

# 微服务注册中心

---

## 一、微服务注册中心简介

### 1 什么是微服务注册中心

提供者将自己提供服务的名称及自己主机详情（IP、端口等）写入到另一台主机中的一个列表中，这个列表称为服务注册表；所有消费者需要调用微服务时，首先从这台主机中将服务注册表下载到本地，然后根据消费者本地设置好的负载均衡策略选择一个服务提供者进行调用。那么，这台主机就称为微服务注册中心。

### 2 注册中心架构

 1621241482828

### 3 注册中心的主要功能

微服务注册中心的主要功能主要有以下几点：

- 实现provider与consumer间的解耦合。提供者对于消费者来说是透明的，不固定的
- 使消费者调用提供者实现负载均衡成为可能
- 通过微服务注册中心的Dashboard监控微服务的运行状态

### 4 常见的注册中心

可以充当服务注册中心的服务器很多，但一般情况下，不同的微服务生态，使用不同的注册中心。

- 若微服务使用的是Dubbo，一般注册中心使用Zookeeper
- 若微服务使用的是Spring Cloud，一般注册中心使用Eureka或Consul
- 若微服务使用的是Spring Cloud Alibaba，一般注册中心使用Nacos

## 二、常见注册中心简介

### 1 Zookeeper

ZooKeeper由雅虎研究院开发，后来捐赠给了Apache。ZooKeeper是一个开源的分布式应用程序协调服务器，其为分布式系统提供一致性服务。其一致性是通过基于Paxos算法的ZAB协议完成的。其主要功能包括：配置维护、命名服务、分布式同步、集群管理、DNS服务、Master选举、分布式锁、分布式队列等。其中DNS服务就是提供的注册中心服务。

## 2 Eureka

Eureka是Netflix开发的服务发现框架，本身是一个基于REST的服务，主要用于定位运行在AWS域中的中间层服务，以达到负载均衡和中间层服务故障转移的目的。SpringCloud将它集成在其子项目spring-cloud-netflix中，实现SpringCloud的服务发现功能。

## Eureka 2.0 (Discontinued)

The existing open source work on eureka 2.0 is discontinued. The code base and artifacts that were released as part of the existing repository of work on the 2.x branch is considered use at your own risk.

Eureka 1.x is a core part of Netflix's service discovery system and is still an active project.

现在的有关eureka 2.0的开源工作已经终止。已经发布的现存库中的关于2.x分支部分的代码库与工程，你的使用将自负风险。

Eureka 1.x是Netflix服务发现系统的核心部分，其仍是一个活跃项目。

## 3 Consul

Consul 是 HashiCorp 公司推出的开源工具，用于实现分布式系统的服务发现与配置。使用Go语言编写，使用Key/Value存储，采用Raft一致性协议，支持多数据中心。

2020年5月，HashiCorp公司官方宣布，不允许中国境内使用、部署和安装该企业旗下的产品和软件。其中就包含Consul。不过，现在的官方声明中已看不到此限制了。下面是官方声明链接：

<https://www.hashicorp.com/terms-of-evaluation>

## 4 Nacos

Nacos, Dynamic **N**aming and **C**onfiguration **S**ervice，动态命名与配置服务。

Nacos是阿里巴巴开源的一款支持服务注册与发现，配置管理以及微服务管理的组件。用来取代以前常用的注册中心（zookeeper，eureka等等），以及配置中心（spring cloud config等等）。Nacos是集成了注册中心和配置中心的功能，做到了二合一。

Nacos = 服务注册中心 + 服务配置中心 = Eureka + Spring Cloud Config + Spring Cloud Bus + Kafka/RabbitMQ

## 5 对比

比较项	ZOOKEEPER	EUREKA	CONSUL	NACOS
CAP	CP	AP	CP	CP/AP
一致性算法	Paxos	-	Raft	Raft
自我保护机制	无	有	无	有
SpringCloud集成	支持	支持	支持	支持
Dubbo集成	支持	不支持	支持	支持
K8S集成	不支持	不支持	支持	支持

## 三、Eureka源码解析预备知识

### 1 Eureka的异地多活

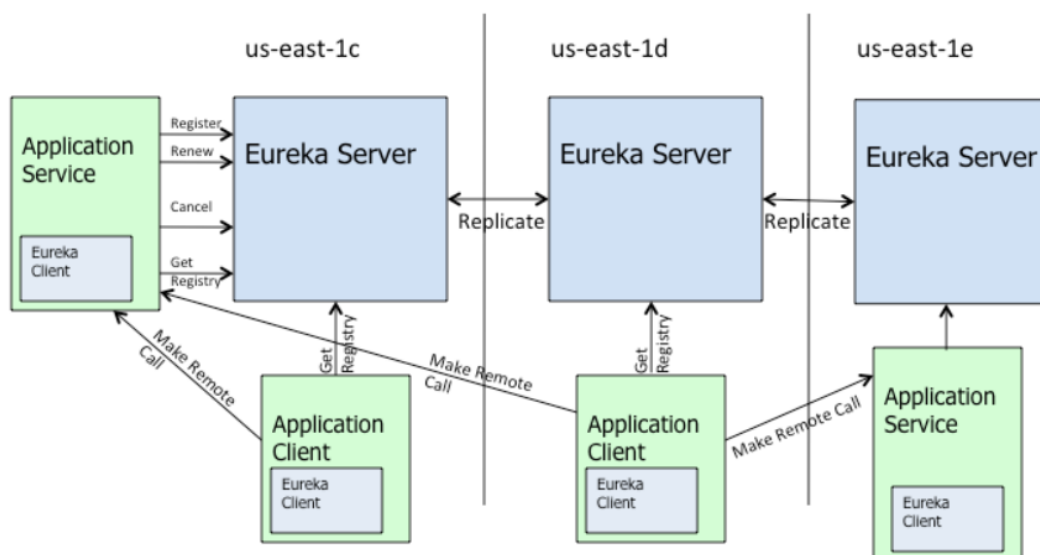
**异地**多活一般是指在不同城市建立独立的数据中心。

**活**是相对于**主备**关系中的热备而言的。热备是指备份机房随时全量备份着主机房中的数据，但平时不支撑业务需求，即不对外提供服务。只有在主机房出现故障时才会切换到备份机房，由备份机房对外提供服务。也就是说，平时只有主机房是**活**的。

**多活**则是指这些机房间属于**主从**关系，即这些机房平时都支撑业务需求，都对外提供服务，相互备份。

### 2 Region与Zone

## High level architecture



Eureka中具有Region与Availability Zone（简称AZ）概念，都是云计算中的概念。

为了方便不同地理区域中用户的使用，大型云服务提供商一般会根据用户需求量在不同的城市、省份、国家或洲创建不同的大型云计算机房。这些不同区域机房间一般是不能“内网连通”的。这些区域就称为一个Region。

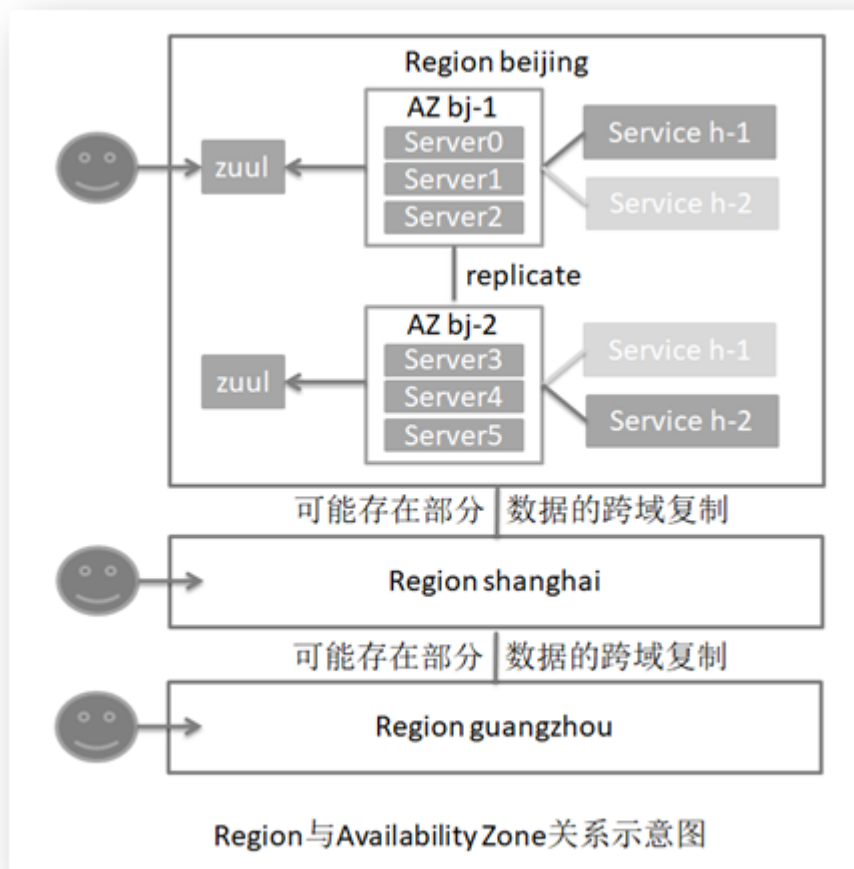
这里存在一个问题：同一Region机房是如何实现同域容灾的？为了增强容灾能力，在一个Region中又设置了不同的Availability Zone。这些AZ间实现了内网连通，且用户可以根据自己所在的具体位置自动选择同域中的不同AZ。当用户所要访问的AZ出现问题后，系统会自动切换到其它可用的AZ。

例如，AWS将全球划分为了很多的Region，例如美国东部区、美国西部区、欧洲区、非洲开普敦区、亚太区等。像Eureka系统架构图中的us-east-1c、us-east-1d、us-east-1e就是us-east-1这个Region中的c、d、e三个AZ。

再如，阿里云在我国境内的Region有杭州、北京、深圳、青岛、香港等，境外Region有亚太东南1区（新加坡）、亚太东南2区（悉尼）、亚太东北1区（东京）等。

### 3 Eureka中的Region与Zone配置

#### 需求



假设某公司的服务器有Beijing、Shanghai等多个Region。Beijing这个Region中存在两个AZ，分别是bj-1与bj-2，每个AZ中有三台Eureka Server。

h-1与h-2两台主机提供的都是相同的Service服务，根据地理位置的不同，这两台主机分别注册到了距离自己最近的不同AZ的Eureka Server。

### Server AZ bj-1配置

```
server:
  port: 8000

eureka:
  instance:
    metadata-map:
      zone: bj-1
  client:
    region: beijing
    register-with-eureka: false
    fetch-registry: false
    service-url:
      defaultZone: http://ip00:8000/eureka,...,http://ip02:8002/eureka,\
                  http://ip10:8000/eureka,...,http://ip12:8002/eureka
```

Server AZ bj-2配置

```
server:
  port: 8000

eureka:
  instance:
    metadata-map:
      zone: bj-2
  client:
    region: beijing
    register-with-eureka: false
    fetch-registry: false
    service-url:
      defaultZone: http://ip00:8000/eureka,...,http://ip02:8002/eureka,\
                  http://ip10:8000/eureka,...,http://ip12:8002/eureka
```

Zuul AZ bj-1配置

```
server:
  port: 9000

spring:
  application:
    name: abcm-sc-zuul-depart

eureka:
  instance:
    metadata-map:
      zone: bj-1
  client:
    region: beijing
    service-url:
      defaultZone: http://ip00:8000/eureka,...,http://ip02:8002/eureka,\
                  http://ip10:8000/eureka,...,http://ip12:8002/eureka

# 其它配置略
# ...
```

## Zuul AZ bj-2配置

```
server:
  port: 9000

spring:
  application:
    name: abcm-sc-zuul-depart

eureka:
  instance:
    metadata-map:
      zone: bj-2
  client:
    region: beijing
    service-url:
      defaultZone: http://ip00:8000/eureka,...,http://ip02:8002/eureka,\
                  http://ip10:8000/eureka,...,http://ip12:8002/eureka

# 其它配置略
# ...
```

## Service AZ bj-1配置



```
server:
  port: 8081

spring:
  application:
    name: abcm-sc-service-depart

eureka:
  instance:
    metadata-map:
      zone: bj-1
  client:
    region: beijing
    service-url:
      defaultZone: http://ip00:8000/eureka,...,http://ip02:8002/eureka,\
                  http://ip10:8000/eureka,...,http://ip12:8002/eureka

# 其它配置略
# ...
```



## Service AZ bj-2配置

```
server:
  port: 8081

spring:
  application:
    name: abcm-sc-service-depart

eureka:
  instance:
    metadata-map:
      zone: bj-2
  client:
    region: beijing
    service-url:
      defaultZone: http://ip00:8000/eureka,...,http://ip02:8002/eureka,\
                  http://ip10:8000/eureka,...,http://ip12:8002/eureka

# 其它配置略
# ...
```



