nacos源码分析

1. nacos源码工程搭建

1.1 环境准备

在nacos的官网介绍中，nacos源码运行，需要的java运行环境有：

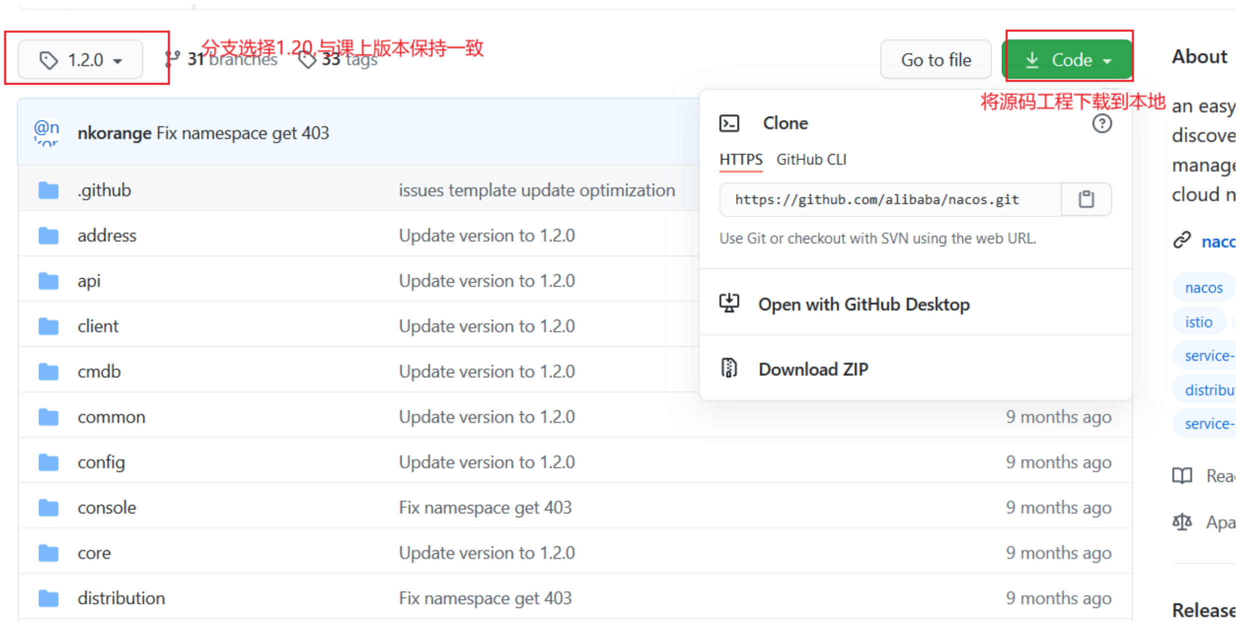
- JDK 1.8+
- Maven 3.2+

1.2 源码构建

1.2.1 源码下载

从github上，下载nacos的源码到本地；

```
https://github.com/alibaba/nacos
```



nkorange Fix namespace get 403

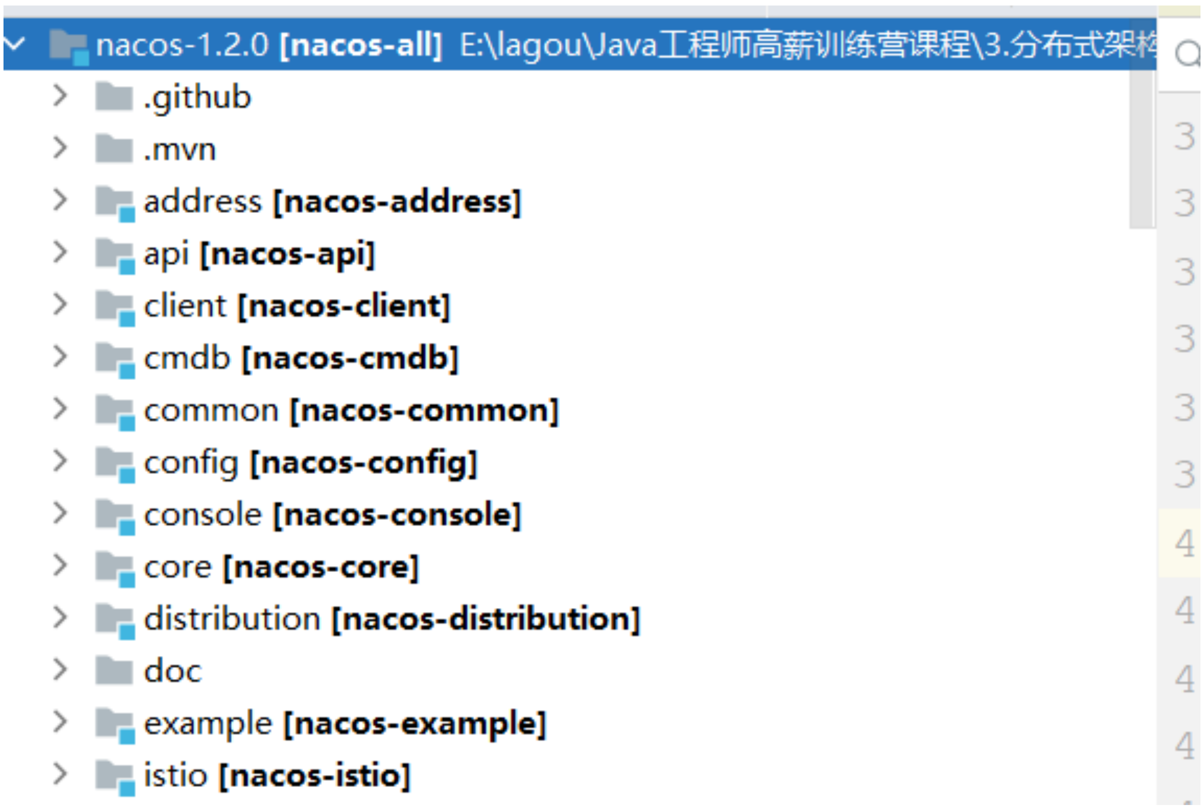
https://github.com/alibaba/nacos.git

Use Git or checkout with SVN using the web URL.

.github	issues template update optimization	
address	Update version to 1.2.0	
api	Update version to 1.2.0	
client	Update version to 1.2.0	
cmdb	Update version to 1.2.0	
common	Update version to 1.2.0	9 months ago
config	Update version to 1.2.0	9 months ago
console	Fix namespace get 403	9 months ago
core	Update version to 1.2.0	9 months ago
distribution	Fix namespace get 403	9 months ago

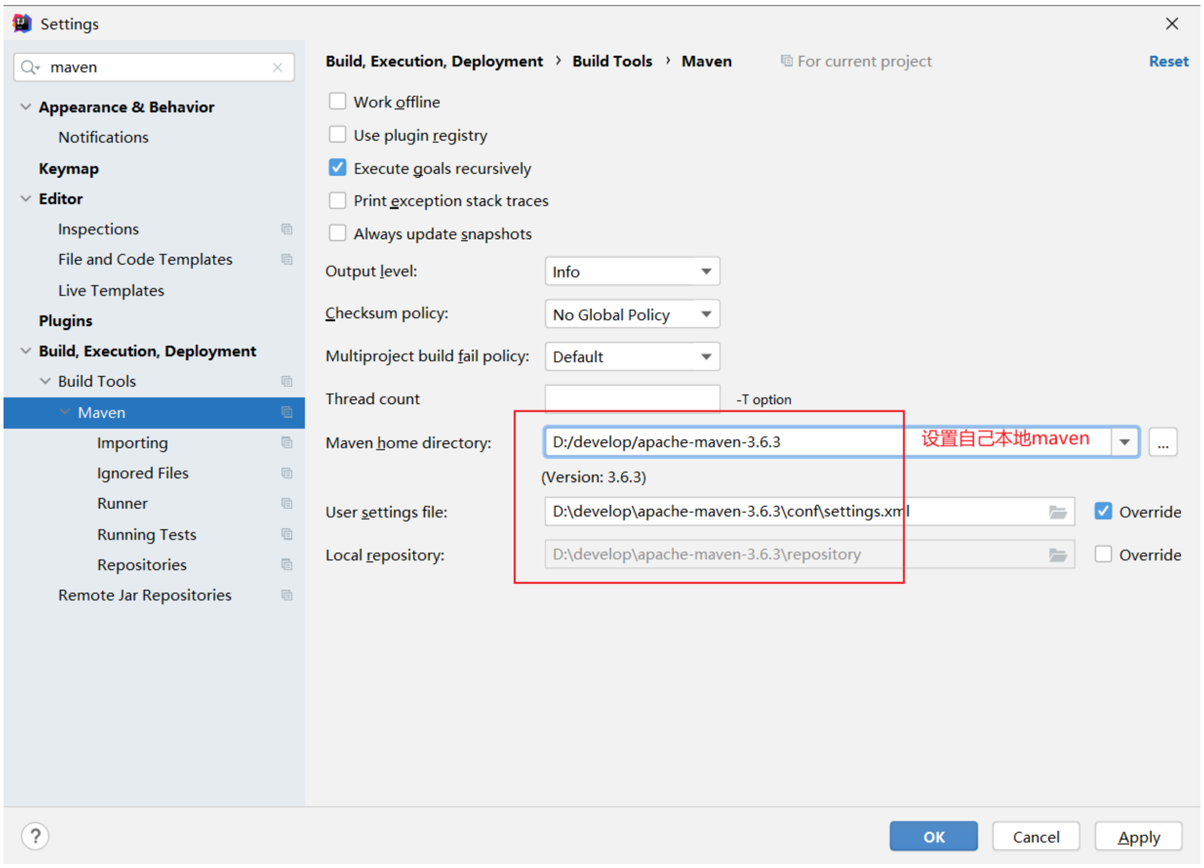
1.2.2 导入idea工程

1. 导入



nacos-1.2.0 [nacos-all] E:\lagou\Java工程师高薪训练营课程\3.分布式架构
.github
.mvn
address [nacos-address]
api [nacos-api]
client [nacos-client]
cmdb [nacos-cmdb]
common [nacos-common]
config [nacos-config]
console [nacos-console]
core [nacos-core]
distribution [nacos-distribution]
doc
example [nacos-example]
istio [nacos-istio]

2. 配置maven环境,下载jar包如果是阿里云大约在5分钟左右



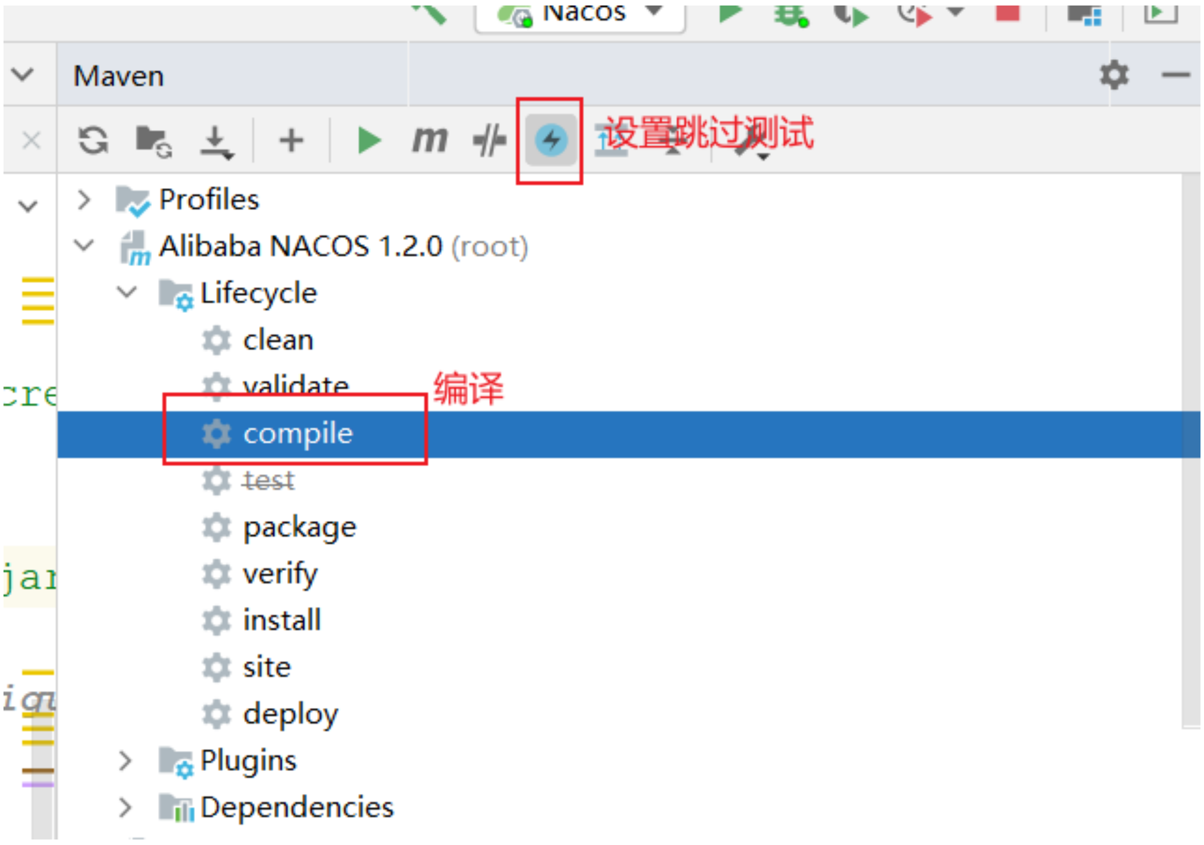
Settings

Build, Execution, Deployment › Build Tools › Maven

- [ ] Work offline
- [ ] Use plugin registry
- [x] Execute goals recursively
- [ ] Print exception stack traces
- [ ] Always update snapshots

Output level: Info
Checksum policy: No Global Policy
Multiproject build fail policy: Default
Thread count: -T option
Maven home directory: D:/develop/apache-maven-3.6.3 设置自己本地maven
(Version: 3.6.3)
User settings file: D:\develop\apache-maven-3.6.3\conf\settings.xml [✓ Override]
Local repository: D:\develop\apache-maven-3.6.3\repository [ ] Override

