微服务框架核心源码深度解析专题

面试题暨重要知识点总结

第 01 次直播课-11

【Q-01】你曾阅读过 Spring Cloud 的源码吗？我们知道，Spring Cloud 是通过 Spring Boot 集成了很多第三方框架构成的。现在准备解析 Spring Cloud 中某子框架的源码，若还没有找到合适的入手位置，那么从哪里开始解析可能是一个不错的选择。请谈一下你的认识。

【RA】我自己曾阅读过 Spring Cloud 中的 Eureka、OpenFeign、Ribbon 等的源码。对于一个未曾阅读过的子框架源码，我认为从自动配置类开始解析可能是一个不错的选择。

我们知道 Spring Cloud 是通过 Spring Boot 将其它第三方框架集成进来的。Spring Boot 最大的特点就是自动配置，我们可以通过导入相关 Starter 来实现需求功能的自动配置、相关核心业务类实例的创建等。也就是说，核心业务类都是集中在自动配置类中的。所以从这里下手分析应该是个不错的选择。

那么从哪里可以找到这个自动配置类呢？从导入的 starter 依赖工程的 META-INF 目录中的 spring.factory 文件中可以找到。该文件的内容为 key-value 对，查找 EnableAutoConfiguration 的全限定性类名作为 key 的 value，这个 value 就是我们要找到的自动配置类。

【Q-02】@EnableConfigurationProperties 注解对于 Starter 的定义很重要，请谈一上你对这个注解的认识。

【RA】@EnableConfigurationProperties 注解在 Starter 定义时主要用于读取 application.yml 配置文件中相关的属性，并封装到指定类型的实例中，以备 Starter 中的核心业务实例使用。

具体来说，它就是开启了对@ConfigurationProperties 注解的 Bean 的自动注册，注解到 Spring 容器中。这种 Bean 有两种注册方式：在配置类使用@Bean 方法注册，或直接使用该注解的 value 属性进行注册。若在配置类中使用@Bean 注册，则需要在配置类中定义一个@Bean 方法，该方法的返回值为“使用@ConfigurationProperties 注解标注”的类。若直接使用该注解的 value 属性进行注册，则需要将这个“使用@ConfigurationProperties 注解标注”的类作为 value 属性值出现即可。

【Q-03】Spring Boot 中定义了很多条件注解，这些注解一般用于对配置类的控制。在这些条件注解中有一个@ConditionalOnMissingBean 注解，你了解过嘛？请谈一下你对它的认识。

【RA】@ConditionalOnMissingBean 注解是 Spring Boot 提供的众多条件注册中的一个。其表示的意义是，当容器中没有指定名称或指定类型的 Bean 时，该条件为 true。不过，这里需要强调一点的是，这里要查找的“容器”是可以指定的。通过 search 属性指定。其 search 的范围有三种：仅搜索当前配置类容器；搜索所有层次的父类容器，但不包含当前配置类容器；搜索当前配置类容器及其所有层次的父类容器，这个是默认搜索范围。

【Q-04】Spring Cloud 中默认情况下对于 Eureka Client 实例的创建中，@RefreshScope 注解是比较重要的，请谈一下你对这个注解的认识。

【RA】@RefreshScope 注解是 Spring Cloud 中定义的一个注解。该注解用于配置类，可以添

加在配置类上，也可以添加在@Bean 方法上。其表示的意思是，该@Bean 方法会以多例的形式生成会自动刷新的 Bean 实例。这种方式就等价于在 Spring 的 xml 配置文件中指定<bean/>标签的 scope 属性值为 refresh。当然，若一个配置类上添加了该注解，则表示该配置类中的所有@Bean 方法创建的实例都是@RefreshScope 的。

【Q-05】Spring Cloud 中默认情况下对于 Eureka Client 实例的创建是在 EurekaClient 的自动配置类中通过@Bean 方法完成的。但在源码中，这个@Bean 方法上同时出现了@RefreshScope、@ConditionalOnMissionBean，与@Lazy 注解，从这些注解的意义来分析，是否存在矛盾呢？它们联合使用又是什么意思呢？请谈一下你的看法。

【RA】首先来说，这三个注解的意义都是比较复杂的。

@RefreshScope 注解是 Spring Cloud 中定义的一个注解。其表示的意思是，该@Bean 方法会以多例的形式生成会自动刷新的 Bean 实例。

@ConditionalOnMissionBean 注解表示的意思是，只有当容器中没有@Bean 要创建的实例时才会创建新的实例，即这里创建的@Bean 实例是单例的。

@Lazy 注解表示延迟实例化。即在当前配置类被实例化时并不会调用这里的@Bean 方法去创建实例，而是在代码执行过程中，真正需要这个@Bean 方法的实例时才会创建。

这三个注解的联用不存在矛盾，其要表达的意思是，这个@Bean 会以延迟实例化的形式创建一个单例的对象，而该对象具有自动刷新功能。

【Q-06】Spring Cloud 中大量地使用了条件注解，其中@ConditionalOnRefreshScope 注解对于 Eureka Client 的创建非常重要。请谈一下你对这个注解的认识。

【RA】首先，关于条件注解，实际是 Spring Boot 中出现的内容，其一般应用于配置类中。表示只有当该条件满足时才会创建该实例。而您提到的@ConditionalOnRefreshScope 注解，其实际是 Eureka Client 的自动配置类中的一个内部注解。该注解不同于 Spring Boot 中的一般性注解的是，其是一个复合条件注解，其复合的条件有三个：

- 在当前类路径下具有 RefreshScope 类
- 在容器中要具有 RefreshAutoConfiguration 类的实例
- 指定的 eureka.client.refresh.enable 属性值为 true。不过，其缺省值就是 true。这也就是为什么我们的配置文件默认支持自动更新的原因。

只有当这个复合注解中的三个条件均成立时，@ConditionalOnRefreshScope 注解才满足条件。此时才有可能会调用创建 Eureka Client 的@Bean 方法。所以，该注解对于 Eureka Client 的创建非常重要。

【Q-07】你刚才已经谈过了对@ConditionalOnRefreshScope 注解的认识，非常不错。不过，与这个注解相对应的另一个注解@ConditionalOnMissingRefreshScope，你是否了解？若关注过，谈一下你的认识。（不将面试者问死誓不罢休）

【RA】@ConditionalOnMissingRefreshScope 我也曾了解过（看来这个面试者很厉害）。这个注解就像@ConditionalOnRefreshScope 注解一样，也是一个复合条件注解，其也包含了三个条件。不同的是，这个注解中的条件是或的关系，只要满足其中一条这个注解就匹配上了。而@ConditionalOnRefreshScope 注解中的三个条件是与的关系，必须所有条件均满足其才能匹配上。

这个或的关系是通过让一个复合条件类继承自一个能够表示或关系的复合条件父类 AnyNestedCondition 实现的。这样的话，这个复合条件类中定义的多个内部条件类中，只要有一个匹配上，那么这个复合条件类就算匹配上了。

【Q-08】Spring Cloud 中 Eureka Client 的源码中有一个非常重要的类 Applications，其被称为客户端注册表。请谈一下你对它的认识。

【RA】Applications 类实例中封装了来自于 Eureka Server 的所有注册信息，通常称其为“客户端注册表”。只所以要强调“客户端”是因为，服务端的注册表不是这样表示的，是一个 Map。

该类中封装着一个非常重要的 Map 集合，key 为微服务名称，而 Value 则为 Application 实例。Application 类中封装了一个 Set 集合，集合元素为“可以提供该微服务的所有主机的 InstanceInfo”。也就是说，Applications 中封装着所有微服务的所有提供者信息。

【Q-09】Eureka 源码中 InstanceInfo 类中具有两个最终修改时间戳，这两个时间戳对于 Eureka 的 Server 端与 Client 端源码的理解都比较重要。这两个时间戳你了解过吗？若了解过，请谈一下你对它们的认识。

【RA】InstanceInfo 实例中封装着一个 Eureka Client 的所有信息，其就可以代表了一个 Eureka Client。其封装的两个最终修改时间戳分别为 lastDirtyTimestamp 与 lastUpdatedTimestamp。这两个时间戳的区别是：

- lastDirtyTimestamp：记录 intance 在 Client 被修改的时间。该修改会被传递到 Server 端。
- lastUpdatedTimestamp：记录 intance 状态在 Server 端被修改的时间。

【Q-10】Eureka 源码中 InstanceInfo 类中具有两个状态属性，是哪两个，你了解过它们吗？

【RA】Eureka 源码中 InstanceInfo 类具有两个状态属性，分别是 status 与 overriddenStatus。下面我依次谈一下我对它们的了解。

status 就是当前 Client 的工作状态。只有在 Server 端注册表中该 Client 的状态为 UP 时，其才会被其它服务发现，才可对外提供提供服务。

overriddenStatus 是在 Client 提交注册请求与 renew 续约请求时用于计算当前 Client 在 Server 端的 Status 状态的。

【Q-11】Spring Cloud 中 Eureka Client 与 Eureka Server 的通信，及 Eureka Server 间的通信是如何实现的？请简单介绍一下。

【RA】Spring Cloud 中 Eureka Client 与 Eureka Server 的通信，及 Eureka Server 间的通信，均采用的是 Jersey 框架。

Jersey 框架是一个开源的 RESTful 框架，实现了 JAX-RS 规范。该框架的作用与 SpringMVC 是相同的，其也是用户提交 URI 后，在处理器中进行路由匹配，路由到指定的后台业务。这个路由功能同样也是通过处理器完成的，只不过这里的处理器不叫 Controller，而叫 Resource。

第 02 次直播课-17

【Q-01】Spring Cloud 中 Eureka Client 在启动时需要从 Eureka Server 中下载注册表到本地进行缓存，以备进行负载均衡调用。请谈一下你对这个启动时下载注册表过程的认识。

【RA】Spring Cloud 中 Eureka Client 在启动时需要从 Eureka Server 中下载注册表到本地进行

缓存。这次下载属于全量下载，即要将 Server 端所有注册信息 Applicaitons 全部下载到本地并缓存。当然，若指定可以从远程 Region 获取，其也会通过其所连接的这个 Server，将远程 Region 中的注册信息也全部获取到。这个过程称为获取客户端注册表。

获取“客户端注册表”最终执行的操作是，通过 Jersey 框架提交了一个 GET 请求，然后获取到的 Applications 实例结果。

若获取失败，其会从本地备用注册表中获取并缓存。

【Q-02】Spring Cloud 中 Eureka Client 需要定时从 Eureka Server 中获取注册表信息，这个过程称为服务发现。请谈一下你对这个获取过程的认识。

【RA】Eureka Client 从 Eureka Server 中获取注册表分为两种情况，一种是将 Server 中所有注册信息全部下载到当前客户端本地并进行缓存，这种称为全量获取；一种是仅获取在 Server 中发生变更的注册信息到本地，然后根据变更修改本地缓存中的注册信息，这种称为增量获取。当 Client 在启动时第一次下载就属于全量获取，而后期每 30 秒从 Server 下载一次的定时下载属于增量下载。无论是哪种情况，Client 都是通过 Jersey 框架向 Server 发送了一个 GET 请求。只不过是，不同的获取方式，提交请求时携带的参数是不同的。

【Q-03】Spring Cloud 中 Eureka Client 需要注册到 Eureka Server 中，请谈一下你对这个注册过程的认识。

【RA】Eureka Client 向 Eureka Server 提交的注册请求，实际是通过 Jersey 框架完成的一次 POST 提交，将当前 Client 的封装对象 intanceInfo 提交到 Server 端，写入到 Server 端的注册表中。

但这个注册请求在默认情况下并不是在 Client 启动时直接提交的，而是在 Client 向 Server 发送续约信息时，由于其未在 Server 中注册，所以 Server 会中其返回 404，在这种情况下，而引发的 Client 注册。当然，若 Client 的续约信息发生了变更 Client 也会提交注册请求。

【Q-04】Spring Cloud 中 Eureka Client 需要定时从 Eureka Server 中更新注册信息。对于这个定时器及其执行过程，请谈一下你的看法。

【RA】Spring Cloud 中 Eureka Client 需要定时从 Eureka Server 中更新注册信息。但这里是使用了 one-shot action 的一次性定时器实现的 repeated 定时执行。这个 repeated 过程是通过其一次性的定时任务实现的：当这个一次性定时任务执行完毕后，会调用启动下一次的定时任务。

【Q-05】Spring Cloud 中 Eureka Client 需要定时从 Eureka Server 中更新注册信息。这个定时任务是通过一个 one-shot action 的定时器完成的。其为什么不直接使用一个 repeated 的定时器呢？请谈一下你的看法。

