# Eureka概念理解

## 6.1 服务注册

当项目一启动，向eureka-server 发送自己的元数据，（运行的ip，port，健康的监控数据）eureka-server在自己内部保存这些元数据。

## 6.2 服务续约（eureka-client）

项目启动成功了，也向eureka-server 注册自己 成功。 项目还会定时的去eureka-server 汇报自己。代表我还活着。

## 6.3 服务的下线

当项目关闭时，项目会给eureka-server 报告自己，说我要下机了！

## 6.4 服务的剔除（eureka-server）

当服务没有项 eureka-server 汇报自己的状态超过一段时间，eureka-server认为它死了，会把它剔除掉！

eureka通过纯内存的注册表，保证了所有的请求都可以在内存处理，确保了极高的性能，服务保存在内存的**CocurrentHashMap<String,Map<String,Lease<InstanceInfo>>>**

,key是serviceid，value中的key是instanceid，value是服务ip端口，最后更新时间等信息。

* 另外,多级缓存机制，确保了不会针对内存数据结构发生频繁的读写并发冲突操作，进一步提升性能

# Eureka的源码分析

## 7.1 Eureka -server 对外提供Restful的服务

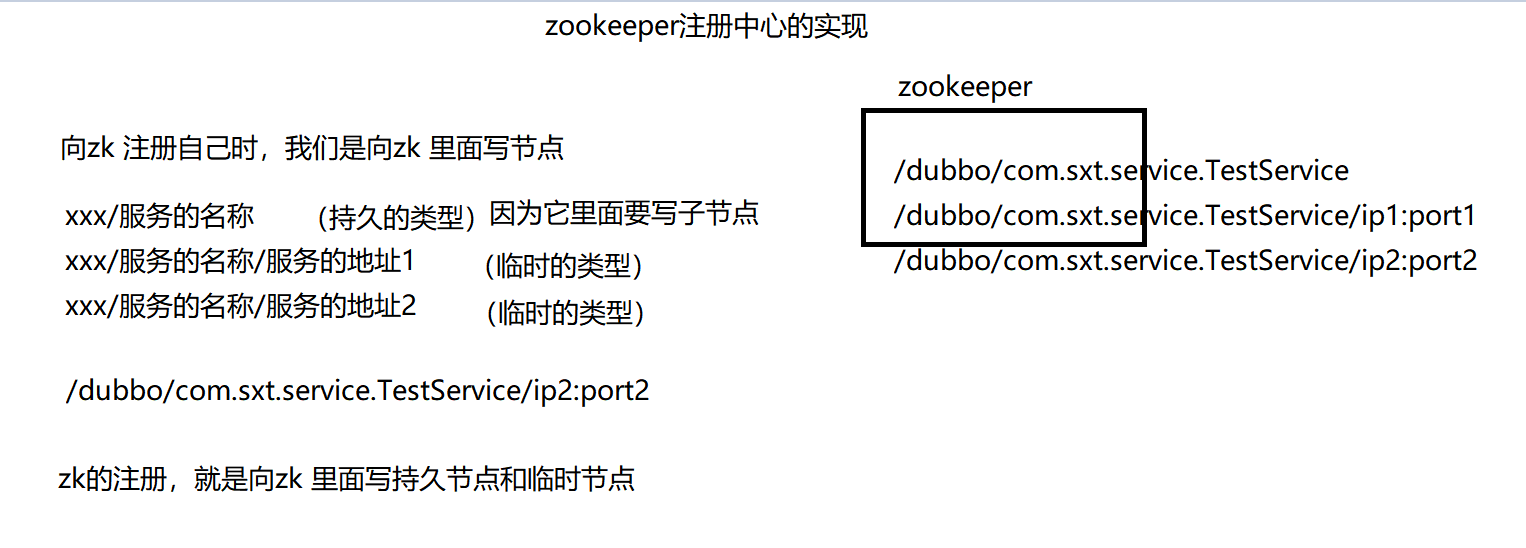
http 服务 + 特定的请求方式 + 特定的url 地址

只要利用这些restful 我们就能对项目实现注册和发现

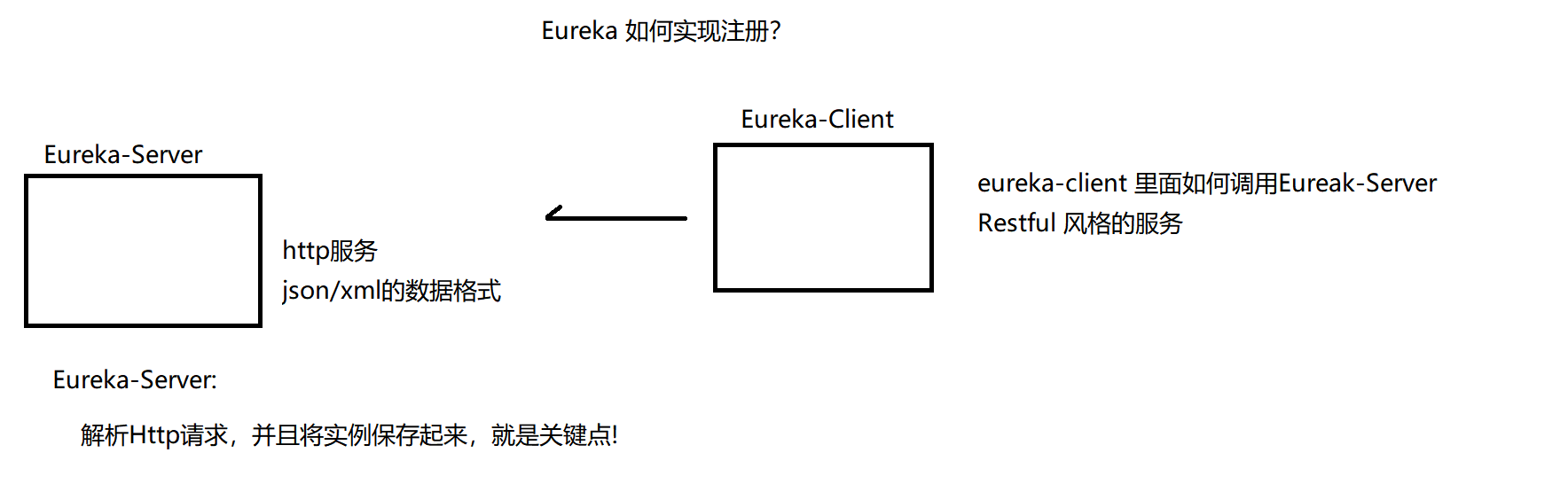
只不过，eureka已经帮我们使用java 语言写了client，让我们的项目只要依赖client 就能实现注册和发现！

只要你会发起Http 请求，那你就有可能自己实现服务的注册和发现。不管你是什么语言！

## 7.2 服务注册的源码分析



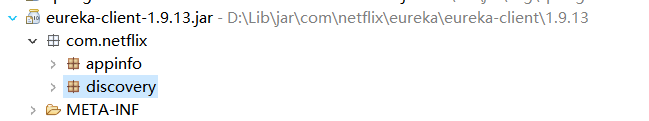
Eureka：



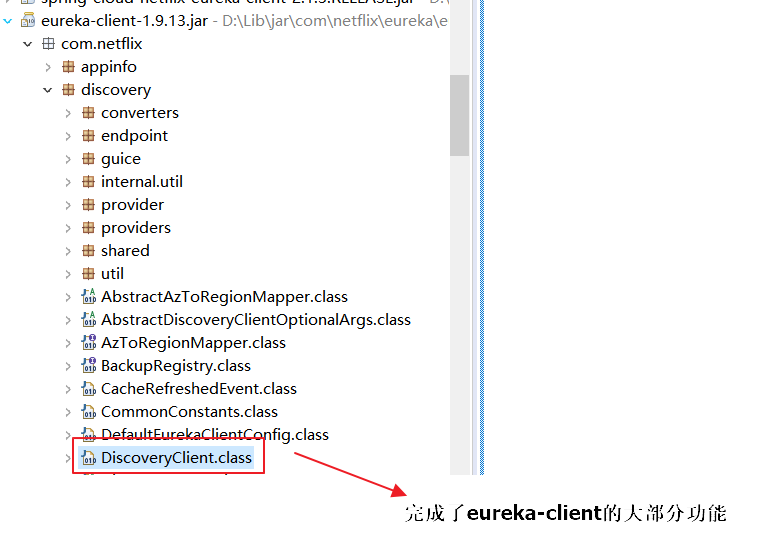
### 7.2.1 eureka-client 会做啥？

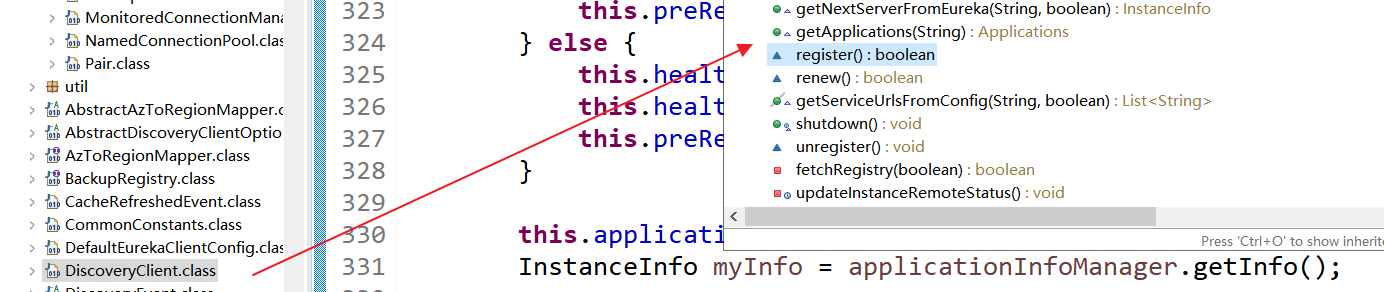
就是对eureka发起http请求，是一个post i请求，当前响应为204 时，代表注册成功了

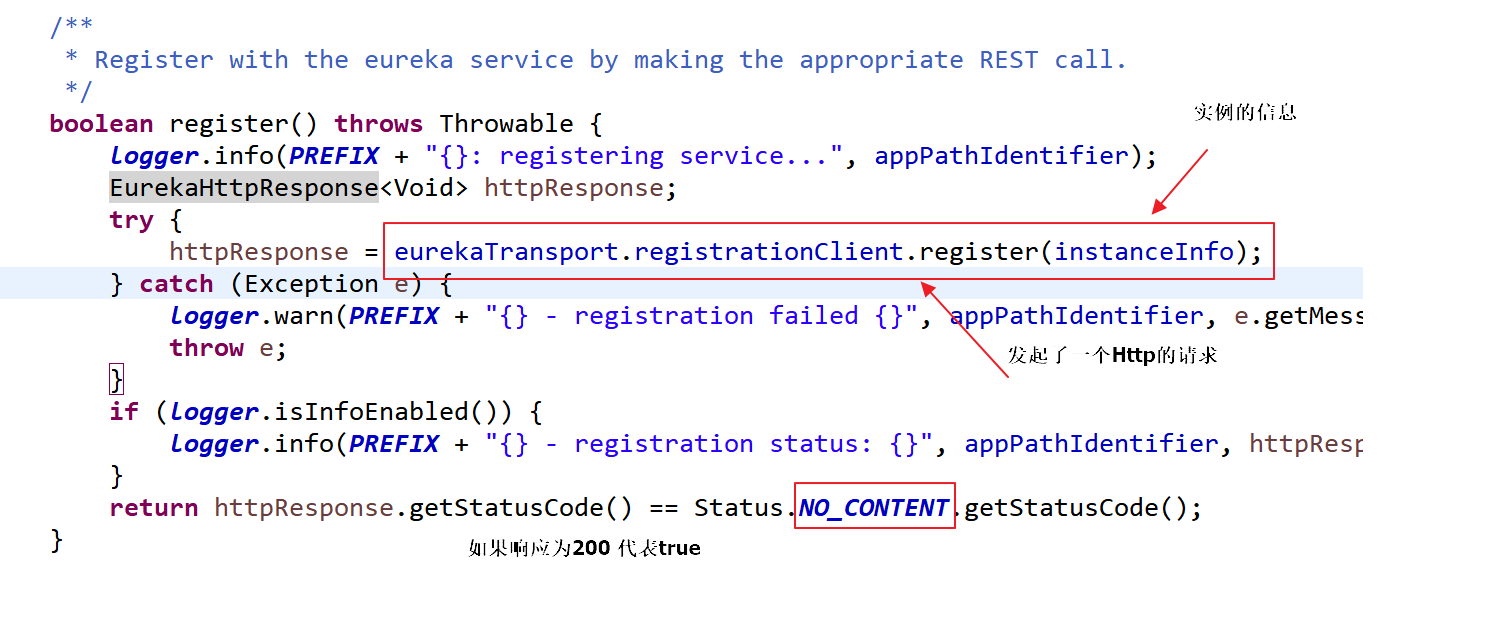
它的很多源码在eureka-client-xxxx.jar 里面



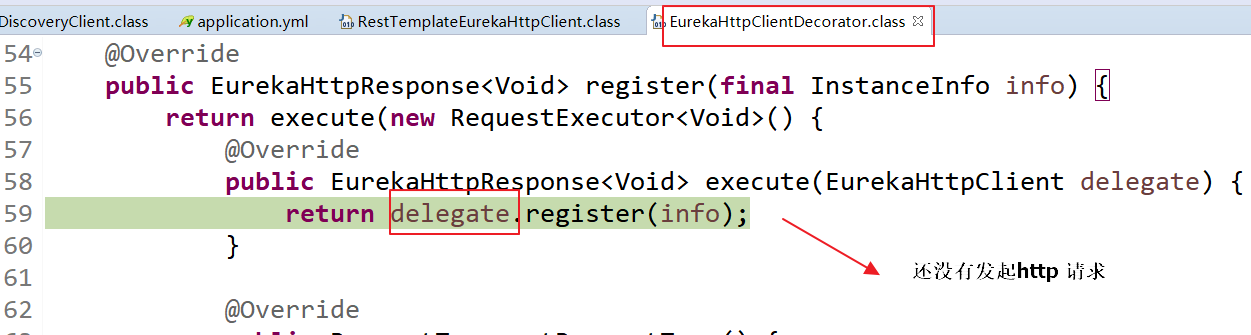
Eureka-client 里面看的是 DiscoveryClient



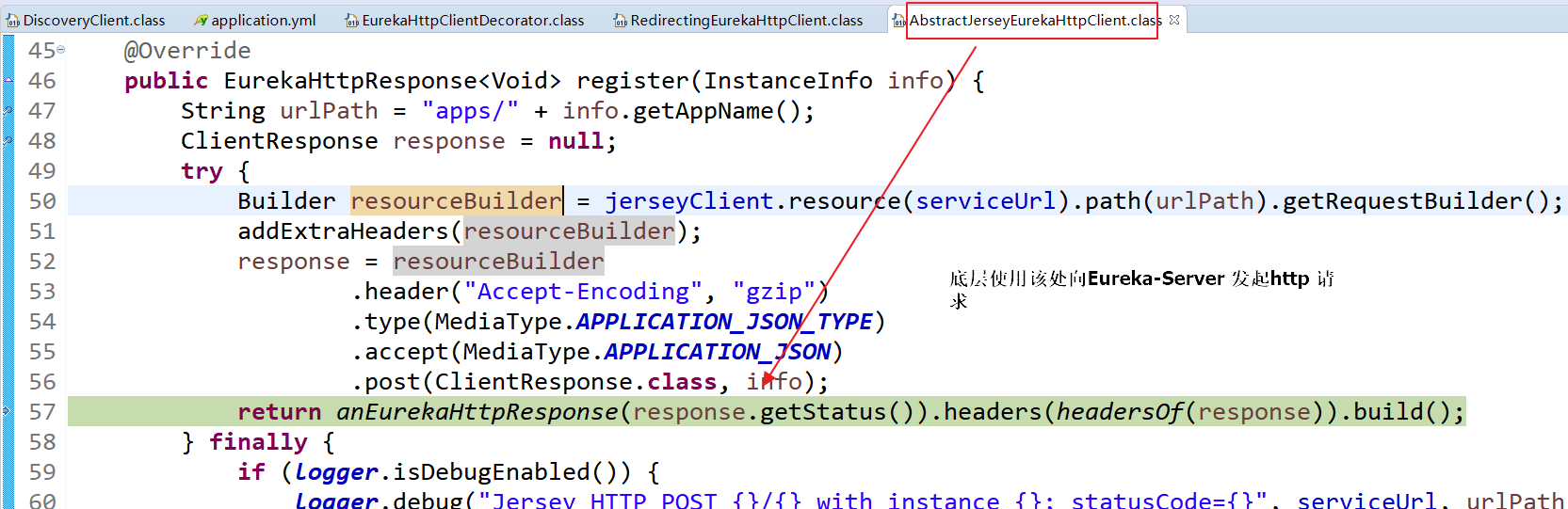




先经过：

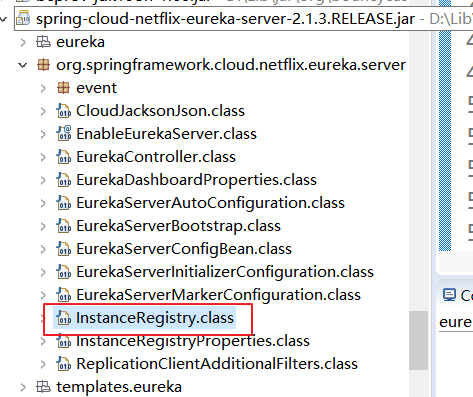


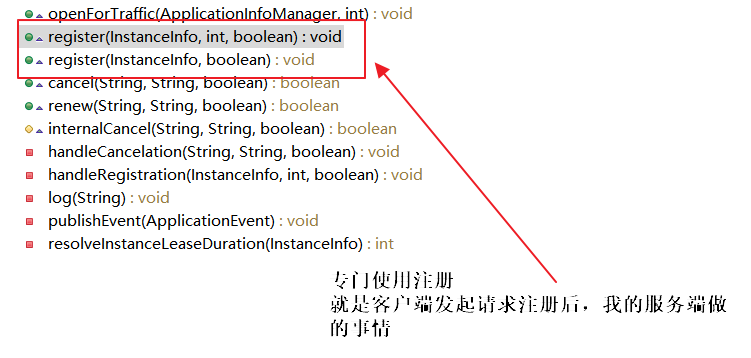
底层使用AbstractJerseyEurekaHttpClient发起了一个Post 请求，对eureka-server



