为什么要看源码：

1、**提升技术功底：**学习源码里的优秀设计思想，比如一些疑难问题的解决思路，还有一些优秀的设计模式，整体提升自己的技术功底

2、**深度掌握技术框架：**源码看多了，对于一个新技术或框架的掌握速度会有大幅提升，看下框架demo大致就能知道底层的实现，技术框架更新再快也不怕

3、**快速定位线上问题：**遇到线上问题，特别是框架源码里的问题(比如bug)，能够快速定位，这就是相比其他没看过源码的人的优势

4、**对面试大有裨益：**面试一线互联网公司对于框架技术一般都会问到源码级别的实现

5、**技术追求：**对技术有追求的人必做之事，使用了一个好的框架，很想知道底层是如何实现的

看源码方法：

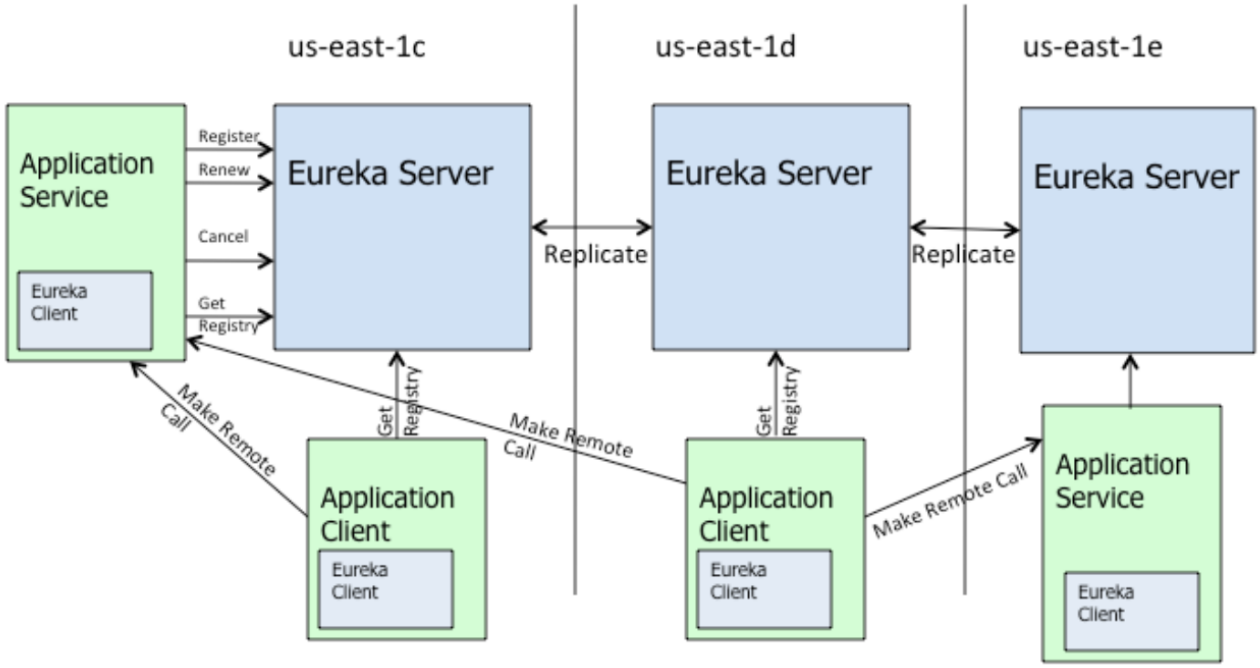
1、**先使用：**先看官方文档快速掌握框架的基本使用

2、**抓主线：**找一个demo入手，顺藤摸瓜快速静态看一遍框架的主线源码(**抓大放小**)，画出源码主流程图，切勿一开始就陷入源码的细枝末节，否则会把自己绕晕

3、**画图做笔记：**总结框架的一些核心功能点，从这些功能点入手深入到源码的细节，边看源码边画源码走向图，并对关键源码的理解做笔记，把源码里的闪光点都记录下来，后续借鉴到工作项目中，理解能力强的可以直接看静态源码，也可以边看源码边debug源码执行过程，观察一些关键变量的值

4、**整合总结：**所有功能点的源码都分析完后，回到主流程图再梳理一遍，争取把自己画的所有图都在脑袋里做一个整合

1、Eureka架构图



2、Eureka核心功能点

服务注册(register)：Eureka Client会通过发送REST请求的方式向Eureka Server注册自己的服务，提供自身的元数据，比如ip地址、端口、运行状况指标的url、主页地址等信息。Eureka Server接收到注册请求后，就会把这些元数据信息存储在一个双层的Map中。

服务续约(renew)：在服务注册后，Eureka Client会维护一个心跳来持续通知Eureka Server，说明服务一直处于可用状态，防止被剔除。Eureka Client在默认的情况下会每隔30秒

(eureka.instance.leaseRenewallIntervalInSeconds)发送一次心跳来进行服务续约。

服务同步(replicate)：Eureka Server之间会互相进行注册，构建Eureka Server集群，不同Eureka Server之间会进行服务同步，用来保证服务信息的一致性。

获取服务(get registry)：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：服务消费者在获取到服务清单后，就可以根据清单中的服务列表信息，查找到其他服务的地址，从而进行远程调用。Eureka有Region和Zone的概念，一个Region可以包含多个Zone，在进行服务调用时，优先访问处于同一个Zone中的服务提供者。

服务下线(cancel)：当Eureka Client需要关闭或重启时，就不希望在这个时间段内再有请求进来，所以，就需要提前先发送REST请求给Eureka Server，告诉Eureka Server自己要下线了，Eureka Server在收到请求后，就会把该服务状态置为下线（DOWN），并把该下线事件传播出去。

服务剔除(evict)：有时候，服务实例可能会因为网络故障等原因导致不能提供服务，而此时该实例也没有发送请求给Eureka Server来进行服务下线，所以，还需要有服务剔除的机制。Eureka Server在启动的时候会创建一个定时任务，每隔一段时间（默认60秒），从当前服务清单中把超时没有续约（默认90秒，eureka.instance.leaseExpirationDurationInSeconds）的服务剔除。

自我保护：既然Eureka Server会定时剔除超时没有续约的服务，那就有可能出现一种场景，网络一段时间内发生了异常，所有的服务都没能够进行续约，Eureka Server就把所有的服务都剔除了，这样显然不太合理。所以，就有了自我保护机制，当短时间内，统计续约失败的比例，如果达到一定阈值，则会触发自我保护的机制，在该机制下，Eureka Server不会剔除任何的微服务，等到正常后，再退出自我保护机制。自我保护开关(eureka.server.enable-self-preservation: false)

3、Eureka Server端源码分析

源码流程图参考：<eureka服务端源码分析>

```
@Configuration
@Import(EurekaServerInitializerConfiguration.class)
@ConditionalOnBean(EurekaServerMarkerConfiguration.Marker.class)
@EnableConfigurationProperties({ EurekaDashboardProperties.class,
 InstanceRegistryProperties.class })
@PropertySource("classpath:/eureka/server.properties")
public class EurekaServerAutoConfiguration extends WebMvcConfigurerAdapter {

 // 此处省略大部分代码，仅抽取一些关键的代码片段

 // 加载EurekaController，spring-cloud 提供了一些额外的接口，用来获取eurekaServer的信息
 @Bean
 @ConditionalOnProperty(prefix = "eureka.dashboard", name = "enabled", matchIfMissing = true)
 public EurekaController eurekaController() {
 return new EurekaController(this.applicationInfoManager);
 }

 //初始化集群注册表
 @Bean
 public PeerAwareInstanceRegistry peerAwareInstanceRegistry(
 ServerCodecs serverCodecs) {
 this.eurekaClient.getApplications(); // force initialization
```

```
23  return new InstanceRegistry(this.eurekaServerConfig, this.eurekaClientConfig,
24  serverCodecs, this.eurekaClient,
25  this.instanceRegistryProperties.getExpectedNumberOfRenewsPerMin(),
26  this.instanceRegistryProperties.getDefaultOpenForTrafficCount());
27  }
28
29  // 配置服务节点信息，这里的作用主要是为了配置Eureka的peer节点，也就是说当有收到有节点注册上来
30  //的时候，需要通知给那些服务节点，（互为一个集群）
31  @Bean
32  @ConditionalOnMissingBean
33  public PeerEurekaNodes peerEurekaNodes(PeerAwareInstanceRegistry registry,
34  ServerCodecs serverCodecs) {
35  return new PeerEurekaNodes(registry, this.eurekaServerConfig,
36  this.eurekaClientConfig, serverCodecs, this.applicationInfoManager);
37  }
38  // EurekaServer的上下文
39  @Bean
40  public EurekaServerContext eurekaServerContext(ServerCodecs serverCodecs,
41  PeerAwareInstanceRegistry registry, PeerEurekaNodes peerEurekaNodes) {
42  return new DefaultEurekaServerContext(this.eurekaServerConfig, serverCodecs,
43  registry, peerEurekaNodes, this.applicationInfoManager);
44  }
45  // 这个类的作用是spring-cloud和原生eureka的胶水代码，通过这个类来启动EurekaSever
46  // 后面这个类会在EurekaServerInitializerConfiguration被调用，进行eureka启动
47  @Bean
48  public EurekaServerBootstrap eurekaServerBootstrap(PeerAwareInstanceRegistry registry,
49  EurekaServerContext serverContext) {
50  return new EurekaServerBootstrap(this.applicationInfoManager,
51  this.eurekaClientConfig, this.eurekaServerConfig, registry,
52  serverContext);
53  }
54  // 配置拦截器，ServletContainer里面实现了jersey框架，通过他来实现eurekaServer对外的restFull接口
55  @Bean
56  public FilterRegistrationBean jerseyFilterRegistration(
57  javax.ws.rs.core.Application eurekaJerseyApp) {
58  FilterRegistrationBean bean = new FilterRegistrationBean();
59  bean.setFilter(new ServletContainer(eurekaJerseyApp));
60  bean.setOrder(Ordered.LOWEST_PRECEDENCE);
61  bean.setUrlPatterns(
62  Collections.singletonList(EurekaConstants.DEFAULT_PREFIX + "/*"));
63
64  return bean;
65  }
66 }
```

