

源码流程图

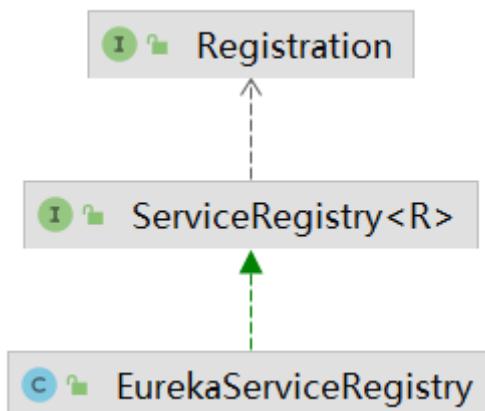
<https://processon.com/view/5f0409307d9c0844204efe9f?fromnew=1#pc>

服务提供者服务注册

spring cloud是一个生态，它提供了一套标准，这套标准可以通过不同的组件来实现，其中就包含服务注册/发现、熔断、负载均衡等，在spring-cloud-common这个包中，`org.springframework.cloud.serviceregistry`路径下，可以看到一个服务注册的接口定义`ServiceRegistry`。它就是定义了spring cloud中服务注册的一个接口。

```
public interface ServiceRegistry<R extends Registration> {  
    void register(R registration);  
  
    void deregister(R registration);  
  
    void close();  
  
    void setStatus(R registration, String status);  
  
    <T> T getStatus(R registration);  
}
```

我们看一下它的类关系图，这个接口有一个唯一的实现`EurekaServiceRegistry`。表示采用的是Eureka Server作为服务注册中心。



自动注册的触发机制

Eureka自动注册，是通过`EurekaAutoServiceRegistration`这个对象来触发的。

在Spring Boot项目启动时，会基于自动装配机制，在`EurekaClientAutoConfiguration`这个配置类中，初始化一个`EurekaAutoServiceRegistration`这个Bean对象，代码如下。

```

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

```

EurekaAutoServiceRegistration这个类的定义如下。

```

public class EurekaAutoServiceRegistration implements AutoServiceRegistration,
SmartLifecycle, Ordered, SmartApplicationListener {
    //省略
    @Override
    public void start() {
        // only set the port if the nonSecurePort or securePort is 0 and this.port != 0
        if (this.port.get() != 0) {
            if (this.registration.getNonSecurePort() == 0) {
                this.registration.setNonSecurePort(this.port.get());
            }

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

        // only initialize if nonSecurePort is greater than 0 and it isn't already running
        // because of containerPortInitializer below
        if (!this.running.get() && this.registration.getNonSecurePort() > 0) {

            this.serviceRegistry.register(this.registration);

            this.context.publishEvent(new InstanceRegisteredEvent<>(this,
                this.registration.getInstanceConfig()));
            this.running.set(true);
        }
    }
    //省略...
}

```

我们发现，EurekaAutoServiceRegistration实现了 SmartLifecycle 接口，当 Spring 容器加载完所有的 Bean 并且初始化之后，会继续回调实现了 SmartLifeCycle 接口的类中对应的方法，比如 (start)。

SmartLifeCycle 知识拓展

我拓展一下 SmartLifeCycle 这块的知识， SmartLifeCycle 是一个接口，当 Spring 容器加载完所有的 Bean 并且初始化之后，会继续回调实现了 SmartLifeCycle 接口的类中对应的方法，比如 (start)。

实际上我们自己也可以拓展，比如在springboot工程的main方法同级目录下，写一个测试类，实现SmartLifecycle接口，并且通过@Service声明为一个bean，因为要被spring去加载，首先得是bean。

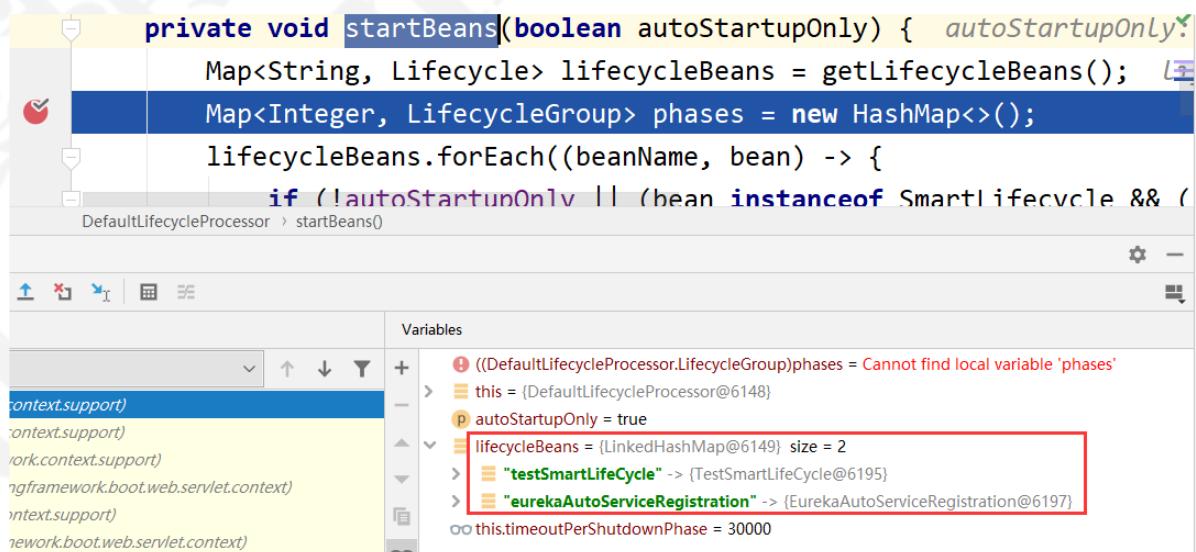
```
@Service
public class TestSmartLifecycle implements SmartLifecycle {
    @Override
    public void start() {
        System.out.println("start");
    }

    @Override
    public void stop() {
        System.out.println("stop");
    }

    @Override
    public boolean isRunning() {
        return false;
    }
}
```

接着，我们启动spring boot应用后，可以看到控制台输出了start字符串。

我们在DefaultLifecycleProcessor.startBeans方法上加一个debug，可以很明显的看到我们自己定义的TestSmartLifecycle被扫描到了，并且最后会调用该bean的start方法。



在`startBeans`方法中，我们可以看到它首先会获得所有实现了`SmartLifecycle`的Bean，然后会循环调用实现了`SmartLifecycle`的bean的`start`方法，代码如下。

```
private void startBeans(boolean autoStartupOnly) {
    Map<String, Lifecycle> lifecycleBeans = this.getLifecycleBeans();
    Map<Integer, DefaultLifecycleProcessor.LifecycleGroup> phases = new HashMap();
    lifecycleBeans.forEach((beanName, bean) -> {
        if (!autoStartupOnly || bean instanceof SmartLifecycle &&
        ((SmartLifecycle)bean).isAutoStartup()) {
            int phase = this.getPhase(bean);
            DefaultLifecycleProcessor.LifecycleGroup group =
            (DefaultLifecycleProcessor.LifecycleGroup)phases.get(phase);
            if (group == null) {
                group = new DefaultLifecycleProcessor.LifecycleGroup(phase,
                this.timeoutPerShutdownPhase, lifecycleBeans, autoStartupOnly);
                phases.put(phase, group);
            }
        }
    })
}
```

```

        group.add(beanName, bean);
    }

});

if (!phases.isEmpty()) {
    List<Integer> keys = new ArrayList(phases.keySet());
    Collections.sort(keys);
    Iterator var5 = keys.iterator();

    while(var5.hasNext()) {
        Integer key = (Integer)var5.next();
        ((DefaultLifecycleProcessor.LifecycleGroup)phases.get(key)).start(); //循环
调用实现了SmartLifeCycle接口的start方法。
    }
}

}