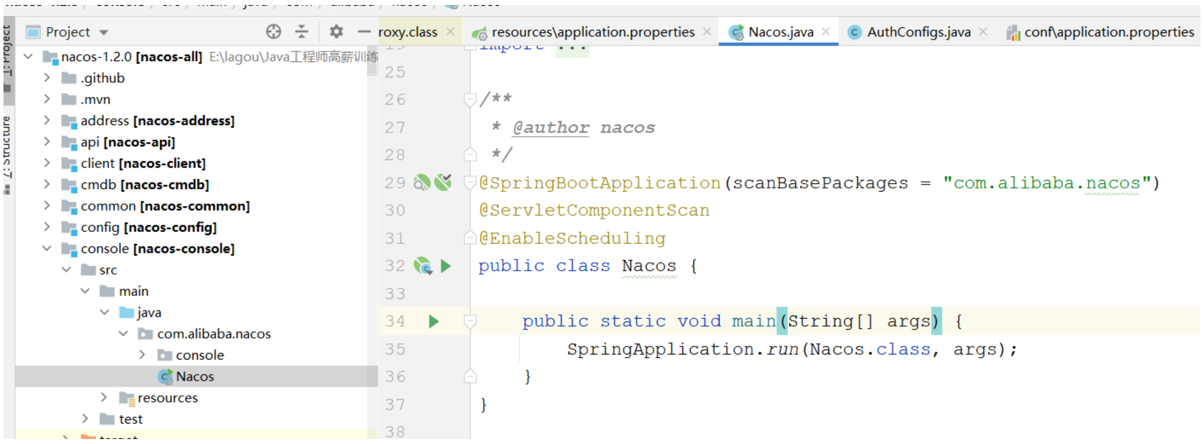
3. 编译工程



1.2.3 源码运行

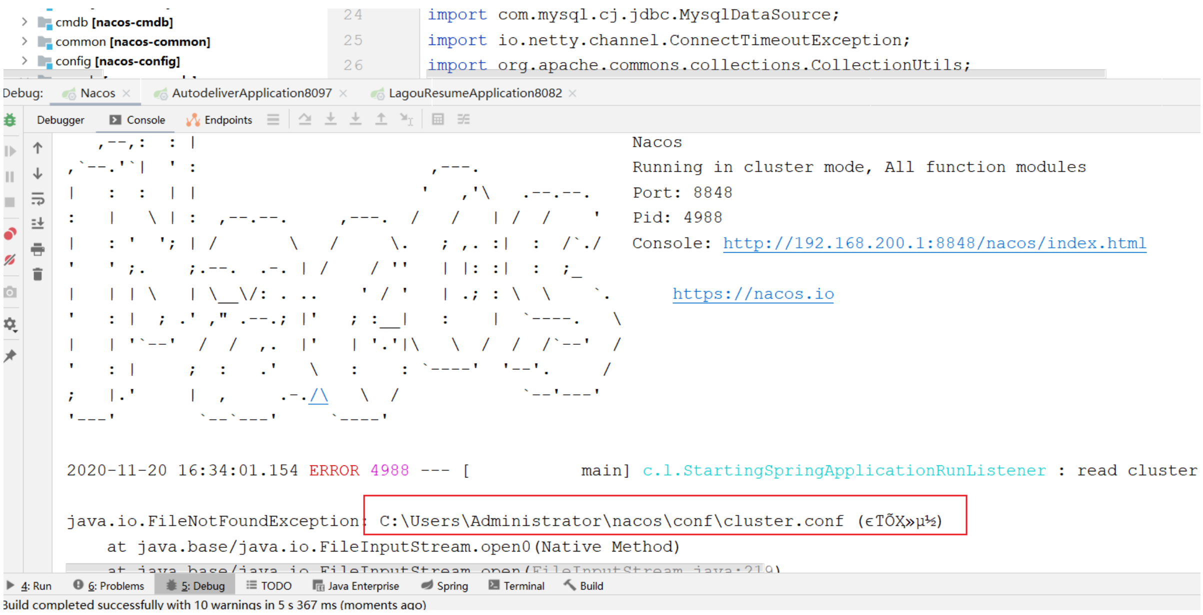
1. 工程启动

   进入到nacos-console模块下，启动该模块下的com.alibaba.nacos.Nacos类。



```
/**
 * @author nacos
 */
@SpringBootApplication(scanBasePackages = "com.alibaba.nacos")
@ServletComponentScan
@EnableScheduling
public class Nacos {

    public static void main(String[] args) {
        SpringApplication.run(Nacos.class, args);
    }
}
```

但通常情况下，会报如下错误：



```
import com.mysql.cj.jdbc.MysqlDataSource;
import io.netty.channel.ConnectTimeoutException;
import org.apache.commons.collections.CollectionUtils;
```

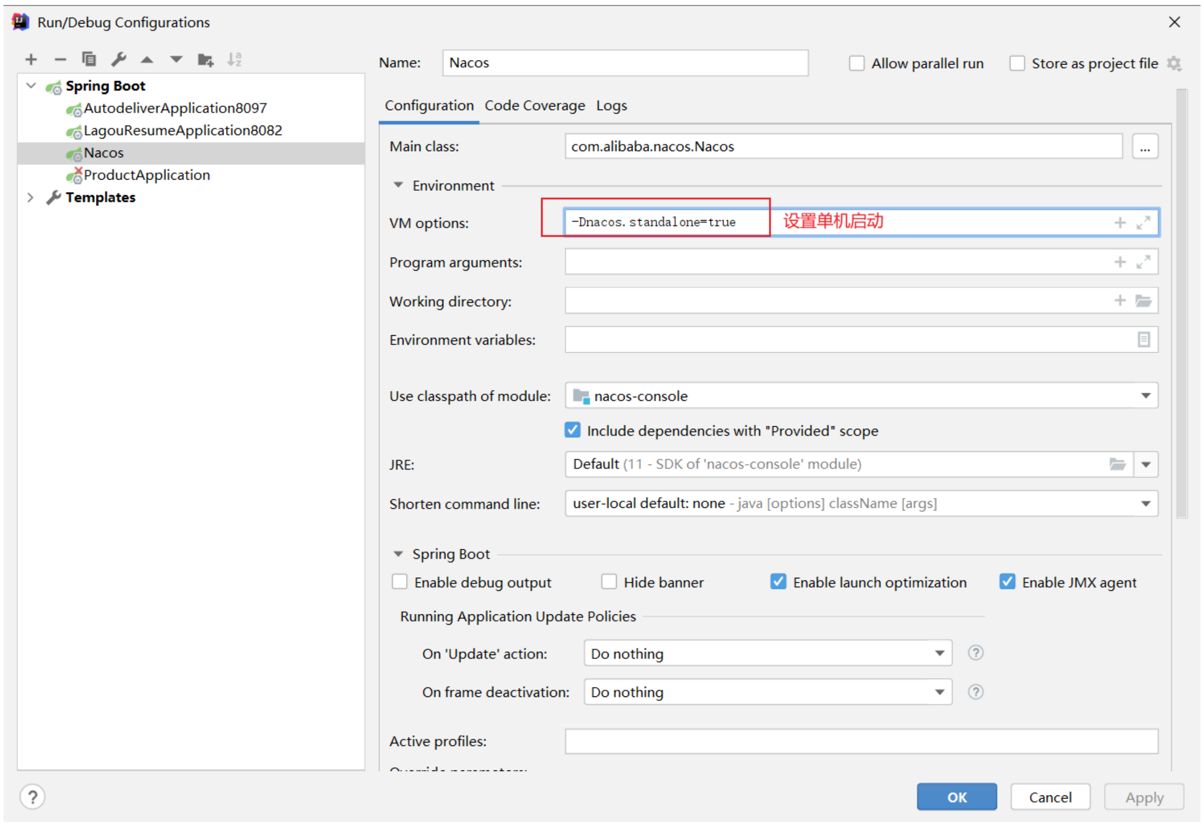
```
       ,--,:  : |                                           Nacos
 ,`--.'`|  ' :                       ,---.                  Running in cluster mode, All function modules
 |   :  :  | |                      '   ,'\   .--.--.       Port: 8848
 :   |   \ | :  ,--.--.     ,---.  /   /   | /  /    '      Pid: 4988
 |   : '  '; | /       \   /     \.   ; ,. :|  :  /`./      Console: http://192.168.200.1:8848/nacos/index.html
 '   ' ;.    ;.--.  .-. | /    / ''   | |: :|  :  ;_
 |   | | \   | \__\/: . ..    ' / '   | .; : \  \    `.          https://nacos.io
 '   : |  ; .' ," .--.; |'   ; :__|   :    |  `----.   \
 |   | '`--'  /  /  ,.  |'   | '.'|\   \  /  /  /`--'  /
 '   : |     ;  :   .'   \   :    : `----'  '--'.     /
 ;   |.'     |  ,     .-./\   \  /            `--'---'
 '---'        `--`---'     `----'

2020-11-20 16:34:01.154 ERROR 4988 --- [           main] c.l.StartingSpringApplicationRunListener : read cluster

java.io.FileNotFoundException: C:\Users\Administrator\nacos\conf\cluster.conf (ϵTÕX»µ½)
	at java.base/java.io.FileInputStream.open0(Native Method)
	at java.base/java.io.FileInputStream.open(FileInputStream.java:219)
```

Build completed successfully with 10 warnings in 5 s 367 ms (moments ago)

这是由于nacos默认使用的是集群方式，启动时会到默认的配置路径下，寻找集群配置文件cluster.conf。

我们源码运行时，通常使用的是单机模式，因此需要在启动参数中进行设置，在jvm的启动参数中，添加

```
-Dnacos.standalone=true
```

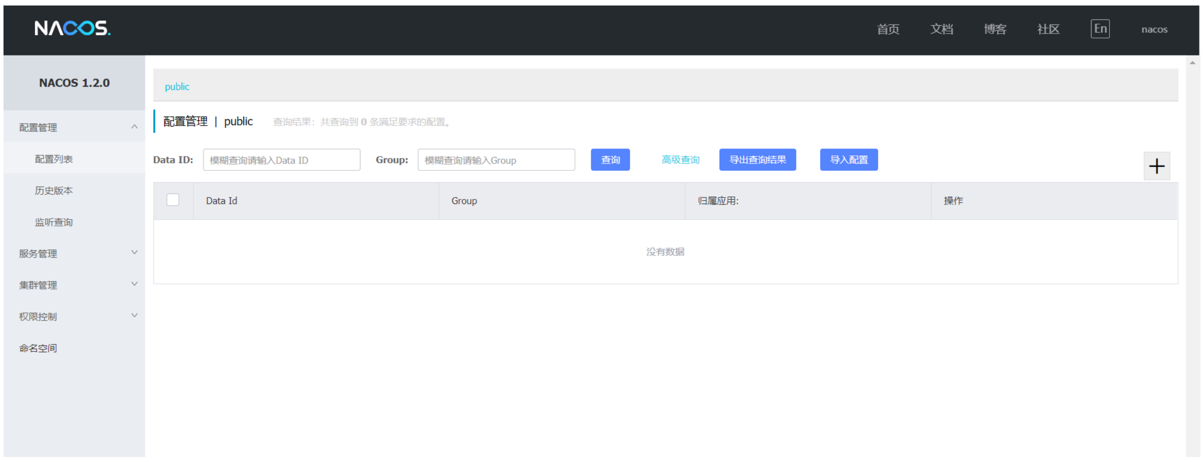


Run/Debug Configurations

Spring Boot
- AutodeliverApplication8097
- LagouResumeApplication8082
- Nacos
- ProductApplication

Templates

Name:	Nacos
Main class:	com.alibaba.nacos.Nacos
VM options:	-Dnacos.standalone=true
Program arguments:	
Working directory:	
Environment variables:	
Use classpath of module:	nacos-console
	✓ Include dependencies with "Provided" scope
JRE:	Default (11 - SDK of 'nacos-console' module)
Shorten command line:	user-local default: none - java [options] className [args]
On 'Update' action:	Do nothing
On frame deactivation:	Do nothing
Active profiles:	



