配置中心-BJF集成组件

开发与使用指南

# 介绍

配置中心与BJF开发框架集成的部分主要在配置文件加载实现上，使用统一的配置加载方式，以方便使用BJF开发框架与Spring开发框架的开发人员平滑的使用配置中心功能。

本文档包含两大部分，前面部分介绍如何使用配置中心的BJF集成组件，后面部分侧重基于开放的API进行二次开发。

# 使用介绍

1. 集成Spring的配置组件，提供统一的配置加载。

基本的配置示例如下：

<bean class="com.baidu.cc.spring.ConfigCenterPropertyPlaceholderConfigurer">

<property name="callbackInteval" value="500"></property>

<property name="ccServerUrl"

value="http://localhost:8080/rpc/ConfigService"></property>

<property name="ccUser" value="user"></property>

<property name="ccPassword" value="password"></property>

<property name="ccVersion" value="1"></property>

<property name="enableUpdateCallback" value="false"></property>

<property name="localResource" value="/local/localPropFile"></property>

</bean>

示例bean

<bean class=”test.Bean”>

<property name=”name” value=”${value}”/>

</bean>

参数属性说明：

|  |  |  |
| --- | --- | --- |
| 属性名称 | 说明 | 默认值 |
| callbackInteval | 配置变更回调监听时间间隔，单位是ms | 2000 |
| ccServerUrl | 配置中心服务url, 如果JVM参数中指定了该-DccServerUrl变量，将忽略本设置 |  |
| ccUser | 配置中心登录用户名如果JVM参数中指定了该-DccUser变量，将忽略本设置 |  |
| ccPassword | 配置中心登录密码如果JVM参数中指定了该-DccPassword变量，将忽略本设置 |  |
| ccVersion | 配置中心访问的版本号如果JVM参数中指定了该-DccVersion变量，将忽略本设置 |  |
| ccEnvId | 配置中心访问的运行环境id号如果JVM参数中指定了该-DccEvnId变量，将忽略本设置 |  |
| enableUpdateCallback | 是否开启变更回调功能 | False关闭 |
| localResource | 设置本地配置缓存文件 | 默认为null，则关闭本地缓存功能 |
| connectionTimeout | 连接配置中心服务器超时设置，单位ms | 2000 |
| readTimeout | 配置中心访问超时设置，单位ms | 2000 |
| logProperties | 是否在配置加载成功后，输出加载的配置内容 | False关闭 |
| projectName | 工程名称 |  |
| envName | 运行环境名称 |  |

属性的加载优先级说明：

配置的加载的方式最终都需要通过version id从配置中心服务器获取配置内容。

第一优先加载的是 projectName和envName属性，如果这两个都不会空，则会根据这两个值从配置中心获取最新的一份versionid.

第二优先加载的evnId，运行环境id, ，如果这个值不为空，则会根据这个值从配置中心获取最新的一份versionid.

最后加载的方式是通过指定version id来获取。

线上方案，为了防止配置服务的设置错误，建议线上的配置由OP通过JDK环境参数来指定配置中心的相关信息，完整的示例如下：

建议的配置方案：

OP配置内容，在JDK启动时，加入以下配置信息

Java –DccServerUrl=http://localhost:8080/configserver/rpc/ConfigServerService –DccUser= -DccPassword= -DccEnvName=

OP配置的内容为配置中心服务器地址, 登录的用户名和密码， 线上配置的运行环境名称。

RD配置内容

<bean class="com.baidu.cc.spring.ConfigCenterPropertyPlaceholderConfigurer">

<property name="callbackInteval" value="500"></property>

<property name="projectName" value="panggu\_support"></property>

<property name="enableUpdateCallback" value="true"></property>

<property name="localResource" value="/local/localPropFile"></property>

</bean>

每次运行都会取指定运行环境下的最新一份配置版本。

1. 配置导入功能

配置导入功能可以支持把本地加载的配置内容导入到配置中心服务器(只支持开发环境)

该功能可以很方便让原来不支持配置中心的项目快速使用配置中心应用。

配置如下：

<bean class=*"com.baidu.cc.spring.ConfigCenterPropertyExtractor"*>

<property name=*"ccVersion"* value=*"1"*></property>

<property name=*"importer"* ref=*" ConfigCenterPropertyPlaceholderConfigurer"*></property>

<property name=*"extractor"* ref=*"PropertyPlaceholderConfigurer"*></property>

</bean>

为了安全，只有针对空的版本id，才可以导入。同时线上的服务将关闭该导入的功能

导入只显示成一组。

ccVersion: 表示需要导入到的目标版本号id

importer: 指定需要导入的配置中心服务器。 ConfigCenterPropertyPlaceholderConfigurer

extractor: 需要导入的配置内容。支持Spring的*PropertyPlaceholderConfigurer*和BJF的*JndiPropertyPlaceholderConfigurer*