【RA】Spring Cloud 中 Eureka Client 需要定时从 Eureka Server 中更新注册信息。这个定时任务是通过一个 one-shot action 的定时器完成的。只所以选择 one-shot action 的定时器来完成一个 repeated 的事情，其主要目的就是方便控制 delay，保证任务顺利完成。

定时器要执行的任务是通过网络从 Server 下载注册信息。而我们知道，网络传输存在不稳定性，不同的传输数据可能走的网络链路是不同的，而不同的链路的传输时间可能也是不同的。本次传输超时，下次重试可能走的链路不同就不超时了。

repeated 定时器的执行原理是，本次任务的开始，必须在上一次的任务完成后，不存在超时。即只要没完成就不会通过执行下次任务进行重试。

使用 one-shot action 定时器完成 repeated 定时任务时，若本次定时任务出现了超时，则可以在下次任务执行之前增大定时器的 delay。当然，若下载速率都很快，也可将已经增

大的 delay 再进行减小。方便控制 delay，保证任务的顺利完成。

【Q-06】Futue 是 JDK5 中提供的一个接口。其方法 cancel()是一个经常被用到的方法，请谈一下你对这个方法的认识。

【RA】Futue 是 JDK5 中提供的一个 JUC 的接口，其用于执行一些异步任务。对于其执行的异步任务，可以通过 cancel()方法尝试着进行取消。其用法为：

- 若任务已经完成或已经取消，则本次取消失败。
- 若任务在启动之前就调用了 cancel()，则这个任务将不会再执行。
- 若任务已经启动，此时再执行 cancel()，那么这个执行的任务是否立即被中断，取决于 cancel()方法的参数。若参数为 true，则会立即中断任务的执行；若参数为 false，则会让正在执行中的任务执行完毕，然后再中断。
- 若 cancel()方法执行完毕后，顺序执行 isDone()或 isCancelled()方法，这些方法均返回 true。

【Q-07】Futue 是 JDK5 中提供的一个接口。其方法 get()是一个经常被用到的方法，请谈一下你对这个方法的认识。

【RA】Futue 是 JDK5 中提供的一个 JUC 的接口，其用于执行一些异步任务，并通过 get()方法可以获得该异步任务的结果。get()方法是一个阻塞的方法，可以为其指定阻塞的最长时长。

- 若在指定时长内异步操作完成，则阻塞会被立即唤醒；
- 若在指定时长内该异步操作被 cancel()，则阻塞也会被立即唤醒，并抛出一个 CancellationException；
- 若在指定时长内未发生任务事情，则阻塞也会被唤醒，并抛出一个 TimeoutException；
- 若 get()方法没有指定阻塞时长，则其会一直阻塞下去，直到任务完成或取消。

【Q-08】Spring Cloud 中 Eureka Client 向 Eureka Server 发送增量获取注册表请求，Server 会返回给 Client 一个 delta，关于这个返回值 delta，请谈一下你的看法。

【RA】Spring Cloud 中 Eureka Client 向 Eureka Server 发送增量获取注册表请求，Server 会返回给 Client 一个 delta，这个 delta 是一个 Applications 类型的变量。

- Server 返回的这个 delta 值若为 null，则表示 Server 端基于安全考虑，禁止增量下载，其会自动进行全量下载。
- Server 返回的这个 delta 值不为 null，但其包含的 application 数量为 0，则表示没有更新内容。若数量大于 0，则表示有更新内容。有更新内容，则需要将更新添加到本地缓存的注册表 applications 中。

【Q-09】Spring Cloud 中 Eureka Client 向 Eureka Server 发送增量获取注册表请求，Server 端是如何知道哪些 intance 对这个 Client 来说是变更过的？请谈一下你的看法。

【RA】当 Eureka Server 接收到 Eureka Client 发送的增量获取注册表请求后，其并不知道哪些 instance 对于这个 Client 来说是更新过的。但是在 Server 中维护着一个“最近变更队列”，无论对于哪个 Client 的增量请求，Server 端都是将该队列中的 instance 变更信息发送给了 Client。当然，Server 中有一个定时任务，当这个 instance 的变更信息不再属于“最近”时，会将该 instance 变更信息从队列中清除。

当 Client 接收到 Server 发送的增量变更信息后，Client 端有一种判断机制，可以判断出其接收到这个增量信息对它自己来说是否出现了更新丢失。即出现了队列中清除掉的变更信息，并没有更新到当前 Client 本地。若发生更新丢失，Client 会再发起全量获取，以保证 Client 获取到的注册表是最完整的注册表。

【Q-10】Spring Cloud 中 Eureka Client 从 Eureka Server 中增量获取到的注册信息，都是在 Server 端发生了变更的 instanceInfo 信息。这些变更信息中包含变更的类型。其中对于 ADDED 与 MODIFIED 这两种类型的变更，我们发现处理方式是相同的，为什么？请谈一下你的看法。

【RA】Eureka Client 接收到的来自 Server 的增量注册信息后，对于添加变更与修改变更的处理方式的确是相同的，都是采用了添加变更的处理方式。其实，无论是添加的 InstanceInfo 还是修改的 InstanceInfo，Client 首先都是根据该 InstanceInfo 的 id，从本地 InstanceInfo 的 Set 集合中将其删除，然后再将新来的变更的 InstanceInfo 加入 Set 集合。只不过，对于添加变更，其在原来的 Set 集合中找不到其要删除的 InstanceInfo 而已。

只所以能够被根据 InstanceId 在 Set 集合中进行删除，是因为 Set 集合的元素 InstanceInfo 重写了 equals()方法——根据 instanceId 进行元素相等判断。

【Q-11】Spring Cloud 中 Eureka Client 从 Eureka Server 中获取注册信息，有全量获取与增量获取之分。为什么全量获取到 Client 本地后没有做 Region 的分离，而增量下载后却将这些发生变更的信息分为了本地 Region 与远程 Region？

【RA】Spring Cloud 中 Eureka Client 从 Eureka Server 中获取注册信息，有全量获取与增量获取之分。

全量获取一般是在应用启动时第一次获取注册表数据时发起的。这次获取的目的就是为了让应用能够马上进行服务。所以 Server 把返回的注册信息并没有区分 Region，但这些注册信息包含了所有本地 Region 与远程 Region 中的注册信息。

增量获取一般是从应用第二次获取注册表数据时发起的。这次获取的目的不仅是为了能够获取到距离自己最近的微服务信息，而且是为了保证服务的可用性（Eureka 是 AP 的）。若当前 Region 中无需要的服务时，可以从远程 Region 获取。所以 Server 返回的注册信息对 Rgion 进行了区分，而 Client 端也对本地 Region 与远程 Region 中的注册信息进行了分别缓存。

【Q-12】Spring Cloud 中 Eureka Client 会向 Eureka Server 进行定时续约，请谈一下你对这个定时续约的认识。

【RA】Eureka Client 会向 Eureka Server 进行定时续约，即会进行定时心跳。其最终就是通过 Jersey 框架向 Eureka Server 提交一个 PUT 请求。该 PUT 请求没有携带任何请求体，而 Eureka Server 仅仅就是接收到一个 PUT 请求。但通过请求，Server 能够知道这个请求的发送者 instanceId。而 Eureka Server 就是通过这个发送者 instanceId 从注册表中查找其 instance 信息的。当然，Server 也给 Client 发送来了响应信息：若从 Server 的注册表中找到了该续约的 instance，则返回该 instanceInfo 实例；若没有找到，则返回 404。

【Q-13】Spring Cloud 中 Eureka Client 会定时检测 Client 的 instanceInfo 是否发生了变更。请谈一下你对这个定时任务的认识。

【RA】Eureka Client 会定时检测 Client 的 instanceInfo 是否发生了变更，其主要是检测了两样内容：一个是检测数据中心中的关于当前 instanceInfo 的信息是否变更，一个是检测配置文件中当前 instanceInfo 中的续约信息是否变更。只要发生了变更，则将变化后的信息发送给 Server，这个发送执行的是 register()注册。

【Q-14】Spring Cloud 中 Eureka Client 向 Eureka Server 提交 registrer()注册请求的时机较多，请简单总结一下。

【RA】Eureka Client 向 Eureka Server 提交的 register()注册请求的情况有三种：

- 在 Client 实例初始化时直接提交 register()注册请求
- 定时发送心跳时，服务端返回 404，此时 Client 会发出 registrer()注册请求
- 定时更新 Client 续约信息给 Server 时，只要 Client 续约信息发生变更，其提交的就是 register()注册请求

【Q-15】Spring Cloud 中 Eureka Client 在做定时更新续约信息给 Server 时有一个定时任务，定时查看本地配置文件中的 instanceInfo 是否发生了变更。这个定时任务在 Eureka Client 启动时通过 start()启动了定时任务，该定时器是一个 one-shot action 定时器，其会调用 InstanceInfoReplicator 的 run()方法。而该方法会再次启动一个 one-shot action 的定时任务，实现了 repeated 定时执行。然而，当 instanceInfo 的状态发生变更后会调用一个按需更新方法 onDemandUpdate()，该方法同样会调用 InstanceInfoReplicator 的 run()方法，再次启动一个 one-shot action 的定时任务，实现了 repeated 定时执行。这样的话，只要发生一次状态变更，就会启动一个 repeated 的定时任务持续执行下去。那么若 InstanceInfo 的状态多次发生变更，是否就会启动很多的一直持续执行的该定时任务了？请谈一下你的看法。
【RA】答案当然是否定的。关键就在于两点：一个是，前面题目中无论是 start()方法还是 InstanceInfoReplicator 的 run()方法，在启动了定时任务后，都会将定时任务实例 future 写入到一个原子引用类型的缓存中，且后放入的会将先放入的覆盖，即这个缓存中存放的始终为最后一个定时任务。第二个关键点是这个 onDemandUpdate()方法。其在调用 InstanceInfoReplicator 的 run()方法之前首先将这个缓存中的异步操作 cancel()，即将最后一个定时任务结束，然后才会再启动一个新的定时任务。所以，只会同时存在一个该定时任务。

【Q-16】Spring Cloud 中 Eureka Client 的续约配置信息默认情况下是允许动态变更的。为了限制变更的频率，Eureka Client 使用了一种限流策略，是什么策略？请谈一下你对这种策略的认识。
【RA】Spring Cloud 中 Eureka Client 的续约配置信息默认情况下是允许动态变更的。为了限制变更的频率，Eureka Client 使用了令牌桶算法。

该算法实现中维护着一个队列，首先所有元素需要进入到队列中，当队列满时，未进入到队列的元素将丢弃。进入到队列中的元素是否可以被处理，需要看其是否能够从令牌桶中拿到令牌。一个元素从令牌桶中拿到一个令牌，那么令牌桶中的令牌数量就会减一。若元素生成速率较快，则其从令牌桶中获取到令牌的速率就会较大。一旦令牌桶中没有了令牌，则队列很快就会变满，那么再来的元素将被丢弃。

【Q-17】Spring Cloud 中 Eureka Client 的哪些操作会引发客户端 InstanceInfo 的最新修改时间戳 lastDirtyTimestamp 的变化？请谈一下你的认识。
【RA】Spring Cloud 中 Eureka Client 的操作中有两处操作可以引发客户端 InstanceInfo 的最新修改时间戳 lastDirtyTimestamp 的变化。

- 在进行第一次心跳发送时，由于 Server 中没有发现该 InstanceInfo 而向其返回了 404。此时的 Client 会修改 lastDirtyTimestamp 的值。
- 在续约信息发生更新时修改 lastDirtyTimestamp 的值。

第 03 次直播课-9

【Q-01】Spring Cloud 中 Eureka Client 通过 Actuator 提交的 POST 请求的 shutdown 进行服务下架时，其调用的下架处理方法我们从哪里找到它？请谈一下你的思路。
【RA】Spring Cloud 中 Eureka Client 通过 Actuator 提交的 POST 请求的 shutdown 进行服务下架，就是要销毁 Eureka Client 实例。而该 Eureka Client 实例是在 EurekaClient 的自动配置类中通过@Bean 方法创建的。所以，这个下架处理方法应该是@Bean 注解的 destroyMethod 属性指定的方法。

【Q-02】Spring Cloud 中 Eureka Client 通过 Actuator 提交的 POST 请求的 shutdown 进行服务下架时，其 Client 内部都做了些什么重要工作？请谈一下你对这个服务下架的认识。
【RA】Eureka Client 通过 Actuator 提交的 POST 请求的 shutdown 进行服务下架时，其内部主要完成了四样工作：