EurekaServerAutoConfiguration会导入EurekaServerInitializerConfiguration

```
1 /**
2  * @author Dave Syer
3  */
4 @Configuration
5 @CommonsLog
```

```
public class EurekaServerInitializerConfiguration
 implements ServletContextAware, SmartLifecycle, Ordered {

 @Autowired
  private EurekaServerConfig eurekaServerConfig;

  private ServletContext servletContext;

  @Autowired
  private ApplicationContext applicationContext;

  @Autowired
  private EurekaServerBootstrap eurekaServerBootstrap;

  private boolean running;

  private int order = 1;

  @Override
  public void setServletContext(ServletContext servletContext) {
  this.servletContext = servletContext;
  }

  @Override
  public void start() {
  // 启动一个线程
  new Thread(new Runnable() {
  @Override
  public void run() {
  try {
  //初始化EurekaServer，同时启动Eureka Server
  eurekaServerBootstrap.contextInitialized(EurekaServerInitializerConfiguration.this.servletConte
xt);
  log.info("Started Eureka Server");
  // 发布EurekaServer的注册事件
  publish(new EurekaRegistryAvailableEvent(getEurekaServerConfig()));
  // 设置启动的状态为true
  EurekaServerInitializerConfiguration.this.running = true;
  // 发送Eureka Start 事件 ，  其他还有各种事件，我们可以监听这种时间，然后做一些特定的业务需求
  publish(new EurekaServerStartedEvent(getEurekaServerConfig()));
  }
  catch (Exception ex) {
  // Help!
  log.error("Could not initialize Eureka servlet context", ex);
  }
  }
  }).start();
  }

  private EurekaServerConfig getEurekaServerConfig() {
  return this.eurekaServerConfig;
  }
```

```
57
58  private void publish(ApplicationEvent event) {
59  this.applicationContext.publishEvent(event);
60  }
61
62  @Override
63  public void stop() {
64  this.running = false;
65  eurekaServerBootstrap.contextDestroyed(this.servletContext);
66  }
67
68  @Override
69  public boolean isRunning() {
70  return this.running;
71  }
72
73  @Override
74  public int getPhase() {
75  return 0;
76  }
77
78  @Override
79  public boolean isAutoStartup() {
80  return true;
81  }
82
83  @Override
84  public void stop(Runnable callback) {
85  callback.run();
86  }
87
88  @Override
89  public int getOrder() {
90  return this.order;
91  }
92
93 }
```

EurekaServerBootstrap的contextInitialized初始化方法

```
1 //初始化EurekaServer的运行环境和上下文
2 public void contextInitialized(ServletContext context) {
3  try {
4  initEurekaEnvironment();
5  initEurekaServerContext();
6
7  context.setAttribute(EurekaServerContext.class.getName(), this.serverContext);
8  }
9  catch (Throwable e) {
10  log.error("Cannot bootstrap eureka server :", e);
11  throw new RuntimeException("Cannot bootstrap eureka server :", e);
12  }
```

```
 }

 初始化EurekaServer的上下文
 protected void initEurekaServerContext() throws Exception {
 // For backward compatibility
 JsonXStream.getInstance().registerConverter(new V1AwareInstanceInfoConverter(),
 XStream.PRIORITY_VERY_HIGH);
 XmlXStream.getInstance().registerConverter(new V1AwareInstanceInfoConverter(),
 XStream.PRIORITY_VERY_HIGH);

 if (isAws(this.applicationInfoManager.getInfo())) {
 this.awsBinder = new AwsBinderDelegate(this.eurekaServerConfig,
 this.eurekaClientConfig, this.registry, this.applicationInfoManager);
 this.awsBinder.start();
 }

 //初始化eureka server上下文
 EurekaServerContextHolder.initialize(this.serverContext);

 log.info("Initialized server context");

 // Copy registry from neighboring eureka node
 // 从相邻的eureka节点复制注册表
 int registryCount = this.registry.syncUp();
 // 默认每30秒发送心跳，1分钟就是2次
 // 修改eureka状态为up
 // 同时，这里面会开启一个定时任务，用于清理60秒没有心跳的客户端。自动下线
 this.registry.openForTraffic(this.applicationInfoManager, registryCount);

 // Register all monitoring statistics.
 EurekaMonitors.registerAllStats();
}

@Override
 public int syncUp() {
 // Copy entire entry from neighboring DS node
 int count = 0;

 for (int i = 0; ((i < serverConfig.getRegistrySyncRetries()) && (count == 0)); i++) {
 if (i > 0) {
 try {
 Thread.sleep(serverConfig.getRegistrySyncRetryWaitMs());
 } catch (InterruptedException e) {
 logger.warn("Interrupted during registry transfer..");
 break;
 }
 }
 Applications apps = eurekaClient.getApplications();
 for (Application app : apps.getRegisteredApplications()) {
 for (InstanceInfo instance : app.getInstances()) {
 try {
 if (isRegisterable(instance)) {
```

```
65  //将其他节点的实例注册到本节点
66  register(instance, instance.getLeaseInfo().getDurationInSecs(), true);
67  count++;
68  }
69  } catch (Throwable t) {
70  logger.error("During DS init copy", t);
71  }
72  }
73  }
74  }
75  return count;
76  }
77
78 @Override
79 public void openForTraffic(ApplicationInfoManager applicationInfoManager, int count) {
80  // Renewals happen every 30 seconds and for a minute it should be a factor of 2.
81  // 计算每分钟最大续约数
82  this.expectedNumberOfRenewsPerMin = count * 2;
83  // 每分钟最小续约数
84  this.numberOfRenewsPerMinThreshold =
85  (int) (this.expectedNumberOfRenewsPerMin * serverConfig.getRenewalPercentThreshold());
86  logger.info("Got " + count + " instances from neighboring DS node");
87  logger.info("Renew threshold is: " + numberOfRenewsPerMinThreshold);
88  this.startupTime = System.currentTimeMillis();
89  if (count > 0) {
90  this.peerInstancesTransferEmptyOnStartup = false;
91  }
92  DataCenterInfo.Name selfName = applicationInfoManager.getInfo().getDataCenterInfo().getName();
93  boolean isAws = Name.Amazon == selfName;
94  if (isAws && serverConfig.shouldPrimeAwsReplicaConnections()) {
95  logger.info("Priming AWS connections for all replicas..");
96  primeAwsReplicas(applicationInfoManager);
97  }
98  logger.info("Changing status to UP");
99  // 设置实例的状态为UP
100  applicationInfoManager.setInstanceStatus(InstanceStatus.UP);
101  // 开启定时任务，默认60秒执行一次，用于清理60秒之内没有续约的实例
102  super.postInit();
103 }
104
105 protected void postInit() {
106  renewsLastMin.start();
107  if (evictionTaskRef.get() != null) {
108  evictionTaskRef.get().cancel();
109  }
110  evictionTaskRef.set(new EvictionTask());
111  //服务剔除任务
112  evictionTimer.schedule(evictionTaskRef.get(),
113  serverConfig.getEvictionIntervalTimerInMs(),
114  serverConfig.getEvictionIntervalTimerInMs());
115  }
```

