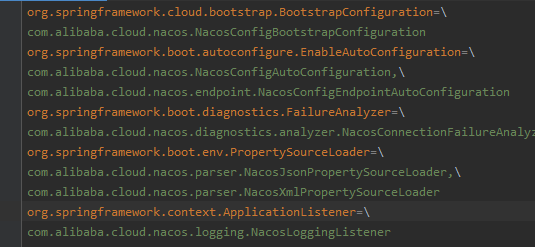
# Nacos功能查看

## 配置项表格

|  |  |  |  |
| --- | --- | --- | --- |
| **配置项** | **key** | **默认值** | **说明** |
| 服务端地址 | spring.cloud.nacos.config.server-addr |  | 服务器ip和端口 |
| DataId前缀 | spring.cloud.nacos.config.prefix | ${spring.application.name} | DataId的前缀，默认值为应用名称 |
| Group | spring.cloud.nacos.config.group | DEFAULT\_GROUP |  |
| DataId后缀及内容文件格式 | spring.cloud.nacos.config.file-extension | properties | DataId的后缀，同时也是配置内容的文件格式，目前只支持 properties |
| 用户名 | spring.cloud.nacos.config.username |  |  |
| 密码 | spring.cloud.nacos.config. password |  |  |
| 配置内容的编码方式 | spring.cloud.nacos.config.encode | UTF-8 | 配置的编码 |
| 获取配置的超时时间 | spring.cloud.nacos.config.timeout | 3000 | 单位为 ms |
| 配置的命名空间 | spring.cloud.nacos.config.namespace |  | 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隔离等。 |
| AccessKey | spring.cloud.nacos.config.access-key |  |  |
| SecretKey | spring.cloud.nacos.config.secret-key |  |  |
| 相对路径 | spring.cloud.nacos.config.context-path |  | 服务端 API 的相对路径 |
| 接入点 | spring.cloud.nacos.config.endpoint |  | 地域的某个服务的入口域名，通过此域名可以动态地拿到服务端地址 |
| 是否开启监听和自动刷新 | spring.cloud.nacos.config.refresh-enabled | true |  |
| 集群服务名 | spring.cloud.nacos.config.cluster-name |  |  |
| 配置长轮询超时时间 | spring.cloud.nacos.config.configLongPollTimeout |  |  |
| 配置重试时间 | spring.cloud.nacos.config.configRetryTime |  |  |
| 重试最大次数 | spring.cloud.nacos.config.maxRetry |  |  |
| 监听器首次添加时拉取远端配置 | spring.cloud.nacos.config.enableRemoteSyncConfig | false |  |
| <ECS/ACK绑定的RAM角色> | spring.cloud.nacos.config.ramRoleName |  |  |
| 共享配置文件 | spring.cloud.nacos.config.sharedConfigs |  |  |
| 扩展配置 | spring.cloud.nacos.config.extensionConfigs |  |  |
| 服务端地址 | spring.cloud.nacos.discovery.server-addr |  |  |
| 服务名 | spring.cloud.nacos.discovery.service | spring.application.name |  |
| 定时器30s一次 | spring.cloud.nacos.discovery.watchDelay | 30s |  |
| 权重 | spring.cloud.nacos.discovery.weight | 1 | 取值范围 1 到 100，数值越大，权重越大 |
| 网卡名 | spring.cloud.nacos.discovery.network-interface |  | 当IP未配置时，注册的IP为此网卡所对应的IP地址，如果此项也未配置，则默认取第一块网卡的地址 |
| 注册的IP地址 | spring.cloud.nacos.discovery.ip |  | 优先级最高 |
| 注册的端口 | spring.cloud.nacos.discovery.port | -1 | 默认情况下不用配置，会自动探测 |
| 命名空间 | spring.cloud.nacos.discovery.namespace |  | 常用场景之一是不同环境的注册的区分隔离，例如开发测试环境和生产环境的资源（如配置、服务）隔离等。 |
| AccessKey | spring.cloud.nacos.discovery.access-key |  |  |
| SecretKey | spring.cloud.nacos.discovery.secret-key |  |  |
| Metadata | spring.cloud.nacos.discovery.metadata |  | 使用Map格式配置 |
| 日志文件名 | spring.cloud.nacos.discovery.log-name |  |  |
| 接入点 | spring.cloud.nacos.discovery.endpoint | UTF-8 | 地域的某个服务的入口域名，通过此域名可以动态地拿到服务端地址 |
| 是否集成Ribbon | ribbon.nacos.enabled | true |  |
| 网卡名 | spring.cloud.nacos.discovery.network-interface |  | 当IP未配置时，注册的IP为此网卡所对应的IP地址，如果此项也未配置，则默认取第一块网卡的地址 |
| 注册的IP地址 | spring.cloud.nacos.discovery.ip |  | 优先级最高 |
| 注册的端口 | spring.cloud.nacos.discovery.port | -1 | 默认情况下不用配置，会自动探测 |
| 命名空间 | spring.cloud.nacos.discovery.namespace |  | 常用场景之一是不同环境的注册的区分隔离，例如开发测试环境和生产环境的资源（如配置、服务）隔离等。 |
| AccessKey | spring.cloud.nacos.discovery.access-key |  |  |
| SecretKey | spring.cloud.nacos.discovery.secret-key |  |  |
| Metadata | spring.cloud.nacos.discovery.metadata |  | 使用Map格式配置 |
| 日志文件名 | spring.cloud.nacos.discovery.log-name |  |  |
| 接入点 | spring.cloud.nacos.discovery.endpoint | UTF-8 | 地域的某个服务的入口域名，通过此域名可以动态地拿到服务端地址 |
| 是否集成Ribbon | ribbon.nacos.enabled | true |  |

