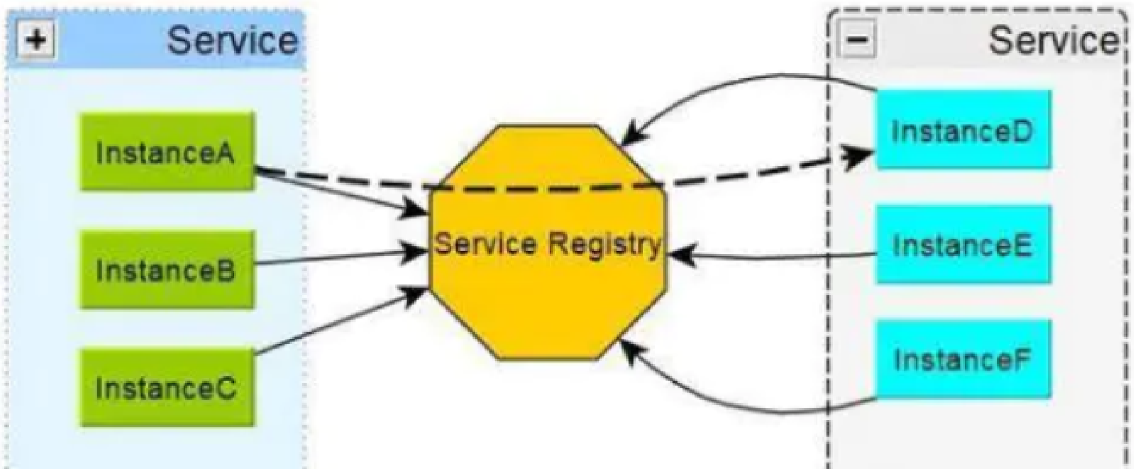
http://note.youdao.com/noteshare?id=fb80603f2b5053b4afa870da0c311f60&sub=DFF6A058FA9B403A844146B906F1AF2F

1. 背景

1.1. 注册中心是什么

注册中心可以说是微服务架构中的”通讯录 “，它记录了服务和服务地址的映射关系。在分布式架构中，服务会注册到这里，当服务需要调用其它服务时，就到这里找到服务的地址，进行调用。



1.2. 为什么需要注册中心

在分布式系统中，服务可能有上千个，然后每个服务都有好几个实例，如果通过 ip + port 进行服务之间通信则会使系统变得难维护，并且还需要考虑其他复杂的问题：

1. 服务注册后，如何被及时发现
2. 服务宕机后，如何及时下线
3. 服务如何有效的水平扩展
4. 如何获取服务列表
5. 注册中心如何实现自身的高可用

2. Eureka

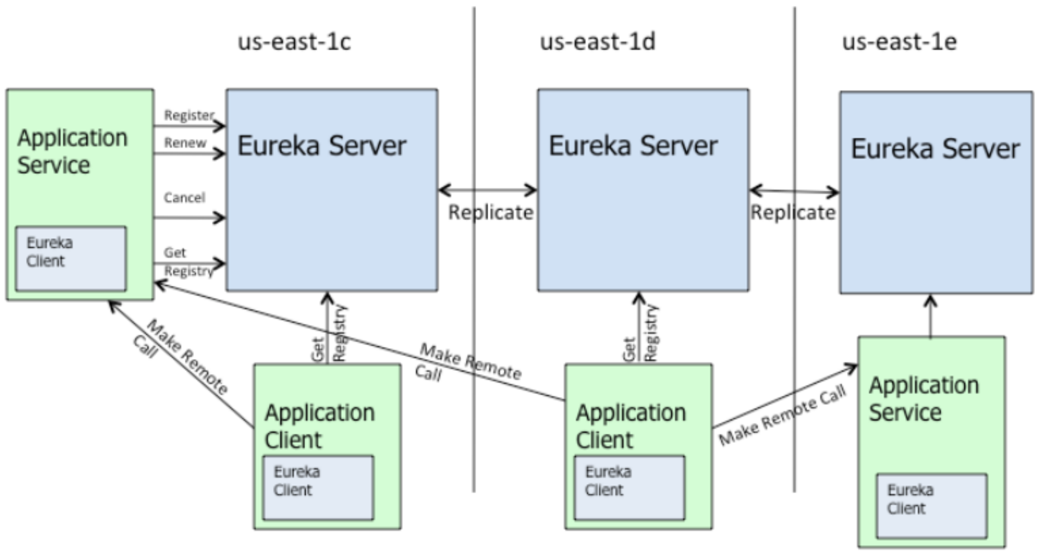
2.1. 世面上的流行的注册中心

组件名称	组件简介

Zookeeper	zookeeper是一个分布式协调工具，可以实现注册中心功能
Eureka	springcloud的注册中心
Consul	Consul 简化了分布式环境中的服务的注册和发现流程，国外比较流行
Nacos	Nacos 致力于帮助您发现、配置和管理微服务。SpringCloudAlibaba

2.2. Eureka

https://github.com/Netflix/eureka



服务注册中心(可以是一个集群)，对外暴露自己的地址

注册中心有 Eureka Service, Eureka Client,Eureka Client又分为提供者和消费者；

(某一个服务既可以是提供者也可以是消费者)

服务提供者

- **服务注册**： 启动的时候会通过发送REST请求的方式将自己注册到Eureka Server上，同时带上了自身服务的一些元数据信息。
- **服务续约**： 在注册完服务之后，服务提供者会维护一个心跳(默认30S) 用来持续告诉Eureka Server: "我还活着 ”
- **服务下线**： 当服务实例进行正常的关闭操作时，它会触发一个服务下线的REST请求 给Eureka Server, 告诉服务注册中心： “我要下线了 ” 。

服务消费者

- **获取服务**： 服务消费者（Eureka Client）在启动的时候，会发送一个REST请求给Eureka Server，获 取上面注册的服务清单，并且缓存在Eureka Client本地，默认缓存30秒

(eureka.client.registryFetchIntervalSeconds)。同时，为了性能考虑，Eureka Server也会维护一份只读的服务清单缓存，该缓存每隔30秒更新一次。

- **服务调用**： 服务消费者在获取服务清单后，通过服务名可以获得具体提供服务的实例名和该实例的元数据信息。在进行服务调用的时候，优先访问同处一个Zone中的服务提供方。

Eureka Server(服务注册中心)

- **失效剔除：【在关闭自我保护才有效】** 默认每隔一段时间（默认为60秒） 将当前清单中超时（默认为90秒）没有续约的服务剔除出去。

- **自我保护**： EurekaServer 在运行期间，如果在15分钟内超过85%的客户端节点都没有正常的心跳(通常由于网络不稳定导致)。 Eureka Server会将当前的实例注册信息保护起来， 让这些实例不会过期，尽可能保护这些注册信息。此时会出现以下几种情况：

  - Eureka Server不再从注册列表中移除因为长时间没收到心跳而应该过期的服务。
  - Eureka Server仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上，保证当前节点依然可用。
  - 当网络稳定时，当前Eureka Server新的注册信息会被同步到其它节点中。