## 优先选择同zone服务配置

```
server:
  port: 8081

spring:
  application:
    name: abcmvc-service-depart

eureka:
  instance:
    metadata-map:
      zone: bj-1
  client:
    region: beijing
    service-url:
      defaultZone: http://ip00:8000/eureka,...,http://ip02:8002/eureka,\
                  http://ip10:8000/eureka,...,http://ip12:8002/eureka
      # 优先选择zone相同的服务, 默认为true
      prefer-same-zone-eureka: true
```

当一个region有多个zone时，微服务调用应用时优先调用同zone内的应用。原因是eureka client有个配置prefer-same-zone-eureka，默认为true。当同zone中的应用均不可用时，才会调用其它zone中的服务。

## 指定远程Region配置

```
eureka:
  instance:
    metadata-map:
      zone: bj-1
  client:
    region: beijing
    service-url:
      defaultZone: http://ip00:8000/eureka,...,http://ip02:8002/eureka,\
                  http://ip10:8000/eureka,...,http://ip12:8002/eureka,\
                  # ... 其它Region中的eureka地址

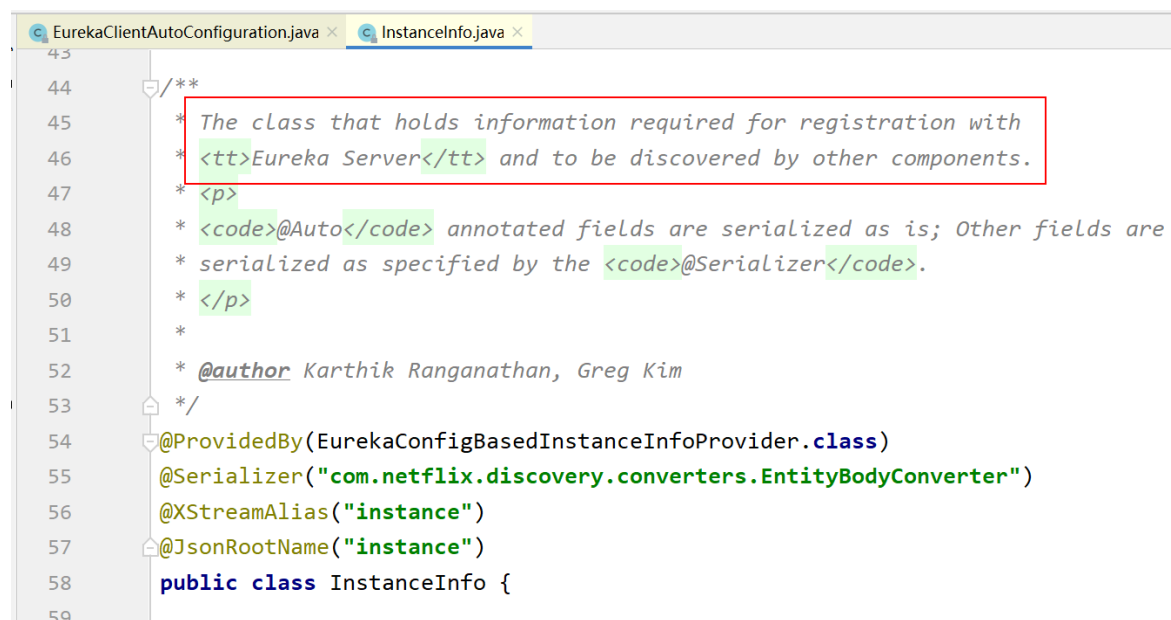
      # 指定要连接的远程Region
      fetch-remote-regions-registry: shanghai,guangzhou
```

对于本地Region中的所有服务均不可用时，可以指定从远程Region获取。不过这种情况获取到的结果可能与从本地Region获取的结果不同，除非这些Region实现了跨域数据复制。若要想从远程Region获取服务，客户端需要通过设置如上属性指定远程Region，并在defaultZone中指出这些远程域中Eureka Server的地址。

## 四、Eureka Client源码解析

### 1 Eureka Client源码的重要API

#### InstanceInfo



该类用于保存一个微服务主机的信息。一个该类实例就代表了一个微服务主机。该主机注册到Eureka Server就是将其InstanceInfo写入到了Eureka注册表，且被其它Server读取到的该Server的信息也是这个InstanceInfo。

```
1 // 记录当前InstanceInfo在Server端被修改的时间戳
2 private volatile Long lastUpdatedTimestamp;
3 // 记录当前InstanceInfo在Client端被修改的时间戳
4 private volatile Long lastDirtyTimestamp;
```

```
1 // 记录当前Client在Server端的状态
2 private volatile InstanceStatus status = InstanceStatus.UP;
3 // 该状态用于计算Client在Server端的状态status(在Client提交注册请求与Renew续约请求时)
4 private volatile InstanceStatus overriddenStatus =
  InstanceStatus.UNKNOWN;
```

```
1 public enum InstanceStatus {
```

```

2      UP, // Ready to receive traffic
3      DOWN, // Do not send traffic- healthcheck callback failed
4      STARTING, // Just about starting- initializations to be done -
do not
5      // send traffic
6      OUT_OF_SERVICE, // Intentionally shutdown for traffic
7      UNKNOWN;
8
9      public static InstanceStatus toEnum(String s) {
10         if (s != null) {
11             try {
12                 return InstanceStatus.valueOf(s.toUpperCase());
13             } catch (IllegalArgumentException e) {
14                 // ignore and fall through to unknown
15                 Logger.debug("illegal argument supplied to
InstanceStatus.valueOf: {}, defaulting to {}", s, UNKNOWN);
16             }
17         }
18         return UNKNOWN;
19     }
20 }

```

```

1 // 重写了equals()方法: 只要两个InstanceInfo的instanceId相同, 那么这两个
InstanceInfo就相同
2 @Override
3     public boolean equals(Object obj) {
4         if (this == obj) {
5             return true;
6         }
7         if (obj == null) {
8             return false;
9         }
10        if (getClass() != obj.getClass()) {
11            return false;
12        }
13        InstanceInfo other = (InstanceInfo) obj;
14        // 获取到instanceId
15        String id = getId();
16        if (id == null) {
17            if (other.getId() != null) {
18                return false;
19            }
20            // 比较两个instanceId
21        } else if (!id.equals(other.getId())) {
22            return false;
23        }
24        return true;

```

## Application

```
EurekaClientAutoConfiguration.java × InstanceInfo.java × Application.java ×
45
46  /**
47   * The application class holds the list of instances for a particular
48   * application.
49   *
50   * @author Karthik Ranganathan
51   *
52   */
53  @Serializer("com.netflix.discovery.converters.EntityBodyConverter")
54  @XStreamAlias("application")
55  @JsonRootName("application")
56  public class Application {
57
```

一个Application实例中保存着一个特定微服务的所有提供者实例。

```
1 // 保存着当前name所指定的微服务名称的所有InstanceInfo
2 private final Set<InstanceInfo> instances;
3
4 private final AtomicReference<List<InstanceInfo>> shuffledInstances;
5 // key为instanceId, value为instanceInfo
6 private final Map<String, InstanceInfo> instancesMap;
7
```

## Applications

```
EurekaClientAutoConfiguration.java × InstanceInfo.java × Application.java × Applications.java ×
49  /**
50   * The class that wraps all the registry information returned by eureka server.
51   *
52   * <p>
53   * Note that the registry information is fetched from eureka server as specified
54   * in {@link EurekaClientConfig#getRegistryFetchIntervalSeconds()}. Once the
55   * information is fetched it is shuffled and also filtered for instances with
56   * {@link InstanceStatus#UP} status as specified by the configuration
57   * {@link EurekaClientConfig#shouldFilterOnlyUpInstances()}.
58   * </p>
59   *
60   * @author Karthik Ranganathan
61   *
62   */
63   @Serializer("com.netflix.discovery.converters.EntityBodyConverter")
64   @XStreamAlias("applications")
65   @JsonRootName("applications")
66   public class Applications {
```

该类封装了来自于Eureka Server的所有注册信息。我们可以称其为“客户端注册表”。之所以要强调“客户端”是因为，服务端的注册表不是这样表示的，是一个Map。

```
1 // key为微服务名称, value为Application
2 private final Map<String, Application> appNameApplicationMap;
```

## Jersey框架

Spring Cloud中Eureka Client与Eureka Server的通信，及Eureka Server间的通信，均采用的是Jersey框架。

Jersey框架是一个开源的RESTful框架，实现了JAX-RS规范。该框架的作用与Spring MVC是相同的，其也是用户提交URI后，在处理器中进行路由匹配，路由到指定的后台业务。这个路由功能同样也是通过处理器完成的，只不过这里的处理器不叫Controller，而叫Resource。

```
1 /**
2  * A <em>jersey</em> resource that handles operations for a particular
3  * instance.
4  *
5  * @author Karthik Ranganathan, Greg Kim
6  *
7  */
8  @Produces({"application/xml", "application/json"})
9  public class InstanceResource {
```

## 2 Eureka Client解析入口中的重要类

### EurekaClientAutoConfiguration类

```

85  * @author Tim Ysewyn
86  */
87  @Configuration(proxyBeanMethods = false)
88  @EnableConfigurationProperties
89  @ConditionalOnClass(EurekaClientConfig.class)
90  @Import(DiscoveryClientOptionalArgsConfiguration.class)
91  @ConditionalOnProperty(value = "eureka.client.enabled", matchIfMissing = true)
92  @ConditionalOnDiscoveryEnabled
93  @AutoConfigureBefore({ NoopDiscoveryClientAutoConfiguration.class,
94      CommonsClientAutoConfiguration.class, ServiceRegistryAutoConfiguration.class })
95  @AutoConfigureAfter(name = {
96      "org.springframework.cloud.autoconfigure.RefreshAutoConfiguration",
97      "org.springframework.cloud.netflix.eureka.EurekaDiscoveryClientConfiguration",
98      "org.springframework.cloud.client.serviceregistry.AutoServiceRegistrationConfiguration" })
99  public class EurekaClientAutoConfiguration {
100

```

这是一个配置类，在应用启动时会创建该类中满足条件的Bean。其中就包含EurekaClient。

## 封装配置文件中的两个属性的类

### EurekaClientConfigBean

其读取的是eureka.client前缀的配置信息。这个类已经被@ConfigurationProperties注解了，所以这些配置信息可以被自动封装并注册到容器。

```

112
113  @Bean
114  @ConditionalOnMissingBean(value = EurekaClientConfig.class, search = SearchStrategy.CURRENT)
115  public EurekaClientConfigBean eurekaClientConfigBean(ConfigurableEnvironment env) {
116      EurekaClientConfigBean client = new EurekaClientConfigBean();
117      if ("bootstrap".equals(this.env.getProperty("spring.config.name"))) {
118          // We don't register during bootstrap by default, but there will be another
119          // chance later.
120          client.setRegisterWithEureka(false);
121      }
122      return client;
123  }

```

```

43  */
44  @ConfigurationProperties(EurekaClientConfigBean.PREFIX)
45  public class EurekaClientConfigBean implements EurekaClientConfig, Ordered {
46
47      /**
48       * Default prefix for Eureka client config properties.
49       */
50      public static final String PREFIX = "eureka.client";
51
52      /**
53       * Default Eureka URL.
54       */
55      public static final String DEFAULT_URL = "http://localhost:8761" + DEFAULT_PREFIX
56          + "/";
57

```

## EurekaInstanceConfigBean

其读取的是eureka.instance的属性值。这个类也已经被@ConfigurationProperties注解了，所以这些配置信息可以被自动封装并注册到容器。

```

134
135  @Bean
136  @ConditionalOnMissingBean(value = EurekaInstanceConfig.class, search = Search
137  @ public EurekaInstanceConfigBean eurekaInstanceConfigBean(InetUtils inetUtils,
138      ManagementMetadataProvider managementMetadataProvider) {
139      String hostname = getProperty("eureka.instance.hostname");
140      boolean preferIpAddress = Boolean
141          .parseBoolean(getProperty("eureka.instance.prefer-ip-address"));
142      String ipAddress = getProperty("eureka.instance.ip-address");
143      boolean isSecurePortEnabled = Boolean

```

```

40  */
41  @ConfigurationProperties("eureka.instance")
42  public class EurekaInstanceConfigBean
43      implements CloudEurekaInstanceConfig, EnvironmentAware {
44
45      private static final String UNKNOWN = "unknown";
46
47      private HostInfo hostInfo;
48
49      private InetUtils inetUtils;
50