```

SmartLifeCycle接口的回调，是在SpringBoot启动时触发，具体的执行路径如下！

```

.SpringApplication.run() -> this.refreshContext(context);-
>this.refresh(context);->ServletWebServerApplicationContext.refresh()-
>this.finishRefresh();->AbstractApplicationContext.finishRefresh-
>DefaultLifecycleProcessor.onRefresh()-> this.startBeans-
>this.start()->this.doStart()->

```

服务注册

因此，当SpringBoot启动时，会触发在EurekaAutoServiceRegistration中的start方法，代码如下。

```

public class EurekaAutoServiceRegistration implements AutoServiceRegistration,
SmartLifecycle, Ordered, SmartApplicationListener {
    //省略
    @Override
    public void start() {
        // only set the port if the nonSecurePort or securePort is 0 and this.port != 0
        if (this.port.get() != 0) {
            if (this.registration.getNonSecurePort() == 0) {
                this.registration.setNonSecurePort(this.port.get());
            }

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

        // only initialize if nonSecurePort is greater than 0 and it isn't already
running
        // because of containerPortInitializer below
        if (!this.running.get() && this.registration.getNonSecurePort() > 0) {
            //实现服务注册。
            this.serviceRegistry.register(this.registration);
            //发布一个事件
            this.context.publishEvent(new InstanceRegisteredEvent<>(this,

```

```

        this.registration.getInstanceConfig());
        this.running.set(true);
    }
}
//省略...
}

```

EurekaServiceRegistry.register

`this.serviceRegistry.register(this.registration);`, 实际调用的是 `EurekaServiceRegistry` 这个对象中的 `register` 方法, 代码如下。

```

public class EurekaServiceRegistry implements ServiceRegistry<EurekaRegistration> {

    private static final Log log = LoggerFactory.getLog(EurekaServiceRegistry.class);

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

        if (log.isInfoEnabled()) {
            log.info("Registering application "
                    + reg.getApplicationInfoManager().getInfo().getAppName()
                    + " with eureka with status "
                    + reg.getInstanceConfig().getInitialStatus());
        }
        //设置当前实例的状态, 一旦这个实例的状态发生变化, 只要状态不是DOWN, 那么就会被监听器监听并且
        //执行服务注册。
        reg.getApplicationInfoManager()
            .setInstanceStatus(reg.getInstanceConfig().getInitialStatus());
        //设置健康检查的处理
        reg.getHealthCheckHandler().ifAvailable(healthCheckHandler -> reg
            .getEurekaClient().registerHealthCheck(healthCheckHandler));
    }
}

```

从上述代码来看, 注册方法中并没有真正调用Eureka的方法去执行注册, 而是仅仅设置了一个状态以及设置健康检查处理器。我们继续看一下

`reg.getApplicationInfoManager().setInstanceStatus` 方法。

```

public synchronized void setInstanceStatus(InstanceStateStatus status) {
    InstanceStatus next = instanceStatusMapper.map(status);
    if (next == null) {
        return;
    }

    InstanceStatus prev = instanceInfo.setStatus(next);
    if (prev != null) {
        for (StatusChangeListener listener : listeners.values()) {
            try {
                listener.notify(new StatusChangeEvent(prev, next));
            } catch (Exception e) {
                logger.warn("failed to notify listener: {}", listener.getId(), e);
            }
        }
    }
}

```

在这个方法中，它会通过监听器来发布一个状态变更事件。ok，此时listener的实例是 `StatusChangeListener`，也就是调用 `StatusChangeListener` 的 `notify()` 方法。这个事件是触发一个服务状态变更，应该是有地方会监听这个事件，然后基于这个事件。

这个时候我们以为找到了方向，然后点击进去一看，乍击，发现它是一个接口。而且我们发现它是静态的内部接口，还无法直接看到它的实现类。

依我多年源码阅读经验，于是又往回找，因为我基本上能猜测到一定是在某个地方做了初始化的工作，于是，我想找到 `EurekaServiceRegistry.register` 方法中的 `reg.getApplicationInfoManager` 这个实例是什么，而且我们发现 `ApplicationInfoManager` 是来自于 `EurekaRegistration` 这个类中的属性。

```
public class EurekaRegistration implements Registration {  
  
    private final ApplicationInfoManager applicationInfoManager;  
  
    private ObjectProvider<HealthCheckHandler> healthCheckHandler;  
  
    private EurekaRegistration(CloudEurekaInstanceConfig instanceConfig,  
                               EurekaClient eurekaClient, ApplicationInfoManager  
applicationInfoManager,  
                               ObjectProvider<HealthCheckHandler> healthCheckHandler)  
{  
        this.eurekaClient = eurekaClient;  
        this.instanceConfig = instanceConfig;  
        this.applicationInfoManager = applicationInfoManager;  
        this.healthCheckHandler = healthCheckHandler;  
    }  
}
```

而 `EurekaRegistration` 又是在 `EurekaAutoServiceRegistration` 这个类中实例化的。

那我们去 `EurekaAutoServiceRegistration` 这个配置类中，找一下 `applicationInfoManager` 的实例化过程，代码如下：

```
@Configuration(proxyBeanMethods = false)  
 @ConditionalOnMissingRefreshScope  
protected static class EurekaClientConfiguration {  
  
    @Bean  
    @ConditionalOnMissingBean(value = ApplicationInfoManager.class,  
                           search = SearchStrategy.CURRENT)  
    @org.springframework.cloud.context.config.annotation.RefreshScope  
    @Lazy  
    public ApplicationInfoManager eurekaApplicationInfoManager(  
        EurekaInstanceConfig config) {  
        InstanceInfo instanceInfo = new InstanceInfoFactory().create(config);  
        return new ApplicationInfoManager(config, instanceInfo); //构建了一个  
ApplicationInfoManager实例。  
    }  
}
```

在 `ApplicationInfoManager` 的构造方法中，初始化了一个 `listeners` 对象，它是一个 `ConcurrentHashMap` 集合，但是初始化的时候，这个集合并没有赋值。

```
@Inject  
public ApplicationInfoManager(EurekaInstanceConfig config, InstanceInfo instanceInfo,  
OptionalArgs optionalArgs) {  
    this.config = config;
```

```

    this.instanceInfo = instanceInfo;
    this.listeners = new ConcurrentHashMap<String, StatusChangeListener>();
    if (optionalArgs != null) {
        this.instanceStatusMapper = optionalArgs.getInstanceStatusMapper();
    } else {
        this.instanceStatusMapper = NO_OP_MAPPER;
    }

    // Hack to allow for getInstance() to use the DI'd ApplicationInfoManager
    instance = this;
}

```

遇到这个问题，我们先别慌，先来看一下ApplicationInfoManager这个类中对listeners赋值的方法如下。

```

public void registerStatusChangeListener(StatusChangeListener listener) {
    listeners.put(listener.getId(), listener);
}

```

这个方法唯一的调用方是：DiscoveryClient.initScheduledTasks方法。

这个方法又是在哪里调用的呢？

DiscoveryClient

在EurekaClientAutoConfiguration这个自动配置类的静态内部类EurekaClientConfiguration中，通过@Bean注入了一个CloudEurekaClient实例，代码如下。

```

@Configuration(proxyBeanMethods = false)
@ConditionalOnMissingRefreshScope
protected static class EurekaClientConfiguration {

    @Autowired
    private ApplicationContext context;

    @Autowired
    private AbstractDiscoveryClientOptionalArgs<?> optionalArgs;

    @Bean(destroyMethod = "shutdown")
    @ConditionalOnMissingBean(value = EurekaClient.class,
                           search = SearchStrategy.CURRENT)
    public EurekaClient eurekaClient(ApplicationInfoManager manager,
                                      EurekaClientConfig config) {
        return new CloudEurekaClient(manager, config, this.optionalArgs,
                                     this.context);
    }
}