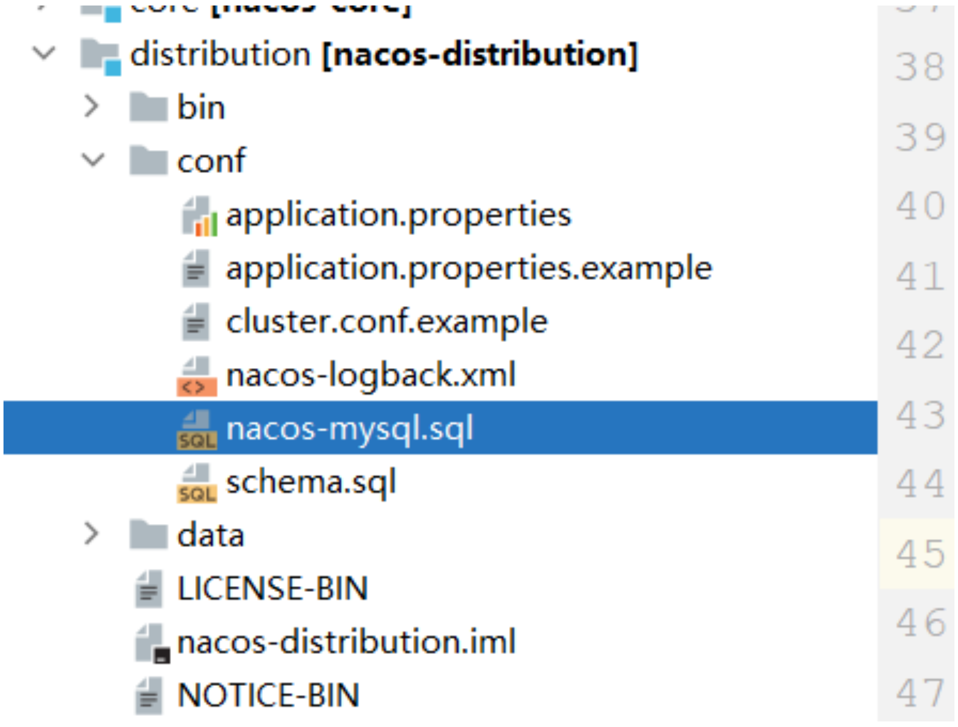
查询结果：共查询到 0 条满足要求的配置。

2. 配置数据库

修改console模块中的配置文件application.properties文件

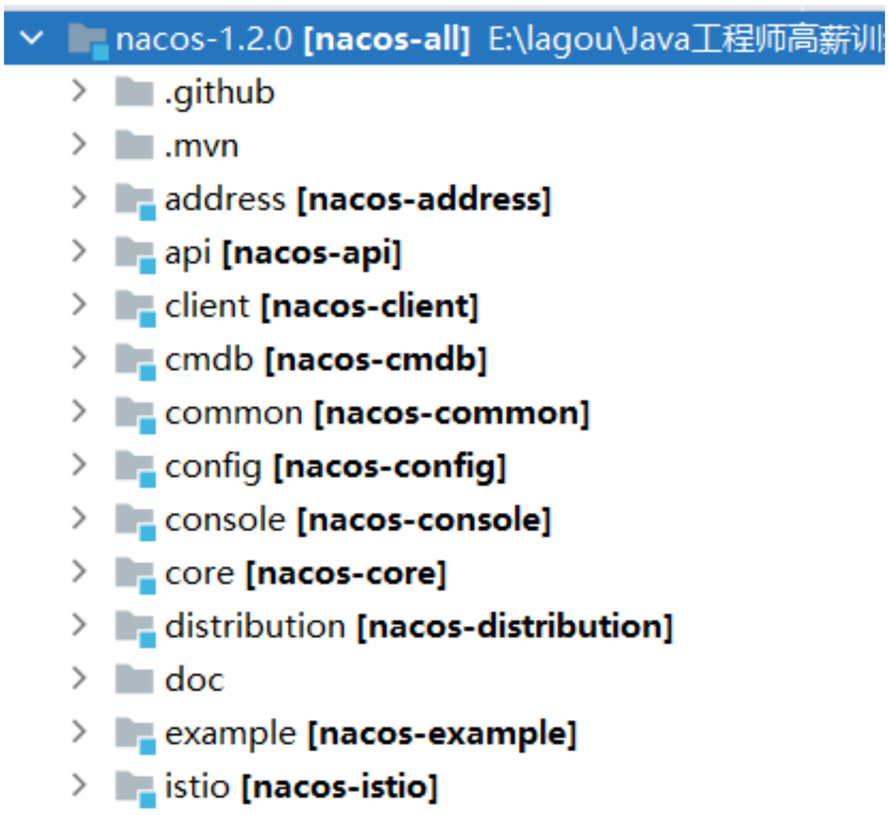
```
#关闭认证缓存
nacos.core.auth.caching.enabled=false
#*************** Config Module Related Configurations ***************#
### If user MySQL as datasource:
spring.datasource.platform=mysql
### Count of DB:
db.num=1
### Connect URL of DB:
db.url.0=jdbc:mysql://127.0.0.1:3306/nacos?characterEncoding=utf8
db.user=root
db.password=root
```

创建nacos数据库,并执行distribution模块中的SQL脚本

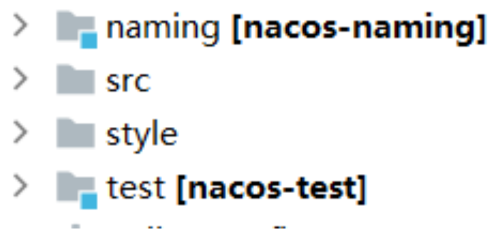


1.3 nacos项目结构

先来看下整个nacos项目结构



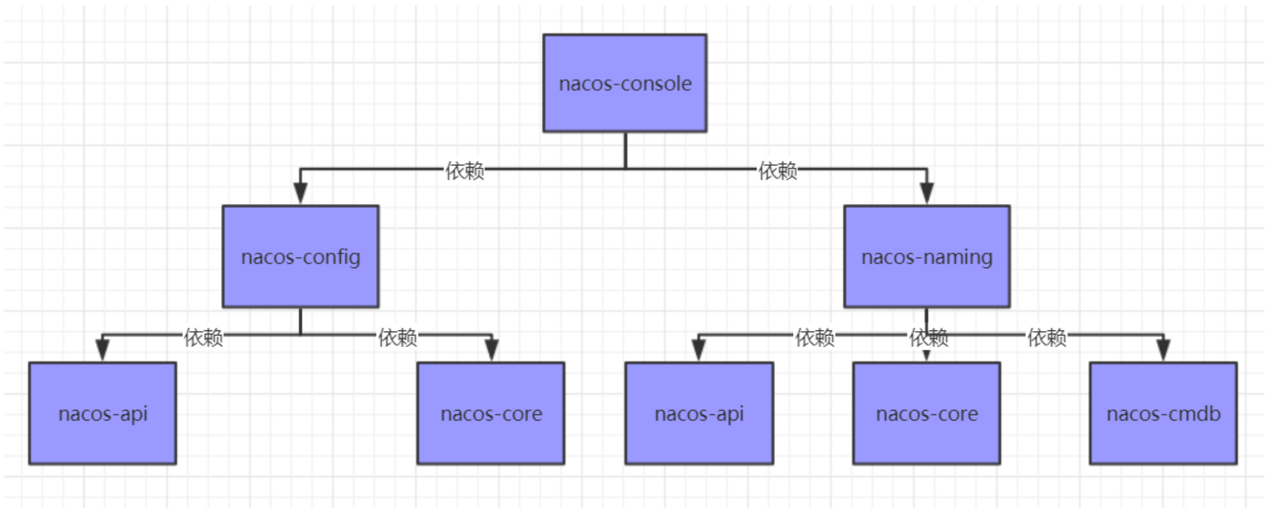
nacos-1.2.0 **[nacos-all]** E:\lagou\Java工程师高薪训
- .github
- .mvn
- address **[nacos-address]**
- api **[nacos-api]**
- client **[nacos-client]**
- cmdb **[nacos-cmdb]**
- common **[nacos-common]**
- config **[nacos-config]**
- console **[nacos-console]**
- core **[nacos-core]**
- distribution **[nacos-distribution]**
- doc
- example **[nacos-example]**
- istio **[nacos-istio]**



- naming **[nacos-naming]**
- src
- style
- test **[nacos-test]**

- address模块: 主要查询nacos集群中节点个数以及IP的列表.
- api模块: 主要给客户端调用的api接口的抽象.
- common模块: 主要是通用的工具包和字符串常量的定义
- client模块: 主要是对依赖api模块和common模块,对api的接口的实现,给nacos的客户端使用.
- cmdb模块: 主要是操作的数据的存储在内存中,该模块提供一个查询数据标签的接口.
- config模块: 主要是服务配置的管理,即配置中心, 提供api给客户端拉去配置信息,以及提供更新配置的,客户端通过长轮询的更新配置信息.数据存储是mysql.
- naming模块: 主要是作为服务注册中心的实现模块,具备服务的注册和服务发现的功能.
- console模块: 主要是实现控制台的功能.具有权限校验、服务状态、健康检查等功能.
- core模块: 主要是实现Spring的PropertySource的后置处理器,用于属性加载，初始化，监听器相关操作
- distribution模块: 主要是打包nacos-server的操作,使用maven-assembly-plugin进行自定义打包,

下面就是各个模块的依赖关系:



2. nacos服务注册发现源码

2.1 @EnableDiscoveryClient 注解

1. @EnableDiscoveryClient注解

```
/**
 * Annotation to enable a DiscoveryClient implementation.
 * @author Spencer Gibb
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import(EnableDiscoveryClientImportSelector.class)
public @interface EnableDiscoveryClient {

  /**
   *  如果为true，ServiceRegistry将自动注册本地服务器。
   */
  boolean autoRegister() default true;
}
```

   EnableDiscoveryClien引用了EnableDiscoveryClientImportSelector类

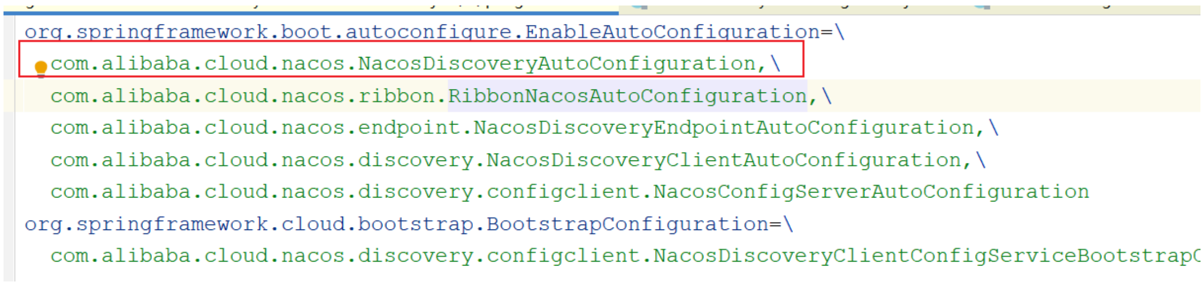
2. EnableDiscoveryClientImportSelector类

```
/*
 * Copyright 2013-2015 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.cloud.client.discovery;

import org.springframework.cloud.commons.util.SpringFactoryImportSelector;
import org.springframework.core.Ordered;
import org.springframework.core.annotation.AnnotationAttributes;
import org.springframework.core.annotation.Order;
import org.springframework.core.env.ConfigurableEnvironment;
import org.springframework.core.env.Environment;
import org.springframework.core.env.MapPropertySource;
import org.springframework.core.type.AnnotationMetadata;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedHashMap;
import java.util.List;

/**
 * @author Spencer Gibb
 */
@Order(Ordered.LOWEST_PRECEDENCE - 100)
public class EnableDiscoveryClientImportSelector
    extends SpringFactoryImportSelector<EnableDiscoveryClient> {

  @Override
  public String[] selectImports(AnnotationMetadata metadata) {
    String[] imports = super.selectImports(metadata);
    //获取注解属性
    AnnotationAttributes attributes = AnnotationAttributes.fromMap(
        metadata.getAnnotationAttributes(getAnnotationClass().getName(),
true));
    //判断是否为true自动服务注册
    boolean autoRegister = attributes.getBoolean("autoRegister");
    //当autoRegister=true 时，将AutoServiceRegistrationConfiguration类添加到
自动装配中，系统就会去自动装配AutoServiceRegistrationConfiguration类
    if (autoRegister) {
      List<String> importsList = new ArrayList<>(Arrays.asList(imports));

importsList.add("org.springframework.cloud.client.serviceregistry.AutoServ
iceRegistrationConfiguration");
```

```
        imports = importsList.toArray(new String[0]);
      } else {
        Environment env = getEnvironment();
        if(ConfigurableEnvironment.class.isInstance(env)) {
          ConfigurableEnvironment configEnv = (ConfigurableEnvironment)env;
          LinkedHashMap<String, Object> map = new LinkedHashMap<>();
          map.put("spring.cloud.service-registry.auto-registration.enabled",
false);
          MapPropertySource propertySource = new MapPropertySource(
              "springCloudDiscoveryClient", map);
          configEnv.getPropertySources().addLast(propertySource);
        }

      }

      return imports;
    }

}
```

3. 开启自动服务注册后会加载spring-cloud-alibaba-nacos-discovery-2.1.0.RELEASE-sources.jar!\META-INF\spring.factories文件中的

   DiscoveryAutoConfiguration配置类,开启nacos服务自动注册

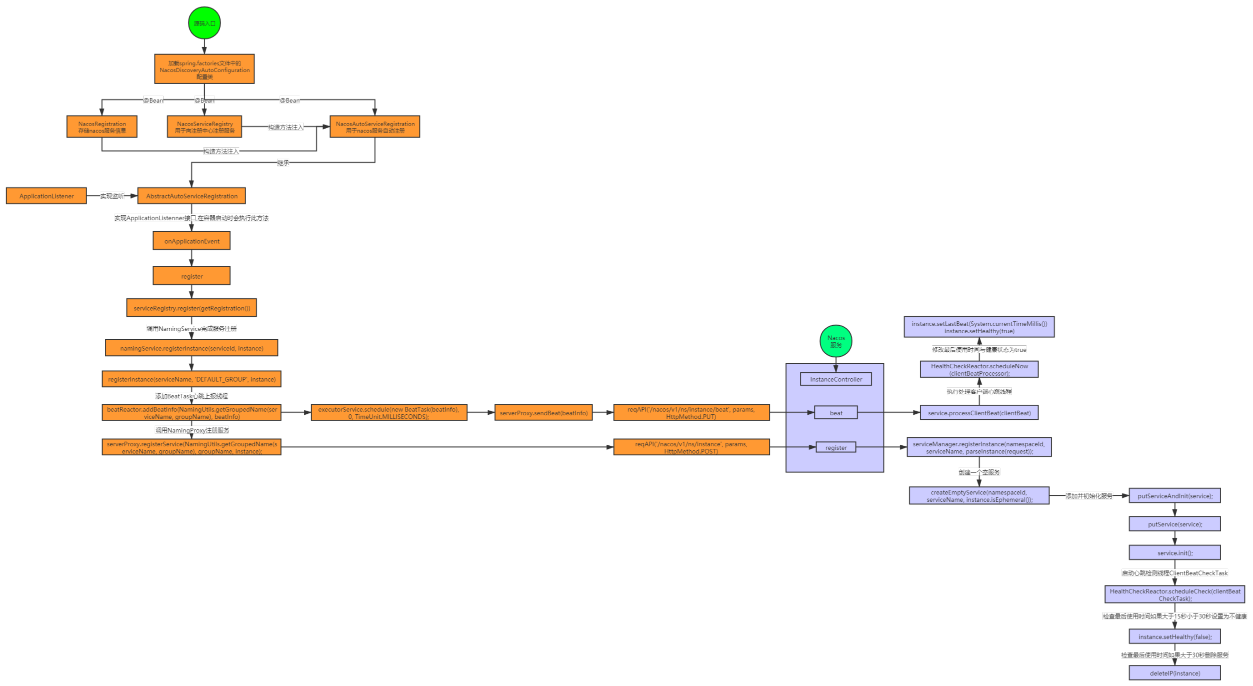


```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
  com.alibaba.cloud.nacos.NacosDiscoveryAutoConfiguration,\
  com.alibaba.cloud.nacos.ribbon.RibbonNacosAutoConfiguration,\
  com.alibaba.cloud.nacos.endpoint.NacosDiscoveryEndpointAutoConfiguration,\
  com.alibaba.cloud.nacos.discovery.NacosDiscoveryClientAutoConfiguration,\
  com.alibaba.cloud.nacos.discovery.configclient.NacosConfigServerAutoConfiguration
org.springframework.cloud.bootstrap.BootstrapConfiguration=\
  com.alibaba.cloud.nacos.discovery.configclient.NacosDiscoveryClientConfigServiceBootstrapC
```

2.2 nacos服务注册

在上一节中我们知道nacos服务注册的入口已经找到, 那么本节我们看下如何完成服务自动发现的.

