主讲老师： Fox老师

课前须知：

1. 中途加入的同学建议从第一节课开始学习，如果没有微服务基础的同学建议先学习spring cloud alibaba基础课程。

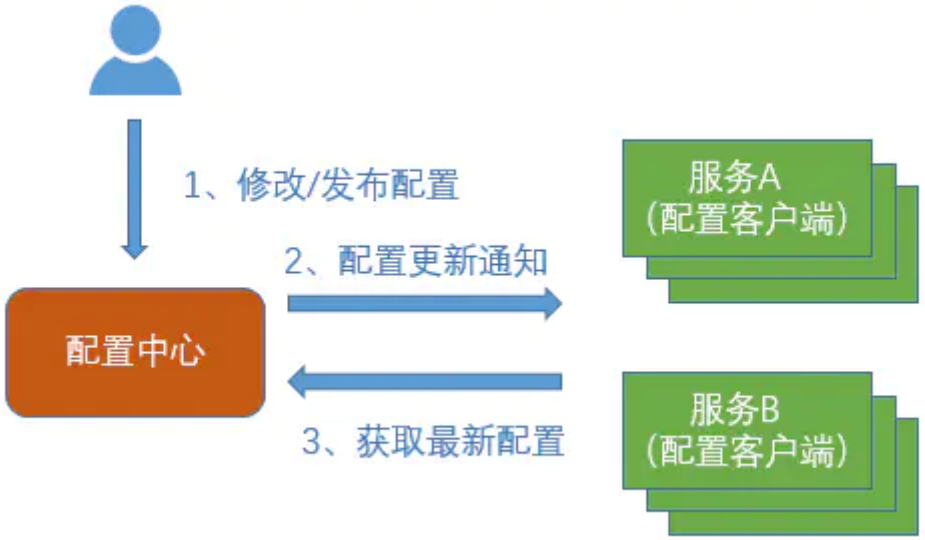
2. 微服务组件的源码分析过程中会涉及到Spring,SpringBoot相关的源码（大量核心扩展点运用），这块不熟悉的同学可以课后去补一补Spring,SpringBoot的源码课

3. nacos版本： v 2.1.0

```
文档：06. Alibaba微服务组件Nacos配置中心实...
链接：http://note.youdao.com/noteshare?id=968f9f6e62248d986dd9ca9c8d9b61cf&sub=4BCE4A117E364A91AF5AEF7A569A4127
```

1. 配置中心

在微服务架构中，当系统从一个单体应用，被拆分成分布式系统上一个个服务节点后，配置文件也必须跟着迁移（分割），这样配置就分散了，不仅如此，分散中还包含着冗余。配置中心将配置从各应用中剥离出来，对配置进行统一管理，应用自身不需要自己去管理配置。



配置中心的服务流程如下：

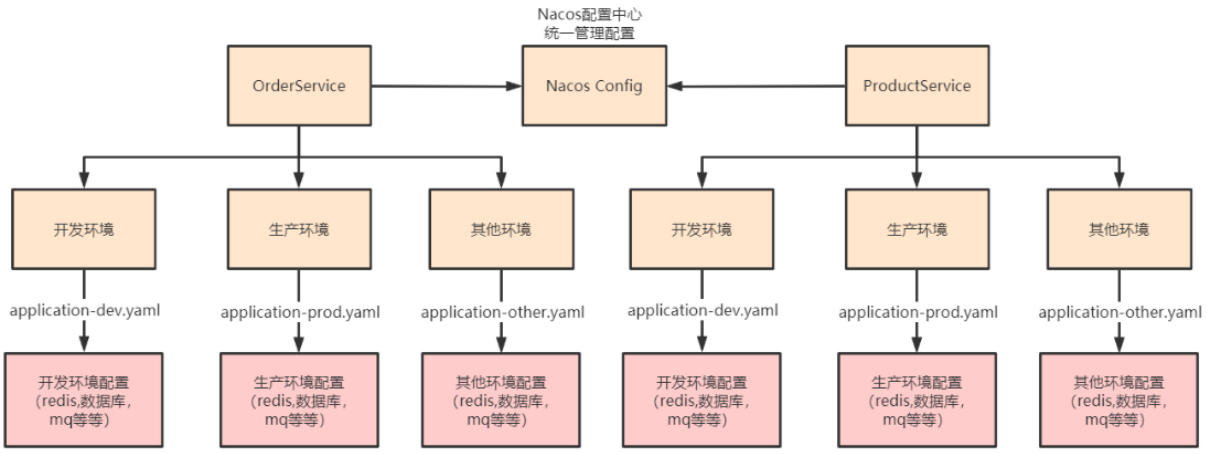
1. 用户在配置中心更新配置信息。
2. 服务A和服务B及时得到配置更新通知，从配置中心获取配置。