```

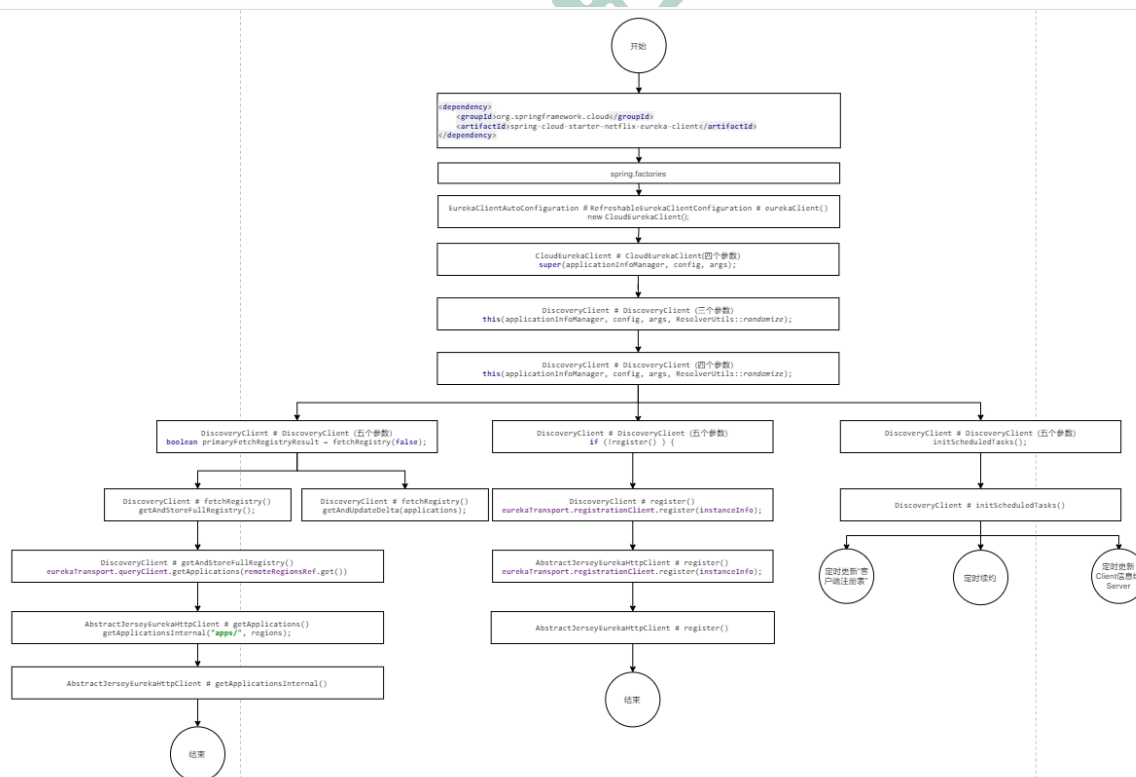
## EurekaClient

这就是我们寻找的客户端实例。默认情况下创建的Eureka Client本身可以实现动态更新，即配置文件中相关的配置信息发生了变更，这个Client注册到Server的信息也会自动变更到Eureka Server的注册表中。

```
EurekaClientAutoConfiguration.java
292 @Configuration(proxyBeanMethods = false)
293 @ConditionalOnRefreshScope
294 protected static class RefreshableEurekaClientConfiguration {
295
296     @Autowired
297     private ApplicationContext context;
298
299     @Autowired
300     private AbstractDiscoveryClientOptionalArgs<?> optionalArgs;
301
302     @Bean(destroyMethod = "shutdown")
303     @ConditionalOnMissingBean(value = EurekaClient.class,
304                               search = SearchStrategy.CURRENT)
305     @org.springframework.cloud.context.config.annotation.RefreshScope
306     @Lazy
307     public EurekaClient eurekaClient(ApplicationInfoManager manager,
308                                     EurekaInstanceConfig config, EurekaInstanceConfig instance,
309                                     @Autowired(required = false) HealthCheckHandler healthCheckHandler)
```

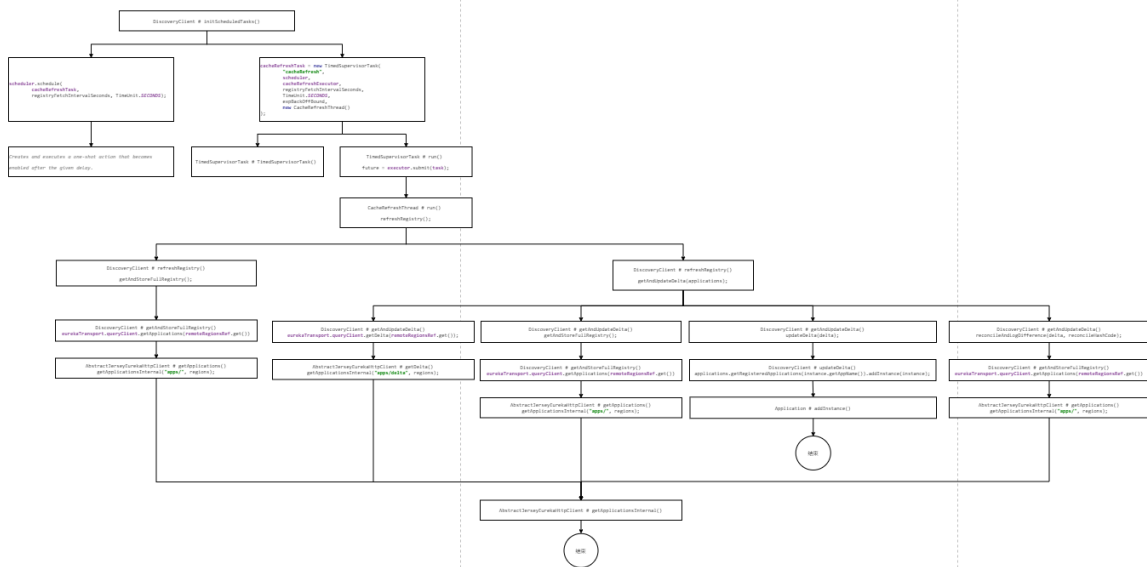
### 3 Eureka Client源码解析

#### 解析总流程





## 定时更新“客户端注册表”解析



```

1 # DiscoveryClient
2 private boolean fetchRegistry(boolean forceFullRegistryFetch) {
3     Stopwatch tracer = FETCH_REGISTRY_TIMER.start();
4
5     try {
6         // If the delta is disabled or if it is the first time, get
all
7         // applications
8         Applications applications = getApplications();
9
10        if (clientConfig.shouldDisableDelta()
11            ||
12            (!Strings.isNullOrEmpty(clientConfig.getRegistryRefreshSingleVipAddress(
13            )))
14            || forceFullRegistryFetch
15            || (applications == null)
16            || (applications.getRegisteredApplications().size()
17            == 0)
18            || (applications.getVersion() == -1)) //Client
19            application does not have latest library supporting delta
20        {
21            logger.info("Disable delta property : {}",
22            clientConfig.shouldDisableDelta());
23            logger.info("Single vip registry refresh property : {}",
24            clientConfig.getRegistryRefreshSingleVipAddress());
25            logger.info("Force full registry fetch : {}",
26            forceFullRegistryFetch);
27            logger.info("Application is null : {}", (applications ==
28            null));
29            logger.info("Registered Applications size is zero : {}",
30            applications.getRegisteredApplications().size());
31        }
32    }
33 }

```

```

22         (applications.getRegisteredApplications().size()
    == 0));
23         logger.info("Application version is -1: {}",
    (applications.getVersion() == -1));
24         // 全量获取注册表
25         getAndStoreFullRegistry();
26     } else {
27         // 增量获取注册表
28         getAndUpdateDelta(applications);
29     }
30
    applications.setAppsHashCode(applications.getReconcileHashCode());
31     logTotalInstances();
32 } catch (Throwable e) {
33     logger.info(PREFIX + "{} - was unable to refresh its cache!
    This periodic background refresh will be retried in {} seconds. status =
    {} stacktrace = {}",
34         appPathIdentifier,
    clientConfig.getRegistryFetchIntervalSeconds(), e.getMessage(),
    ExceptionUtils.getStackTrace(e));
35     return false;
36 } finally {
37     if (tracer != null) {
38         tracer.stop();
39     }
40 }

```

```

1 eureka:
2   client:
3     service-url:
4       defaultZone: http://localhost:8000/eureka
5     # 客户端从服务端下载更新注册表的时间间隔，默认为30s
6     registry-fetch-interval-seconds: 30
7     # 指定client从server更新注册表的最大时间间隔指数（倍数），默认为10
8     cache-refresh-executor-exponential-back-off-bound: 10

```

```

1 # DiscoveryClient
2 private void getAndUpdateDelta(Applications applications) throws
    Throwable {
3     long currentUpdateGeneration = fetchRegistryGeneration.get();
4
5     Applications delta = null;
6     EurekaHttpResponse<Applications> httpResponse =
    eurekaTransport.queryClient.getDelta(remoteRegionsRef.get());
7     if (httpResponse.getStatusCode() == Status.OK.getStatusCode()) {

```

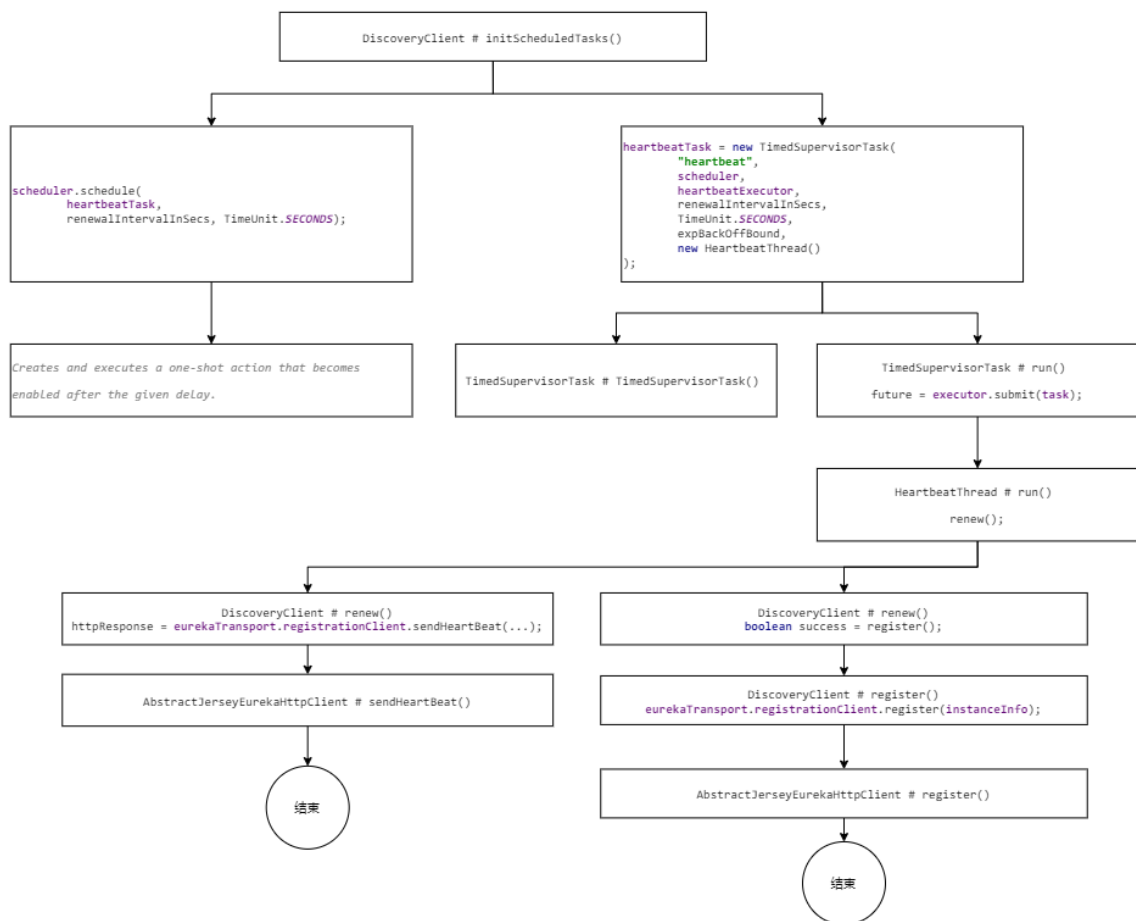
```

8         delta = httpResponse.getEntity();
9     }
10
11     if (delta == null) {
12         logger.warn("The server does not allow the delta revision to
13         be applied because it is not safe. "
14             + "Hence got the full registry.");
15         getAndStoreFullRegistry();
16     } else if
17     (fetchRegistryGeneration.compareAndSet(currentUpdateGeneration,
18     currentUpdateGeneration + 1)) {
19         logger.debug("Got delta update with apps hashCode {}",
20         delta.getAppsHashCode());
21         String reconcileHashCode = "";
22         if (fetchRegistryUpdateLock.tryLock()) {
23             try {
24                 // 这里要将从Server获取到的所有变更信息更新到本地缓存。这些变
25                 // 更信
26                 // 来自于两类Region：本地Region与远程Region。而本地缓存也也
27                 // 分为
28                 // 两类：缓存本地Region的applications与缓存所有远程Region
29                 // 的注
30                 // 册信息的map(key为远程Region, value为该远程Region的注册
31                 // 表)
32                 updateDelta(delta);
33                 reconcileHashCode =
34                 getReconcileHashCode(applications);
35             } finally {
36                 fetchRegistryUpdateLock.unlock();
37             }
38         } else {
39             logger.warn("Cannot acquire update lock, aborting
40             getAndUpdateDelta");
41         }
42         // There is a diff in number of instances for some reason
43         if (!reconcileHashCode.equals(delta.getAppsHashCode()) ||
44         clientConfig.shouldLogDeltaDiff()) {
45             reconcileAndLogDifference(delta, reconcileHashCode); //
46             this makes a remoteCall
47         }
48     } else {
49         logger.warn("Not updating application delta as another
50         thread is updating it already");
51         logger.debug("Ignoring delta update with apps hashCode {},
52         as another thread is updating it already", delta.getAppsHashCode());
53     }
54 }