- 将状态变更监听器删除
- 停止了一堆定时任务的执行，并关闭了定时器
- 通过 Jersey 框架向 Eureka Server 提交了一个 DELETE 请求
- 关闭了一些其它相关工作

【Q-03】对于 Spring Cloud 中的 Eureka，用户通过平滑上下线方式进行 Client 状态的修改。这个状态修改请求是被客户端的哪个类处理的，这个类实例是在何时创建的？请谈一下你的认识。
【RA】对于 Spring Cloud 中的 Eureka，用户通过平滑上下线方式进行 Client 状态的修改。这个状态修改请求是被客户端的 ServiceRegistryEndpoint 类实例的 setStatus()方法处理的。这个类实例是在 Spring Cloud 应用启动时被加载创建的。具体来说，spring-cloud-starter 依赖于 spring-cloud-common，而该 common 依赖加载并实例化了 ServiceRegistryAutoConfigration 配置类。在该配置类中实例化了 ServiceRegistryEndpoint 类。

【Q-04】对于 Spring Cloud 中的 Eureka，用户通过 Actuator 的 service-registry 监控终端提交状态修改请求，请谈一下你对这个请求处理过程的认识。
【RA】对于 Spring Cloud 中的 Eureka，用户通过 Actuator 的 service-registry 监控终端提交状态修改请求，服务平滑上下线就属于这种情况，但这种情况不仅限于服务平滑上下线。
当 Client 提交的状态为 UP 或 OUT_OF_SERVICE 时，属于平滑上下线场景。该请求会被 Client 接收并直接再以 PUT 请求的方式提交给 Server，在 Client 端并未修改 Client 的任何状态。Server 在接收到这个请求后，会从注册表中找到该 Client 对应的 InstanceInfo，修改其 overriddenStatus、status 为指定状态。

当 Client 提交的状态为 CANCEL_OVERRIDE 时，是要将 Server 端当前 Client 对应 InstanceInfo 的 overriddenStatus 从一个缓存 map 中删除，并将其 overriddenStatus 与 status 修改为 UNKNOWN 状态。这个缓存 map 中记录着注册到当前 Server 中的每一个 instanceInfo 对应的 overriddenStatus，这个 map 中的状态值对于其它 Client 发现一个 InstanceInfo 的对外表现状态 status 非常重要。当服务端的某 InstanceInfo 的 overriddenStatus 变为 UNKNOWN 时，该 Client 发送的心跳 Server 是不接收的。Server 会向该 Client 返回 404。

【Q-05】对于 Spring Cloud 中的 Eureka，用户通过 Actuator 的 service-registry 监控终端提交

状态修改请求，如果用户提交的状态是一个非法状态会怎么样？请谈一下你的认识。

【RA】对于 Spring Cloud 中的 Eureka，用户通过 Actuator 的 service-registry 监控终端提交状态修改请求，这个状态值一般为五个标准状态 UP、DOWN、OUT_OF_SERVICE、STARTING、UNKNOWN 中的 UP 或 OUT_OF_SERVICE，也可以是 CANCEL_OVERRIDE。但若用户提交的不是这六种情况之一，系统会将其最终归结为 UNKNOWN 状态。

【Q-06】Spring Cloud 中 EurekaServerAutoConfiguration 自动配置类被实例化的一个前提条件是，容器中要有一个 Marker 实例，这个 Marker 实例是在哪被添加到了容器？

【RA】Spring Cloud 中关于 Eureka Server 的这个 Marker 实例，就是 Eureka Server 的一个标识，一个开关。该实例被添加到了容器，Eureka Server 就开启了。其是在 Eureke Server 启动类上的@EnableEurekaServer 注解中被实例化并添加到的容器。所以，若没有添加该注解，Eureka Server 启动类的启动是不会创建启动 Eureka Server 的。

【Q-07】Spring Cloud 中 Eureka Client 会注册到 Eureka Server 的注册表中，在 Server 中这个注册表是以怎样的形式存在的？当 Eureka Client 将 Eureke Server 中的注册信息下载到本地后，这个注册表又是以怎样的形式存在的？请谈一下你的认识。

【RA】Spring Cloud 的 Eureka Server 中的这个注册表是一个双层 Map，外层 map 的 key 为微服务名称，value 为内层 map。内层 map 的 key 为 instanceId，value 为 Lease 对象。Lease 对象中包含一个执有者 Holder 属性，表示该 Lease 对象所属的 InstanceInfo。

当 Eureka Client 将 Server 端的注册表下载到了本地，该注册表是以 Applications 形式出现的。Applications 中维护着一上 Map 集合，key 为微服务名称，value 为 Application 实例。该 Application 实例中包含了所有提供该微服务名称的 InstanceInfo 信息。因为 Application 中也维护着一个 Map，key 为 instanceId，value 为 InstanceInfo。

【Q-08】Eureka Client 提交的状态修改请求，Eureka Server 是如何处理的，Server 端都做了哪些变更？请谈一下你的认识。

【RA】Eureka Client 提交的状态修改请求，Eureka Server 在接收到后，首先根据该 Client 的微服务名称及 instanceId 在 Server 端注册表中进行了查找。若没有找到，则直接返回 404；若找到了，其会执行两大任务：

- 任务一：将客户端修改请求中的新状态写入到注册表中。
- 任务二：将写入到当前 server 注册表中新的状态同步到其它 server。

而在第一项任务中完成的重要工作有如下几项：

- 使用新状态替换缓存 overriddenInstanceStatusMap 中的老状态
- 修改 instanceInfo 的 overriddenStatus 状态为新状态
- 修改 instanceInfo 的 status 状态为新状态
- 更新来自于请求参数的客户端修改时间戳 lastDirtyTimestamp
- 将本次修改形为记录到 instanceInfo 的行为类型中
- 修改服务端修改时间戳 lastUpdatedTimestamp
- 将本次修改记录到最近更新队列 recentlyChanngeQueue 中

【Q-09】Eureka Server 中的注册表发生了变更，其是怎样将变更同步到其它 Server 的？请谈一下你的看法。

【RA】Eureka Server 中的注册表发生了变更，并不是一定要同步给其它 Server 的，需要分

情况处理。

若这个变更是由 Client 端直接引发，则当前 Server 会遍历所有其它 Server，通过 Jersey 框架向每一个其它 Server 发送变更请求。这样就实现了同步。

若这个变更是由其它 Server 端发送的变更，则其仅仅会在本地变更一下即可，不会再向其它 Server 发送变更请求，以防止出现无限递归。

第 04 次直播课-7

【Q-01】Eureka Client 提交的 CANCEL_OVERRIDE 状态修改请求，Eureka Server 是如何处理的，Server 端都做了哪些变更？请谈一下你的认识。

【RA】Eureka Client 提交的 CANCEL_OVERRIDE 状态修改请求，Eureka Server 在接收到后，首先根据该 Client 的微服务名称及 instanceId 在 Server 端注册表中进行了查找。若没有找到，则直接返回 404；若找到了，其会执行两大任务：

- 任务一：将 Server 端删除 overridden 状态。
- 任务二：将此变更同步到其它 server。

删除 overridden 状态，从哪里删除呢？这个删除处理，在 Server 端主要完成了三样工作：

- 将该 instanceInfo 在缓存集合 overriddenInstanceStatusMap 中的 Entry 删除。
- 将 instanceInfo 中的 overridden 状态变更为 UNKNOWN。
- 将 instanceInfo 中的 status 状态变更为 UNKNOWN。

overriddenInstanceStatusMap 中的缓存的 instanceInfo 的 overridden 状态，对于后续注册、续约等请求处理时 instanceInfo 的状态判断起很大作用。另外此时，该 Client 发送的心跳 Server 是不接收的。Server 会向该 Client 返回 404。

【Q-02】Eureka Client 提交的续约请求，Eureka Server 是如何处理的？请谈一下你的认识。

【RA】客户端续约请求，即客户端向服务端发送心跳。客户端的心跳提交的是一个没有携带任何请求体的 PUT 请求，不过其在请求 URI 中携带了心跳的发出者的 InstanceId，及当前心跳发出者的状态。Server 端对于续约请求，主要完成了两若项任务：

- 当 Server 接收到 Client 发送的续约信息后，根据注册表中当前 InstanceInfo 的状态信息计算出其状态，然后更新为新的 status。
- 将心跳同步到其它 server。
- 在 Server 间的同步过程可能会导致 Server 间 overridden 状态的不一致。所以又进行了该状态的统一。

【Q-03】Eureka Server 间进行续约同步过程中可能会导致 overridden 状态的不一致，为什么？请谈一下你的看法。

【RA】首先要清楚一点，Eureka 是 AP 的，是允许出现 Server 间数据不一致的。例如，当前 Eureka 中由于客户端下架请求而从注册表中删除了某 Client，在进行 Server 间同步时，由于另一个 Server 处于自我保护模式，所以其是不能删除 Client 的。此时就出现了 Server 间数据的不一致。

下面再来说续约。无论是直接处理Client的续约请求，还是处理Server间续约同步，Server端对于续约的处理，根本不涉及 lastDirtyTimestamp 时间戳，及 overridden 状态。这一点从续约的源码中是可以看出来的。那么有可能会出现以下场景：Client 通过 Actuator 修改了状态，而这个状态修改操作在 Server 间同步时并没有同步成功，出现了 Server 间对于同一个 InstanceInfo 中 overridden 状态的不一致。

虽然 Eureka 本身是 AP 的，但其仍是尽量想让 Eureka 间实现同步，所以在其发生最频繁的续约中解决了这个问题。只不过，由于续约本身根本不涉及 overridden 状态，仅靠续约是解决不了的。所以需要在 Eureka Server 的配置文件中添加专门的配置解决这个问题。不过，这个属性设置默认是开启的。

【Q-04】Eureka Server 在进行续约处理时，若发现其计算的当前 isntanceInfo 的 status 状态为 UNKNOWN，说明什么？请谈一下你的看法。

【RA】Eureka Server 在进行续约处理时，若发现其计算的当前 instanceInfo 的 status 状态为 UNKNOWN，说明这个计算结果是在 OverrideExistsRule 规则中计算出的结果，即当前 instanceInfo 的 overridden 状态为 UNKNOWN。对于一个 InstanceInfo 来说，可以从缓存 map 中读取到其 overridden 状态为 UNKNOWN，只能有一种情况：这个 UNKNOWN 的 overridden 状态是通过 Actuator 的 CANCEL_OVERRIDE 修改的状态，即用户取消了该 InstanceInfo 的 overridden 状态。

那么也就是说，Eureka Server 在进行续约处理时，若发现其计算的当前 instanceInfo 的 status 状态为 UNKNOWN，则说明该 InstanceInfo 已经不对外提供服务了。

【Q-05】Eureka Client 的注册请求被 Server 接收到后，会首先判断该 Client 在注册表中是否存在。按理说，这里是注册处理，注册表中应该是不存在该 Client 的 lease 信息的，但的确会出现新注册的 Client 在注册表中存在的情况，这是为什么？请谈一下你的看法。

【RA】Eureka Client 的注册请求被 Server 接收到后，注册表中可能会出现新注册的 Client 在注册表中已经存在的情况。这是因为，当 Client 的 instanceInfo 的续约信息发生了变更，Client 的“定时更新 InstanceInfo 信息给 Server”的定时任务发出的是 register()请求，但该客户端其实在之前已经注册过了。此时就会出现注册表中已经存在该 instanceId 的情况。

【Q-06】Eureka Server 在处理新注册的 Client 在注册表中已经存在的情况时会出现一种比较奇怪的情况：当前新注册的 InstanceInfo 中的 lastDirtyTimestamp，比注册表中缓存的当前这个 InstanceInfo 中的 lastDirtyTimestamp 小。即后注册的反而过时，这是为什么？请谈一下你的看法。

【RA】Eureka Server 在处理新注册的 Client 在注册表中已经存在的情况时会出现，当前新注册的 InstanceInfo 中的 lastDirtyTimestamp，比注册表中缓存的当前这个 InstanceInfo 中的 lastDirtyTimestamp 小的情况，即后注册的反而过时。这是因为，若 Client 的 instanceInfo 的续约信息相继发生了两次变更，Client 就提交了两次 register()请求。但是由于网络原因，第二次注册请求先到达 Server。当第一次注册请求到达后就会出现“后到达的注册请求中携带的 instanceInfo 的最后修改时间反而过时”的情况。

【Q-07】Eureka Server 在处理 Client 的注册时请求时，是否可能出现新注册的 registrant 的 OverriddenStatus 状态不是 UNKNOWN 的场景呢？为什么？请谈一下你的看法。