从上面的EurekaServerAutoConfiguration类，我们可以看到有个初始化EurekaServerContext的方法

```
@Bean
public EurekaServerContext eurekaServerContext(ServerCodecs serverCodecs,
 PeerAwareInstanceRegistry registry, PeerEurekaNodes peerEurekaNodes) {
 return new DefaultEurekaServerContext(this.eurekaServerConfig, serverCodecs,
 registry, peerEurekaNodes, this.applicationInfoManager);
}
```

DefaultEurekaServerContext 这个类里面的的initialize()方法是被@PostConstruct 这个注解修饰的，在应用加载的时候，会执行这个方法

```
public void initialize() throws Exception {
 logger.info("Initializing ...");
 // 启动一个线程，读取其他集群节点的信息，后面后续复制
 peerEurekaNodes.start();
 //
 registry.init(peerEurekaNodes);
 logger.info("Initialized");
}
```

peerEurekaNodes.start()主要是启动一个只拥有一个线程的线程池，第一次进去会更新一下集群其他节点信息然后启动了一个定时线程，每60秒更新一次，也就是说后续可以根据配置动态的修改节点配置。（原生的spring cloud config支持）

```
public void start() {
 taskExecutor = Executors.newSingleThreadScheduledExecutor(
 new ThreadFactory() {
 @Override
 public Thread newThread(Runnable r) {
 Thread thread = new Thread(r, "Eureka-PeerNodesUpdater");
 thread.setDaemon(true);
 return thread;
 }
 }
 );
 try {
 // 首次进来，更新集群节点信息
 updatePeerEurekaNodes(resolvePeerUrls());
 // 搞个线程
 Runnable peersUpdateTask = new Runnable() {
 @Override
 public void run() {
 try {
 updatePeerEurekaNodes(resolvePeerUrls());
 } catch (Throwable e) {
 logger.error("Cannot update the replica Nodes", e);
 }

 }
 };
 taskExecutor.scheduleWithFixedDelay(
```

```
 peersUpdateTask,
 serverConfig.getPeerEurekaNodesUpdateIntervalMs(),
 serverConfig.getPeerEurekaNodesUpdateIntervalMs(),
 TimeUnit.MILLISECONDS
 );
 } catch (Exception e) {
 throw new IllegalStateException(e);
 }
 for (PeerEurekaNode node : peerEurekaNodes) {
 logger.info("Replica node URL: " + node.getServiceUrl());
 }
}
// 根据URL 构建PeerEurekaNode信息
protected PeerEurekaNode createPeerEurekaNode(String peerEurekaNodeUrl) {
 HttpReplicationClient replicationClient = JerseyReplicationClient.createReplicationClient(serve
rConfig, serverCodecs, peerEurekaNodeUrl);
 String targetHost = hostFromUrl(peerEurekaNodeUrl);
 if (targetHost == null) {
 targetHost = "host";
 }
 return new PeerEurekaNode(registry, targetHost, peerEurekaNodeUrl, replicationClient, serverCon
fig);
}
```

4、Eureka Client端源码分析

源码流程图参考：<eureka客户端源码分析>

client初始化

```
@Inject
DiscoveryClient(ApplicationInfoManager applicationInfoManager, EurekaClientConfig config, Abstra
ctDiscoveryClientOptionalArgs args,
 Provider<BackupRegistry> backupRegistryProvider) {
//省略非关键代码。。。

logger.info("Initializing Eureka in region {}", clientConfig.getRegion());

//省略非关键代码。。。

try {
 // default size of 2 - 1 each for heartbeat and cacheRefresh
 scheduler = Executors.newScheduledThreadPool(2,
 new ThreadFactoryBuilder()
 .setNameFormat("DiscoveryClient-%d")
 .setDaemon(true)
 .build());

 heartbeatExecutor = new ThreadPoolExecutor(
 1, clientConfig.getHeartbeatExecutorThreadPoolSize(), 0, TimeUnit.SECONDS,
 new SynchronousQueue<Runnable>(),
 new ThreadFactoryBuilder()
 .setNameFormat("DiscoveryClient-HeartbeatExecutor-%d")
```

```
 .setDaemon(true)
 .build()
 ); // use direct handoff

 cacheRefreshExecutor = new ThreadPoolExecutor(
 1, clientConfig.getCacheRefreshExecutorThreadPoolSize(), 0, TimeUnit.SECONDS,
 new SynchronousQueue<Runnable>(),
 new ThreadFactoryBuilder()
 .setNameFormat("DiscoveryClient-CacheRefreshExecutor-%d")
 .setDaemon(true)
 .build()
 ); // use direct handoff

 eurekaTransport = new EurekaTransport();
 scheduleServerEndpointTask(eurekaTransport, args);

 AzToRegionMapper azToRegionMapper;
 if (clientConfig.shouldUseDnsForFetchingServiceUrls()) {
 azToRegionMapper = new DNSBasedAzToRegionMapper(clientConfig);
 } else {
 azToRegionMapper = new PropertyBasedAzToRegionMapper(clientConfig);
 }
 if (null != remoteRegionsToFetch.get()) {
 azToRegionMapper.setRegionsToFetch(remoteRegionsToFetch.get().split(","));
 }
 instanceRegionChecker = new InstanceRegionChecker(azToRegionMapper, clientConfig.getRegion());
} catch (Throwable e) {
 throw new RuntimeException("Failed to initialize DiscoveryClient!", e);
}

if (clientConfig.shouldFetchRegistry() && !fetchRegistry(false)) {
 fetchRegistryFromBackup();
}

// call and execute the pre registration handler before all background tasks (inc registration)
is started
if (this.preRegistrationHandler != null) {
 this.preRegistrationHandler.beforeRegistration();
}

if (clientConfig.shouldRegisterWithEureka() && clientConfig.shouldEnforceRegistrationAtInit()) {
 try {
 if (!register() ) {
 throw new IllegalStateException("Registration error at startup. Invalid server response.");
 }
 } catch (Throwable th) {
 logger.error("Registration error at startup: {}", th.getMessage());
 throw new IllegalStateException(th);
 }
}

//最核心代码
```

```
// finally, init the schedule tasks (e.g. cluster resolvers, heartbeat, instanceInfo replicator, fetch
initScheduledTasks();

try {
 Monitors.registerObject(this);
} catch (Throwable e) {
 logger.warn("Cannot register timers", e);
}

// This is a bit of hack to allow for existing code using DiscoveryManager.getInstance()
// to work with DI'd DiscoveryClient
DiscoveryManager.getInstance().setDiscoveryClient(this);
DiscoveryManager.getInstance().setEurekaClientConfig(config);

initTimestampMs = System.currentTimeMillis();
logger.info("Discovery Client initialized at timestamp {} with initial instances count: {}",
 initTimestampMs, this.getApplications().size());
}
```