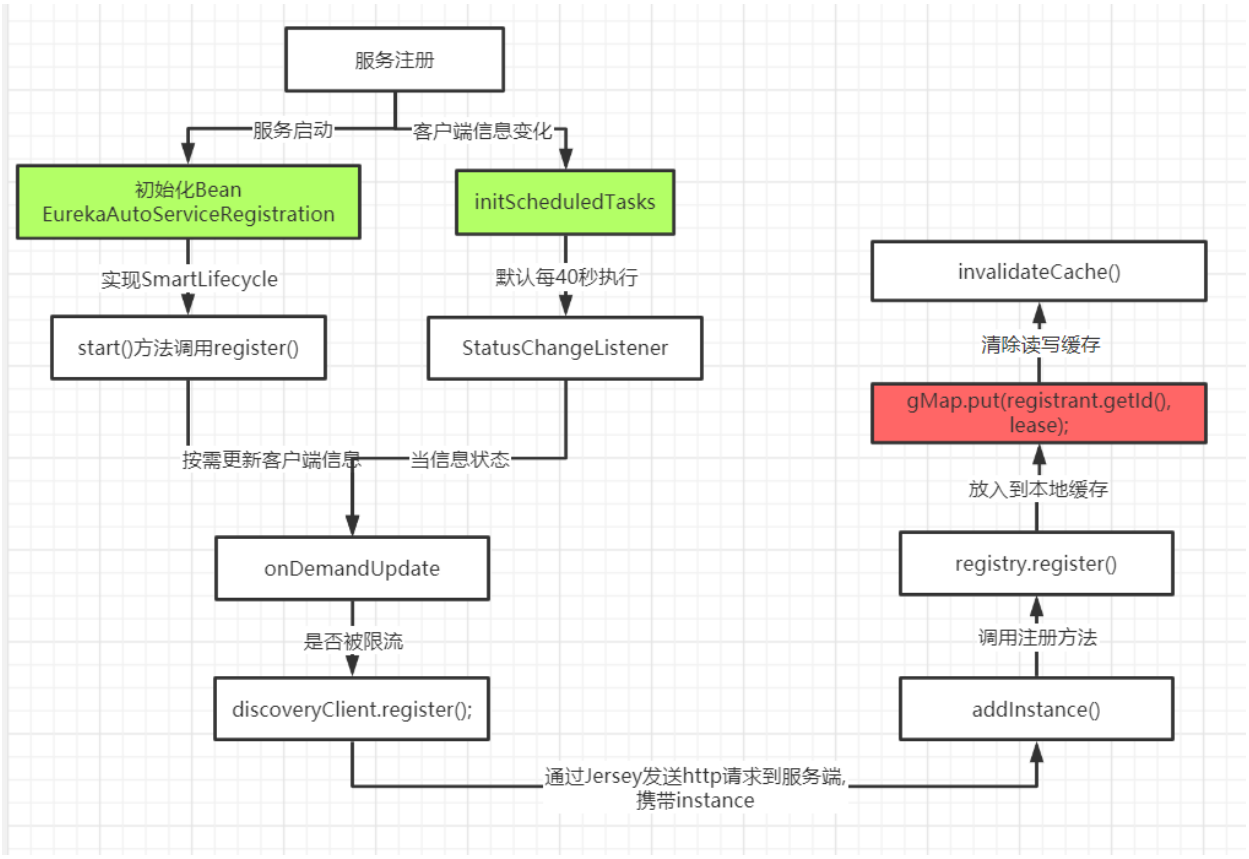
  因此Eureka Server**可以很好的应对因网络故障导致部分节点失联的情况**，而不会像ZK那样如果有一半不可用的情况会导 致整个集群不可用而变成瘫痪。

3.1. 服务注册

3.1.1. Eureka-Client

啥时候会被注册

1. 当客户端刚刚启动的时候
2. 当客户端的instance信息发生改动



当我们的客户端引入了Eureka-Client，当主方法启动时，@SpringBootApplication会扫描所有的META-INF/spring.factories文件下的 xxxAutoConfiguration。这时候 EurekaClientAutoConfiguration 也会被加载。

上面这段代码，很简单，就是实例化了一个Bean，主要是这个Bean实现了SmartLifecycle， 当重写方法 isAutoStartup() 返回值为true，会启动start()方法。 下面可以详细看看这个代码。

```
EurekaClientAutoConfiguration.java
@Bean
@ConditionalOnBean(AutoServiceRegistrationProperties.class)
@ConditionalOnProperty(value = "spring.cloud.service-registry.auto-registration.enabled",
matchIfMissing = true)
public EurekaAutoServiceRegistration eurekaAutoServiceRegistration(ApplicationContext
context, EurekaServiceRegistry registry, EurekaRegistration registration) {
   return new EurekaAutoServiceRegistration(context, registry, registration);
}

EurekaAutoServiceRegistration.java
public class EurekaAutoServiceRegistration implements AutoServiceRegistration,
SmartLifecycle, Ordered {

    public EurekaAutoServiceRegistration(ApplicationContext context,
EurekaServiceRegistry serviceRegistry, EurekaRegistration registration) {
        this.context = context;
        this.serviceRegistry = serviceRegistry;
        this.registration = registration;
    }

    @Override
    public void start() {
        // 设置端口
        if (this.port.get() != 0) {
            if (this.registration.getNonSecurePort() == 0) {
                this.registration.setNonSecurePort(this.port.get());
            }

            if (this.registration.getSecurePort() == 0 && this.registration.isSecure()) {
                this.registration.setSecurePort(this.port.get());
            }
        }


        // 该实例还未启动
        if (!this.running.get() && this.registration.getNonSecurePort() > 0) {
            // 重点; 自动去注册服务
            this.serviceRegistry.register(this.registration);
            // 发布 节点注册事件
            this.context.publishEvent(
                    new InstanceRegisteredEvent<>(this,
this.registration.getInstanceConfig()));
            this.running.set(true);
        }
    }

    @Override
    public boolean isAutoStartup() {
        return true;
    }
}

EurekaServiceRegistry.java
@Override
public void register(EurekaRegistration reg) {
    maybeInitializeClient(reg);

    if (log.isInfoEnabled()) {
        log.info("Registering application " +
reg.getApplicationInfoManager().getInfo().getAppName()
                + " with eureka with status "
                + reg.getInstanceConfig().getInitialStatus());
    }

    // 设置初始化状态为 UP
    // 一旦这个实例的状态发生改变, 那么就会被监听器收到, 最终执行注册逻辑
    reg.getApplicationInfoManager()
            .setInstanceStatus(reg.getInstanceConfig().getInitialStatus());

    //
    reg.getHealthCheckHandler().ifAvailable(healthCheckHandler ->
            // 注册到
            reg.getEurekaClient().registerHealthCheck(healthCheckHandler));
}

InstanceInfoReplicator.java
public boolean onDemandUpdate() {
    if (rateLimiter.acquire(burstSize, allowedRatePerMinute)) {
        if (!scheduler.isShutdown()) {
            scheduler.submit(new Runnable() {
                @Override
                public void run() {
                    logger.debug("Executing on-demand update of local InstanceInfo");

                    Future latestPeriodic = scheduledPeriodicRef.get();
                    if (latestPeriodic != null && !latestPeriodic.isDone()) {
                        logger.debug("Canceling the latest scheduled update, it will be
rescheduled at the end of on demand update");
                        latestPeriodic.cancel(false);
                    }
                    // 调用run方法
                    InstanceInfoReplicator.this.run();
                }
            });
            return true;
        } else {
            logger.warn("Ignoring onDemand update due to stopped scheduler");
            return false;
        }
    } else {
        logger.warn("Ignoring onDemand update due to rate limiter");
        return false;
    }
}

public void run() {
    try {
         // 刷新实例信息。
        discoveryClient.refreshInstanceInfo();

        Long dirtyTimestamp = instanceInfo.isDirtyWithTime();
        if (dirtyTimestamp != null) {

            // 注册自己的服务
            discoveryClient.register();
            instanceInfo.unsetIsDirty(dirtyTimestamp);
        }
    } catch (Throwable t) {
        logger.warn("There was a problem with the instance info replicator", t);
    } finally {
        Future next = scheduler.schedule(this, replicationIntervalSeconds,
TimeUnit.SECONDS);
        scheduledPeriodicRef.set(next);
    }
}
```

第二种是当我们客户端instance信息发生变化

```
private void initScheduledTasks() {
    //省略，刷新缓存的定时器

    // 监听instance的状态变更
    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()) {
                logger.warn("Saw local status change event {}", statusChangeEvent);
            } else {
                logger.info("Saw local status change event {}", statusChangeEvent);
            }
            // 调用方法
            instanceInfoReplicator.onDemandUpdate();
        }
    };

    if (clientConfig.shouldOnDemandUpdateStatusChange()) {
        applicationInfoManager.registerStatusChangeListener(statusChangeListener);
    }


instanceInfoReplicator.start(clientConfig.getInitialInstanceInfoReplicationIntervalSecond
s());
}
```

总结： 服务注册分为2种。
第一种： 当应用启动的时候，如果应用开启了自动注册（默认开启）， 那么在自动配置类加载的时候，会通过EurekaAutoServiceRegistration实例化的时候，去改变instance的status，然后调用注册。
第二种： 主要应用于启动之后，当应用的信息发生改变之后，每40每秒执行一次的线程，检测到了，也会自动去注册一次。

DiscoveryClient.register()

```
DiscoveryClient.java
/**
 * Register with the eureka service by making the appropriate REST call.
 */
boolean register() throws Throwable {
    logger.info(PREFIX + "{}: registering service...", appPathIdentifier);
    EurekaHttpResponse<Void> httpResponse;
    try {
        //发起HTTP请求
        httpResponse = eurekaTransport.registrationClient.register(instanceInfo);
    } catch (Exception e) {
        logger.warn(PREFIX + "{} - registration failed {}", appPathIdentifier,
e.getMessage(), e);
```

```
        throw e;
    }
    if (logger.isInfoEnabled()) {
        logger.info(PREFIX + "{} - registration status: {}", appPathIdentifier,
httpResponse.getStatusCode());
    }
    return httpResponse.getStatusCode() == 204;
}
```

