自动续约的阈值设置

在EurekaServerBootstrap这个类的contextInitialized方法中，会调用initEurekaServerContext进行初始化

```
public void contextInitialized(ServletContext context) {
    try {
        initEurekaEnvironment();
        initEurekaServerContext();

        context.setAttribute(EurekaServerContext.class.getName(), this.serverContext);
    }
    catch (Throwable e) {
        log.error("Cannot bootstrap eureka server :", e);
        throw new RuntimeException("Cannot bootstrap eureka server :", e);
    }
}
```

继续往下看。

```
protected void initEurekaServerContext() throws Exception {
        EurekaServerConfig eurekaServerConfig = new DefaultEurekaServerConfig();
    //...
    registry.openForTraffic(applicationInfoManager, registryCount);
}
```

> 在openForTraffic方法中，会初始化expectedNumberOfClientsSendingRenews这个值，这个值的含义是： 预期每分钟收到续约的客户端数量，取决于注册到eureka server上的服务数量

```
@Override
public void openForTraffic(ApplicationInfoManager applicationInfoManager, int count) {
    // Renewals happen every 30 seconds and for a minute it should be a factor of 2.
    this.expectedNumberOfClientsSendingRenews = count; //初始值是1.
    updateRenewsPerMinThreshold();
    logger.info("Got {} instances from neighboring DS node", count);
    logger.info("Renew threshold is: {}", numberOfRenewsPerMinThreshold);
    this.startupTime = System.currentTimeMillis();
    if (count > 0) {
        this.peerInstancesTransferEmptyOnStartup = false;
    }
    DataCenterInfo.Name selfName =
applicationInfoManager.getInfo().getDataCenterInfo().getName();
    boolean isAws = Name.Amazon == selfName;
    if (isAws && serverConfig.shouldPrimeAwsReplicaConnections()) {
        logger.info("Priming AWS connections for all replicas..");
        primeAwsReplicas(applicationInfoManager);
    }
    logger.info("Changing status to UP");
    applicationInfoManager.setInstanceStatus(InstanceStatus.UP);
    super.postInit();
}
```

updateRenewsPerMinThreshold

接着调用updateRenewsPerMinThreshold方法，会更新一个每分钟最小的续约数量，也就是Eureka Server期望每分钟收到客户端实例续约的总数的阈值。如果小于这个阈值，就会触发自我保护机制。

```
protected void updateRenewsPerMinThreshold() {
    this.numberOfRenewsPerMinThreshold = (int)
(this.expectedNumberOfClientsSendingRenews
            * (60.0 / serverConfig.getExpectedClientRenewalIntervalSeconds())
            * serverConfig.getRenewalPercentThreshold());
}
//自我保护阀值 = 服务总数 * 每分钟续约数(60S/客户端续约间隔) * 自我保护续约百分比阀值因子
```

- getExpectedClientRenewalIntervalSeconds，客户端的续约间隔，默认为30s
- getRenewalPercentThreshold，自我保护续约百分比阈值因子，默认0.85。 也就是说每分钟的续约数量要大于85%

预期值的变化触发机制

expectedNumberOfClientsSendingRenews和numberOfRenewsPerMinThreshold 这两个值，会随着新增服务注册以及服务下线的触发而发生变化。

PeerAwareInstanceRegistryImpl.cancel

当服务提供者主动下线时，表示这个时候Eureka-Server要剔除这个服务提供者的地址，同时也代表这这个心跳续约的阈值要发生变化。所以在PeerAwareInstanceRegistryImpl.cancel中可以看到数据的更新

> 调用路径 PeerAwareInstanceRegistryImpl.cancel ->
> AbstractInstanceRegistry.cancel->internalCancel

服务下线之后，意味着需要发送续约的客户端数量递减了，所以在这里进行修改

```
protected boolean internalCancel(String appName, String id, boolean isReplication) {
  //....
    synchronized (lock) {
        if (this.expectedNumberOfClientsSendingRenews > 0) {
            // Since the client wants to cancel it, reduce the number of clients to
send renews.
            this.expectedNumberOfClientsSendingRenews =
this.expectedNumberOfClientsSendingRenews - 1;
            updateRenewsPerMinThreshold();
        }
    }
}
```