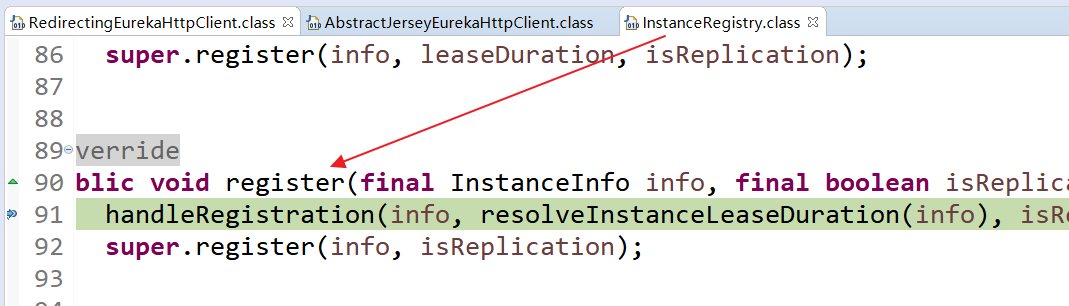
Eureka 会处理该请求

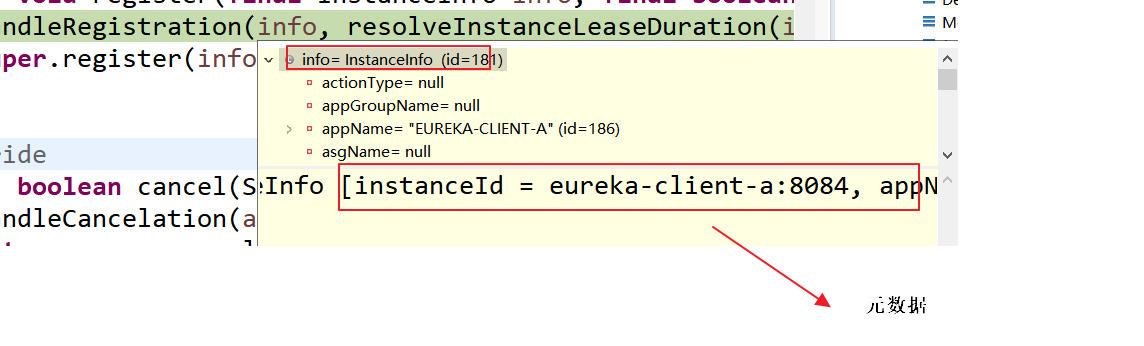
### 7.2.2 eureka-server会做啥？（重要）



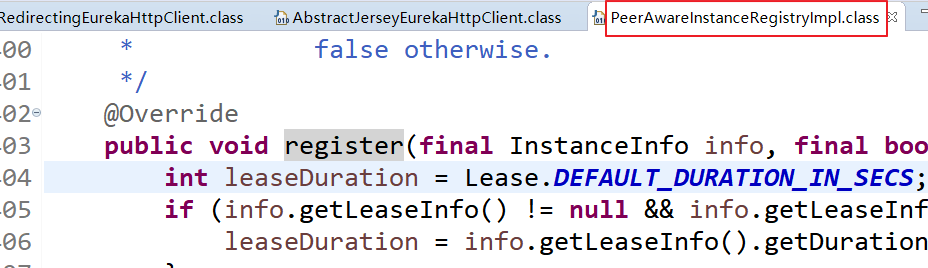


请求过来，先进入

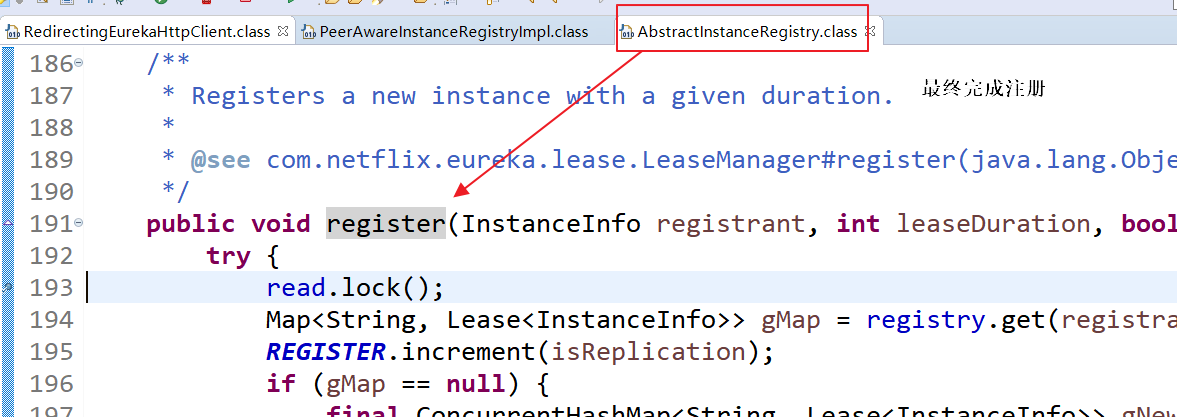


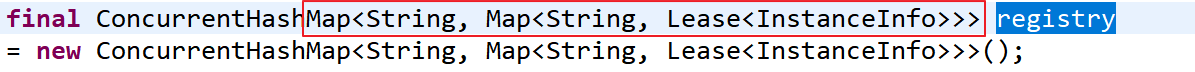


再次经过：

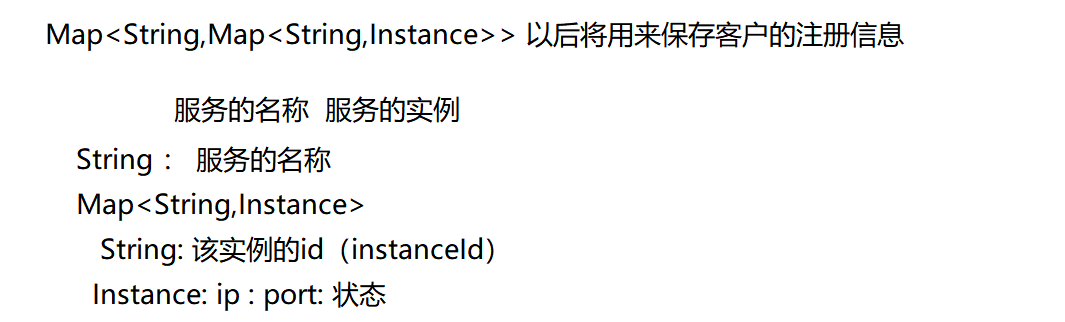


|  |
| --- |
| @Override  **public** **void** register(**final** InstanceInfo info, **final** **boolean** isReplication) {  **int** leaseDuration = Lease.***DEFAULT\_DURATION\_IN\_SECS***;  **if** (info.getLeaseInfo() != **null** && info.getLeaseInfo().getDurationInSecs() > 0) {  leaseDuration = info.getLeaseInfo().getDurationInSecs();  }  // 注册自己  **super**.register(info, leaseDuration, isReplication);  // 把该注册信息 也向别的注册去注册  replicateToPeers(Action.***Register***, info.getAppName(), info.getId(), info, **null**, isReplication);  } |



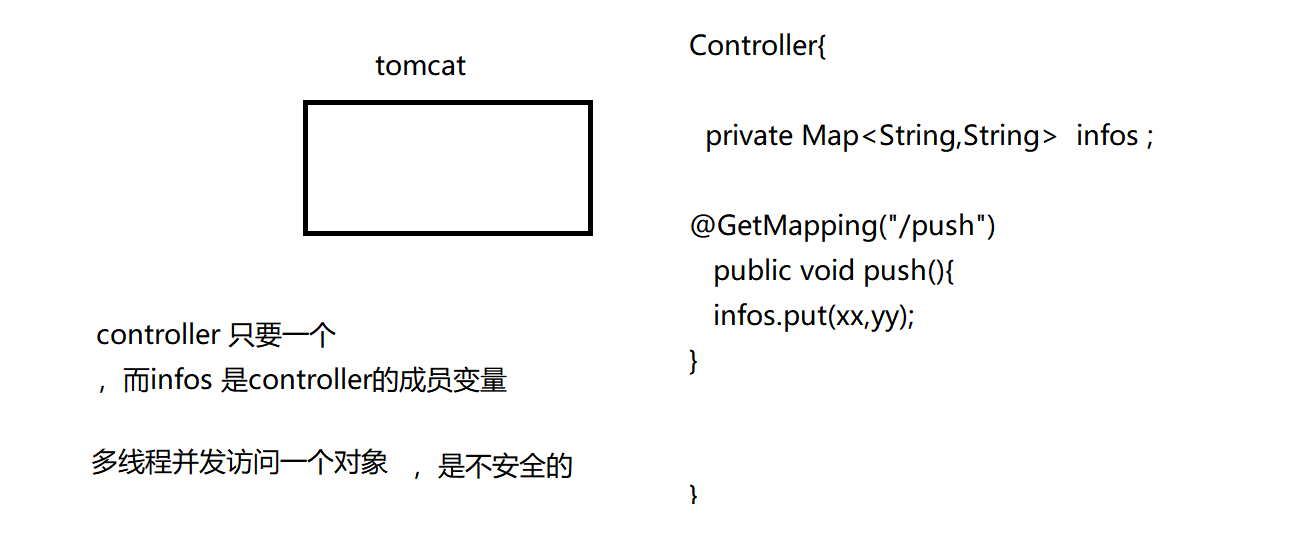


**Map 集合重要**



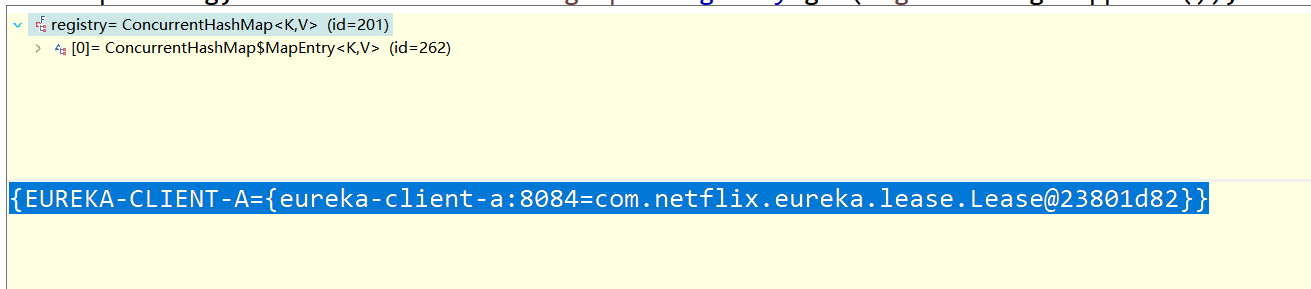
为什么使用ConcurrentHashMap：是安全的hashmap。

因为HashMap是线程不安全：



源码的注释：（完成注册的关键点）

|  |
| --- |
| **public** **void** register(InstanceInfo registrant, **int** leaseDuration, **boolean** isReplication) {  **try** {  read.lock();  // 通过服务的名称去得到注册的实例  Map<String, Lease<InstanceInfo>> gMap = registry.get(registrant.getAppName());  ***REGISTER***.increment(isReplication);  // 若实例的列表为null，因为这是第一个注册  **if** (gMap == **null**) {  // 新建一个实例的列表，通过map 集合来封装实例的信息  **final** ConcurrentHashMap<String, Lease<InstanceInfo>> gNewMap = **new** ConcurrentHashMap<String, Lease<InstanceInfo>>();  // 将该新建出的gNewMap 放在注册中心里面  gMap = registry.putIfAbsent(registrant.getAppName(), gNewMap);  **if** (gMap == **null**) {  gMap = gNewMap;  }  }  // gMap 就是该服务的实例  // Map<String,Instance> registrant.getId():实例的id （instanceId）  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);  // this is a > instead of a >= because if the timestamps are equal, we still take the remote transmitted  // InstanceInfo instead of the server local copy.  **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");  registrant = existingLease.getHolder();  }  } **else** {  // 开始注册  **synchronized** (lock) {  **if** (**this**.expectedNumberOfClientsSendingRenews > 0) {  // Since the client wants to register it, increase the number of clients sending renews  **this**.expectedNumberOfClientsSendingRenews = **this**.expectedNumberOfClientsSendingRenews + 1;  updateRenewsPerMinThreshold();  }  }  ***logger***.debug("No previous lease information found; it is new registration");  }  // 新建了一个实例  Lease<InstanceInfo> lease = **new** Lease<InstanceInfo>(registrant, leaseDuration);  **if** (existingLease != **null**) {  lease.setServiceUpTimestamp(existingLease.getServiceUpTimestamp());  }  // 将该实例放入注册的map 列表里面  gMap.put(registrant.getId(), lease);  **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);  // If the lease is registered with UP status, set lease service up timestamp  **if** (InstanceStatus.***UP***.equals(registrant.getStatus())) {  lease.serviceUp();  }  registrant.setActionType(ActionType.***ADDED***);  recentlyChangedQueue.add(**new** RecentlyChangedItem(lease));  registrant.setLastUpdatedTimestamp();  invalidateCache(registrant.getAppName(), registrant.getVIPAddress(), registrant.getSecureVipAddress());  ***logger***.info("Registered instance {}/{} with status {} (replication={})",  registrant.getAppName(), registrant.getId(), registrant.getStatus(), isReplication);  } **finally** {  read.unlock();  }  } |



### 7.2.3 重要的类

1 DiscoveryClient(eurek-client) 里面有个register 方法来完成注册的构成

2 eureka-client 使用AbstractJerseyEurekaHttpClient 对Eureka-server发起了请求

3 InstanceRegistry(eureka-server)里面有个register 来让客户端完成注册的

4 InstanceRegistry 里面的register 方法会调用父类AbstractInstanceRegistry里面的register 方法，该方法的本质就是往map 集合里面写入了注册信息

5(\*\*) Eureka-Server 里面的Map 集合

Map<String,Map<String,Instance>>

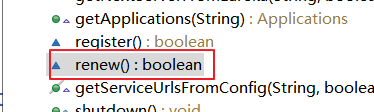
|  |
| --- |
| **private** **final** ConcurrentHashMap<String, Map<String, Lease<InstanceInfo>>> registry  = **new** ConcurrentHashMap<String, Map<String, Lease<InstanceInfo>>>(); |

## 7.3 服务的续约

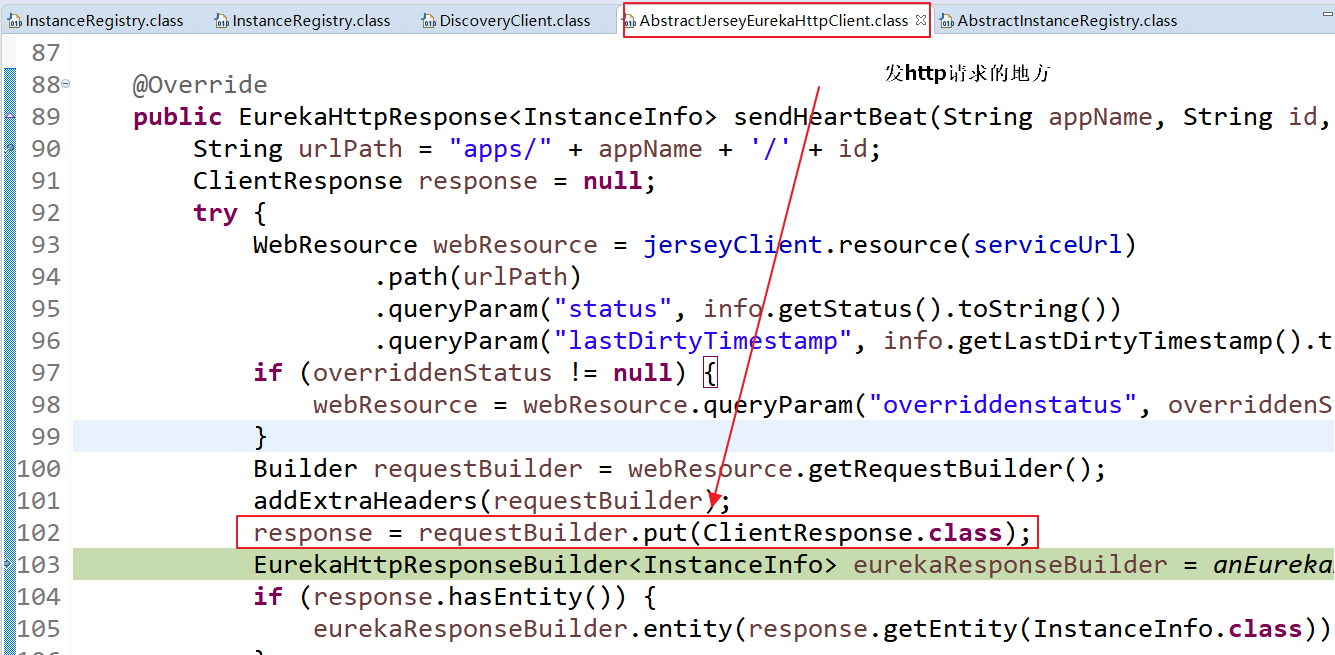
### 7.3.1 eureka-client

对eureka-server 发起续约请求

DiscoveryClient

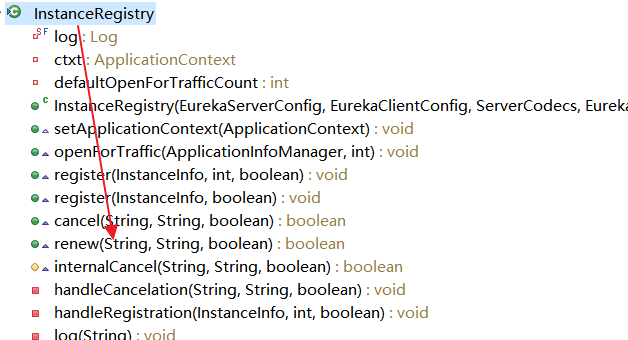


|  |
| --- |
| **boolean** renew() {  EurekaHttpResponse<InstanceInfo> httpResponse;  **try** {  httpResponse =  // 发起续约的http请求  eurekaTransport.registrationClient.sendHeartBeat(instanceInfo.getAppName(), instanceInfo.getId(), instanceInfo, **null**);  ***logger***.debug(***PREFIX*** + "{} - Heartbeat status: {}", appPathIdentifier, httpResponse.getStatusCode());  **if** (httpResponse.getStatusCode() == Status.***NOT\_FOUND***.getStatusCode()) {  REREGISTER\_COUNTER.increment();  ***logger***.info(***PREFIX*** + "{} - Re-registering apps/{}", appPathIdentifier, instanceInfo.getAppName());  **long** timestamp = instanceInfo.setIsDirtyWithTime();  **boolean** success = register();  **if** (success) {  instanceInfo.unsetIsDirty(timestamp);  }  **return** success;  }  **return** httpResponse.getStatusCode() == Status.***OK***.getStatusCode();  } **catch** (Throwable e) {  ***logger***.error(***PREFIX*** + "{} - was unable to send heartbeat!", appPathIdentifier, e);  **return** **false**;  }  } |