配置中心就是一种统一管理各种应用配置的基础服务组件。

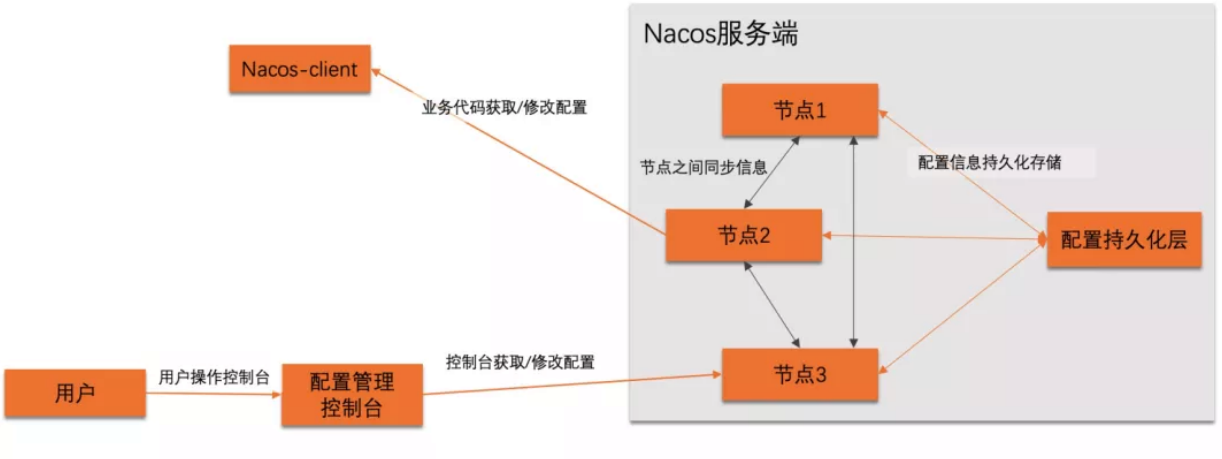
1.1 什么是Nacos配置中心

官方文档： https://github.com/alibaba/spring-cloud-alibaba/wiki/Nacos-config

Nacos 提供用于存储配置和其他元数据的 key/value 存储，为分布式系统中的外部化配置提供服务器端和客户端支持。使用 Spring Cloud Alibaba Nacos Config，您可以在 Nacos Server 集中管理你 Spring Cloud 应用的外部属性配置。