1. 配置变更监听实现

组件提供接口和注解两种方式来支持开发人员实现配置变量的事件接入

下面介绍了两个实现方式

1. 基于ConfigItemChangedCallable接口实现

* 实现ConfigItemChangedCallable接口
* 把实现的Bean配置到Spring容器中，即可

示例如下：

@Service("configChangedCallbackMock")

**public** **class** ConfigChangedCallbackMock **implements** ConfigItemChangedCallable {

/\* (non-Javadoc)

\* @see com.baidu.cc.interfaces.ConfigItemChangedCallable#changed(java.util.List)

\*/

**public** **void** changed(List<ChangedConfigItem> changedConfigItemList) {

**for** (ChangedConfigItem changedConfigItem : changedConfigItemList) {

//get change notify content

System.*out*.println("Changed configuration key: " + changedConfigItem.getKey());

System.*out*.println("original value: " + changedConfigItem.getOldValue());

System.*out*.println("new value: " + changedConfigItem.getNewValue());

}

}

}

1. 基于注解的实现

使用示例：

@Service("configChangedMock")

**public** **class** ConfigChangedMock {

@ConfigChangeNotifer

**public** **void** doChangedAll(List<ChangedConfigItem> changedConfigItems) {

System.*out*.println("doChangedAll call back");

System.*out*.println(changedConfigItems);

}

@ConfigChangeNotifer(keys = "key3,key2,key1")

**public** **void** doChangedByFilter(List<ChangedConfigItem> changedConfigItems) {

System.*out*.println("doChangedByFilter call back");

System.*out*.println(changedConfigItems);

}

}

@ConfigChangeNotifer只有一个参数，默认为空，表示接收所有的配置变更内容。如果指定，则可以根据”,”分隔多个key名称，可以只回调指定key的变更配置内容。

要支持注解的功能，需要增加以下Spring配置

<bean

class=*"com.baidu.bjf.beans.context.annotation.CommonAnnotationBeanPostProcessor"*>

<property name=*"callback"*>

<bean

class=*"com.baidu.cc.annotation.ConfigChangeNotiferAnnotationParser"*>

</bean>

</property>

</bean>

# POM配置说明

Pom配置说明：

<dependency>

<groupId>com.baidu</groupId>

<artifactId>configcenter-bjf</artifactId>

<version>1.0.0.0</version>

<type>jar</type>

<scope>compile</scope>

</dependency>

需要依赖BJF 1.0.1.0及以上版本。

# 动态加载组件开发说明

本节将介绍如何使用BJF与配置中心SDK集成的API快速开发动态配置加载组件的开发

开发方式有两种：

1. 继承BJF提供的*AbstractConfiguationReloadable*抽象类。下面是相关类图



继承*AbstractConfiguationReloadable*抽象类后，需要实现 propertyChangedCallback方法，回调用的一个List<ChangedConfigItem>参数。 ChangedConfigItem对象有三个属性，key表示对应修改的对象field名称, oldValue表示修改前的值, newValue表示修改后的值。

下面是一个实现示例：

@Override

**public** **void** propertyChanagedCallback(List<ChangedConfigItem> list) {

**if** (CollectionUtils.*isEmpty*(list)) {

**if** (*LOGGER*.isInfoEnabled()) {

*LOGGER*.info("propertyChanagedCallback ignore due to empty ChangedConfigItem list. ");

}

**return**;

}

//set the changed value back to dataSource

//build setXXX method name

PropertyUtils.*setPropertiesByReflection*(list, ComboPooledDataSource.**class**, dataSource);

}

把回调的参数设置到需要修改的对象中即可。

为了让这段配置生成，我们还需要配置配置中心的Key与我们对象属性名称的对应关，并且把当前需要修改的实例注入到Spring容器中，下面演示了C3p0数据源改造后支持动态配置加载的配置说明：

<bean id=*"parentDataSource"* class=*"com.baidu.bjf.datasource.c3p0.ReloadableComboPooledDataSourceDelegate"*