```

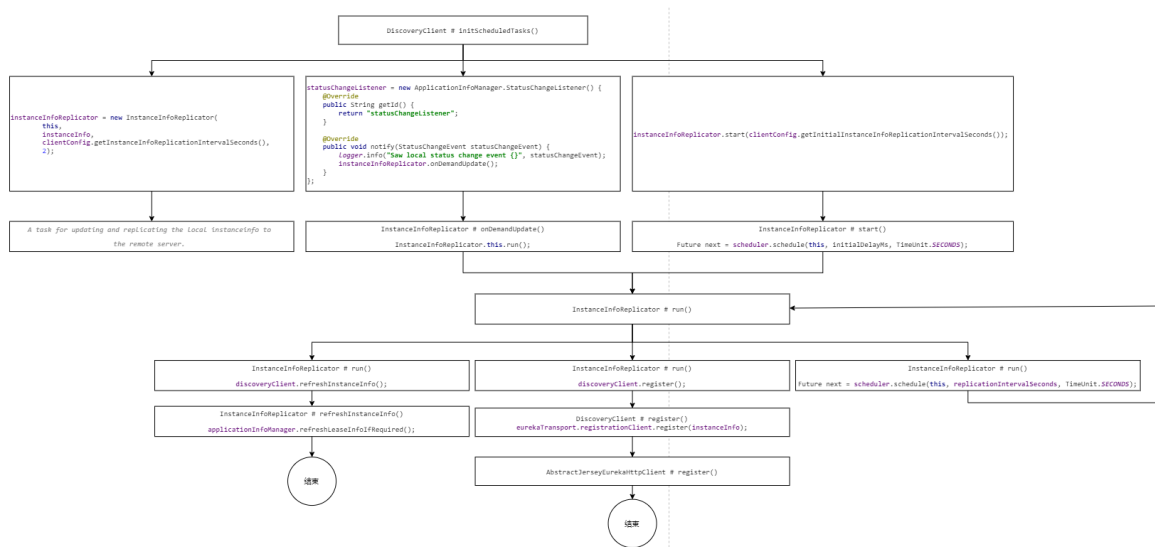


## 定时续约解析



## 定时更新Client信息给Server





```

1 # ApplicationInfoManager
2 public void refreshLeaseInfoIfRequired() {
3     LeaseInfo leaseInfo = instanceInfo.getLeaseInfo();
4     if (leaseInfo == null) {
5         return;
6     }
7     int currentLeaseDuration =
8     config.getLeaseExpirationDurationInSeconds();
9     int currentLeaseRenewal =
10    config.getLeaseRenewalIntervalInSeconds();
11    if (leaseInfo.getDurationInSecs() != currentLeaseDuration ||
12    leaseInfo.getRenewalIntervalInSecs() != currentLeaseRenewal) {
13        // 这是 迭代稳定性 的变化使用
14        LeaseInfo newLeaseInfo = LeaseInfo.Builder.newBuilder()
15        .setRenewalIntervalInSecs(currentLeaseRenewal)
16        .setDurationInSecs(currentLeaseDuration)
17        .build();
18        instanceInfo.setLeaseInfo(newLeaseInfo);
19        instanceInfo.setIsDirty();
20    }
21 }

```

迭代稳定性：一般情况下，对于多线程操作的共享数组/集合，我们在对其元素进行修改操作时，不要直接对该数组/集合进行操作，而是重新创建一个临时的数组/集合，将原数组/集合中的数据复制给临时数组/集合，然后再对这个临时数组/集合执行修改操作。执行完毕后再将临时数组/集合赋值给原数组/集合。这个操作是为了保证迭代稳定性。当然，这里的操作要保证互斥（加锁）。

## 总结

Client提交register()请求的情况有三种：

- 在应用启动时就可以直接进行register(), 不过, 需要提前在配置文件中配置
- 在renew时, 如果server端返回的是NOT\_FOUND, 则提交register()
- 当Client的配置信息发生了变更, 则Client提交register()

## 4 服务离线源码解析

服务离线, 即某服务不能对外提供服务了。服务离线的原因有两种: 服务下架与服务下线。

### 基于Actuator监控器实现

提交如下POST请求, 可实现相应的服务离线操作:

- 服务下架: <http://localhost:8081/actuator/shutdown>, 无需请求体
- 服务下线: <http://localhost:8081/actuator/registry>, 请求体为 (该方法称为服务平滑上下线)

```
1 {  
2   "status": "OUT_OF_SERVICE"  
3 }
```

```
1 {  
2   "status": "UP"  
3 }
```

注意, 从Spring Cloud 2020.0.0版本开始, 服务平滑上下线的监控终端由service-registry变更为serviceregistry

### 直接向EurekaServer提交请求

可以通过直接向Eureka Server提交不同的请求的方式来实现指定服务离线操作。

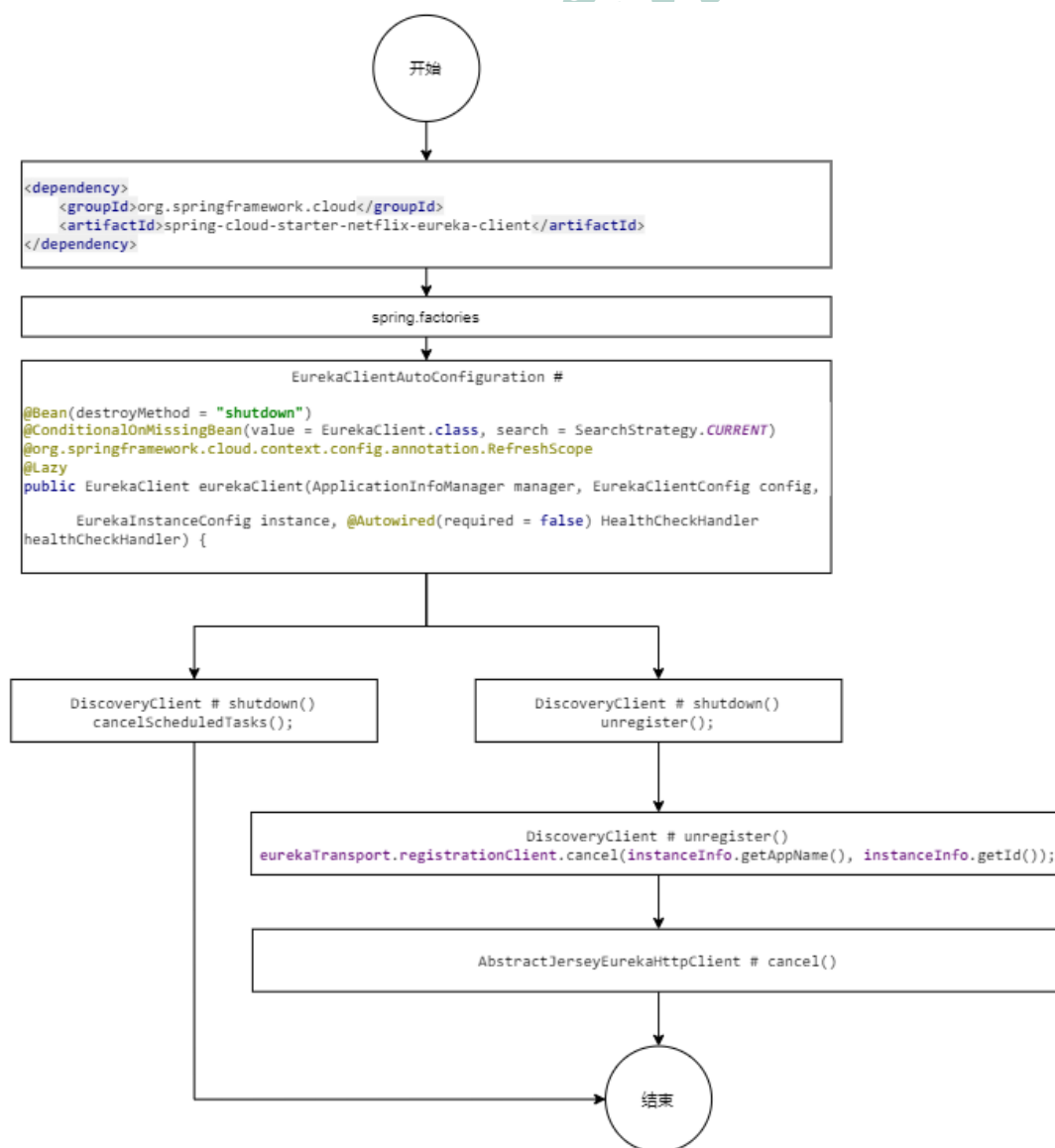
- 服务下架: 通过向eureka server发送DELETE请求来删除指定client的服务  
[http://\\${server}:\\${port}/eureka/apps/\\${serviceName}/\\${instanceId}](http://${server}:${port}/eureka/apps/${serviceName}/${instanceId})
- 服务下线: 通过向eureka server发送PUT请求来修改指定client的status, 其中\${value}的取值为: OUT\_OF\_SERVICE或UP  
[http://\\${server}:\\${port}/eureka/apps/\\${serviceName}/\\${instanceId}/status?value=\\${value}](http://${server}:${port}/eureka/apps/${serviceName}/${instanceId}/status?value=${value})

### 特殊状态CANCEL\_OVERRIDE

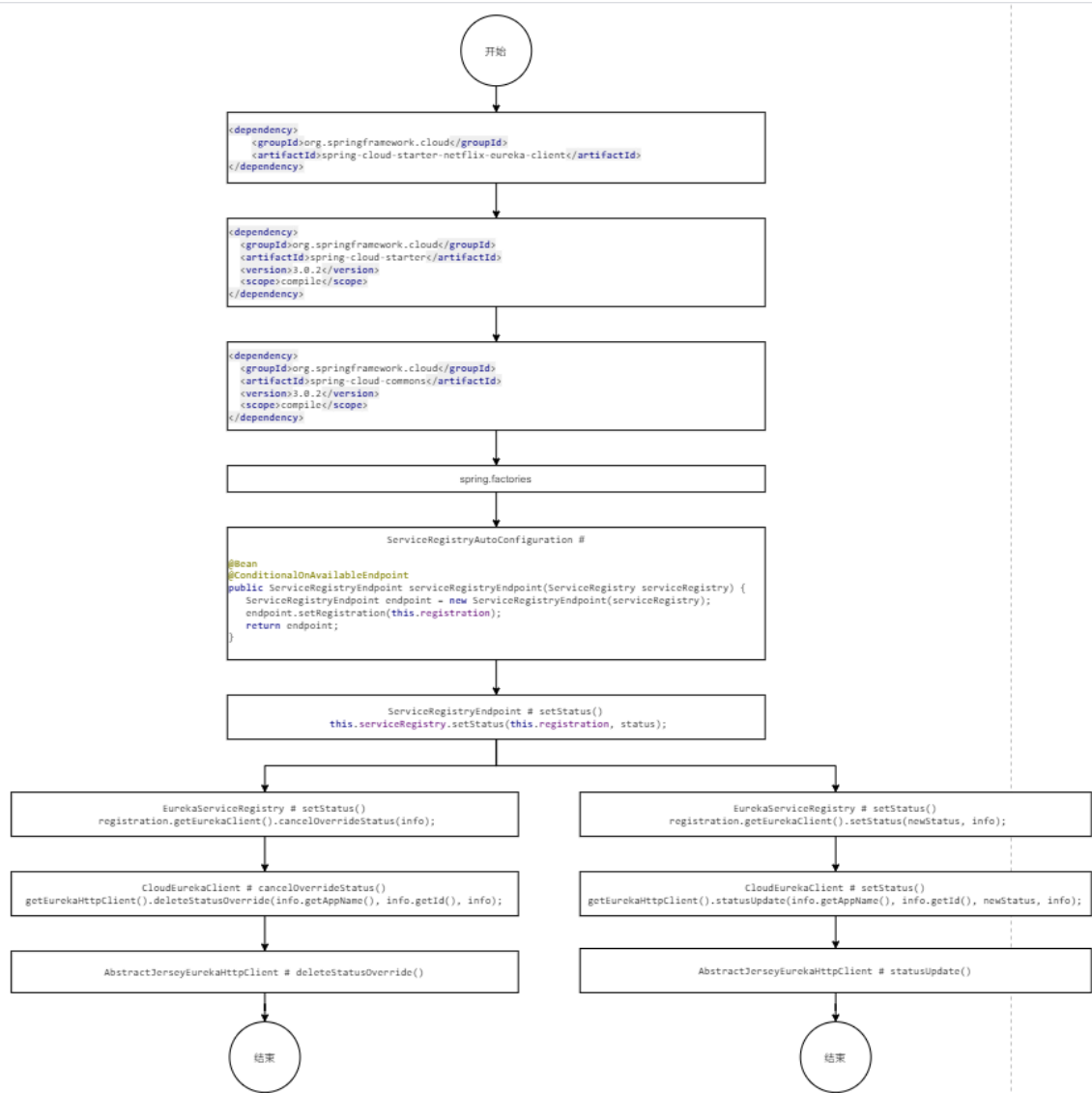
用户提交的状态修改请求中指定的状态, 除了InstanceInfo的内置枚举类InstanceStatus中定义的状态外, 还可以是CANCEL\_OVERRIDE状态。