```

从名字不难猜测出，EurekaClient应该是专门负责和EurekaServer进行交互的客户端实现类，而这里返回的实例对象是CloudEurekaClient，构造代码如下。

```

public CloudEurekaClient(ApplicationInfoManager applicationInfoManager,
                         EurekaClientConfig config,
                         AbstractDiscoveryClientOptionalArgs<?> args,
                         ApplicationEventPublisher publisher) {
    super(applicationInfoManager, config, args);
    this.applicationInfoManager = applicationInfoManager;
    this.publisher = publisher;
    this.eurekaTransportField = ReflectionUtils.findField(DiscoveryClient.class,
                                                          "eurekaTransport");
    ReflectionUtils.makeAccessible(this.eurekaTransportField);
}

```

注意，在`CloudEurekaClient`这个类的构造方法中，传递了`ApplicationInfoManager`这个实例，后续会用到。

同时，该构造方法中会同步调用`super(applicationInfoManager, config, args);`，也就是调用父类`DiscoveryClient`的构造方法，代码如下。

```

public DiscoveryClient(ApplicationInfoManager applicationInfoManager, final
EurekaClientConfig config, AbstractDiscoveryClientOptionalArgs args) {
    this(applicationInfoManager, config, args, ResolverUtils::randomize);
}

```

最终会调用`DiscoveryClient`中重载的如下方法，代码比较长，把非关键代码省略。

```

DiscoveryClient(ApplicationInfoManager applicationInfoManager, EurekaClientConfig
config, AbstractDiscoveryClientOptionalArgs args,
                 Provider<BackupRegistry> backupRegistryProvider, EndpointRandomizer
endpointRandomizer) {
    //省略....
    if (config.shouldFetchRegistry()) { //是否要从eureka server上获取服务地址信息
        this.registryStalenessMonitor = new ThresholdLevelsMetric(this,
METRIC_REGISTRY_PREFIX + "lastUpdateSec_", new long[]{15L, 30L, 60L, 120L, 240L,
480L});
    } else {
        this.registryStalenessMonitor = ThresholdLevelsMetric.NO_OP_METRIC;
    }
    //是否要注册到eureka server上
    if (config.shouldRegisterWithEureka()) {
        this.heartbeatStalenessMonitor = new ThresholdLevelsMetric(this,
METRIC_REGISTRATION_PREFIX + "lastHeartbeatSec_", new long[]{15L, 30L, 60L, 120L,
240L, 480L});
    } else {
        this.heartbeatStalenessMonitor = ThresholdLevelsMetric.NO_OP_METRIC;
    }
    logger.info("Initializing Eureka in region {}", clientConfig.getRegion());
    //如果不需要注册并且不需要更新服务地址
    if (!config.shouldRegisterWithEureka() && !config.shouldFetchRegistry()) {
        logger.info("Client configured to neither register nor query for data.");
        scheduler = null;
        heartbeatExecutor = null;
        cacheRefreshExecutor = null;
        eurekaTransport = null;
        instanceRegionChecker = new InstanceRegionChecker(new
PropertyBasedAzToRegionMapper(config), clientConfig.getRegion());
        // 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();
        initRegistrySize = this.getApplications().size();
        registrySize = initRegistrySize;
        logger.info("Discovery Client initialized at timestamp {} with initial
instances count: {}",
                    initTimestampMs, initRegistrySize);

        return; // no need to setup up an network tasks and we are done
    }

    try {
        // default size of 2 - 1 each for heartbeat and cacheRefresh
        //构建一个延期执行的线程池
        scheduler = Executors.newScheduledThreadPool(2,
                                                    new ThreadFactoryBuilder()
                                                        .setNameFormat("DiscoveryClient-
%d"))
                                                    .setDaemon(true)
                                                    .build());

        //处理心跳的线程池
        heartbeatExecutor = new ThreadPoolExecutor(
            1, clientConfig.getHeartbeatExecutorThreadPoolSize(), 0, TimeUnit.SECONDS,
            new SynchronousQueue<Runnable>(),
            new ThreadFactoryBuilder()
                .setNameFormat("DiscoveryClient-HeartbeatExecutor-%d")
                .setDaemon(true)
                .build());
        ); // use direct handoff
        //处理缓存刷新的线程池
        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);
    }
}

```

```
//如果需要注册到Eureka server并且是开启了初始化的时候强制注册，则调用register()发起服务
注册
if (clientConfig.shouldFetchRegistry()) {
    try {
        //从Eureka-Server中拉去注册地址信息
        boolean primaryFetchRegistryResult = fetchRegistry(false);
        if (!primaryFetchRegistryResult) {
            logger.info("Initial registry fetch from primary servers failed");
        }
        //从备用地址拉去服务注册信息
        boolean backupFetchRegistryResult = true;
        if (!primaryFetchRegistryResult && !fetchRegistryFromBackup()) {
            backupFetchRegistryResult = false;
            logger.info("Initial registry fetch from backup servers failed");
        }
        //如果还是没有拉取到，并且配置了强制拉取注册表的话，就会抛异常
        if (!primaryFetchRegistryResult && !backupFetchRegistryResult &&
clientConfig.shouldEnforceFetchRegistryAtInit()) {
            throw new IllegalStateException("Fetch registry error at startup.
Initial fetch failed.");
        }
    } catch (Throwable th) {
        logger.error("Fetch registry error at startup: {}", th.getMessage());
        throw new IllegalStateException(th);
    }
}

// call and execute the pre registration handler before all background tasks (inc
registration) is started
//这里是判断一下有没有预注册处理器，有的话就执行一下
if (this.preRegistrationHandler != null) {
    this.preRegistrationHandler.beforeRegistration();
}

//如果需要注册到Eureka server并且是开启了初始化的时候强制注册，则调用register()发起服
务注册(默认情况下，shouldEnforceRegistrationAtInit为false)
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);
    }
}

// finally, init the schedule tasks (e.g. cluster resolvers, heartbeat,
instanceInfo replicator, fetch
//初始化一个定时任务，负责心跳、实例数据更新
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();
    initRegistrySize = this.getApplications().size();
    registrySize = initRegistrySize;
    logger.info("Discovery Client initialized at timestamp {} with initial instances count: {}",
                initTimestampMs, initRegistrySize);
}

```

DiscoveryClient.initScheduledTasks

`initScheduledTasks`去启动一个定时任务。

- 如果配置了开启从注册中心刷新服务列表，则会开启cacheRefreshExecutor这个定时任务
- 如果开启了服务注册到Eureka，则通过需要做几个事情。
 - 建立心跳检测机制
 - 通过内部类来实例化StatusChangeListener 实例状态监控接口，这个就是前面我们在分析启动过程中所看到的，调用notify的方法，实际上会在这里体现。

```

private void initScheduledTasks() {
    //如果配置了开启从注册中心刷新服务列表，则会开启cacheRefreshExecutor这个定时任务
    if (clientConfig.shouldFetchRegistry()) {
        // registry cache refresh timer
        //registryFetchIntervalSeconds:30s
        int registryFetchIntervalSeconds =
clientConfig.getRegistryFetchIntervalSeconds();
        //expBackOffBound:10
        int expBackOffBound =
clientConfig.getCacheRefreshExecutorExponentialBackOffBound();
        cacheRefreshTask = new TimedSupervisorTask(
            "cacheRefresh",
            scheduler,
            cacheRefreshExecutor,
            registryFetchIntervalSeconds,
            TimeUnit.SECONDS,
            expBackOffBound,
            new CacheRefreshThread()
        );
        scheduler.schedule(
            cacheRefreshTask,
            registryFetchIntervalSeconds, TimeUnit.SECONDS);
    }
    //如果开启了服务注册到Eureka，则通过需要做几个事情
    if (clientConfig.shouldRegisterWithEureka()) {
        int renewalIntervalInSecs =
instanceInfo.getLeaseInfo().getRenewalIntervalInSecs();
        int expBackOffBound =
clientConfig.getHeartbeatExecutorExponentialBackOffBound();
        logger.info("Starting heartbeat executor: " + "renew interval is: {}",
renewalIntervalInSecs);

        // 开启一个心跳任务
        heartbeatTask = new TimedSupervisorTask(
            "heartbeat",
            scheduler,