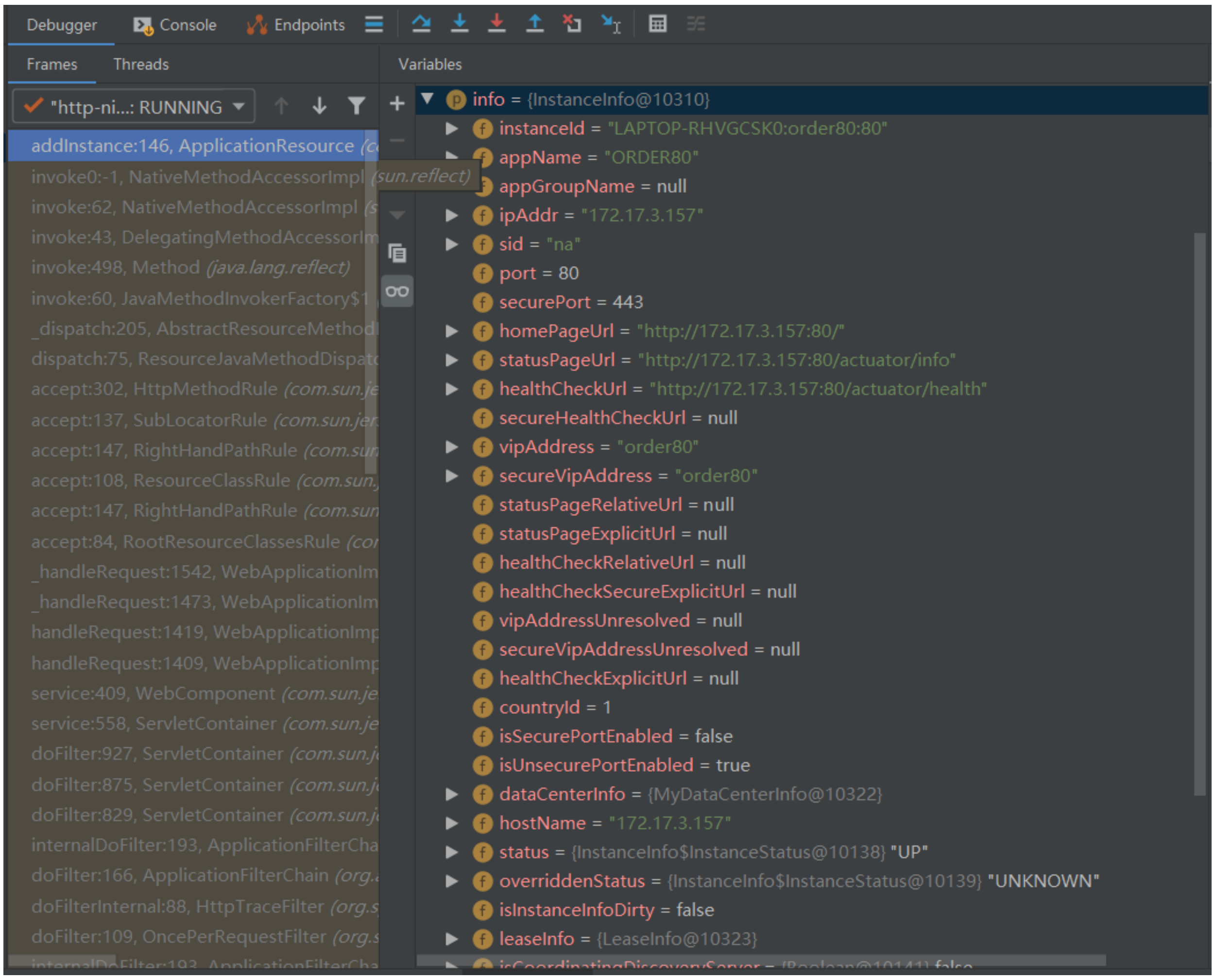
使用的Jersey框架来完成http的请求调用

```
AbstractJerseyEurekaHttpClient.java
@Override
public EurekaHttpResponse<Void> register(InstanceInfo info) {
    // 请求url
    String urlPath = "apps/" + info.getAppName();
    ClientResponse response = null;
    try {
        Builder resourceBuilder =
jerseyClient.resource(serviceUrl).path(urlPath).getRequestBuilder();
        addExtraHeaders(resourceBuilder);
        response = resourceBuilder
                .header("Accept-Encoding", "gzip")
                .type(MediaType.APPLICATION_JSON_TYPE)
                .accept(MediaType.APPLICATION_JSON)
                 // post请求; 请求参数
                .post(ClientResponse.class, info);
        return
anEurekaHttpResponse(response.getStatus()).headers(headersOf(response)).build();
    } finally {
        if (logger.isDebugEnabled()) {
            logger.debug("Jersey HTTP POST {}/{} with instance {}; statusCode={}",
serviceUrl, urlPath, info.getId(),
                    response == null ? "N/A" : response.getStatus());
        }
        if (response != null) {
            response.close();
        }
    }
}
```

POST 请求 Eureka-Server 的 apps/${APP_NAME} 接口，参数为 InstanceInfo ，实现注册实例信息的注册。

3.1.2. Eureka-Service

```
ApplicationResource.java
@POST
@Consumes({"application/json", "application/xml"})
public Response addInstance(InstanceInfo info,
                            @HeaderParam(PeerEurekaNode.HEADER_REPLICATION) String
isReplication) {
    logger.debug("Registering instance {} (replication={})", info.getId(),
isReplication);
    // 参数校验
    if (isBlank(info.getId())) {
        return Response.status(400).entity("Missing instanceId").build();
    } else if (isBlank(info.getHostName())) {
        return Response.status(400).entity("Missing hostname").build();
    } else if (isBlank(info.getIPAddr())) {
        return Response.status(400).entity("Missing ip address").build();
    }

    // handle cases where clients may be registering with bad DataCenterInfo with missing
data
    DataCenterInfo dataCenterInfo = info.getDataCenterInfo();
    if (dataCenterInfo instanceof UniqueIdentifier) {
        String dataCenterInfoId = ((UniqueIdentifier) dataCenterInfo).getId();
        if (isBlank(dataCenterInfoId)) {
            boolean experimental =
"true".equalsIgnoreCase(serverConfig.getExperimental("registration.validation.dataCenterI
nfoId"));
            if (experimental) {
                String entity = "DataCenterInfo of type " + dataCenterInfo.getClass() + "
must contain a valid id";
                return Response.status(400).entity(entity).build();
            } else if (dataCenterInfo instanceof AmazonInfo) {
                AmazonInfo amazonInfo = (AmazonInfo) dataCenterInfo;
                String effectiveId = amazonInfo.get(AmazonInfo.MetaDataKey.instanceId);
                if (effectiveId == null) {

amazonInfo.getMetadata().put(AmazonInfo.MetaDataKey.instanceId.getName(), info.getId());
                }
            } else {
                logger.warn("Registering DataCenterInfo of type {} without an appropriate
id", dataCenterInfo.getClass());
            }
        }
    }

    // 重点代码
    registry.register(info, "true".equals(isReplication));
    return Response.status(204).build();  // 204 to be backwards compatible
}
```



```
"http-ni...: RUNNING
addInstance:146, ApplicationResource (c
invoke0:-1, NativeMethodAccessorImpl (sun.reflect)
invoke:62, NativeMethodAccessorImpl (s
invoke:43, DelegatingMethodAccessorIm
invoke:498, Method (java.lang.reflect)
invoke:60, JavaMethodInvokerFactory$1
_dispatch:205, AbstractResourceMethod
dispatch:75, ResourceJavaMethodDispatc
accept:302, HttpMethodRule (com.sun.je
accept:137, SubLocatorRule (com.sun.jer
accept:147, RightHandPathRule (com.sun
accept:108, ResourceClassRule (com.sun.
accept:147, RightHandPathRule (com.sun
accept:84, RootResourceClassesRule (co
_handleRequest:1542, WebApplicationIm
_handleRequest:1473, WebApplicationIm
handleRequest:1419, WebApplicationImp
handleRequest:1409, WebApplicationImp
service:409, WebComponent (com.sun.je
service:558, ServletContainer (com.sun.je
doFilter:927, ServletContainer (com.sun.j
doFilter:875, ServletContainer (com.sun.j
doFilter:829, ServletContainer (com.sun.j
internalDoFilter:193, ApplicationFilterCha
doFilter:166, ApplicationFilterChain (org.
doFilterInternal:88, HttpTraceFilter (org.s
doFilter:109, OncePerRequestFilter (org.s
```

```
info = {InstanceInfo@10310}
  instanceId = "LAPTOP-RHVGCSK0:order80:80"
  appName = "ORDER80"
  appGroupName = null
  ipAddr = "172.17.3.157"
  sid = "na"
  port = 80
  securePort = 443
  homePageUrl = "http://172.17.3.157:80/"
  statusPageUrl = "http://172.17.3.157:80/actuator/info"
  healthCheckUrl = "http://172.17.3.157:80/actuator/health"
  secureHealthCheckUrl = null
  vipAddress = "order80"
  secureVipAddress = "order80"
  statusPageRelativeUrl = null
  statusPageExplicitUrl = null
  healthCheckRelativeUrl = null
  healthCheckSecureExplicitUrl = null
  vipAddressUnresolved = null
  secureVipAddressUnresolved = null
  healthCheckExplicitUrl = null
  countryId = 1
  isSecurePortEnabled = false
  isUnsecurePortEnabled = true
  dataCenterInfo = {MyDataCenterInfo@10322}
  hostName = "172.17.3.157"
  status = {InstanceInfo$InstanceStatus@10138} "UP"
  overriddenStatus = {InstanceInfo$InstanceStatus@10139} "UNKNOWN"
  isInstanceInfoDirty = false
  leaseInfo = {LeaseInfo@10323}
```

