时间

private long duration;持续时间

### 注册服务

在AbstractInstanceRegistry中定义了registry这个ConcurrentHashMap对象，作为注册中存放注册信息的容器

ConcurrentHashMap<String, Map<String, Lease<InstanceInfo>>> registry

= new ConcurrentHashMap<String, Map<String, Lease<InstanceInfo>>>();

key是应用的名称，value是一个map，这个map的key是实例的id，value为租约lease对象

com.netflix.eureka.registry.AbstractInstanceRegistry#register 这方法是负责服务的注册的

根据服务名称从registry中取出租约，如果没有就创建新的，创建新的leaselease中包含了服务实例，然后将器放到map中，如果该实例的租约已经存在，比较最后的更新时间戳大小，取最大值的注册信息，那么就拿出之前存在的实例，更新ServiceUpTimestamp

1.取出实例名称，从registry中获取map对象，

判断map对象是否存在（存在表示之前已经注册过了）

如果不存在那么就是创建一个map对象

2.取出租约，

如果租约存在那么表示已经注册过了

比较最后的更新时间戳大小，取最大值的注册信息信息，取出租约对应的注册实例

如果租约不存在，还没有注册过，

3.创建一个租约实例，注入实例信息和持续时间

4，将租约放到注册中心

### 心跳服务

在Eureka Client完成服务的注册后，需要定时向Eureka Server发送心跳请求（默认30s）,维持自己在EurekaServer的租约有效性

renew方法

（com.netflix.eureka.registry.AbstractInstanceRegistry#renew）

在Eureka Client完成服务的注册后，需要定时向Eureka Server发送心跳请求（默认30s）,

维持自己在EurekaServer的租约有效性

1.根据appName获取服务集群租约集合

2.如果租约不存在，直接返回false

Map<String, Lease<InstanceInfo>> gMap = registry.get(appName);

Lease<InstanceInfo> leaseToRenew = null;

if (gMap != null) {

leaseToRenew = gMap.get(id);

}

if (leaseToRenew == null) {

RENEW\_NOT\_FOUND.increment(isReplication);

logger.warn("DS: Registry: lease doesn't exist, registering resource: {} - {}", appName, id);

return false;

}

3.得到服务的最终状态，如果状态为UNKNOWN，取消续约

4.跟新续约有效时间

### 服务剔除

com.netflix.eureka.registry.AbstractInstanceRegistry#evict：剔除方法

这是个定时任务调用的方法

com.netflix.eureka.registry.AbstractInstanceRegistry#postInit中使用

AbstractInstanceRegistry.EvictionTask 负责调用(默认60s)

1.如果自我保护状态，不允许剔除服务

2.遍历注册表registry，获取所有过期的租约

3.获取注册表租约总数

4.计算最多允许剔除的阈值，两者中取小的值，为本常剔除的数量

5.逐个剔除

### 服务下线

EurekaClient在应用销毁时候，会向Eureka Server发送下线请求

对于服务端的服务下线，其主要代码对应在

com.netflix.eureka.registry.AbstractInstanceRegistry#cancel

1.根据appName获取服务实例集群

2.移除服务实例租约

3.租约不存在，返回false

4.设置租约的下线时间

5.设置缓存过期

### 集群同步

如果Eureka Server是通过集群方式进行部署，为了为维护整个集群中注册表数据一致性所以集群同步也是非常重要得事情。

集群同步分为两部分

1. EurekaServer在启动过程中从他的peer节点中拉取注册表信息，并讲这些服务实例注册到本地注册表中；

2. 另一部分是eureka server每次对本地注册表进行操作时，同时会讲操作同步到他的peer节点中，达到数据一致；

#### Eureka Server初始化本地注册表信息

在eureka server启动过程中，会从它的peer节点中拉取注册表来初始化本地注册表，这部分主要通过

com.netflix.eureka.registry.PeerAwareInstanceRegistryImpl#**syncUp**

他从可能存在的peer节点中，拉取peer节点中的注册表信息，并将其中的服务实例的信息注册到本地注册表中。

#### Eureka Server之间注册表信息同步复制