【RA】这种场景是可能会出现的。且只有一种情况：当前注册的 registrant 是由于其续约信息发生了变更而引发的注册，且在续约信息变更之前用户通过 Actuator 修改过状态。

当然，这种通过 Actuator 修改的状态仅仅修改的是 Server 端注册表中的状态，并没有修改客户端的任何状态。这个修改的结果实际是通过客户端定时更新客户端注册表时，将所有变更信息下载到的客户端，其中就包含它自己状态的修改信息。

第 05 次直播课-4

【Q-01】Eureka Client 提交的下架请求，Eureka Server 是如何处理的？请谈一下你的认识。
【RA】Eureka Server 在接收到 Client 的下架请求后，主要完成了两项任务：

- 从注册表中将该提交请求的 intanceInfo 删除，并将其 overridden 状态从缓存 map 中删除，记录的删除时间戳，最后修改时间戳 lastUpdatedTimestamp 等。
- 将下架请求同步到其它 server。

【Q-02】Eureka Server 在处理 Client 的全量下载注册信息请求时可以设置从自己的只读缓存 readOnlyCacheMap 中获取到所有注册到自己的注册信息，而 readOnlyCacheMap 中的数据是定时同步的读写缓存 readWriteCacheMap 的。这样的话就存在一个问题：这个定时更新无论更新频率多么高，一定存在用户从 readOnlyCacheMap 中读取的数据与 readWriteCacheMap 中不一致的情况，为什么不直接从 readWriteCacheMap 中读取？也就是说，这样的设计好处是什么？请谈一下你的认识。
【RA】这样设计的目的是为了保证在并发环境下“集合迭代的稳定性”。集合迭代的稳定性指的是，当一个共享变量是集合且存在并发读写操作时，要保证在对共享集合进行读操作时能够读取到稳定的数据，即在读取时不能对其执行写操作，但又不能妨碍了写操作的执行。此时就可以将集合的读写功能进行分离，创建出两个共享集合：一个专门用于处理写功能，外来的数据写入到这个集合；一个专门用于处理读功能，定时同步写集合的数据。这种方案存在的弊端是，无法保证只读集合中的数据与读写集合中数据的随时完全一致。当然，这种不一致在下一次定时同步时就会达到一致。所以这种方案的应用场景是对数据的实时性要求不是很高的情况。

【Q-03】Eureka Server 在处理 Client 的全量下载注册信息请求与处理增量下载请求有什么不同？请谈一下你的认识。
【RA】Eureka Server 在处理 Client 的全量下载注册信息请求时，其读取的是当前 Server 注册表 registry 中的注册信息，而处理增量下载请求时根本就没有操作注册表 registry，而是直接读取了最近更新队列 recentlyChangeQueue 中的信息。这两种请求的处理方式，操作了两上不同的共享集合。

【Q-04】Eureka Server 在处理 Client 的增量下载注册信息请求时是从 recentlyChangeQueue 最近更新队列中直接获取的数据，这个 recently 是多久？如果超过了这个 recently 时间又是如何处理的？请谈一下你的认识。
【RA】最近更新队列 recentlyChangeQueue 中的 recently 是可能通过配置文件属性指定的，默认为 3 分钟。当 recentlyChangeQueue 中的元素超过了 3 分钟，那么系统会自动将这些过期的元素删除。只不过这个删除操作是一个 repeated 定时任务，在 AbstractInstanceRegistry 类的构造器中被创建并启动，默认每 30 秒执行一次。

第 06 次直播课-11

【Q-01】Eurek Server 对于像 Client 注册、状态修改等写操作添加的都是读锁，而对于像增量下载请求添加的是写锁，为什么？为什么对于续约请求这种写操作处理中没有添加读锁？为什么全量下载中没有添加写锁？对于 Eureka Server 中添加锁的方式，请谈一下你的认识。

【RA】总体来说，这种读/写锁的添加方式就是为了解决两个共享集合 recentlyChangedQueue 与注册表 registry 的“集合迭代稳定性”问题。即增加读锁是为了限制其增加写锁，而增加写锁也是为了限制其增加读/写锁。若仅是考虑到 recentlyChangedQueue 集合的迭代稳定性问题，完全可以在处理注册、状态修改、删除 overridden 状态、下架、续约等写操作请求时添加写锁，而在处理全量下载、增量下载请求时添加读锁。

但这样做对于注册表 registry 集合来说就会出现问题。由于续约请求是一个发生频率非常高的写操作处理，若为其添加了写锁，则意味着在进行续约处理时，其它任何对 registry 的读/写操作均将阻塞。所以，续约处理是不能加写锁的。那为其添加读锁是否可以呢？也不行。因为对于这么一个发生频率很高的处理，若添加了读锁，那么，几乎这个 registry 就会被读锁给锁定，其它任何写操作均将被阻塞。所以，续约处理不能加锁。

处理注册、状态修改、删除 overridden 状态、下架等写操作请求时，为什么要添加读锁呢？添加写锁不行吗？若添加写锁，则意味着，任意一个对 registry 的写操作请求处理，均将阻塞所有其它对 registry 的读/写操作，效率非常低。而若这些写操作添加的是共享锁读锁，则意味着，这些写操作可以同时进行。即使可能会出现对这些写操作同时操作同一个 registry 中的相同 instanceInfo 的情况，也不会出现问题。因为 registry 及 recentlyChangedQueue 都是 JUC 的，是线程安全的。

由于那些写操作添加了读锁，所以增量下载这种读操作添加了写锁，以保证对共享集合读/写操作的互斥。

为什么增量下载添加了写锁，而全量下载没有添加呢？因为增量下载中没有涉及对共享集合注册表 registry 的操作，而全量下载读取了 registry。若为全量下载添加写锁，则必然会导致其在读取期间出现续约请求处理被阻塞的情况。对于这种频率非常高的续约处理是不能停止的。

【Q-02】为什么写操作要添加读锁？

【RA】若对这些写操作添加写锁，是否可以呢？写锁是排它锁。若要为这些对 recentlyChangedQueue 进行的写操作添加写锁，则意味着当有一个写操作发生时，对 recentlyChangedQueue 的所有其它读/写操作，均会发生排队等待（阻塞），会导致效率低下。而若要添加读锁，则会使所有对 recentlyChangedQueue 执行的写操作实现并行，提高了执行效率。不过，这些写操作会引发线程安全问题吗？不会。因为 recentlyChangedQueue 是 JUC 的队列，是线程安全的。

需要注意，虽然我们的关注点一直都在 recentlyChangedQueue 上，但从代码角度来说，也为 registry 的写操作添加了读锁。不会影响 registry 的并行效率吗？不会。因为读锁是戈共享锁。

【Q-03】为什么读操作添加写锁？

【RA】为了保证对共享集合 recentlyChangedQueue 的读/写操作的互斥。不过，该方式会导致读操作效率的低下，因为读操作无法实现并行读取，只能排队读取。因为写锁是排它的。

【Q-04】读写锁反加应用场景是什么？
【RA】写操作相对于读操作更加频繁的场景。

【Q-05】续约操作能否添加写锁？
【RA】不能。因为续约操作是一个发生频率非常高的写操作，若为其添加了写锁，则意味着其它 client 的续约处理无法实现并行，发生排队等待。因为写锁是排它锁。

【Q-06】续约操作能否添加读锁？
【RA】不能。因为添加读锁的目的是为了与写锁操作实现互斥。在上述所有方法中，对 registry 的所有操作中均没有添加写锁，所以这里的写操作也无需添加读锁。

【Q-07】如果不加锁会怎样？
【RA】若不对 recentlyChangedQueue 的操作加锁，可能会存在同时对 recentlyChangedQueue 进行读写操作的情况。可能会引发对 recentlyChangedQueue 的迭代稳定性问题。

【Q-08】为什么全量下载没有添加写锁？
【RA】若为其添加了写锁，则必须会导致某 client 在读取期间，其它 client 的续约请求处理被阻塞的情况。

【Q-09】Eurek Server 对于续约过期的 Client 会定期进行清除，这个定时任务是何时启动的？由谁来完成的？又做了些什么？请谈一下你的认识。
【RA】Eurek Server 对于续约过期的 Client 的定时清除任务是在 Eureka Server 启动时专门创建了一个新的线程来执行的。确切地说，是 EurekaServerAutoConfiguration 在做实例化过程中完成的该定时任务线程的创建、执行。

这个定时任务首先查看了自我保护模型是否已经开启，若已经开启则不进行清除。若没有开启，则首先会将所有失联的 instance 集中到一个集合中，然后再计算“为了不启动自我保护模型，最多只能清除多少个 instance”，在进行清除时不能越过此最大值。

【Q-10】Eurek Server 对于续约过期的 Client 会定期进行清除时有一个“补偿时间”概念。什么是补偿时间，请谈一下你的认识。
【RA】对于“补偿时间”的理解，首先要清楚 repeated 定时器的执行原理。本次任务的执行条件有两个，这两个条件必须同时满足才会执行任务。一是，上次任务执行完毕，二是，按照配置文件中设置的清除间隔，本次任务的执行时间点也已经到了或过了。

那么什么是“补偿时间”呢？举例来说。

例如，正常情况下，若每 5s 清除一次过期对象，而清除一次需要 2s，则在第 5s 时开始清除 0s-5s 期间的过期对象。第 10s 开始清除 5s-10s 期间的过期对象。

若清除操作需要 6s，则在第 5s 时会开始清除，其清除的是 0s-5s 期间的过期对象。然而这次清除用时 6s，也就是说，在第 11s 时才清除完毕 0s-5s 期间的过期对象。按理说应该在第 10s 时开始清除 5s-10s 期间的过期对象，但由于上次的清除任务还未结束，所以在第 10s 时不能开始清除，而在第 11s 时开始清除操作。因为已经过了 10s 这个时间点。此时要清除的对象就应该是 5s-11s 期间过期的，而多出的那 1s 就是需要补偿的时间。

【Q-11】Eurek Server 对于续约过期的 Client 会定期进行清除，而这个清除过程中，系统会从过期对象数量 expiredLeases.size，与开启自我保护的阈值数量 evictionLimit 两个数值中选

择一个最小的数进行清除。根据这个最小值进行清除，是不是会导致自我保护模式永远无法启动的情况？若出现了很多的过期对象，即这个 expiredLeases.size 很大，而 evictionLimit 是固定的。那么其清除的一定是 evictionLimit 个过期对象。这样的话是否自我保护模型永远无法启动？

【RA】答案是否定的。自我保护模式的启动并不是由这个清除任务决定的，而是由其它线程决定。只要发现收到的续约数据低于阈值了，那么就会启动自我保护模式。这里选择最少的进行清除，是为了尽量少些清除，给那些没有被清除的对象以“改过自新，浪子回头”的机会。可能由于网络抖动导致的失联，网络现在好了，其还可以恢复。若直接清除掉，那么就没有这个恢复的机会了。

第 07 次直播课-12

【Q-01】Nacos 官方对于服务的描述用于到了四个概念：namespace、group、service。它们之间是什么关系，请谈一下你的认识。

【RA】从包含关系来说，这三个概念是越来越小。一个 namespace 下可以有很多的 group，一个 group 下可以包含很多的 service。但 service 并不是一个简单的微服务提供者，而是一类提供者的集合。service 除了包含微服务名称以外，还可以包含很多的 cluster，每个 cluster 中可以包含很多的 instance 提供者。instance 才是真正的微服务提供者主机。

【Q-02】Nacos 中注册的实例有临时实例与持久实例之分，它们有什么区别，请谈一下你的认识。

【RA】临时实例与持久实例的实例的存储位置与健康检测机制是不同的。

- 临时实例：默认情况。服务实例仅会注册在 Nacos 内存，不会持久化到 Nacos 磁盘。其健康检测机制为 Client 模式，即 Client 主动向 Server 上报其健康状态。默认心跳间隔为 5 秒。在 15 秒内 Server 未收到 Client 心跳，则会将其标记为“不健康”状态；在 30 秒内若收到了 Client 心跳，则重新恢复“健康”状态，否则该实例将从 Server 端内存清除。即对于不健康的实例，Server 会自动清除。
- 持久实例：服务实例不仅会注册到 Nacos 内存，同时也会被持久化到 Nacos 磁盘。其健康检测机制为 Server 模式，即 Server 会主动去检测 Client 的健康状态，默认每 20 秒检测一次。健康检测失败后服务实例会被标记为“不健康”状态，但不会被清除，因为其是持久化在磁盘的。其对不健康持久实例的清除，需要专门进行。

