# 组件简介

## 组件功能介绍

统一配置功能功能包含在commons.configuration.jar中，他主要提供一下几个能力：

* 发布properties文件的配置项到ZK的指定节点：
* 业务配置文件：\*.serviceConfig.properties
* 资源配置文件：\*. resource.properties
* 错误码文件：\*.resultcode.properties
* 用户自定义的properties文件

提供了一组简化的API来增删改ZK上的节点和对应的值

供了一组简化的API来监听ZK上节点的增删改事件

支持ZK断链恢复后，客户端自动重连，数据监听不丢失

* 提供了redis连接池，数据库连接池自动刷新，spring bean属性动态刷新
* 提供了updatecfg,showerr,showcfg等DFx命令来修改和展示ZK上的配置信息和错误码信息

# 涉及到的外部组件

## Zookeeper

Zookeeper服务启动正常

# 配置文件和API

## 配置文件

### 配置文件名称和路径约束

配置文件必须要遵守以下的约束

* 文件名必须以zk-cluster.xml结尾。前面可加上对应的模块名。如subscribe.zk-cluster.xml
* 文件的路径：直接放在classpath下

private final String ZKCLUSTER\_PATH = "classpath\*:\*zk-cluster.xml";

### 配置文件内容示范

<cluster>

<connector>

<name>subscribe.configerZKClient</name>

<url>10.179.133.171:2183</url>

<retryTimes>5</retryTimes>

<module>subscribe</module>

<retryIntervalMs>3000</retryIntervalMs>

<connectionTimeoutMs>30000</connectionTimeoutMs>

<sessionTimeoutMs>60000</sessionTimeoutMs>

<initNodePath>

/config/sub/resource,/config/sub/serviceConfig,/config/sub/symbolNodePath

</initNodePath>

<symbolNodePath>/config/sub/symbolNodePath</symbolNodePath> <resourcePath>

subscribe.resource.properties|/config/sub/resource,subscribe.serviceConfig.properties|/config/sub/serviceConfig

</resourcePath>

<errorCoderesourcePath>

subscribe.resultcode.properties|/config/resultcode/sub

</errorCoderesourcePath>

<dumpFixedSeconds>1000</dumpFixedSeconds>

</connector>

</cluster>

### 配置文件说明

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 是否必填 |
| name | Connector的唯一标示 | 是 |
| module | 业务模块名称 | 否 |
| url | ZK的IP地址和端口 | 是 |
| retryTimes | 客户端重试策略-重试次数 | 是 |
| retryIntervalMs | 客户端重试策略-重试的时间间隔 | 是 |
| connectionTimeoutMs | 连接超时时间 | 是 |
| sessionTimeoutMs | 会话超时时间 | 是 |
| initNodePath | 在ZK上预先创建节点 | 否 |
| symbolNodePath | 防止重复加载数据的标志位。1：加载完成 0：未加载 | 否 |
| resourcePath | 将文件里面的配置项写入ZK对应的节点 | 否 |
| errorCoderesourcePath | 将文件的错误码和错误描述写入ZK上对应节点 | 否 |
| dumpFixedSeconds | 多长时间把ZK上的数据备份到本地一次 | 是 |

### ZK数据备份

dumpFixedSeconds:每隔xx秒就把initNodePath配置的zk节点数据备份到本地，当zk上的节点有变化时，也会立刻更新本地文件zkdata.properties

{user\_home}/zkdata/zkdata.properties

windows:C:\Users\j00318697\zkdata

linux:/home/dsdp/zkdata

zkdata.properties文件内容如下：

/config/sub/serviceConfig/memPeak=10

/config/sub/resource/iread.jdbc.maxWait=60000

/config/sub/serviceConfig/cpuPercent=80

手工同步ZK上的数据和zkdata.properties：

直接删除zkdata.properties文件，重启应用会自动用ZK上的initNodePath节点上的数据重新更新zkdata.properties文件

## API使用介绍

下面分别介绍统一配置组建的使用

### 组件的启动

ConfigerServerStarter.*init*();

* 使用组件前必须执行
* 主要完成以下两件事

1．根据\*zk-cluster.xml初始化ZK客户端，并缓存

2．将\*zk-cluster.xml里面的资源文件内容注册到对应的ZK节点上;

### 组件的关闭

ConfigerServerStarter.*destroy*();

* 主要完成以下动作

1．关闭所有的ZK客户端

### 获取ZK客户端

ZKClientUtil clientUtil = new ZKClientUtil("subscribe.configerZKClient");