# Nacos Cofnig 源码解析

## spring.factories



springboot按照这个顺序去加载jar包下的东西，需要注意的是，cloud的bootstrap的优先级是高于boot的autoconfigure的

## NacosConfigBootstrapConfiguration

// 加proxyBeanMethods=false代表@bean表示的方法的参数每次不再从spring容器里获取，而是直接new一个，并且new的这个对象不经过spring的生命周期，也就是不走cglib代理。

@Configuration(proxyBeanMethods = false)

// 这个是自动配置的开关。默认是开启

@ConditionalOnProperty(name = "spring.cloud.nacos.config.enabled", matchIfMissing = true)

public class NacosConfigBootstrapConfiguration {

// 第一步先注册一个NacosConfigProperties

@Bean

@ConditionalOnMissingBean

public NacosConfigProperties nacosConfigProperties() {

return new NacosConfigProperties();

}

// 第二步注册一个NacosConfigManager

@Bean

@ConditionalOnMissingBean

public NacosConfigManager nacosConfigManager(

NacosConfigProperties nacosConfigProperties) {

return new NacosConfigManager(nacosConfigProperties);

}

// 第二步注册一个NacosPropertySourceLocator

@Bean

public NacosPropertySourceLocator nacosPropertySourceLocator(

NacosConfigManager nacosConfigManager) {

return new NacosPropertySourceLocator(nacosConfigManager);

}

}

### NacosConfigProperties

没有构造方法，默认走的空参构造。

// 就是把配置类的一些属性读进来

@ConfigurationProperties(NacosConfigProperties.PREFIX)

public class NacosConfigProperties {

/\*\*

\* Prefix of {@link NacosConfigProperties}.

\*/

public static final String PREFIX = "spring.cloud.nacos.config";

### NacosConfigManager

public NacosConfigManager(NacosConfigProperties nacosConfigProperties) {

this.nacosConfigProperties = nacosConfigProperties;

// 与NacoConfigProperties中的旧代码兼容，它将在将来删除.

createConfigService(nacosConfigProperties);

}

static ConfigService createConfigService(

NacosConfigProperties nacosConfigProperties) {

// 双重检查

if (Objects.isNull(service)) {

synchronized (NacosConfigManager.class) {

try {

if (Objects.isNull(service)) {

service = NacosFactory.createConfigService(

nacosConfigProperties.assembleConfigServiceProperties());

}

}

catch (NacosException e) {

log.error(e.getMessage());

throw new NacosConnectionFailureException(

nacosConfigProperties.getServerAddr(), e.getMessage(), e);

}

}

}

return service;