若用户提交的状态为CANCEL\_OVERRIDE，则Client会通过Jersey向Server提交一个DELETE请求，用于在Server端将对应InstanceInfo的overridenStatus修改为UNKNWON，即删除了原来的overridenStatus的状态值。此时，该Client发送的心跳Server是不接收的。Server会向该Client返回404。

## 服务下架源码解析



## 服务下线源码解析



## 五、Eureka Server源码解析

### 1 Eureka Sever解析入口分析

#### 关于Marker实例

通过Eureka Server的依赖我们从spring.factories中找到EurekaServer的自动配置类。

```

@Configuration(proxyBeanMethods = false)
@Import(EurekaServerInitializerConfiguration.class)
@ConditionalOnBean(EurekaServerMarkerConfiguration.Marker.class)
@EnableConfigurationProperties({ EurekaDashboardProperties.class, Instance
@PropertySource("classpath:/eureka/server.properties")
public class EurekaServerAutoConfiguration implements WebMvcConfigurer {

```



该配置类具有一个条件注解：要求必须要有一个EurekaServerMarkerConfiguration.Marker实例这个配置类才会起作用。那么这个实例是在哪里创建的呢？

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServer {

    public static void main(String[] args) {
        SpringApplication.run(EurekaServer.class, args);
    }
}
```

打开Eureka Server的启动类，其必须要添加@EnableEurekaServer注解，为什么呢？

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(EurekaServerMarkerConfiguration.class)
public @interface EnableEurekaServer {

}
```

打开这个注解，其是一个复合注解，该注解中导入了一个类EurekaServerMarkerConfiguration。打开这个类，可以看到该类是一个配置类，在其中创建了Marker实例。

```

@Configuration(proxyBeanMethods = false)
public class EurekaServerMarkerConfiguration {

    @Bean
    public Marker eurekaServerMarkerBean() {
        return new Marker();
    }

    class Marker {

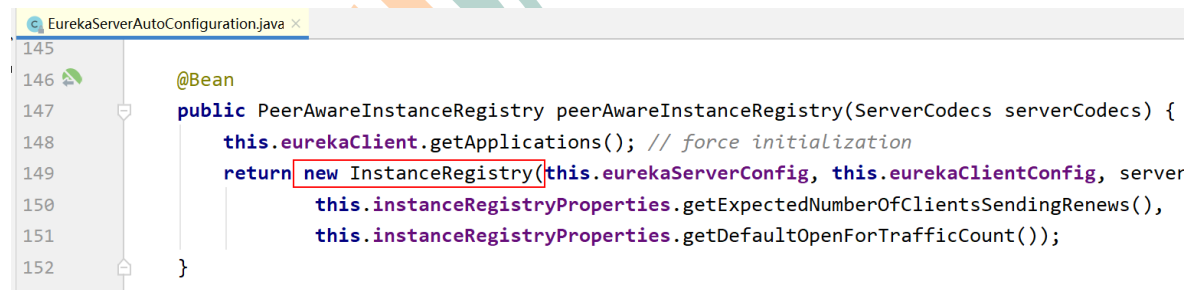
    }

}

```

这就是为什么在EurekaServer启动类上必须添加@EnableEurekaServer注解的原因。

## 重要实例的创建



```

145
146
147 @Bean
148 public PeerAwareInstanceRegistry peerAwareInstanceRegistry(ServerCodecs serverCodecs) {
149     this.eurekaClient.getApplications(); // force initialization
150     return new InstanceRegistry(this.eurekaServerConfig, this.eurekaClientConfig, server
151         this.instanceRegistryProperties.getExpectedNumberOfClientsSendingRenews(),
152         this.instanceRegistryProperties.getDefaultOpenForTrafficCount());
153 }

```

在这个Eureka Server自动配置类启动后，其会创建一个很重要的实例，InstanceRegistry。该实例及其父类中的很多方法是我们后面要调用到的。

## 2 处理Client状态修改请求

Server完成的任务：

- 修改了注册表中该instanceInfo的status
- 将新的状态记录到了overriddenInstanceStatusMap缓存中
- 将本次修改记录到了recentlyChangedQueue缓存中



## 5 处理客户端下架请求

 1621648392882

### 对比处理删除overridden状态请求与下架请求

处理删除overridden状态请求完成的主要任务：

- 从缓存map中删除指定client的overriddenStatus
- 修改注册表中该client的overriddenStatus为UNKNOWN
- 修改注册表中该client的status为UNKNOWN
- 将本次操作记录到recentlyChangedQueue
- 修改注册表中该client的lastUpdatedTimestamp

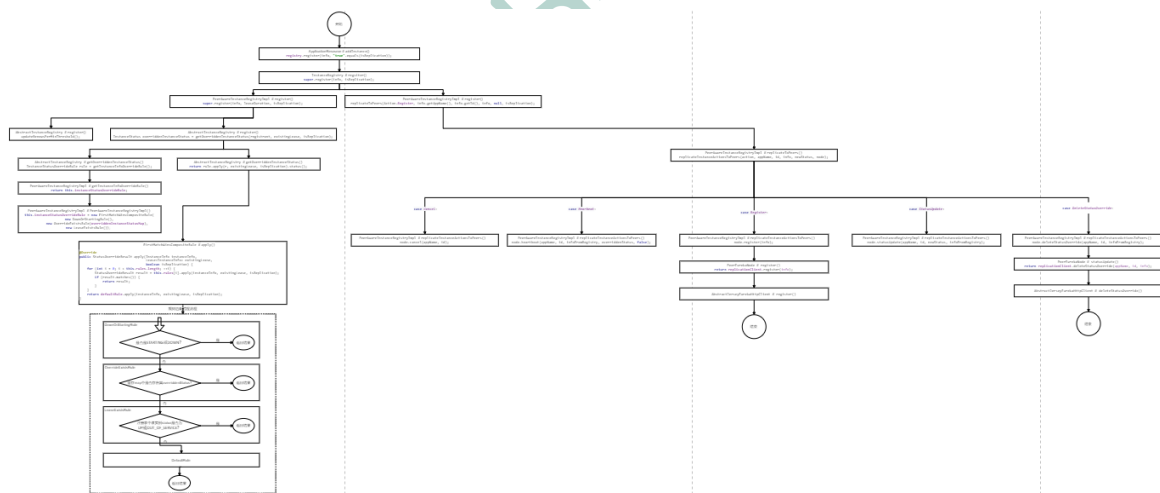
处理下架请求完成的主要任务：

- 将该client从注册表中删除
- 从缓存map中删除指定client的overriddenStatus
- 将本次操作记录到recentlyChangedQueue
- 修改注册表中该client的lastUpdatedTimestamp

## 6 处理客户端注册请求

Server完成的任务：

- 将最新的Client写入到注册表



```

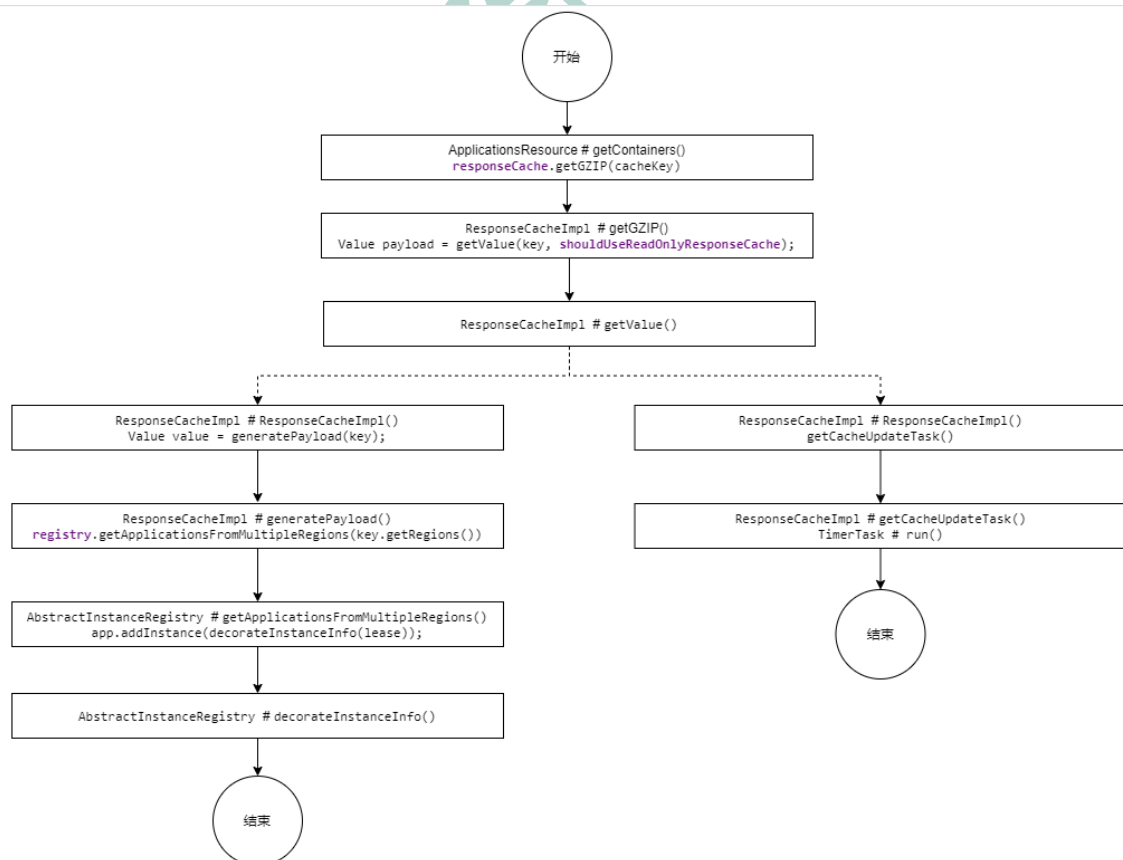
1  this.numberOfRenewsPerMinThreshold = (int)
   (expectedNumberOfClientsSendingRenews
2
   * (60.0 / serverConfig.getExpectedClientRenewalIntervalSeconds())
3
   * serverConfig.getRenewalPercentThreshold())
4  = (客户端数量 * (60 / 心跳间隔) * 自我保护开启的阈值因子)
5  = (客户端数量 * 每个客户端每分钟发送心跳的数量 * 阈值因子)
6  = (所有客户端每分钟发送的心跳数量 * 阈值因子)
7  = 当前Server开启自我保护机制的每分钟最小心跳数量

```

一旦自我保护机制开启了，那么就将当前Server **保护** 了起来，即当前server注册表中的所有client均不会过期，即当client没有指定时间内（默认90秒）发送续约，也不会将其从注册表中删除。为什么？就是为了保证server的 **可用性**，即保证 **AP**。

expectedNumberOfClientsSendingRenews设置的越大，**当前Server开启自我保护机制的每分钟最小心跳数量** 就越大，就越容易发生自我保护。

## 7 处理客户端全量下载请求



### 问题1

readWriteCacheMap中的数据从哪里来？

答：在ResponseCacheImpl构造器中创建并初始化了这个读写缓存map。

## 问题2

readOnlyCacheMap的数据来自于readWriteCacheMap，但readWriteCacheMap中的数据若发生了变更，那么readOnlyCacheMap中的数据一定也需要发生变化，那么readOnlyCacheMap中的数据在最里发生的变更？