### 7.3.2eureka-server 处理客户端的续约请求

InstanceRegistry

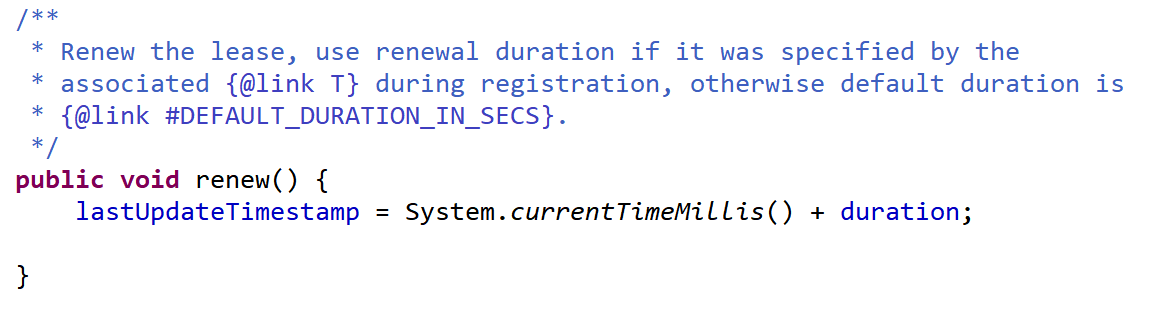


会首先进入：

InstanceRegistry 里面的-renew 方法->xxx->AbstractInstanceRegistry 里面的续约renew 方法

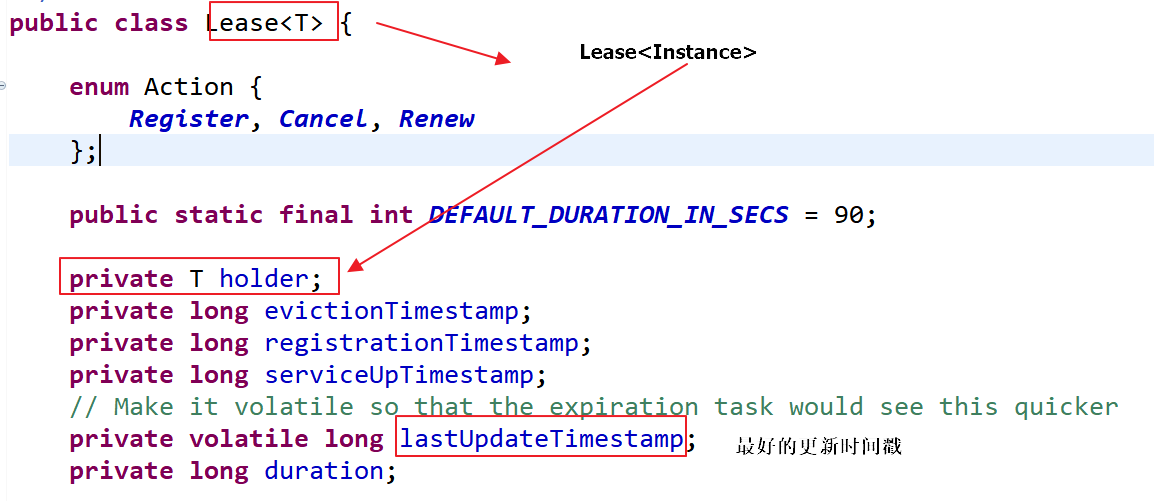
|  |
| --- |
| **public** **boolean** renew(String appName, String id, **boolean** isReplication) {  ***RENEW***.increment(isReplication);  // 获取该服务的实例列表（Map）  Map<String, Lease<InstanceInfo>> gMap = registry.get(appName);  Lease<InstanceInfo> leaseToRenew = **null**;  **if** (gMap != **null**) {  // 通过实例的id 得到要续约的实例  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**;  } **else** {  // 得到续约的实例  InstanceInfo instanceInfo = leaseToRenew.getHolder();  **if** (instanceInfo != **null**) {  // touchASGCache(instanceInfo.getASGName());  InstanceStatus overriddenInstanceStatus = **this**.getOverriddenInstanceStatus(  instanceInfo, leaseToRenew, isReplication);  **if** (overriddenInstanceStatus == InstanceStatus.***UNKNOWN***) {  ***logger***.info("Instance status UNKNOWN possibly due to deleted override for instance {}"  + "; re-register required", instanceInfo.getId());  ***RENEW\_NOT\_FOUND***.increment(isReplication);  **return** **false**;  }  **if** (!instanceInfo.getStatus().equals(overriddenInstanceStatus)) {  ***logger***.info(  "The instance status {} is different from overridden instance status {} for instance {}. "  + "Hence setting the status to overridden status", instanceInfo.getStatus().name(),  instanceInfo.getOverriddenStatus().name(),  instanceInfo.getId());  instanceInfo.setStatusWithoutDirty(overriddenInstanceStatus);  }  }  renewsLastMin.increment();  // 最好才正常的发生了续约  leaseToRenew.renew();  **return** **true**;  }  } |

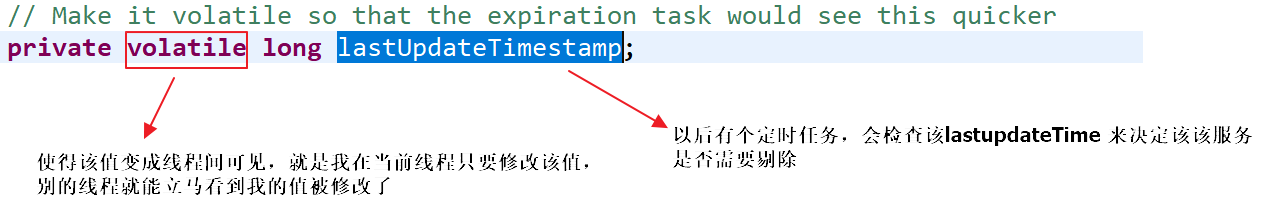
续约的本质是修改最后的 修改时间



duration：代表注册中心最长的忍耐时间：

并不是30s没有续约就里面剔除，而是30 +duration 期间内没有续约，才剔除服务



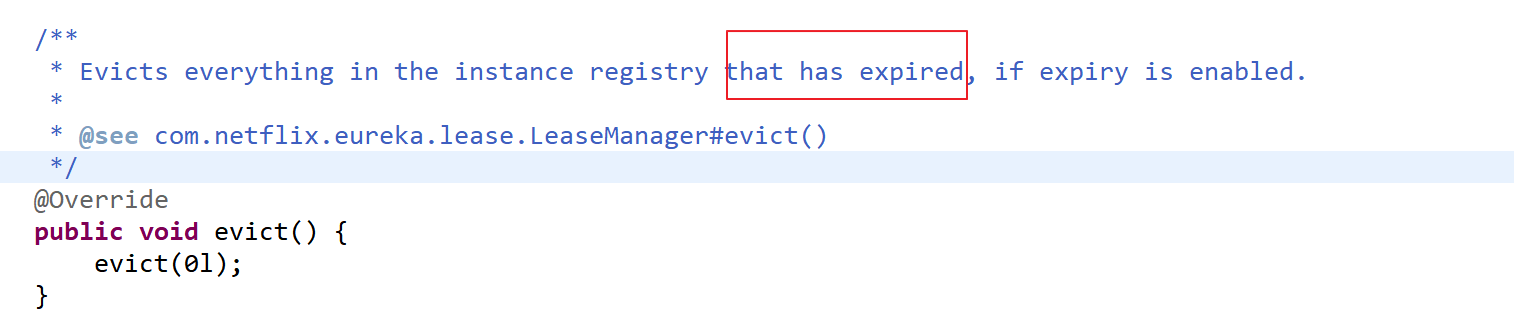


## 7.4 服务的剔除

### 7.4.1 服务剔除的实现

当eureka-server 发生有的实例没有续约超过一段时间，则将该服务从注册列表里面删除它

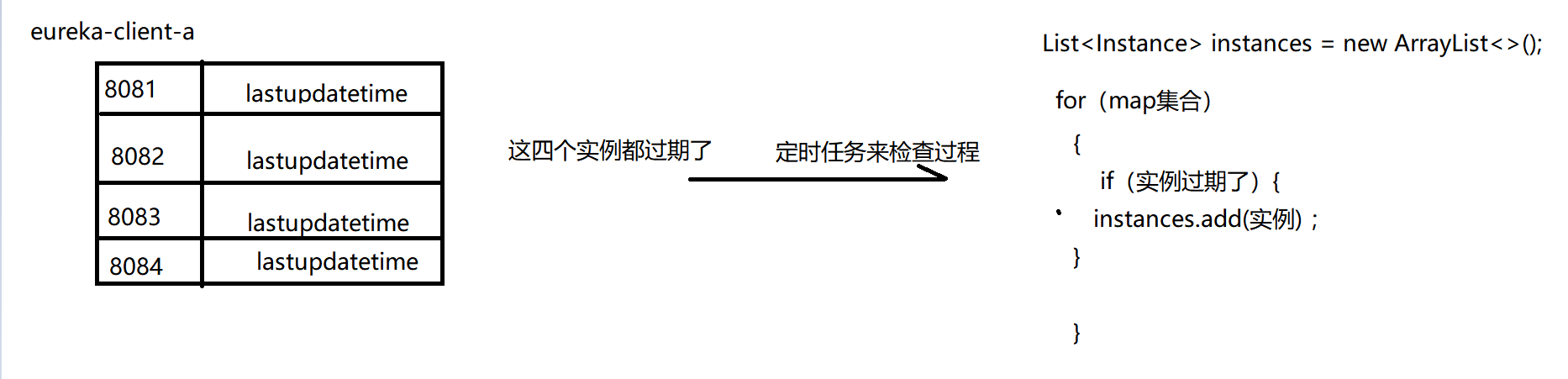
AbstractInstanceRegistry：



|  |
| --- |
| **public** **void** evict(**long** additionalLeaseMs) {  ***logger***.debug("Running the evict task");  **if** (!isLeaseExpirationEnabled()) {  ***logger***.debug("DS: lease expiration is currently disabled.");  **return**;  }  // We collect first all expired items, to evict them in random order. For large eviction sets,  // if we do not that, we might wipe out whole apps before self preservation kicks in. By randomizing it,  // 新建一个过期实例的列表  List<Lease<InstanceInfo>> expiredLeases = **new** ArrayList<>();  // 开始循环registry 来完成对过期实例的检测工作  **for** (Entry<String, Map<String, Lease<InstanceInfo>>> groupEntry : registry.entrySet()) {  Map<String, Lease<InstanceInfo>> leaseMap = groupEntry.getValue();  **if** (leaseMap != **null**) {  **for** (Entry<String, Lease<InstanceInfo>> leaseEntry : leaseMap.entrySet()) {  Lease<InstanceInfo> lease = leaseEntry.getValue();  // 判断该实例是否过期  **if** (lease.isExpired(additionalLeaseMs) && lease.getHolder() != **null**) {  // 将过期的实例放在 过期实例的列表里面  expiredLeases.add(lease);  }  }  }  }  **int** registrySize = (**int**) getLocalRegistrySize();  **int** registrySizeThreshold = (**int**) (registrySize \* serverConfig.getRenewalPercentThreshold());  **int** evictionLimit = registrySize - registrySizeThreshold;  **int** toEvict = Math.*min*(expiredLeases.size(), evictionLimit);  **if** (toEvict > 0) {  ***logger***.info("Evicting {} items (expired={}, evictionLimit={})", toEvict, expiredLeases.size(), evictionLimit);  Random random = **new** Random(System.*currentTimeMillis*());  **for** (**int** i = 0; i < toEvict; i++) {  // Pick a random item (Knuth shuffle algorithm)  **int** next = i + random.nextInt(expiredLeases.size() - i);  // 交互位置，达到一个公平性  Collections.*swap*(expiredLeases, i, next);  // 在获取一个马上要死的  Lease<InstanceInfo> lease = expiredLeases.get(i);  String appName = lease.getHolder().getAppName();  String id = lease.getHolder().getId();  ***EXPIRED***.increment();  ***logger***.warn("DS: Registry: expired lease for {}/{}", appName, id);  // 这一个不完成死的过程  internalCancel(appName, id, **false**);  }  }  } |

实际让它死的代码：

|  |
| --- |
| **protected** **boolean** internalCancel(String appName, String id, **boolean** isReplication) {  **try** {  read.lock();  ***CANCEL***.increment(isReplication);  Map<String, Lease<InstanceInfo>> gMap = registry.get(appName);  Lease<InstanceInfo> leaseToCancel = **null**;  **if** (gMap != **null**) {  // 让它死  leaseToCancel = gMap.remove(id);  }  **synchronized** (recentCanceledQueue) {  recentCanceledQueue.add(**new** Pair<Long, String>(System.*currentTimeMillis*(), appName + "(" + id + ")"));  }  InstanceStatus instanceStatus = overriddenInstanceStatusMap.remove(id);  **if** (instanceStatus != **null**) {  ***logger***.debug("Removed instance id {} from the overridden map which has value {}", id, instanceStatus.name());  }  **if** (leaseToCancel == **null**) {  ***CANCEL\_NOT\_FOUND***.increment(isReplication);  ***logger***.warn("DS: Registry: cancel failed because Lease is not registered for: {}/{}", appName, id);  **return** **false**;  } **else** {  leaseToCancel.cancel();  InstanceInfo instanceInfo = leaseToCancel.getHolder();  String vip = **null**;  String svip = **null**;  **if** (instanceInfo != **null**) {  instanceInfo.setActionType(ActionType.***DELETED***);  recentlyChangedQueue.add(**new** RecentlyChangedItem(leaseToCancel));  instanceInfo.setLastUpdatedTimestamp();  vip = instanceInfo.getVIPAddress();  svip = instanceInfo.getSecureVipAddress();  }  invalidateCache(appName, vip, svip);  ***logger***.info("Cancelled instance {}/{} (replication={})", appName, id, isReplication);  **return** **true**;  }  } **finally** {  read.unlock();  }  } |



实例是否过期的方法的判断：

|  |
| --- |
| // 判断该实例是否过期  **if** (lease.isExpired(additionalLeaseMs){} |

|  |
| --- |
| \*/  **public** **boolean** isExpired(**long** additionalLeaseMs) {  **return** (evictionTimestamp > 0 || System.*currentTimeMillis*() > (lastUpdateTimestamp + duration + additionalLeaseMs));  } |