}

public static ConfigService createConfigService(Properties properties) throws NacosException {

try {

Class<?> driverImplClass = Class.forName("com.alibaba.nacos.client.config.NacosConfigService");

Constructor constructor = driverImplClass.getConstructor(Properties.class);

ConfigService vendorImpl = (ConfigService) constructor.newInstance(properties);

return vendorImpl;

} catch (Throwable e) {

throw new NacosException(-400, e.getMessage());

}

}

1. 利用双重检查，new一个ConfigService的单例
2. 传入的是上面new的配置类，配置类已经读取了配置信息
3. 利用反射，new了一个NacosConfigService实例。

### NacosConfigService

public NacosConfigService(Properties properties) throws NacosException {

// 设置一些编码格式

String encodeTmp = properties.getProperty(PropertyKeyConst.ENCODE);

if (StringUtils.isBlank(encodeTmp)) {

encode = Constants.ENCODE;

} else {

encode = encodeTmp.trim();

}

initNamespace(properties);

//发送请求的http工具类

agent = new MetricsHttpAgent(new ServerHttpAgent(properties));

agent.start();

worker = new ClientWorker(agent, configFilterChainManager);

}

1. 创建一个ScheduledThreadPoolExecutor线程池为主线程池；
2. 创建一个ScheduledThreadPoolExecutor线城池，定时执行checkConfigInfo()

public ClientWorker(final HttpAgent agent, final ConfigFilterChainManager configFilterChainManager) {

this.agent = agent;

this.configFilterChainManager = configFilterChainManager;

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;

}

});

executorService = Executors.newCachedThreadPool(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;

}

});

executor.scheduleWithFixedDelay(new Runnable() {

public void run() {

try {

checkConfigInfo();

} catch (Throwable e) {

LOGGER.error("[" + agent.getName() + "] [sub-check] rotate check error", e);

}

}

}, 1L, 10L, TimeUnit.MILLISECONDS);

}

1. checkConfigInfo方法中，是给线程做调度，给数据分批，默认是3000个一批，如果批数增加一个，就增加一个LongPollingRunnable线程

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

executorService.execute(new LongPollingRunnable(i));

}

currentLongingTaskCount = longingTaskCount;

}

}

1. 检查本地配置，这里面有三种情况：

* 如果isUseLocalConfigInfo为false，表示不使用本地配置，但是本地缓存路径的文件是存在的，于是把isUseLocalConfigInfo设置为true，并且更新cacheData的内容以及文件的更新时间
* 如果isUseLocalConfigInfo为true，表示使用本地配置文件，但是本地缓存文件不存在，则设置为false，不通知监听器。
* 如果isUseLocalConfigInfo为true，并且本地缓存文件也存在，但是缓存的的时间和文件的更新时间不一致，则更新cacheData中的内容，并且isUseLocalConfigInfo设置为true。

private void checkLocalConfig(CacheData cacheData) {

final String dataId = cacheData.dataId;

final String group = cacheData.group;

final String tenant = cacheData.tenant;

File path = LocalConfigInfoProcessor.getFailoverFile(agent.getName(), dataId, group, tenant);

// 没有 -> 有

if (!cacheData.isUseLocalConfigInfo() && path.exists()) {

String content = LocalConfigInfoProcessor.getFailover(agent.getName(), dataId, group, tenant);

final String md5 = MD5Utils.md5Hex(content, Constants.ENCODE);

cacheData.setUseLocalConfigInfo(true);

cacheData.setLocalConfigInfoVersion(path.lastModified());

cacheData.setContent(content);

String encryptedDataKey = LocalEncryptedDataKeyProcessor

.getEncryptDataKeyFailover(agent.getName(), dataId, group, tenant);

cacheData.setEncryptedDataKey(encryptedDataKey);

LOGGER.warn(

"[{}] [failover-change] failover file created. dataId={}, group={}, tenant={}, md5={}, content={}",

agent.getName(), dataId, group, tenant, md5, ContentUtils.truncateContent(content));

return;

}

// 有 -> 没有。不通知业务监听器，从server拿到配置后通知。

// If use local config info, then it doesn't notify business listener and notify after getting from server.

if (cacheData.isUseLocalConfigInfo() && !path.exists()) {

cacheData.setUseLocalConfigInfo(false);