1.2 Nacos配置中心的架构



1.2.1 Nacos配置中心核心API

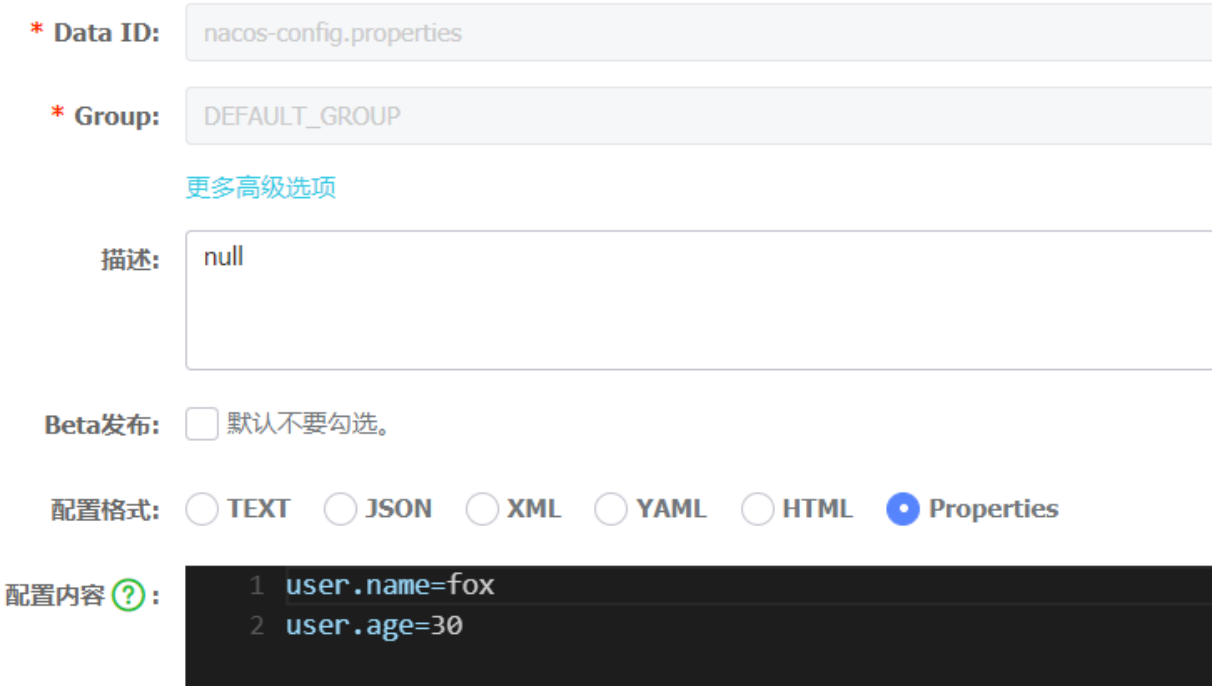
```
1  public class ConfigServerDemo {
2
3   public static void main(String[] args) throws NacosException, InterruptedException {
4   String serverAddr = "localhost";
5   String dataId = "nacos-config-demo.yaml";
6   String group = "DEFAULT_GROUP";
7   Properties properties = new Properties();
8   properties.put(PropertyKeyConst.SERVER_ADDR, serverAddr);
9   //获取配置服务
10   ConfigService configService = NacosFactory.createConfigService(properties);
11   //获取配置
12   String content = configService.getConfig(dataId, group, 5000);
```

```
13   System.out.println(content);
14   //注册监听器
15   configService.addListener(dataId, group, new Listener() {
16   @Override
17   public void receiveConfigInfo(String configInfo) {
18   System.out.println("===recieve:" + configInfo);
19   }
20
21   @Override
22   public Executor getExecutor() {
23   return null;
24   }
25   });
26
27   //发布配置
28   //boolean isPublishOk = configService.publishConfig(dataId, group, "con
tent");
29   //System.out.println(isPublishOk);
30   //发送properties格式
31   configService.publishConfig(dataId,group,"common.age=30", ConfigType.PR
OPERTIES.getType());
32
33   Thread.sleep(3000);
34   content = configService.getConfig(dataId, group, 5000);
35   System.out.println(content);
36
37  // boolean isRemoveOk = configService.removeConfig(dataId, group);
38  // System.out.println(isRemoveOk);
39  // Thread.sleep(3000);
40
41  // content = configService.getConfig(dataId, group, 5000);
42  // System.out.println(content);
43  // Thread.sleep(300000);
44
45   }
46  }
```

1.3 Spring Cloud 整合Nacos配置中心快速开始

1.3.1 nacos server配置中心中准备配置数据

nacos server中新建nacos-config.properties



1.3.2 微服务接入配置中心

1）引入依赖

```
<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
```

2）添加bootstrap.properties

注意：必须使用 bootstrap.properties 配置文件来配置Nacos Server 地址

```
spring.application.name=nacos-config
# 配置中心地址
spring.cloud.nacos.config.server-addr=127.0.0.1:8848

# dataid 为 yaml 的文件扩展名配置方式
spring.cloud.nacos.config.file-extension=yaml
#profile粒度的配置 `${spring.application.name}-${profile}.${file-extension:properties}`
spring.profiles.active=prod
```