2.2.1 服务注册流程分析


源码入口
加载spring.factories文件中的NacosDiscoveryAutoConfiguration配置类
@Bean
@Bean
@Bean
NacosRegistration 存储nacos服务信息
NacosServiceRegistry 用于向注册中心注册服务
构造方法注入
NacosAutoServiceRegistration 用于nacos服务自动注册
构造方法注入
继承
ApplicationListener
实现监听
AbstractAutoServiceRegistration
实现ApplicationListenner接口,在容器启动时会执行此方法
onApplicationEvent
register
serviceRegistry.register(getRegistration())
调用NamingService完成服务注册
namingService.registerInstance(serviceId, instance)
registerInstance(serviceName, 'DEFAULT_GROUP', instance)
添加BeatTask心跳上报线程
beatReactor.addBeatInfo(NamingUtils.getGroupedName(serviceName, groupName), beatInfo)
executorService.schedule(new BeatTask(beatInfo), 0, TimeUnit.MILLISECONDS);
serverProxy.sendBeat(beatInfo)
reqAPI('/nacos/v1/ns/instance/beat', params, HttpMethod.PUT)
调用NamingProxy注册服务
serverProxy.registerService(NamingUtils.getGroupedName(serviceName, groupName), groupName, instance);
reqAPI('/nacos/v1/ns/instance', params, HttpMethod.POST)
Nacos 服务
InstanceController
beat
register
instance.setLastBeat(System.currentTimeMillis()) instance.setHealthy(true)
修改最后使用时间与健康状态为true
HealthCheckReactor.scheduleNow(clientBeatProcessor);
执行处理客户端心跳线程
service.processClientBeat(clientBeat)
serviceManager.registerInstance(namespaceId, serviceName, parseInstance(request));
创建一个空服务
createEmptyService(namespaceId, serviceName, instance.isEphemeral());
添加并初始化服务
putServiceAndInit(service);
putService(service);
service.init();
启动心跳检测线程ClientBeatCheckTask
HealthCheckReactor.scheduleCheck(clientBeatCheckTask);
检查最后使用时间如果大于15秒小于30秒设置为不健康
instance.setHealthy(false);
检查最后使用时间如果大于30秒删除服务
deleteIP(instance)

2.2.2 主要源码跟踪

1. NacosDiscoveryAutoConfiguration类

```java
/*
 * Copyright (C) 2018 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.alibaba.cloud.nacos;

import org.springframework.boot.autoconfigure.AutoConfigureAfter;
import org.springframework.boot.autoconfigure.condition.ConditionalOnBean;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import
org.springframework.boot.context.properties.EnableConfigurationProperties;
```

```
import
org.springframework.cloud.client.serviceregistry.AutoServiceRegistrationAu
toConfiguration;
import
org.springframework.cloud.client.serviceregistry.AutoServiceRegistrationCo
nfiguration;
import
org.springframework.cloud.client.serviceregistry.AutoServiceRegistrationPr
operties;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.alibaba.cloud.nacos.registry.NacosAutoServiceRegistration;
import com.alibaba.cloud.nacos.registry.NacosRegistration;
import com.alibaba.cloud.nacos.registry.NacosServiceRegistry;

/**
 * @author xiaojing
 * @author <a href="mailto:mercyblitz@gmail.com">Mercy</a>
 */
@Configuration
@EnableConfigurationProperties
@ConditionalOnNacosDiscoveryEnabled
@ConditionalOnProperty(value = "spring.cloud.service-registry.auto-
registration.enabled", matchIfMissing = true)
@AutoConfigureAfter({ AutoServiceRegistrationConfiguration.class,
    AutoServiceRegistrationAutoConfiguration.class })
public class NacosDiscoveryAutoConfiguration {
  /**
     *
     *  声明向注册中心注册服务的bean
     */
  @Bean
  public NacosServiceRegistry nacosServiceRegistry(
      NacosDiscoveryProperties nacosDiscoveryProperties) {
    return new NacosServiceRegistry(nacosDiscoveryProperties);
  }

  /**
     *
     *  声明存储nacos服务信息的bean
     */
  @Bean
  @ConditionalOnBean(AutoServiceRegistrationProperties.class)
  public NacosRegistration nacosRegistration(
      NacosDiscoveryProperties nacosDiscoveryProperties,
      ApplicationContext context) {
    return new NacosRegistration(nacosDiscoveryProperties, context);
  }
  /**
     *
     *  声明用于nacos服务自动注册的bean
     */
  @Bean
  @ConditionalOnBean(AutoServiceRegistrationProperties.class)
  public NacosAutoServiceRegistration nacosAutoServiceRegistration(
      NacosServiceRegistry registry,
      AutoServiceRegistrationProperties autoServiceRegistrationProperties,
      NacosRegistration registration) {
    return new NacosAutoServiceRegistration(registry,
        autoServiceRegistrationProperties, registration);
  }
}
```

2. NacosAutoServiceRegistration继承AbstractAutoServiceRegistration,AbstractAutoServiceRegistration实现了ApplicationListener监听,所以会执行onApplicationEvent方法

```

public abstract class AbstractAutoServiceRegistration<R extends
Registration>
    implements AutoServiceRegistration, ApplicationContextAware,
ApplicationListener<WebServerInitializedEvent> {


  ..........
  @Override
  @SuppressWarnings("deprecation")
  public void onApplicationEvent(WebServerInitializedEvent event) {
    bind(event);
  }

  @Deprecated
  public void bind(WebServerInitializedEvent event) {
    ApplicationContext context = event.getApplicationContext();
    if (context instanceof ConfigurableWebServerApplicationContext) {
      if ("management".equals(
          ((ConfigurableWebServerApplicationContext)
context).getServerNamespace())) {
        return;
      }
    }
    this.port.compareAndSet(0, event.getWebServer().getPort());
    this.start();//启动
  }

  public void start() {
    if (!isEnabled()) {
      if (logger.isDebugEnabled()) {
        logger.debug("Discovery Lifecycle disabled. Not starting");
      }
      return;
    }

    // only initialize if nonSecurePort is greater than 0 and it isn't
already running
    // because of containerPortInitializer below
    if (!this.running.get()) {
      this.context.publishEvent(new InstancePreRegisteredEvent(this,
getRegistration()));
      register();//执行注册
      if (shouldRegisterManagement()) {
        registerManagement();
      }
      this.context.publishEvent(
          new InstanceRegisteredEvent<>(this, getConfiguration()));
      this.running.compareAndSet(false, true);
    }
  }
}
```

3. NacosNamingService的registerInstance方法

```
public void registerInstance(String serviceName, String groupName,
Instance instance) throws NacosException {

        if (instance.isEphemeral()) {
            BeatInfo beatInfo = new BeatInfo();

 beatInfo.setServiceName(NamingUtils.getGroupedName(serviceName,
groupName));
            beatInfo.setIp(instance.getIp());
            beatInfo.setPort(instance.getPort());
            beatInfo.setCluster(instance.getClusterName());
            beatInfo.setWeight(instance.getWeight());
            beatInfo.setMetadata(instance.getMetadata());
            beatInfo.setScheduled(false);
            long instanceInterval =
instance.getInstanceHeartBeatInterval();
            beatInfo.setPeriod(instanceInterval == 0 ?
DEFAULT_HEART_BEAT_INTERVAL : instanceInterval);
      //添加心跳上报线程

 beatReactor.addBeatInfo(NamingUtils.getGroupedName(serviceName,
groupName), beatInfo);
        }
```

```
    //注册服务

 serverProxy.registerService(NamingUtils.getGroupedName(serviceName,
groupName), groupName, instance);
    }
```

addBeatInfo方法

```
public void addBeatInfo(String serviceName, BeatInfo beatInfo) {
        NAMING_LOGGER.info("[BEAT] adding beat: {} to beat map.",
beatInfo);
        dom2Beat.put(buildKey(serviceName, beatInfo.getIp(),
beatInfo.getPort()), beatInfo);
        executorService.schedule(new BeatTask(beatInfo), 0,
TimeUnit.MILLISECONDS);
        MetricsMonitor.getDom2BeatSizeMonitor().set(dom2Beat.size());
    }
```

```
class BeatTask implements Runnable {

        BeatInfo beatInfo;

        public BeatTask(BeatInfo beatInfo) {
            this.beatInfo = beatInfo;
        }

        @Override
        public void run() {
            if (beatInfo.isStopped()) {
                return;
            }
            long result = serverProxy.sendBeat(beatInfo);//发送心跳
            long nextTime = result > 0 ? result : beatInfo.getPeriod();
            executorService.schedule(new BeatTask(beatInfo), nextTime,
TimeUnit.MILLISECONDS);
        }
    }
```

registerService方法

```
public void registerService(String serviceName, String groupName, Instance
instance) throws NacosException {

        NAMING_LOGGER.info("[REGISTER-SERVICE] {} registering service {}
with instance: {}",
            namespaceId, serviceName, instance);

        final Map<String, String> params = new HashMap<String, String>(9);
        params.put(CommonParams.NAMESPACE_ID, namespaceId);
        params.put(CommonParams.SERVICE_NAME, serviceName);
        params.put(CommonParams.GROUP_NAME, groupName);
        params.put(CommonParams.CLUSTER_NAME, instance.getClusterName());
        params.put("ip", instance.getIp());
        params.put("port", String.valueOf(instance.getPort()));
        params.put("weight", String.valueOf(instance.getWeight()));
        params.put("enable", String.valueOf(instance.isEnabled()));
        params.put("healthy", String.valueOf(instance.isHealthy()));
        params.put("ephemeral", String.valueOf(instance.isEphemeral()));
        params.put("metadata", JSON.toJSONString(instance.getMetadata()));

        reqAPI(UtilAndComs.NACOS_URL_INSTANCE, params, HttpMethod.POST);//
发送注册服务请求

    }
```

4. InstanceController类

```
@RestController
@RequestMapping(UtilsAndCommons.NACOS_NAMING_CONTEXT + "/instance")
public class InstanceController {


   .....

    @CanDistro
    @PostMapping
    @Secured(parser = NamingResourceParser.class, action =
ActionTypes.WRITE)
    public String register(HttpServletRequest request) throws Exception {

        String serviceName = WebUtils.required(request,
CommonParams.SERVICE_NAME);
        String namespaceId = WebUtils.optional(request,
CommonParams.NAMESPACE_ID, Constants.DEFAULT_NAMESPACE_ID);
        // 注册实例
        serviceManager.registerInstance(namespaceId, serviceName,
parseInstance(request));
        return "ok";
    }

    @GetMapping
    @Secured(parser = NamingResourceParser.class, action =
ActionTypes.READ)
    public JSONObject detail(HttpServletRequest request) throws Exception
{

        String namespaceId = WebUtils.optional(request,
CommonParams.NAMESPACE_ID,
            Constants.DEFAULT_NAMESPACE_ID);
        String serviceName = WebUtils.required(request,
CommonParams.SERVICE_NAME);
        String cluster = WebUtils.optional(request,
CommonParams.CLUSTER_NAME, UtilsAndCommons.DEFAULT_CLUSTER_NAME);
        String ip = WebUtils.required(request, "ip");
        int port = Integer.parseInt(WebUtils.required(request, "port"));

        Service service = serviceManager.getService(namespaceId,
serviceName);
        if (service == null) {
            throw new NacosException(NacosException.NOT_FOUND, "no service
" + serviceName + " found!");
        }

        List<String> clusters = new ArrayList<>();
        clusters.add(cluster);

        List<Instance> ips = service.allIPs(clusters);
        if (ips == null || ips.isEmpty()) {
            throw new NacosException(NacosException.NOT_FOUND,
                "no ips found for cluster " + cluster + " in service " +
serviceName);
        }

        for (Instance instance : ips) {
            if (instance.getIp().equals(ip) && instance.getPort() == port)
{
                JSONObject result = new JSONObject();
                result.put("service", serviceName);
                result.put("ip", ip);
                result.put("port", port);
                result.put("clusterName", cluster);
                result.put("weight", instance.getWeight());
                result.put("healthy", instance.isHealthy());
                result.put("metadata", instance.getMetadata());
                result.put("instanceId", instance.getInstanceId());
                return result;
            }
        }

        throw new NacosException(NacosException.NOT_FOUND, "no matched ip
found!");
    }

    @CanDistro
    @PutMapping("/beat")
    @Secured(parser = NamingResourceParser.class, action =
ActionTypes.WRITE)
```

```java
    public JSONObject beat(HttpServletRequest request) throws Exception {

        JSONObject result = new JSONObject();

        result.put("clientBeatInterval",
switchDomain.getClientBeatInterval());
        String serviceName = WebUtils.required(request,
CommonParams.SERVICE_NAME);
        String namespaceId = WebUtils.optional(request,
CommonParams.NAMESPACE_ID,
            Constants.DEFAULT_NAMESPACE_ID);
        String clusterName = WebUtils.optional(request,
CommonParams.CLUSTER_NAME,
            UtilsAndCommons.DEFAULT_CLUSTER_NAME);
        String ip = WebUtils.optional(request, "ip", StringUtils.EMPTY);
        int port = Integer.parseInt(WebUtils.optional(request, "port",
"0"));
        String beat = WebUtils.optional(request, "beat",
StringUtils.EMPTY);

        RsInfo clientBeat = null;
        if (StringUtils.isNotBlank(beat)) {
            clientBeat = JSON.parseObject(beat, RsInfo.class);
        }

        if (clientBeat != null) {
            if (StringUtils.isNotBlank(clientBeat.getCluster())) {
                clusterName = clientBeat.getCluster();
            }
            ip = clientBeat.getIp();
            port = clientBeat.getPort();
        }

        if (Loggers.SRV_LOG.isDebugEnabled()) {
            Loggers.SRV_LOG.debug("[CLIENT-BEAT] full arguments: beat: {},
serviceName: {}", clientBeat, serviceName);
        }

        Instance instance = serviceManager.getInstance(namespaceId,
serviceName, clusterName, ip, port);

        if (instance == null) {
            if (clientBeat == null) {
                result.put(CommonParams.CODE,
NamingResponseCode.RESOURCE_NOT_FOUND);
                return result;
            }
            instance = new Instance();
            instance.setPort(clientBeat.getPort());
            instance.setIp(clientBeat.getIp());
            instance.setWeight(clientBeat.getWeight());
            instance.setMetadata(clientBeat.getMetadata());
            instance.setClusterName(clusterName);
            instance.setServiceName(serviceName);
            instance.setInstanceId(instance.getInstanceId());
            instance.setEphemeral(clientBeat.isEphemeral());

            serviceManager.registerInstance(namespaceId, serviceName,
instance);
        }

        Service service = serviceManager.getService(namespaceId,
serviceName);

        if (service == null) {
            throw new NacosException(NacosException.SERVER_ERROR,
                "service not found: " + serviceName + "@" + namespaceId);
        }
        if (clientBeat == null) {
            clientBeat = new RsInfo();
            clientBeat.setIp(ip);
            clientBeat.setPort(port);
            clientBeat.setCluster(clusterName);
        }
        //处理客户端心跳
        service.processClientBeat(clientBeat);

        result.put(CommonParams.CODE, NamingResponseCode.OK);
        result.put("clientBeatInterval",
instance.getInstanceHeartBeatInterval());
        result.put(SwitchEntry.LIGHT_BEAT_ENABLED,
switchDomain.isLightBeatEnabled());
        return result;
    }
}
```