### 7.4.2 服务的剔除涉及了那个重要的技能点？

怎么删除一个集合里面过期的数据？

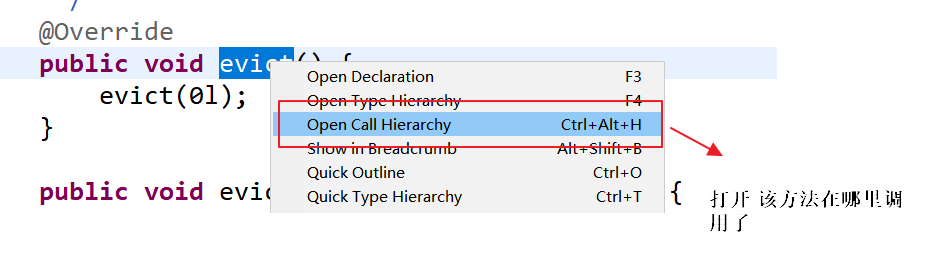
Redis 怎么清除过期的key

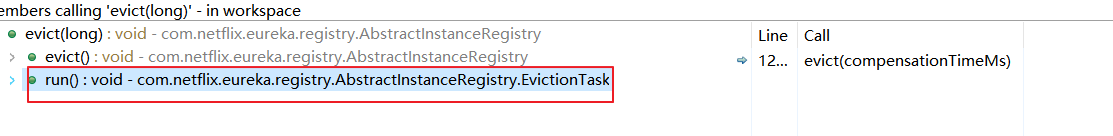
1 定时（k-thread）

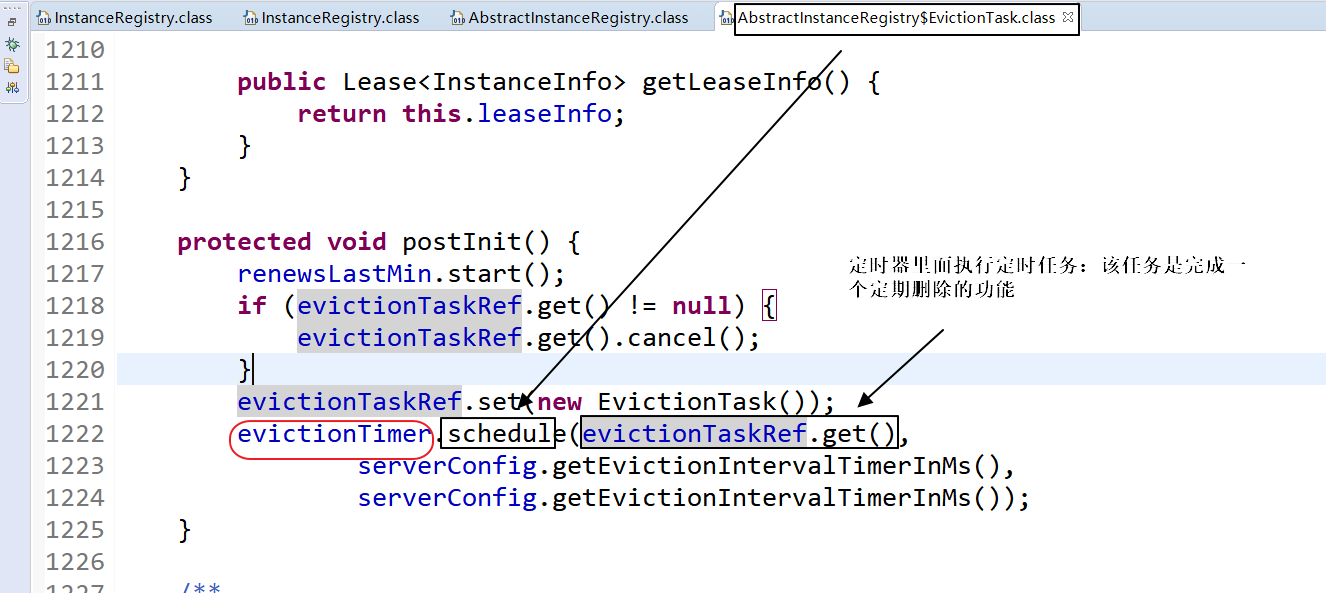
2 惰性 （在再次访问该key时有作用）

3 定期 （使用一个线程来完成清除任务）

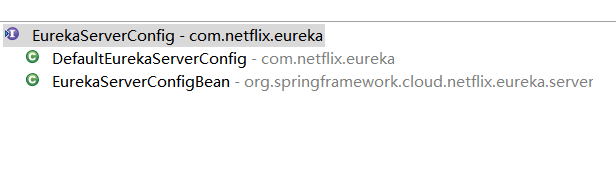
定期（实时性差） + 惰性

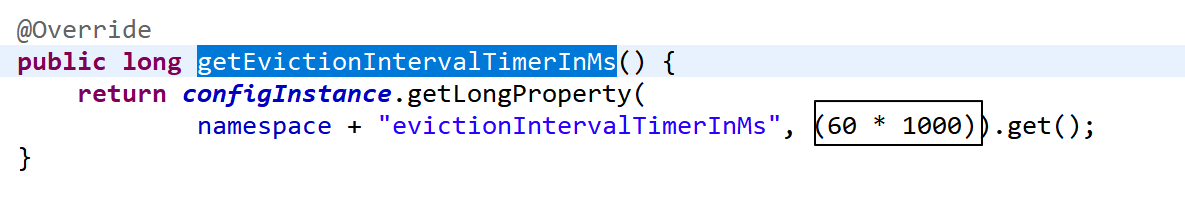






发现该定时任务默认为：





## 7.5 服务的下线

当我们的eureka-client （项目被关闭时，它不会里面关闭，而是做好了‘善后’工作才关、

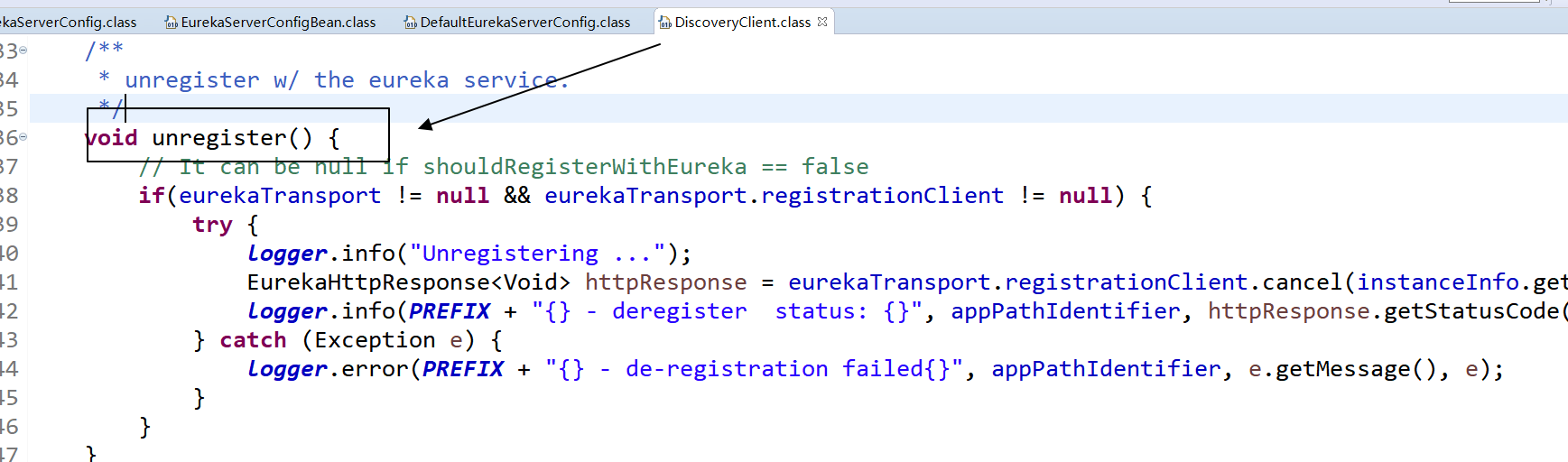
闭）

善后：服务的下线：当项目被关闭时，eureka-client会再发请求eureka-server ，说我要下线了)

Eureka-client 发请求给eureka-server ，说下线了

### 7.5.1 eureka-client怎么发请求？

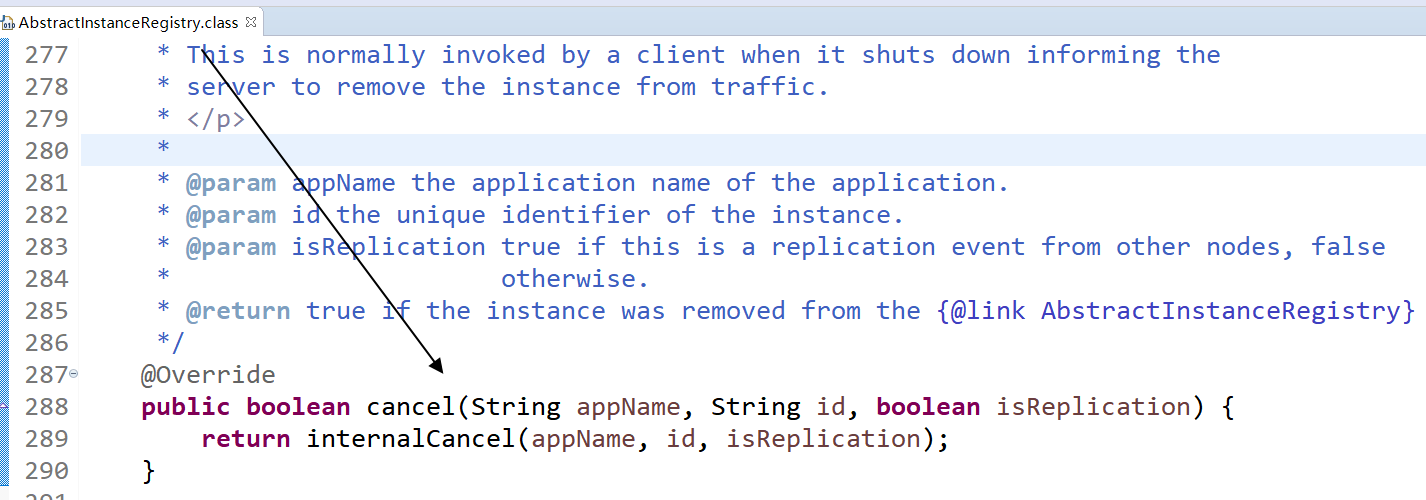
DiscoveryClient



|  |
| --- |
| EurekaHttpResponse<Void> httpResponse = eurekaTransport.registrationClient.cancel(instanceInfo.getAppName(), instanceInfo.getId()); |



### 7.5.2 eureka-server 怎么处理该请求



我们可以看见服务的下线再eureka-server 里面直接调用

|  |
| --- |
| **protected** **boolean** internalCancel(String appName, String id, **boolean** isReplication) {  **try** {  read.lock();  ***CANCEL***.increment(isReplication);  Map<String, Lease<InstanceInfo>> gMap = registry.get(appName);  Lease<InstanceInfo> leaseToCancel = **null**;  **if** (gMap != **null**) {  leaseToCancel = gMap.remove(id);  }  **synchronized** (recentCanceledQueue) {  recentCanceledQueue.add(**new** Pair<Long, String>(System.*currentTimeMillis*(), appName + "(" + id + ")"));  }  InstanceStatus instanceStatus = overriddenInstanceStatusMap.remove(id);  **if** (instanceStatus != **null**) {  ***logger***.debug("Removed instance id {} from the overridden map which has value {}", id, instanceStatus.name());  }  **if** (leaseToCancel == **null**) {  ***CANCEL\_NOT\_FOUND***.increment(isReplication);  ***logger***.warn("DS: Registry: cancel failed because Lease is not registered for: {}/{}", appName, id);  **return** **false**;  } **else** {  leaseToCancel.cancel();  InstanceInfo instanceInfo = leaseToCancel.getHolder();  String vip = **null**;  String svip = **null**;  **if** (instanceInfo != **null**) {  instanceInfo.setActionType(ActionType.***DELETED***);  recentlyChangedQueue.add(**new** RecentlyChangedItem(leaseToCancel));  instanceInfo.setLastUpdatedTimestamp();  vip = instanceInfo.getVIPAddress();  svip = instanceInfo.getSecureVipAddress();  }  invalidateCache(appName, vip, svip);  ***logger***.info("Cancelled instance {}/{} (replication={})", appName, id, isReplication);  **return** **true**;  }  } **finally** {  read.unlock();  }  } |

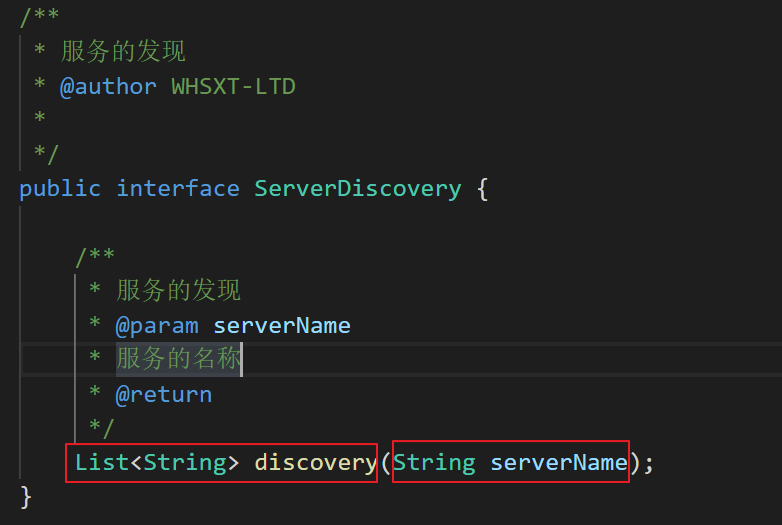
# 服务的发现

## 8.1 什么是服务的发现？

通过服务的名称发现服务的实例！

## 8.2 复习在RPC 里面写的服务发现的代码？

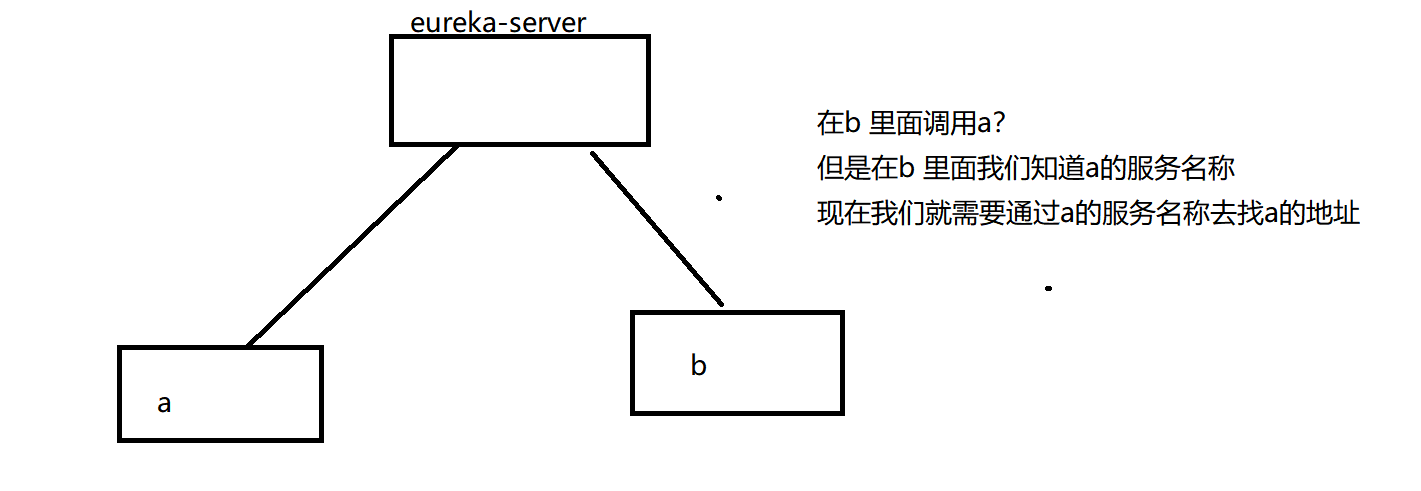
Discovery



实现类：

|  |
| --- |
| /\*\*   \* 使用zk完成服务的发现   \* @author WHSXT-LTD   \*   \*/  public class ZkServerDiscoveryImpl implements ServerDiscovery {        private Map<String,List<String>> serverCache = new HashMap<String,List<String>>();        private static final String ZKSERVERS = "www.yanli.ltd:2181";        private static ZkClient zkclient = null ;      // 初始化zk      static {          zkclient = new ZkClient(ZKSERVERS, 5000, 10000);      }      /\*\*       \* 怎么改进服务的发现       \* 1 参考求职       \*/      @Override      public List<String> discovery(String serverName) {  //  /java开发/110                                                  //  /java开发/120          if(serverCache.containsKey(serverName)) { // 缓存列表里面有              return serverCache.get(serverName);          }            if(!zkclient.exists("/"+serverName)) {              throw new RuntimeException("没有"+serverName+"提供者");          }          List<String> servers = zkclient.getChildren("/"+serverName);            // 缓存的脏读          // 怎么解决缓存脏读问题          // 1 当提供者的节点从zk 里面删除后，我能把map 集合里面的服务的列表改好          zkclient.subscribeChildChanges("/"+serverName, new IZkChildListener() { // 新开线程                @Override              public void handleChildChange(String parentPath, List<String> currentChilds) throws Exception {                  // 当节点删除或新增后，能自动触发改过程  //              parentPath== 父节点 有没有带 /java  //              currentChilds== 父节点里面的子节点                  String serverName = parentPath.substring(1); //   /java -> java                  serverCache.put(serverName, currentChilds);                  System.out.println("通过订阅，我已经解决了缓存的脏读问题"+serverName);              }          });            System.out.println("订阅节点"+serverName+"完成");            serverCache.put(serverName, servers); // 将服务的列表缓存起来 [localhost:7777,localhost:8888,localhost:9999]            return servers;      }  } |

## 8.3 服务发现的演示

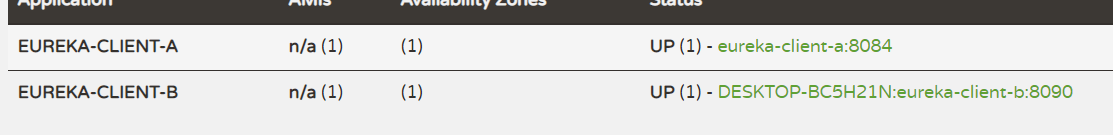


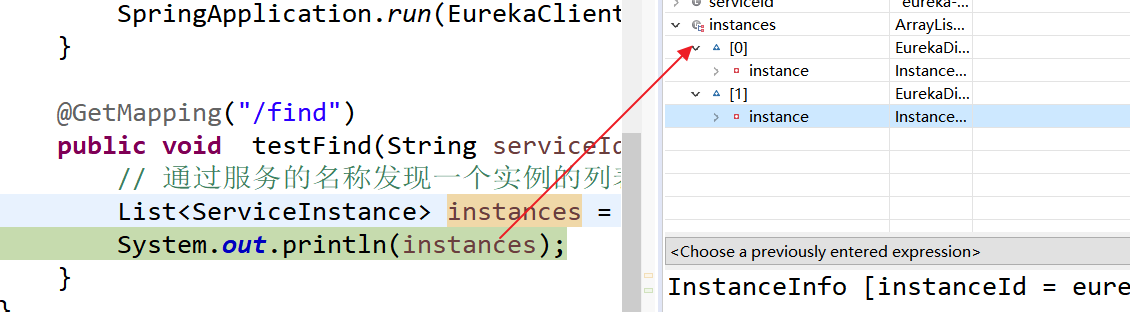
## 8.4 在 b 里面做服务的发现

|  |
| --- |
| @SpringBootApplication  @RestController  **public** **class** EurekaClientBApplication {  /\*\*  \* 该接口的实现类已经在容器里面有一个了  \*/  @Autowired  **private** DiscoveryClient discoveryClient;    **public** **static** **void** main(String[] args) {  SpringApplication.*run*(EurekaClientBApplication.**class**, args);  }  @GetMapping("/find")  **public** **void** testFind(String serviceId) {  // 通过服务的名称发现一个实例的列表  List<ServiceInstance> instances = discoveryClient.getInstances(serviceId);  System.***out***.println(instances);  }  } |

