序列化器和spi机制

序列化器

支持多种序列化器

JSON序列化器 Java

```java
package com.yupi.yurpc.serializer;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.yupi.yurpc.model.RpcRequest;
import com.yupi.yurpc.model.RpcResponse;

import java.io.IOException;

/**
 * Json 序列化器
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @learn <a href="https://codefather.cn">编程宝典</a>
 * @from <a href="https://yupi.icu">编程导航知识星球</a>
 */
public class JsonSerializer implements Serializer {
    private static final ObjectMapper OBJECT_MAPPER = new ObjectMapper();

    @Override
    public <T> byte[] serialize(T obj) throws IOException {
        return OBJECT_MAPPER.writeValueAsBytes(obj);
    }

    @Override
    public <T> T deserialize(byte[] bytes, Class<T> classType) throws IOException {
        T obj = OBJECT_MAPPER.readValue(bytes, classType);
        if (obj instanceof RpcRequest) {
            return handleRequest((RpcRequest) obj, classType);
        }
        if (obj instanceof RpcResponse) {
            return handleResponse((RpcResponse) obj, classType);
        }
        return obj;
    }

    /**
     * 由于 Object 的原始对象会被擦除，导致反序列化时会被作为 LinkedHashMap 无法转换成原始对象，因此这里做了特殊处理
     *
     * @param rpcRequest rpc 请求
     * @param type       类型
     * @return {@link T}
     * @throws IOException IO异常
     */
```

```
    private <T> T handleRequest(RpcRequest rpcRequest, Class<T> type) throws IOException {
        Class<?>[] parameterTypes = rpcRequest.getParameterTypes();
        Object[] args = rpcRequest.getArgs();

        // 循环处理每个参数的类型
        for (int i = 0; i < parameterTypes.length; i++) {
            Class<?> clazz = parameterTypes[i];
            // 如果类型不同，则重新处理一下类型
            if (!clazz.isAssignableFrom(args[i].getClass())) {
                byte[] argBytes = OBJECT_MAPPER.writeValueAsBytes(args[i]);
                args[i] = OBJECT_MAPPER.readValue(argBytes, clazz);
            }
        }
        return type.cast(rpcRequest);
    }

    /**
     * 由于 Object 的原始对象会被擦除，导致反序列化时会被作为 LinkedHashMap 无法转换成原始对象，因此这里做了特殊处理
     *
     * @param rpcResponse rpc 响应
     * @param type        类型
     * @return {@link T}
     * @throws IOException IO异常
     */
    private <T> T handleResponse(RpcResponse rpcResponse, Class<T> type) throws IOException {
        // 处理响应数据
        byte[] dataBytes = OBJECT_MAPPER.writeValueAsBytes(rpcResponse.getData());
        rpcResponse.setData(OBJECT_MAPPER.readValue(dataBytes, rpcResponse.getDataType()));
        return type.cast(rpcResponse);
    }
}
```

Kryo 序列化器 Java

```java
package com.yupi.yurpc.serializer;

import com.esotericsoftware.kryo.Kryo;
import com.esotericsoftware.kryo.io.Input;
import com.esotericsoftware.kryo.io.Output;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;

/**
 * Kryo 序列化器
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @learn <a href="https://codefather.cn">编程宝典</a>
 * @from <a href="https://yupi.icu">编程导航知识星球</a>
 */
public class KryoSerializer implements Serializer {
    /**
     * kryo 线程不安全，使用 ThreadLocal 保证每个线程只有一个 Kryo
     */
    private static final ThreadLocal<Kryo> KRYO_THREAD_LOCAL = ThreadLocal.withInitial(() -> {
        Kryo kryo = new Kryo();
        // 设置动态动态序列化和反序列化类，不提前注册所有类（可能有安全问题）
        kryo.setRegistrationRequired(false);
        return kryo;
    });

    @Override
    public <T> byte[] serialize(T obj) {
        ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
        Output output = new Output(byteArrayOutputStream);
        KRYO_THREAD_LOCAL.get().writeObject(output, obj);
        output.close();
        return byteArrayOutputStream.toByteArray();
    }

    @Override
    public <T> T deserialize(byte[] bytes, Class<T> classType) {
        ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(bytes);
        Input input = new Input(byteArrayInputStream);
        T result = KRYO_THREAD_LOCAL.get().readObject(input, classType);
        input.close();
```

```
        return result;
    }
}