浏览器发送 localhost:7001/eureka/apps

```
<applications>
    <versions__delta>1</versions__delta>
    # applicaitons的组成的hash
    <apps__hashcode>UP_3_</apps__hashcode>
    <application>
    # 应用名
    <name>CLOUD-PROVIDER-PAYMENT</name>
    # 实例
    <instance>
        # 实例ID需要唯一
        <instanceId>LAPTOP-RHVGCSK0:payment:8001</instanceId>
        <hostName>192.168.31.193</hostName>
        <app>CLOUD-PROVIDER-PAYMENT</app>
        <ipAddr>192.168.31.193</ipAddr>
        <status>UP</status>
        <overriddenstatus>UNKNOWN</overriddenstatus>
        <port enabled="true">8001</port>
        <securePort enabled="false">443</securePort>
        <countryId>1</countryId>
        <dataCenterInfo class="com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo">
            <name>MyOwn</name>
        </dataCenterInfo>
        <leaseInfo>
            <renewalIntervalInSecs>30</renewalIntervalInSecs>
            <durationInSecs>90</durationInSecs>
```

```
                <registrationTimestamp>1617633199552</registrationTimestamp>
                <lastRenewalTimestamp>1617633829600</lastRenewalTimestamp>
                <evictionTimestamp>0</evictionTimestamp>
                <serviceUpTimestamp>1617633199552</serviceUpTimestamp>
            </leaseInfo>
            <metadata>
                <management.port>8001</management.port>
            </metadata>
            <homePageUrl>http://192.168.31.193:8001/</homePageUrl>
            <statusPageUrl>http://192.168.31.193:8001/actuator/info</statusPageUrl>
            <healthCheckUrl>http://192.168.31.193:8001/actuator/health</healthCheckUrl>
            <vipAddress>cloud-provider-payment</vipAddress>
            <secureVipAddress>cloud-provider-payment</secureVipAddress>
            <isCoordinatingDiscoveryServer>false</isCoordinatingDiscoveryServer>
            <lastUpdatedTimestamp>1617633199552</lastUpdatedTimestamp>
            <lastDirtyTimestamp>1617633199491</lastDirtyTimestamp>
            <actionType>ADDED</actionType>
        </instance>
        </application>
        <application>
            <name>ORDER80</name>
            # 多个实例
            <instance>
                <instanceId>LAPTOP-RHVGCSK0:order:80</instanceId>
                <hostName>192.168.31.193</hostName>
                <app>ORDER80</app>
                <ipAddr>192.168.31.193</ipAddr>
                <status>UP</status>
                <overriddenstatus>UNKNOWN</overriddenstatus>
                <port enabled="true">80</port>
                <securePort enabled="false">443</securePort>
                <countryId>1</countryId>
                <dataCenterInfo
  class="com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo">
                    <name>MyOwn</name>
                </dataCenterInfo>
                <leaseInfo>
                    <renewalIntervalInSecs>30</renewalIntervalInSecs>
                    <durationInSecs>90</durationInSecs>
                    <registrationTimestamp>1617633135195</registrationTimestamp>
                    <lastRenewalTimestamp>1617633825249</lastRenewalTimestamp>
                    <evictionTimestamp>0</evictionTimestamp>
                    <serviceUpTimestamp>1617633135195</serviceUpTimestamp>
                </leaseInfo>
                <metadata>
                    <management.port>80</management.port>
                </metadata>
                <homePageUrl>http://192.168.31.193:80/</homePageUrl>
                <statusPageUrl>http://192.168.31.193:80/actuator/info</statusPageUrl>
                <healthCheckUrl>http://192.168.31.193:80/actuator/health</healthCheckUrl>
                <vipAddress>order80</vipAddress>
                <secureVipAddress>order80</secureVipAddress>
                <isCoordinatingDiscoveryServer>false</isCoordinatingDiscoveryServer>
                <lastUpdatedTimestamp>1617633135195</lastUpdatedTimestamp>
                <lastDirtyTimestamp>1617633135119</lastDirtyTimestamp>
                <actionType>ADDED</actionType>
            </instance>
            <instance>
                <instanceId>LAPTOP-RHVGCSK0:order:81</instanceId>
                <hostName>192.168.31.193</hostName>
                <app>ORDER80</app>
                <ipAddr>192.168.31.193</ipAddr>
                <status>UP</status>
                <overriddenstatus>UNKNOWN</overriddenstatus>
                <port enabled="true">81</port>
                <securePort enabled="false">443</securePort>
                <countryId>1</countryId>
                <dataCenterInfo
    class="com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo">
                <name>MyOwn</name>
                </dataCenterInfo>
                <leaseInfo>
                    <renewalIntervalInSecs>30</renewalIntervalInSecs>
                    <durationInSecs>90</durationInSecs>
                    <registrationTimestamp>1617631878936</registrationTimestamp>
                    <lastRenewalTimestamp>1617633829226</lastRenewalTimestamp>
                    <evictionTimestamp>0</evictionTimestamp>
                    <serviceUpTimestamp>1617631878937</serviceUpTimestamp>
                </leaseInfo>
                <metadata>
                    <management.port>81</management.port>
                </metadata>
                <homePageUrl>http://192.168.31.193:81/</homePageUrl>
                <statusPageUrl>http://192.168.31.193:81/actuator/info</statusPageUrl>
                <healthCheckUrl>http://192.168.31.193:81/actuator/health</healthCheckUrl>
                <vipAddress>order80</vipAddress>
                <secureVipAddress>order80</secureVipAddress>
                <isCoordinatingDiscoveryServer>false</isCoordinatingDiscoveryServer>
                <lastUpdatedTimestamp>1617631878937</lastUpdatedTimestamp>
                <lastDirtyTimestamp>1617631878931</lastDirtyTimestamp>
                <actionType>ADDED</actionType>
            </instance>
        </application>
    </applications>
```

上面的register方法，最终调用的是PeerAwareInstanceRegistryImpl的方法

```
PeerAwareInstanceRegistryImpl.java
@Override
public void register(final InstanceInfo info, final boolean isReplication) {
    // 默认的过期时间
    int leaseDuration = Lease.DEFAULT_DURATION_IN_SECS;
    // 如果客户端有自定义的过期换时间，以客户端的为准
    if (info.getLeaseInfo() != null && info.getLeaseInfo().getDurationInSecs() > 0) {
        leaseDuration = info.getLeaseInfo().getDurationInSecs();
    }
    // 实例注册
    super.register(info, leaseDuration, isReplication);
    // 复制到同等服务节点上去
    replicateToPeers(Action.Register, info.getAppName(), info.getId(), info, null,
isReplication);
}
```

需要先了解一下Lease这个对象，因为Eureka-Server最终处理注册信息的时候，都会转化为这个对象来处理。

```java
public class Lease<T> {

    // 三种行为：服务注册、服务下线、服务续约
    enum Action {
        Register, Cancel, Renew
    };

    // 默认的过期时间 90s
    public static final int DEFAULT_DURATION_IN_SECS = 90;

    // 实例信息
    private T holder;

    // 服务剔除是时间，当服务下线的时候，会过来更新这个时间戳registrationTimestamp
    private long evictionTimestamp;
    // 服务注册的时间
    private long registrationTimestamp;
    // 服务启动时间 ，当客户端在注册的时候，instanceInfo的status为UP的时候，则更新这个 时间戳
    private long serviceUpTimestamp;
    // Make it volatile so that the expiration task would see this quicker
    // 最后更新时间，每次续约的时候，都会更新这个时间戳，在判断实例是否过期时，需要用到这个属性。
    private volatile long lastUpdateTimestamp;
    // 过期时间
    private long duration;

    public Lease(T r, int durationInSecs) {
        holder = r;
        registrationTimestamp = System.currentTimeMillis();
        lastUpdateTimestamp = registrationTimestamp;
        duration = (durationInSecs * 1000);
    }

    public void renew() {
        lastUpdateTimestamp = System.currentTimeMillis() + duration;
    }

    public void cancel() {
        if (evictionTimestamp <= 0) {
            evictionTimestamp = System.currentTimeMillis();
        }
    }

    public void serviceUp() {
        if (serviceUpTimestamp == 0) {
            serviceUpTimestamp = System.currentTimeMillis();
        }
    }

    public boolean isExpired() {
        return isExpired(0l);
    }

    /**
```