## 8.5 测试服务的发现

确保服务都已经上线了

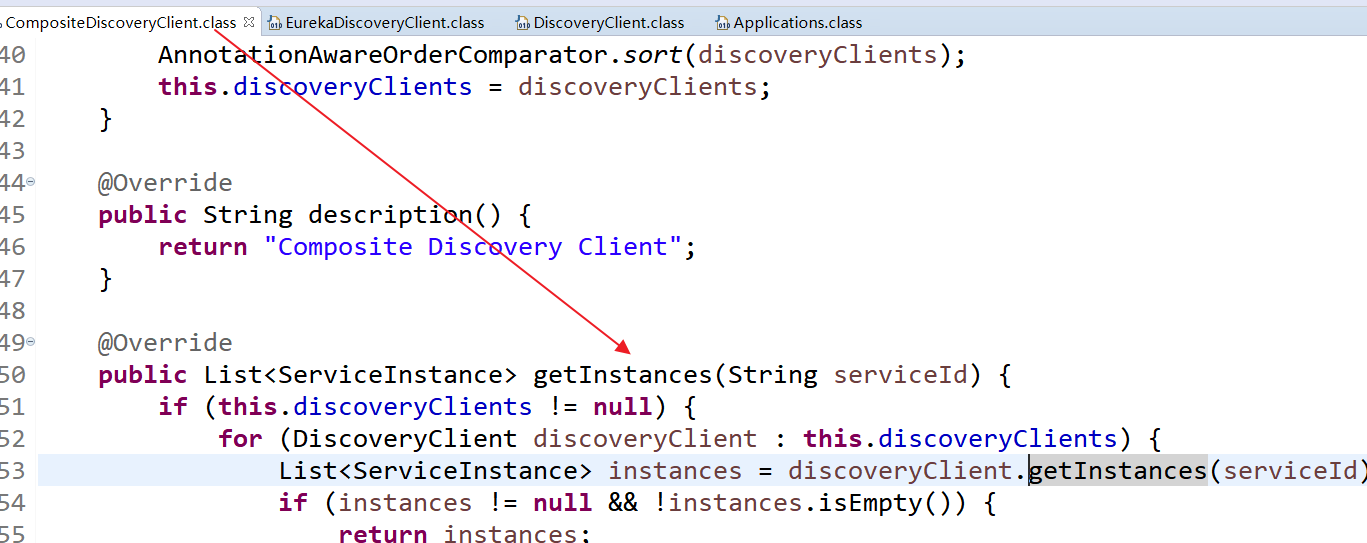




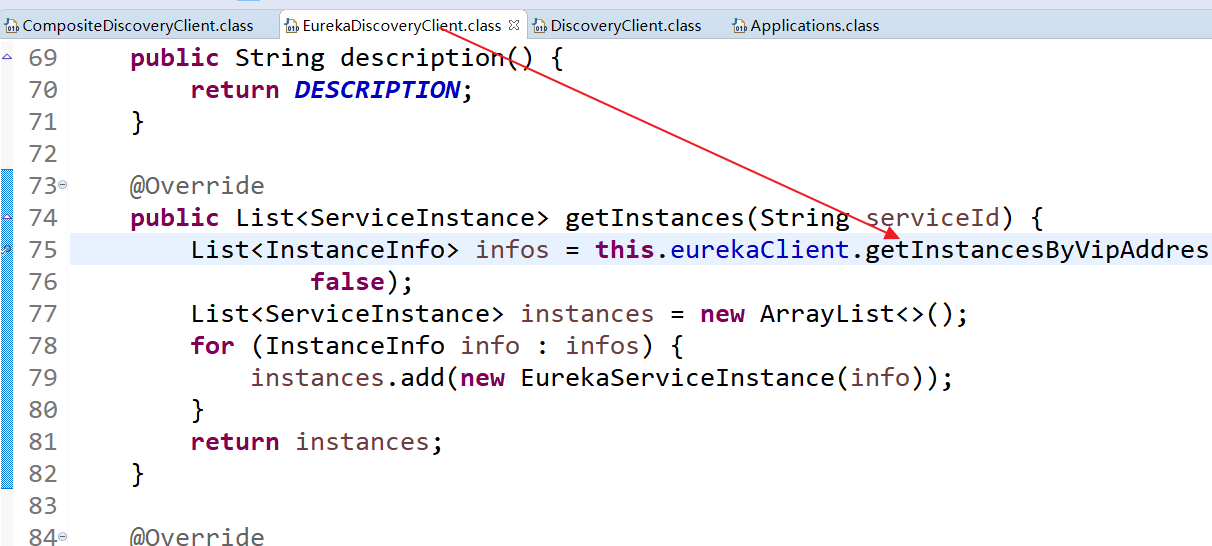
服务的发现已经是ok 的 了

## 8.6 服务发现的源码分析

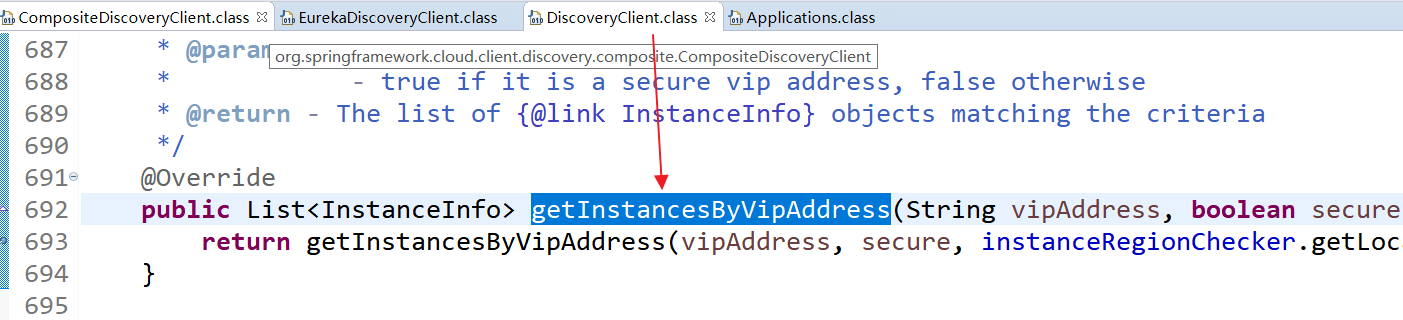
1



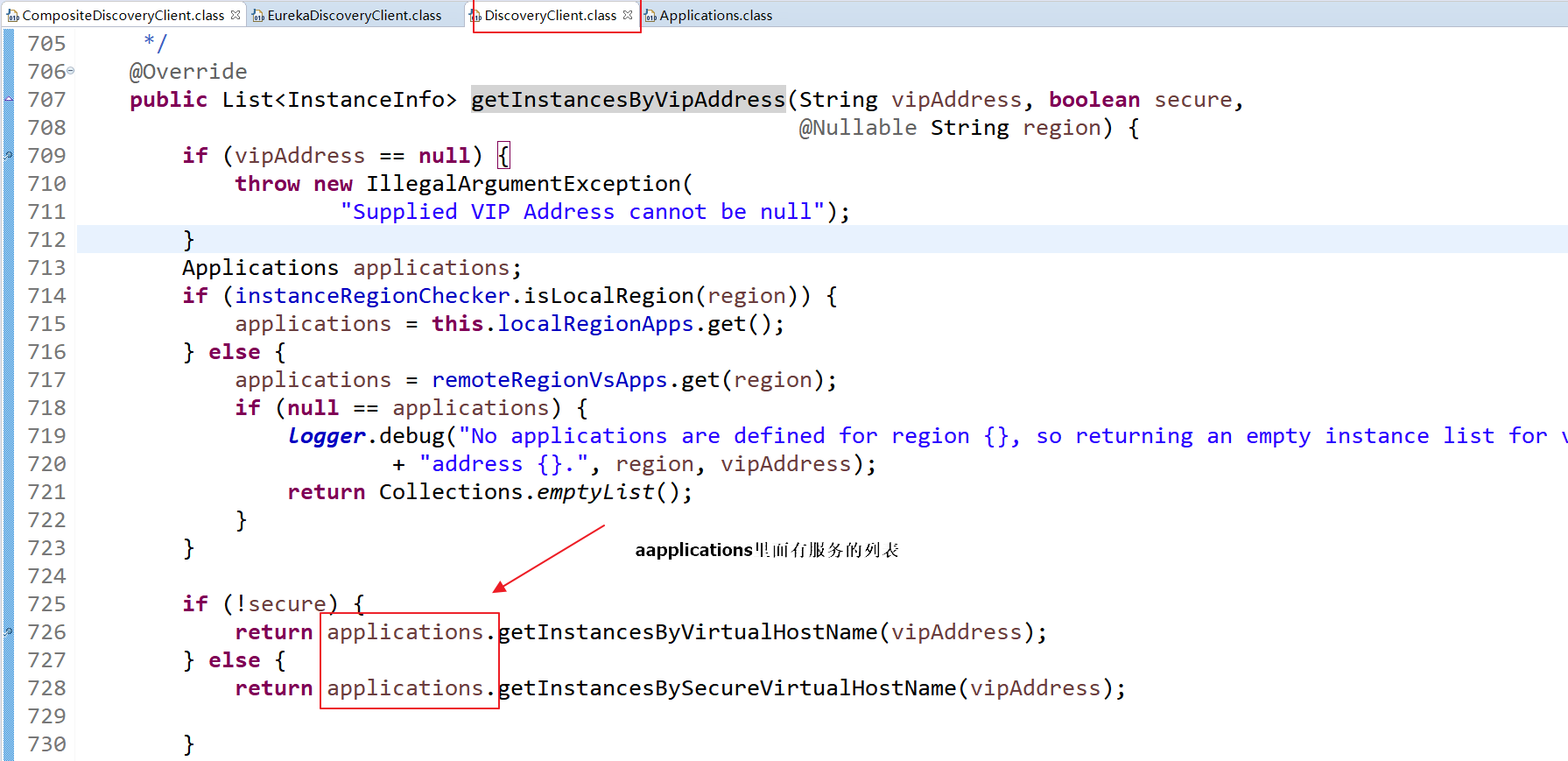
2

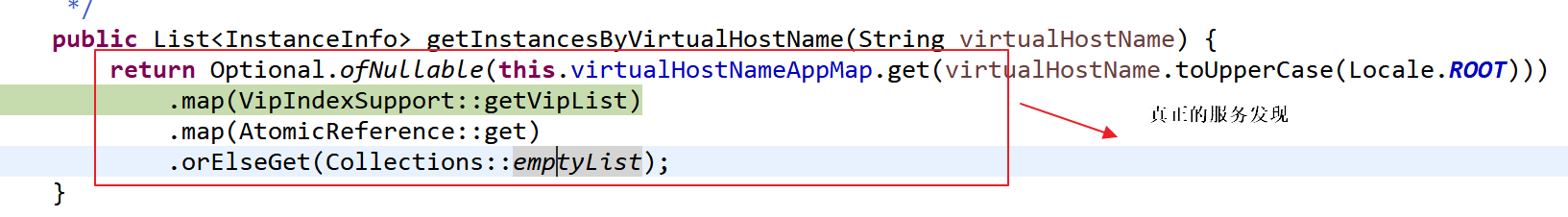


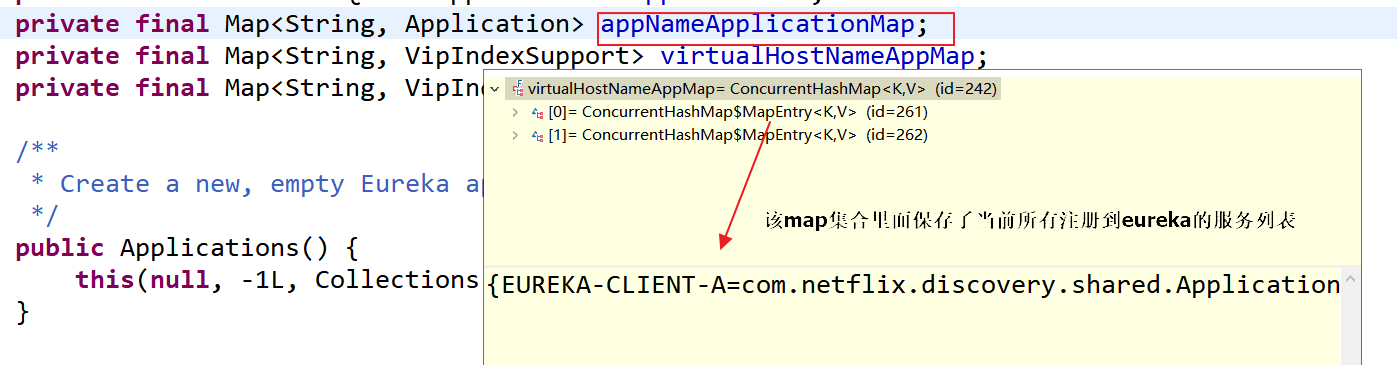
3



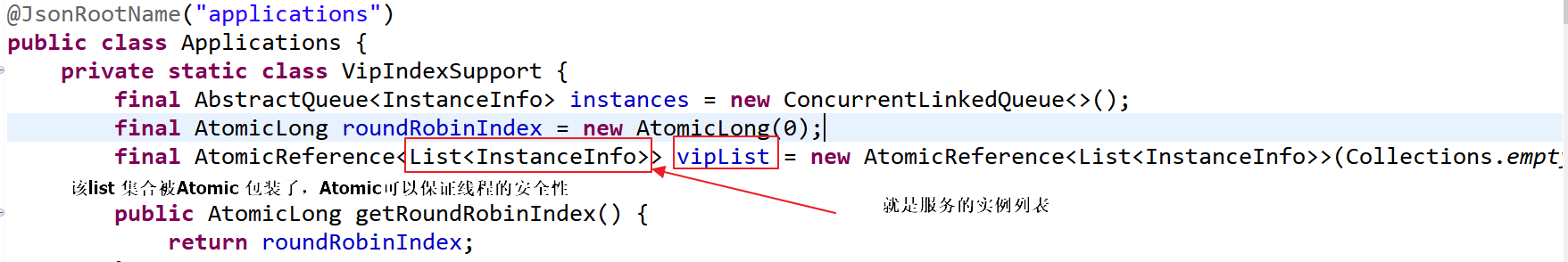
4 在DiscoveryClient 类里面继续调用



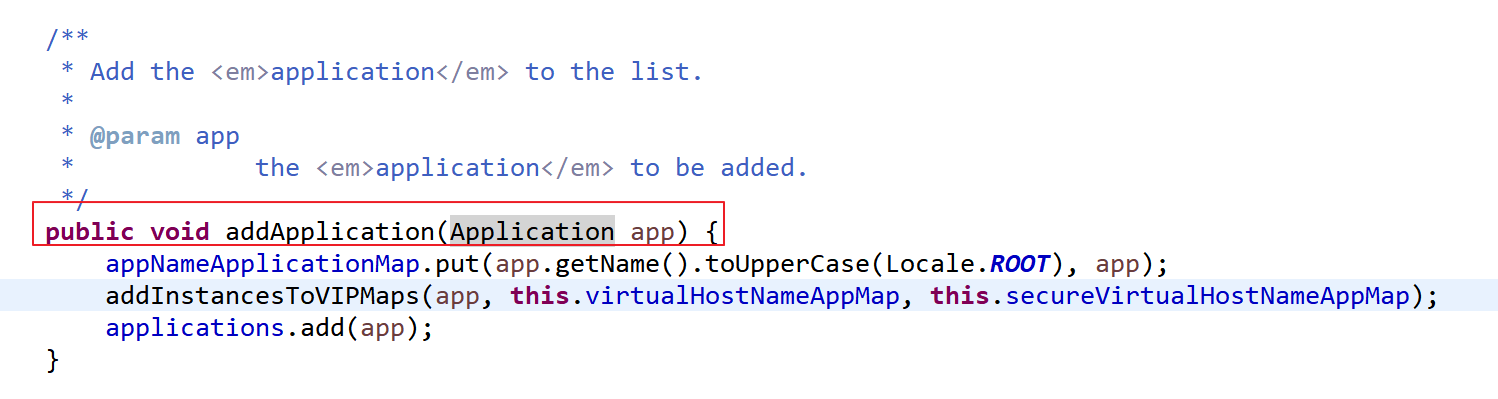




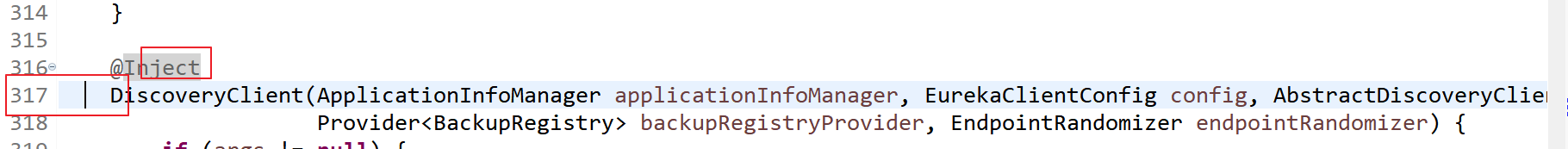
我们还没有做服务的发现，但是该集合里面已经有值了，说明我们的项目启动后，会自动去拉取服务，并且将拉取的服务缓存起来