```

```

        heartbeatExecutor,
        renewalIntervalInSecs,
        TimeUnit.SECONDS,
        expBackOffBound,
        new HeartbeatThread()
    );
    scheduler.schedule(
        heartbeatTask,
        renewalIntervalInSecs, TimeUnit.SECONDS);

    //创建一个instanceInfoReplicator实例信息复制器
    instanceInfoReplicator = new InstanceInfoReplicator(
        this,
        instanceInfo,
        clientConfig.getInstanceInfoReplicationIntervalSeconds(),
        2); // burstSize
    //初始化一个状态变更监听器
    statusChangeListener = new ApplicationInfoManager.StatusChangeListener() {
        @Override
        public String getId() {
            return "statusChangeListener";
        }

        @Override
        public void notify(StatusChangeEvent statusChangeEvent) {
            logger.info("Saw local status change event {}", statusChangeEvent);
            instanceInfoReplicator.onDemandUpdate();
        }
    };
    //注册实例状态变化的监听
    if (clientConfig.shouldOnDemandUpdateStatusChange()) {
        applicationInfoManager.registerStatusChangeListener(statusChangeListener);
    }
    //注意 (case)
}
//启动一个实例信息复制器，主要就是为了开启一个定时线程，每40秒判断实例信息是否变更，如果变更了则重新注册

instanceInfoReplicator.start(clientConfig.getInitialInstanceInfoReplicationIntervalSeconds());
} else {
    logger.info("Not registering with Eureka server per configuration");
}
}
}

```

在上述代码中，我们发现了一个很重要的逻辑

逻辑：applicationInfoManager.registerStatusChangeListener(statusChangeListener);

这个代码是注册一个StatusChangeListener，保存到ApplicationInfoManager中的listener集合中。（还记得前面源码分析中的服务注册逻辑吗？当服务器启动或者停止时，会调用ApplicationInfoManager.listener，逐个遍历调用listener.notify方法），而这个listener集合中的对象是在DiscoveryClient初始化的时候完成的。

instanceInfoReplicator.onDemandUpdate()

这个方法的主要作用是根据实例数据是否发生变化，来触发服务注册中心的数据。

```

public boolean onDemandUpdate() {
    if (rateLimiter.acquire(burstSize, allowedRatePerMinute)) { //限流判断
        if (!scheduler.isShutdown()) { //提交一个任务

```

```

scheduler.submit(new Runnable() {
    @Override
    public void run() {
        logger.debug("Executing on-demand update of local InstanceInfo");
        //取出之前已经提交的任务，也就是在start方法中提交的更新任务，如果任务还没有执行完成，则取消之前的任务。
        Future latestPeriodic = scheduledPeriodicRef.get();
        if (latestPeriodic != null && !latestPeriodic.isDone()) {
            logger.debug("Canceling the latest scheduled update, it will
be rescheduled at the end of on demand update");
            latestPeriodic.cancel(false); //如果此任务未完成，就立即取消
        }
        //通过调用run方法，令任务在延时后执行，相当于周期性任务中的一次
        InstanceInfoReplicator.this.run();
    }
});
return true;
} else {
    logger.warn("Ignoring onDemand update due to stopped scheduler");
    return false;
}
} else {
    logger.warn("Ignoring onDemand update due to rate limiter");
    return false;
}
}
}
}

```

InstanceInfoReplicator.this.run();

run方法调用register方法进行服务注册，并且在finally中，每30s会定时执行一下当前的run方法进行检查。

```

public void run() {
    try {
        //刷新实例信息
        discoveryClient.refreshInstanceInfo();
        //是否有状态更新过了，有的话获取更新的时间
        Long dirtyTimestamp = instanceInfo.isDirtyWithTime();
        if (dirtyTimestamp != null) { //有脏数据，要重新注册
            discoveryClient.register();
            instanceInfo.unsetIsDirty(dirtyTimestamp);
        }
    } catch (Throwable t) {
        logger.warn("There was a problem with the instance info replicator", t);
    } finally {
        //每隔30s，执行一次当前的`run`方法
        Future next = scheduler.schedule(this, replicationIntervalSeconds,
        TimeUnit.SECONDS);
        scheduledPeriodicRef.set(next);
    }
}

```

DiscoveryClient.register

记过上述分析后，最终我们找到了Eureka的服务注册方法：`eurekaTransport.registrationClient.register`，最终调用的是`AbstractJerseyEurekaHttpClient#register(...)`。

```
boolean register() throws Throwable {
```

```

logger.info(PREFIX + "{}: registering service...", appPathIdentifier);
EurekaHttpResponse<Void> httpResponse;
try {
    httpResponse = eurekaTransport.registrationClient.register(instanceInfo);
} catch (Exception e) {
    logger.warn(PREFIX + "{} - registration failed {}", appPathIdentifier,
e.getMessage(), e);
    throw e;
}
if (logger.isInfoEnabled()) {
    logger.info(PREFIX + "{} - registration status: {}", appPathIdentifier,
httpResponse.getStatusCode());
}
return httpResponse.getStatusCode() == Status.NO_CONTENT.getStatusCode();
}

```

AbstractJerseyEurekaHttpClient#register

很显然，这里是发起了一次http请求，访问Eureka-Server的`apps/${APP_NAME}`接口，将当前服务实例的信息发送到Eureka Server进行保存。

至此，我们基本上已经知道Spring Cloud Eureka 是如何在启动的时候把服务信息注册到Eureka Server上的了

```

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

```

服务注册总结

服务注册的过程分两个步骤

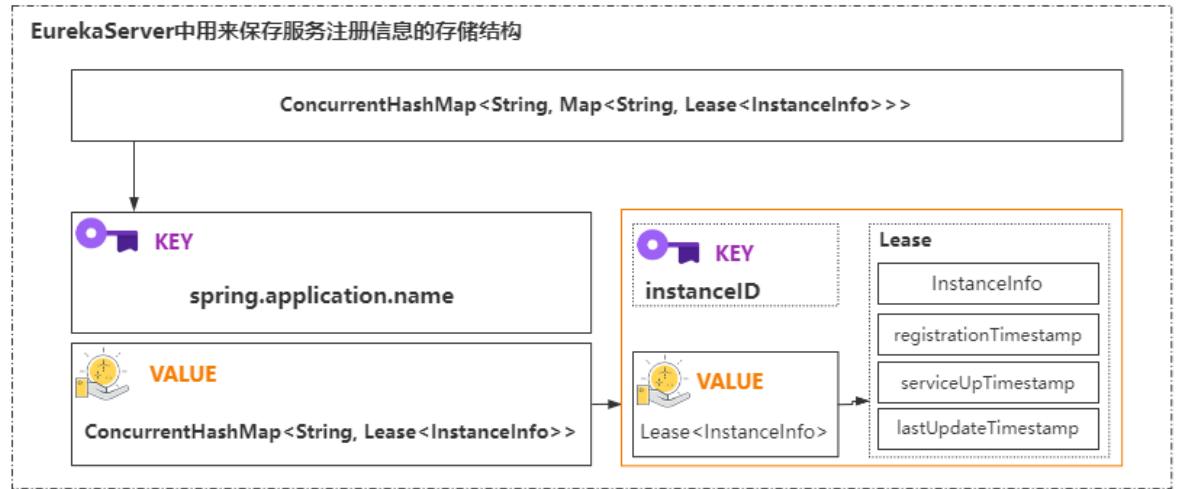
1. DiscoveryClient这个对象，在初始化时，调用`initScheduledTask()`方法，构建一个`StatusChangeListener`监听。
2. Spring Cloud应用在启动时，基于`SmartLifecycle`接口回调，触发`StatusChangeListener`事件通知