LOGGER.warn("[{}] [failover-change] failover file deleted. dataId={}, group={}, tenant={}", agent.getName(),

dataId, group, tenant);

return;

}

// 有变更

if (cacheData.isUseLocalConfigInfo() && path.exists() && cacheData.getLocalConfigInfoVersion() != path

.lastModified()) {

String content = LocalConfigInfoProcessor.getFailover(agent.getName(), dataId, group, tenant);

final String md5 = MD5Utils.md5Hex(content, Constants.ENCODE);

cacheData.setUseLocalConfigInfo(true);

cacheData.setLocalConfigInfoVersion(path.lastModified());

cacheData.setContent(content);

String encryptedDataKey = LocalEncryptedDataKeyProcessor

.getEncryptDataKeyFailover(agent.getName(), dataId, group, tenant);

cacheData.setEncryptedDataKey(encryptedDataKey);

LOGGER.warn(

"[{}] [failover-change] failover file changed. dataId={}, group={}, tenant={}, md5={}, content={}",

agent.getName(), dataId, group, tenant, md5, ContentUtils.truncateContent(content));

}

}

1. 在LongPollingRunnable.run中，先通过本地配置的读取和检查来判断数据是否发生变化从而实现变化的通知。接着，当前的线程还需要去远程服务器上获得最新的数据，检查哪些数据发生了变化

* 通过checkUpdateDataIds获取远程服务器上数据变更的dataid
* 遍历这些变化的集合，然后调用getServerConfig从远程服务器获得对应的内容
* 更新本地的cache，设置为服务器端返回的内容
* 最后遍历cacheDatas，找到变化的数据进行通知

// check server config

//从服务端获取发生变化的数据的DataID列表，保存在List<String>集合中

List<String> changedGroupKeys = checkUpdateDataIds(cacheDatas, inInitializingCacheList);

if (!CollectionUtils.isEmpty(changedGroupKeys)) {

LOGGER.info("get changedGroupKeys:" + changedGroupKeys);

}

//遍历发生了变更的配置项

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 {

//逐项根据这些配置项获取配置信息

ConfigResponse response = getServerConfig(dataId, group, tenant, 3000L);

//把配置信息保存到CacheData中

CacheData cache = cacheMap.get(GroupKey.getKeyTenant(dataId, group, tenant));

cache.setContent(response.getContent());

cache.setEncryptedDataKey(response.getEncryptedDataKey());

if (null != response.getConfigType()) {

cache.setType(response.getConfigType());

}

LOGGER.info("[{}] [data-received] dataId={}, group={}, tenant={}, md5={}, content={}, type={}",

agent.getName(), dataId, group, tenant, cache.getMd5(),

ContentUtils.truncateContent(response.getContent()), response.getConfigType());

} 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);

}

}

//再遍历CacheData这个集合，找到发生变化的数据进行通知

for (CacheData cacheData : cacheDatas) {

if (!cacheData.isInitializing() || inInitializingCacheList

.contains(GroupKey.getKeyTenant(cacheData.dataId, cacheData.group, cacheData.tenant))) {

cacheData.checkListenerMd5();

cacheData.setInitializing(false);

}

}

inInitializingCacheList.clear();

//继续传递当前线程进行轮询

executorService.execute(this);

1. checkListenerMd5()中会根据每部分数据的md5去匹配具体的封装listener的wrap中的md5，然后调用safeNotifyListener()方法

void checkListenerMd5() {

// 轮询所有数据，将文件的md5和封装listener的wrap中旧的md5去比较

for (ManagerListenerWrap wrap : listeners) {

if (!md5.equals(wrap.lastCallMd5)) {

safeNotifyListener(dataId, group, content, md5, wrap);

}

}

}

1. safeNotifyListener中去回调这个listener的receiveConfigInfo()方法。

private void safeNotifyListener(final String dataId, final String group, final String content,

final String md5, final ManagerListenerWrap listenerWrap) {

final Listener listener = listenerWrap.listener;

Runnable job = new Runnable() {

public void run() {

ClassLoader myClassLoader = Thread.currentThread().getContextClassLoader();

ClassLoader appClassLoader = listener.getClass().getClassLoader();

try {

if (listener instanceof AbstractSharedListener) {

AbstractSharedListener adapter = (AbstractSharedListener)listener;

adapter.fillContext(dataId, group);