初始化时启动核心功能定时任务

```
private void initScheduledTasks() {
 //获取服务注册列表信息
 if (clientConfig.shouldFetchRegistry()) {
 //服务注册列表更新的周期时间
 int registryFetchIntervalSeconds = clientConfig.getRegistryFetchIntervalSeconds();
 int expBackOffBound = clientConfig.getCacheRefreshExecutorExponentialBackOffBound();
 //定时更新服务注册列表
 scheduler.schedule(
 new TimedSupervisorTask(
  "cacheRefresh",
  scheduler,
  cacheRefreshExecutor,
  registryFetchIntervalSeconds,
  TimeUnit.SECONDS,
  expBackOffBound,
  new CacheRefreshThread() //该线程执行更新的具体逻辑
  ),
  registryFetchIntervalSeconds, TimeUnit.SECONDS);
  }

  if (clientConfig.shouldRegisterWithEureka()) {
  //服务续约的周期时间
  int renewalIntervalInSecs = instanceInfo.getLeaseInfo().getRenewalIntervalInSecs();
  int expBackOffBound = clientConfig.getHeartbeatExecutorExponentialBackOffBound();
  //应用启动可见此日志，内容是：Starting heartbeat executor: renew interval is: 30
  logger.info("Starting heartbeat executor: " + "renew interval is: " + renewalIntervalInSecs);
  // 服务定时续约
  scheduler.schedule(
  new TimedSupervisorTask(
  "heartbeat",
```

```
 scheduler,
 heartbeatExecutor,
 renewalIntervalInSecs,
 TimeUnit.SECONDS,
 expBackOffBound,
 new HeartbeatThread() //该线程执行续约的具体逻辑
 ),
 renewalIntervalInSecs, TimeUnit.SECONDS);

 //这个Runable中含有服务注册的逻辑
 instanceInfoReplicator = new InstanceInfoReplicator(
 this,
 instanceInfo,
 clientConfig.getInstanceInfoReplicationIntervalSeconds(),
 2); // burstSize

 statusChangeListener = new ApplicationInfoManager.StatusChangeListener() {
 @Override
 public String getId() {
 return "statusChangeListener";
 }

 @Override
 public void notify(StatusChangeEvent statusChangeEvent) {
 if (InstanceStatus.DOWN == statusChangeEvent.getStatus() ||
 InstanceStatus.DOWN == statusChangeEvent.getPreviousStatus()) {
 // log at warn level if DOWN was involved
 logger.warn("Saw local status change event {}", statusChangeEvent);
 } else {
 logger.info("Saw local status change event {}", statusChangeEvent);
 }
 instanceInfoReplicator.onDemandUpdate();
 }
 };

 if (clientConfig.shouldOnDemandUpdateStatusChange()) {
 applicationInfoManager.registerStatusChangeListener(statusChangeListener);
 }
 //服务注册
 instanceInfoReplicator.start(clientConfig.getInitialInstanceInfoReplicationIntervalSeconds());
 } else {
 logger.info("Not registering with Eureka server per configuration");
 }
}
```

TimedSupervisorTask是一个Runnable接口实现，看下它的run方法

```
@Override
public void run() {
 Future<?> future = null;
 try {
 future = executor.submit(task);
```

```
6   threadPoolLevelGauge.set((long) executor.getActiveCount());
7   //指定等待子线程的最长时间
8   future.get(timeoutMillis, TimeUnit.MILLISECONDS); // block until done or timeout
9   //delay是个关键变量，后面会用到，这里记得每次执行任务成功都会将delay重置
10   delay.set(timeoutMillis);
11   threadPoolLevelGauge.set((long) executor.getActiveCount());
12   } catch (TimeoutException e) {
13   logger.warn("task supervisor timed out", e);
14   timeoutCounter.increment();
15
16   long currentDelay = delay.get();
17   //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
18   long newDelay = Math.min(maxDelay, currentDelay * 2);
19   //设置为最新的值，考虑到多线程，所以用了CAS
20   delay.compareAndSet(currentDelay, newDelay);
21   } catch (RejectedExecutionException e) {
22   //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
23   if (executor.isShutdown() || scheduler.isShutdown()) {
24   logger.warn("task supervisor shutting down, reject the task", e);
25   } else {
26   logger.warn("task supervisor rejected the task", e);
27   }
28
29   rejectedCounter.increment();
30   } catch (Throwable e) {
31   if (executor.isShutdown() || scheduler.isShutdown()) {
32   logger.warn("task supervisor shutting down, can't accept the task");
33   } else {
34   logger.warn("task supervisor threw an exception", e);
35   }
36
37   throwableCounter.increment();
38   } finally {
39   //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；
40   if (future != null) {
41   future.cancel(true);
42   }
43   //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
44   if (!scheduler.isShutdown()) {
45   //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expBac
kOffBound)
46   //如果最近一次任务没有超时，那么就在30秒后开始新任务，
47   //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）
48   scheduler.schedule(this, delay.get(), TimeUnit.MILLISECONDS);
49   }
50   }
51 }
```

scheduler.schedule(this, delay.get(), TimeUnit.MILLISECONDS)，从代码注释上可以看出这个方法是一次性调用方法，但是实际上这个方法执行的任务会反复执行，秘密就在this对应的这个类TimedSupervisorTask的run方法

里，run方法任务执行完最后，会再次调用schedule方法，在指定的时间之后执行一次相同的任务，这个间隔时间和最近一次任务是否超时有关，如果超时了则下一次执行任务的间隔时间就会变大；

源码精髓：

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，另外还有CAS来控制多线程同步，这些是我们看源码需要学习到的设计技巧

定时更新服务注册列表线程CacheRefreshThread

```
/**
 * The task that fetches the registry information at specified intervals.
 *
 */
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();
 }
 }

 //获取注册信息方法
```

```
42  boolean success = fetchRegistry(remoteRegionsModified);
43  if (success) {
44  registrySize = localRegionApps.get().size();
45  lastSuccessfulRegistryFetchTimestamp = System.currentTimeMillis();
46  }
47
48  //省略非关键代码。。。
49  } catch (Throwable e) {
50  logger.error("Cannot fetch registry from server", e);
51  }
52 }
53
54 private boolean fetchRegistry(boolean forceFullRegistryFetch) {
55  Stopwatch tracer = FETCH_REGISTRY_TIMER.start();
56
57  try {
58  // If the delta is disabled or if it is the first time, get all
59  // applications
60  // 取出本地缓存之前获取的服务列表信息
61  Applications applications = getApplications();
62
63  //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：
64  //1. 是否禁用增量更新；
65  //2. 是否对某个region特别关注；
66  //3. 外部调用时是否通过入参指定全量更新；
67  //4. 本地还未缓存有效的服务列表信息；
68  if (clientConfig.shouldDisableDelta()
69  || (!Strings.isNullOrEmpty(clientConfig.getRegistryRefreshSingleVipAddress()))
70  || forceFullRegistryFetch
71  || (applications == null)
72  || (applications.getRegisteredApplications().size() == 0)
73  || (applications.getVersion() == -1)) //Client application does not have latest library support
ing delta
74  {
75  logger.info("Disable delta property : {}", clientConfig.shouldDisableDelta());
76  logger.info("Single vip registry refresh property : {}", clientConfig.getRegistryRefreshSingleV
ipAddress());
77  logger.info("Force full registry fetch : {}", forceFullRegistryFetch);
78  logger.info("Application is null : {}", (applications == null));
79  logger.info("Registered Applications size is zero : {}",
80  (applications.getRegisteredApplications().size() == 0));
81  logger.info("Application version is -1: {}", (applications.getVersion() == -1));
82  //全量更新
83  getAndStoreFullRegistry();
84  } else {
85  //增量更新
86  getAndUpdateDelta(applications);
87  }
88  //重新计算和设置一致性hash码
89  applications.setAppsHashCode(applications.getReconcileHashCode());
90  logTotalInstances();
91  } catch (Throwable e) {
```

```
92   logger.error(PREFIX + "{} - was unable to refresh its cache! status = {}", appPathIdentifier, e.getMessage(), e);
93   return false;
94   } finally {
95   if (tracer != null) {
96   tracer.stop();
97   }
98   }
99
100   // Notify about cache refresh before updating the instance remote status
101   //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
102   onCacheRefreshed();
103
104   // Update remote status based on refreshed data held in the cache
105   //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，
106   //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，
107   //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
108   updateInstanceRemoteStatus();
109
110   // registry was fetched successfully, so return true
111   return true;
112 }
```

全量更新getAndStoreFullRegistry

```
1 private void getAndStoreFullRegistry() throws Throwable {
2   long currentUpdateGeneration = fetchRegistryGeneration.get();
3
4   logger.info("Getting all instance registry info from the eureka server");
5
6   Applications apps = null;
7   //由于并没有配置特别关注的region信息，因此会调用eurekaTransport.queryClient.getApplications方法从服务端获取服务列表
8   EurekaHttpResponse<Applications> httpResponse = clientConfig.getRegistryRefreshSingleVipAddress() == null
9   ? eurekaTransport.queryClient.getApplications(remoteRegionsRef.get())
10   : eurekaTransport.queryClient.getVip(clientConfig.getRegistryRefreshSingleVipAddress(), remoteRegionsRef.get());
11   if (httpResponse.getStatusCode() == Status.OK.getStatusCode()) {
12   //返回对象就是服务列表
13   apps = httpResponse.getEntity();
14   }
15   logger.info("The response status is {}", httpResponse.getStatusCode());
16
17   if (apps == null) {
18   logger.error("The application is null for some reason. Not storing this information");
19   }
20   //考虑到多线程同步，只有CAS成功的线程，才会把自己从Eureka server获取的数据来替换本地缓存
21   else if (fetchRegistryGeneration.compareAndSet(currentUpdateGeneration, currentUpdateGeneration + 1)) {
22   //localRegionApps就是本地缓存，是个AtomicReference实例
23   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");
 }
}
```