PeerAwareInstanceRegistryImpl.register

当有新的服务提供者注册到eureka-server上时，需要增加续约的客户端数量，所以在register方法中会进行处理

> register ->super.register(AbstractInstanceRegistry)

```
public void register(InstanceInfo registrant, int leaseDuration, boolean
isReplication) {
    //....
    // The lease does not exist and hence it is a new registration
    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();
        }
    }
}
```

每隔15分钟刷新自我保护阈值

> PeerAwareInstanceRegistryImpl.scheduleRenewalThresholdUpdateTask

每隔15分钟，更新一次自我保护阈值！

```
private void updateRenewalThreshold() {
    try {
        // 1. 计算应用实例数
        Applications apps = eurekaClient.getApplications();
        int count = 0;
        for (Application app : apps.getRegisteredApplications()) {
            for (InstanceInfo instance : app.getInstances()) {
                if (this.isRegisterable(instance)) {
                    ++count;
                }
            }
        }

        synchronized (lock) {
            // Update threshold only if the threshold is greater than the
            // current expected threshold or if self preservation is disabled.
            //当节点数量count大于最小续约数量时，或者没有开启自我保护机制的情况下，重新计算
expectedNumberOfClientsSendingRenews和numberOfRenewsPerMinThreshold
            if ((count) > (serverConfig.getRenewalPercentThreshold() *
expectedNumberOfClientsSendingRenews)
                || (!this.isSelfPreservationModeEnabled())) {
                this.expectedNumberOfClientsSendingRenews = count;
                updateRenewsPerMinThreshold();
            }
        }
        logger.info("Current renewal threshold is : {}",
numberOfRenewsPerMinThreshold);
    } catch (Throwable e) {
        logger.error("Cannot update renewal threshold", e);
    }
}
```

自我保护机制的触发

在AbstractInstanceRegistry的postInit方法中，会开启一个EvictionTask的任务，这个任务用来检测是否需要开启自我保护机制。

> 这个方法也是在EurekaServerBootstrap方法启动时触发。

```
protected void postInit() {
    renewsLastMin.start(); //开启一个定时任务，用来实现每分钟的续约数量，每隔60s归0重新计算
    if (evictionTaskRef.get() != null) {
        evictionTaskRef.get().cancel();
    }
    evictionTaskRef.set(new EvictionTask()); //启动一个定时任务EvictionTask,每隔60s执行一
次
    evictionTimer.schedule(evictionTaskRef.get(),
                           serverConfig.getEvictionIntervalTimerInMs(),
                           serverConfig.getEvictionIntervalTimerInMs());
}
```

其中，EvictionTask的代码如下。

```
private final AtomicLong lastExecutionNanosRef = new AtomicLong(0l);

@Override
public void run() {
    try {
        //获取补偿时间毫秒数
        long compensationTimeMs = getCompensationTimeMs();
        logger.info("Running the evict task with compensationTime {}ms",
compensationTimeMs);
        evict(compensationTimeMs);
    } catch (Throwable e) {
        logger.error("Could not run the evict task", e);
    }
}
```

evict方法

```
public void evict(long additionalLeaseMs) {
    logger.debug("Running the evict task");
     // 是否需要开启自我保护机制，如果需要，那么直接RETURE， 不需要继续往下执行了
    if (!isLeaseExpirationEnabled()) {
        logger.debug("DS: lease expiration is currently disabled.");
        return;
    }

    //这下面主要是做服务自动下线的操作的。
}
```

isLeaseExpirationEnabled

- 是否开启了自我保护机制，如果没有，则跳过，默认是开启
- 计算是否需要开启自我保护，判断最后一分钟收到的续约数量是否大于 numberOfRenewsPerMinThreshold

```
public boolean isLeaseExpirationEnabled() {
    if (!isSelfPreservationModeEnabled()) {
        // The self preservation mode is disabled, hence allowing the instances to
expire.
        return true;
    }
    return numberOfRenewsPerMinThreshold > 0 && getNumOfRenewsInLastMin() >
numberOfRenewsPerMinThreshold;
}
```