3. 在StatusChangeListener的回调方法中，通过调用onDemandUpdate方法，去更新客户端的地址信息，从而完成服务注册。

Eureka注册信息如何存储？

Eureka Server收到客户端的服务注册请求后，需要把信息存储到Eureka Server中，它的存储结构如下图所示。

EurekaServer采用了ConcurrentHashMap集合的方式。来存储服务提供者的地址信息，其中，每个节点的实例信息的最终存储对象是InstanceInfo。>



Eureka Server接收请求处理

请求入口在：

`com.netflix.eureka.resources.ApplicationResource.addInstance()`。

大家可以发现，这里所提供的REST服务，采用的是jersey来实现的。Jersey是基于JAX-RS标准，提供REST的实现的支持，这里就不展开分析了。

Eureka Server端定义的服务注册接口实现如下：

```
@POST
@Consumes({"application/json", "application/xml"})
public Response addInstance(InstanceInfo info,
                            @HeaderParam(PeerEurekaNode.HEADER_REPLICATION) String
isReplication) {
    logger.debug("Registering instance {} (replication={})", info.getId(),
isReplication);

    // handle cases where clients may be registering with bad DataCenterInfo with
    missing data
    //实例部署的数据中心， 这里是AWS实现的数据相关的逻辑，这里不涉及到，所以不需要去关心
    DataCenterInfo dataCenterInfo = info.getDataCenterInfo();
    if (dataCenterInfo instanceof UniqueIdentifier) {
        String dataCenterInfoId = ((UniqueIdentifier) dataCenterInfo).getId();
        if (isBlank(dataCenterInfoId)) {
            boolean experimental =
"true".equalsIgnoreCase(serverConfig.getExperimental("registration.validation.dataCent
erInfoId"));
            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());
    }
}
}

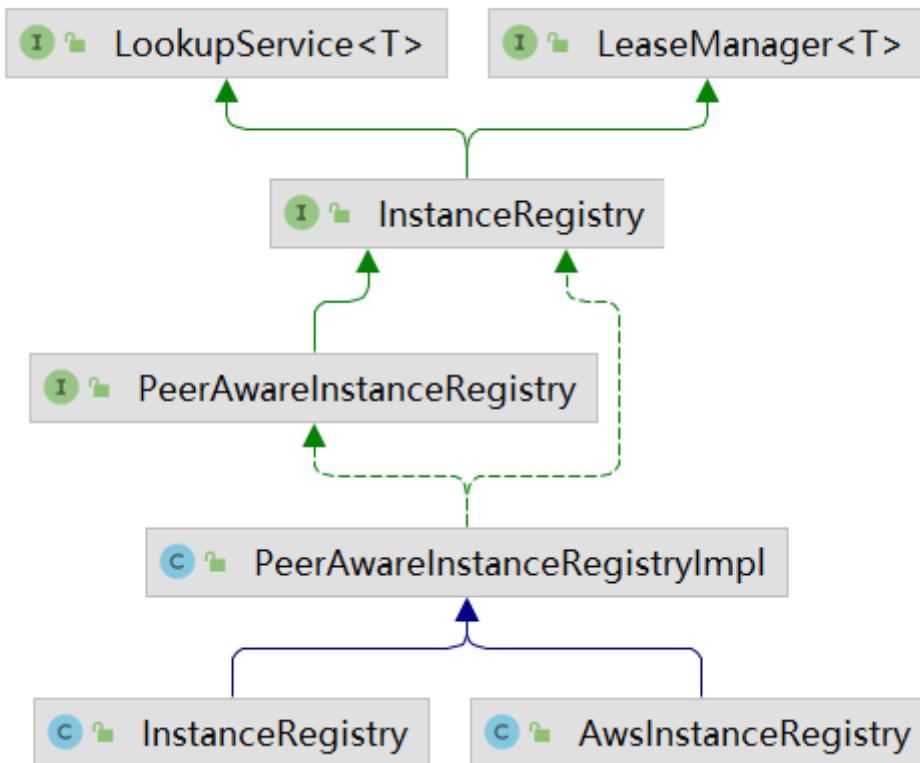
//在这里会调用服务注册方法，传递 `info`，表示客户端的服务实例信息。
registry.register(info, "true".equals(isReplication));
return Response.status(204).build(); // 204 to be backwards compatible
}

```

PeerAwareInstanceRegistryImpl.register

我们先来看PeerAwareInstanceRegistryImpl的类关系图，从类关系图可以看出，PeerAwareInstanceRegistry的最顶层接口为LeaseManager与LookupService，

- 其中LookupService定义了最基本的发现实例的行为。
- LeaseManager定义了处理客户端注册，续约，注销等操作。



InstanceRegistry.register

接着进入到InstanceRegistry的register方法，在这个方法中，增加了一个handleRegistration方法的调用，这个方法用来发布一个EurekaInstanceRegisteredEvent事件。

```
@Override  
public void register(final InstanceInfo info, final boolean isReplication) {  
    handleRegistration(info, resolveInstanceLeaseDuration(info), isReplication);  
    super.register(info, isReplication);  
}
```

父类的register方法

接着调用父类PeerAwareInstanceRegistryImpl的register方法，代码如下。

```
@Override  
public void register(final InstanceInfo info, final boolean isReplication) {  
    int leaseDuration = Lease.DEFAULT_DURATION_IN_SECS; //租约过期时间  
    if (info.getLeaseInfo() != null && info.getLeaseInfo().getDurationInSecs() > 0) {  
        //如果客户端有自己定义心跳超时时间，则采用客户端的  
        leaseDuration = info.getLeaseInfo().getDurationInSecs();  
    }  
    super.register(info, leaseDuration, isReplication); //节点注册  
    replicateToPeers(Action.Register, info.getAppName(), info.getId(), info, null,  
    isReplication); //把注册信息同步到其他集群节点。  
}
```

其中：

- leaseDuration 表示租约过期时间，默认是90s，也就是当服务端超过90s没有收到客户端的心跳，则主动剔除该节点
- 调用super.register发起节点注册
- 将信息复制到Eureka Server集群中的其他机器上，同步的实现也很简单，就是获得集群中的所有节点，然后逐个发起注册

AbstractInstanceRegistry.register

最终在这个抽象类的实例注册类中完成服务注册的实现，代码如下。

```
public void register(InstanceInfo registrant, int leaseDuration, boolean  
isReplication) {  
    read.lock();  
    try {  
        //从registry中获得当前实例信息，根据appName， registry中保存了所有客户端的实例数据  
        Map<String, Lease<InstanceInfo>> gMap = registry.get(registrant.getAppName());  
        REGISTER.increment(isReplication); //原子递增，做数据统计  
        if (gMap == null) { //如果gMap为空，说明当前服务端没有保存该实例数据，则通过下面代码  
            //进行初始化  
            final ConcurrentHashMap<String, Lease<InstanceInfo>> gNewMap = new  
            ConcurrentHashMap<String, Lease<InstanceInfo>>();  
            gMap = registry.putIfAbsent(registrant.getAppName(), gNewMap);  
            if (gMap == null) {  
                gMap = gNewMap;  
            }  
        }  
        //从gMap中查询已经存在的Lease信息，Lease中文翻译为租约，实际上它把服务提供者的实例信  
息包装成了一个lease，里面提供了对于改服务实例的租约管理
```

```

        Lease<InstanceInfo> existingLease = gMap.get(registrant.getId());
        // 当instance已经存在时，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
        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.
            // 比较lastDirtyTimestamp，以lastDirtyTimestamp大的为准
            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(); //重新赋值registrant为服务端最
                新的实例信息
            }
        } else {
            // 如果lease不存在，则认为是一个新的实例信息，执行下面这段代码（后续单独分析它的
            作用）
            synchronized (lock) {
                if (this.expectedNumberOfClientsSendingRenews > 0) {
                    // Since the client wants to register it, increase the number of
                    clients sending renew
                    this.expectedNumberOfClientsSendingRenews =
                    this.expectedNumberOfClientsSendingRenews + 1;
                    updateRenewPerMinThreshold();
                }
            }
            logger.debug("No previous lease information found; it is new
            registration");
        }
        //创建一个Lease租约信息
        Lease<InstanceInfo> lease = new Lease<InstanceInfo>(registrant,
        leaseDuration);
        if (existingLease != null) { // 当原来存在Lease的信息时，设置serviceUpTimestamp，
        保证服务启动的时间一直是第一次注册的那个（避免状态变更影响到服务启动时间）
            lease.setServiceUpTimestamp(existingLease.getServiceUpTimestamp());
        }
        gMap.put(registrant.getId(), lease); //把当前服务实例保存到gMap中。

        recentRegisteredQueue.add(new Pair<Long, String>(
            System.currentTimeMillis(),
            registrant.getAppName() + "(" + registrant.getId() + ")");
        // This is where the initial state transfer of overridden status happens
        //如果实例状态不等于UNKNOWN，则把当前实例状态添加到overriddenInstanceStateMap中
        if (!InstanceState.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 (!overriddenInstanceStateMap.containsKey(registrant.getId())) {
                logger.info("Not found overridden id {} and hence adding it",
                registrant.getId());
            }
        }
    }
}