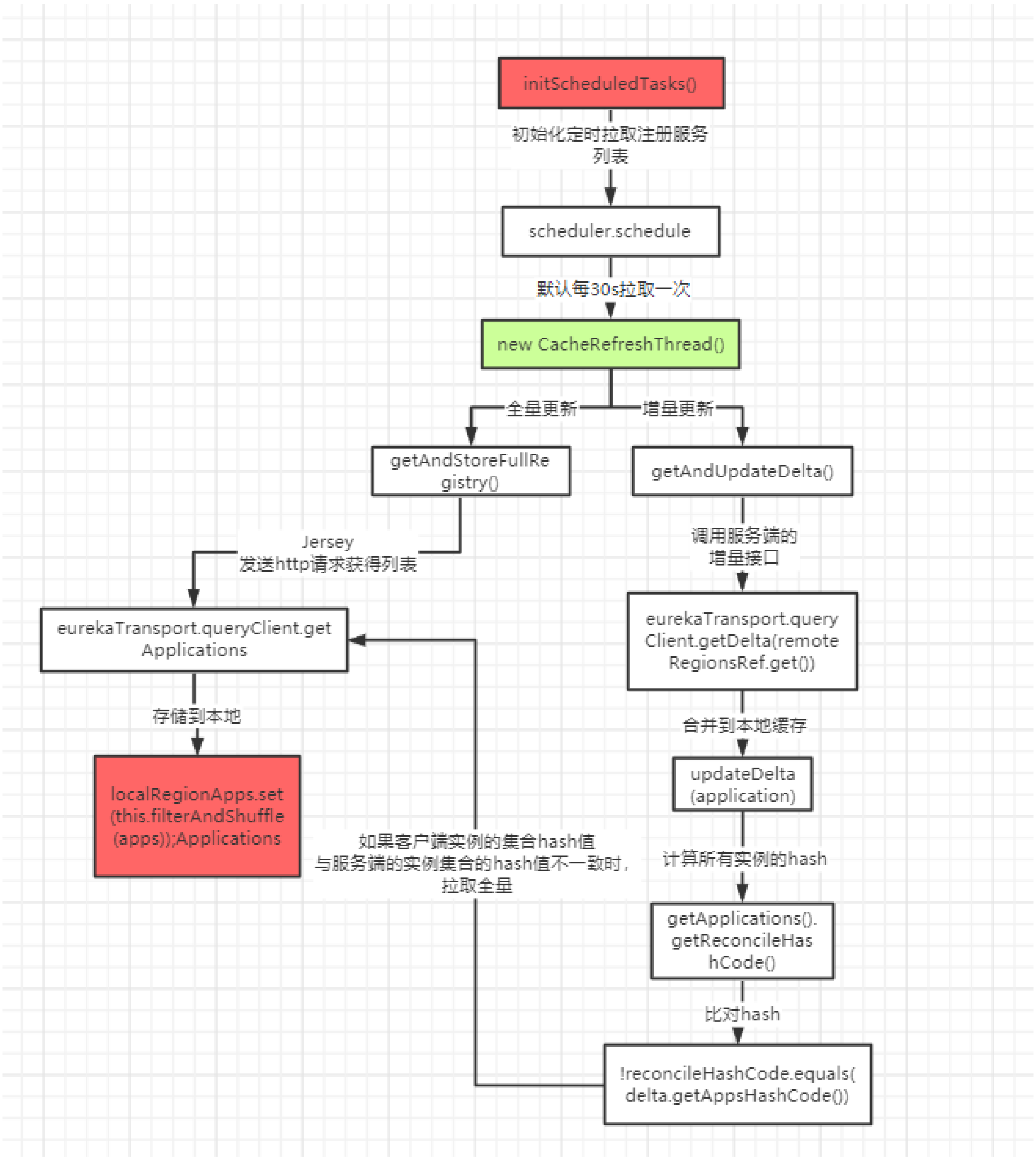
```
      * 服务是否过期
      */
    public boolean isExpired(long additionalLeaseMs) {
        return (evictionTimestamp > 0 || System.currentTimeMillis() >
(lastUpdateTimestamp + duration + additionalLeaseMs));
    }
```

服务注册重要代码

```
 public void register(InstanceInfo registrant, int leaseDuration, boolean isReplication)
{
    try {
        // 上读锁
        read.lock();
         // 通过服务名从本地MAP里面获取当前服务列表。
        Map<String, Lease<InstanceInfo>> gMap = registry.get(registrant.getAppName());
        REGISTER.increment(isReplication);
        // 如果第一次进来，那么gMap为空，则创建一个ConcurrentHashMap放入到registry里面去
        if (gMap == null) {
            final ConcurrentHashMap<String, Lease<InstanceInfo>> gNewMap = new
ConcurrentHashMap<String, Lease<InstanceInfo>>();
            // putIfAbsent方法主要是在向ConcurrentHashMap中添加键—值对的时候，它会先判断该键
值对是否已经存在。
            // 如果不存在（新的entry），那么会向map中添加该键值对，并返回null。
            // 如果已经存在，那么不会覆盖已有的值，直接返回已经存在的值。
            // 线程安全操作
            gMap =
            registry.putIfAbsent(registrant.getAppName(), gNewMap);
            // 表明map中确实不存在，则设置gMap为最新创建的那个
            if (gMap == null) {
                gMap = gNewMap;
            }
        }
        // 从MAP中查询已经存在的Lease信息 （比如第二次来）
        Lease<InstanceInfo> existingLease = gMap.get(registrant.getId());
        // Retain the last dirty timestamp without overwriting it, if there is already a
lease
        if (existingLease != null && (existingLease.getHolder() != null)) {
            Long existingLastDirtyTimestamp =
existingLease.getHolder().getLastDirtyTimestamp();
            Long registrationLastDirtyTimestamp = registrant.getLastDirtyTimestamp();
            logger.debug("Existing lease found (existing={}, provided={}",
existingLastDirtyTimestamp, registrationLastDirtyTimestamp);

            // 如果已经存在的实例的时间 > 注册时间的时间
            if (existingLastDirtyTimestamp > registrationLastDirtyTimestamp) {
                logger.warn("There is an existing lease and the existing lease's dirty
timestamp {} is greater" +
                        " than the one that is being registered {}",
existingLastDirtyTimestamp, registrationLastDirtyTimestamp);
                logger.warn("Using the existing instanceInfo instead of the new
instanceInfo as the registrant");
                // 那服务信息还是取服务端的实例
                registrant = existingLease.getHolder();
            }
        } else {
            // The lease does not exist and hence it is a new registration
            synchronized (lock) {
                if (this.expectedNumberOfRenewsPerMin > 0) {
                    this.expectedNumberOfRenewsPerMin = this.expectedNumberOfRenewsPerMin
+ 2;
                    this.numberOfRenewsPerMinThreshold =
                            (int) (this.expectedNumberOfRenewsPerMin *
serverConfig.getRenewalPercentThreshold());
                }
            }
            logger.debug("No previous lease information found; it is new registration");
        }
        // 构建一个最新的Lease信息
        Lease<InstanceInfo> lease = new Lease<InstanceInfo>(registrant, leaseDuration);
        if (existingLease != null) {
            // 如果该实例是第一次启动，设置启动启动
            lease.setServiceUpTimestamp(existingLease.getServiceUpTimestamp());
        }

        // 放入本地Map中
        gMap.put(registrant.getId(), lease);
        // 添加到最近的注册队列里面去，以时间戳作为Key， 名称作为value，主要是为了运维界面的统计
数据。
        synchronized (recentRegisteredQueue) {
            recentRegisteredQueue.add(new Pair<Long, String>(
                    System.currentTimeMillis(),
                    registrant.getAppName() + "(" + registrant.getId() + ")"));
        }
        // This is where the initial state transfer of overridden status happens
        if (!InstanceStatus.UNKNOWN.equals(registrant.getOverriddenStatus())) {
            logger.debug("Found overridden status {} for instance {}. Checking to see if
needs to be add to the "
                            + "overrides", registrant.getOverriddenStatus(),
registrant.getId());
            if (!overriddenInstanceStatusMap.containsKey(registrant.getId())) {
                logger.info("Not found overridden id {} and hence adding it",
registrant.getId());
                overriddenInstanceStatusMap.put(registrant.getId(),
registrant.getOverriddenStatus());
            }
        }
        InstanceStatus overriddenStatusFromMap =
overriddenInstanceStatusMap.get(registrant.getId());
        if (overriddenStatusFromMap != null) {
            logger.info("Storing overridden status {} from map",
overriddenStatusFromMap);
            registrant.setOverriddenStatus(overriddenStatusFromMap);
        }

        // Set the status based on the overridden status rules
        InstanceStatus overriddenInstanceStatus = getOverriddenInstanceStatus(registrant,
existingLease, isReplication);
        registrant.setStatusWithoutDirty(overriddenInstanceStatus);
        // 得到instanceStatus，判断是否是UP状态，
        if (InstanceStatus.UP.equals(registrant.getStatus())) {
            lease.serviceUp();
        }

         // 设置注册类型为添加
        registrant.setActionType(ActionType.ADDED);
        // 最近变更记录队列，记录了实例的每次变化，  用于注册信息的增量获取、
        recentlyChangedQueue.add(new RecentlyChangedItem(lease));
        registrant.setLastUpdatedTimestamp();
         // 清除读写缓存
        invalidateCache(registrant.getAppName(), registrant.getVIPAddress(),
registrant.getSecureVipAddress());
        logger.info("Registered instance {}/{} with status {} (replication={})",
                registrant.getAppName(), registrant.getId(), registrant.getStatus(),
isReplication);
    } finally {
        read.unlock();
    }
```