心跳超时过期剔除

在AbstractInstanceRegsitry.evict方法中，会执行心跳超时的实例的剔除

```
public void evict(long additionalLeaseMs) {
    logger.debug("Running the evict task");

    if (!isLeaseExpirationEnabled()) { //如果开启了自我保护，则不能剔除过期的服务
        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,
    // the impact should be evenly distributed across all applications.
    //遍历注册中心的节点
    List<Lease<InstanceInfo>> expiredLeases = new ArrayList<>();
    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();
                //判断节点过期状态，如果已过期，则添加到expiredLeases中
                if (lease.isExpired(additionalLeaseMs) && lease.getHolder() != null) {
                    expiredLeases.add(lease);
                }
            }
        }
    }

    // To compensate for GC pauses or drifting local time, we need to use current
registry size as a base for
    // triggering self-preservation. Without that we would wipe out full registry.
      // 获取注册的实例数量
    int registrySize = (int) getLocalRegistrySize();
    //主要是为了避免开启自动保护机制， 所以会逐步过期
    int registrySizeThreshold = (int) (registrySize *
serverConfig.getRenewalPercentThreshold());
    //可以过期的数量
    int evictionLimit = registrySize - registrySizeThreshold;
   //在过期数量和可以过期的数量中间取最小值。
    int toEvict = Math.min(expiredLeases.size(), evictionLimit);
    if (toEvict > 0) {
        logger.info("Evicting {} items (expired={}, evictionLimit={})", toEvict,
expiredLeases.size(), evictionLimit);
```

```
        //随机过期
        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);
            Collections.swap(expiredLeases, i, next);
            Lease<InstanceInfo> lease = expiredLeases.get(i); //随机取一个过期的节点

            String appName = lease.getHolder().getAppName();
            String id = lease.getHolder().getId();
            EXPIRED.increment(); //增加过期数量
            logger.warn("DS: Registry: expired lease for {}/{}", appName, id);
            internalCancel(appName, id, false); //调用cancel方法执行下线
        }
    }
}
```

internalCancel

该方法用来实现服务下线的功能。

```
protected boolean internalCancel(String appName, String id, boolean isReplication) {
    read.lock();
    try {
        CANCEL.increment(isReplication); //统计数据
        Map<String, Lease<InstanceInfo>> gMap = registry.get(appName); // 根据服务名称获取实例信息
        Lease<InstanceInfo> leaseToCancel = null;
        if (gMap != null) { //如果服务实例存在，则直接移除
            leaseToCancel = gMap.remove(id);
        }
        //保存到最近移除的队列
        recentCanceledQueue.add(new Pair<Long, String>(System.currentTimeMillis(), appName + "(" + id + ")"));
        //获取覆盖实例状态的信息
        InstanceStatus instanceStatus = overriddenInstanceStatusMap.remove(id);
        if (instanceStatus != null) {
            logger.debug("Removed instance id {} from the overridden map which has value {}", id, instanceStatus.name());
        }
        //如果被cancel的lease为空，则表示该lease不存在与服务注册中心，不需要处理
        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();//执行服务的下线
            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();
            }
            //让缓存失效
            invalidateCache(appName, vip, svip);
```

```
            logger.info("Cancelled instance {}/{} (replication={})", appName, id,
isReplication);
        }
    } finally {
        read.unlock();
    }

    //更新续约阈值，因为下线了一个服务，所以续约阈值需要发生变化。
    synchronized (lock) {
        if (this.expectedNumberOfClientsSendingRenews > 0) {
            // Since the client wants to cancel it, reduce the number of clients to
send renews.
            this.expectedNumberOfClientsSendingRenews =
this.expectedNumberOfClientsSendingRenews - 1;
            updateRenewsPerMinThreshold();
        }
    }

    return true;
}
```