LOGGER.info("[{}] [notify-context] dataId={}, group={}, md5={}", name, dataId, group, md5);

}

// 执行回调之前先将线程classloader设置为具体webapp的classloader，以免回调方法中调用spi接口是出现异常或错用（多应用部署才会有该问题）。

Thread.currentThread().setContextClassLoader(appClassLoader);

ConfigResponse cr = new ConfigResponse();

cr.setDataId(dataId);

cr.setGroup(group);

cr.setContent(content);

configFilterChainManager.doFilter(null, cr);

String contentTmp = cr.getContent();

// 这里回调我们注册的listener

listener.receiveConfigInfo(contentTmp);

listenerWrap.lastCallMd5 = md5;

LOGGER.info("[{}] [notify-ok] dataId={}, group={}, md5={}, listener={} ", name, dataId, group, md5,

listener);

} catch (NacosException de) {

LOGGER.error("[{}] [notify-error] dataId={}, group={}, md5={}, listener={} errCode={} errMsg={}", name,

dataId, group, md5, listener, de.getErrCode(), de.getErrMsg());

} catch (Throwable t) {

LOGGER.error("[{}] [notify-error] dataId={}, group={}, md5={}, listener={} tx={}", name, dataId, group,

md5, listener, t.getCause());

} finally {

Thread.currentThread().setContextClassLoader(myClassLoader);

}

}

};

final long startNotify = System.currentTimeMillis();

try {

if (null != listener.getExecutor()) {

listener.getExecutor().execute(job);

} else {

job.run();

}

} catch (Throwable t) {

LOGGER.error("[{}] [notify-error] dataId={}, group={}, md5={}, listener={} throwable={}", name, dataId, group,

md5, listener, t.getCause());

}

final long finishNotify = System.currentTimeMillis();

LOGGER.info("[{}] [notify-listener] time cost={}ms in ClientWorker, dataId={}, group={}, md5={}, listener={} ",

name, (finishNotify - startNotify), dataId, group, md5, listener);

}

### NacosPropertySourceLocator

public NacosPropertySourceLocator(NacosConfigManager nacosConfigManager) {

this.nacosConfigManager = nacosConfigManager;

this.nacosConfigProperties = nacosConfigManager.getNacosConfigProperties();

}

通过构造方法new了一个NacosPropertySourceLocator，然后把2.1的nacosConfigProperties和2.2的nacosConfigManager都指向到自己类成员变量上。

## NacosConfigAutoConfiguration

@Configuration(proxyBeanMethods = false)

@ConditionalOnProperty(name = "spring.cloud.nacos.config.enabled", matchIfMissing = true)

public class NacosConfigAutoConfiguration {

// 这个之前已经加载过了，从老的上下文中直接拿过来

@Bean

public NacosConfigProperties nacosConfigProperties(ApplicationContext context) {

if (context.getParent() != null

&& BeanFactoryUtils.beanNamesForTypeIncludingAncestors(

context.getParent(), NacosConfigProperties.class).length > 0) {

return BeanFactoryUtils.beanOfTypeIncludingAncestors(context.getParent(),

NacosConfigProperties.class);

}

return new NacosConfigProperties();

}

// 这个就是自动刷新的开关，关闭修改配置不会自动刷新

@Bean

public NacosRefreshProperties nacosRefreshProperties() {

return new NacosRefreshProperties();

}

// 这个是历史数据

@Bean

public NacosRefreshHistory nacosRefreshHistory() {

return new NacosRefreshHistory();

}

// 这个会重新加载一下，但是因为存在双重检查，不会新建NacosConfigService

@Bean

public NacosConfigManager nacosConfigManager(

NacosConfigProperties nacosConfigProperties) {

return new NacosConfigManager(nacosConfigProperties);

}

@Bean

public NacosContextRefresher nacosContextRefresher(

NacosConfigManager nacosConfigManager,

NacosRefreshHistory nacosRefreshHistory) {

// 认为无需与以前的配置兼容，并在必要的时候使用新配置

return new NacosContextRefresher(nacosConfigManager, nacosRefreshHistory);