在 Nacos Spring Cloud 中，dataId 的完整格式如下：

${prefix}-${spring.profiles.active}.${file-extension}

- prefix 默认为 spring.application.name 的值，也可以通过配置项 spring.cloud.nacos.config.prefix来配置。
- spring.profiles.active 即为当前环境对应的 profile，详情可以参考 Spring Boot文档。 注意：当 spring.profiles.active 为空时，对应的连接符 - 也将不存在，dataId 的拼接格式变成 ${prefix}.${file-extension}
- file-exetension 为配置内容的数据格式，可以通过配置项 spring.cloud.nacos.config.file-extension 来配置。

3) 启动服务，测试微服务是否使用配置中心的配置

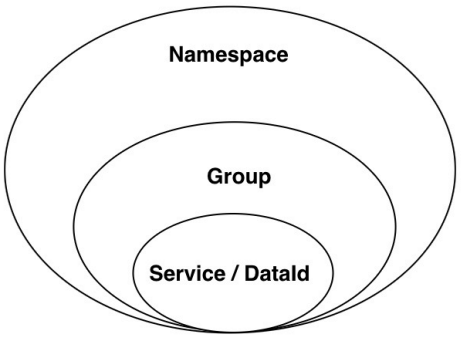
```
1  @SpringBootApplication
2  public class NacosConfigApplication {
3
4   public static void main(String[] args) {
5   ConfigurableApplicationContext applicationContext = SpringApplication.ru
n(NacosConfigApplication.class, args);
6   String userName = applicationContext.getEnvironment().getProperty("commo
n.name");
7   String userAge = applicationContext.getEnvironment().getProperty("commo
n.age");
8   System.out.println("common name :"+userName+"; age: "+userAge);
9   }
10  }
```

```
2020-07-31 15:33:43.474  WARN 45352 --- [           main] c.a.c.n.c.NacosPropertySourceBuilder     : Ignore the empty
2020-07-31 15:33:43.479  INFO 45352 --- [           main] c.a.c.n.c.NacosPropertySourceBuilder     : Loading nacos da
user.age=30
2020-07-31 15:33:43.481  INFO 45352 --- [           main] b.c.PropertySourceBootstrapConfiguration : Located property
2020-07-31 15:33:43.484  INFO 45352 --- [           main] bat.ke.qq.com.NacosConfigApplication     : No active profil
2020-07-31 15:33:43.634  INFO 45352 --- [           main] o.s.cloud.context.scope.GenericScope     : BeanFactory id=6
2020-07-31 15:33:43.638  INFO 45352 --- [           main] trationDelegate$BeanPostProcessorChecker : Bean 'org.spring
2020-07-31 15:33:45.202  INFO 45352 --- [           main] o.s.cloud.commons.util.InetUtils         : Cannot determine
2020-07-31 15:33:45.243  INFO 45352 --- [           main] bat.ke.qq.com.NacosConfigApplication     : Started NacosCon
user name :fox; age: 30
```

1.4 Config相关配置

Nacos 数据模型 Key 由三元组唯一确定, Namespace默认是空串，公共命名空间（public），分组默认是 DEFAULT_GROUP

Nacos data model



- **支持配置的动态更新**

```
@SpringBootApplication
public class NacosConfigApplication {

 public static void main(String[] args) throws InterruptedException {
 ConfigurableApplicationContext applicationContext = SpringApplication.run(NacosConfigApplication.class, args);

 while(true) {
 //当动态配置刷新时，会更新到 Enviroment中，因此这里每隔一秒中从Enviroment中获取配置
 String userName = applicationContext.getEnvironment().getProperty("common.name");
 String userAge = applicationContext.getEnvironment().getProperty("common.age");
 System.err.println("common name :" + userName + "; age: " + userAge);
 TimeUnit.SECONDS.sleep(1);
 }

 }

}

```

- **支持profile粒度的配置**

spring-cloud-starter-alibaba-nacos-config 在加载配置的时候，不仅仅加载了以 dataid 为 ${spring.application.name}.${file-extension:properties} 为前缀的基础配置，还加载了dataid为 ${spring.application.name}-${profile}.${file-extension:properties} 的基础配置。在日常开发中如果遇到多套环境下的不同配置，可以通过Spring 提供的 ${spring.profiles.active} 这个配置项来配置。

```
spring.profiles.active=dev
```