3.2. 拉取服务列表

3.2.1. Eureka-Client



initScheduledTasks()
初始化定时拉取注册服务列表
scheduler.schedule
默认每30s拉取一次
new CacheRefreshThread()
全量更新
增量更新
getAndStoreFullRegistry()
getAndUpdateDelta()
Jersey 发送http请求获得列表
调用服务端的增量接口
eurekaTransport.queryClient.getApplications
eurekaTransport.queryClient.getDelta(remoteRegionsRef.get())
存储到本地
合并到本地缓存
localRegionApps.set(this.filterAndShuffle(apps));Applications
updateDelta(application)
如果客户端实例的集合hash值与服务端的实例集合的hash值不一致时，拉取全量
计算所有实例的hash
getApplications().getReconcileHashCode()
比对hash
!reconcileHashCode.equals(delta.getAppsHashCode())

还是在 initScheduledTasks() 初始化所有的定时任务 这个方法中：

```
 private void initScheduledTasks() {
    if (clientConfig.shouldFetchRegistry()) {
        // registry cache refresh timer
        // 拉取服务30秒；每30秒刷新一次
        int registryFetchIntervalSeconds =
clientConfig.getRegistryFetchIntervalSeconds();
        int expBackOffBound =
clientConfig.getCacheRefreshExecutorExponentialBackOffBound();
        scheduler.schedule(
                new TimedSupervisorTask(
                        "cacheRefresh",
                        scheduler,
                        cacheRefreshExecutor,
                        registryFetchIntervalSeconds,
                        TimeUnit.SECONDS,
                        expBackOffBound,
                        new CacheRefreshThread()
                ),
                registryFetchIntervalSeconds, TimeUnit.SECONDS);
    }
    // 其他代码
 }
```

定时更新服务注册列表线程CacheRefreshThread

```
class CacheRefreshThread implements Runnable {
        public void run() {
            refreshRegistry();
        }
    }

    @VisibleForTesting
    void refreshRegistry() {
        try {
            boolean isFetchingRemoteRegionRegistries =
isFetchingRemoteRegionRegistries();

            boolean remoteRegionsModified = false;
            // This makes sure that a dynamic change to remote regions to fetch is
honored.
            String latestRemoteRegions = clientConfig.fetchRegistryForRemoteRegions();
            // 不做aws环境的配置这个if逻辑不会执行
            if (null != latestRemoteRegions) {
                String currentRemoteRegions = remoteRegionsToFetch.get();
                if (!latestRemoteRegions.equals(currentRemoteRegions)) {
                    // Both remoteRegionsToFetch and AzToRegionMapper.regionsToFetch need
to be in sync
                    synchronized (instanceRegionChecker.getAzToRegionMapper()) {
                        if (remoteRegionsToFetch.compareAndSet(currentRemoteRegions,
latestRemoteRegions)) {
                            String[] remoteRegions = latestRemoteRegions.split(",");
                            remoteRegionsRef.set(remoteRegions);

instanceRegionChecker.getAzToRegionMapper().setRegionsToFetch(remoteRegions);
                            remoteRegionsModified = true;
                        } else {
                            logger.info("Remote regions to fetch modified concurrently,"
+
                                    " ignoring change from {} to {}",
currentRemoteRegions, latestRemoteRegions);
                        }
                    }
                } else {
                    // Just refresh mapping to reflect any DNS/Property change
                    instanceRegionChecker.getAzToRegionMapper().refreshMapping();
                }
            }
            // 重要代码，拉取服务列表
            boolean success = fetchRegistry(remoteRegionsModified);
            if (success) {
                registrySize = localRegionApps.get().size();
                lastSuccessfulRegistryFetchTimestamp = System.currentTimeMillis();
            }

        } catch (Throwable e) {
            logger.error("Cannot fetch registry from server", e);
        }
    }
}
```

由上可以看到，系统在启动的时候，初始化了一个定时器，每30秒一次，用来刷新本地缓存信息。

获得实例信息

```
/**
 * 客户端的服务列表
 */
private final AtomicReference<Applications> localRegionApps = new
AtomicReference<Applications>();

private boolean fetchRegistry(boolean forceFullRegistryFetch) {
    Stopwatch tracer = FETCH_REGISTRY_TIMER.start();
    try {
        // 取出之前获取的服务列表
        Applications applications = getApplications();

        // 判断多个条件，确定是否触发全量更新，如下任一个满足都会全量更新：
        // 1. 是否禁用增量
        // 2. 是否对某个region特别关注
        // 3. 外部传参是否要全量拉取
        // 4. 本地缓存服务列表是否为empty
        if (clientConfig.shouldDisableDelta()
                ||
(!Strings.isNullOrEmpty(clientConfig.getRegistryRefreshSingleVipAddress()))
                || forceFullRegistryFetch
                || (applications == null)
                || (applications.getRegisteredApplications().size() == 0)
                || (applications.getVersion() == -1)) //Client application does not have
latest library supporting delta
        {
            logger.info("Disable delta property : {}",
clientConfig.shouldDisableDelta());
            logger.info("Single vip registry refresh property : {}",
clientConfig.getRegistryRefreshSingleVipAddress());
            logger.info("Force full registry fetch : {}", forceFullRegistryFetch);
            logger.info("Application is null : {}", (applications == null));
            logger.info("Registered Applications size is zero : {}",
                    (applications.getRegisteredApplications().size() == 0));
            logger.info("Application version is -1: {}", (applications.getVersion() ==
-1));
            // 拉取全量信息
            getAndStoreFullRegistry();
        } else {
            // 拉取并更新增量信息
            getAndUpdateDelta(applications);
        }
        // 重新计算hash值
        applications.setAppsHashCode(applications.getReconcileHashCode());
        logTotalInstances();
    } catch (Throwable e) {
        logger.error(PREFIX + "{} - was unable to refresh its cache! status = {}",
appPathIdentifier, e.getMessage(), e);
        return false;
    } finally {
        if (tracer != null) {
            tracer.stop();
        }
    }


    onCacheRefreshed();

    updateInstanceRemoteStatus();

    return true;
}
```

全量获取

```
private final AtomicReference<Applications> localRegionApps = new
AtomicReference<Applications>();

 private void getAndStoreFullRegistry() throws Throwable {
    long currentUpdateGeneration = fetchRegistryGeneration.get();

    logger.info("Getting all instance registry info from the eureka server");

    Applications apps = null;
    EurekaHttpResponse<Applications> httpResponse =
clientConfig.getRegistryRefreshSingleVipAddress() == null
            ?
            // 全量获取
            eurekaTransport.queryClient.getApplications(remoteRegionsRef.get())
            :
eurekaTransport.queryClient.getVip(clientConfig.getRegistryRefreshSingleVipAddress(),
remoteRegionsRef.get());
    if (httpResponse.getStatusCode() == Status.OK.getStatusCode()) {
        apps = httpResponse.getEntity();
    }
    logger.info("The response status is {}", httpResponse.getStatusCode());

    if (apps == null) {
        logger.error("The application is null for some reason. Not storing this
information");
    } else if (fetchRegistryGeneration.compareAndSet(currentUpdateGeneration,
currentUpdateGeneration + 1)) {
        // 设置到本地缓存里面去
        localRegionApps.set(this.filterAndShuffle(apps));
        logger.debug("Got full registry with apps hashcode {}", apps.getAppsHashCode());
    } else {
        logger.warn("Not updating applications as another thread is updating it
already");
    }
}
```