```

```

        overriddenInstanceStatusMap.put(registrant.getId(),
registrant.getOverriddenStatus());
    }
}
//重写实例状态
InstanceState 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
InstanceState overriddenInstanceStatus =
getOverriddenInstanceStatus(registrant, existingLease, isReplication);
registrant.setStatusWithoutDirty(overriddenInstanceStatus); // 设置实例信息的状态, 但不标记 dirty

// If the lease is registered with UP status, set lease service up timestamp
if (InstanceState.UP.equals(registrant.getStatus())) { //如果服务实例信息为UP状态, 则更新该实例的启动时间。
    lease.serviceUp();
}
registrant.setActionType(ActionType.ADDED); // 设置注册类型为添加
recentlyChangedQueue.add(new RecentlyChangedItem(lease)); // 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
registrant.setLastUpdatedTimestamp(); //修改最后一次更新时间
//让缓存失效
invalidateCache(registrant.getAppName(), registrant.getVIPAddress(),
registrant.getSecureVipAddress());
logger.info("Registered instance {}/{} with status {} (replication={})",
            registrant.getAppName(), registrant.getId(),
            registrant.getStatus(), isReplication);
} finally {
    read.unlock();
}
}
}

```

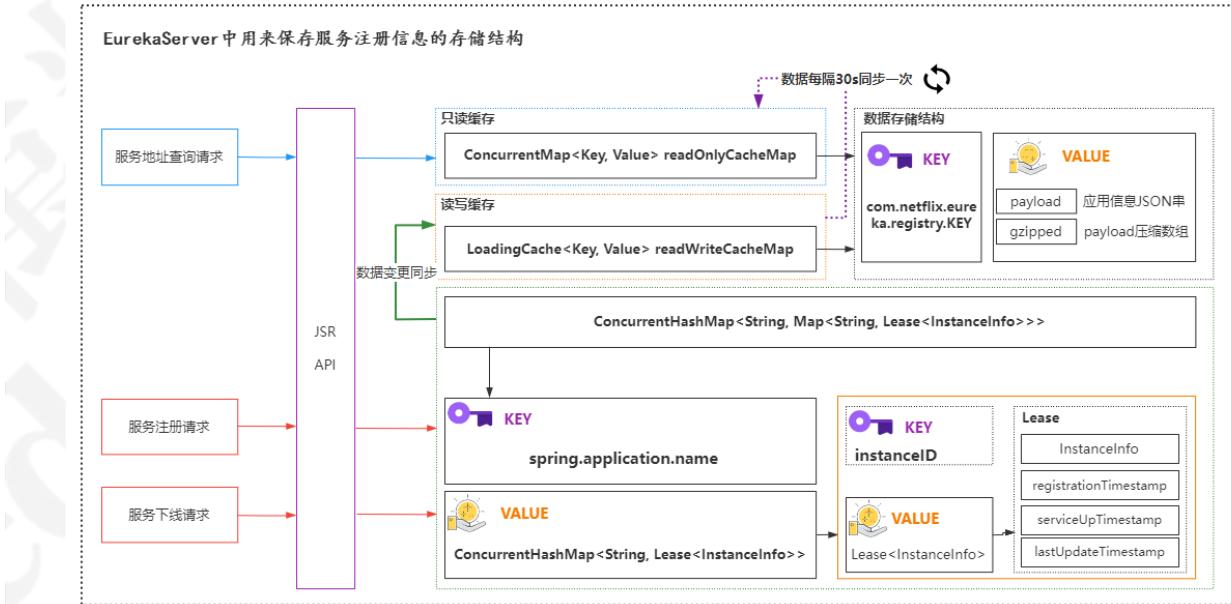
EurekaServer注册信息存储总结

至此，我们就把服务注册在客户端和服务端的处理过程做了一个详细的分析，实际上在 Eureka Server 端，会把客户端的地址信息保存到 ConcurrentHashMap 中存储。并且服务提供者和注册中心之间，会建立一个心跳检测机制。

用于监控服务提供者的健康状态。

Eureka多级缓存设计

Eureka Server 为了提供响应效率，提供了两层的缓存结构，将 Eureka Client 所需要的注册信息，直接存储在缓存结构中，实现原理如下图所示。



第一层缓存: readOnlyCacheMap, 本质上是 ConcurrentHashMap, 依赖定时从 readWriteCacheMap 同步数据, 默认时间为 30 秒。

readOnlyCacheMap : 是一个 ConcurrentHashMap 只读缓存, 这个主要是为了供客户端获取注册信息时使用, 其缓存更新, 依赖于定时器的更新, 通过和 readWriteCacheMap 的值做对比, 如果数据不一致, 则以 readWriteCacheMap 的数据为准。

第二层缓存: readWriteCacheMap, 本质上是 Guava 缓存。

readWriteCacheMap: readWriteCacheMap 的数据主要同步于存储层。当获取缓存时判断缓存中是否没有数据, 如果不存在此数据, 则通过 CacheLoader 的 load 方法去加载, 加载成功之后将数据放入缓存, 同时返回数据。

readWriteCacheMap 缓存过期时间, 默认为 180 秒, 当服务下线、过期、注册、状态变更, 都会来清除此缓存中的数据。

Eureka Client 获取全量或者增量的数据时, 会先从一级缓存中获取; 如果一级缓存中不存在, 再从二级缓存中获取; 如果二级缓存也不存在, 这时候先将存储层的数据同步到缓存中, 再从缓存中获取。

通过 Eureka Server 的二层缓存机制, 可以非常有效地提升 Eureka Server 的响应时间, 通过数据存储层和缓存层的数据切割, 根据使用场景来提供不同的数据支持。

多级缓存的意义

这里为什么要设计多级缓存呢? 原因很简单, 就是当存在大规模的服务注册和更新时, 如果只是修改一个ConcurrentHashMap数据, 那么势必因为锁的存在导致竞争, 影响性能。

而Eureka又是AP模型, 只需要满足最终可用就行。所以它在这里用到多级缓存来实现读写分离。注册方法写的时候直接写内存注册表, 写完表之后主动失效读写缓存。

获取注册信息接口先从只读缓存取, 只读缓存没有再去读写缓存取, 读写缓存没有再去内存注册表里取(不只是取, 此处较复杂)。并且, 读写缓存会更新回写只读缓存

- responseCacheUpdateIntervalMs : readOnlyCacheMap 缓存更新的定时器时间间隔, 默认为30秒
- responseCacheAutoExpirationInSeconds : readWriteCacheMap 缓存过期时间, 默认为 180 秒。