- **支持自定义 namespace 的配置**

用于进行租户粒度的配置隔离。不同的命名空间下，可以存在相同的 Group 或 Data ID 的配置。Namespace 的常用场景之一是不同环境的配置的区分隔离，例如开发测试环境和生产环境的资源（如配置、服务）隔离等。

在没有明确指定 ${spring.cloud.nacos.config.namespace} 配置的情况下， 默认使用的是 Nacos 上 Public 这个namespace。如果需要使用自定义的命名空间，可以通过以下配置来实现：

```
spring.cloud.nacos.config.namespace=71bb9785-231f-4eca-b4dc-6be446e12ff8
```

- **支持自定义 Group 的配置**

Group是组织配置的维度之一。通过一个有意义的字符串（如 Buy 或 Trade ）对配置集进行分组，从而区分 Data ID 相同的配置集。当您在 Nacos 上创建一个配置时，如果未填写配置分组的名称，则配置分组的名称默认采用 DEFAULT_GROUP 。配置分组的常见场景：不同的应用或组件使用了相同的配置类型，如 database_url 配置和 MQ_topic 配置。

在没有明确指定 ${spring.cloud.nacos.config.group} 配置的情况下，默认是 DEFAULT_GROUP 。如果需要自定义自己的 Group，可以通过以下配置来实现：

```
spring.cloud.nacos.config.group=DEVELOP_GROUP
```

- **支持自定义扩展的 Data Id 配置**

Data ID 是组织划分配置的维度之一。Data ID 通常用于组织划分系统的配置集。一个系统或者应用可以包含多个配置集，每个配置集都可以被一个有意义的名称标识。Data ID 通常采用类 Java 包（如 com.taobao.tc.refund.log.level）的命名规则保证全局唯一性。此命名规则非强制。

通过自定义扩展的 Data Id 配置，既可以解决多个应用间配置共享的问题，又可以支持一个应用有多个配置文件。

```
# 自定义 Data Id 的配置
#不同工程的通用配置 支持共享的 DataId
spring.cloud.nacos.config.sharedConfigs[0].data-id= common.yaml
spring.cloud.nacos.config.sharedConfigs[0].group=REFRESH_GROUP
spring.cloud.nacos.config.sharedConfigs[0].refresh=true

# config external configuration
# 支持一个应用多个 DataId 的配置
spring.cloud.nacos.config.extensionConfigs[0].data-id=ext-config-common01.properties
spring.cloud.nacos.config.extensionConfigs[0].group=REFRESH_GROUP
spring.cloud.nacos.config.extensionConfigs[0].refresh=true

spring.cloud.nacos.config.extensionConfigs[1].data-id=ext-config-common02.properties
spring.cloud.nacos.config.extensionConfigs[1].group=REFRESH_GROUP
```

```
spring.cloud.nacos.config.extensionConfigs[1].refresh=true
```

1.5 配置的优先级

Spring Cloud Alibaba Nacos Config 目前提供了三种配置能力从 Nacos 拉取相关的配置。

- A: 通过 spring.cloud.nacos.config.shared-configs 支持多个共享 Data Id 的配置
- B: 通过 spring.cloud.nacos.config.ext-config[n].data-id 的方式支持多个扩展 Data Id 的配置
- C: 通过内部相关规则(应用名、应用名+ Profile)自动生成相关的 Data Id 配置

当三种方式共同使用时，他们的一个优先级关系是:A < B < C

优先级从高到低：

- ${spring.application.name}-${profile}.${file-extension:properties}
- ${spring.application.name}.${file-extension:properties}
- ${spring.application.name}
- extensionConfigs　一个微服务的多个配置，比如 nacos,mybatis
- sharedConfigs　多个微服务公共配置，比如 redis

1.6 @RefreshScope实现动态感知

@Value注解可以获取到配置中心的值，但是无法动态感知修改后的值，需要利用@RefreshScope注解

```
@RestController
@RefreshScope
public class TestController {

 @Value("${common.age}")
 private String age;

 @GetMapping("/common")
 public String hello() {
  return age;
  }

}
```