增量获取

```
DiscoveryClient.java
private void getAndUpdateDelta(Applications applications) throws Throwable {
    long currentUpdateGeneration = fetchRegistryGeneration.get();

    // 增量获取信息
    Applications delta = null;
    EurekaHttpResponse<Applications> httpResponse =
eurekaTransport.queryClient.getDelta(remoteRegionsRef.get());
    if (httpResponse.getStatusCode() == Status.OK.getStatusCode()) {
        delta = httpResponse.getEntity();
    }
    // 增量获取为空，则全量返回
    if (delta == null) {
        logger.warn("The server does not allow the delta revision to be applied because
it is not safe. "
                + "Hence got the full registry.");
        getAndStoreFullRegistry();
    }
    // CAS
    else if (fetchRegistryGeneration.compareAndSet(currentUpdateGeneration,
currentUpdateGeneration + 1)) {
        logger.debug("Got delta update with apps hashcode {}", delta.getAppsHashCode());
        String reconcileHashCode = "";
        // 这里设置原子锁的原因是怕某次调度网络请求时间过长，导致同一时间有多线程拉取到增量信息并
发修改
        if (fetchRegistryUpdateLock.tryLock()) {
            try {
                //  将获取到的增量信息和本地缓存信息合并
                updateDelta(delta);
                // 计算本地的hash
                reconcileHashCode = getReconcileHashCode(applications);
            } finally {
                fetchRegistryUpdateLock.unlock();
            }
        } else {
            logger.warn("Cannot acquire update lock, aborting getAndUpdateDelta");
        }
        // 如果本地的hash与service的hash不一致，全量去拉取
        if (!reconcileHashCode.equals(delta.getAppsHashCode()) ||
clientConfig.shouldLogDeltaDiff()) {
            reconcileAndLogDifference(delta, reconcileHashCode);  // this makes a
remoteCall
        }
    } else {
        logger.warn("Not updating application delta as another thread is updating it
already");
        logger.debug("Ignoring delta update with apps hashcode {}, as another thread is
updating it already", delta.getAppsHashCode());
    }
}
```

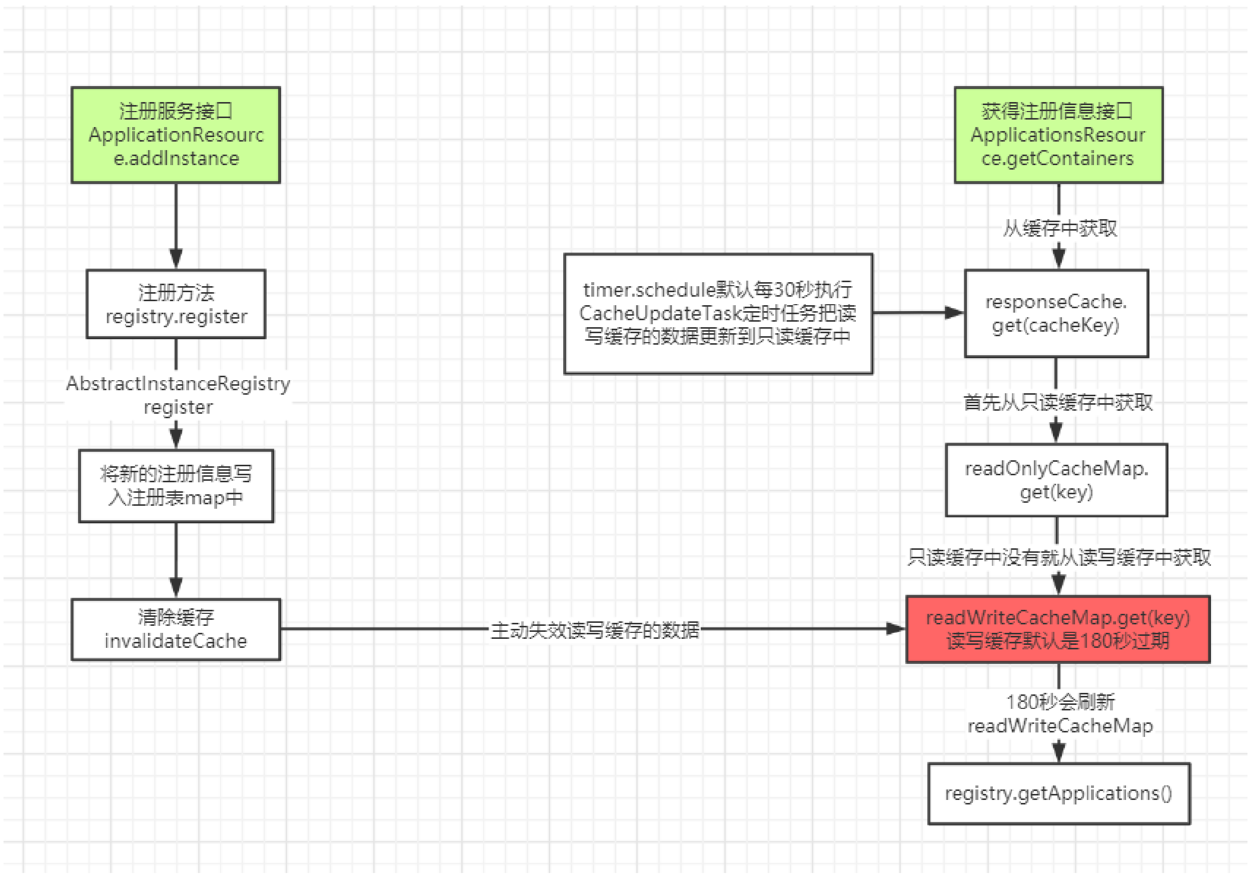
1. 发起http请求，将服务端的客户端变化的信息拉取过来，如： register， cancle, modify 有过这些操作的数据
2. 上锁，防止某次调度网络请求时间过长，导致同一时间有多线程拉取到增量信息并发修改
3. 将请求过来的增量数据和本地的数据做合并
4. 计算hashCode
5. 如果hashCode不一致，增量更新错误，则又会去服务端发起一次全量获取

合并数据

```
private void updateDelta(Applications delta) {
    int deltaCount = 0;
    // 循环增量数据列表
    for (Application app : delta.getRegisteredApplications()) {
        // 编译这个服务中的实例(一个服务是会有多个实例的)
        for (InstanceInfo instance : app.getInstances()) {
            // 获得本地的注册表
            Applications applications = getApplications();
            String instanceRegion = instanceRegionChecker.getInstanceRegion(instance);
            if (!instanceRegionChecker.isLocalRegion(instanceRegion)) {
                Applications remoteApps = remoteRegionVsApps.get(instanceRegion);
                if (null == remoteApps) {
                    remoteApps = new Applications();
                    remoteRegionVsApps.put(instanceRegion, remoteApps);
                }
                applications = remoteApps;
            }

            ++deltaCount;
            // 添加行为
            if (ActionType.ADDED.equals(instance.getActionType())) {
                Application existingApp =
applications.getRegisteredApplications(instance.getAppName());
                if (existingApp == null) {
                    applications.addApplication(app);
                }
                logger.debug("Added instance {} to the existing apps in region {}",
instance.getId(), instanceRegion);
                // 为这个应用添加实例

applications.getRegisteredApplications(instance.getAppName()).addInstance(instance);
            }
            // 修改行为
            else if (ActionType.MODIFIED.equals(instance.getActionType())) {
                Application existingApp =
applications.getRegisteredApplications(instance.getAppName());
                if (existingApp == null) {
                    applications.addApplication(app);
                }
                logger.debug("Modified instance {} to the existing apps ",
instance.getId());
                // 为这个应用添加实例(修改则覆盖)

applications.getRegisteredApplications(instance.getAppName()).addInstance(instance);

            }
            // 删除行为
            else if (ActionType.DELETED.equals(instance.getActionType())) {
                Application existingApp =
applications.getRegisteredApplications(instance.getAppName());
                if (existingApp == null) {
                    applications.addApplication(app);
                }
                logger.debug("Deleted instance {} to the existing apps ",
instance.getId());
                // 为这个应用删除实例

applications.getRegisteredApplications(instance.getAppName()).removeInstance(instance);
            }
        }
    }
    logger.debug("The total number of instances fetched by the delta processor : {}",
deltaCount);

    getApplications().setVersion(delta.getVersion());
    getApplications().shuffleInstances(clientConfig.shouldFilterOnlyUpInstances());

    for (Applications applications : remoteRegionVsApps.values()) {
        applications.setVersion(delta.getVersion());
        applications.shuffleInstances(clientConfig.shouldFilterOnlyUpInstances());
    }
}
```

步骤说明：

1. 从服务端获取了最近这段时间，新注册新来的客户端信息，有过修改的，被删除的， 这三大类的实例信息
2. 然后通过覆盖本地的数据，移除数据，来达到数据合并的需求。