5. ClientBeatCheckTask类客户端心跳检查线程

```
@Override
    public void run() {
        try {
            if (!getDistroMapper().responsible(service.getName())) {
                return;
            }

            if (!getSwitchDomain().isHealthCheckEnabled()) {
                return;
            }
      //1. 获取实例
            List<Instance> instances = service.allIPs(true);


            //2.检查客户端实例最后使用时间是否超时
            for (Instance instance : instances) {
                if (System.currentTimeMillis() - instance.getLastBeat() >
instance.getInstanceHeartBeatTimeOut()) {
                    if (!instance.isMarked()) {
                        if (instance.isHealthy()) {
                          //3.如果超时15秒设置健康状态为false
                            instance.setHealthy(false);
                            Loggers.EVT_LOG.info("{POS} {IP-DISABLED}
valid: {}:{}@{}@{}, region: {}, msg: client timeout after {}, last beat:
{}",
                                instance.getIp(), instance.getPort(),
instance.getClusterName(), service.getName(),
                                UtilsAndCommons.LOCALHOST_SITE,
instance.getInstanceHeartBeatTimeOut(), instance.getLastBeat());
                            getPushService().serviceChanged(service);
                            SpringContext.getAppContext().publishEvent(new
InstanceHeartbeatTimeoutEvent(this, instance));
                        }
                    }
                }
            }

            if (!getGlobalConfig().isExpireInstance()) {
                return;
            }

            // then remove obsolete instances:
            for (Instance instance : instances) {

                if (instance.isMarked()) {
                    continue;
                }
        // 4. 检查是否超过30秒
                if (System.currentTimeMillis() - instance.getLastBeat() >
instance.getIpDeleteTimeout()) {
                    // delete instance
                    Loggers.SRV_LOG.info("[AUTO-DELETE-IP] service: {},
ip: {}", service.getName(), JSON.toJSONString(instance));
                    // 5. 如果超过30秒则删除实例
                    deleteIP(instance);
                }
            }
        } catch (Exception e) {
            Loggers.SRV_LOG.warn("Exception while processing client beat
time out.", e);
        }

    }
```

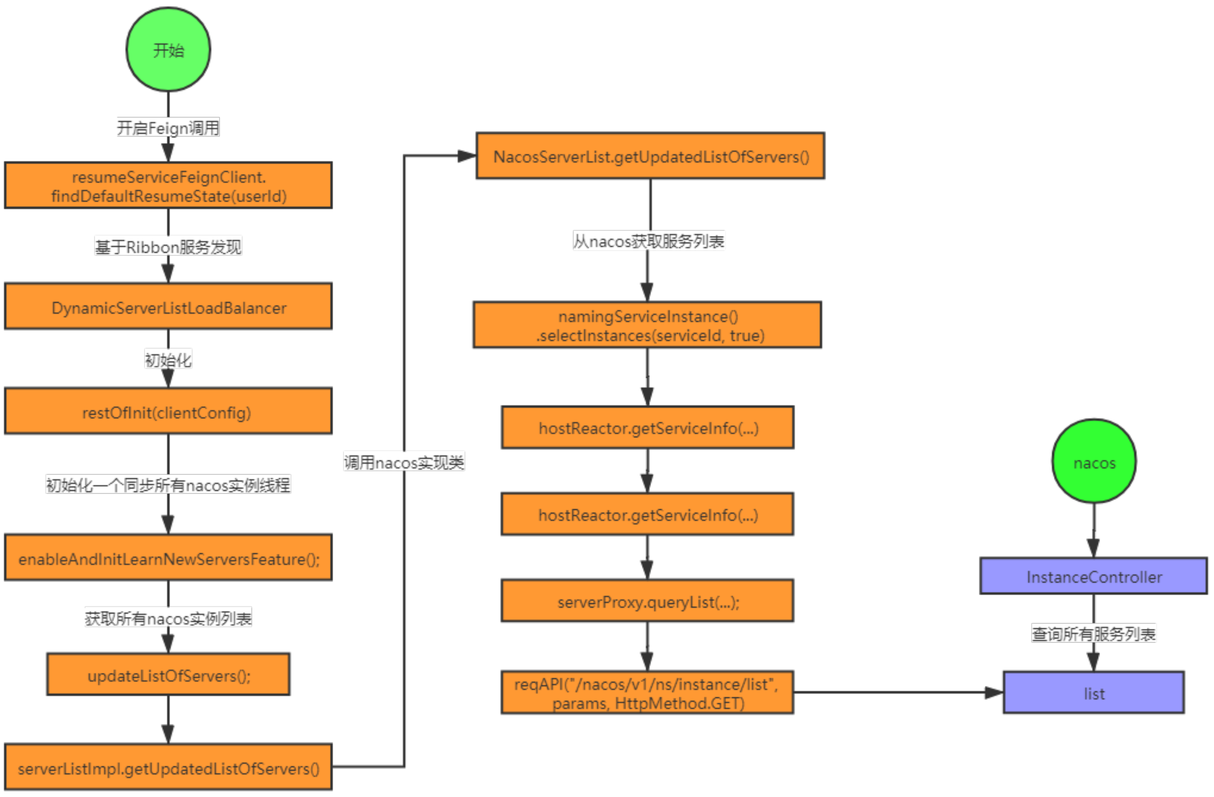
6. ClientBeatProcessor客户端心跳处理类

```
@Override
    public void run() {
        Service service = this.service;
        if (Loggers.EVT_LOG.isDebugEnabled()) {
            Loggers.EVT_LOG.debug("[CLIENT-BEAT] processing beat: {}",
rsInfo.toString());
        }

        String ip = rsInfo.getIp();
        String clusterName = rsInfo.getCluster();
        int port = rsInfo.getPort();
        Cluster cluster = service.getClusterMap().get(clusterName);
        //1.获取所有实例
        List<Instance> instances = cluster.allIPs(true);

        for (Instance instance : instances) {
            if (instance.getIp().equals(ip) && instance.getPort() == port)
{
                if (Loggers.EVT_LOG.isDebugEnabled()) {
                    Loggers.EVT_LOG.debug("[CLIENT-BEAT] refresh beat:
{}", rsInfo.toString());
                }
                //2. 设置实例的最后使用时间
                instance.setLastBeat(System.currentTimeMillis());
                if (!instance.isMarked()) {
                    if (!instance.isHealthy()) {
                        instance.setHealthy(true);
                        Loggers.EVT_LOG.info("service: {} {POS} {IP-
ENABLED} valid: {}:{}@{}, region: {}, msg: client beat ok",
                            cluster.getService().getName(), ip, port,
cluster.getName(), UtilsAndCommons.LOCALHOST_SITE);
                        getPushService().serviceChanged(service);
                    }
                }
            }
        }
    }
```

2.3 nacos服务发现

服务发现是在feign 调用时产生的, 那么我们来看下如何结合Ribbon来完成服务发现的

2.3.1 服务发现流程分析


开始
开启Feign调用
resumeServiceFeignClient.
findDefaultResumeState(userId)
基于Ribbon服务发现
DynamicServerListLoadBalancer
初始化
restOfInit(clientConfig)
初始化一个同步所有nacos实例线程
enableAndInitLearnNewServersFeature();
获取所有nacos实例列表
updateListOfServers();
serverListImpl.getUpdatedListOfServers()
调用nacos实现类
NacosServerList.getUpdatedListOfServers()
从nacos获取服务列表
namingServiceInstance()
.selectInstances(serviceId, true)
hostReactor.getServiceInfo(...)
hostReactor.getServiceInfo(...)
serverProxy.queryList(...);
reqAPI("/nacos/v1/ns/instance/list",
params, HttpMethod.GET)
nacos
InstanceController
查询所有服务列表
list

2.3.2 主要源码跟踪

- 客户端代码

基于Ribbon的服务发现DynamicServerListLoadBalancer

```
public class DynamicServerListLoadBalancer<T extends Server> extends
BaseLoadBalancer {
    private static final Logger LOGGER =
LoggerFactory.getLogger(DynamicServerListLoadBalancer.class);

    ...........

    public DynamicServerListLoadBalancer(IClientConfig clientConfig, IRule
rule, IPing ping,
                                         ServerList<T> serverList,
ServerListFilter<T> filter,
                                         ServerListUpdater serverListUpdater)
{
        super(clientConfig, rule, ping);
        this.serverListImpl = serverList;
        this.filter = filter;
        this.serverListUpdater = serverListUpdater;
        if (filter instanceof AbstractServerListFilter) {
            ((AbstractServerListFilter)
filter).setLoadBalancerStats(getLoadBalancerStats());
        }
        //1. 初始化
        restOfInit(clientConfig);
    }


    void restOfInit(IClientConfig clientConfig) {
        boolean primeConnection = this.isEnablePrimingConnections();
        // turn this off to avoid duplicated asynchronous priming done in
BaseLoadBalancer.setServerList()
        this.setEnablePrimingConnections(false);
        //2. 开启初始化同步服务列表线程
        enableAndInitLearnNewServersFeature();
    //3. 更新服务列表
        updateListOfServers();
        if (primeConnection && this.getPrimeConnections() != null) {
            this.getPrimeConnections()
                    .primeConnections(getReachableServers());
        }
        this.setEnablePrimingConnections(primeConnection);
        LOGGER.info("DynamicServerListLoadBalancer for client {} initialized:
{}", clientConfig.getClientName(), this.toString());
    }
    public void enableAndInitLearnNewServersFeature() {
        LOGGER.info("Using serverListUpdater {}",
serverListUpdater.getClass().getSimpleName());
        serverListUpdater.start(updateAction);
    }

    private String getIdentifier() {
        return this.getClientConfig().getClientName();
    }

    public void stopServerListRefreshing() {
        if (serverListUpdater != null) {
            serverListUpdater.stop();
        }
    }

    @VisibleForTesting
    public void updateListOfServers() {
        List<T> servers = new ArrayList<T>();
        if (serverListImpl != null) {
            //4.调用serverListImpl--->NacosServerList.getUpdatedListOfServers()
方法,获取服务列表
            servers = serverListImpl.getUpdatedListOfServers();
            LOGGER.debug("List of Servers for {} obtained from Discovery
client: {}",
                    getIdentifier(), servers);

            if (filter != null) {
                servers = filter.getFilteredListOfServers(servers);
                LOGGER.debug("Filtered List of Servers for {} obtained from
Discovery client: {}",
                        getIdentifier(), servers);
            }
        }
        updateAllServerList(servers);
    }

    /**
     * Update the AllServer list in the LoadBalancer if necessary and enabled
     *
     * @param ls
     */
    protected void updateAllServerList(List<T> ls) {
        // other threads might be doing this - in which case, we pass
        if (serverListUpdateInProgress.compareAndSet(false, true)) {
            try {
                for (T s : ls) {
                    s.setAlive(true); // set so that clients can start using
these
                                      // servers right away instead
                                      // of having to wait out the ping cycle.
                }
                setServersList(ls);
                super.forceQuickPing();
            } finally {
                serverListUpdateInProgress.set(false);
            }
        }
    }
}
```

serverListImpl.getUpdatedListOfServers()方法

```
private List<NacosServer> getServers() {
    try {
      // 调用namingService获取服务列表
      List<Instance> instances = discoveryProperties.namingServiceInstance()
          .selectInstances(serviceId, true);
      return instancesToServerList(instances);
    }
    catch (Exception e) {
      throw new IllegalStateException(
          "Can not get service instances from nacos, serviceId=" + serviceId,
          e);
    }
  }
```

NacosNamingService

```
public List<Instance> selectInstances(String serviceName, String groupName,
List<String> clusters, boolean healthy, boolean subscribe) throws
NacosException {

        ServiceInfo serviceInfo;
        if (subscribe) {
        // 调用hostReactor的getServiceInfo方法
            serviceInfo =
hostReactor.getServiceInfo(NamingUtils.getGroupedName(serviceName, groupName),
StringUtils.join(clusters, ","));
        } else {
            serviceInfo =
hostReactor.getServiceInfoDirectlyFromServer(NamingUtils.getGroupedName(servic
eName, groupName), StringUtils.join(clusters, ","));
        }
        return selectInstances(serviceInfo, healthy);
    }
```

HostReactor类

```
public void updateServiceNow(String serviceName, String clusters) {
        ServiceInfo oldService = getServiceInfo0(serviceName, clusters);
        try {
      // 调用NamingProxy查询服务列表
            String result = serverProxy.queryList(serviceName, clusters,
pushReceiver.getUDPPort(), false);
            if (StringUtils.isNotEmpty(result)) {
                processServiceJSON(result);
            }
        } catch (Exception e) {
            NAMING_LOGGER.error("[NA] failed to update serviceName: " +
serviceName, e);
```

```
        } finally {
            if (oldService != null) {
                synchronized (oldService) {
                    oldService.notifyAll();
                }
            }
        }
    }
```