服务消费者的服务获取

我们继续来研究服务的发现过程，就是客户端需要能够满足两个功能

- 在启动的时候获取指定服务提供者的地址列表
- Eureka server端地址发生变化时，需要动态感知

服务启动时的默认fetch

如果开启了eureka.client.fetch-registry=true配置，那么DiscoveryClient在启动时，就会去服务器上拉取一次地址信息，代码实现如下。

DiscoveryClient构造时进行查询

构造方法中，如果当前的客户端默认开启了fetchRegistry，则会从eureka-server中拉取数据。

```
DiscoveryClient(ApplicationInfoManager applicationInfoManager, EurekaClientConfig
config, AbstractDiscoveryClientOptionalArgs args,
                Provider<BackupRegistry> backupRegistryProvider,
EndpointRandomizer endpointRandomizer) {
    if (clientConfig.shouldFetchRegistry() && !fetchRegistry(false)) {
            fetchRegistryFromBackup();
        }
}
```

DiscoveryClient.fetchRegistry

```
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();
            //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. 是否禁用增量更新；
                //2. 是否对某个region特别关注；
                //3. 外部调用时是否通过入参指定全量更新；
                //4. 本地还未缓存有效的服务列表信息；
            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));
                getAndStoreFullRegistry();//调用全量更新
            } else {//否则，执行增量同步
                getAndUpdateDelta(applications);
            }
            applications.setAppsHashCode(applications.getReconcileHashCode());
            logTotalInstances();
        } catch (Throwable e) {
            logger.info(PREFIX + "{} - was unable to refresh its cache! This periodic
background refresh will be retried in {} seconds. status = {} stacktrace = {}",
                    appPathIdentifier, clientConfig.getRegistryFetchIntervalSeconds(),
e.getMessage(), ExceptionUtils.getStackTrace(e));
            return false;
        } finally {
            if (tracer != null) {
                tracer.stop();
            }
        }

         //将本地缓存更新的事件广播给所有已注册的监听器，注意该方法已被CloudEurekaClient类重写
        onCacheRefreshed();

        // Update remote status based on refreshed data held in the cache
         //检查刚刚更新的缓存中，有来自Eureka server的服务列表，其中包含了当前应用的状态，
        //当前实例的成员变量lastRemoteInstanceStatus，记录的是最后一次更新的当前应用状态，
        //上述两种状态在updateInstanceRemoteStatus方法中作比较 ，如果不一致，就更新
lastRemoteInstanceStatus，并且广播对应的事件
        updateInstanceRemoteStatus();

        // registry was fetched successfully, so return true
        return true;
    }
```

DiscoveryClient.getAndUpdateDelta

默认情况下，会执行增量同步逻辑，代码如下。

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

    Applications delta = null;
    //发起远程通信
    EurekaHttpResponse<Applications> httpResponse =
eurekaTransport.queryClient.getDelta(remoteRegionsRef.get());
    if (httpResponse.getStatusCode() == Status.OK.getStatusCode()) {
        delta = httpResponse.getEntity();
    }

    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();
    } else if (fetchRegistryGeneration.compareAndSet(currentUpdateGeneration,
currentUpdateGeneration + 1)) {
        logger.debug("Got delta update with apps hashcode {}",
delta.getAppsHashCode());
        String reconcileHashCode = "";
        if (fetchRegistryUpdateLock.tryLock()) {
            try {
                updateDelta(delta);
                reconcileHashCode = getReconcileHashCode(applications);
            } finally {
                fetchRegistryUpdateLock.unlock();
            }
        } else {
            logger.warn("Cannot acquire update lock, aborting getAndUpdateDelta");
        }
        // 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());
    }
}
```

定时任务更新缓存

Eureka Client每隔30s，会刷新一次客户端地址列表缓存。

在**DiscoveryClient**构造的时候，会初始化一些任务，这个在前面咱们分析过了。其中有一个任务动态更新本地服务地址列表，叫cacheRefreshTask。

这个任务最终执行的是CacheRefreshThread这个线程。它是一个周期性执行的任务，具体我们来看一下。