【Q-03】Nacos Client 中 NamingService 是一个比较重要的接口，请谈一下你的认识。

【RA】NamingService 用于完成 Client 与 Server 间的通信，例如注册/取消注册，订阅/取消订阅，获取 Server 状态，获取 Server 中指定的 Instance 等。这个接口只有一个实现类，那就是 NacosNamingService。当然，NacosNamingService 实例最终是调用 NamingProxy 实例完成的与 Server 间的通信。

【Q-04】Nacos 源码中经常会出现 serviceName 这个名，其指的是什么，是微服务名称吗？请谈一下你的认识。

【RA】Nacos 源码中的确出现了很多的 serviceName 变量名。这个变量名在不同的类中、不同的方法中其意义是不同的。有时其的确就仅仅代表微服务名称，但有时并不是。例如，

Instance 实体类中有一个 String 类型的成员变量 serviceName，其并不是简单的微服务名称，而是由 groupId@@微服务名称构成的字符串，表示这是一个服务名称。只所以要在微服务名称前添加上 groupId 作为服务名称，是因为在 Nacos 中允许出现在不同的 group 中存在相同微服务名称的服务，而这些服务是不同的服务。

【Q-05】公司需要将自研的注册中心与 Spring Cloud 整合，完成其 Client 的自动注册？请谈一下你的认识。
【RA】若要使一种注册中心与 Spring Cloud 整合后完成其 Client 的自动注册，那么就需要该注册中心的 Client 依赖实现 AutoServiceRegistrationAutoConfiguration 规范。确切地说，就是要自定义一个 Starter，完成 AutoServiceRegistration 实例的创建与注入。

【Q-06】NacosAutoServiceRegistration 中的 register()方法完成了自动注册，那么这个 register()方法是什么时候被调用执行的呢？请谈一下你的认识。
【RA】NacosAutoServiceRegistration 中的 register()方法完成了自动注册，而该方法只所以能够被调用执行，是因为 NacosAutoServiceRegistration 类实现了 ApplicationListener 接口，实现了对 Web 容器启动初始化的监听。一旦 Web 容器初始化完毕，就会触发 ApplicationListener 接口中 onApplicationEvent()方法的执行。而该方法的执行，调用了 register()方法的执行。

【Q-07】NacosAutoServiceRegistration 中的 register()方法完成了自动注册。但这个类其实仅仅是 Nacos 与 Spring Cloud 整合项目中的类，真正注册的实现是调用了 Nacos 核心代码中的注册完成的。请问，Nacos 核心源码中哪个类完成了 Client 的注册？请谈一下你的认识。
【RA】Nacos 的 Client 真正注册的实现是通过调用 Nacos 核心源码中 NacosNamingService 类中的 registerInstance()方法完成的。

【Q-08】Nacos Client 向 Server 发送心跳，是何时开始的？请谈一下你的认识。
【RA】Nacos Client 向 Server 发送心跳是在该 Client 自动注册到 Nacos Server 时开始的。不过，从源码上看，好像是先发送心跳然后才进行了注册。但由于发送心跳是由其它线程执行的一个定时任务，其实际开启这个定时任务是在完成了注册之后。

【Q-09】为了达到高可用效果，Nacos Server 都是以集群方式出现的。Nacos Client 配置文件中将 Nacos Server 集群中所有 Server 地址全部写上后，其到底连接的是哪台 Server 呢？请谈一下你的认识。
【RA】Nacos Client 会首先获取到其配置的所有 Server 地址，然后再随机选择一个 Server 进行连接。如果连接失败，其会以轮询方式再尝试连接下一台，直到将所有 Server 都进行了尝试。如果最终没有任何一台能够连接成功，则抛出异常。

【Q-10】Nacos Client 是通过什么技术向 Nacos Server 发送连接请求的？请谈一下你的认识。
【RA】简单来说 Nacos Client 是通过 NamingService 完成的与 Nacos Server 间的注册、订阅等连接功能。但这些连接功能实际是通过 NamingProxy 完成的，包括心跳的发送。其实，从根本上说，所有这些都是通过 Nacos 自定义的一个 HttpClientRequest 发出的所有向 Server 的请求。但这个 HttpClientRequest 最终是对 JDK 的 HttpURLConnection 进行的封装，发出的连接请求。

【Q-11】Nacos Client 向 Nacos Server 发送的注册、心跳、订阅等连接请求，都是通过

NamingService 完成的，对吧？请谈一下你的认识。
【RA】Nacos Client 向 Nacos Server 发送的注册、订阅、获取状态等连接请求是通过 NamingService 完成的，但心跳请求不是。如果想统一请求的提交，只能说 Nacos Client 向 Nacos Server 发送的所有请求都是通过 NamingProxy 完成的提交。

【Q-12】Nacos Client 向 Nacos Server 发送的注册、心跳、订阅等连接请求，都是通过 NamingProxy 发送的哪种请求方式的请求？请谈一下你的认识。
【RA】Nacos Client 向 Nacos Server 发送的注册、心跳、订阅等连接请求，分别提交的是 POST、PUT、GET 请求。当然，最终是通过其自研的、封装了 JDK 的 HttpURLConnection 的 HttpClientRequest 发出的请求。

第 08 次直播课-6

【Q-01】Nacos Discovery 应用在启动时会首先获取到所有可用的服务。对于这些服务的获取是通过谁获取到的？请谈一下你的认识。
【RA】Nacos Discovery 应用在启动时会首先获取到所有可用的服务。对于这些服务的获取实际是在应用启动时，通过 Actuator 的 health 监控终端获取到的。

【Q-02】Nacos Discovery 应用在启动时会完成其必须的自动配置。其中会完成以下两个配置类的加载与创建：NacosDiscoveryAutoConfiguration 与 NacosDiscoveryClientConfiguration。这两个配置类的主要功能是什么，它们间的关系是怎样的？请谈一下你的认识。
【RA】Nacos Discovery 应用在启动时会完成其必须的自动配置，会加载与创建上述两个自动配置类。其中 NacosDiscoveryAutoConfiguration 主要用于创建一个 NacosServiceDiscovrey 实例，而 NacosServiceDiscovrey 主要用于进行服务发现：获取到所有指定服务，或获取到与 Server 通信的 NamingService 等。NacosDiscoveryClientConfiguration 主要用于生成一个 DiscoveryClient 实例，而 Discovery 实例会通过 NacosServiceDiscovrey 完成指定服务的获取。所以，NacosDiscoveryClientConfiguration 配置类要求必须在 NacosDiscoveryAutoConfiguration 配置类完成创建后才可创建。这种先后关系是通过 NacosDiscoveryClientConfiguration 配置类中的@AutoConfigureAfter 完成的。

【Q-03】Nacos Client 是如何对其本地注册表进行更新的？请谈一下你的认识。
【RA】在 Client 启动时就会创建一个 NacosWatch 类。在创建该类实例时，不仅会向 Server 提交服务订阅请求，而且还会定时更新当前服务的服务列表。不过这里更新的是当前应用所提供服务的所有提供者列表，不包含其它服务的列表。即并不是更新的“本地注册表”，虽然本地注册表中可能会存在其它服务列表。

【Q-04】Nacos Client 的服务订阅与 Eureka Client 的服务订阅有什么区别与联系？请谈一下你的认识。
【RA】Nacos Client 的服务订阅与 Eureka Client 的服务订阅都是从 Server 中下载服务列表。但不同的是，Eureka Client 的服务订阅是定时获取 Server 端发生变更的服务实例并更新到本地注册表，而 Nacos Client 的服务订阅仅仅是定时从 Server 获取当前服务的所有实例并更新到本地。

【Q-05】Nacos Client 的 HostReactor 中有一个缓存 updatingMap，它的作用是什么？请谈一下你的认识。

【RA】Nacos Client 的 HostReactor 中有一个缓存 updatingMap，其中用于存放当前正在发生变更的服务。这个缓存 map 有些特殊，其 key 为 serviceName，即 groupId@@微服务名称，而 value 却为一个 new Object()。这个 map 真正有用的是其 key，value 实际没有什么意义。其就是利用了 map 中 key 的唯一性特征。当一个服务的 ServiceInfo 发生变更

【Q-06】Nacos Client 在启动后会定时更新本地注册表中其自己服务的数据，而本地注册表中还包含很多其它服务。那么，这些服务是何时更新的？请谈一下你的认识。

【RA】Nacos Client 对本地注册表中当前服务之外服务的更新是在其第一次对该服务的访问开始的。这次访问不仅从注册表中获取到了其要访问服务，而且还开启了对该服务的定时更新操作。

第 09 次直播课-12

【Q-01】Nacos 源码工程可以以调试的方式启动运行吗？请谈一下你的认识。

【RA】Nacos 源码工程可以以调试的方式启动运行。有两种方式：一种是单机模式，一种是集群模式。

- 对于单机模式，若使用的是 Nacos 1.3.1 及其之前版本，直接找到 console 模块下的 Nacos 启动类运行即可。若为 Nacos1.3.1 之后版本，则需要在运行之前添加动态参数 -Dnacos.standalone=true。因为这些版本默认是以集群模式启动的。
- 对于集群模式，若使用的是 Nacos 1.3.1 之后版本，其默认就是集群模式启动。但仍需修改 console 模块中 resources 目录下的 application.properties 配置文件。修改两处内容：不同的 server 指定不同的端口号，指定外部存储设备。因为集群模式不允许使用内置的 mysql。

【Q-02】Nacos Server 对于 Client 提交请求的接受与处理是在哪里开始的？请谈一下你的认识。

【RA】Nacos Server 对于 Client 提交请求的接受与处理是由 Controller 完成的，包含 Client 的注册、订阅、心跳等请求。这些 Controller 一般是在 Nacos 源码工程的 naming 模块下的 com.alibaba.nacos.naming.controllers 包中定义。

【Q-03】Nacos 中有一个 SCI 模型。请谈一下你的认识。

【RA】Nacos 中的 SCI 模型，即 Service-Cluster-Instance 模型。其描述的关系是，一个微服务名称指定的服务 Service，可能是由很多的实例 Instance 提供，这些 Instance 实例可以被划分到多个 Cluster 集群中，每个 Cluster 集群中包含若干的 Instance。所以，在 Service 类中我们可以看到其包含一个很重要的集合，clusterMap，其 key 为 clusterName，而 value 是 Cluster。而在 Cluster 类中包含两个重要集合：持久实例 Set 集合 persistentInstances，与暂时实例 Set 集合 ephemeralInstances。所以简单来说，SCI 模型就是，一个 Service 包含很多的 Cluster，而一个 Cluster 包含很多的 Instance，这些 Instance 提供的就是 Service 所指定的服务。

【Q-04】Nacos 中的服务具有保护阈值，Eureka 中也设置有保护阈值。这两种保护阈值有什

么异同点，请谈一下你的认识。

【RA】Nacos 中的服务具有保护阈值，Eureka 中也设置有保护阈值。这两种保护阈值的异同点是：

- 相同点：都是一个 0-1 的小数，表示健康实例占所有实例的比例。
- 保护方式不同：Eureka 的保护一旦开启，其将不再从注册表中清除那些不健康的实例，是为了等待这些不健康实例能够“迷途知返，改过自新，重返健康”。但 Nacos 的保护一旦开启，则会让消费者消费不健康实例。通过牺牲消费者权益来达到自我保护的目的。注意，Eureka 任何时候都不能访问不健康的 instance。
- 范围不同：Nacos 的这个阈值是针对某个具体 Service 的，而不是针对所有服务的。而 Eureka 的自我保护阈值则是针对所有服务的所有提供者实例的。

【Q-05】Nacos Server 源码中大量的使用了 WebUtils 中的两个方法 optional()与 required()，这两个方法有什么区别与联系？请谈一下你的认识。

【RA】Nacos Server 源码中大量的使用了 WebUtils 中的两个方法 optional()与 required()，它们的异同点为：

- optional()方法是从请求中获取指定属性值，这个属性可能在请求中不存在。若不存在，则给其一个指定的默认值。
- required()方法也是从请求中获取指定属性值，但这个属性在请求是是一定存在的，是必须的值，所以其没有默认值。

【Q-06】Nacos Server 是如何处理 Client 的注册请求的？请谈一下你的认识。

【RA】Nacos Server 对于 Client 的注册请求处理过程，主要由两大环节构成：

- 若本要注册表中不存在注册 instance 的 service，则创建一个。
- 从其它 nacos 中同步获取当前 service 所包含的所有“有效”instance，然后将这个注册的 instance 写入到这个“有效”instance 集合，然后通过一致性服务，将这个“有效”instance 集合同步到所有 server，当然也包括当前 server。