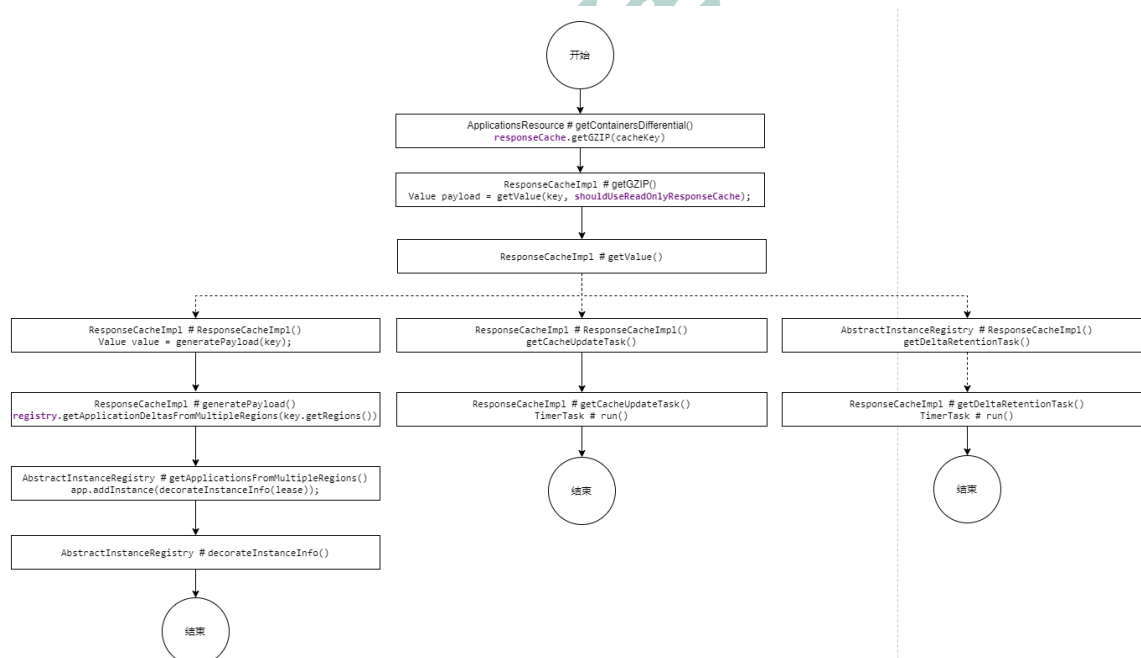
答：在ResponseCacheImpl构造器中定义并开启了一个定时任务，用于定时从readWriteCacheMap中更新readOnlyCacheMap中的数据。

## 问题3

为什么不直接从readWriteCacheMap中获取数据，而是从readOnlyCacheMap获取？即这种方案的好处是什么？

答：为了保证对readWriteCacheMap的 **迭代稳定性**。即将读写进行了分离，分离到了两个共享集合。但这种解决方案存在一个很严重的弊端：读、写两个集合的数据无法保证 **强一致性**，即只能做到 **最终一致性**。所以这种方案的应用场景是，对数据的实时性要求不是很高，对数据是否是最新数据要求不高。

## 8 处理客户端增量下载请求



## 9 定时清除过期Client



## 问题的发现

(1) 读/写锁是反着加的，为什么？

方法名	操作的共享集合	读/写操作	添加的锁
register()	registry、recentlyChangedQueue	写	读
statusUpdate()	registry、recentlyChangedQueue	写	读
internalCancel()	registry、recentlyChangedQueue	写	读
deleteStatusOverride()	registry、recentlyChangedQueue	写	读
renew()	registry	写	无
全量下载方法	registry	读	无
增量下载方法	recentlyChangedQueue	读	写

(2) 同样都是写操作，为什么处理续约请求的方法中却没有加锁？

(3) 同样都是读操作，为什么全量下载方法中却没有加锁？

## 加锁方法的特征

所有对recentlyChangedQueue共享集合操作的方法都添加了锁。而没有对其进行操作的方法，没有加锁。

## 为什么写操作要添加读锁？

若对这些写操作添加写锁，是否可以呢？写锁是排它锁。若要为这些对recentlyChangedQueue进行的写操作添加写锁，则意味着当有一个写操作发生时，对recentlyChangedQueue的所有其它读/写操作，均会发生排队等待（阻塞），会导致效率低下。而

而若要添加读锁，则会使所有对recentlyChangedQueue执行的写操作实现并行，提高了执行效率。不过，这些写操作会引发线程安全问题吗？不会。因为recentlyChangedQueue是JUC的队列，是线程安全的。

需要注意，虽然我们的关注点一直都在recentlyChangedQueue上，但从代码角度来说，也为registry的写操作添加了读锁。不会影响registry的并行效率吗？不会。因为读锁是共享锁。

## 为什么读操作添加写锁？

为了保证对共享集合recentlyChangedQueue的读/写操作的互斥。不过，该方式会导致读操作效率的低下，因为读操作无法实现并行读取，只能排队读取。因为写锁是排它的。

## 读写锁反加应用场景

写操作相对于读操作更加频繁的场景。

## 续约操作能否添加写锁

不能。因为续约操作是一个发生频率非常高的写操作，若为其添加了写锁，则意味着其它client的续约处理无法实现并行，发生排队等待。因为写锁是排它锁。

## 续约操作能否添加读锁

不能。因为添加读锁的目的是为了与写锁操作实现互斥。在上述所有方法中，对registry的所有操作中均没有添加写锁，所以这里的写操作也无需添加读锁。

## 如果不加锁会怎样



若不对recentlyChangedQueue的操作加锁，可能会存在同时对recentlyChangedQueue进行读写操作的情况。可能会引发对recentlyChangedQueue的迭代稳定性问题。

## 为什么全量下载没有添加写锁

若为其添加了写锁，则必须会导致某client在读取期间，其它client的续约请求处理被阻塞的情况。

## 六、Nacos Client源码解析

### 1 预备知识

#### Nacos版本选择

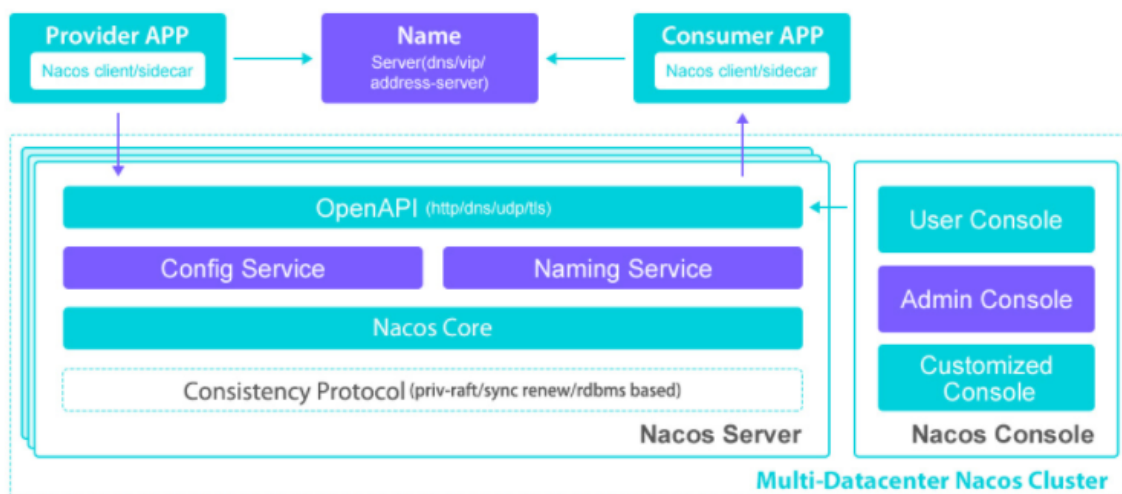
使用spring cloud alibaba时特别需要注意版本间的兼容关系，这些关系包括spring cloud alibaba、spring cloud与spring boot间的版本兼容关系，包括spring cloud alibaba与使用的alibaba中间件版本间的兼容关系。这些关系说明在spring cloud alibaba的github官网wiki首页的“版本说明”中有详细说明。所以，我们对于要下载的Nacos源码的版本，直接与SCA的版本是相关的。

我们这里选择spring cloud alibaba 2.2.5RELEASE版本，所以就应该选择Nacos1.3.3版本。

#### 导入Nacos源码工程

将Nacos源码下载并解压后直接在Idea中导入Maven工程即可。

#### Nacos系统架构



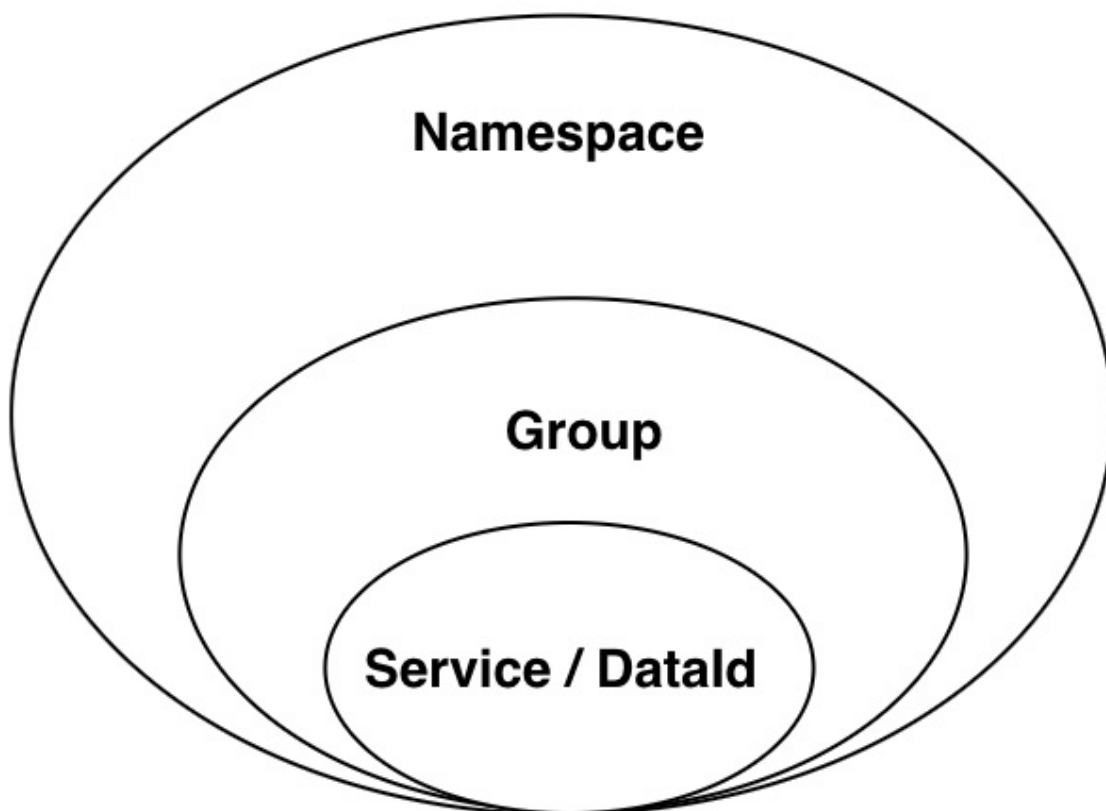
sidecar，是从SCA2.1.1版本后增加的模块，是一种多语言异构模块，用于实现其它语言使用SCA相关组件的模块。

## 数据模型



Name Server用于记录各个命名空间namespace中的实例信息。namespace、group与服务service或资源dataId间的关系如下图。

## Nacos data model



DEFAULT\_GROUP%40%40abcm-sc-provider-depart@@DEFAULT

即 DEFAULT\_GROUP@@abcm-sc-provider-depart@@DEFAULT

即 groupId@@微服务名称@@clusterName

注意，图中的Service不是一个简单的服务提供者，而是很多提供者实例的集合。而这个集合中的提供者又可以分属于很多的Cluster。

 1622463720135

## 临时实例与持久实例

### 配置

在服务注册时有一个属性ephemeral用于描述当前实例在注册时是否以临时实例出现。为true则为临时实例，默认值；为false则为持久实例。

```
spring:
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848
        ephemeral: true
```

## 区别

临时实例与持久实例的实例的存储位置与健康检测机制是不同的。

- **临时实例**：默认情况。服务实例仅会注册在Nacos内存，不会持久化到Nacos磁盘。其健康检测机制为Client模式，即Client主动向Server上报其健康状态（类似于推模式）。默认心跳间隔为5秒。在15秒内Server未收到Client心跳，则会将其标记为“不健康”状态；在30秒内若收到了Client心跳，则重新恢复“健康”状态，否则该实例将从Server端内存清除。即对于不健康的实例，Server会自动清除。
- **持久实例**：服务实例不仅会注册到Nacos内存，同时也会被持久化到Nacos磁盘。其健康检测机制为Server模式，即Server会主动去检测Client的健康状态（类似于拉模式），默认每20秒检测一次。健康检测失败后服务实例会被标记为“不健康”状态，但不会被清除，因为其是持久化在磁盘的。其对不健康持久实例的清除，需要专门进行。