@RefreshScope 导致@Scheduled定时任务失效问题

当利用@RefreshScope刷新配置后会导致定时任务失效

```
@SpringBootApplication
@EnableScheduling // 开启定时任务功能
public class NacosConfigApplication {
}

@RestController
@RefreshScope //动态感知修改后的值
public class TestController {

  @Value("${common.age}")
  String age;
  @Value("${common.name}")
  String name;

  @GetMapping("/common")
  public String hello() {
  return name+","+age;
  }

  //触发@RefreshScope执行逻辑会导致@Scheduled定时任务失效
  @Scheduled(cron = "*/3 * * * * ?") //定时任务每隔3s执行一次
  public void execute() {
  System.out.println("定时任务正常执行。。。。。。");
  }


}
```

测试结果：

- 当在配置中心变更属性后，定时任务失效
- 当再次访问http://localhost:8010/common，定时任务生效

原因：@RefreshScope修饰的bean的属性发生变更后，会从缓存中清除。此时没有这个bean，定时任务当然也就不生效了。

详细原因如下：

1. @RefreshScope 注解标注了@Scope 注解，并默认了ScopedProxyMode.TARGET_CLASS属性，此属性的功能就是创建一个代理，在每次调用的时候都用它来调用GenericScope#get 方法来获取bean对象。

2. 在GenericScope 里面包装了一个内部类 BeanLifecycleWrapperCache 来对加了@RefreshScope 的bean进行缓存，使其在不刷新时获取的都是同一个对象。

3. 如属性发生变更会调用 ContextRefresher#refresh()——>RefreshScope#refreshAll() 进行缓存清理方法调用，并发送刷新事件通知 ——> 调用GenericScope#destroy() 实现清理缓存

4. 当下一次使用此bean对象时，代理对象会调用GenericScope#get(String name, ObjectFactory<?> objectFactory) 方法创建一个新的bean对象，并存入缓存中，此时新对象因为Spring 的装配机制就是新的属性了

后面会结合源码分析，核心源码：GenericScope#get

解决方案

实现Spring事件监听器，监听 RefreshScopeRefreshedEvent事件，监听方法中进行一次定时方法的调用

```
@RestController
@RefreshScope //动态感知修改后的值
public class TestController implements ApplicationListener<RefreshScopeRefreshedEvent>{

 @Value("${common.age}")
 String age;
 @Value("${common.name}")
 String name;

 @GetMapping("/common")
 public String hello() {
 return name+","+age;
 }

 //触发@RefreshScope执行逻辑会导致@Scheduled定时任务失效
 @Scheduled(cron = "*/3 * * * * ?") //定时任务每隔3s执行一次
 public void execute() {
 System.out.println("定时任务正常执行。。。。。。");
 }


 @Override
 public void onApplicationEvent(RefreshScopeRefreshedEvent event) {
 this.execute();
 }
```