```

Hessian 序列化器 Java

```java
package com.yupi.yurpc.serializer;

import com.caucho.hessian.io.HessianInput;
import com.caucho.hessian.io.HessianOutput;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

/**
 * Hessian 序列化器
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @learn <a href="https://codefather.cn">编程宝典</a>
 * @from <a href="https://yupi.icu">编程导航知识星球</a>
 */
public class HessianSerializer implements Serializer {
    @Override
    public <T> byte[] serialize(T object) throws IOException {
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        HessianOutput ho = new HessianOutput(bos);
        ho.writeObject(object);
        return bos.toByteArray();
    }

    @Override
    public <T> T deserialize(byte[] bytes, Class<T> tClass) throws IOException {
        ByteArrayInputStream bis = new ByteArrayInputStream(bytes);
        HessianInput hi = new HessianInput(bis);
        return (T) hi.readObject(tClass);
    }
}

```

SPI机制*

有了spi机制，我们就可以动态增加的序列化器了。

1. 创建一个序列化器名称常量
2. 在全局配置类里添加序列化配置属性
3. 实现序列化器接口实现序列化器
4. 在META-INFO/services/接口的类路径名称文件，将实现的序列化器全类名写到文件里。

代码实现*

使用流程*

1. 先创建几个常量，用来存储已加载的类的ConcurrentHashMap<String,<String,Class<?>>>，对象实例缓存ConcurrentHashMap<String,Object>,SPI机制路径，系统序列化器路径，用户自定义序列化器路径，所有需要实现spi的接口信息集合。
2. rpc初始化时，spi加载所有实现类。
    a. 根据路径读取每个文件的key和value
    b. 使用反射，Class类信息，放入 **存储已加载的类的ConcurrentHashMap**
    c. 继续加载下边的类
3. 获取某个序列化器实现
    a. 参数是接口信息和序列化器常量
    b. 先根据接口信息获取**存储已加载的类的ConcurrentHashMap**，获取对应的class文件
    c. 先判断 **对象实例缓存ConcurrentHashMap<String,Object>**有没有对象，如果有直接返回，如果没有，使用反射创建。
4. 代码实现