destroy-method=*"close"* abstract=*"true"*>

<property name=*"driverClass"*>

<value>${jdbc.driverClassName}</value>

</property>

<property name=*"jdbcUrl"*>

<value>${jdbc.url}</value>

</property>

<property name=*"user"*>

<value>${jdbc.username}</value>

</property>

<property name=*"password"*>

<value>${jdbc.password}</value>

</property>

<property name=*"maxPoolSize"*>

<value>${jdbc.maxPoolSize}</value>

</property>

<property name=*"minPoolSize"*>

<value>${jdbc.minPoolSize}</value>

</property>

<property name=*"initialPoolSize"*>

<value>${jdbc.initialPoolSize}</value>

</property>

<property name=*"idleConnectionTestPeriod"*>

<value>${jdbc.idleConnectionTestPeriod}

</value>

</property>

<property name=*"maxIdleTime"*>

<value>${jdbc.maxIdleTime}</value>

</property>

<property name=*"checkoutTimeout"*>

<value>${jdbc.checkoutTimeout}</value>

</property>

<property name=*"propertyMapping"*>

<map>

<entry key=*"jdbc.url"* value=*"jdbcUrl"*></entry>

<entry key=*"jdbc.maxPoolSize"* value=*"maxPoolSize"*></entry>

</map>

</property>

</bean>

上面的配置可以支持url和maxPoolSize两个属性的修改后动态加载的支持。

1. 加一个方式只实现*ConfiguationReloadable*接口。

针对这种方式的实现，由于不能直接使用抽象中的提供的配置过滤与对应关系处理方法，可以直接使用*AbstractConfiguationReloadable*抽象类的静态方法处理。下面是BJF rmi客户端的代码实现片段

**public** **class** ReloadableRmiProxyFactoryBean **extends**

MultiInterfacesRmiProxyFactoryBean **implements** ConfiguationReloadable {

//省略常量定义

/\*\*

\* Property name and configuration item name mapping

\*/

**private** Map<String, String> mapping;

/\*\*

\* get property mapping.

\* **@return** the mapping key is configuration item name, value is property name

\*/

**protected** Map<String, String> getMapping() {

**return** mapping;

}

/\*\*

\* set property name and configuration item name mapping.

\*

\* **@param** mapping key is configuration item name, value is property name

\*/

**public** **void** setPropertyMapping(Map<String, String> mapping) {

**this**.mapping = mapping;

}

/\*\*

\* 配置中心值变更回调接口

\* ChangedConfigItem有三个属性 key/oldValue/newValue

\* **@param** changedConfigItemList 发生变更的key列表

\*/

**public** **void** changed(List<ChangedConfigItem> changedConfigItemList) {

//calculate mapping property

List<ChangedConfigItem> filteredList = AbstractConfiguationReloadable.*getPropertyChanged*(

mapping, changedConfigItemList);

//do call back here

**if** (CollectionUtils.*isNotEmpty*(filteredList)) {

**boolean** toRefreshConnector = **false**;

//do call back here

**for** (ChangedConfigItem changedConfigItem : filteredList) {

String propName = changedConfigItem.getKey();

**if** (*LOGGER*.isInfoEnabled()) {

*LOGGER*.info("ReloadableSmartRmiProxyFactoryBean do reload property '" + propName + "'");

}

**boolean** refreshConnector = setProperty(propName, changedConfigItem.getNewValue());

**if** (refreshConnector) {

toRefreshConnector = **true**;

}

}

**if** (toRefreshConnector) {

**try** {

//to refresh connector

refreshAndRetry(**null**);

} **catch** (Throwable e) {

//here will must throw null point exception.

**if** (!(e **instanceof** NullPointerException)) {

*LOGGER*.error(e.getMessage(), e);

}

}

}

}

}

/\*\*

\* set property value by property name.

\*

\* **@param** propName property name

\* **@param** value target value to set

\* **@return** if need refresh connector return true.

\*/

**protected** **boolean** setProperty(String propName, String value) {

Boolean refreshConnector = *SUPPORTED\_RELOADABLE\_PROPERTIES*.get(propName);

**if** (refreshConnector == **null**) {

//ignore it

**if** (*LOGGER*.isInfoEnabled()) {

*LOGGER*.info("property '" + propName + "' is not support reload in ReloadableSmartRmiProxyFactoryBean");

}

**return** **false**;

}

**if** (*REFRESH\_STUB\_ON\_CONNECT\_FAILURE*.equals(propName)) {

setRefreshStubOnConnectFailure(Boolean.*valueOf*(value));

} **else** **if** (*CACHE\_STUB*.equals(propName)) {

setCacheStub(Boolean.*valueOf*(value));

} **else** **if** (*LOOKUP\_STUB\_ON\_STARTUP*.equals(propName)) {

setLookupStubOnStartup(Boolean.*valueOf*(value));

} **else** **if** (*SERVICE\_URL*.equals(propName)) {

setServiceUrl(value);

} **else** **if** (*OP\_TIMEOUT*.equals(propName)) {

setOpTimeout(Integer.*valueOf*(value));

}

**return** refreshConnector;

}

}

# BJF配置动态加载组件使用说明

本次升级时，针对BJF提供的组件也实现了配置动态加载功能的支持。