```
private void initScheduledTasks() {
```

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

TimedSupervisorTask

从整体上看，TimedSupervisorTask是固定间隔的周期性任务，一旦遇到超时就会将下一个周期的间隔时间调大，如果连续超时，那么每次间隔时间都会增大一倍，一直到达外部参数设定的上限为止，一旦新任务不再超时，间隔时间又会自动恢复为初始值。这种设计还是值得学习的。

```
public void run() {
    Future future = null;
    try {
        //使用Future，可以设定子线程的超时时间，这样当前线程就不用无限等待了
        future = executor.submit(task);
        threadPoolLevelGauge.set((long) executor.getActiveCount());
        //指定等待子线程的最长时间
        future.get(timeoutMillis, TimeUnit.MILLISECONDS);  // block until done or
timeout
        //delay是个很有用的变量，后面会用到，这里记得每次执行任务成功都会将delay重置
        delay.set(timeoutMillis);
        threadPoolLevelGauge.set((long) executor.getActiveCount());
    } catch (TimeoutException e) {
        logger.error("task supervisor timed out", e);
        timeoutCounter.increment();

        long currentDelay = delay.get();
        //任务线程超时的时候，就把delay变量翻倍，但不会超过外部调用时设定的最大延时时间
        long newDelay = Math.min(maxDelay, currentDelay * 2);
        //设置为最新的值，考虑到多线程，所以用了CAS
        delay.compareAndSet(currentDelay, newDelay);
    } catch (RejectedExecutionException e) {
        //一旦线程池的阻塞队列中放满了待处理任务，触发了拒绝策略，就会将调度器停掉
        if (executor.isShutdown() || scheduler.isShutdown()) {
            logger.warn("task supervisor shutting down, reject the task", e);
        } else {
            logger.error("task supervisor rejected the task", e);
        }

        rejectedCounter.increment();
    } catch (Throwable e) {
        //一旦出现未知的异常，就停掉调度器
```

```
            if (executor.isShutdown() || scheduler.isShutdown()) {
                logger.warn("task supervisor shutting down, can't accept the task");
            } else {
                logger.error("task supervisor threw an exception", e);
            }

            throwableCounter.increment();
        } finally {
            //这里任务要么执行完毕，要么发生异常，都用cancel方法来清理任务；
            if (future != null) {
                future.cancel(true);
            }

            //只要调度器没有停止，就再指定等待时间之后在执行一次同样的任务
            if (!scheduler.isShutdown()) {
                //这里就是周期性任务的原因：只要没有停止调度器，就再创建一次性任务，执行时间时
dealy的值，
                //假设外部调用时传入的超时时间为30秒（构造方法的入参timeout），最大间隔时间为50秒
(构造方法的入参expBackOffBound)
                //如果最近一次任务没有超时，那么就在30秒后开始新任务，
                //如果最近一次任务超时了，那么就在50秒后开始新任务（异常处理中有个乘以二的操作，
乘以二后的60秒超过了最大间隔50秒）
                scheduler.schedule(this, delay.get(), TimeUnit.MILLISECONDS);
            }
        }
    }
```

CacheRefreshThread.refreshRegistry

这段代码主要两个逻辑

- 判断remoteRegions是否发生了变化
- 调用fetchRegistry获取本地服务地址缓存

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

        boolean remoteRegionsModified = false;
        //如果部署在aws环境上，会判断最后一次远程区域更新的信息和当前远程区域信息进行比较，如
果不想等，则更新
        String latestRemoteRegions = clientConfig.fetchRegistryForRemoteRegions();
        if (null != latestRemoteRegions) {
            String currentRemoteRegions = remoteRegionsToFetch.get();
            if (!latestRemoteRegions.equals(currentRemoteRegions)) {
                //判断最后一次
            }

            boolean success = fetchRegistry(remoteRegionsModified);
            if (success) {
                registrySize = localRegionApps.get().size();
                lastSuccessfulRegistryFetchTimestamp = System.currentTimeMillis();
            }
            // 省略
        } catch (Throwable e) {
            logger.error("Cannot fetch registry from server", e);
        }
    }
```