其中最重要的一段代码eurekaTransport.queryClient.getApplications(remoteRegionsRef.get())，和Eureka server交互的逻辑都在这里面，方法getApplications的具体实现是在EurekaHttpClientDecorator类

```
@Override
public EurekaHttpResponse<Applications> getApplications(final String... regions) {
 return execute(new RequestExecutor<Applications>() {
 @Override
 public EurekaHttpResponse<Applications> execute(EurekaHttpClient delegate) {
 return delegate.getApplications(regions);
 }

 @Override
  public RequestType getRequestType() {
  //本次向Eureka server请求的类型：获取服务列表
  return RequestType.GetApplications;
  }
  });
}
```

debug进去delegate.getApplications(regions)方法会发现delegate实际用的是AbstractJerseyEurekaHttpClient，里面都是具体的jersey实现的网络接口请求

```
@Override
public EurekaHttpResponse<Applications> getApplications(String... regions) {
 //取全量数据的path是"apps"
 return getApplicationsInternal("apps/", regions);
}

@Override
public EurekaHttpResponse<Applications> getDelta(String... regions) {
 //取增量数据的path是"apps/delta"
  return getApplicationsInternal("apps/delta", regions);
}

//具体的请求响应处理都在此方法中
private EurekaHttpResponse<Applications> getApplicationsInternal(String urlPath, String[] regions) {
 ClientResponse response = null;
 String regionsParamValue = null;
 try {
 //jersey、resource这些关键词都预示着这是个restful请求
 WebResource webResource = jerseyClient.resource(serviceUrl).path(urlPath);
 if (regions != null && regions.length > 0) {
 regionsParamValue = StringUtil.join(regions);
 webResource = webResource.queryParam("regions", regionsParamValue);
 }
 Builder requestBuilder = webResource.getRequestBuilder();
 addExtraHeaders(requestBuilder);
 //发起网络请求，将响应封装成ClientResponse实例
```

```
response = requestBuilder.accept(MediaType.APPLICATION_JSON_TYPE).get(ClientResponse.class);

Applications applications = null;
if (response.getStatus() == Status.OK.getStatusCode() && response.hasEntity()) {
//取得全部应用信息
applications = response.getEntity(Applications.class);
}
return anEurekaHttpResponse(response.getStatus(), Applications.class)
.headers(headersOf(response))
.entity(applications)
.build();
} finally {
if (logger.isDebugEnabled()) {
logger.debug("Jersey HTTP GET {}/{}?{}; statusCode={}",
serviceUrl, urlPath,
regionsParamValue == null ? "" : "regions=" + regionsParamValue,
response == null ? "N/A" : response.getStatus()
);
}
if (response != null) {
response.close();
}
}
}
```

获取全量数据，是通过jersey-client库的API向Eureka server发起restful请求http://localhost:8761/eureka/apps实现的，并将响应的服务列表数据放在一个成员变量中作为本地缓存

```
<applications>
<versions__delta>1</versions__delta>
<apps__hashcode>UP_1_</apps__hashcode>
<application>
<name>MICROSERVICE-PROVIDER-USER</name>
<instance>
<instanceId>localhost:microservice-provider-user:8002</instanceId>
<hostName>192.168.101.1</hostName>
<app>MICROSERVICE-PROVIDER-USER</app>
<ipAddr>192.168.101.1</ipAddr>
<status>UP</status>
<overriddenstatus>UNKNOWN</overriddenstatus>
<port enabled="true">8002</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>1554360812763</registrationTimestamp>
<lastRenewalTimestamp>1554360812763</lastRenewalTimestamp>
```

```
<evictionTimestamp>0</evictionTimestamp>
<serviceUpTimestamp>1554360812763</serviceUpTimestamp>
</leaseInfo>
<metadata>
<management.port>8002</management.port>
<jmx.port>61822</jmx.port>
</metadata>
<homePageUrl>http://192.168.101.1:8002/</homePageUrl>
<statusPageUrl>http://192.168.101.1:8002/actuator/info</statusPageUrl>
<healthCheckUrl>http://192.168.101.1:8002/actuator/health</healthCheckUrl>
<vipAddress>microservice-provider-user</vipAddress>
<secureVipAddress>microservice-provider-user</secureVipAddress>
<isCoordinatingDiscoveryServer>false</isCoordinatingDiscoveryServer>
<lastUpdatedTimestamp>1554360812764</lastUpdatedTimestamp>
<lastDirtyTimestamp>1554360812649</lastDirtyTimestamp>
<actionType>ADDED</actionType>
</instance>
</application>
</applications>
```

获取服务列表信息的增量更新getAndUpdateDelta

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

 Applications delta = null;
 //增量信息是通过eurekaTransport.queryClient.getDelta方法完成的
 EurekaHttpResponse<Applications> httpResponse = eurekaTransport.queryClient.getDelta(remoteRegi
onsRef.get());
 if (httpResponse.getStatusCode() == Status.OK.getStatusCode()) {
 //delta中保存了Eureka server返回的增量更新
 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.");
 //如果增量信息为空，就直接发起一次全量更新
 getAndStoreFullRegistry();
 }
 //考虑到多线程同步问题，这里通过CAS来确保请求发起到现在是线程安全的，
 //如果这期间fetchRegistryGeneration变了，就表示其他线程也做了类似操作，因此放弃本次响应的数据
 else if (fetchRegistryGeneration.compareAndSet(currentUpdateGeneration, currentUpdateGeneratior
+ 1)) {
 logger.debug("Got delta update with apps hashcode {}", delta.getAppsHashCode());
 String reconcileHashCode = "";
 if (fetchRegistryUpdateLock.tryLock()) {
 try {
 //用Eureka返回的增量数据和本地数据做合并操作，这个方法稍后会细说
 updateDelta(delta);
```

```
27  //用合并了增量数据之后的本地数据来生成一致性哈希码
28  reconcileHashCode = getReconcileHashCode(applications);
29  } finally {
30  fetchRegistryUpdateLock.unlock();
31  }
32  } else {
33  logger.warn("Cannot acquire update lock, aborting getAndUpdateDelta");
34  }
35  //Eureka server在返回增量更新数据时，也会返回服务端的一致性哈希码，
36  //理论上每次本地缓存数据经历了多次增量更新后，计算出的一致性哈希码应该是和服务端一致的，
37  //如果发现不一致，就证明本地缓存的服务列表信息和Eureka server不一致了，需要做一次全量更新
38  if (!reconcileHashCode.equals(delta.getAppsHashCode()) || clientConfig.shouldLogDeltaDiff()) {
39  //一致性哈希码不同，就在reconcileAndLogDifference方法中做全量更新
40  reconcileAndLogDifference(delta, reconcileHashCode); // this makes a remoteCall
41  }
42  } else {
43  logger.warn("Not updating application delta as another thread is updating it already");
44  logger.debug("Ignoring delta update with apps hashcode {}, as another thread is updating it alr
eady", delta.getAppsHashCode());
45  }
46 }
```

updateDelta方法将增量更新数据和本地数据做合并

```
1  private void updateDelta(Applications delta) {
2  int deltaCount = 0;
3  //遍历所有服务
4  for (Application app : delta.getRegisteredApplications()) {
5  //遍历当前服务的所有实例
6  for (InstanceInfo instance : app.getInstances()) {
7  //取出缓存的所有服务列表，用于合并
8  Applications applications = getApplications();
9  String instanceRegion = instanceRegionChecker.getInstanceRegion(instance);
10  //判断正在处理的实例和当前应用是否在同一个region
11  if (!instanceRegionChecker.isLocalRegion(instanceRegion)) {
12  //如果不是同一个region，接下来合并的数据就换成专门为其他region准备的缓存
13  Applications remoteApps = remoteRegionVsApps.get(instanceRegion);
14  if (null == remoteApps) {
15  remoteApps = new Applications();
16  remoteRegionVsApps.put(instanceRegion, remoteApps);
17  }
18  applications = remoteApps;
19  }
20
21  ++deltaCount;
22
23  if (ActionType.ADDED.equals(instance.getActionType())) { //对新增的实例的处理
24  Application existingApp = applications.getRegisteredApplications(instance.getAppName());
25  if (existingApp == null) {
26  applications.addApplication(app);
27  }
```