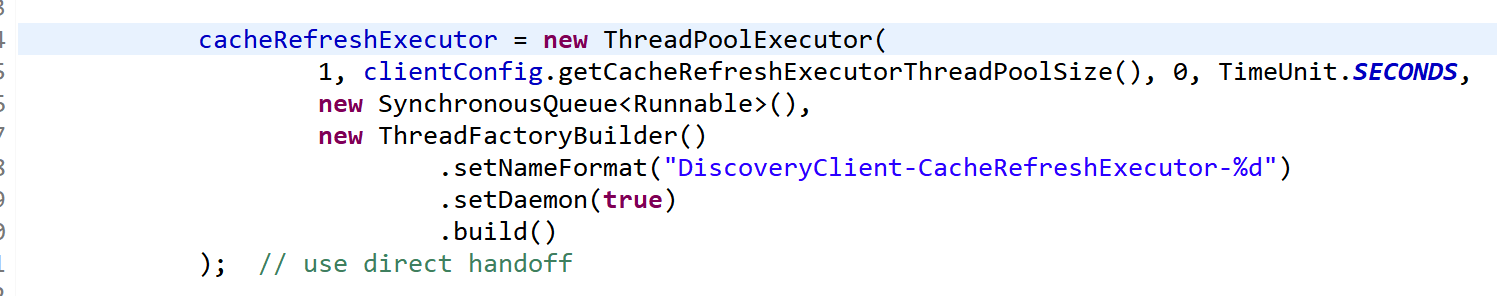


缓存在Applications 里面，到底时什么时候它被放进去的！

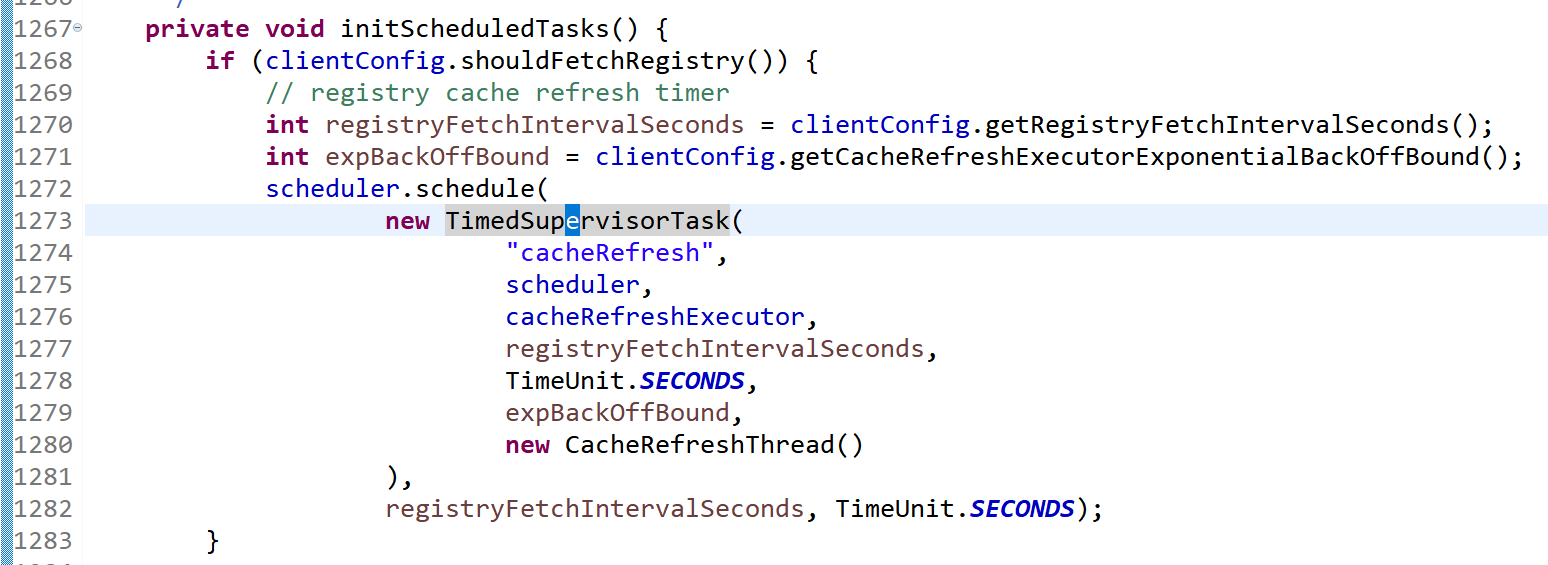


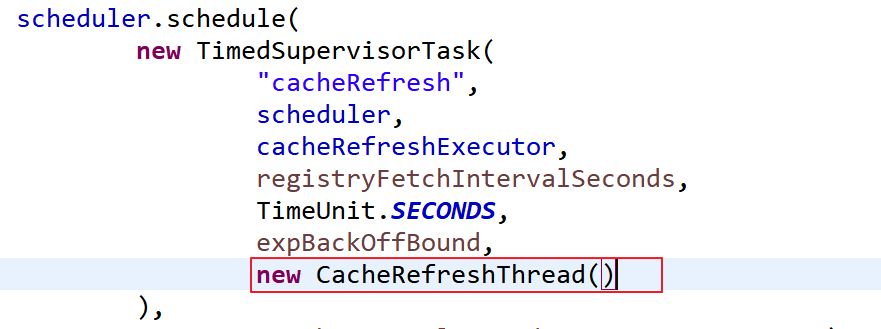
在DiscoveryClient的构造器里面，有个任务：

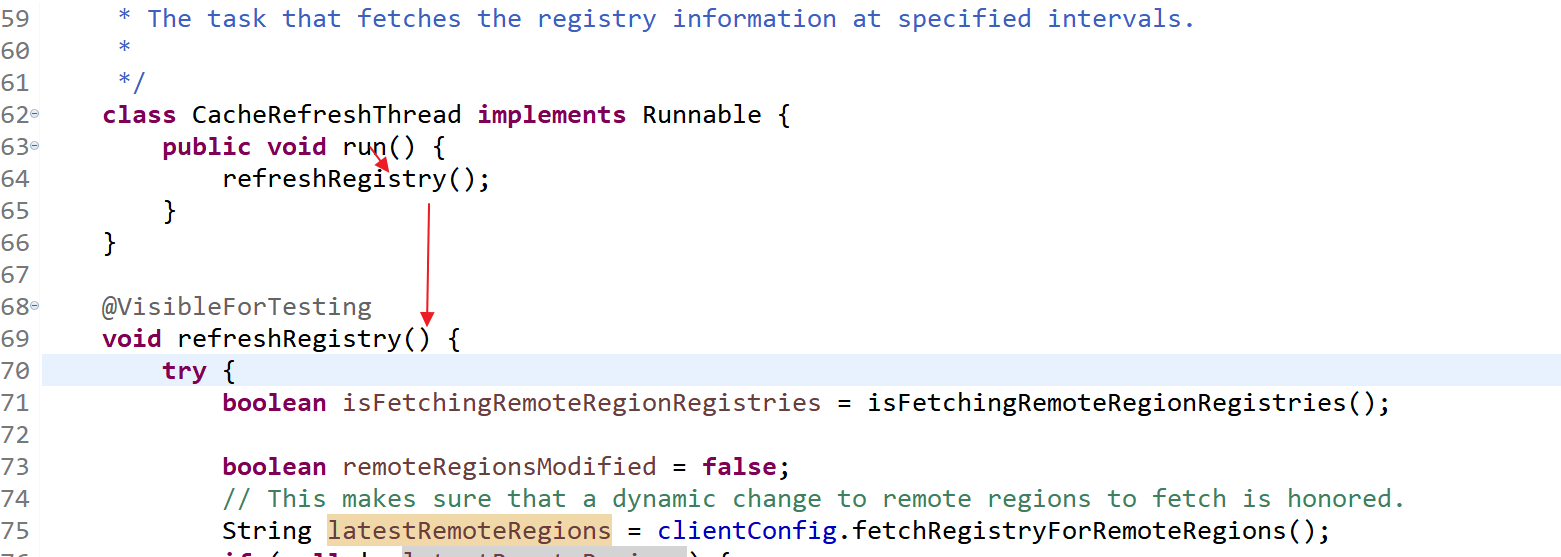
在构造器里面有这个任务的调度线程池：该线程池将用来在服务列表的拉取：



该线程的调度 ，调度的是下面的任务：





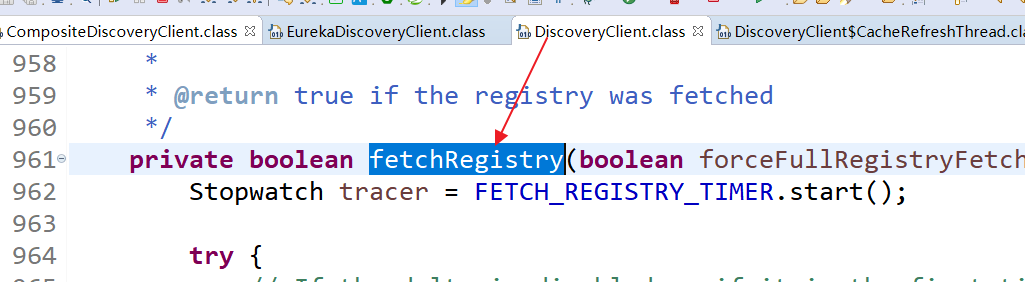


源码的注释：

（晚上讲）

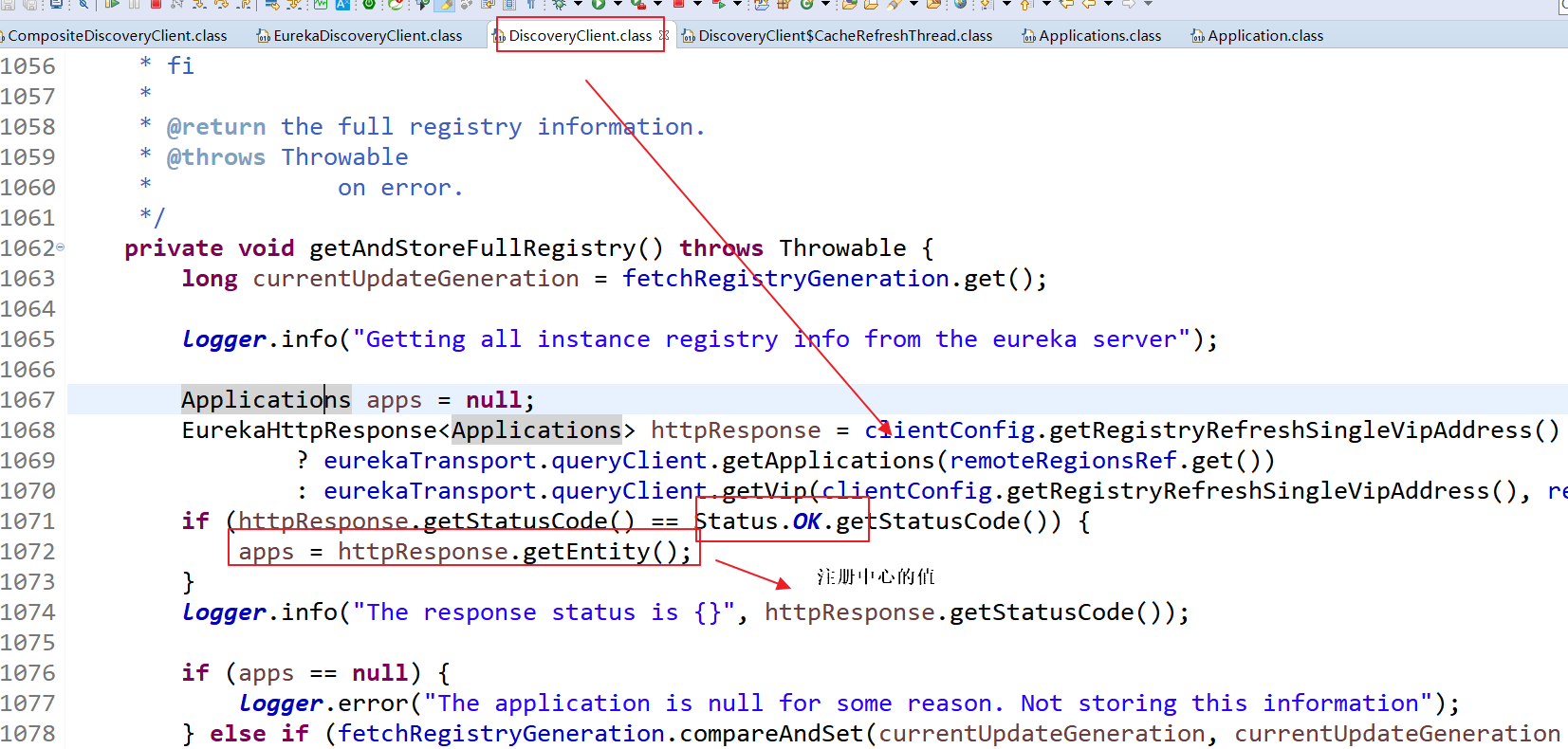
|  |
| --- |
| @VisibleForTesting  **void** refreshRegistry() {  **try** {  **boolean** isFetchingRemoteRegionRegistries = isFetchingRemoteRegionRegistries();  **boolean** remoteRegionsModified = **false**;  // This makes sure that a dynamic change to remote regions to fetch is honored.  String latestRemoteRegions = clientConfig.fetchRegistryForRemoteRegions();  **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();  }  }  // 在在里面真正的执行服务列表的获取  **boolean** success = fetchRegistry(remoteRegionsModified);  **if** (success) {  registrySize = localRegionApps.get().size();  lastSuccessfulRegistryFetchTimestamp = System.*currentTimeMillis*();  }  **if** (***logger***.isDebugEnabled()) {  StringBuilder allAppsHashCodes = **new** StringBuilder();  allAppsHashCodes.append("Local region apps hashcode: ");  allAppsHashCodes.append(localRegionApps.get().getAppsHashCode());  allAppsHashCodes.append(", is fetching remote regions? ");  allAppsHashCodes.append(isFetchingRemoteRegionRegistries);  **for** (Map.Entry<String, Applications> entry : remoteRegionVsApps.entrySet()) {  allAppsHashCodes.append(", Remote region: ");  allAppsHashCodes.append(entry.getKey());  allAppsHashCodes.append(" , apps hashcode: ");  allAppsHashCodes.append(entry.getValue().getAppsHashCode());  }  ***logger***.debug("Completed cache refresh task for discovery. All Apps hash code is {} ",  allAppsHashCodes);  }  } **catch** (Throwable e) {  ***logger***.error("Cannot fetch registry from server", e);  }  } |