Java

```java
package com.ayouok.spi;

import cn.hutool.core.io.resource.ResourceUtil;
import com.ayouok.serializer.Serializer;
import lombok.extern.slf4j.Slf4j;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

/**
 * SPI 加载器（支持键值对映射）
 * @author ayouokk
 */
@Slf4j
public class SpiLoader {

    /**
     * 存储已加载的类：接口名 => (key => 实现类)
     */
    private static final Map<String, Map<String, Class<?>>> LOADER_MAP = new ConcurrentHashMap<>();

    /**
     * 对象实例缓存（避免重复 new），类路径 => 对象实例，单例模式
     */
    private static final Map<String, Object> INSTANCE_CACHE = new ConcurrentHashMap<>();

    /**
     * 系统 SPI 目录
     */
    private static final String RPC_SYSTEM_SPI_DIR = "META-INF/rpc/system/";

    /**
     * 用户自定义 SPI 目录
     */
    private static final String RPC_CUSTOM_SPI_DIR = "META-INF/rpc/custom/";

    /**
```

```
     * 扫描路径
     */
    private static final String[] SCAN_DIRS = new String[]{RPC_SYSTEM_SPI_DIR, RPC_CUSTOM_SPI_DIR};

    /**
     * 动态加载的类列表
     */
    private static final List<Class<?>> LOAD_CLASS_LIST = Collections.singletonList(Serializer.class);

    /**
     * 加载所有类型
     */
    public static void loadAll() {
        log.info("加载所有 SPI");
        for (Class<?> aClass : LOAD_CLASS_LIST) {
            load(aClass);
        }
    }

    /**
     * 获取某个接口的实例
     *
     * @param tClass
     * @param key
     * @param <T>
     * @return
     */
    public static <T> T getInstance(Class<?> tClass, String key) {
        String tClassName = tClass.getName();
        Map<String, Class<?>> keyClassMap = LOADER_MAP.get(tClassName);
        if (keyClassMap == null) {
            throw new RuntimeException(String.format("SpiLoader 未加载 %s 类型", tClassName));
        }
        if (!keyClassMap.containsKey(key)) {
            throw new RuntimeException(String.format("SpiLoader 的 %s 不存在 key=%s 的类型", tClassName, key));
        }
        // 获取到要加载的实现类型
        Class<?> implClass = keyClassMap.get(key);
        // 从实例缓存中加载指定类型的实例
        String implClassName = implClass.getName();
        if (!INSTANCE_CACHE.containsKey(implClassName)) {
            try {
                INSTANCE_CACHE.put(implClassName, implClass.getDeclaredConstructor().newInstance());
```

```
            } catch (Exception e) {
                String errorMsg = String.format("%s 类实例化失败", implClassName);
                throw new RuntimeException(errorMsg, e);
            }
        }
        Object instance = INSTANCE_CACHE.get(implClassName);
        if (!tClass.isInstance(instance)){
            throw new RuntimeException(String.format("实例 %s 不是 %s 类型", implClassName, tClassName));
        }
        return (T) instance;
    }

    /**
     * 加载某个类型
     *
     * @param loadClass
     * @throws IOException
     */
    public static Map<String, Class<?>> load(Class<?> loadClass) {
        log.info("加载类型为 {} 的 SPI", loadClass.getName());
        // 扫描路径，用户自定义的 SPI 优先级高于系统 SPI
        Map<String, Class<?>> keyClassMap = new HashMap<>(8);
        for (String scanDir : SCAN_DIRS) {
            List<URL> resources = ResourceUtil.getResources(scanDir + loadClass.getName());
            // 读取每个资源文件
            for (URL resource : resources) {
                try {
                    //输入流
                    InputStreamReader inputStreamReader = new InputStreamReader(resource.openStream());
                    //读取
                    BufferedReader bufferedReader = new BufferedReader(inputStreamReader);
                    String line;
                    //按行读取
                    while ((line = bufferedReader.readLine()) != null) {
                        String[] strArray = line.split("=");
                        if (strArray.length > 1) {
                            String key = strArray[0];
                            String className = strArray[1];
                            keyClassMap.put(key, Class.forName(className));
                        }
                    }
                } catch (Exception e) {
```

```
                    log.error("spi resource load error", e);
                }
            }
        }
        LOADER_MAP.put(loadClass.getName(), keyClassMap);
        return keyClassMap;
    }
}
```

注：代码中读取文件是使用ResourceUtil，不是通过文件路径获取的，因为rpc框架被引入后，不能通过路径读取

5. 修改序列化器工厂，动态的获取序列化器

Java

```java
package com.yupi.yurpc.serializer;

import com.yupi.yurpc.spi.SpiLoader;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

/**
 * 序列化器工厂（用于获取序列化器对象）
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @learn <a href="https://codefather.cn">编程宝典</a>
 * @from <a href="https://yupi.icu">编程导航知识星球</a>
 */
public class SerializerFactory {

    static {
        SpiLoader.load(Serializer.class);
    }

    /**
     * 默认序列化器
     */
    private static final Serializer DEFAULT_SERIALIZER = new JdkSerializer
();

    /**
     * 获取实例
     *
     * @param key
     * @return
     */
    public static Serializer getInstance(String key) {
        return SpiLoader.getInstance(Serializer.class, key);
    }

}

```