NamingProxy

```
public String queryList(String serviceName, String clusters, int udpPort,
boolean healthyOnly)
        throws NacosException {
    //1. 封装参数
        final Map<String, String> params = new HashMap<String, String>(8);
        params.put(CommonParams.NAMESPACE_ID, namespaceId);
        params.put(CommonParams.SERVICE_NAME, serviceName);
        params.put("clusters", clusters);
        params.put("udpPort", String.valueOf(udpPort));
        params.put("clientIP", NetUtils.localIP());
        params.put("healthyOnly", String.valueOf(healthyOnly));
    //2. 发送请求
        return reqAPI(UtilAndComs.NACOS_URL_BASE + "/instance/list", params,
HttpMethod.GET);
    }
```

- nacos服务端代码

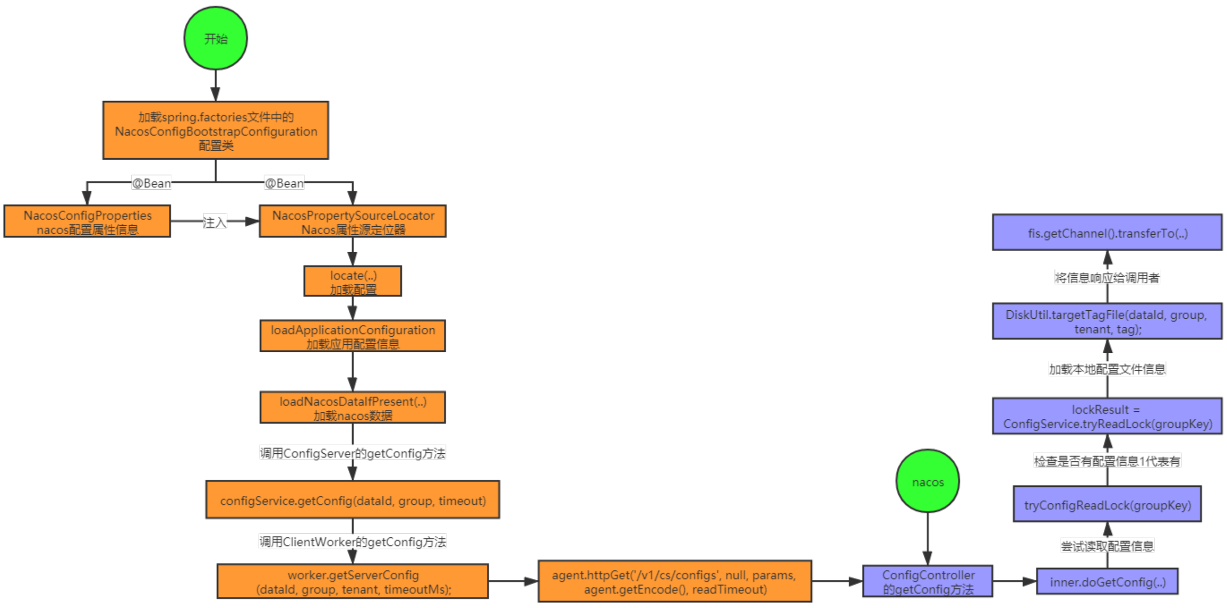
```
@RestController
@RequestMapping(UtilsAndCommons.NACOS_NAMING_CONTEXT + "/instance")
public class InstanceController {

    /**
     * 服务列表查询
     *
     */
    @GetMapping("/list")
    @Secured(parser = NamingResourceParser.class, action = ActionTypes.READ)
    public JSONObject list(HttpServletRequest request) throws Exception {

        String namespaceId = WebUtils.optional(request,
CommonParams.NAMESPACE_ID,
            Constants.DEFAULT_NAMESPACE_ID);

        String serviceName = WebUtils.required(request,
CommonParams.SERVICE_NAME);
        String agent = WebUtils.getUserAgent(request);
```

```java
        String clusters = WebUtils.optional(request, "clusters",
StringUtils.EMPTY);
        String clientIP = WebUtils.optional(request, "clientIP",
StringUtils.EMPTY);
        Integer udpPort = Integer.parseInt(WebUtils.optional(request,
"udpPort", "0"));
        String env = WebUtils.optional(request, "env", StringUtils.EMPTY);
        boolean isCheck = Boolean.parseBoolean(WebUtils.optional(request,
"isCheck", "false"));

        String app = WebUtils.optional(request, "app", StringUtils.EMPTY);

        String tenant = WebUtils.optional(request, "tid", StringUtils.EMPTY);

        boolean healthyOnly = Boolean.parseBoolean(WebUtils.optional(request,
"healthyOnly", "false"));

        return doSrvIPXT(namespaceId, serviceName, agent, clusters, clientIP,
udpPort, env, isCheck, app, tenant,
            healthyOnly);
    }

    private void checkIfDisabled(Service service) throws Exception {
        if (!service.getEnabled()) {
            throw new Exception("service is disabled now.");
        }
    }

    public JSONObject doSrvIPXT(String namespaceId, String serviceName, String
agent, String clusters, String clientIP,
                                int udpPort,
                                String env, boolean isCheck, String app,
String tid, boolean healthyOnly)
        throws Exception {

        ClientInfo clientInfo = new ClientInfo(agent);
        JSONObject result = new JSONObject();
        //1.获取服务信息
        Service service = serviceManager.getService(namespaceId, serviceName);

        if (service == null) {
            if (Loggers.SRV_LOG.isDebugEnabled()) {
                Loggers.SRV_LOG.debug("no instance to serve for service: {}",
serviceName);
            }
            result.put("name", serviceName);
            result.put("clusters", clusters);
            result.put("hosts", new JSONArray());
            return result;
        }
    // 2.检查服务是否可用
        checkIfDisabled(service);

        long cacheMillis = switchDomain.getDefaultCacheMillis();


        try {
            if (udpPort > 0 && pushService.canEnablePush(agent)) {
        // 3.添加客户端信息
                pushService.addClient(namespaceId, serviceName,
                    clusters,
                    agent,
                    new InetSocketAddress(clientIP, udpPort),
                    pushDataSource,
                    tid,
                    app);
                cacheMillis = switchDomain.getPushCacheMillis(serviceName);
            }
        } catch (Exception e) {
            Loggers.SRV_LOG.error("[NACOS-API] failed to added push client {}, 
{}:{}", clientInfo, clientIP, udpPort, e);
            cacheMillis = switchDomain.getDefaultCacheMillis();
        }

        List<Instance> srvedIPs;

        srvedIPs = service.srvIPs(Arrays.asList(StringUtils.split(clusters, 
",")));

        ........

        result.put("hosts", hosts);
        if (clientInfo.type == ClientInfo.ClientType.JAVA &&
            clientInfo.version.compareTo(VersionUtil.parseVersion("1.0.0")) >= 
0) {
            result.put("dom", serviceName);
        } else {
            result.put("dom", NamingUtils.getServiceName(serviceName));
        }
        result.put("name", serviceName);
        result.put("cacheMillis", cacheMillis);
        result.put("lastRefTime", System.currentTimeMillis());
        result.put("checksum", service.getChecksum());
        result.put("useSpecifiedURL", false);
        result.put("clusters", clusters);
        result.put("env", env);
        result.put("metadata", service.getMetadata());
        return result;
        }
    }
```

3. nacos配置中心源码

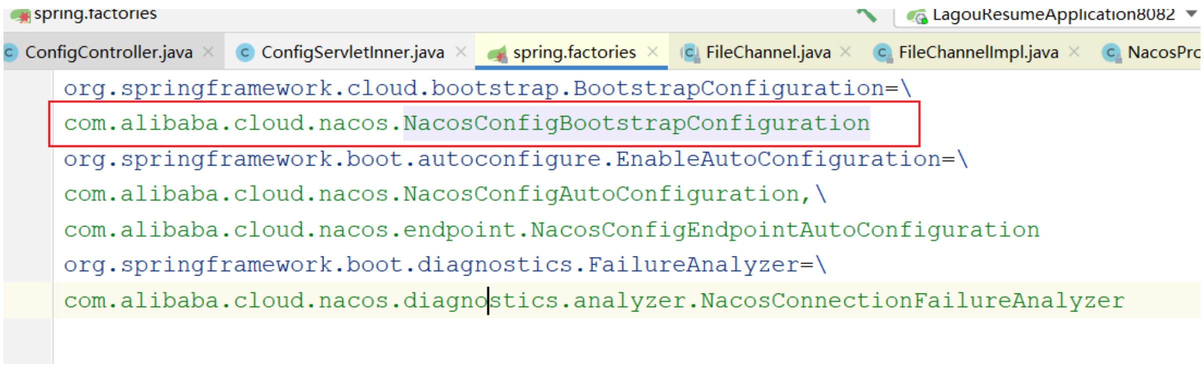
nacos作为配置中心他可以完成配置加载与动态感知nacos配置中心的配置变化, 那么他的底层源码是如何实现的呢? 我们一起来看下

3.1.1 加载配置中心源码分析


开始
加载spring.factories文件中的 NacosConfigBootstrapConfiguration 配置类
@Bean
@Bean
NacosConfigProperties nacos配置属性信息
注入
NacosPropertySourceLocator Nacos属性源定位器
locate(..) 加载配置
loadApplicationConfiguration 加载应用配置信息
loadNacosDataIfPresent(..) 加载nacos数据
调用ConfigServer的getConfig方法
configService.getConfig(dataId, group, timeout)
调用ClientWorker的getConfig方法
worker.getServerConfig (dataId, group, tenant, timeoutMs);
agent.httpGet('/v1/cs/configs', null, params, agent.getEncode(), readTimeout)
nacos
ConfigController 的getConfig方法
inner.doGetConfig(..)
尝试读取配置信息
tryConfigReadLock(groupKey)
检查是否有配置信息1代表有
lockResult = ConfigService.tryReadLock(groupKey)
加载本地配置文件信息
DiskUtil.targetTagFile(dataId, group, tenant, tag);
将信息响应给调用者
fis.getChannel().transferTo(..)

3.1.2 加载配置中心主要源码跟踪

1. springboot项目自动启动会加载spring-cloud-alibaba-nacos-config下面的spring.factories



```
org.springframework.cloud.bootstrap.BootstrapConfiguration=\
com.alibaba.cloud.nacos.NacosConfigBootstrapConfiguration
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.alibaba.cloud.nacos.NacosConfigAutoConfiguration,\
com.alibaba.cloud.nacos.endpoint.NacosConfigEndpointAutoConfiguration
org.springframework.boot.diagnostics.FailureAnalyzer=\
com.alibaba.cloud.nacos.diagnostics.analyzer.NacosConnectionFailureAnalyzer
```

2. NacosConfigBootstrapConfiguration类中会声明2个Bean

```
public class NacosConfigBootstrapConfiguration {
    /**
     *  nacos配置属性对象声明
     *
     */
   @Bean
   @ConditionalOnMissingBean
```

```
    public NacosConfigProperties nacosConfigProperties() {
        return new NacosConfigProperties();
    }
    /**
     *  nacos属性源定位器声明
     *
     */
    @Bean
    public NacosPropertySourceLocator nacosPropertySourceLocator(
            NacosConfigProperties nacosConfigProperties) {
        return new NacosPropertySourceLocator(nacosConfigProperties);
    }

}
```

3. 在NacosPropertySourceLocator类中有个locate方法在spring boot 启动时会最终调用这个方法.

```
@Override
public PropertySource<?> locate(Environment env) {

    ConfigService configService =
nacosConfigProperties.configServiceInstance();

    if (null == configService) {
        log.warn("no instance of config service found, can't load config
from nacos");
        return null;
    }
    long timeout = nacosConfigProperties.getTimeout();
    nacosPropertySourceBuilder = new
NacosPropertySourceBuilder(configService,
            timeout);
    String name = nacosConfigProperties.getName();

    String dataIdPrefix = nacosConfigProperties.getPrefix();
    if (StringUtils.isEmpty(dataIdPrefix)) {
        dataIdPrefix = name;
    }

    if (StringUtils.isEmpty(dataIdPrefix)) {
        dataIdPrefix = env.getProperty("spring.application.name");
    }

    CompositePropertySource composite = new CompositePropertySource(
            NACOS_PROPERTY_SOURCE_NAME);
    //加载共享配置
    loadSharedConfiguration(composite);
    //加载扩展配置
    loadExtConfiguration(composite);
    //加载应用配置
    loadApplicationConfiguration(composite, dataIdPrefix,
nacosConfigProperties, env);

    return composite;
}
```

4. 主要看加载应用配置loadApplicationConfiguration这个方法

```
private void loadApplicationConfiguration(
        CompositePropertySource compositePropertySource, String
dataIdPrefix,
        NacosConfigProperties properties, Environment environment) {
    // 获得后缀扩展类型
    String fileExtension = properties.getFileExtension();
    // 获得配置的分组信息
    String nacosGroup = properties.getGroup();
    // 加载nacos数据
    loadNacosDataIfPresent(compositePropertySource,
            dataIdPrefix + DOT + fileExtension, nacosGroup, fileExtension,
true);
    for (String profile : environment.getActiveProfiles()) {
        String dataId = dataIdPrefix + SEP1 + profile + DOT + fileExtension;
        loadNacosDataIfPresent(compositePropertySource, dataId, nacosGroup,
                fileExtension, true);
    }
}
```

5. loadNacosDataIfPresent方法

```
private void loadNacosDataIfPresent(final CompositePropertySource
composite,
        final String dataId, final String group, String fileExtension,
        boolean isRefreshable) {
    if (NacosContextRefresher.getRefreshCount() != 0) {
        NacosPropertySource ps;
        if (!isRefreshable) {
            ps =
NacosPropertySourceRepository.getNacosPropertySource(dataId);
        }
        else {
            ps = nacosPropertySourceBuilder.build(dataId, group,
fileExtension, true);
        }

        composite.addFirstPropertySource(ps);
    }
    else {
      // 从远程获取NacosPropertySource
      NacosPropertySource ps = nacosPropertySourceBuilder.build(dataId,
group,
            fileExtension, isRefreshable);
      composite.addFirstPropertySource(ps);
   }
}
```