需要注意的是，这个注册过程并没有直接将 instance 写入到当前 server 的注册表中，而是通过对“有效”instance 集合操作，并将这个“有效”instance 集合同步到所有 server 实现的注册。这做好的好处是不仅更新了本地注册表中的该服务，同时也更新了其它 Server 注册表中的该服务。一石二鸟，一箭双雕，一举多得。

【Q-07】Nacos Server 中当一个 Service 创建完毕后，系统会为其执行哪些主要操作？请谈一下你的认识。

【RA】当一个 Service 创建完毕后，一般会为其执行三项重要操作：

- 将该新建 Service 放入到注册表中，即放入到 ServiceManager 中的 serviceMap 缓存中。
- 为其内部的临时实例与持久实例添加一致性服务监听。
- 初始化该 service 内部的健康检测任务。

【Q-08】Nacos Server 中当一个 Service 创建完毕后，系统会为其初始化内部的健康检测任务。这些健康检测任务具体指的是什么？请谈一下你的认识。

【RA】Nacos Server 中当一个 Service 创建完毕后，系统会为其初始化内部的健康检测任务。主要包含两方面内容：开启定时清除过期 instance 任务，及为其所包含的每一个 cluster 初始化心跳定时任务。

【Q-09】Nacos Server 中当一个被创建完毕后，系统会为其初始化健康检测任务。什么是 Cluster 的健康检测任务？请谈一下你的认识。

【RA】Nacos Server 中当一个被创建完毕后，系统会为其初始化健康检测任务。这个健康检测任务，其实就是将 cluster 中所有持久实例的心跳检测任务定时添加到一个任务队列 taskQueue 中。后期再由其它线程来定时处理 taskQueue 中的任务。因为持久实例的健康检测是 Server 向 Client 发送心跳。

【Q-10】Nacos Server 中当检测到某 Instance 实例超时超过了 30s，需要将其从注册表中删除。这个删除操作是如何完成的？请谈一下你的认识。

【RA】Nacos Server 中当检测到某 Instance 实例超时超过了 30s，需要将其从注册表中删除。这个删除操作是以异步方式，通过向自己提交一个删除请求来完成的。而这个删除请求是通过 Nacos 自研的 HttpClient 提交的。这个 HttpClient 实际是对 Apache 的异步 HttpClient，即 CloseableHttpAsyncClient 进行的封装，也就是说，请求最终是通过 CloseableHttpAsyncClient 提交的。

【Q-11】Nacos Server 中对过期的 Instance 是如何实现清除的？请谈一下你的认识。

【RA】在 Service 的初始化时开启了清除过期 Instance 实例的定时任务，只不过，其清除操作与“处理客户端注销请求”进行了合并。由当前 Nacos Server 向自己提交一个 delete 请求，由 Server 端的“处理客户端注销请求”的方法进行处理。

【Q-12】Nacos Server 是如何处理 Client 的注销请求的？请谈一下你的认识。

【RA】Nacos Server 对于 Client 的注销请求处理过程并没有从本地 Server 注册表中直接清除掉指定的 instance，而是先从其它 nacos 中同步获取到要注销 instance 所属的 service 所包含的所有“有效”instance 集合，然后将这个要注销的 instance 从这个“有效”instance 集合中清除。最后通过一致性服务，将这个“有效”instance 集合同步到所有 server，当然也包括当前 server。

这做好的好处是不仅更新了本地注册表中的该服务，同时也更新了其它 Server 注册表中的该服务。一石二鸟，一箭双雕，一举多得。

第 10 次直播课-9

【Q-01】Nacos Server 对于 Client 的心跳处理逻辑是怎样的？请谈一下你的认识。

【RA】Nacos 处理客户端心跳，就是在注册表中找到这个 instance。若没有该 instance，则创建一个后再注册到注册表。若有该 instance，则更新其最后心跳时间。

【Q-02】为什么会出现心跳请求到了 Server 而注册表中却没有找到该 instnace 的情况？请谈一下你的认识。

【RA】出现心跳请求到了而注册表中却没有该 instnace 的情况，有两种可能：

- 注册请求先提交了，但由于网络原因，该请求到达 Server 之前心跳请求先到达了
- 由于网络抖动，Client 正常发送的心跳没有到达 Server，Server 将该 instance 从注册表中清除了。后来网络恢复正常后，Server 又收到了 Client 的心跳。

【Q-03】Server 中对于临时实例与持久实例健康状态的记录有什么不同？请谈一下你的认识。
【RA】临时实例与持久实例在 Server 端健康状态的记录方式是不同的。持久实例通过 marked 属性来表示：为 false 则为未标识，表示健康；为 true 则为被标识，表示不健康。临时实例是通过 health 属性来表示：为 true 表示健康；为 false 表示不健康。对于临时实例，其 marked 属性永远为 false。也就是说，只要一个实例的 marked 属性为 true，那么其一定是一个持久实例，且为不健康的。但仅凭 marked 为 false 是无法判断一个实例是否为临时实例，更无法判断其健康状态。

【Q-04】在处理 Client 订阅请求时，Server 主要做了哪些工作？请谈一下你的认识。
【RA】在处理 Client 订阅请求时，Server 主要完成了两项重要任务：

- 创建了该 Nacos Client 对应的 UDP 通信客户端 PushClient，并将其写入到了缓存 clientMap
- 从注册表中获取到指定服务的所有可用的 instance，并将其封装为 JSON

【Q-05】只所以 Nacos Server 能够与 Nacos Client 间保持 UDP 通信，是因为在 Nacos Server 中维护着一个缓存 map，其中存放着用于通信的数据。请谈一下你对这个缓存 map 的认识。
【RA】只所以 Nacos Server 能够与 Nacos Client 间保持 UDP 通信，是因为在 Nacos Server 中维护着一个缓存 map，这个 map 是一个双重 map。外层 map 的 key 为服务名称，格式为 namespaceId##微服务名称，value 为内层 map。而内层 map 的 key 为 instanceId，value 为 Nacos Server 与 Nacos Client 进行 UDP 连接的 PushClient。这个 PushClient 是包含这个 Nacos Client 的 port 等数据。也就是说，Nacos Server 中维护着每一个注册在其中的 Nacos Client 的 PushClient。

【Q-06】Nacos Server 与 Client 间可以通过 UDP 进行通信，这种通信可能会发生在什么情况下？请谈一下你的认识。
【RA】当 Nacos Server 通过心跳机制检测到其注册表中维护的 instance 实例 healthy 健康状态变为了 false 时，其需要将这个变更通知到所有订阅该服务的所有 Client。这个通知的发送方式是：变更触发一个服务变更事件的发布，而该事件会触发 Server 通过 UDP 通信将数据报发送给 Client。

【Q-07】Nacos Server 与 Nacos Client 间的 UDP 通信，谁是 Server，谁是 Client？请谈一下你的认识。
【RA】Nacos Server 与 Nacos Client 间的 UDP 通信，Nacos Server 充当着 UDP 通信的 Client 端，而 Nacos Client 充当着 UDP 通信的 Server 端。所以，在 Nacos Client 中有一个线程处于无限循环中，以随时检测到 Nacos Server 推送来的数据。

【Q-08】Nacos Server 与 Nacos Client 间的 UDP 通信，Nacos Server 作为 UDP 通信的 Client 端，其需要知道其要连接的 UDP Server，即 Nacos Client 的端口号。其是如何知道这个端口号的？请谈一下你的认识。
【RA】Nacos Server 与 Nacos Client 间的 UDP 通信，Nacos Server 作为 UDP 通信的 Client 端，其需要知道其要连接的 UDP Server，即 Nacos Client 的端口号。在 Nacos Client 定时从 Nacos Server 获取数据时，会随着请求将其 port 发送给 Nacos Server。

【Q-09】Nacos Server 在哪些情况下可能会引发其维护的注册表中 instance 的 healthy 状态发生变更？请谈一下你的认识。

【RA】当 Server 端定时清除过期 instance 任务检测到某 instance 超过 15s 未发送心跳时，会将其 healthy 由 true 变为 false；当 Server 又重新收到 healthy 状态为 false 的 instance 的心跳时，会将其 healthy 由 false 变为 true。

第 11 次直播课-6

【Q-01】Server 中 ServiceManager 管理着当前 Server 中所有的服务数据。ServiceManager 实例在创建完毕后会开启三项非常重要的任务，哪三项？请谈一下你的认识。
【RA】Server 中 ServiceManager 管理着当前 Server 中所有的服务数据。ServiceManager 实例在创建完毕后会开启三项非常重要的任务，其中包含两项定时任务：

- 开启定时任务：每 60 秒向其它 nacos 发送一次本机注册表
- 从其它 nacos 获取注册表中所有 instance 的最新状态并更新到本地
- 开启定时任务：每 20 秒清理一次注册表中的空 service

【Q-02】Server 中是如何清除没有 instance 实例的 Service 的？请谈一下你的认识。
【RA】Server 在启动时就启动了一个定时任务：清除没有 instance 实例的 Service，即清除为空的 Service。但这个清除并不是直接的暴力清除，即并不是在执行定时任务时一经发现空的 Service 就立即将其清除，而是仅使一个记录该 Service 为空的计数器增一。当发现计数器值超出了设定好的清除阈值时才将该 Service 清除。另外，这个清除工作并不是直接在本地注册表中将其清除，而是通过一个一致性操作来完成的。这做好的好处是不仅清除了本地注册表中的该服务，同时也清除了其它 Server 注册表中的该服务。一石二鸟，一箭双雕，一举多得。

【Q-03】Spring Cloud 中 OpenFeign 的@EnableFeignClients 与@FeignClient 两个注解有怎样的区别与联系。请谈一下你的认识。
【RA】@EnableFeignClients 注解主要用于查找所有的@FeignClient 接口，并为这些 Feign Client 设置一些默认的配置。而@FeignClient 则是就某一个特定的 Feign Client 进行配置。一个 Consumer 中只能有一个@EnableFeignClients，但可以有多个@FeignClient。

【Q-04】Spring Cloud 中 OpenFeign 中@FeignClient 注解中有一个 name 属性与 path 属性。这两个属性是什么意思？请谈一下你的认识。
【RA】Spring Cloud 中 OpenFeign 中@FeignClient 注解中的 name 属性用于指定当前 Feign Client 的名称，这个名称为其要消费的提供者微服务名称。消费者对提供者的负载均衡就是通过这个名称实现的。而 path 属性用于指定直连方式的连接的提供者的 ip 与 port。一旦指定了 path 属性，就不再使用负载均衡方式来选择提供者了。

【Q-05】Spring Cloud 中 OpenFeign 中有一个 FeignClientSpecification 类，这个类是干嘛的？请谈一下你的认识。
【RA】FeignClientSpecification 是一个 Feign Client 的生成规范。所谓生成规范就是该实例中定义了生成 Feign Client 的各种配置信息，在动态生成 Feign Client 时需要按照这些配置信息来生成。具体的规范存放在 configuration 属性中。

【Q-06】Spring Cloud 中 OpenFeign 中有一个 FeignContext 类，这个类是干嘛的？请谈一下你的认识。

【RA】FeignContext 是一个为 Feign Client 创建所准备的上下文对象，从这个对象中可以获取到每一个要创建的 Feign Client 所需要的 Spring 子容器，而在这些子容器中存放着创建每一个不同的 Feign Client 所需要的“原材料”bean。一个应用只会有一个 FeignContext 实例。

第 12 次直播课-12

【Q-01】Spring Cloud 中 OpenFeign 中对于 Feign Client 的创建源码解析，从哪里下手开始呢？请谈一下你的思路。

【RA】对于 Feign Client 的创建源码解析从@EnableFeignClients 着手分析是比较好的。该注解中@Import 了一个 FeignClientsRegistrar 类。该类实现了 ImportBeanDefinitionRegistrar 接口，即实现了该接口的 registerBeanDefinitions()方法。这个接口就是专门处理配置类的。而 @EnableFeignClients 注解的的 defaultConfiguration 与@FeignClient 注解中的 configuration 属性都是配置类，就是由这个接口方法处理的。所以就从 FeignClientsRegistrar 类的方法 registerBeanDefinitions()入手分析。

【Q-02】关于 volatile 与 synchronized，很多时候会一起使用，为什么？请谈一下你的认识。

【RA】首先要清楚，在系统中存在这样的两类内存。一类是为每个线程单独分配的工作内存，即高速缓存。一类是为当前应用（进程）分配的主内存。对于共享变量，其是同时存在于每个线程的工作内存与主内存中的。一个线程若要修改共享变量的值，其首先需要从主内存中将共享变量值读取到自己的工作内存，然后在自己的工作内存中进行修改。即对于共享变量的修改是在各个线程的工作内存中进行的。而对于共享变量的读取，则是从主内存中读取的。