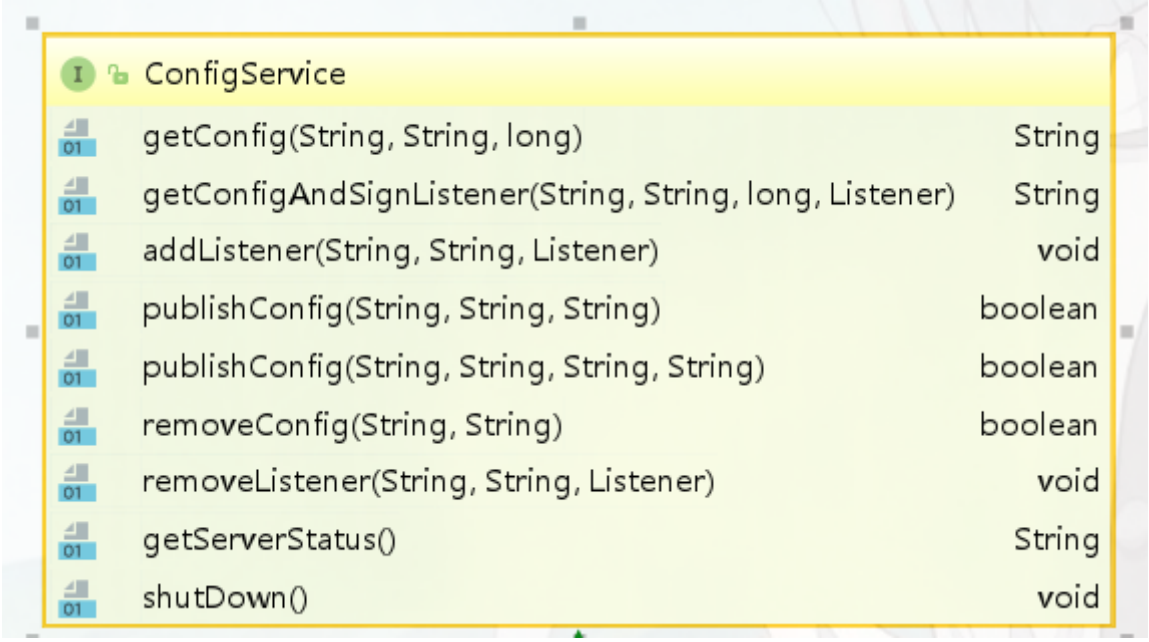
```
26 }
```

2. Nacos配置中心源码分析

https://www.processon.com/view/link/62d678c31e08531cf8db16ef

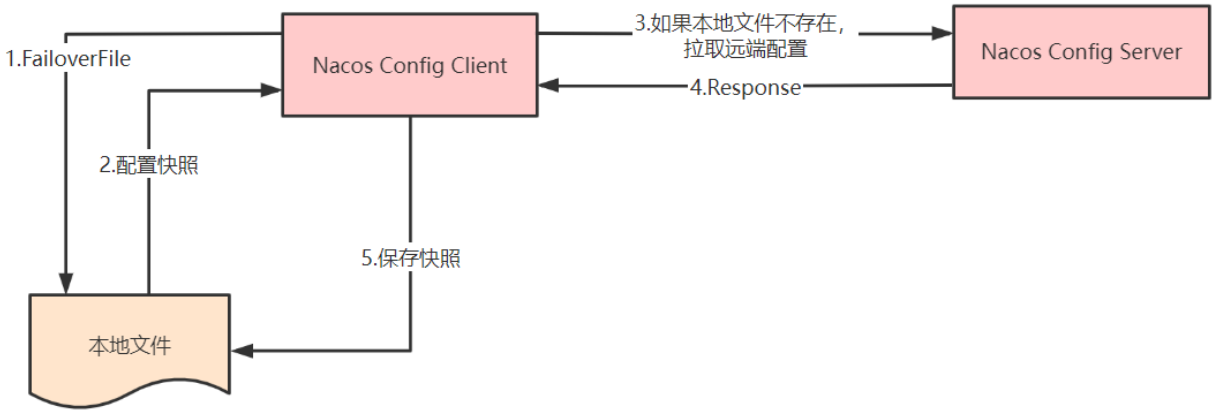
2.1 nacos config client源码分析

配置中心核心接口ConfigService



获取配置

获取配置的主要方法是 NacosConfigService 类的 getConfig 方法，通常情况下该方法直接从本地文件中取得配置的值，如果本地文件不存在或者内容为空，则再通过grpc从远端拉取配置，并保存到本地快照中。