从上述代码发现，最终又会调用fetchRegistry方法，来实现服务数据的获取。

服务地址的本地缓存

从eureka server端获取服务注册中心的地址信息后，会缓存到本地内存中，具体的实现如下：

```
private void getAndUpdateDelta(Applications applications) throws Throwable {
    Applications delta = null;
    EurekaHttpResponse<Applications> httpResponse =
eurekaTransport.queryClient.getDelta(remoteRegionsRef.get());
    if (httpResponse.getStatusCode() == Status.OK.getStatusCode()) {
        delta = httpResponse.getEntity();
    }
    if (delta == null) { //增量获取数据为空，则通过getAndStoreFullRegistry进行全量获取
        logger.warn("The server does not allow the delta revision to be applied
because it is not safe. "
                    + "Hence got the full registry.");
        getAndStoreFullRegistry();
    } else if (fetchRegistryGeneration.compareAndSet(currentUpdateGeneration,
currentUpdateGeneration + 1)) { //修改一个当前更新的周期，
        logger.debug("Got delta update with apps hashcode {}",
delta.getAppsHashCode());
        String reconcileHashCode = "";
        if (fetchRegistryUpdateLock.tryLock()) {
            try {
                updateDelta(delta); //更新增量数据
                reconcileHashCode = getReconcileHashCode(applications);
            } finally {
                fetchRegistryUpdateLock.unlock();
            }
        } else {
            logger.warn("Cannot acquire update lock, aborting getAndUpdateDelta");
        }
        // 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());
    }
}
```

DiscoveryClient.updateDelta

这个方法，主要是针对本地缓存进行数据的新增、修改、删除。

```
private void updateDelta(Applications delta) {
    int deltaCount = 0;
    for (Application app : delta.getRegisteredApplications()) { //遍历从服务端获取的注册
信息
        for (InstanceInfo instance : app.getInstances()) { //遍历应用对应的instanceInfo
            Applications applications = getApplications(); //从本地服务缓存中获取应用列表
```

```
            //获取实例所属区域
            String instanceRegion = instanceRegionChecker.getInstanceRegion(instance);
            //如果不是localRegion，则调用下面的逻辑处理，这个和aws有关系，我们不需要关心
            if (!instanceRegionChecker.isLocalRegion(instanceRegion)) {
                Applications remoteApps = remoteRegionVsApps.get(instanceRegion);
                if (null == remoteApps) {
                    remoteApps = new Applications();
                    remoteRegionVsApps.put(instanceRegion, remoteApps);
                }
                applications = remoteApps;
            }

            ++deltaCount; //递增
            //根据动作类型分别进行处理
            if (ActionType.ADDED.equals(instance.getActionType())) { //如果是新增
                //从本地缓存表中获取对应实例的名词，判断实例如果不存在，则添加到本地应用缓存
表
                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);

 applications.getRegisteredApplications(instance.getAppName()).addInstance(instance);
//添加实例信息到本地缓存表
            } else if (ActionType.MODIFIED.equals(instance.getActionType())) { //修改
                Application existingApp =
applications.getRegisteredApplications(instance.getAppName());
                if (existingApp == null) {
                    applications.addApplication(app);
                }
                logger.debug("Modified instance {} to the existing apps ",
instance.getId());


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

            } else if (ActionType.DELETED.equals(instance.getActionType())) { //删除
                Application existingApp =
applications.getRegisteredApplications(instance.getAppName());
                if (existingApp != null) {
                    logger.debug("Deleted instance {} to the existing apps ",
instance.getId());
                    existingApp.removeInstance(instance);
                    /*
                     * We find all instance list from application(The status of
instance status is not only the status is UP but also other status)
                     * if instance list is empty, we remove the application.
                     */
                    if (existingApp.getInstancesAsIsFromEureka().isEmpty()) {
                        applications.removeApplication(existingApp);
                    }
                }
            }
        }
    }
    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());
    }
}
```