为了使共享变量在线程的工作内存中修改后的值能够立即更新到进程主内存，就需要为共享变量添加 volitile 修饰符，即 volitile 可以保证共享变量值对所有线程的“可见性”。但在并发环境下对于共享变量的修改“有序性”就需要使用 synchronized 了。将对共享变量的修改代码放入到 synchronized 语句块中，就可以保证了对共享变量修改的“有序性”。

将 synchronized 与 volatile 联合使用就可以保证了“我在修改时，你们都不能改。但我改过了，你们都可以看到”。当然，若对主内存中的共享变量的某个时刻的值，同时有多个线程读取到了各自的工作内存，其是通过“乐观锁”机制解决版本冲突问题的。

【Q-03】什么是迭代稳定性问题？请谈一下你的认识。

【RA】迭代稳定性问题其实就是共享集合迭代稳定性问题。其描述的问题是，在并发处理的场景下，对某实例中共享集合的访问中，若在对该集合进行写操作的同时，又有线程对其执行迭代访问，此时迭代访问的结果可能会出现稳定性问题。

为了解决这个问题，可以采用以下三种方案：

- 读/写操作均添加读/写锁
- 引入专门的只读集合
- 集合替换

【Q-04】Spring Cloud 中 OpenFeign 的@EnableFeignClients 注解中 value、basePackages 与 basePackageClasses 属性均可以用于指定要扫描 Feign Client 的基本包。这三个属性值是什么关系？请谈一下你的认识。

【RA】Spring Cloud 中 OpenFeign 的@EnableFeignClients 注解中 value、basePackages 与 basePackageClasses 属性均可以用于指定要扫描 Feign Client 的基本包，只要这三个属性都指定了值，那么，这些指定的包均会扫描。

【Q-05】Spring Cloud 中 OpenFeign 的@EnableFeignClients 注解中 value、basePackages、basePackageClasses 属性与 clients 属性均可以用于指定要扫描 Feign Client 的位置。这五个属性值是什么关系？请谈一下你的认识。

【RA】Spring Cloud 中 OpenFeign 的@EnableFeignClients 注解中 value、basePackages、basePackageClasses 属性与 clients 属性均可以用于指定要扫描 Feign Client 的位置。其中 value、basePackages、basePackageClasses 属性用于指定要扫描的基本包，而 clients 属性用于指定与 Feign Client 同包的类或接口。若指定了 clients 属性，则用于指定基本包的属性将不起作用。

【Q-06】Spring Cloud 中 OpenFeign 的@FeignClient 注解中 value、contextId、name 与 serviceId 属性均可以用于指定要生成的 Feign Client 的名称。这四个属性值是什么关系？请谈一下你的认识。

【RA】Spring Cloud 中 OpenFeign 的@FeignClient 注解中 value、contextId、name 与 serviceId 属性均可以用于指定要生成的 Feign Client 的名称。这四个属性值若都指定了不同的值，那么只会有一个起作用，它们的优先级由高到低分别是 contextId、value、name、serviceId。

【Q-07】对于共享集合，为何有的需要考虑迭代稳定性而有的不需要？请谈一下你的看法。

【RA】对于共享集合，在并发环境下，若存在某些线程对其进行修改时，其它线程可能会对其进行遍历迭代访问的情况，那么就需要考虑迭代稳定性问题。若不存在遍历迭代的情况，就无需考虑。

【Q-08】Spring Cloud 中 OpenFeign 的 FeignContext 对于 Feign Client 的创建非常的重要。其中维护着一个重要集合 contexts，该集合中都保存了什么内容？请谈一下你的认识。

【RA】Spring Cloud 中 OpenFeign 的 FeignContext 中维护着一个重要集合 contexts，该集合是一个线程安全的 map。其 key 为每个@FeignClient 的 name 属性，即微服务名称，而 value 则为该@FeignClient 所对应的一个 Spring 子容器对象，该容器中存放着该 Feign Client 创建所需要的所有对象。

【Q-09】Spring Cloud 中 OpenFeign 的 FeignContext 对于 Feign Client 的创建非常的重要。其中维护着一个重要集合 configurations，该集合中都保存了什么内容？请谈一下你的认识。

【RA】Spring Cloud 中 OpenFeign 的 FeignContext 中维护着一个重要集合 configurations，其是一个线程安全的 map。其 key-value 分为两类：

- 第一类只有一个，key 是字符串“default + 当前启动类的全限定性类名”，而 value 为@EnableFeignClients 中 configuration 的值；
- 第二类可以有多个，key 为当前@FeignClient 的 name 属性名，即微服务名称，value 则为每个@FeignClient 的 configuration 的属性值。这一类 key-value 对会有很多，有多少的@FeignClient 就会有多少的这类 key-value。

【Q-10】在一个类或方法上出现的 synthetic 是什么意思？请谈一下你的认识。
【RA】synthetic，合成的。当一个类或方法前出现了该关键字，则说明该类或方法是由编译器自动生成的。这个关键字一般不是我们人为写到代码中的，而是由编译器编译生成的字节码文件中。

例如，当一个内部类中的方法要访问外部类的 private 成员时，编译器会为该内部类方法前自动添加上 synthetic 修饰符，表示该类中的方法可以访问到那个 private 成员。
编译器为什么要为其添加这个修饰符呢？因为对于编译器来说，无论是内部类还是外部类，其编译出的字节码文件都是一个独立的文件。为了标识出这个内部类文件的方法与其它类文件的方法的不同，编译器会为该类自动添加上这个修饰符，以使其可以访问到另一个类的 private 成员。

当然，若一个方法是由编译器内部生成的，那么这个编译器生成的这个方法前也会自动添加上 synthetic 关键字的。

【Q-11】Spring Cloud 中 OpenFeign 中创建的 Feign Client 是一个什么对象？请谈一下你的看法。
【RA】Spring Cloud 中 OpenFeign 中创建的 Feign Client 是一个 JDK 的 Proxy 动态代理对象。确切地说，这个 proxy 并不是 Feign Client 的代理对象，而是 Feign Client 与当前被调用的 Feign 方法结合体的代理对象。即当前 Feign Client 有多个 Feign 方法被调用，就会创建出多个 proxy 代理对象。当然，若当前某一个 Feign 方法同时被多个线程调用，那么就会创建出多个 proxy 代理对象。

【Q-12】Spring Cloud 中 OpenFeign 中 Feign Client 的一个方法调用是如何发出去的？如果查看这个源码？请谈一下你的思路。
【RA】Spring Cloud 中 OpenFeign 中创建的 Feign Client 是一个 JDK 的 Proxy 动态代理对象，所以若要解析 Feign 方法的调用，可以找到 JDK 的 Proxy 生成时的 InvocationHandler 实例的 invoke()方法，该方法是 Feign 方法执行后系统立即调用的方法。

第 13 次直播课-13

【Q-01】Ribbon 负载均衡策略是允许自定义的。如何定义？请谈一下你的看法。
【RA】Ribbon 负载均衡策略是允许自定义的。只需定义一个 IRule 接口的实现类，实现其 choose()方法即可。这里需要说明一点的是，这个 choose()方法有一个 Object 类型的参数 key，其作用是，如果选择算法需要外部传入一些值作为选择条件，那么可以通过这个参数 key 传入。由于这个 key 为 Object 类型，所以其可以传入任意数据，包括包含了很多条件的自定义类型实例，或集合实例等。

【Q-02】Ribbon 负载均衡策略实现中大量使用到了 Thread.yield()方法。这个方法的作用是什么？请谈一下你的看法。
【RA】Ribbon 负载均衡策略实现中大量使用到了 Thread.yield()方法。Thread.yield()方法是让当前线程暂时让出当前的 CPU 执行权，即让出当前的时间片，由“执行态”变为“就绪态”。目的是，如果现在获取到的 Server 为空，就让当前线程先等上一会儿，但也不是阻塞，而是再到就绪队列中排队等待再次分配到 CPU。可能这个等待的时间中，很多不可用的 Server

变为了可用的。

【Q-03】Ribbon 内置的负载均衡策略中默认的是轮询策略。在轮询查找到的 server 若不可用，其会进行重试。但重试后找到的 server 仍不可用，那么其会一直重试下去直到找到为止吗？请谈一下你的看法。

【RA】Ribbon 内置的负载均衡策略中默认的是轮询策略。在轮询查找到的 server 若不可用，其会进行重试。但其最多重试 10 次，若还未找到可用的 server，则返回 null。并且，这个最多重试 10 次也是不能修改的。

【Q-04】Ribbon 内置的负载均衡策略中有随机选择策略。在查找到的 server 若不可用，其会进行重试。但重试后找到的 server 仍不可用，那么其会一直重试下去直到找到为止。这是与轮询策略不同的，为什么？请谈一下你的看法。

【RA】Ribbon 内置的负载均衡策略中有随机选择策略。在查找到的 server 若不可用，其会进行重试。但重试后找到的 server 仍不可用，那么其会一直重试下去直到找到为止。而轮询策略最多会重试 10 次，若还未找到，则返回 null。为什么这两种策略对于找不到可用 server 的重试机制不同呢？因为轮询策略是从所有 server 列表 allServers 中查找的，其中可能会存在不可用的 server；而随机策略则是从所有可用 server 列表 upServers 中查找的，其中基本都是可用的。如果有不可用的，也是刚刚发生了变化，变为了不可用。但这种情况很少。所有轮询策略若允许一直重试下去的话，有可能会永远重试。而随机策略基本不会重试几次就可找到可用 server。

【Q-05】Ribbon 内置的负载均衡策略中的随机选择策略，在获取随机数时使用的是线程安全的随机数生成器 ThreadLocalRandom。为什么没有使用普通的随机数生成器呢？请谈一下你的看法。

【RA】Ribbon 内置的负载均衡策略中的随机选择策略，在获取随机数时使用的是线程安全的随机数生成器 ThreadLocalRandom，而没有直接使用普通的随机数生成器。本地线程随机生成器与普通随机生成器的区别主要是随机种子的产生不同。普通随机生成器（例如 JDK 的 Random 随机生成器）的随机种子，一般是固定的，即只要生成器生成了，那么其随机种子就计算完毕并固定了。固定随机种子产生的随机数实质上是伪随机数，只要随机种子相同，其生成的随机数顺序是固定的。ThreadLocalRandom 的随机种子是不固定的，本次随机数获取所所使用的随机种子，是从当前线程的 ThreadLoacl 中获取到的上次的随机种子变化而来的。对于每个线程，其每次获取到的随机数都是真正的随机数，是不可“复制”的。

【Q-06】Ribbon 内置的负载均衡策略中的重试选择策略，是在选择出的 Server 不可用时，可以进行选择重试。可以重试几次呢？请谈一下你的看法。

【RA】Ribbon 内置的负载均衡策略中的重试选择策略，是在使用轮询方式选择出的 Server 不可用时，可以进行选择重试。不过，其重试不是从次数上进行限制的，而是从时间上进行限制的。默认可以在 500ms 内重试，如果仍没找到可用的 server，则返回 null。

【Q-07】对于一套微服务系统平台来说，为什么需要网关，网关的作用是什么？请谈一下你的认识。

【RA】对于一套微服务系统平台来说，网关是必须的组成部分。其作为系统唯一对外的入口，介于客户端与业务微服务器之间，用于完成对请求的鉴权、限流、路由、监控等功能。

【Q-08】Reactor、Reactive 及 WebFlux 间的关系是怎样的？请谈一下你的认识。

【RA】Reactive 本质上是一套技术栈，其中 Reactive Programming 是一种新的编程范式、编程思想。而 Reactive Streams 则是这种思想在 Java 语言中的规范的定义。而 Reactor 则是 Reactive Streams 这套规范的具体实现库。WebFlux 是使用 Reactor 这套库实现的框架。这就是它们间的关系。

【Q-09】RxJava 与 Reactive 间的关系是怎样的？请谈一下你的认识。

【RA】Rx，即 Reactive Extensions ，最早是.Net 平台对 Reactive Programming 的实现库。后来迁移到 Java 平台之后就变为了著名的 RxJava 库。所以，RxJava 是一套 Reactive Streams 规范库。并且，其还是 Reactive Streams 规范的基础，产生于 Reactive Streams 规范之前。不过，虽然可以和 Reactive Streams 规范的接口进行转换，但是由于底层实现的不同，使用起来并不是很方便，存在很多不足。