6. nacosPropertySourceBuilder.build方法

```
NacosPropertySource build(String dataId, String group, String
fileExtension,
      boolean isRefreshable) {
   //加载nacos数据
   Properties p = loadNacosData(dataId, group, fileExtension);
   NacosPropertySource nacosPropertySource = new
NacosPropertySource(group, dataId,
         propertiesToMap(p), new Date(), isRefreshable);

NacosPropertySourceRepository.collectNacosPropertySources(nacosPropertySou
rce);
   return nacosPropertySource;
}

private Properties loadNacosData(String dataId, String group, String
fileExtension) {
   String data = null;
   try {
      //调用NacosConfigService（这个类就类似Nacos作为注册中心的逻辑的
NamingService）获取数据
      data = configService.getConfig(dataId, group, timeout);
      if (!StringUtils.isEmpty(data)) {
         log.info(String.format("Loading nacos data, dataId: '%s', group:
'%s'",
               dataId, group));

         if (fileExtension.equalsIgnoreCase("properties")) {
            Properties properties = new Properties();

            properties.load(new StringReader(data));
            return properties;
         }
         else if (fileExtension.equalsIgnoreCase("yaml")
               || fileExtension.equalsIgnoreCase("yml")) {
            YamlPropertiesFactoryBean yamlFactory = new
YamlPropertiesFactoryBean();
            yamlFactory.setResources(new
ByteArrayResource(data.getBytes()));
            return yamlFactory.getObject();
            }

        }
    }
    catch (NacosException e) {
        log.error("get data from Nacos error,dataId:{}, ", dataId, e);
    }
    catch (Exception e) {
        log.error("parse data from Nacos error,dataId:{},data:{},", dataId,
data, e);
    }
    return EMPTY_PROPERTIES;
}
```

7. configService.getConfig经过层层调用会调用getConfigInner(...)方法

```
private String getConfigInner(String tenant, String dataId, String group,
long timeoutMs) throws NacosException {
    group = null2defaultGroup(group);
    ParamUtils.checkKeyParam(dataId, group);
    ConfigResponse cr = new ConfigResponse();

    cr.setDataId(dataId);
    cr.setTenant(tenant);
    cr.setGroup(group);

    // 优先使用本地配置
    String content = LocalConfigInfoProcessor.getFailover(agent.getName(),
dataId, group, tenant);
    if (content != null) {
        LOGGER.warn("[{}] [get-config] get failover ok, dataId={}, group=
{}, tenant={}, config={}", agent.getName(),
            dataId, group, tenant, ContentUtils.truncateContent(content));
        cr.setContent(content);
        configFilterChainManager.doFilter(null, cr);
        content = cr.getContent();
        return content;
    }

    try {
        //调用ClientWorker的getServerConfig方法获取内容
        content = worker.getServerConfig(dataId, group, tenant,
timeoutMs);

        cr.setContent(content);

        configFilterChainManager.doFilter(null, cr);
        content = cr.getContent();
```

```
            return content;
        } catch (NacosException ioe) {
            if (NacosException.NO_RIGHT == ioe.getErrCode()) {
                throw ioe;
            }
            LOGGER.warn("[{}] [get-config] get from server error, dataId={}, 
group={}, tenant={}, msg={}",
                agent.getName(), dataId, group, tenant, ioe.toString());
        }

        LOGGER.warn("[{}] [get-config] get snapshot ok, dataId={}, group={}, 
tenant={}, config={}", agent.getName(),
            dataId, group, tenant, ContentUtils.truncateContent(content));
        content = LocalConfigInfoProcessor.getSnapshot(agent.getName(), 
dataId, group, tenant);
        cr.setContent(content);
        configFilterChainManager.doFilter(null, cr);
        content = cr.getContent();
        return content;
    }
```

8. worker.getServerConfig方法

```
public String getServerConfig(String dataId, String group, String tenant, 
long readTimeout)
    throws NacosException {
    if (StringUtils.isBlank(group)) {
        group = Constants.DEFAULT_GROUP;
    }

    HttpResult result = null;
    try {
        List<String> params = null;
        if (StringUtils.isBlank(tenant)) {
            params = Arrays.asList("dataId", dataId, "group", group);
        } else {
            params = Arrays.asList("dataId", dataId, "group", group, 
"tenant", tenant);
        }
        // 默认访问路径：http://ip:port/nacos/v1/ns/configs
        result = agent.httpGet(Constants.CONFIG_CONTROLLER_PATH, null, 
params, agent.getEncode(), readTimeout);
    } catch (IOException e) {
        String message = String.format(
            "[%s] [sub-server] get server config exception, dataId=%s, 
group=%s, tenant=%s", agent.getName(),
            dataId, group, tenant);
        LOGGER.error(message, e);
        throw new NacosException(NacosException.SERVER_ERROR, e);
```

```java
    }

    switch (result.code) {
        case HttpURLConnection.HTTP_OK:
            LocalConfigInfoProcessor.saveSnapshot(agent.getName(), dataId,
group, tenant, result.content);
            return result.content;
        case HttpURLConnection.HTTP_NOT_FOUND:
            LocalConfigInfoProcessor.saveSnapshot(agent.getName(), dataId,
group, tenant, null);
            return null;
        case HttpURLConnection.HTTP_CONFLICT: {
            LOGGER.error(
                "[{}] [sub-server-error] get server config being modified
concurrently, dataId={}, group={}, "
                    + "tenant={}", agent.getName(), dataId, group,
tenant);
            throw new NacosException(NacosException.CONFLICT,
                "data being modified, dataId=" + dataId + ",group=" +
group + ",tenant=" + tenant);
        }
        case HttpURLConnection.HTTP_FORBIDDEN: {
            LOGGER.error("[{}] [sub-server-error] no right, dataId={},
group={}, tenant={}", agent.getName(), dataId,
                group, tenant);
            throw new NacosException(result.code, result.content);
        }
        default: {
            LOGGER.error("[{}] [sub-server-error]  dataId={}, group={},
tenant={}, code={}", agent.getName(), dataId,
                group, tenant, result.code);
            throw new NacosException(result.code,
                "http error, code=" + result.code + ",dataId=" + dataId +
",group=" + group + ",tenant=" + tenant);
        }
    }
}
```

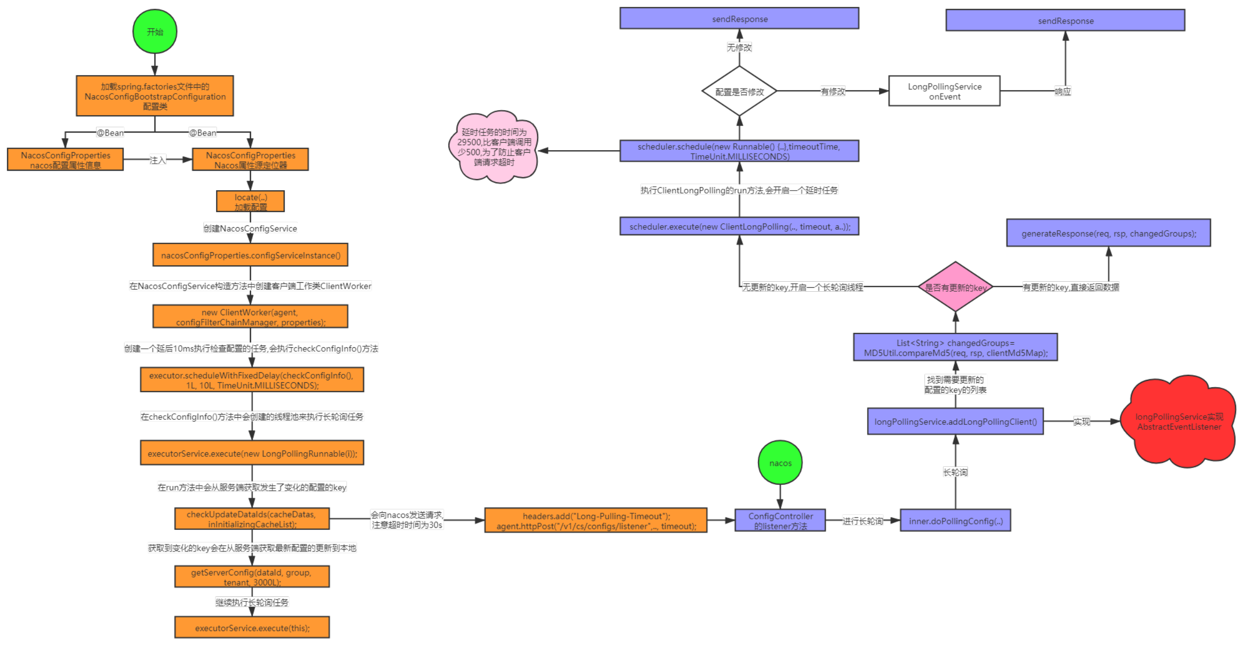
9. 发送http的get请求后会到达nacos的ConfigController的getConfig方法完成查询配置操作

```java
/**
 * 取数据
 *
 * @throws ServletException
 * @throws IOException
 * @throws NacosException
 */
@GetMapping
@Secured(action = ActionTypes.READ, parser = ConfigResourceParser.class)
```

```
public void getConfig(HttpServletRequest request, HttpServletResponse
response,
                      @RequestParam("dataId") String dataId,
@RequestParam("group") String group,
                      @RequestParam(value = "tenant", required = false,
defaultValue = StringUtils.EMPTY)
                          String tenant,
                      @RequestParam(value = "tag", required = false)
String tag)
    throws IOException, ServletException, NacosException {
    // check params
    ParamUtils.checkParam(dataId, group, "datumId", "content");
    ParamUtils.checkParam(tag);

    final String clientIp = RequestUtil.getRemoteIp(request);
    inner.doGetConfig(request, response, dataId, group, tenant, tag,
clientIp);
}
```

3.2.1 客户端动态感知源码分析

上面分析了初始化的时候客户端如何加载配置，那么当服务端的配置信息变更的时候，客户端又是如何动态感知的呢?



开始
加载spring.factories文件中的 NacosConfigBootstrapConfiguration 配置类
@Bean
@Bean
NacosConfigProperties nacos配置属性信息
注入
NacosConfigProperties Nacos属性源定位器
locate(..) 加载配置
创建NacosConfigService
nacosConfigProperties.configServiceInstance()
在NacosConfigService构造方法中创建客户端工作类ClientWorker
new ClientWorker(agent, configFilterChainManager, properties);
创建一个延后10ms执行检查配置的任务,会执行checkConfigInfo()方法
executor.scheduleWithFixedDelay(checkConfigInfo(), 1L, 10L, TimeUnit.MILLISECONDS);
在checkConfigInfo()方法中会创建的线程池来执行长轮询任务
executorService.execute(new LongPollingRunnable(i));
在run方法中会从服务端获取发生了变化的配置的key
checkUpdateDataIds(cacheDatas, inInitializingCacheList);
会向nacos发送请求, 注意超时时间为30s
headers.add("Long-Pulling-Timeout"); agent.httpPost("/v1/cs/configs/listener",.., timeout);
获取到变化的key会在从服务端获取最新配置的更新到本地
getServerConfig(dataId, group, tenant, 3000L);
继续执行长轮询任务
executorService.execute(this);
nacos
ConfigController 的listener方法
进行长轮询
inner.doPollingConfig(..)
长轮询
longPollingService.addLongPollingClient()
实现
longPollingService实现 AbstractEventListener
找到需要更新的 配置的key的列表
List<String> changedGroups= MD5Util.compareMd5(req, rsp, clientMd5Map);
是否有更新的key
有更新的key,直接返回数据
generateResponse(req, rsp, changedGroups);
无更新的key,开启一个长轮询线程
scheduler.execute(new ClientLongPolling(.., timeout, a..));
执行ClientLongPolling的run方法,会开启一个延时任务
scheduler.schedule(new Runnable() {..},timeoutTime, TimeUnit.MILLISECONDS)
延时任务的时间为29500,比客户端调用少500,为了防止客户端请求超时
配置是否修改
无修改
sendResponse
有修改
LongPollingService onEvent
响应
sendResponse

3.2.2 客户端动态感知主要源码跟踪

1. 在NacosPropertySourceLocator类中有个locate方法中会创建NacosConfigService对象

```
public PropertySource<?> locate(Environment env) {
   //创建NacosConfigService对象
   ConfigService configService =
nacosConfigProperties.configServiceInstance();
```

```java
    if (null == configService) {
        log.warn("no instance of config service found, can't load config
from nacos");
        return null;
    }
    long timeout = nacosConfigProperties.getTimeout();
    nacosPropertySourceBuilder = new
NacosPropertySourceBuilder(configService,
            timeout);
    String name = nacosConfigProperties.getName();

    String dataIdPrefix = nacosConfigProperties.getPrefix();
    if (StringUtils.isEmpty(dataIdPrefix)) {
        dataIdPrefix = name;
    }

    if (StringUtils.isEmpty(dataIdPrefix)) {
        dataIdPrefix = env.getProperty("spring.application.name");
    }

    CompositePropertySource composite = new CompositePropertySource(
            NACOS_PROPERTY_SOURCE_NAME);

    loadSharedConfiguration(composite);
    loadExtConfiguration(composite);
    loadApplicationConfiguration(composite, dataIdPrefix,
nacosConfigProperties, env);

    return composite;
}
```

2. NacosConfigService构造方法

```java
public NacosConfigService(Properties properties) throws NacosException {
    String encodeTmp = properties.getProperty(PropertyKeyConst.ENCODE);
    if (StringUtils.isBlank(encodeTmp)) {
        encode = Constants.ENCODE;
    } else {
        encode = encodeTmp.trim();
    }
    initNamespace(properties);
    agent = new MetricsHttpAgent(new ServerHttpAgent(properties));
    agent.start();
    // 这里会初始化一个客户端工作类
    worker = new ClientWorker(agent, configFilterChainManager,
properties);
}
```

3. ClientWorker实例的初始化

```
public ClientWorker(final HttpAgent agent, final ConfigFilterChainManager
configFilterChainManager, final Properties properties) {
    this.agent = agent;
    this.configFilterChainManager = configFilterChainManager;

    // Initialize the timeout parameter

    init(properties);
  // 初始化只有一个核心线程的线程池
    executor = Executors.newScheduledThreadPool(1, new ThreadFactory() {
        @Override
        public Thread newThread(Runnable r) {
            Thread t = new Thread(r);
            t.setName("com.alibaba.nacos.client.Worker." +
agent.getName());
            t.setDaemon(true);
            return t;
        }
    });
   // 初始化用于长轮询的线程池
    executorService =
Executors.newScheduledThreadPool(Runtime.getRuntime().availableProcessors(
), new ThreadFactory() {
        @Override
        public Thread newThread(Runnable r) {
            Thread t = new Thread(r);
            t.setName("com.alibaba.nacos.client.Worker.longPolling." +
agent.getName());
            t.setDaemon(true);
            return t;
        }
    });
   // 延后10ms执行检查配置的任务
    executor.scheduleWithFixedDelay(new Runnable() {
        @Override
        public void run() {
            try {
              // 检查配置
                checkConfigInfo();
            } catch (Throwable e) {
                LOGGER.error("[" + agent.getName() + "] [sub-check] rotate
check error", e);
            }
        }
    }, 1L, 10L, TimeUnit.MILLISECONDS);
```