```
28   logger.debug("Added instance {} to the existing apps in region {}", instance.getId(), instanceR
egion);
29   applications.getRegisteredApplications(instance.getAppName()).addInstance(instance);
30   } else if (ActionType.MODIFIED.equals(instance.getActionType())) { //对修改实例的处理
31   Application existingApp = applications.getRegisteredApplications(instance.getAppName());
32   if (existingApp == null) {
33   applications.addApplication(app);
34   }
35   logger.debug("Modified instance {} to the existing apps ", instance.getId());
36
37   applications.getRegisteredApplications(instance.getAppName()).addInstance(instance);
38
39   } else if (ActionType.DELETED.equals(instance.getActionType())) { //对删除实例的处理
40   Application existingApp = applications.getRegisteredApplications(instance.getAppName());
41   if (existingApp == null) {
42   applications.addApplication(app);
43   }
44   logger.debug("Deleted instance {} to the existing apps ", instance.getId());
45   applications.getRegisteredApplications(instance.getAppName()).removeInstance(instance);
46   }
47   }
48   }
49   logger.debug("The total number of instances fetched by the delta processor : {}", deltaCount);
50
51   getApplications().setVersion(delta.getVersion());
52   //整理数据，使得后续使用过程中，这些应用的实例总是以相同顺序返回
53   getApplications().shuffleInstances(clientConfig.shouldFilterOnlyUpInstances());
54
55   //和当前应用不在同一个region的应用，其实例数据也要整理
56   for (Applications applications : remoteRegionVsApps.values()) {
57   applications.setVersion(delta.getVersion());
58   applications.shuffleInstances(clientConfig.shouldFilterOnlyUpInstances());
59   }
60  }
```

服务续约

```
1  // 服务定时续约
2  scheduler.schedule(
3   new TimedSupervisorTask(
4   "heartbeat",
5   scheduler,
6   heartbeatExecutor,
7   renewalIntervalInSecs,
8   TimeUnit.SECONDS,
9   expBackOffBound,
10   new HeartbeatThread() //该线程执行续约的具体逻辑，会调用下面的renew()方法
11   ),
12   renewalIntervalInSecs, TimeUnit.SECONDS);
13
14   private class HeartbeatThread implements Runnable {
15   public void run() {
```

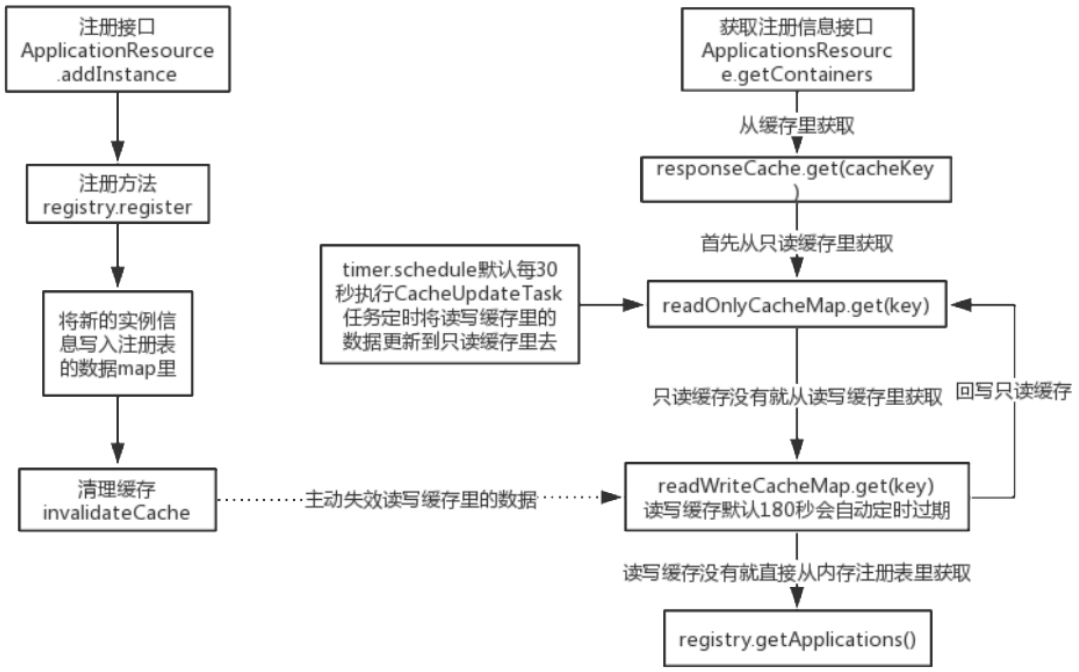
```
  if (renew()) {
  lastSuccessfulHeartbeatTimestamp = System.currentTimeMillis();
  }
  }
 }

  boolean renew() {
  EurekaHttpResponse<InstanceInfo> httpResponse;
  try {
  httpResponse = eurekaTransport.registrationClient.sendHeartBeat(instanceInfo.getAppName(), inst
anceInfo.getId(), instanceInfo, null);
  logger.debug(PREFIX + "{} - Heartbeat status: {}", appPathIdentifier, httpResponse.getStatusCod
e());
  if (httpResponse.getStatusCode() == 404) {
  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() == 200;
  } catch (Throwable e) {
  logger.error(PREFIX + "{} - was unable to send heartbeat!", appPathIdentifier, e);
  return false;
  }
 }
```

服务注册

```
  //服务注册
 instanceInfoReplicator.start(clientConfig.getInitialInstanceInfoReplicationIntervalSeconds());

 public void start(int initialDelayMs) {
  if (started.compareAndSet(false, true)) {
  instanceInfo.setIsDirty(); // for initial register
  Future next = scheduler.schedule(this, initialDelayMs, TimeUnit.SECONDS);
  scheduledPeriodicRef.set(next);
  }
 }

 public void run() {
  try {
  discoveryClient.refreshInstanceInfo();

  Long dirtyTimestamp = instanceInfo.isDirtyWithTime();
  if (dirtyTimestamp != null) {
  discoveryClient.register();
  instanceInfo.unsetIsDirty(dirtyTimestamp);
```

```
20    }
21    } catch (Throwable t) {
22    logger.warn("There was a problem with the instance info replicator", t);
23    } finally {
24    Future next = scheduler.schedule(this, replicationIntervalSeconds, TimeUnit.SECONDS);
25    scheduledPeriodicRef.set(next);
26    }
27  }
```

5、Eureka Server服务端Jersey接口源码分析



注册表map数据

```
{
MICROSERVICE - PROVIDER - USER = {
DESKTOP - 1 SLJLB7: microservice - provider - user: 8002 = com.netflix.eureka.lease.Lease @2cd36af6,
DESKTOP - 1 SLJLB7: microservice - provider - user: 8001 = com.netflix.eureka.lease.Lease @600b7073
}
}
```

服务端Jersey接口处理类ApplicationResource

其中有一个addInstance方法就是用来接收客户端的**注册请求接口**

```
1  //ApplicationResource.java
2  @POST
3  @Consumes({"application/json", "application/xml"})
4  public Response addInstance(InstanceInfo info,
5   @HeaderParam(PeerEurekaNode.HEADER_REPLICATION) String isReplication) {
6   logger.debug("Registering instance {} (replication={})", info.getId(), isReplication);
7   // validate that the instanceinfo contains all the necessary required fields
```

```
// 参数校验，不符合验证规则的，返回400状态码，此处不做详解
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.getAppName())) {
 return Response.status(400).entity("Missing appName").build();
 } else if (!appName.equals(info.getAppName())) {
 return Response.status(400).entity("Mismatched appName, expecting " + appName + " but was " + info.getAppName()).build();
 } else if (info.getDataCenterInfo() == null) {
 return Response.status(400).entity("Missing dataCenterInfo").build();
 } else if (info.getDataCenterInfo().getName() == null) {
 return Response.status(400).entity("Missing dataCenterInfo Name").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.dataCenterInfoId"));
 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());
 }
 }
 }
 // 重点在这里
 registry.register(info, "true".equals(isReplication));
 return Response.status(204).build(); // 204 to be backwards compatible
}
```

AbstractInstanceRegistry的注册方法

```
public void register(InstanceInfo registrant, int leaseDuration, boolean isReplication) {
 try {
 // 上只读锁
 read.lock();
 // 从本地MAP里面获取当前实例的信息。
 Map<String, Lease<InstanceInfo>> gMap = registry.get(registrant.getAppName());
 // 增加注册次数到监控信息里面去。
```