说到 RxJava，不得不提的是 RxJava2。RxJava2 也是一套 Reactive Streams 规范库，不过，其产生于 Reactive Streams 规范之后，在 RxJava 的基础上做了很多的更新，在设计和实现时考虑到了与 Reactive Streams 规范的整合，不过为了保持与 RxJava 的兼容性，很多地方在实现时受到了 RxJava 的严重束缚，使用起来仍然很不方便。

【Q-10】RxJava、RxJava2 与 Reactor 间的区别与联系是怎样的？请谈一下你的认识。

【RA】RxJava、RxJava2 与 Reactor 都是 Reactive Streams 规范的实现库。不同的是，它们产生的时间不同，对 Reactive Streams 实现的程度不同。

- RxJava，产生于 Reactive Streams 规范之前，是 Reactive Streams 规范的产生基础。它其实并不能算是 Reactive Streams 规范的实现库。
- RxJava2，产生于 Reactive Streams 规范之后，实现了 Reactive Streams 规范，但为了兼容 RxJava，其又受到 RxJava 的严重束缚。对 Reactive Streams 规范的实现并不纯粹。
- Reactor，产生于 Reactive Streams 规范之后，是一套完全基于 Reactive Streams 规范的、全新的库。其与 RxJava 没有任何关系。

【Q-11】在 Spring WebFlux 中有两个很重要的概念：Flux 与 Mono。请谈一下你以这两个概念的认识。

【RA】在 Spring WebFlux 中有两个很重要的概念：Flux 与 Mono。这两个都是异步序列，简单来说，Reactive Streams 规范就是对这两个异步序列中元素的处理。

- Flux：一个包含 0 个或多个元素的异步序列。
- Mono：一个包含 0 个或 1 个元素的异步序列。

【Q-12】在 Spring Cloud Gateway 中有三个非常重要的概念：route、predicate 与 filter。请谈一下你对它们的认识。

【RA】在 Spring Cloud Gateway 中有三个非常重要的概念：route、predicate 与 filter。它们的关系是，route 是网关的最基本组成，由一个 routeId、一个目标地址 url，一组 predicate 工厂及一组 filter 组成。若 predicate 的结果为 true，则请求将经由 filter 链的 pre 部分处理后被路由到目标 url。在获取到了响应后，响应再经由 filter 链的 post 部分处理后返回给用户。

【Q-13】若要使用 Spring Cloud Gateway，除了在 pom 文件中添加其依赖外，还有什么要求？请谈一下你对它们的认识。

【RA】若要使用 Spring Cloud Gateway 则需要注意，在类路径下还不能具有 servlet 中的

DispatcherServlet 类，必须具有 reactive 中的 DispatcherHandler 这个类。所以，需要具有 spring boot starter webFlux 依赖，但不能导入 spring cloud starter web 依赖。在 pom 中导入 spring cloud starter gateway 的依赖即可。

第 14 次直播课-4

【Q-01】请简单谈一下你对 Zuul 与 Zuul2.0 的认识。

【RA】Zuul 是由 Netflix 开源的 API 网关，其目前有两个大的版本 1.0 与 2.0。

Zuul1.0 是基于 servlet 的，使用阻塞 API，它不支持任何长连接。即一个线程处理一次连接请求，这种方式在内部延迟严重，在由于设备故障而引发存活的连接增多的情况下，占用的线程数量急剧增加的情况。

Zuul2.0 的巨大区别是它运行在异步和无阻塞框架上，客户端与 Zuul 之间为长连接，每个 CPU 内核占用一个线程，通过 Reactor 模型处理所有连接上的请求和响应，请求和响应的生命周期是通过事件和回调来处理的，这种方式减少了线程数量，因此开销较小。

【Q-02】一个请求，从进入到 Spring Cloud Gateway 中，到最后向客户端响应，都经历了些怎样的重要步骤与过程？请谈一下你对它们的认识。

【RA】其实当 Client 发送请求到 Gateway 后，请求会被转发给分发处理器 DispatcherHandler。该处理器会将请求分发到相应的 RoutePredicateHandlerMapping。这个 Mapping 会遍历所有 Route，逐个尝试着与它们的 Predicate 进行匹配。匹配上一个路由后，就会将请求发送到该路由对应的 FilteringWebHandler，由这个处理器负责组装过滤器链。

通过过滤器链中各个 Filter 的处理后的请求，最终会通过 Netty 将请求发送给目标服务器。目标服务器处理后的响应会再次经过过滤器链的各个 Filter 处理，最终返回给 Client。

【Q-03】在 WebFlux 编程中经常会见到用于异步序列中元素转换的方法，例如 flatMap()、concatMap()、map()等。这些方法有什么区别？请谈一下你对它们的认识。

【RA】像 flatMap()、concatMap()、map()等这些方法，在 WebFlux 编程中经常会见到。它们的作用都是将异步序列中的元素进行转换的。不过，它们的不同点也是很明显的。

- concatMap()：其是 Flux 的方法，其不仅会将其原有元素转换为指定的元素。这个指定的元素来自于一个 Mono 中的元素，而这个 Mono 一般则是由一个返回 Mono 的方法返回的。最关键的是，序列中的原有元素是有序的，在转换为新的元素后，序列中的新元素还可以保持原有元素的有序性。
- flatMap()：其是 Flux 与 Mono 中都存在的方法。与 concatMap()的不同点是，其不存在有序性问题。
- map()：其是 Flux 与 Mono 中都存在的方法。这个方法与 concatMap()及 flatMap()有着很大的区别。map()的新的元素不是来自于任何其它 Mono 的元素，而是直接由代码返回的元素。

【Q-04】在 WebFlux 编程中有个很重要类 ServerWebExchange，请谈一下你对它们的认识。

【RA】ServerWebExchange 是 HTTP 请求-响应交互的约定类。提供对 HTTP 请求和响应的访问，还公开其他与服务器端处理相关的属性和特性，如请求属性等。通过这个实例，可以获取到请求、响应、属性等各种信息。这个类其实就相当于一个 Context 上下文，所以其非常

重要。

第 15 次直播课-11

【Q-01】什么是长轮询模型？请谈一下你的认识。
【RA】长轮询模型整合了 Push 与 Pull 模型的优势。Client 仍定时发起 Pull 请求，查看 Server 端数据是否更新。若发生了更新，则 Server 立即将更新数据以响应的形式发送给 Client 端；若没有发生更新，Server 端不会发送任何信息，但其会临时性的保持住这个连接一段时间。若在此时间段内，Server 端数据发生了变更，这个变更就会触发 Server 向 Client 发送变更结果。这次发送的执行，就是因为长连接的存在。若此期间仍未发生变更，则放弃这个连接。等待着下一次 Client 的 Pull 请求。

【Q-02】nacos config client 中是以异步线程池的方式向 server 发出的长轮询任务请求的。请问，这个异步执行线程的线程池核心线程数是多少？请谈一下你的认识。
【RA】nacos config client 中是以异步线程池的方式向 server 发出的长轮询任务请求的。为了保证执行效率，这个异步执行线程的线程池核心线程数是当前主机可用的逻辑内核数量。这样可以保证一个逻辑内核处理一个线程。

【Q-03】nacos config client 向 Server 发出的长轮询任务连接请求数量是怎样的？与配置文件数量是否有关？请谈一下你的认识。
【RA】nacos config client 向 Server 发出的长轮询任务连接请求数量是与可动态变更的配置文件数量有关的。默认情况下，每 3000 个配置文件会发出一个长连接请求。Server 端会对这个长连接中的 3000 个配置进行轮询检测其是否发生变更。

【Q-04】从 nacos config client 角度来看，其是如何做到配置文件自动更新的？请谈一下你的认识。
【RA】如果仅从 nacos config client 角度来说，其是这样做到配置文件的自动更新的。

在 Nacos Config Client 启动时会创建一个 NacosConfigService 实例，用于处理 NacosConfig 相关的操作。但实际上这些操作都是由 ClientWorker 实例在完成。在创建 NacosConfigService 实例时会创建一个 ClientWorker 实例。

在创建 ClientWorker 实例时，其会启动一个周期性执行的定时任务：从 Server 中获取到发生了变更的配置数据并更新到这些配置文件在本地的缓存 cacheMap 中。client 中 @RefreshScope 实例获取更新数据就是从 cacheMap 中的 value 中获取的。

【Q-05】nacos config client 中获取到的来自于 config server 的发生了变更的配置信息存放在哪里？请谈一下你的认识。
【RA】nacos config client 中获取到的来自于 config server 的发生了变更的配置信息存放在一个缓存 Map 集合 cacheMap 中。cacheMap 是个很重要缓存 Map 集合。其中存放着当前 client 中其所需要的每个配置文件对应的本地缓存 CacheData。在 config client 接收到来自于 config server 的变更数据后，会将这个变更内容更新到本地其对应的 CacheData 中。client 中 @RefreshScope 实例获取更新数据就是从 CacheData 中获取的。cacheMap 的 key 为配置文件的 key，即 dataId+groupId，value 则为该配置文件对应的 CacheData。

【Q-06】nacos config client 会为其每个配置文件创建一个 CacheData，用于存放来自 config server 的配置变更数据。CacheData 是何时创建的？请谈一下你的认识。

【RA】nacos config client 会为其每个配置文件创建一个 CacheData，用于存放来自 config server 的配置变更数据。这个 CacheData 的创建时机是 client 启动完毕。一旦 client 应用启动完毕，其就会触发对 CacheData 的创建。首先会遍历所有配置文件，为每个配置文件创建一个 CacheData，然后再为每个 CacheData 添加一个监听器。一旦监听到 CacheData 中的数据发生了变更，就会引发回调的执行。这个回调会发布一个刷新事件 RefreshEvent。RefreshEvent 事件的发布会触发所有被@RefreshScope 注解的类实例被重新创建并初始化，而这个初始化时用到的会更新数据就来自于 CacheData。

【Q-07】Spring Cloud 是如何完成可刷新 Bean 的管理的？请谈一下你的认识。

【RA】首先要清楚，在 Spring 容器中，对于 Bean 是分区域进行管理的，每个 scope 就是一个区域。例如，我们熟悉的 singleton、prototype 等。在 spring cloud 中，又新增了一个自动刷新的 scope 区域，refresh。

对于可刷新 Bean 的管理，Spring Cloud 首先是将 Spring 容器中 refresh 区域的所有 Bean 全部清除掉。然后再在使用这些 Bean 时重新创建并初始化这些 Bean。而初始化时用到的数据就来最新的数据。

【Q-08】nacos config server 是如何处理 config client 提交的“配置更新检测”请求的？请谈一下你的认识。

【RA】nacos config server 在接收到 client 的请求后，首先从请求中获取到 client 所要检测的配置文件 key，然后对它些配置文件立即进行配置变更检测。如果有配置文件发生了变更，则立即将这些发生了变更的配置文件 key 以 response 的方式发送给 client。若没有检测到变更，则为这个发送请求的 client 创建一个长轮询客户端实例，在这些客户端实例中定义并异步执行了一个 29.5s 的定时任务。该任务是再次查找发生变更的配置文件 key，并将这些 key 以 response 的方式发送给 client。

【Q-09】nacos config server 是如何判断配置文件是否发生变更的呢？请谈一下你的认识。

【RA】nacos config server 是对于配置文件是否发生变更的判断，其实就是把来自于 client 的配置文件 key 的 md5 与 server 端这些配置文件的 md5 进行对比，相等则未变更，不同则发生变更。

【Q-10】nacos config server 创建的长轮询客户端实例，是为每个 config client 创建一个该实例吗？请谈一下你的认识。

【RA】不是的，对于 config client 所提交的每个“配置更新检测请求”，server 会为其创建一个长轮询客户端。而这一次请求中，最多检测 3000 个配置文件的更新情况。

【Q-11】nacos config server 对于非固定时长长轮询发生了配置变更事件，是如何通知到其对应的 client 的？请谈一下你的认识。

【RA】Nacos Config Server 在启动时会创建 LongPollingService 实例。该实例用于处理长轮询相关的操作。

在创建 LongPollingService 实例时，会注册一个 LocalDataChangeEvent 的订阅者。一旦 Server 中的某个配置文件发生变更，就会触发订阅者回调方法的执行。而回调方法则会引发 DataChangeTask 的异步执行。DataChangeTask 任务就是从当前 server 所 hold 的所有长轮询

客户端集合 allSubs 中查找到底是哪个长轮询客户端的这个配置文件发生了变更，然后将这个发生变更的配置文件 key 发送给这个 client。