缓存初始化

readWriteCacheMap使用的是LoadingCache对象，它是guava中提供的用来实现内存缓存的一个api。创建方式如下

```
>LoadingCache<Long, String> cache = CacheBuilder.newBuilder()
    //缓存池大小，在缓存项接近该大小时，Guava开始回收旧的缓存项
    .maximumSize(10000)
    //设置时间对象没有被读/写访问则对象从内存中删除(在另外的线程里面不定期维护)
    .expireAfterAccess(10, TimeUnit.MINUTES)
    //移除监听器,缓存项被移除时会触发
    .removalListener(new RemovalListener<Long, String>() {
        @Override
        public void onRemoval(RemovalNotification<Long, String> rn) {
            //执行逻辑操作
        }
    })
    .recordStats()//开启Guava Cache的统计功能
    .build(new CacheLoader<String, Object>() {
        @Override
        public Object load(String key) {
            //从SQL或者NoSql获取对象
        }
    });
//CacheLoader类 实现自动加载
```

其中，CacheLoader是用来实现缓存自动加载的功能，当触发`readWriteCacheMap.get(key)`方法时，就会回调`CacheLoader.load`方法，根据key去服务注册信息中去查找实例数据进行缓存

```
ResponseCacheImpl(EurekaServerConfig serverConfig, ServerCodecs serverCodecs,
AbstractInstanceRegistry registry) {
    this.serverConfig = serverConfig;
    this.serverCodecs = serverCodecs;
    this.shouldUseReadOnlyResponseCache =
serverConfig.shouldUseReadOnlyResponseCache();
    this.registry = registry;

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

    CacheBuilder.newBuilder().initialCapacity(serverConfig.getInitialCapacityOfResponseCa
che())
        .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);
                }
            }
        });
//CacheLoader类 实现自动加载
```

```

        Value value = generatePayload(key); //注意这里
        return value;
    }
});
}

```

而缓存的加载，是基于`generatePayload`方法完成的，代码如下。

```

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.getApplicationDeltas());
                    }
                } 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;
        }
    } finally {
        if (tracer != null) {
            tracer.stop();
        }
    }
}

```

此方法接受一个 `Key` 类型的参数，返回一个 `Value` 类型。其中 `Key` 中重要的字段有：

- `KeyType`，表示payload文本格式，有 `JSON` 和 `XML` 两种值。
- `EntityType`，表示缓存的类型，有 `Application`，`VIP`，`SVIP` 三种值。
- `entityName`，表示缓存的名称，可能是单个应用名，也可能是 `ALL_APPS` 或 `ALL_APPS_DELTA`。

`Value` 则有一个 `String` 类型的payload和一个 `byte` 数组，表示gzip压缩后的字节。

缓存同步

在 `ResponseCacheImpl` 这个类的构造实现中，初始化了一个定时任务，这个定时任务每个

```
ResponseCacheImpl(EurekaServerConfig serverConfig, ServerCodecs serverCodecs,
AbstractInstanceRegistry registry) {
    //省略...
    if (shouldUseReadOnlyResponseCache) {
        timer.schedule(getCacheUpdateTask(),
            new Date(((System.currentTimeMillis() /
responseCacheUpdateIntervalMs) * responseCacheUpdateIntervalMs)
                + responseCacheUpdateIntervalMs),
            responseCacheUpdateIntervalMs);
    }
}
```

默认每30s从 `readWriteCacheMap` 更新有差异的数据同步到 `readOnlyCacheMap` 中

```
private TimerTask getCacheUpdateTask() {
    return new TimerTask() {
        @Override
        public void run() {
            logger.debug("Updating the client cache from response cache");
            for (Key key : readOnlyCacheMap.keySet()) { //遍历只读集合
                if (logger.isDebugEnabled()) {
                    logger.debug("Updating the client cache from response cache for
key : {} {} {} {}", key.getEntityType(), key.getName(), key.getVersion(),
key.getType());
                }
                try {
                    CurrentRequestVersion.set(key.getVersion());
                    Value cacheValue = readWriteCacheMap.get(key);
                    Value currentCacheValue = readOnlyCacheMap.get(key);
                    if (cacheValue != currentCacheValue) { //判断差异信息，如果有差异，则
更新
                        readOnlyCacheMap.put(key, cacheValue);
                    }
                } catch (Throwable th) {
                    logger.error("Error while updating the client cache from response
cache for key {}", key.toStringCompact(), th);
                } finally {
                    CurrentRequestVersion.remove();
                }
            }
        }
    };
}
```

缓存失效

在AbstractInstanceRegistry.register这个方法中，当完成服务信息保存后，会调用invalidateCache失效缓存

```
public void register(InstanceInfo registrant, int leaseDuration, boolean  
isReplication) {  
    //....  
    invalidateCache(registrant.getAppName(), registrant.getVIPAddress(),  
registrant.getSecureVipAddress());  
    //....  
}
```

最终调用ResponseCacheImpl.invalidate方法，完成缓存的失效机制

```
public void invalidate(Key... keys) {  
    for (Key key : keys) {  
        logger.debug("Invalidating the response cache key : {} {} {} {}, {}",  
key.getEntityType(), key.getName(), key.getVersion(),  
key.getType(), key.getEurekaAccept());  
  
        readWriteCacheMap.invalidate(key);  
        Collection<Key> keysWithRegions = regionSpecificKeys.get(key);  
        if (null != keysWithRegions && !keysWithRegions.isEmpty()) {  
            for (Key keysWithRegion : keysWithRegions) {  
                logger.debug("Invalidating the response cache key : {} {} {} {}, {}",  
key.getEntityType(), key.getName(), key.getVersion(),  
key.getType(), key.getEurekaAccept());  
                readWriteCacheMap.invalidate(keysWithRegion);  
            }  
        }  
    }  
}
```

Eureka是如何判断一个服务不可用的呢？

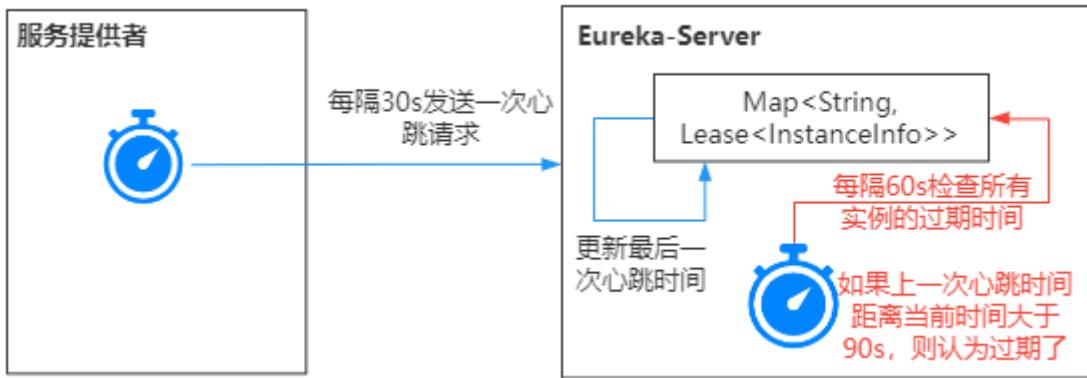
既然通过上述的案例分析到了服务不可用剔除的问题，那么大家是否知道，Eureka-Server是如何判断一个服务不可用的？

Eureka是通过心跳续约的方式来检查各个服务提供者的健康状态。

实际上，在判断服务不可用这个部分，会分为两块逻辑。

1. Eureka-Server需要定期检查服务提供者的健康状态。
2. Eureka-Client在运行过程中需要定期更新注册信息。

Eureka的心跳续约机制如下图所示。



1. 客户端在启动时，会开启一个心跳任务，每隔30s向服务单发送一次心跳请求。
2. 服务端维护了每个实例的最后一次心跳时间，客户端发送心跳包过来后，会更新这个心跳时间。
3. 服务端在启动时，开启了一个定时任务，该任务每隔60s执行一次，检查每个实例的最后一次心跳时间是否超过90s，如果超过则认为过期，需要剔除。