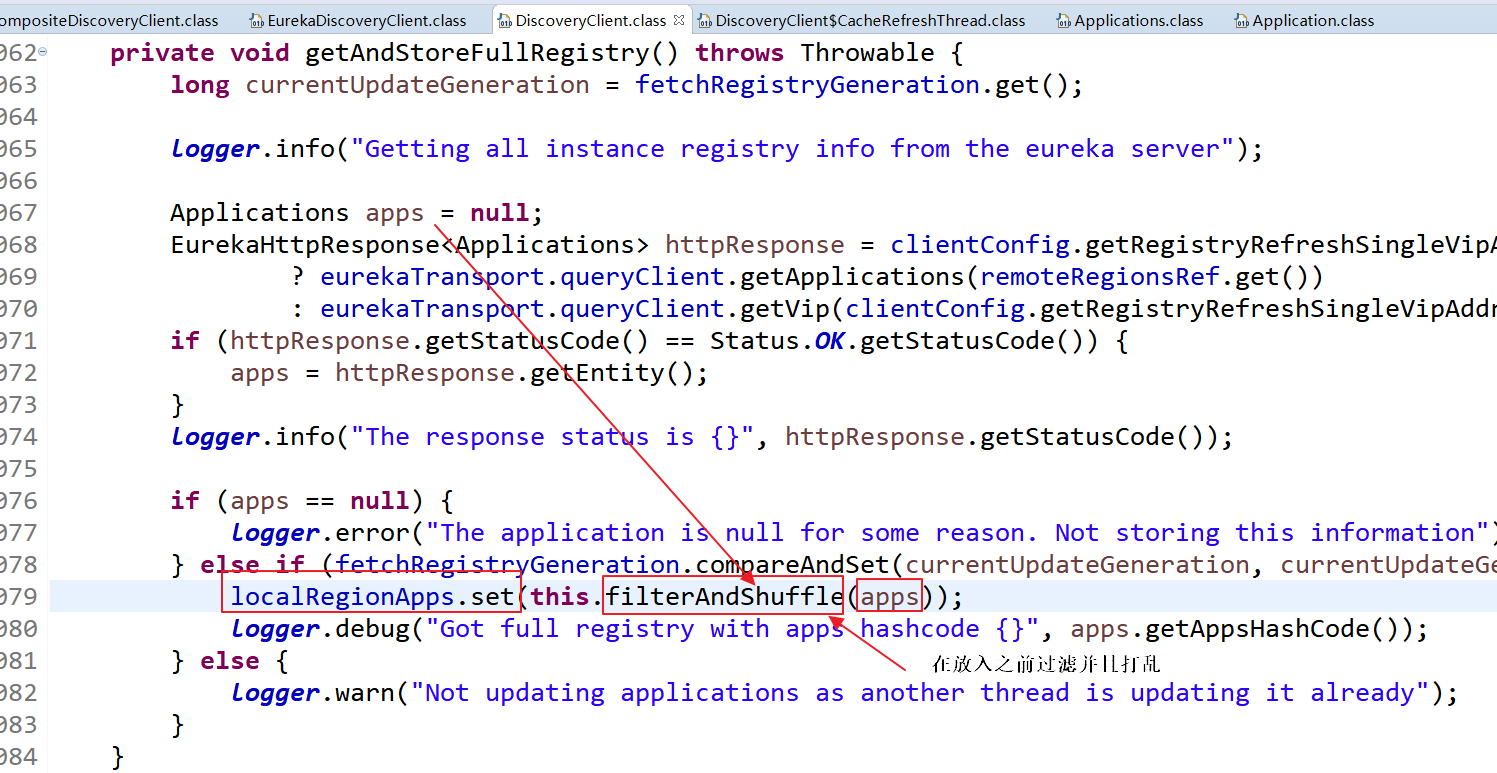
该类的

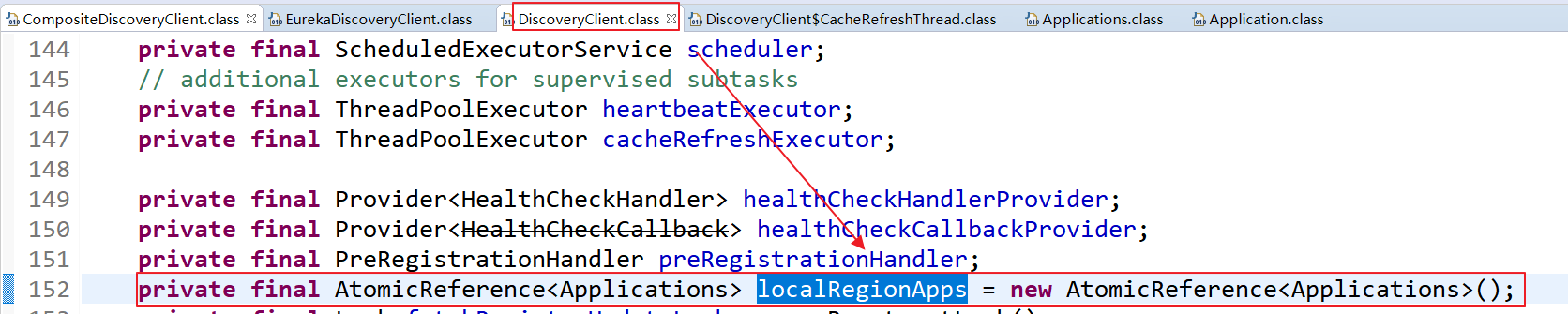


|  |
| --- |
| **private** **boolean** fetchRegistry(**boolean** forceFullRegistryFetch) {  Stopwatch tracer = FETCH\_REGISTRY\_TIMER.start();  **try** {  // If the delta is disabled or if it is the first time, get all  // applications  Applications applications = getApplications();  **if** (clientConfig.shouldDisableDelta()  || (!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));  // 全量的拉取（代表拉取所有的注册中心）  // 当缓存为null 或者它里面的数据为empty时，进行全量的拉取  getAndStoreFullRegistry();  } **else** {  // 增量（只拉取修改的注册中心）  getAndUpdateDelta(applications);  }  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();  }  } |

全量的拉取：



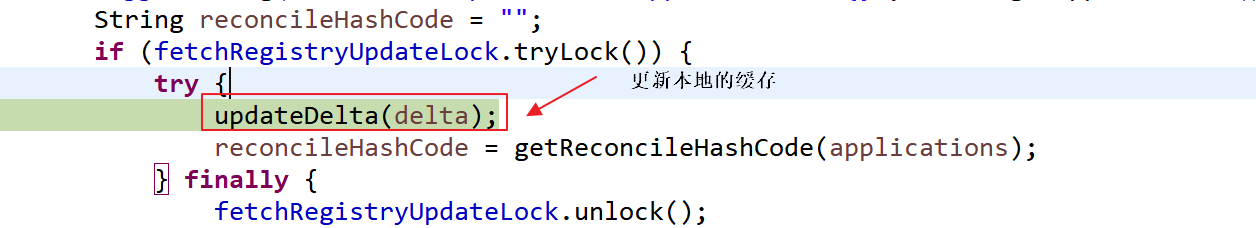


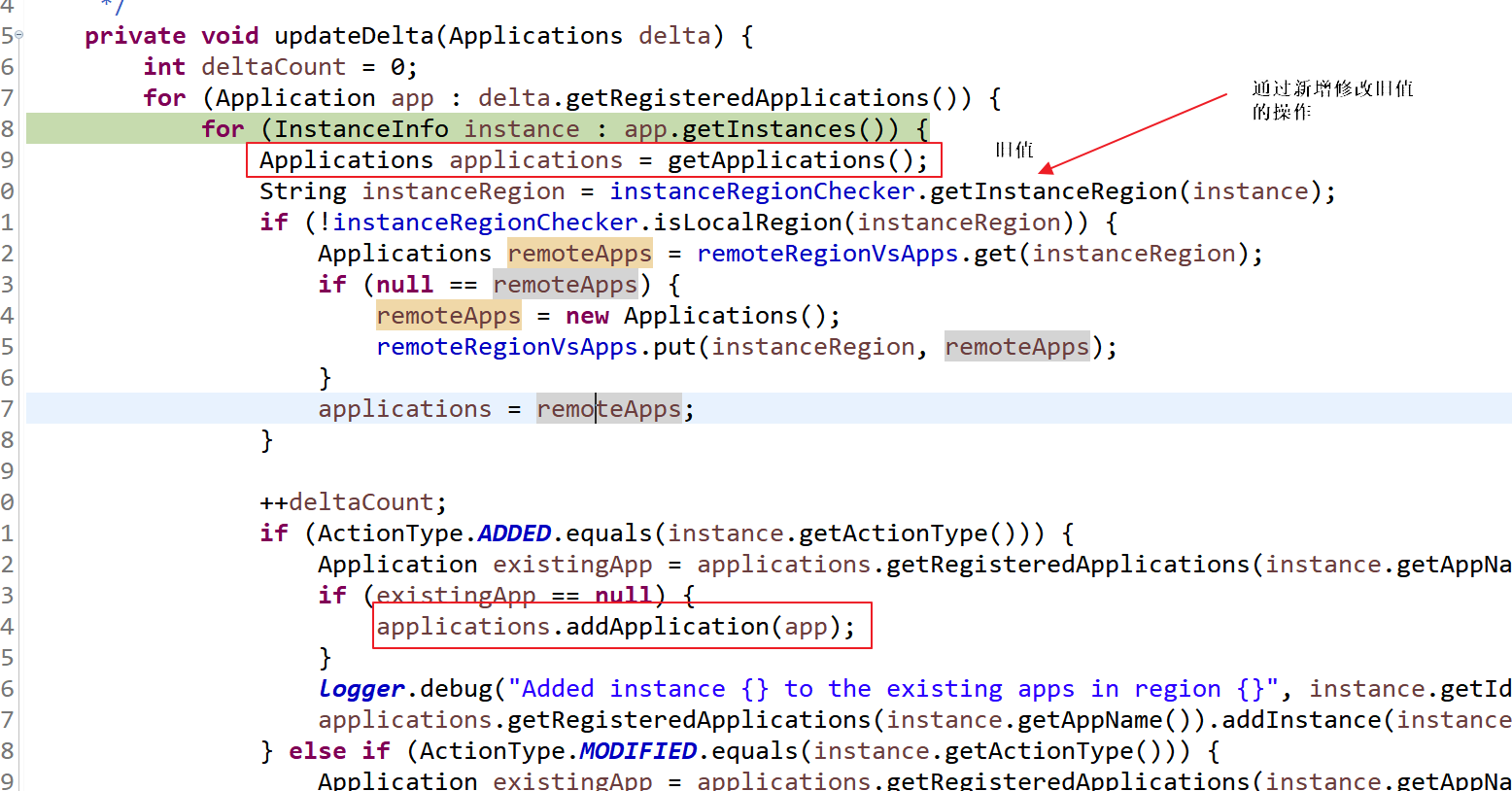


增量的拉取：

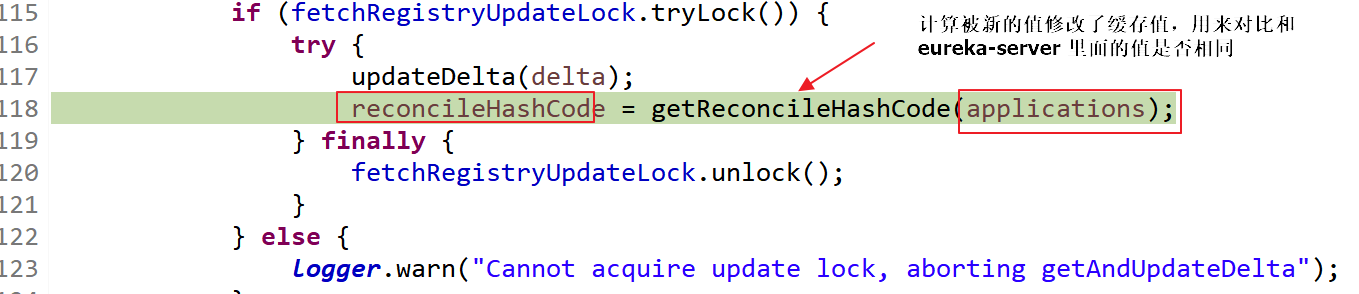
|  |
| --- |
| **private** **void** getAndUpdateDelta(Applications applications) **throws** Throwable {  **long** currentUpdateGeneration = fetchRegistryGeneration.get();  Applications delta = **null**;  // 发起Http 请求来完成一个增量的拉取  EurekaHttpResponse<Applications> httpResponse = eurekaTransport.queryClient.getDelta(remoteRegionsRef.get());  **if** (httpResponse.getStatusCode() == Status.***OK***.getStatusCode()) {  // 在eureka-server 里面修改或新增的集合  delta = httpResponse.getEntity();  }  // 若拉取的集合为null  **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.");  // 全量拉取  getAndStoreFullRegistry();  } **else** **if** (fetchRegistryGeneration.compareAndSet(currentUpdateGeneration, currentUpdateGeneration + 1)) {  ***logger***.debug("Got delta update with apps hashcode {}", delta.getAppsHashCode());  String reconcileHashCode = "";  **if** (fetchRegistryUpdateLock.tryLock()) {  **try** {  // eureka-server 里面变化的集合，通过它里面的值更新本地的缓存  updateDelta(delta);  // 一致性的hashcode值  // 一致性hash 用来校验远程的Eureka-server 集合和本地的eureka-server 集合是否一样  reconcileHashCode = getReconcileHashCode(applications);  } **finally** {  fetchRegistryUpdateLock.unlock();  }  } **else** {  ***logger***.warn("Cannot acquire update lock, aborting getAndUpdateDelta");  }  //若hashcode值相等，则不拉取，否则，在拉取一次  // There is a diff in number of instances for some reason  **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());  }  }  /\*\*  \* Logs the total number of non-filtered instances stored locally.  \*/  **private** **void** logTotalInstances() {  **if** (***logger***.isDebugEnabled()) {  **int** totInstances = 0;  **for** (Application application : getApplications().getRegisteredApplications()) {  totInstances += application.getInstancesAsIsFromEureka().size();  }  ***logger***.debug("The total number of all instances in the client now is {}", totInstances);  }  } |