4. checkConfigInfo(),该方法开始会检查配置

```
public void checkConfigInfo() {
    // 分任务
    int listenerSize = cacheMap.get().size();
    // 向上取整为批数
    int longingTaskCount = (int) Math.ceil(listenerSize /
ParamUtil.getPerTaskConfigSize());
    if (longingTaskCount > currentLongingTaskCount) {
        for (int i = (int) currentLongingTaskCount; i < longingTaskCount;
i++) {
            // 要判断任务是否在执行 这块需要好好想想。 任务列表现在是无序的。变化过程
可能有问题
            // 通过初始化的时候创建的线程池来执行长轮询任务
            executorService.execute(new LongPollingRunnable(i));
        }
        currentLongingTaskCount = longingTaskCount;
    }
}
```

5. LongPollingRunnable的run方法

```
public void run() {

        List<CacheData> cacheDatas = new ArrayList<CacheData>();
        List<String> inInitializingCacheList = new ArrayList<String>();
        try {
            // check failover config
            for (CacheData cacheData : cacheMap.get().values()) {
                if (cacheData.getTaskId() == taskId) {
                    cacheDatas.add(cacheData);
                    try {
                        checkLocalConfig(cacheData);
                        if (cacheData.isUseLocalConfigInfo()) {
                            cacheData.checkListenerMd5();
                        }
                    } catch (Exception e) {
                        LOGGER.error("get local config info error", e);
                    }
                }
            }

             // 从服务端获取发生了变化的配置的key (chengedGroupKeys表示服务端告诉
客户端, 哪些配置发生了变化)
            List<String> changedGroupKeys = checkUpdateDataIds(cacheDatas,
inInitializingCacheList);
      // 遍历发生了变化的key, 并根据key去服务端请求最新配置, 并更新到内存缓存中
            for (String groupKey : changedGroupKeys) {
                String[] key = GroupKey.parseKey(groupKey);
                String dataId = key[0];
                String group = key[1];
                String tenant = null;
                if (key.length == 3) {
                    tenant = key[2];
                }
                try {
                  // 从远程服务端获取最新的配置，并缓存到内存中
                    String content = getServerConfig(dataId, group,
tenant, 3000L);
                    CacheData cache =
cacheMap.get().get(GroupKey.getKeyTenant(dataId, group, tenant));
                    cache.setContent(content);
                    LOGGER.info("[{}] [data-received] dataId={}, group={},
tenant={}, md5={}, content={}",
                        agent.getName(), dataId, group, tenant,
cache.getMd5(),
                        ContentUtils.truncateContent(content));
                } catch (NacosException ioe) {
                    String message = String.format(
                        "[%s] [get-update] get changed config exception.
dataId=%s, group=%s, tenant=%s",
                        agent.getName(), dataId, group, tenant);
                    LOGGER.error(message, ioe);
                }
            }
            for (CacheData cacheData : cacheDatas) {
                if (!cacheData.isInitializing() || inInitializingCacheList
                    .contains(GroupKey.getKeyTenant(cacheData.dataId,
cacheData.group, cacheData.tenant))) {
                    cacheData.checkListenerMd5();
                    cacheData.setInitializing(false);
                }
            }
            inInitializingCacheList.clear();
      // 继续执行该任务
            executorService.execute(this);

        } catch (Throwable e) {

            // If the rotation training task is abnormal, the next
execution time of the task will be punished
            LOGGER.error("longPolling error : ", e);
            executorService.schedule(this, taskPenaltyTime,
TimeUnit.MILLISECONDS);
        }
    }
}
```

6. checkUpdateDataIds方法会调用checkUpdateConfigStr方法

```java
/**
 * 从Server获取值变化了的DataID列表。返回的对象里只有dataId和group是有效的。 保证不
返回NULL。
 */
List<String> checkUpdateConfigStr(String probeUpdateString, boolean
isInitializingCacheList) throws IOException {

    List<String> params = Arrays.asList(Constants.PROBE_MODIFY_REQUEST,
probeUpdateString);

    List<String> headers = new ArrayList<String>(2);
    headers.add("Long-Pulling-Timeout");
    headers.add("" + timeout);

    // told server do not hang me up if new initializing cacheData added
in
    if (isInitializingCacheList) {
      //添加长轮询请求头
        headers.add("Long-Pulling-Timeout-No-Hangup");
        headers.add("true");
    }

    if (StringUtils.isBlank(probeUpdateString)) {
        return Collections.emptyList();
    }

    try {
      // 请求路径: http://ip:port/nacos/v1/ns/configs/listener
        HttpResult result =
agent.httpPost(Constants.CONFIG_CONTROLLER_PATH + "/listener", headers,
params,
            agent.getEncode(), timeout);

        if (HttpURLConnection.HTTP_OK == result.code) {
            setHealthServer(true);
            return parseUpdateDataIdResponse(result.content);
        } else {
            setHealthServer(false);
            LOGGER.error("[{}] [check-update] get changed dataId error,
code: {}", agent.getName(), result.code);
        }
    } catch (IOException e) {
        setHealthServer(false);
        LOGGER.error("[" + agent.getName() + "] [check-update] get changed
dataId exception", e);
        throw e;
    }
    return Collections.emptyList();
}
```

7. 发送http的get请求后会到达nacos的ConfigController的getConfig方法完成查询配置操作

```
/**
 * 比较MD5
 */
@PostMapping("/listener")
@Secured(action = ActionTypes.READ, parser = ConfigResourceParser.class)
public void listener(HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException {
    request.setAttribute("org.apache.catalina.ASYNC_SUPPORTED", true);
    String probeModify = request.getParameter("Listening-Configs");
    if (StringUtils.isBlank(probeModify)) {
        throw new IllegalArgumentException("invalid probeModify");
    }

    log.info("listen config id:" + probeModify);

    probeModify = URLDecoder.decode(probeModify, Constants.ENCODE);

    Map<String, String> clientMd5Map;
    try {
        clientMd5Map = MD5Util.getClientMd5Map(probeModify);
    } catch (Throwable e) {
        throw new IllegalArgumentException("invalid probeModify");
    }

    log.info("listen config id 2:" + probeModify);

    // 执行长轮询
    inner.doPollingConfig(request, response, clientMd5Map,
probeModify.length());
}
```

8. doPollingConfig方法

```
/**
 * 轮询接口
 */
public String doPollingConfig(HttpServletRequest request,
HttpServletResponse response,
                              Map<String, String> clientMd5Map, int
probeRequestSize)
    throws IOException {

    // 长轮询,判断请求头中是否支持长轮询
    if (LongPollingService.isSupportLongPolling(request)) {
        longPollingService.addLongPollingClient(request, response,
clientMd5Map, probeRequestSize);
        return HttpServletResponse.SC_OK + "";
    }

    // else 兼容短轮询逻辑
    List<String> changedGroups = MD5Util.compareMd5(request, response,
clientMd5Map);

    // 兼容短轮询result
    String oldResult = MD5Util.compareMd5OldResult(changedGroups);
    String newResult = MD5Util.compareMd5ResultString(changedGroups);

    String version = request.getHeader(Constants.CLIENT_VERSION_HEADER);
    if (version == null) {
        version = "2.0.0";
    }
    int versionNum = Protocol.getVersionNumber(version);

    /**
     * 2.0.4版本以前, 返回值放入header中
     */
    if (versionNum < START_LONGPOLLING_VERSION_NUM) {
        response.addHeader(Constants.PROBE_MODIFY_RESPONSE, oldResult);
        response.addHeader(Constants.PROBE_MODIFY_RESPONSE_NEW,
newResult);
    } else {
        request.setAttribute("content", newResult);
    }

    Loggers.AUTH.info("new content:" + newResult);

    // 禁用缓存
    response.setHeader("Pragma", "no-cache");
    response.setDateHeader("Expires", 0);
    response.setHeader("Cache-Control", "no-cache,no-store");
    response.setStatus(HttpServletResponse.SC_OK);
    return HttpServletResponse.SC_OK + "";
}
```

9. addLongPollingClient方法

```
public void addLongPollingClient(HttpServletRequest req,
HttpServletResponse rsp, Map<String, String> clientMd5Map,
                                 int probeRequestSize) {

    String str = req.getHeader(LongPollingService.LONG_POLLING_HEADER);
    String noHangUpFlag =
req.getHeader(LongPollingService.LONG_POLLING_NO_HANG_UP_HEADER);
```

```
    String appName = req.getHeader(RequestUtil.CLIENT_APPNAME_HEADER);
    String tag = req.getHeader("Vipserver-Tag");
    int delayTime =
SwitchService.getSwitchInteger(SwitchService.FIXED_DELAY_TIME, 500);
    /**
     * 提前500ms返回响应，为避免客户端超时
     */
    long timeout = Math.max(10000, Long.parseLong(str) - delayTime);
    if (isFixedPolling()) {
        timeout = Math.max(10000, getFixedPollingInterval());
    } else {
        long start = System.currentTimeMillis();
        // 获取改变的key
        List<String> changedGroups = MD5Util.compareMd5(req, rsp,
clientMd5Map);
        if (changedGroups.size() > 0) {
            // 如果有改变的key直接返回
            generateResponse(req, rsp, changedGroups);
            LogUtil.clientLog.info("{}|{}|{}|{}|{}|{}|{}",
                System.currentTimeMillis() - start, "instant",
RequestUtil.getRemoteIp(req), "polling",
                clientMd5Map.size(), probeRequestSize,
changedGroups.size());
            return;
        } else if (noHangUpFlag != null &&
noHangUpFlag.equalsIgnoreCase(TRUE_STR)) {
            LogUtil.clientLog.info("{}|{}|{}|{}|{}|{}|{}",
System.currentTimeMillis() - start, "nohangup",
                RequestUtil.getRemoteIp(req), "polling",
clientMd5Map.size(), probeRequestSize,
                changedGroups.size());
            return;
        }
    }
    String ip = RequestUtil.getRemoteIp(req);
    // 一定要由HTTP线程调用，否则离开后容器会立即发送响应
    final AsyncContext asyncContext = req.startAsync();
    // AsyncContext.setTimeout()的超时时间不准，所以只能自己控制
    asyncContext.setTimeout(0L);
    // 开启一个长轮询线程
    scheduler.execute(
        new ClientLongPolling(asyncContext, clientMd5Map, ip,
probeRequestSize, timeout, appName, tag));
}
```

10. ClientLongPolling的run方法

```
public void run() {
    // 调度一个延时执行的任务，在这段延时的时间内，会监听配置的修改
    asyncTimeoutFuture = scheduler.schedule(new Runnable() {
        @Override
        public void run() {
            try {
                getRetainIps().put(ClientLongPolling.this.ip,
System.currentTimeMillis());
                allSubs.remove(ClientLongPolling.this);

                if (isFixedPolling()) {
                    LogUtil.clientLog.info("{}|{}|{}|{}|{}|{}",
                        (System.currentTimeMillis() - createTime),
                        "fix",
RequestUtil.getRemoteIp((HttpServletRequest)asyncContext.getRequest()),
                        "polling",
                        clientMd5Map.size(), probeRequestSize);
                    List<String> changedGroups = MD5Util.compareMd5(
                        (HttpServletRequest)asyncContext.getRequest(),
                        (HttpServletResponse)asyncContext.getResponse(),
clientMd5Map);
                    if (changedGroups.size() > 0) {
                        sendResponse(changedGroups);
                    } else {
                        sendResponse(null);
                    }
                } else {
                    LogUtil.clientLog.info("{}|{}|{}|{}|{}|{}",
                        (System.currentTimeMillis() - createTime),
                        "timeout",
RequestUtil.getRemoteIp((HttpServletRequest)asyncContext.getRequest()),
                        "polling",
                        clientMd5Map.size(), probeRequestSize);
                    sendResponse(null);
                }
            } catch (Throwable t) {
                LogUtil.defaultLog.error("long polling error:" +
t.getMessage(), t.getCause());
            }

        }

    }, timeoutTime, TimeUnit.MILLISECONDS);

    allSubs.add(this);
}
```

11. onEvent方法

```
public void onEvent(Event event) {
    if (isFixedPolling()) {
        // ignore
    } else {
        if (event instanceof LocalDataChangeEvent) {
            LocalDataChangeEvent evt = (LocalDataChangeEvent)event;
            //启动线程
            scheduler.execute(new DataChangeTask(evt.groupKey, evt.isBeta,
evt.betaIps));
        }
    }
}
```

12. DataChangeTask

```
public void run() {
    try {
        ConfigService.getContentBetaMd5(groupKey);
        for (Iterator<ClientLongPolling> iter = allSubs.iterator();
iter.hasNext(); ) {
            ClientLongPolling clientSub = iter.next();
            if (clientSub.clientMd5Map.containsKey(groupKey)) {
                // 如果beta发布且不在beta列表直接跳过
                if (isBeta && !betaIps.contains(clientSub.ip)) {
                    continue;
                }

                // 如果tag发布且不在tag列表直接跳过
                if (StringUtils.isNotBlank(tag) &&
!tag.equals(clientSub.tag)) {
                    continue;
                }

                getRetainIps().put(clientSub.ip,
System.currentTimeMillis());
                iter.remove(); // 删除订阅关系
                LogUtil.clientLog.info("{}|{}|{}|{}|{}|{}|{}",
                    (System.currentTimeMillis() - changeTime),
                    "in-advance",

 RequestUtil.getRemoteIp((HttpServletRequest)clientSub.asyncContext.getReq
uest()),
                    "polling",
                    clientSub.clientMd5Map.size(),
clientSub.probeRequestSize, groupKey);
                clientSub.sendResponse(Arrays.asList(groupKey));
            }
        }
    } catch (Throwable t) {
        LogUtil.defaultLog.error("data change error:" + t.getMessage(),
t.getCause());
    }
}
```