```
 REGISTER.increment(isReplication);
 if (gMap == null) {
  // 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
  final ConcurrentHashMap<String, Lease<InstanceInfo>> gNewMap = new ConcurrentHashMap<String, Le
ase<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。
  gMap = registry.putIfAbsent(registrant.getAppName(), gNewMap);
  if (gMap == null) {
  // 表明map中确实不存在，则设置gMap为最新创建的那个
  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());
  // 当Lease的对象不为空时。
  if (existingLease != null && (existingLease.getHolder() != null)) {
  // 当instance已经存在是，和客户端的instance的信息做比较，时间最新的那个，为有效instance信息
  Long existingLastDirtyTimestamp = existingLease.getHolder().getLastDirtyTimestamp(); // server
  Long registrationLastDirtyTimestamp = registrant.getLastDirtyTimestamp(); // client
  logger.debug("Existing lease found (existing={}, provided={}", existingLastDirtyTimestamp, regi
strationLastDirtyTimestamp);
  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, registrationLastDirtyT
imestamp);
  logger.warn("Using the existing instanceInfo instead of the new instanceInfo as the
registrant");
  registrant = existingLease.getHolder();
  }
  } else {
  // 这里只有当existinglease不存在时，才会进来。 像那种恢复心跳，信息过期的，都不会进入这里。
  // Eureka-Server的自我保护机制做的操作，为每分钟最大续约数+2 ，同时重新计算每分钟最小续约数
  synchronized (lock) {
  if (this.expectedNumberOfRenewsPerMin > 0) {
  // Since the client wants to cancel it, reduce the threshold
  // (1 for 30 seconds, 2 for a minute)
  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) {
  // 当原来存在Lease的信息时，设置他的serviceUpTimestamp，保证服务开启的时间一直是第一次的那个
  lease.setServiceUpTimestamp(existingLease.getServiceUpTimestamp());
  }
```

```
55  // 放入本地Map中
56  gMap.put(registrant.getId(), lease);
57  // 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。
58  synchronized (recentRegisteredQueue) {
59  recentRegisteredQueue.add(new Pair<Long, String>(
60  System.currentTimeMillis(),
61  registrant.getAppName() + "(" + registrant.getId() + ")"));
62  }
63  // This is where the initial state transfer of overridden status happens
64  // 分析instanceStatus
65  if (!InstanceStatus.UNKNOWN.equals(registrant.getOverriddenStatus())) {
66  logger.debug("Found overridden status {} for instance {}. Checking to see if needs to be add to
the "
67  + "overrides", registrant.getOverriddenStatus(), registrant.getId());
68  if (!overriddenInstanceStatusMap.containsKey(registrant.getId())) {
69  logger.info("Not found overridden id {} and hence adding it", registrant.getId());
70  overriddenInstanceStatusMap.put(registrant.getId(), registrant.getOverriddenStatus());
71  }
72  }
73  InstanceStatus overriddenStatusFromMap = overriddenInstanceStatusMap.get(registrant.getId());
74  if (overriddenStatusFromMap != null) {
75  logger.info("Storing overridden status {} from map", overriddenStatusFromMap);
76  registrant.setOverriddenStatus(overriddenStatusFromMap);
77  }
78
79  // Set the status based on the overridden status rules
80  InstanceStatus overriddenInstanceStatus = getOverriddenInstanceStatus(registrant,
existingLease, isReplication);
81  registrant.setStatusWithoutDirty(overriddenInstanceStatus);
82
83  // If the lease is registered with UP status, set lease service up timestamp
84  // 得到instanceStatus，判断是否是UP状态，
85  if (InstanceStatus.UP.equals(registrant.getStatus())) {
86  lease.serviceUp();
87  }
88  // 设置注册类型为添加
89  registrant.setActionType(ActionType.ADDED);
90  // 租约变更记录队列，记录了实例的每次变化， 用于注册信息的增量获取、
91  recentlyChangedQueue.add(new RecentlyChangedItem(lease));
92  registrant.setLastUpdatedTimestamp();
93  // 清理缓存 ，传入的参数为key
94  invalidateCache(registrant.getAppName(), registrant.getVIPAddress(), registrant.getSecureVipAdd
ress());
95  logger.info("Registered instance {}/{} with status {} (replication={})",
96  registrant.getAppName(), registrant.getId(), registrant.getStatus(), isReplication);
97  } finally {
98  read.unlock();
99  }
100 }
```

理解上面的register还需要先了解下注册实例信息存放的的map，这是个两层的ConcurrentHashMap<String, Map<String, Lease<InstanceInfo>>>，外层map的key是appName，也就是服务名，内层map的key是

instanceId，也就是实例名
注册表map数据示例如下：
{
MICROSERVICE - PROVIDER - USER = {
DESKTOP - 1 SLJLB7: microservice - provider - user: 8002 = com.netflix.eureka.lease.Lease @2cd36af6,
DESKTOP - 1 SLJLB7: microservice - provider - user: 8001 = com.netflix.eureka.lease.Lease @600b7073
}
}

内层map的value对应的类Lease需要重点理解下

```
public class Lease<T> {

 enum Action {
 Register, Cancel, Renew
 };

 public static final int DEFAULT_DURATION_IN_SECS = 90;

 private T holder;
  private long evictionTimestamp;
  private long registrationTimestamp;
  private long serviceUpTimestamp;
  // Make it volatile so that the expiration task would see this quicker
  private volatile long lastUpdateTimestamp;
  private long duration;

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

  }

  /**
  * Renew the lease, use renewal duration if it was specified by the
  * associated {@link T} during registration, otherwise default duration is
  * {@link #DEFAULT_DURATION_IN_SECS}.
  */
  public void renew() {
  lastUpdateTimestamp = System.currentTimeMillis() + duration; //有个小bug，不应该加duration

  }

  /**
  * Cancels the lease by updating the eviction time.
  */
  public void cancel() {
```

```
39  if (evictionTimestamp <= 0) {
40  evictionTimestamp = System.currentTimeMillis();
41  }
42  }
43
44  /**
45  * Mark the service as up. This will only take affect the first time called,
46  * subsequent calls will be ignored.
47  */
48  public void serviceUp() {
49  if (serviceUpTimestamp == 0) {
50  serviceUpTimestamp = System.currentTimeMillis();
51  }
52  }
53
54  /**
55  * Set the leases service UP timestamp.
56  */
57  public void setServiceUpTimestamp(long serviceUpTimestamp) {
58  this.serviceUpTimestamp = serviceUpTimestamp;
59  }
60
61  /**
62  * Checks if the lease of a given {@link com.netflix.appinfo.InstanceInfo} has expired or not.
63  */
64  public boolean isExpired() {
65  return isExpired(0l);
66  }
67
68  /**
69  * Checks if the lease of a given {@link com.netflix.appinfo.InstanceInfo} has expired or not.
70  *
71  * Note that due to renew() doing the 'wrong" thing and setting lastUpdateTimestamp to +duration
more than
72  * what it should be, the expiry will actually be 2 * duration. This is a minor bug and should o
nly affect
73  * instances that ungracefully shutdown. Due to possible wide ranging impact to existing usage,
this will
74  * not be fixed.
75  *
76  * @param additionalLeaseMs any additional lease time to add to the lease evaluation in ms.
77  */
78  public boolean isExpired(long additionalLeaseMs) {
79  return (evictionTimestamp > 0 || System.currentTimeMillis() > (lastUpdateTimestamp + duration +
additionalLeaseMs));
80  }
81
82  /**
83  * Gets the milliseconds since epoch when the lease was registered.
84  *
85  * @return the milliseconds since epoch when the lease was registered.
86  */
```

```
87   public long getRegistrationTimestamp() {
88   return registrationTimestamp;
89   }
90
91   /**
92   * Gets the milliseconds since epoch when the lease was last renewed.
93   * Note that the value returned here is actually not the last lease renewal time but the renewal
+ duration.
94   *
95   * @return the milliseconds since epoch when the lease was last renewed.
96   */
97   public long getLastRenewalTimestamp() {
98   return lastUpdateTimestamp;
99   }
100
101   /**
102   * Gets the milliseconds since epoch when the lease was evicted.
103   *
104   * @return the milliseconds since epoch when the lease was evicted.
105   */
106   public long getEvictionTimestamp() {
107   return evictionTimestamp;
108   }
109
110   /**
111   * Gets the milliseconds since epoch when the service for the lease was marked as up.
112   *
113   * @return the milliseconds since epoch when the service for the lease was marked as up.
114   */
115   public long getServiceUpTimestamp() {
116   return serviceUpTimestamp;
117   }
118
119   /**
120   * Returns the holder of the lease.
121   */
122   public T getHolder() {
123   return holder;
124   }
125
126 }
```

DEFAULT_DURATION_IN_SECS : 租约过期的时间常量，默认未90秒，也就说90秒没有心跳过来，那么这边将会自动剔除该节点

holder： 这个租约是属于谁的， 目前占用这个属性的是

instanceInfo，也就是客户端实例信息。

evictionTimestamp： 租约是啥时候过期的，当服务下线的时候，会过来更新这个时间戳registrationTimestamp： 租约的注册时间

serviceUpTimestamp： 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

lastUpdateTimestamp： 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。
duration：过期时间，毫秒单位

服务端Jersey接口处理类ApplicationsResource

其中有一个getContainers方法就是用来**获取所有注册实例信息的接口**