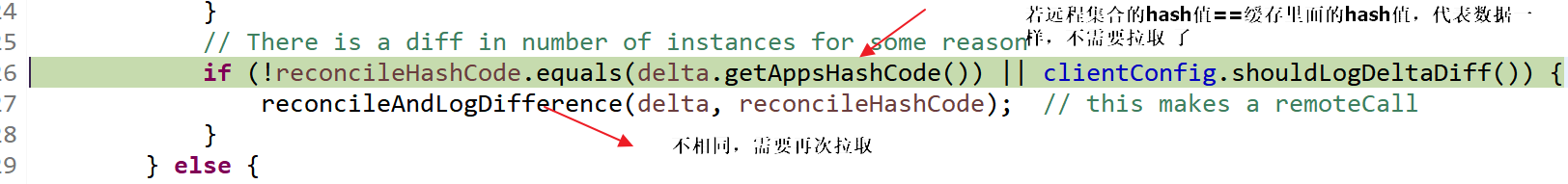
之前本地的缓存只有一个a，现在上线了b ，是一个增量的过程

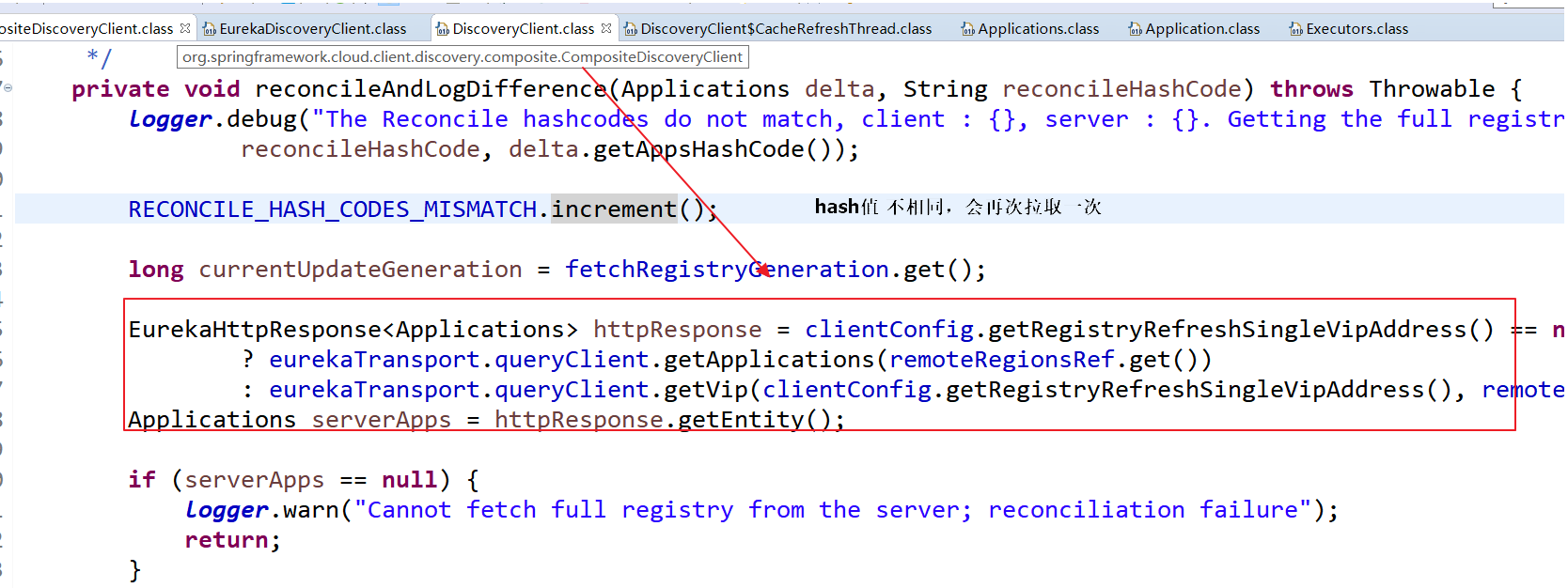




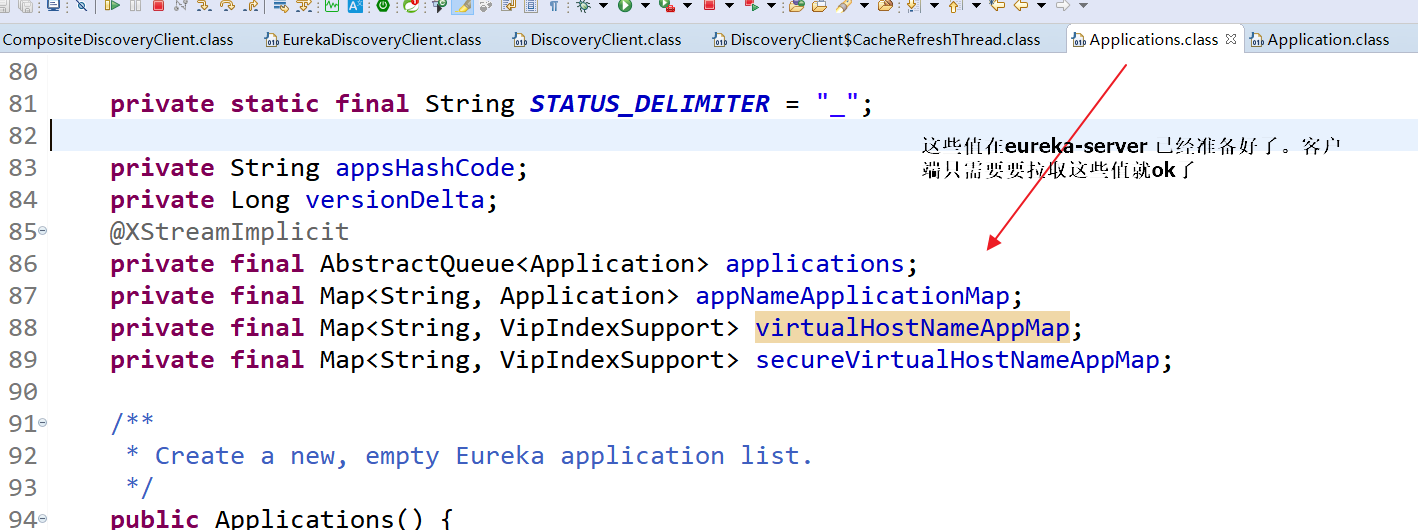
为了保证修改成功：







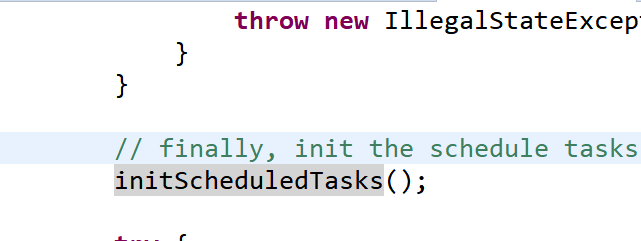


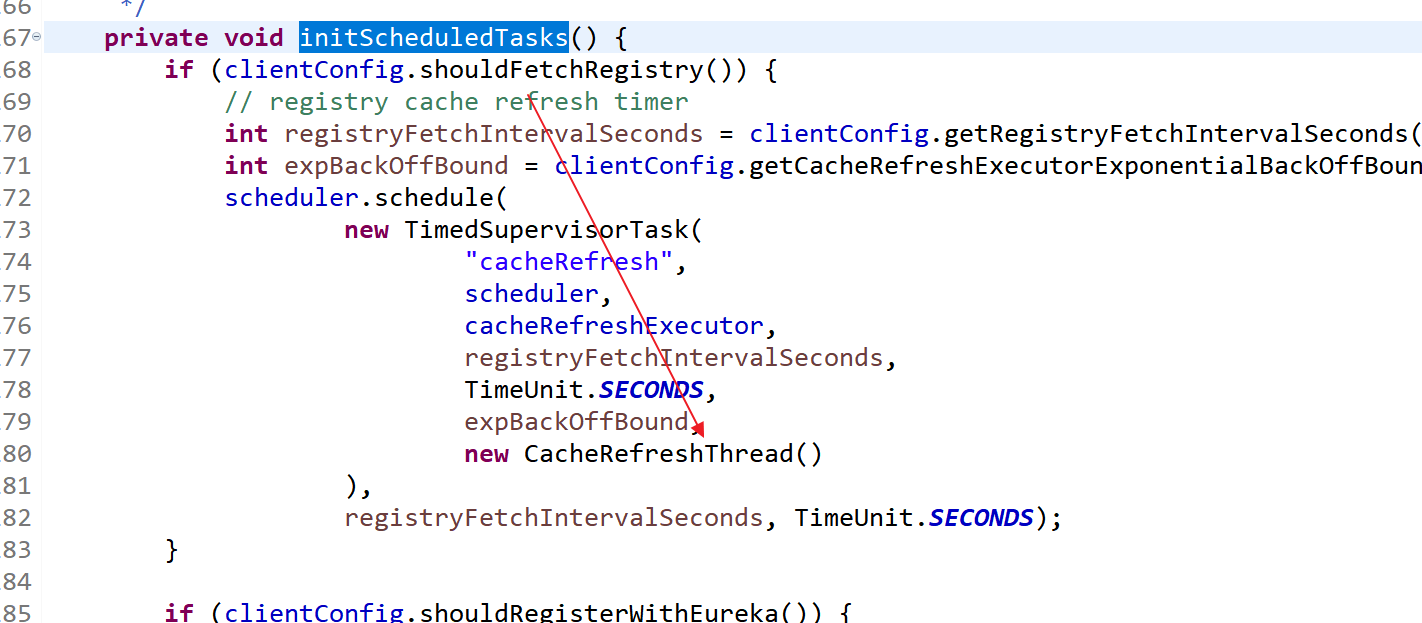


## 8.7 重要的点

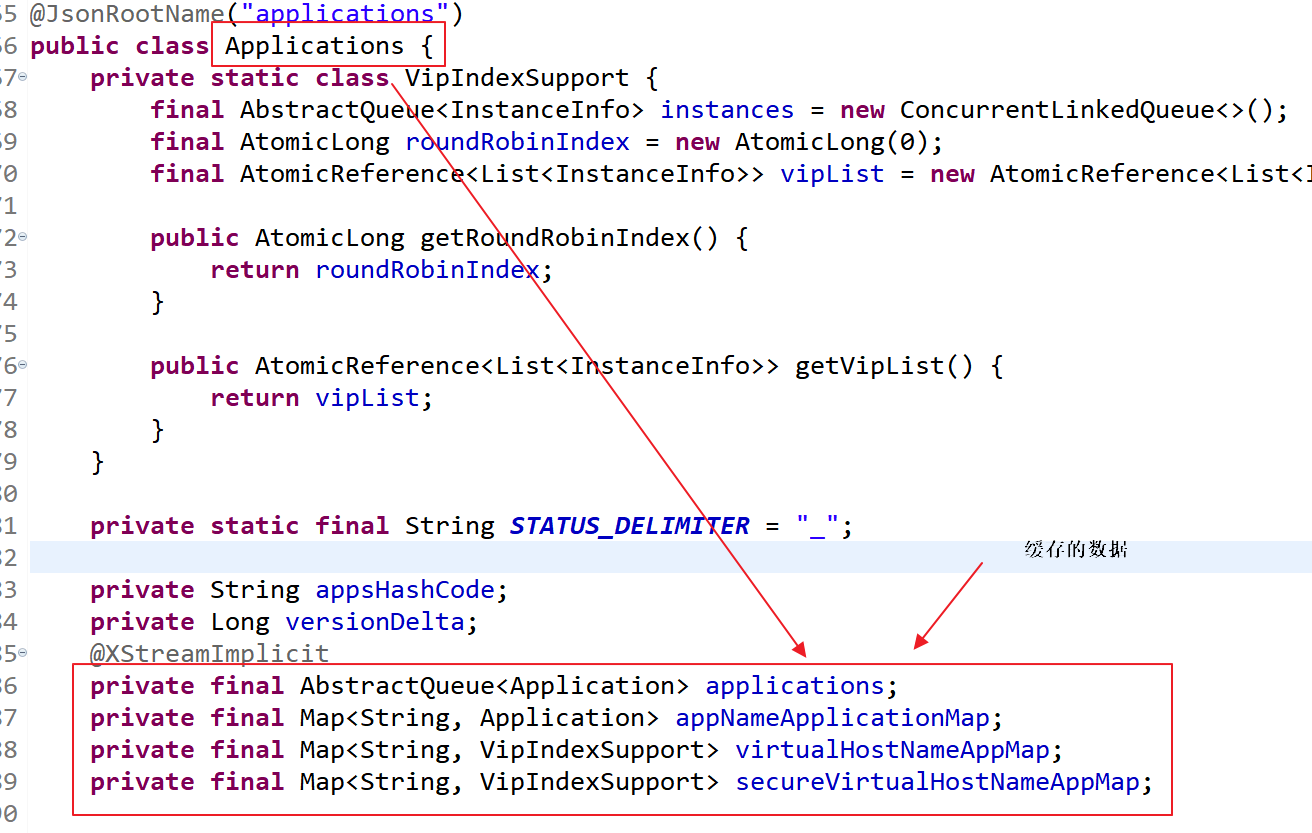
1 服务列表的拉取并不是在服务调用的时候拉取，而是在项目启动时就有定时任务去拉取了

在DiscoveryClient的构造器里面有体现

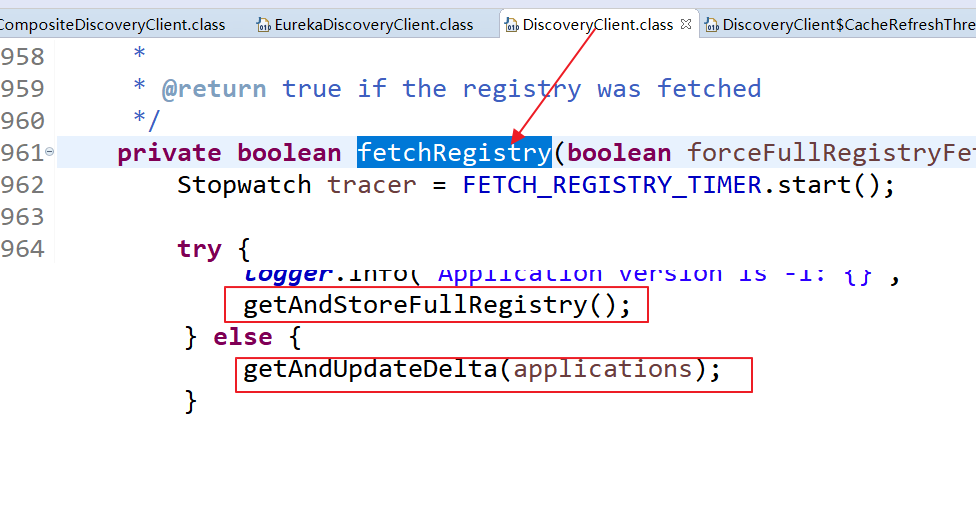




2 我们的服务的实例并不是实时的eureka-server 里面的数据，而是一个本地的（内存）缓存数据



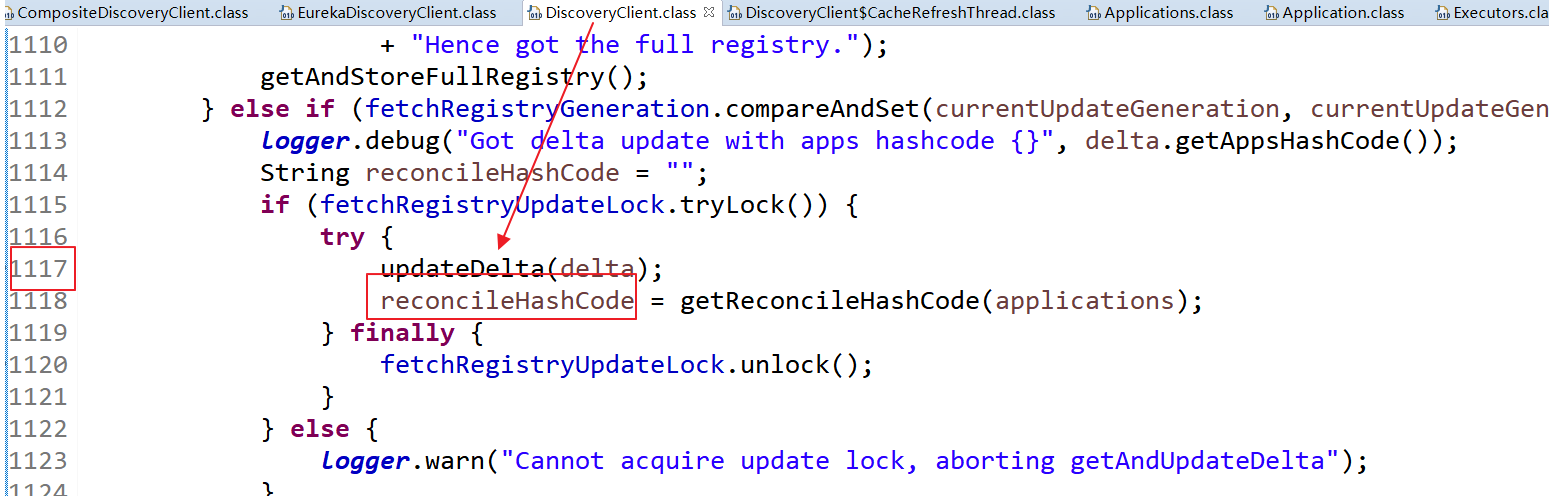
3 缓存的脏读和更新的解决



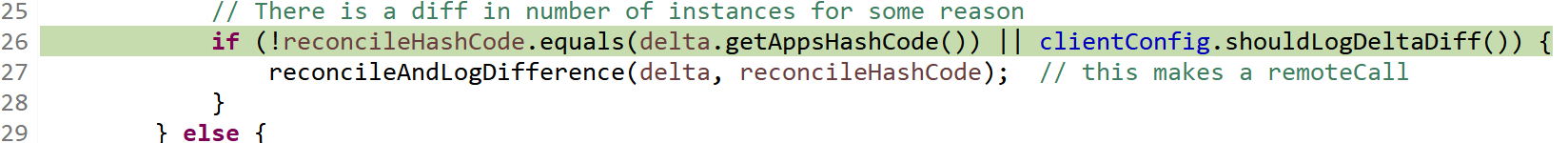
全量拉取发生在：当服务列表为null的情况

增量拉取发生在当列表不为null ，只拉取eureka-server的修改的数据(注册新的服务，上线服务)

增量拉取： 增量更新：



更新是否成功：



# Eureka 的配置文件总结

Client

Servr

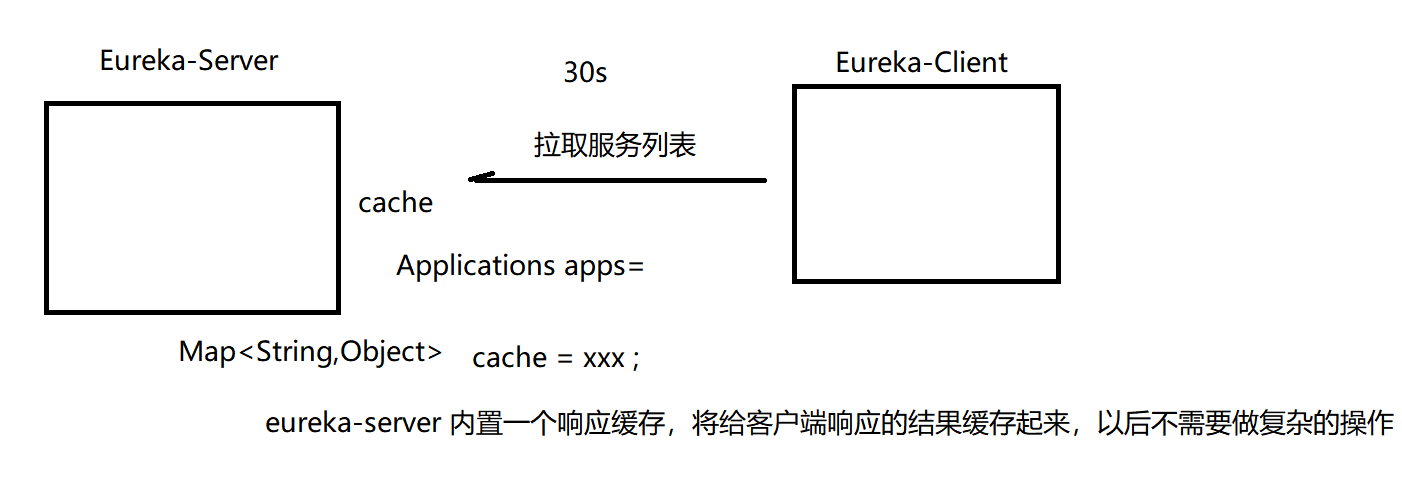
Instance 配置

## 9.1 Server 端（注册中心的配置）

### 9.1.1 响应的缓存配置（为了eureka-server能快速的响应client）

|  |
| --- |
| response-cache-update-interval-ms: 3000  responseCacheAutoExpirationInSeconds: 180 |

Server 端响应给client的数据缓存



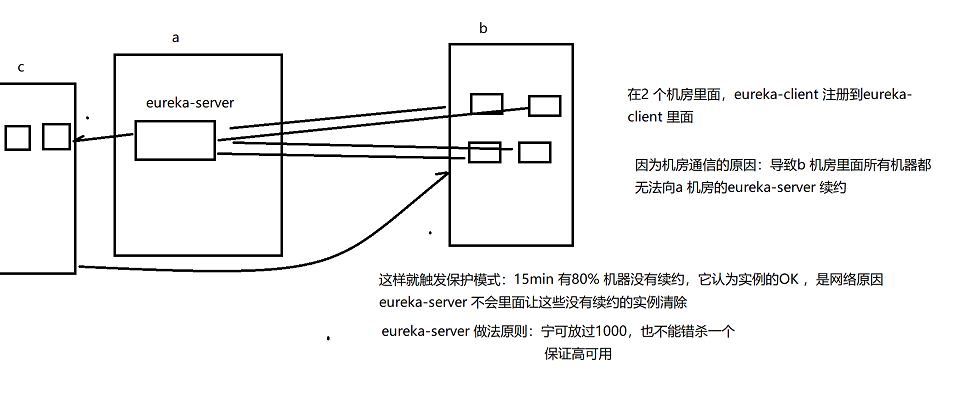
### 9.1.2 server 多久执行一次定期删除

定期删除没有需要的instance(lastupdatetime)

|  |
| --- |
| evictionIntervalTimerInMs：3000 |

### 9.1.3 server的自我保护模式

|  |
| --- |
| enableSelfPreservation: **true** # 本地调试时可fasle关闭。但生产建议打开，可防止因网络不稳定等原因导致误剔除服务。  renewalPercentThreshold: 0.85 # 默认85% |



## 9.2 Client 端

### 9.2.1 续约

|  |
| --- |
| instance:  lease-renewal-interval-in-seconds: 30 # 30s client 向server发一个续签请求，代表我还活着  lease-expiration-duration-in-seconds: 90 # client 多久没有向server发请求，server 会清除它 |
|  |

### 9.2.2 实例id 和ip的配置

|  |
| --- |
| preferIpAddress: **true** # 默认false。应该始终设置为true。如果基于Docker等容器的部署，容器会生成一个随机的主机名，此时DNS不存在该名，无法解析 - John Carnell  instanceId: ${spring.cloud.client.hostname}:${spring.application.name}:${spring.application.instance\_id:${server.port}} |

### 9.2.3 eureka开启和关闭

|  |
| --- |
| # 详见EurekaClientConfigBean（实现EurekaClientConfig）  client:  # 是否启用eureka客户端。默认true  enabled: **true** # 本地调试时，若不想启动eureka，可配置false即可，而不需要注释掉@EnableDiscoveryClient这么麻烦。  registerWithEureka: **true** # 默认true，因此也可省略。  fetchRegistry: **true** # 默认true，此处可不配置。 |

### 9.2.4 eureka-cleint 向server拉取服务列表时，怎么实现的？

默认为30s

可以修改修改该值，来决定多少秒拉取执行一次任务来更新服务列表的缓存

|  |
| --- |
| registry-fetch-interval-seconds: 30 # 如果想eureka server剔除服务后尽快在client体现，我觉得可缩短此时间。 |