注册服务接口 ApplicationResource.addInstance
注册方法 registry.register
AbstractInstanceRegistry register
将新的注册信息写入注册表map中
清除缓存 invalidateCache
主动失效读写缓存的数据
获得注册信息接口 ApplicationsResource.getContainers
从缓存中获取
timer.schedule默认每30秒执行CacheUpdateTask定时任务把读写缓存的数据更新到只读缓存中
responseCache.get(cacheKey)
首先从只读缓存中获取
readOnlyCacheMap.get(key)
只读缓存中没有就从读写缓存中获取
readWriteCacheMap.get(key) 读写缓存默认是180秒过期
180秒会刷新 readWriteCacheMap
registry.getApplications()

```
ApplicationsResource.java
@Path("/{version}/apps")
public class ApplicationsResource {
// 省略代码

@GET
public Response getContainers(@PathParam("version") String version,
                              @HeaderParam(HEADER_ACCEPT) String acceptHeader,
                              @HeaderParam(HEADER_ACCEPT_ENCODING) String acceptEncoding,
                              @HeaderParam(EurekaAccept.HTTP_X_EUREKA_ACCEPT) String
eurekaAccept,
                              @Context UriInfo uriInfo,
                              @Nullable @QueryParam("regions") String regionsStr) {

    boolean isRemoteRegionRequested = null != regionsStr && !regionsStr.isEmpty();
    String[] regions = null;
    if (!isRemoteRegionRequested) {
        EurekaMonitors.GET_ALL.increment();
    } else {
        regions = regionsStr.toLowerCase().split(",");
        Arrays.sort(regions); // So we don't have different caches for same regions
queried in different order.
        EurekaMonitors.GET_ALL_WITH_REMOTE_REGIONS.increment();
    }

    if (!registry.shouldAllowAccess(isRemoteRegionRequested)) {
        return Response.status(Status.FORBIDDEN).build();
    }
    CurrentRequestVersion.set(Version.toEnum(version));
    // 默认key的类型为JSON
    KeyType keyType = Key.KeyType.JSON;
    // 默认返回值的类型为JSON
    String returnMediaType = MediaType.APPLICATION_JSON;
    // 如果acceptHeader不包含JSON字符串（表示客户端可能不接收JSON类型），则设置返回XML类型的
    if (acceptHeader == null || !acceptHeader.contains(HEADER_JSON_VALUE)) {
        keyType = Key.KeyType.XML;
        returnMediaType = MediaType.APPLICATION_XML;
    }
    // 构建全量数据缓存key
    Key cacheKey = new Key(Key.EntityType.Application,
            ResponseCacheImpl.ALL_APPS,
            keyType, CurrentRequestVersion.get(), EurekaAccept.fromString(eurekaAccept),
regions
    );

    Response response;
    if (acceptEncoding != null && acceptEncoding.contains(HEADER_GZIP_VALUE)) {
        response = Response.ok(responseCache.getGZIP(cacheKey))
                .header(HEADER_CONTENT_ENCODING, HEADER_GZIP_VALUE)
                .header(HEADER_CONTENT_TYPE, returnMediaType)
                .build();
    } else {
        // 重点代码
        response = Response.ok(responseCache.get(cacheKey))
                .build();
    }
    return response;
}
```

responseCache.get(cacheKey)从缓存中获取

```
public String get(final Key key) {
    return get(key, shouldUseReadOnlyResponseCache);
}

@VisibleForTesting
String get(final Key key, boolean useReadOnlyCache) {
    // 重点
    Value payload = getValue(key, useReadOnlyCache);
    if (payload == null || payload.getPayload().equals(EMPTY_PAYLOAD)) {
        return null;
    } else {
        return payload.getPayload();
    }
}
Value getValue(final Key key, boolean useReadOnlyCache) {
    Value payload = null;
    try {
        // 是否使用只读缓存，默认为true
        if (useReadOnlyCache) {
            // 从缓存中获取数据
            final Value currentPayload = readOnlyCacheMap.get(key);
            // 如果不为空，直接返回数据
            if (currentPayload != null) {
                payload = currentPayload;
            }
            // 如果为空
            else {
                // 从读写读写中获取
                payload = readWriteCacheMap.get(key);
                // 同时将数据放入只读缓存中
                readOnlyCacheMap.put(key, payload);
            }
        } else {
            // 从读写读写中获取
            payload = readWriteCacheMap.get(key);
        }
    } catch (Throwable t) {
        logger.error("Cannot get value for key : {}", key, t);
    }
    return payload;
}
```

```
// 只读缓存
private final ConcurrentMap<Key, Value> readOnlyCacheMap = new ConcurrentHashMap<Key,
Value>();
// 读写缓存
private final LoadingCache<Key, Value> readWriteCacheMap;

// 在构造器中实现逻辑
ResponseCacheImpl(EurekaServerConfig serverConfig, ServerCodecs serverCodecs,
AbstractInstanceRegistry registry) {
    this.serverConfig = serverConfig;
    this.serverCodecs = serverCodecs;
    // 是否使用只读缓存，默认为true
    this.shouldUseReadOnlyResponseCache = serverConfig.shouldUseReadOnlyResponseCache();
    this.registry = registry;

    // 缓存更新的时间间隔,默认为30s
    long responseCacheUpdateIntervalMs = serverConfig.getResponseCacheUpdateIntervalMs();
    // 读写缓存构造，使用Google的CacheBuilder缓存
    this.readWriteCacheMap =
            CacheBuilder.newBuilder().initialCapacity(1000)
                    // 过期180s

.expireAfterWrite(serverConfig.getResponseCacheAutoExpirationInSeconds(),
TimeUnit.SECONDS)
                    .removalListener(new RemovalListener<Key, Value>() {
                        @Override
                        public void onRemoval(RemovalNotification<Key, Value>
notification) {
                            Key removedKey = notification.getKey();
                            if (removedKey.hasRegions()) {
                                Key cloneWithNoRegions =
removedKey.cloneWithoutRegions();
                                regionSpecificKeys.remove(cloneWithNoRegions,
removedKey);
                            }
                        }
                    })
                    // 缓存加载器，当缓存不存在时，会自动执行load方法，进行缓存加载。同时返回缓
存数据
                    .build(new CacheLoader<Key, Value>() {
                        @Override
                        public Value load(Key key) throws Exception {
                            if (key.hasRegions()) {
                                Key cloneWithNoRegions = key.cloneWithoutRegions();
                                regionSpecificKeys.put(cloneWithNoRegions, key);
                            }
                            // 加载数据
                            Value value = generatePayload(key);
                            return value;
                        }
                    });

    // 是否使用只读缓存，如果使用，此处则启动一个定时器，默认每隔30s用来复制readWriteCacheMap
的数据至readOnlyCacheMap
    if (shouldUseReadOnlyResponseCache) {
        timer.schedule(getCacheUpdateTask(),
                new Date(((System.currentTimeMillis() / responseCacheUpdateIntervalMs) *
responseCacheUpdateIntervalMs)
                        + responseCacheUpdateIntervalMs),
                responseCacheUpdateIntervalMs);
    }
}

private TimerTask getCacheUpdateTask() {
    return new TimerTask() {
        @Override
        public void run() {
            logger.debug("Updating the client cache from response cache");
            // 遍历只读缓存中的key
            for (Key key : readOnlyCacheMap.keySet()) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Updating the client cache from response cache for key :
{} {} {} {}",
                            key.getEntityType(), key.getName(), key.getVersion(),
key.getType());
                }
                try {
                    CurrentRequestVersion.set(key.getVersion());
                    // 从读写缓存中获得数据
                    Value cacheValue = readWriteCacheMap.get(key);
                    // 从只读缓存中获得数据
                    Value currentCacheValue = readOnlyCacheMap.get(key);
                    // 如果两者不一致，以读写缓存为准，覆盖只读缓存的数据
                    if (cacheValue != currentCacheValue) {
                        readOnlyCacheMap.put(key, cacheValue);
                    }
                } catch (Throwable th) {
                    logger.error("Error while updating the client cache from response
cache for key {}", key.toStringCompact(), th);
                }
            }
        }
    };
}
```

总结：

在拉取注册的时候：

1. 首先从ReadOnlyCacheMap里查缓存的注册表；
2. 若没有，就找ReadWriteCacheMap里缓存的注册表；
3. 如果还没有，就从内存中获取实际的注册表数据。

在注册表发生时候：

1. 会在内存中更新变更的注册表数据，同时过期掉ReadWriteCacheMap；
2. 此过程不会影响ReadOnlyCacheMap提供人家查询注册表；
3. ReadOnlyCacheMap 默认30秒会从ReadWriteCacheMap中更新数据；
4. ReadWriteCacheMap 默认是180秒数据会失效。
5. 下次有服务拉取列表或者是ReadOnlyCacheMap更新时， 如果缓存没有命中，都会去注册表重新获取最新的值。

多级缓存的优点：

1. 尽可能保证了内存注册表数据不会出现频繁的读写冲突问题；
2. 进一步保证了对eurekaService的大量请求，都是快速走纯内存。【如我们公司】

参考文档：
https://github.com/Netflix/eureka
https://www.iocoder.cn/categories/Eureka/