* 所有的增删改节点都必须先获取ZKClientUtil对象：
* 参数说明：

1．subscribe.configerZKClient：对应\*zk-cluster.xml文件的里面name节点

### 监听回调函数统一说明

* 监听事件种类

增加节点：ZKNodeEventType.*NODE\_ADDED*

删除节点：ZKNodeEventType. *NODE\_REMOVED*

修改节点：ZKNodeEventType. *NODE\_UPDATED*

增加子节点：ZKNodeEventType. *CHILD\_ADDED*

修改子节点：ZKNodeEventType. *CHILD\_UPDATED*

删除子节点：ZKNodeEventType. *CHILD\_REMOVED*

* 实现监听接口ZKNodeListener

该接口有两个方法：

1. public boolean accept(ZKNodeEvent event)

只有以下事件发生时调用, 根据其执行结果决定onEvent方法是否调用

True：调用onEvent

False: 不调用onEvent

修改节点：ZKNodeEventType. *NODE\_UPDATED*

增加子节点：ZKNodeEventType. *CHILD\_ADDED*

修改子节点：ZKNodeEventType. *CHILD\_UPDATED*

删除子节点：ZKNodeEventType. *CHILD\_REMOVED*

1. public void onEvent(ZKNodeEvent event)

事件发生的具体业务逻辑在这个方法实现

* 监听接口代码示例

class NodeListener implements ZKNodeListener

{

@Override

public boolean accept(ZKNodeEvent event)

{

return true;

}

@Override

public void onEvent(ZKNodeEvent event)

{

ZKNodeEventSource source = (ZKNodeEventSource)event.getSource();

ZKData data = source.getData();

String path = data.getPath();

String value = new String(data.getData());

int version = data.getVersion();

ZKNodeEventType type = source.getType();

System.out.println("path=" + path);

System.out.println("value=" + value);

System.out.println("version=" + version);

System.out.println("type=" + type);

}

}

### 增加节点

* 在ZK上创建一个永久节点

ZKClientUtil clientUtil = **new** ZKClientUtil("subscribe.configerZKClient");

clientUtil.getClient().create("/zk/curator/path3", "1111", CreateMode.*CREATE\_MODE\_PERSISTENT*);

参数说明：

参数1 ：ZK上节点的PATH

参数2 : 对应节点的value值

参数3： 创建的节点类型,共有下面四种类型

永久节点：CreateMode.*CREATE\_MODE\_PERSISTENT*

临时节点：CreateMode.*CREATE\_MODE\_EPHEMERAL*

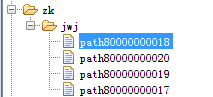
顺序的永久节点：CreateMode. *CREATE\_MODE\_PERSISTENT\_SEQUENTIAL*

临时的永久节点：CreateMode. *CREATE\_MODE\_EPHEMERAL\_SEQUENTIAL*

==============================================================================

每次执行下面的代码都会在/zk/jwj下生成不同的节点

clientUtil.getClient().create("/zk/jwj/path8", "cccccc", CreateMode.CREATE\_MODE\_PERSISTENT\_SEQUENTIAL);



* 监听ZK上节点增加事件：

clientUtil.getClient().registerListener("/zk/jwj/path4", new NodeListener(), ZKNodeEventType.NODE\_ADDED);

### 删除节点

* 在ZK上删除一个path为/zk/curator/path3的节点

clientUtil.getClient().delete("/zk/jwj/path8");

* 监听节点删除事件

clientUtil.getClient().registerListener ("/zk/curator/path3", new NodeListener(),

ZKNodeEventType.NODE\_REMOVED);

### 修改节点

* 调用方式同增加节点

如果该path不存在，就增加，已经增加就修改

clientUtil.getClient().create("/zk/curator/path3", "1111", CreateMode. *CREATE\_MODE\_PERSISTENT*);

* 监听/zk/curator/path3节点值修改的事件

clientUtil.getClient().registerListener("/zk/curator/path3", new NodeListener()，

ZKNodeEventType.NODE\_UPDATED);

### 查询节点和所有的子节点

* 查询某个节点的值

ZKNode node = clientUtil.getClient().getNode("/zk/curator/path1");

System.out.println(new String(node.getBytesValue()));

* 查询某个节点的所有子节点

List<ZKNode> nodes = clientUtil.getClient().getChildNodes("/zk/curator");

for (ZKNode node : nodes)

{

System.out.println(new String(node.getBytesValue()));