## 应用场景

临时实例适合于 **存在突发流量暴增可能** 的互联网项目。因为临时实例可以实现 **弹性扩容**。

## 重要API

### Instance类

实例，代表一个Nacos Client主机实例。

### ServiceInfo类

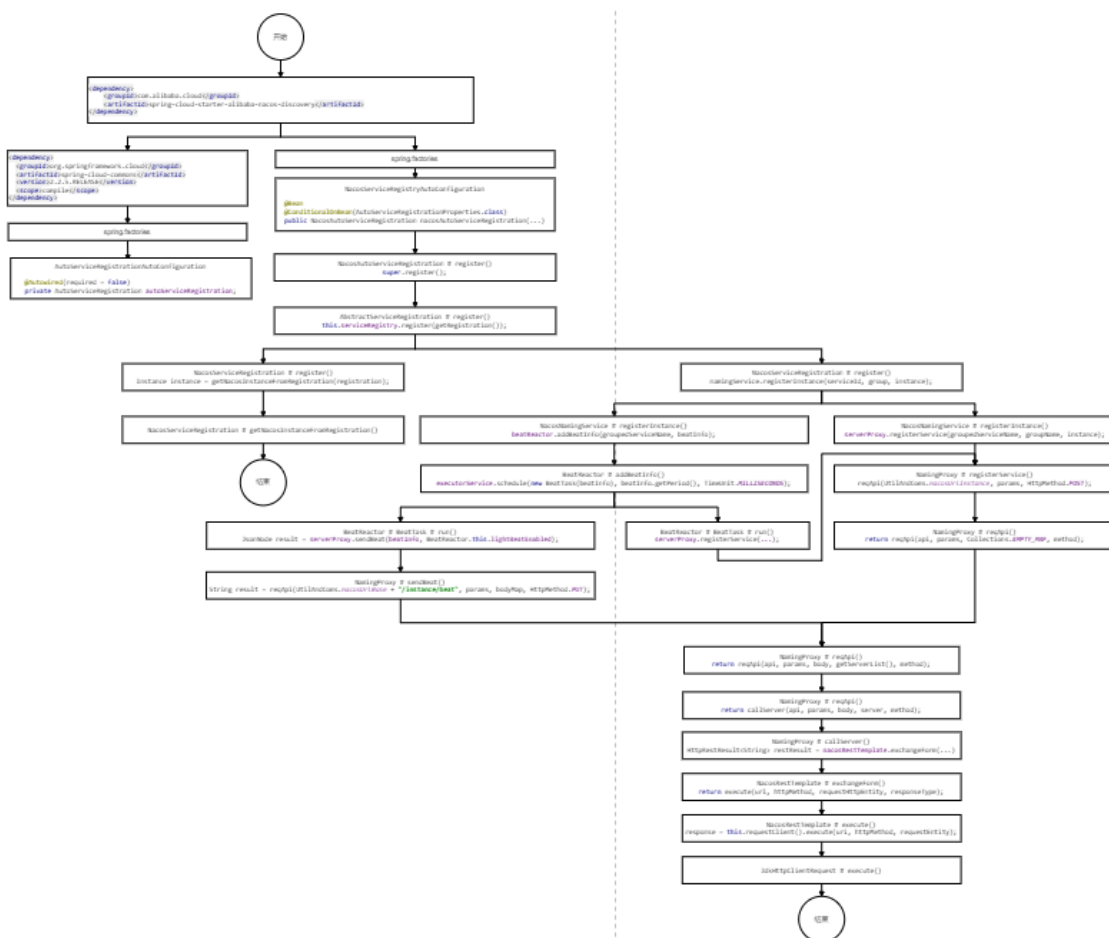
微服务信息实例。其包含着一个Instance列表。

### NamingService接口

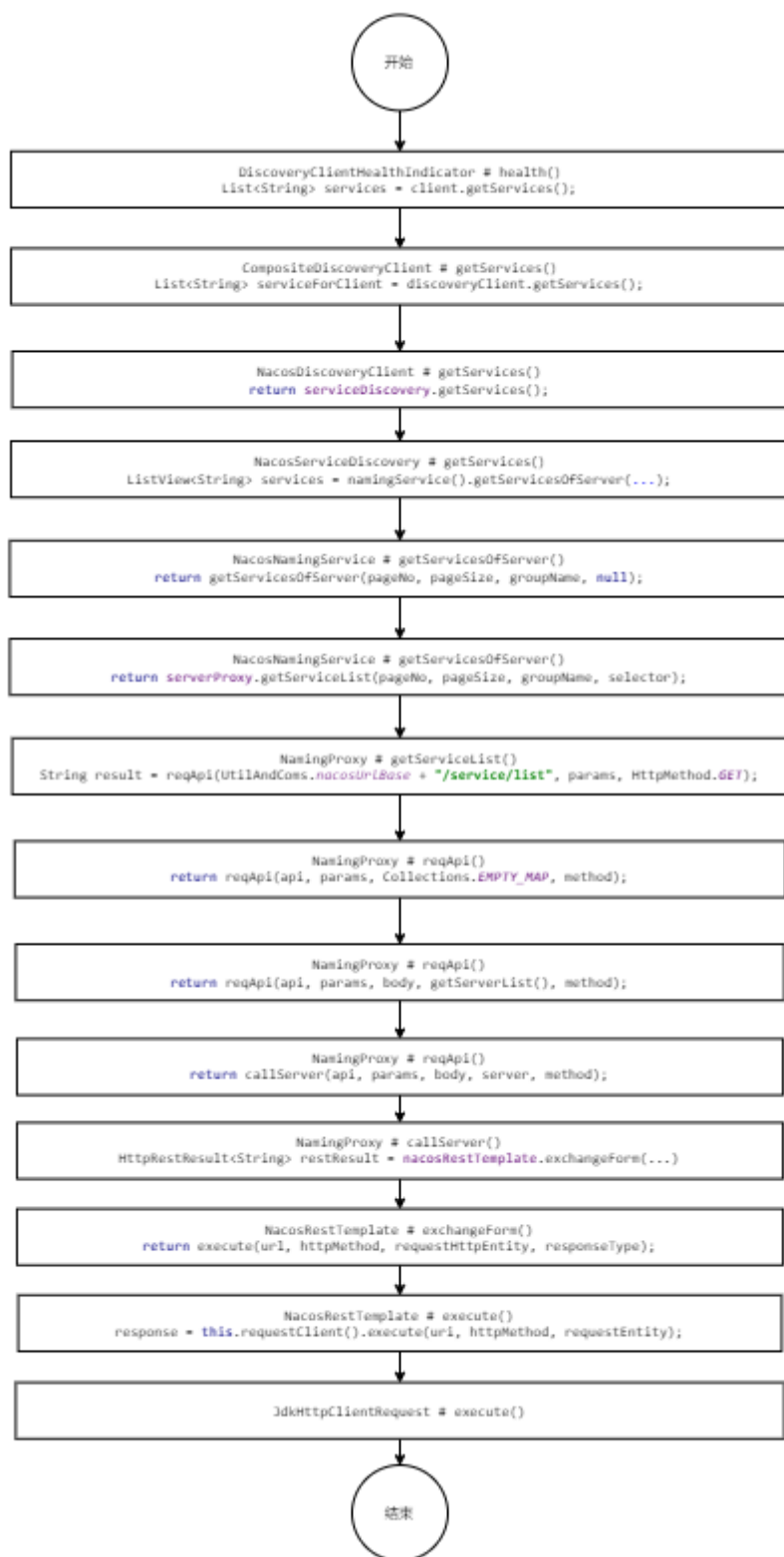
该接口只有一个实现类，NacosNamingService。通过这个类的实例，可以完成Client与Server间的通信，例如注册/取消注册，订阅/取消订阅，获取Server状态，获取Server中指定的Instance。