```
@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 differer
t order.
  EurekaMonitors.GET_ALL_WITH_REMOTE_REGIONS.increment();
  }

  // Check if the server allows the access to the registry. The server can
  // restrict access if it is not
  // ready to serve traffic depending on various reasons.
  if (!registry.shouldAllowAccess(isRemoteRegionRequested)) {
  return Response.status(Status.FORBIDDEN).build();
  }
  CurrentRequestVersion.set(Version.toEnum(version));
  KeyType keyType = Key.KeyType.JSON;
  String returnMediaType = MediaType.APPLICATION_JSON;
  if (acceptHeader == null || !acceptHeader.contains(HEADER_JSON_VALUE)) {
  keyType = Key.KeyType.XML;
  returnMediaType = MediaType.APPLICATION_XML;
  }

  //获取服务实例对应的缓存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 {
 //从缓存里获取服务实例注册信息
 response = Response.ok(responseCache.get(cacheKey))
 .build();
 }
 return response;
}

responseCache.get(cacheKey)对应的源码如下：
@VisibleForTesting
String get(final Key key, boolean useReadOnlyCache) {
 //从多级缓存里获取注册实例信息
 Value payload = getValue(key, useReadOnlyCache);
 if (payload == null || payload.getPayload().equals(EMPTY_PAYLOAD)) {
 return null;
 } else {
 return payload.getPayload();
 }
}

@VisibleForTesting
Value getValue(final Key key, boolean useReadOnlyCache) {
 Value payload = null;
 try {
 if (useReadOnlyCache) {
 final Value currentPayload = readOnlyCacheMap.get(key);
 if (currentPayload != null) {
 payload = currentPayload;
 } else {
 payload = readWriteCacheMap.get(key);
 readOnlyCacheMap.put(key, payload);
 }
 } else {
 payload = readWriteCacheMap.get(key);
 }
 } catch (Throwable t) {
 logger.error("Cannot get value for key : {}", key, t);
 }
 return payload;
}


ResponseCacheImpl(EurekaServerConfig serverConfig, ServerCodecs serverCodecs, AbstractInstanceRe
gistry registry) {
 this.serverConfig = serverConfig;
 this.serverCodecs = serverCodecs;
 this.shouldUseReadOnlyResponseCache = serverConfig.shouldUseReadOnlyResponseCache();
 this.registry = registry;

 long responseCacheUpdateIntervalMs = serverConfig.getResponseCacheUpdateIntervalMs();
 this.readWriteCacheMap =
```

```
CacheBuilder.newBuilder().initialCapacity(1000)
//读写缓存默认180秒会自动定时过期
.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);
 }
 }
 })
 .build(new CacheLoader<Key, Value>() {
 @Override
 public Value load(Key key) throws Exception {
 if (key.hasRegions()) {
 Key cloneWithNoRegions = key.cloneWithoutRegions();
 regionSpecificKeys.put(cloneWithNoRegions, key);
 }
 Value value = generatePayload(key);
 return value;
 }
 });

 if (shouldUseReadOnlyResponseCache) {
 //默认30秒用读写缓存的数据更新只读缓存的数据
 timer.schedule(getCacheUpdateTask(),
 new Date(((System.currentTimeMillis() / responseCacheUpdateIntervalMs) * responseCacheUpdateIn
tervalMs)
 + responseCacheUpdateIntervalMs),
 responseCacheUpdateIntervalMs);
 }

 try {
 Monitors.registerObject(this);
 } catch (Throwable e) {
 logger.warn("Cannot register the JMX monitor for the InstanceRegistry", e);
 }
}

//初始化直接从注册表registry里那数据放入readWriteCacheMap
private Value generatePayload(Key key) {
 Stopwatch tracer = null;
 try {
 String payload;
 switch (key.getEntityType()) {
 case Application:
 boolean isRemoteRegionRequested = key.hasRegions();

 if (ALL_APPS.equals(key.getName())) {
```

```
 if (isRemoteRegionRequested) {
 tracer = serializeAllAppsWithRemoteRegionTimer.start();
 payload = getPayLoad(key, registry.getApplicationsFromMultipleRegions(key.getRegions()));
 } else {
 tracer = serializeAllAppsTimer.start();
 payload = getPayLoad(key, registry.getApplications());
 }
 } else if (ALL_APPS_DELTA.equals(key.getName())) {
 if (isRemoteRegionRequested) {
 tracer = serializeDeltaAppsWithRemoteRegionTimer.start();
 versionDeltaWithRegions.incrementAndGet();
 versionDeltaWithRegionsLegacy.incrementAndGet();
 payload = getPayLoad(key,
 registry.getApplicationDeltasFromMultipleRegions(key.getRegions()));
 } else {
 tracer = serializeDeltaAppsTimer.start();
 versionDelta.incrementAndGet();
 versionDeltaLegacy.incrementAndGet();
 payload = getPayLoad(key, registry.getApplicationDeltas());
 }
 } else {
 tracer = serializeOneApptimer.start();
 payload = getPayLoad(key, registry.getApplication(key.getName()));
 }
 break;
 case VIP:
 case SVIP:
 tracer = serializeViptimer.start();
 payload = getPayLoad(key, getApplicationsForVip(key, registry));
 break;
 default:
 logger.error("Unidentified entity type: {} found in the cache key.", key.getEntityType());
 payload = "";
 break;
 }
 return new Value(payload);
 } finally {
 if (tracer != null) {
 tracer.stop();
 }
 }
 }

//用读写缓存的数据更新只读缓存的数据
private TimerTask getCacheUpdateTask() {
 return new TimerTask() {
 @Override
 public void run() {
 logger.debug("Updating the client cache from response cache");
 for (Key key : readOnlyCacheMap.keySet()) {
 if (logger.isDebugEnabled()) {
```

```
196   logger.debug("Updating the client cache from response cache for key : {} {} {} {}",
197   key.getEntityType(), key.getName(), key.getVersion(), key.getType());
198   }
199   try {
200   CurrentRequestVersion.set(key.getVersion());
201   Value cacheValue = readWriteCacheMap.get(key);
202   Value currentCacheValue = readOnlyCacheMap.get(key);
203   if (cacheValue != currentCacheValue) {
204   readOnlyCacheMap.put(key, cacheValue);
205   }
206   } catch (Throwable th) {
207   logger.error("Error while updating the client cache from response cache for key {}", key.toStr
ingCompact(), th);
208   }
209   }
210   }
211   };
212 }
```

源码精髓：多级缓存设计思想

- 在拉取注册表的时候：
  - 首先从**ReadOnlyCacheMap**里查缓存的注册表。
  - 若没有，就找**ReadWriteCacheMap**里缓存的注册表。
  - 如果还没有，就从**内存中获取实际的注册表数据。**

- 在注册表发生变更的时候：
  - 会在内存中更新变更的注册表数据，同时**过期掉ReadWriteCacheMap。**
  - 此过程不会影响**ReadOnlyCacheMap**提供人家查询注册表。
  - 默认每30秒Eureka Server会将**ReadWriteCacheMap**更新到**ReadOnlyCacheMap里**
  - **默认每180秒**Eureka Server会将**ReadWriteCacheMap里是数据失效**
  - 下次有服务拉取注册表，又会从内存中获取最新的数据了，同时填充 各级缓存。

多级缓存机制的优点：

- 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。
- 并且进一步保证对Eureka Server的大量请求，都是快速从纯内存走，性能极高（可以稍微估计下对于一线互联网公司，内部上千个eureka client实例，每分钟对eureka上千次的访问，一天就是上千万次的访问）

看源码彻底搞懂一些诡异的问题：

看完多级缓存这块源码我们可以搞清楚一个常见的问题，就是当我们eureka服务实例有注册或下线或有实例发生故障，内存注册表虽然会及时更新数据，但是客户端不一定能及时感知到，可能会过30秒才能感知到，因为客户端拉取注册表实例这里面有一个多级缓存机制

还有服务剔除的不是默认90秒没心跳的实例，剔除的是180秒没心跳的实例(eureka的bug导致)