本地缓存的存储方式

在DiscoveryClient中，声明了一个原子引用。

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

并且在DiscoveryClient的构造方法中，初始化了一个Applications的对象

```
@Inject
DiscoveryClient(ApplicationInfoManager applicationInfoManager, EurekaClientConfig
config, AbstractDiscoveryClientOptionalArgs args,
                Provider<BackupRegistry> backupRegistryProvider, EndpointRandomizer
endpointRandomizer) {
    localRegionApps.set(new Applications());
}
```

Applications的构造方法如下。

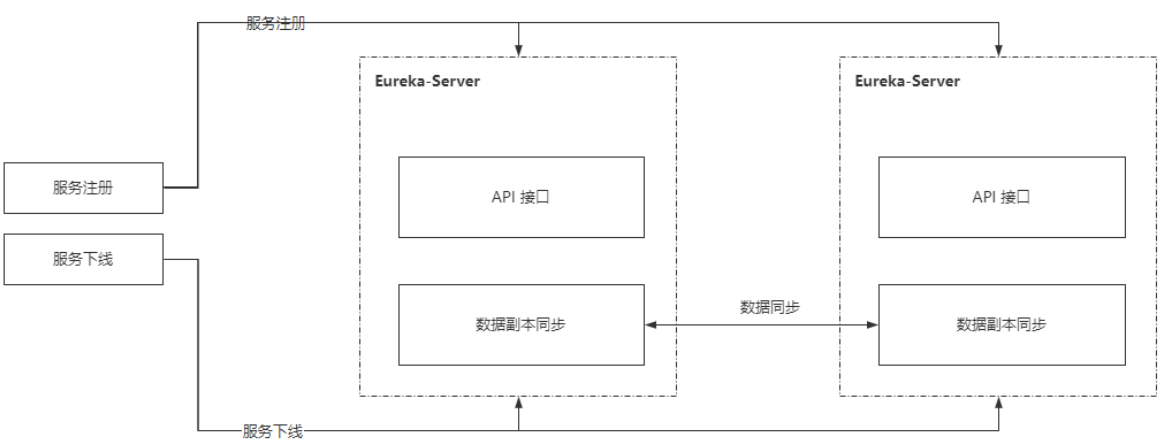
```
public Applications(@JsonProperty("appsHashCode") String appsHashCode,
                    @JsonProperty("versionDelta") Long versionDelta,
                    @JsonProperty("application") List<Application>
registeredApplications) {
    this.applications = new ConcurrentLinkedQueue<Application>();  //用队列存储所有的应
用信息
    this.appNameApplicationMap = new ConcurrentHashMap<String, Application>(); //用map
集合存储所有应用信息，其中key是代表应用应用名称
    this.virtualHostNameAppMap = new ConcurrentHashMap<String, VipIndexSupport>();
    this.secureVirtualHostNameAppMap = new ConcurrentHashMap<String, VipIndexSupport>
();
    this.appsHashCode = appsHashCode;
    this.versionDelta = versionDelta;

    for (Application app : registeredApplications) {
        this.addApplication(app);
    }
}
```

Eureka Client数据同步机制

Eureka Server集群之间的数据一致性是如何保证的呢?

Eureka-Server 集群不区分主从节点，所有节点相同角色(也就是没有角色)，完全对等，是peer to peer模式。没有一致性算法，全靠复制，是最终一致性。

Eureka-Client 可以向任意 Eureka-Client 发起任意读写操作，Eureka-Server 将操作复制到另外的 Eureka-Server 以达到最终一致性，基本原理如下图所示。


服务注册
Eureka-Server
Eureka-Server
服务注册
API 接口
API 接口
服务下线
数据副本同步
数据同步
数据副本同步
服务下线