注意，心跳不是通过这个类实例完成的。

## 2 Nacos Client的注册与心跳

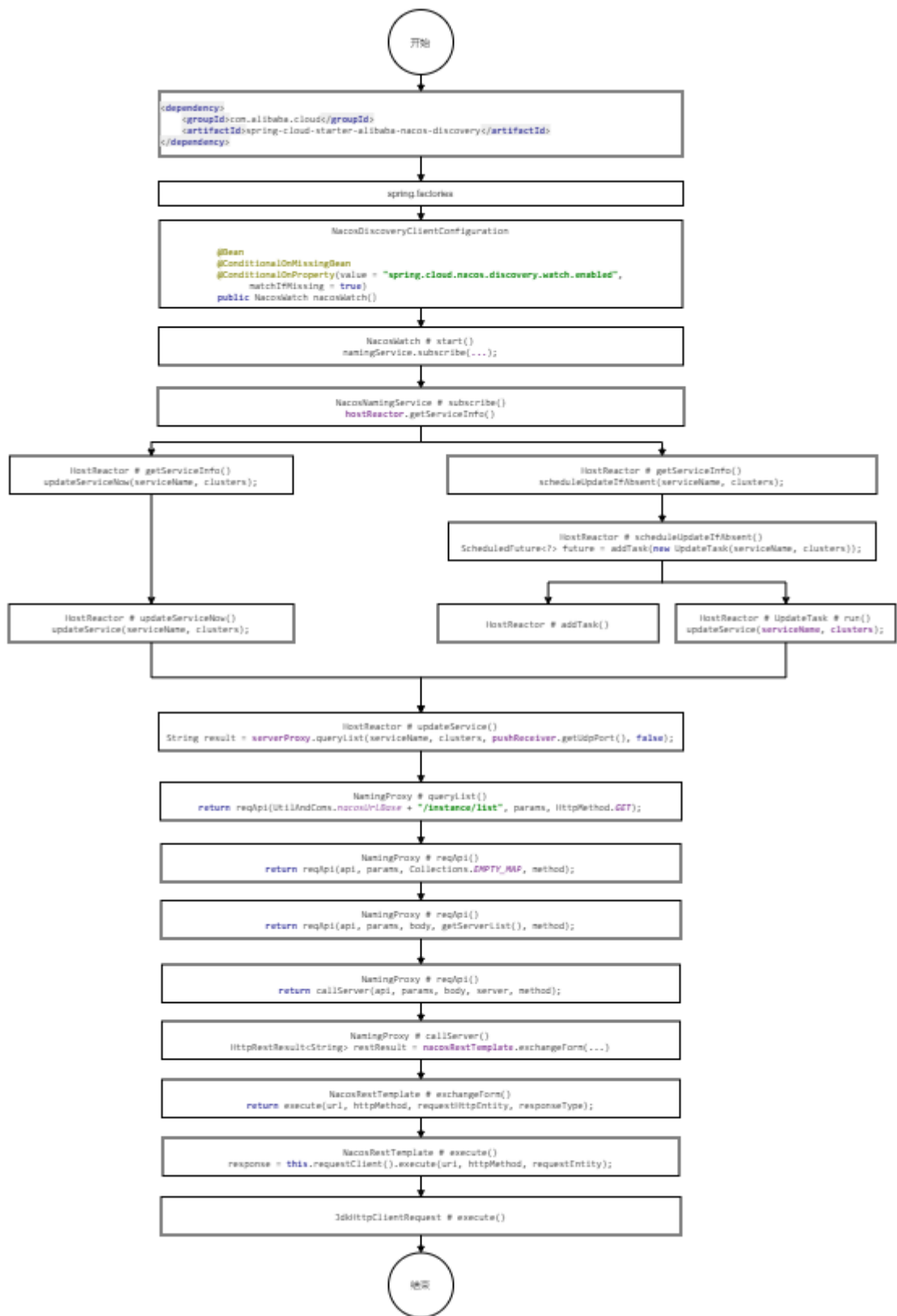


## 3 Nacos Client获取所有服务



#### 4 Nacos Client定时更新本地服务

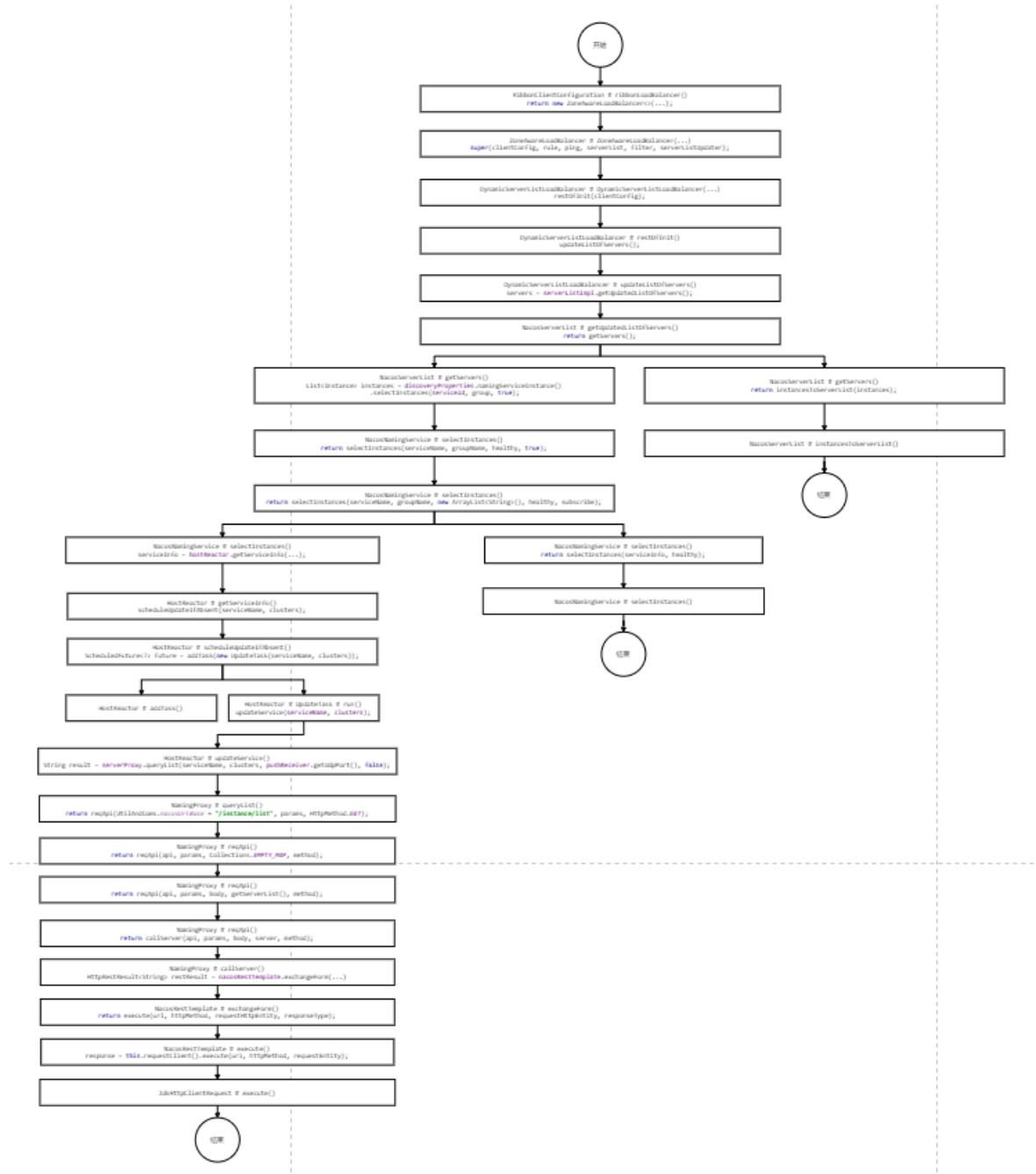




Zookeeper中的重要知识点:

- 一致性算法Paxos
- ZAB协议
- Leader选举算法
- watcher机制

## 5 Client获取要调用服务的提供者列表



## 七、Nacos Server源码解析

### 启动Nacos源码工程

### Nacos单机启动

找到console模块下的Nacos类直接运行，无需做任何的配置。不过，此时无法启动成功，因为其默认以集群方式启动。所以可以在VM options中添加 `-Dnacos.standalone=true` 动态参数，再运行就没有问题了。

## Nacos集群启动

由于使用内嵌Storage无法启动集群，所以若要以集群方式启动Nacos，首先需要修改console模块下的application.properties。将其中的连接数据库URL中的数据库Server地址及要连接的数据库进行替换，并修改数据库连接的用户名与密码。然后在VM options中添加类似 `-Dserver.port=8849` 的动态参数，指定当前启动Nacos的端口号。

同理，设置不同的端口号，启动多台的Nacos Server。

## 重要API

### InstanceController类

该类为一个处理器，用于处理服务实例的心跳、注册等请求。

### core/Service类

在Nacos客户端的一个微服务名称定义的服务，在Nacos服务端是以Service实例的形式出现的。其类似于ServiceInfo，只不过ServiceInfo是客户端服务，而core/Service是服务端服务。

Service类中有一个属性protectThreshold，保护阈值。

### 与Eureka中的保护阈值对比：

- 1) 相同点：都是一个0-1的数值，表示健康实例占有所有实例的比例
- 2) 保护方式不同：
  - Eureka：一旦健康实例数量小于阈值，则不再从注册表中清除不健康的实例
  - Nacos：如果健康实例数量大于阈值，则消费者调用到的都是健康实例。一旦健康实例数量小于阈值，则消费者会从所有实例中进行选择调用，有可能会调用到不健康实例。这样可以保护健康的实例不会被压崩溃。
- 3) 范围不同：
  - Eureka：这个阈值针对的是所有服务中的实例
  - Nacos：这个阈值针对的是当前Service中的服务实例

### RecordListener接口

Service类实现了RecordListener接口。这个接口是一个数据监听的接口。即Service类本身还是一个监听器，用于监听指定数据的变更或删除。

### Record接口



RecordListener接口的泛型为指定了该监听器所要监听的实体类型。这个类型是一个Record接口的子接口。Record是一个在Nacos集群中传输和存储的记录。

### **Cluster类**

提供某一服务的Instance集群，即隶属于某一Service的Instance集群。

### **Instance类**

注册到Nacos中的具体服务实例。

### **ServiceManager类**

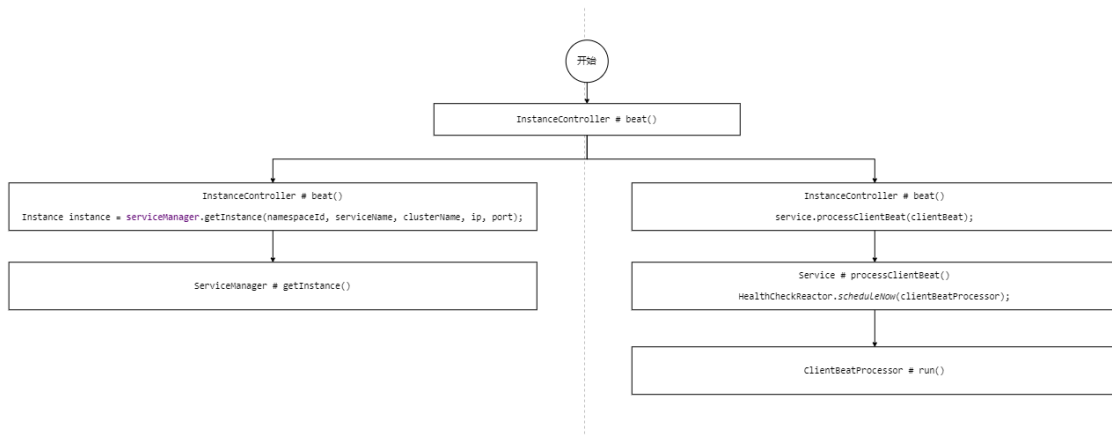
Nacos中所有service的核心管理者。其中一个很重要的属性是serviceMap，就是Nacos中的服务注册表。该接口中有很多的方法，这些方法可以完成在nacos集群中相关操作的同步。

### **Synchronizer接口**

同步器，是当前Nacos主动发起的同步操作。其包含两个方法，分别表示当前Nacos主动发送自己的Message给指定的Nacos；主动从指定Nacos中获取指定key的Message。

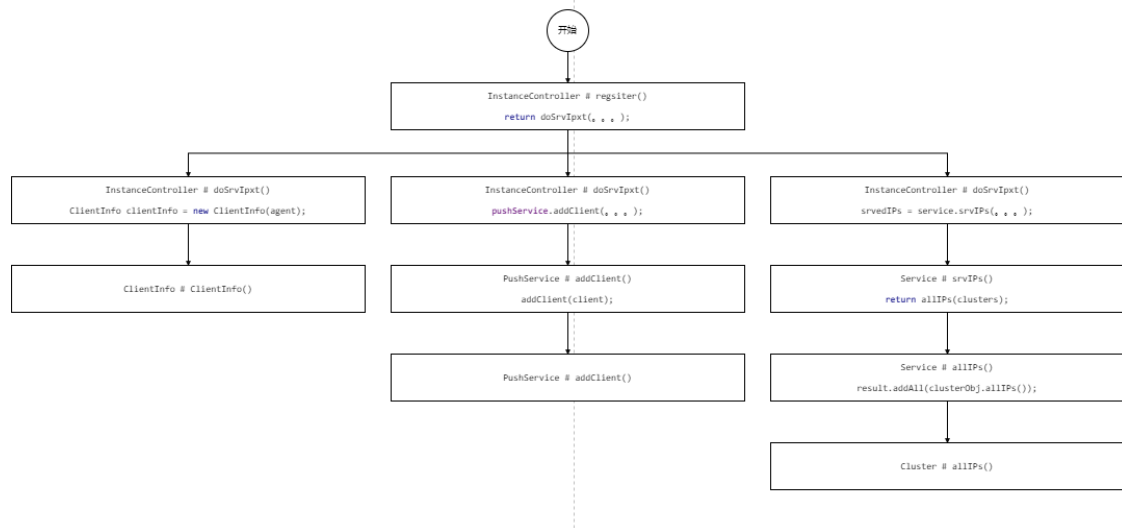
## **2 Server处理Client注册请求**





该处理方式主要就是在注册表中查找这个instance，若没有找到，则创建一个再注册到注册表；若找到了，则更新其最后心跳时间戳。其中比较重要的一项工作是，若这个instance的健康状态发生了变更，其会发布一个服务变更事件，以触发其它该服务的订阅者更新服务。

## 5 Server处理订阅请求

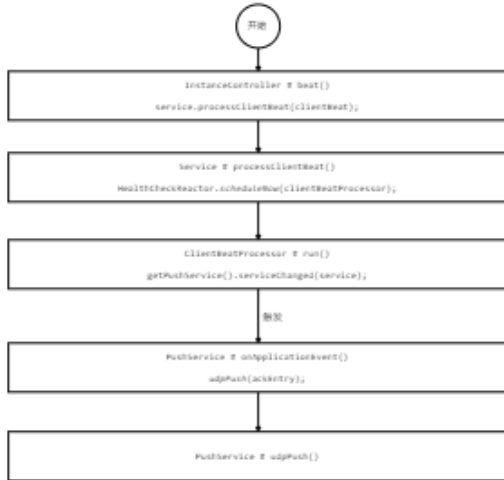


该处理方式主要完成了两项重要任务：

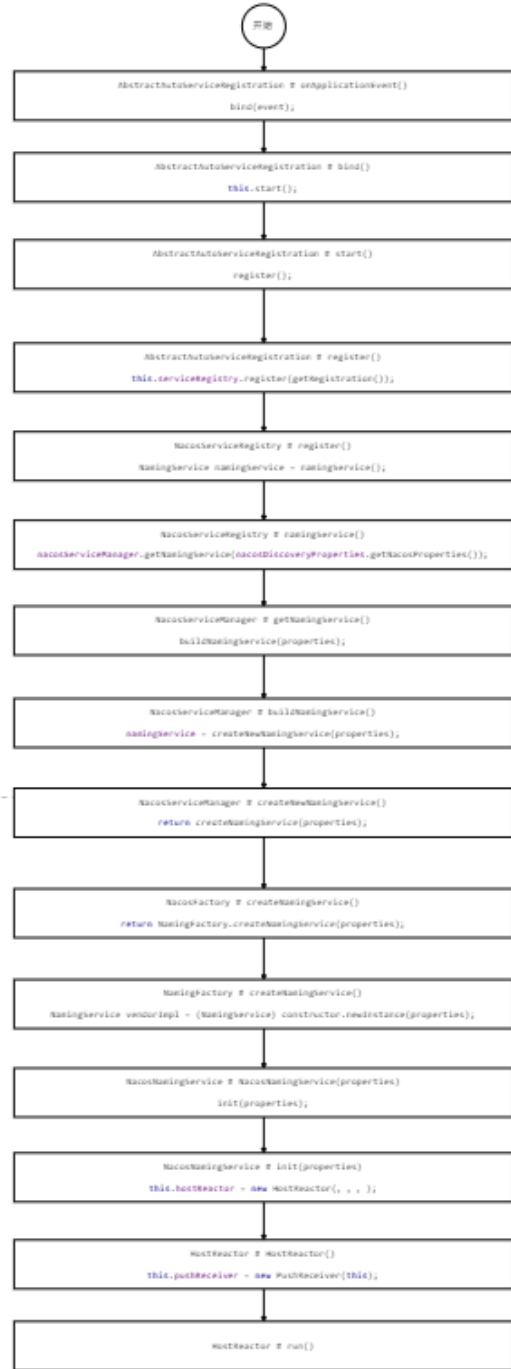
- 创建了该Nacos Client对应的UDP通信客户端PushClient，并将其写入到了一个缓存map
- 从注册表中获取到指定服务的所有 **可用的** instance，并将其封装为JSON

## 6 Server/Client间的UDP通信

Nacos Server向Nacos Client发送UDP数据流程



Nacos Client接收并响应Nacos Server的UDP数据流程



## 7 Server间的操作

宿

- 启动了一个定时任务：每60s当前Server会向其它Nacos Server发送一次本机注册表
- 从其它Nacos Server获取到注册表中的所有instance的最新状态并更新到本地注册表
- 启动了一个定时任务：每20s清理一次注册表中的空service