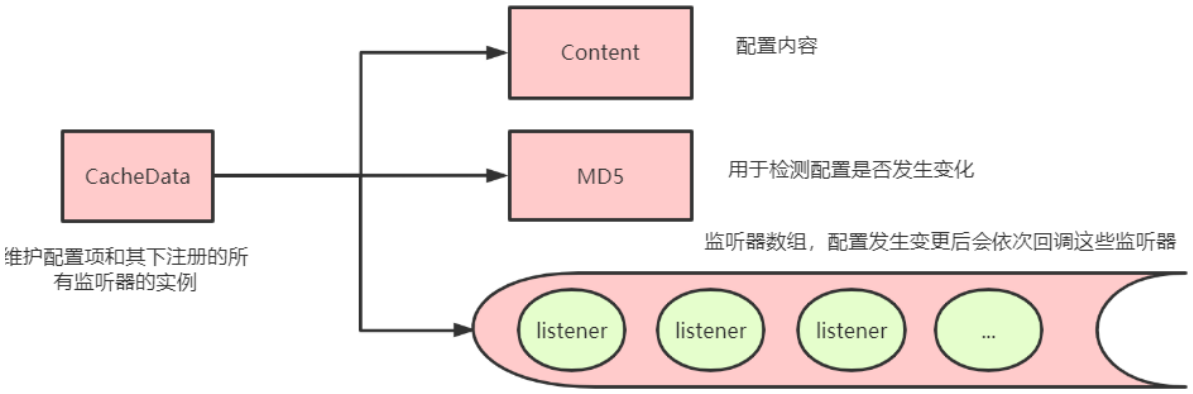
注册监听器

配置中心客户端会通过对配置项注册监听器达到在配置项变更的时候执行回调的功能。

```
1 ConfigService#getConfigAndSignListener
```

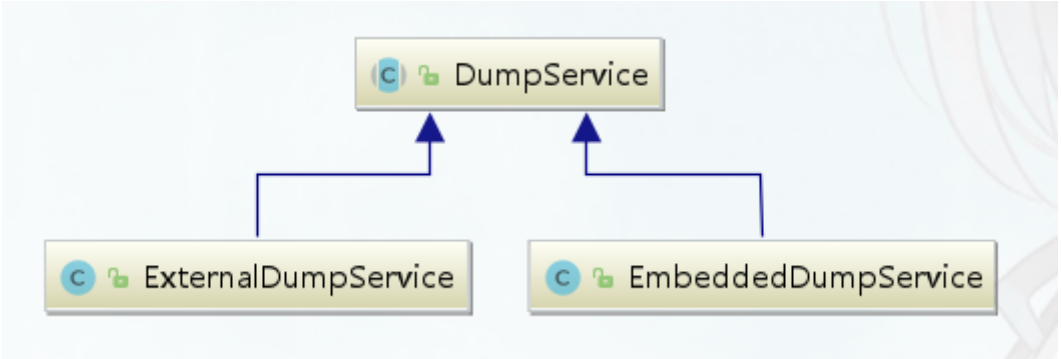
```
2 ConfigService#addListener
```

Nacos 可以通过以上方式注册监听器，它们内部的实现均是调用 ClientWorker 类的 addCacheDataIfAbsent。其中 CacheData 是一个维护配置项和其下注册的所有监听器的实例，所有的 CacheData 都保存在 ClientWorker 类中的原子 cacheMap 中，其内部的核心成员有：



2.2 nacos config server源码分析

配置dump



服务端启动时就会依赖 DumpService 的 init 方法，从数据库中 load 配置存储在本地磁盘上，并将一些重要的元信息例如 MD5 值缓存在内存中。服务端会根据心跳文件中保存的最后一次心跳时间，来判断到底是从数据库 dump 全量配置数据还是部分增量配置数据（如果机器上次心跳间隔是 6h 以内的话）。

全量 dump 当然先清空磁盘缓存，然后根据主键 ID 每次捞取一千条配置刷进磁盘和内存。增量 dump 就是捞取最近六小时的新增配置（包括更新的和删除的），先按照这批数据刷新一遍内存和文件，再根据内存里所有的数据全量去比对一遍数据库，如果有改变的再同步一次，相比于全量 dump 的话会减少一定的数据库 IO 和磁盘 IO 次数。

配置发布

发布配置的代码位于 ConfigController#publishConfig中。集群部署，请求一开始也只会打到一台机器，这台机器将配置插入Mysql中进行持久化。服务端并不是针对每次配置查询都去访问 MySQL ，而是会依赖 dump 功能在本地文件中将配置缓存起来。因此当单

台机器保存完毕配置之后，需要通知其他机器刷新内存和本地磁盘中的文件内容，因此它会发布一个名为 ConfigDataChangeEvent 的事件，这个事件会通过grpc调用通知所有集群节点（包括自身），触发本地文件和内存的刷新。