}

}

### NacosContextRefresher

public NacosContextRefresher(NacosConfigManager nacosConfigManager,

NacosRefreshHistory refreshHistory) {

this.nacosConfigProperties = nacosConfigManager.getNacosConfigProperties();

this.nacosRefreshHistory = refreshHistory;

this.configService = nacosConfigManager.getConfigService();

this.isRefreshEnabled = this.nacosConfigProperties.isRefreshEnabled();

}

注意一点，这个类是个listener：

public class NacosContextRefresher

implements ApplicationListener<ApplicationReadyEvent>, ApplicationContextAware

监听了ApplicationReadyEvent事件，这个事件在springboot的refresh()方法中的最后，可以理解为springboot都启动好了之后，推送这个事件。监听事件的行为：

@Override

public void onApplicationEvent(ApplicationReadyEvent event) {

// many Spring context

if (this.ready.compareAndSet(false, true)) {

this.registerNacosListenersForApplications();

}

}

利用cas，保证registerNacosListenersForApplications()方法只执行一次。

private void registerNacosListenersForApplications() {

// 判断是否允许自动刷新的总开关

if (isRefreshEnabled()) {

// 遍历所有的数据，需要刷新的数据页就注册lintener

for (NacosPropertySource propertySource : NacosPropertySourceRepository

.getAll()) {

if (!propertySource.isRefreshable()) {

continue;

}

String dataId = propertySource.getDataId();

registerNacosListener(propertySource.getGroup(), dataId);

}

}

}

private void registerNacosListener(final String groupKey, final String dataKey) {

String key = NacosPropertySourceRepository.getMapKey(dataKey, groupKey);

Listener listener = listenerMap.computeIfAbsent(key,

lst -> new AbstractSharedListener() {

@Override

public void innerReceive(String dataId, String group,

String configInfo) {

refreshCountIncrement();

nacosRefreshHistory.addRefreshRecord(dataId, group, configInfo);

// todo feature: support single refresh for listening

applicationContext.publishEvent(

new RefreshEvent(this, null, "Refresh Nacos config"));

if (log.isDebugEnabled()) {

log.debug(String.format(

"Refresh Nacos config group=%s,dataId=%s,configInfo=%s",

group, dataId, configInfo));

}

}

});

try {

configService.addListener(dataKey, groupKey, listener);

}

catch (NacosException e) {

log.warn(String.format(

"register fail for nacos listener ,dataId=[%s],group=[%s]", dataKey,

groupKey), e);

}

}

这里相当于注册了一个listener到listenerMap，同时把这listener加到worker的缓存里。在这个listener里，推送了一个真正的刷新事件RefreshEvent。

## NacosConfigEndpointAutoConfiguration

这个配置主要注册一个actuator端点和一个健康检查：

@ConditionalOnWebApplication

@ConditionalOnClass(Endpoint.class)

@ConditionalOnProperty(name = "spring.cloud.nacos.config.enabled", matchIfMissing = true)

public class NacosConfigEndpointAutoConfiguration {

@Autowired

private NacosConfigManager nacosConfigManager;

@Autowired

private NacosRefreshHistory nacosRefreshHistory;

// actuator

@ConditionalOnMissingBean

@ConditionalOnAvailableEndpoint

@Bean

public NacosConfigEndpoint nacosConfigEndpoint() {

return new NacosConfigEndpoint(nacosConfigManager.getNacosConfigProperties(),

nacosRefreshHistory);

}

// 健康检查

@Bean

@ConditionalOnMissingBean

@ConditionalOnEnabledHealthIndicator("nacos-config")

public NacosConfigHealthIndicator nacosConfigHealthIndicator() {

return new NacosConfigHealthIndicator(nacosConfigManager.getConfigService());

}

}

## NacosConnectionFailureAnalyzer

处理自定义异常NacosConnectionFailureException：

public class NacosConnectionFailureAnalyzer

extends AbstractFailureAnalyzer<NacosConnectionFailureException> {

@Override

protected FailureAnalysis analyze(Throwable rootFailure,

NacosConnectionFailureException cause) {

return new FailureAnalysis(

"Application failed to connect to Nacos server: \""

+ cause.getServerAddr() + "\"",

"Please check your Nacos server config", cause);

}

}