}

### 监听某一个节点下所有的子节点的增删改（只监控一级）

clientUtil.getClient().registerListener("/zk/curator/path4", new NodeListener()，

new ZKNodeEventType[] {ZKNodeEventType.CHILD\_ADDED, ZKNodeEventType.CHILD\_UPDATED,

ZKNodeEventType.CHILD\_REMOVED});

### 带ZK断链重连的监听

**ZKClientUtil** clientUtil = new ZKClientUtil(connectionName);

CallBackListener listen1 = new CallBackListener("/zk/curator/path4", new new NodeListener(), new ZKNodeEventType[] {ZKNodeEventType.CHILD\_ADDED, ZKNodeEventType.CHILD\_UPDATED, ZKNodeEventType.CHILD\_REMOVED});

List<**CallBackListener**> list = new ArrayList<CallBackListener>();

list.add(listen1);

clientUtil.setList(list);

clientUtil.getClient().registerListener(listen1.getPath(), listen1.getListener(), listen1.getType());

### Tree监听开发（监控节点下的所有子节点）

package com.huawei.commons.configuration;

import com.huawei.commons.configuration.tree.event.ZKTreeNodeEvent;

import com.huawei.commons.configuration.tree.event.ZKTreeNodeEventSource;

import com.huawei.commons.configuration.tree.event.ZKTreeNodeEventType;

import com.huawei.commons.configuration.tree.listener.ZKTreeNodeListener;

import com.huawei.commons.configuration.util.JsonFormater;

import com.huawei.commons.configuration.util.ZKClientUtil;

import com.huawei.commons.configuration.zookeeper.ZKData;

public class Main

{

public static void main(String[] args)

{

ConfigerServerStarter.init();

System.out.println("start success!");

ZKClientUtil clientUtil = new ZKClientUtil("subscribe.configerZKClient");

clientUtil.getClient().registerTreeListener("/test/aaa",

new Listener(),

new ZKTreeNodeEventType[] {ZKTreeNodeEventType.NODE\_ADDED, ZKTreeNodeEventType.NODE\_REMOVED,

ZKTreeNodeEventType.NODE\_UPDATED});

}

}

class Listener implements ZKTreeNodeListener

{

@Override

public boolean accept(ZKTreeNodeEvent event)

{

return true;

}

@Override

public void onEvent(ZKTreeNodeEvent event)

{

ZKTreeNodeEventSource source = (ZKTreeNodeEventSource)event.getSource();

ZKData data = source.getData();

String path = data.getPath();

String value = new String(data.getData());

value = JsonFormater.formatFromJsonStyle(value);

int version = data.getVersion();

ZKTreeNodeEventType type = source.getType();

System.out.println("path=" + path);

System.out.println("value=" + value);

System.out.println("version=" + version);

System.out.println("type=" + type);

}

}

## 定制场景

### 数据源动态刷新及断链告警

通过spring配置数据源刷新代理

<!--数据库刷新代理 -->

<bean id=*"dataSourceRefershProxy"*

class=*"com.huawei.commons.configuration.proxy.DataSourceRefershProxy"*

init-method=*"initConnectionListener"*>

<property name=*"awareClass"* value=*"com.huawei.jaguar.order.repository.datasource.BasicDataSourceWrapper"*></property>

<property name=*"acceptPaths"*>

<list>

<value>/config/ord/resource</value>

</list>

</property>

<property name=*"refershFilter"* ref=*"defaultZKRefershFilter"*></property>

<property name=*"context"* ref=*"configSpringContext"*></property>

<property name=*"connectionName"* value=*"order.configerZKClient"*></property>

<property name=*"keyPatter"* value=*"orddb\[[0-9]\*\].connect"*></property>

<property name=*"resourcePatter"* value=*"url|username|password|initialSize|maxActive|maxIdle|minIdle|maxWait|defaultAutoCommit|removeAbandoned|removeAbandonedTimeout"*></property>

<property name=*"template"* value=*"orddb[{0}].connect"*></property>

</bean>

属性说明：

|  |  |  |
| --- | --- | --- |
| 字段 | 类型 | 说明 |
| *initConnectionListener* | 初始化方法 | 依赖属性*connectionName* |
| *awareClass* |  |  |
| *acceptPaths* |  | 对应的path路径 |
| *refershFilter* |  |  |
| *context* | com.huawei.commons.configuration.spring.SpringContext | Spring上下文，commons已经定义，只需引用，无需定义，默认*configSpringContext* |
| *connectionName* |  | 保存在*\*zk-cluster.xml*的*cluster.Connector.name*中，如*order.configerZKClient* |
| *keyPatter* |  |  |
| *resourcePatter* |  | 资源名称，以’|’分割， |
| *template* |  |  |