关于上述流程中涉及到的时间，可以通过以下配置来更改。

```
#Server 至上一次收到 Client 的心跳之后，等待下一次心跳的超时时间，在这个时间内若没收到下一次心跳，则将移除该 Instance。
eureka.instance.lease-expiration-duration-in-seconds=90
# Server 清理无效节点的时间间隔，默认6000毫秒，即60秒。
eureka.server.eviction-interval-timer-in-ms=60
```

客户端心跳发起流程

心跳续约是客户端发起的，每隔30s执行一次。

DiscoveryClient.initScheduledTasks

继续回到 `DiscoveryClient.initScheduledTasks` 方法中，

```
private void initScheduledTasks() {
    //省略....
    heartbeatTask = new TimedSupervisorTask(
        "heartbeat",
        scheduler,
        heartbeatExecutor,
        renewalIntervalInSecs,
        TimeUnit.SECONDS,
        expBackOffBound,
        new HeartbeatThread()
    );
    scheduler.schedule(
        heartbeatTask,
        renewalIntervalInSecs, TimeUnit.SECONDS);
    //省略....
}
```

`renewalIntervalInSecs=30s`，默认每隔30s执行一次。

HeartbeatThread

这个线程的实现很简单，调用 `renew()` 续约，如果续约成功，则更新最后一次心跳续约时间。

```
private class HeartbeatThread implements Runnable {

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

在 `renew()` 方法中，调用 EurekaServer 的 `"apps/" + appName + '/' + id;` 这个地址，进行心跳续约。

```
boolean renew() {
    EurekaHttpResponse<InstanceInfo> httpResponse;
    try {
        httpResponse =
eurekaTransport.registrationClient.sendHeartBeat(instanceInfo.getAppName(),
instanceInfo.getId(), instanceInfo, null);
        logger.debug(PREFIX + "{} - Heartbeat status: {}", appPathIdentifier,
httpResponse.getStatusCode());
        if (httpResponse.getStatusCode() == Status.NOT_FOUND.getStatusCode()) {
            REREREGISTER_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;
    }
}
```

服务端收到心跳处理

服务端具体为调用 [com.netflix.eureka.resources] 包下的 `InstanceResource` 类的 `renewLease` 方法进行续约，代码如下

```
@PUT
public Response renewLease(
    @HeaderParam(PeerEurekaNode.HEADER_REPLICATION) String isReplication,
    @QueryParam("overriddenstatus") String overriddenStatus,
    @QueryParam("status") String status,
    @QueryParam("lastDirtyTimestamp") String lastDirtyTimestamp) {
    boolean isFromReplicaNode = "true".equals(isReplication);
    // 调用 renew 进行续约
    boolean isSuccess = registry.renew(app.getName(), id, isFromReplicaNode);

    // Not found in the registry, immediately ask for a register
```

```

    if (!isSuccess) { //如果续约失败，返回异常
        logger.warn("Not Found (Renew): {} - {}", app.getName(), id);
        return Response.status(Status.NOT_FOUND).build();
    }
    // Check if we need to sync based on dirty time stamp, the client
    // instance might have changed some value
    Response response;
    //校验客户端与服务端的时间差异，如果存在问题则需要重新发起注册
    if (lastDirtyTimestamp != null && serverConfig.shouldSyncWhenTimestampDiffers()) {
        response = this.validateDirtyTimestamp(Long.valueOf(lastDirtyTimestamp),
        isFromReplicaNode);
        // Store the overridden status since the validation found out the node that
        replicates wins
        if (response.getStatus() == Response.Status.NOT_FOUND.getStatusCode()
            && (overriddenStatus != null)
            && !(InstanceStatus.UNKNOWN.name().equals(overriddenStatus))
            && isFromReplicaNode) {
            registry.storeOverriddenStatusIfRequired(app.getAppName(), id,
            InstanceStatus.valueOf(overriddenStatus));
        }
    } else {
        response = Response.ok().build(); // 续约成功，返回200
    }
    logger.debug("Found (Renew): {} - {}; reply status={}", app.getName(), id,
    response.getStatus());
    return response;
}

```

InstanceRegistry.renew

renew的实现方法如下，主要有两个流程

1. 从服务注册列表中找到匹配当前请求的实例
2. 发布EurekaInstanceRenewedEvent事件

```

@Override
public boolean renew(final String appName, final String serverId,
                     boolean isReplication) {
    log("renew " + appName + " serverId " + serverId + ", isReplication {}"
        + isReplication);
    //获取所有服务注册信息
    List<Application> applications = getSortedApplications();
    for (Application input : applications) { //逐一遍历
        if (input.getName().equals(appName)) { //如果当前续约的客户端和某个服务注册信息节点相同
            InstanceInfo instance = null;
            for (InstanceInfo info : input.getInstances()) { //遍历这个服务集群下的所有节点，找到某个匹配的实例instance返回。
                if (info.getId().equals(serverId)) {
                    instance = info; //
                    break;
                }
            }
            //发布EurekaInstanceRenewedEvent事件，这个事件在EurekaServer中并没有处理，我们可以监听这个事件来做一些事情，比如做监控。
            publishEvent(new EurekaInstanceRenewedEvent(this, appName, serverId,
                instance, isReplication));
            break;
        }
    }
}

```

```
        return super.renew(appName, serverId, isReplication);
    }
```

super.renew

```
public boolean renew(final String appName, final String id, final boolean
isReplication) {
    if (super.renew(appName, id, isReplication)) { //调用父类的续约方法, 如果续约成功
        replicateToPeers(Action.Heartbeat, appName, id, null, null, isReplication); //同步给集群中的所有节点
        return true;
    }
    return false;
}
```

AbstractInstanceRegistry.renew

在这个方法中，会拿到应用对应的实例列表，然后调用Lease.renew()去进行心跳续约。

```
public boolean renew(String appName, String id, boolean isReplication) {
    RENEW.increment(isReplication);
    Map<String, Lease<InstanceInfo>> gMap = registry.get(appName); //根据服务名字获取实例信息
    Lease<InstanceInfo> leaseToRenew = null;
    if (gMap != null) {
        leaseToRenew = gMap.get(id); //获取需要续约的服务实例,
    }
    if (leaseToRenew == null) { //如果为空, 说明这个服务实例不存在, 直接返回续约失败
        RENEW_NOT_FOUND.increment(isReplication);
        logger.warn("DS: Registry: lease doesn't exist, registering resource: {} - {}",
                    appName, id);
        return false;
    } else { //表示实例存在
        InstanceInfo instanceInfo = leaseToRenew.getHolder(); //获取实例的基本信息
        if (instanceInfo != null) { //实例基本信息不为空
            // touchASGCache(instanceInfo.getASGName());
            // 获取实例的运行状态
            InstanceStatus overriddenInstanceStatus =
                this.getOverriddenInstanceStatus(
                    instanceInfo, leaseToRenew, isReplication);
            if (overriddenInstanceStatus == InstanceStatus.UNKNOWN) { //如果运行状态未知, 也返回续约失败
                logger.info("Instance status UNKNOWN possibly due to deleted override
for instance {}", " + "; re-register required", instanceInfo.getId());
                RENEW_NOT_FOUND.increment(isReplication);
                return false;
            }
            //如果当前请求的实例信息
            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(),
                    overriddenInstanceStatus.name(),
                    instanceInfo.getId());
                instanceInfo.setStatusWithoutDirty(overriddenInstanceStatus);
            }
        }
    }
}
```

```
        }
    }
    //更新上一分钟的续约数量
    renewLastMin.increment();
    leaseToRenew.renew(); //续约
    return true;
}
}
```

续约的实现，就是更新服务端最后一次收到心跳请求的时间。

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