目前只支持apache的DBCP连接池的动态刷新

### Spring属性动态刷新

Spring属性需要通过zk进行刷新，bean自动重新初始化，无需重启系统。

待刷新bean定义

<bean id=*"osstatuschecker"*

class=*"com.huawei.jaguar.commons.log.statistics.impl.OSStatusChecker"*

init-method=*"init"*>

<property name=*"cpuPeak"*>

<value>${cpuPeak}</value>

</property>

<property name=*"memPeak"*>

<value>${memPeak}</value>

</property>

<property name=*"memPercent"*>

<value>${memPercent}</value>

</property>

<property name=*"cpuPercent"*>

<value>${cpuPercent}</value>

</property>

</bean>

其中待舒心属性都通过${XXXX}实现，保证启动时可以通过placeholder模式生效。

配置zk刷新机制如下

<bean id=*"ereading.subscribe.subSpringPlaceholderRefershProxy"*

class=*"com.huawei.commons.configuration.proxy.SpringPlaceholderRefershProxy"*

init-method=*"initConnectionListener"*>

<property name=*"beanName"* value=*"osstatuschecker"*></property>

<property name=*"acceptPaths"*>

<list>

<value>/config/sub/serviceConfig/consumer</value>

</list>

</property>

<property name=*"keyPropertyMap"*>

<map>

<entry key=*"cpuPeak"* value=*"cpuPeak"* />

<entry key=*"memPeak"* value=*"memPeak"* />

<entry key=*"memPercent"* value=*"memPercent"* />

<entry key=*"cpuPercent"* value=*"cpuPercent"* />

</map>

</property>

<property name=*"context"* ref=*"configSpringContext"*></property>

<property name=*"connectionName"* value=*"subscribe.configerZKClient"*></property>

</bean>

属性配置说明如下

|  |  |  |
| --- | --- | --- |
| 字段 | 类型 | 说明 |
| *beanName* |  | 待刷新的beanid |
| *acceptPaths* |  | 对应的zkpath路径 |
| *keyPropertyMap* |  | Zkpath最后的节点与bean属性映射关系，其中key对应zk末节点的名称，value对应被监控bean的属性名称。 |
| *context* | com.huawei.commons.configuration.spring.SpringContext | Spring上下文，commons已经定义，只需引用，无需定义，默认*configSpringContext* |
| *connectionName* |  | 保存在*\*zk-cluster.xml*的*cluster.Connector.name*中，如*order.configerZKClient* |

注意：

为实现上述功能，各组还需要在spring文件中增加*org.springframework.beans.factory.config.PropertyPlaceholderConfigurer*的定义，如

<bean id=*"ereading.subscribe.subPropertyPlaceHolderConfigurer"*

class=*"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"*>

<property name=*"locations"*>

<list>

<value>classpath:subscribe.\*.properties</value>

<value>classpath:subscribe.\*.csv</value>

</list>

</property>

<property name=*"localOverride"* value=*"true"*/>

<property name=*"properties"* ref=*"ereading.subscribe.zkPropertiesFactoryBean"*/>

<property name=*"ignoreUnresolvablePlaceholders"* value=*"true"* />

</bean>

# Maven依赖

场景样例为组件插件的一类功能实现过程，包括场景描述、开发过程。

## 依赖说明

Commons组件规划了一个commons.denpengcy-management工程来规范所有基于commons组件开发jar包的使用，为了防止jar包冲突以及参考公司的优选库，commons.denpengcy-management里面的pom.xml定义了所有优选的jar和对应的版本。请尽量使用commons.denpengcy-management里面的jar。如果确实需要使用不在commons.denpengcy-management范围内的jar。请将jar和对应版本反馈给（朱元庆z00278802）

## 依赖示例

<properties>

<commons.version>V300R003C30B502-SNAPSHOT</commons.version>

</properties>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>com.huawei.jaguar</groupId> <artifactId>commons.denpengcy-management</artifactId>

<version>${commons.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependencies>

<dependency>

<groupId>com.huawei.jaguar</groupId>

<artifactId>commons.configuration</artifactId>

</dependency>

</dependencies>

# 附件

## 一些示范配置文件









# FAQ

插件过程中常见问题。