
回到首页

芋道源码 —— 知识星球

我是一段不羁的公告！
记得给艿艿这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2019-02-15
Dubbo

精尽 Dubbo 源码分析 —— 序列化（一）之总体实现

本文基于 Dubbo 2.6.1 版本，望知悉。

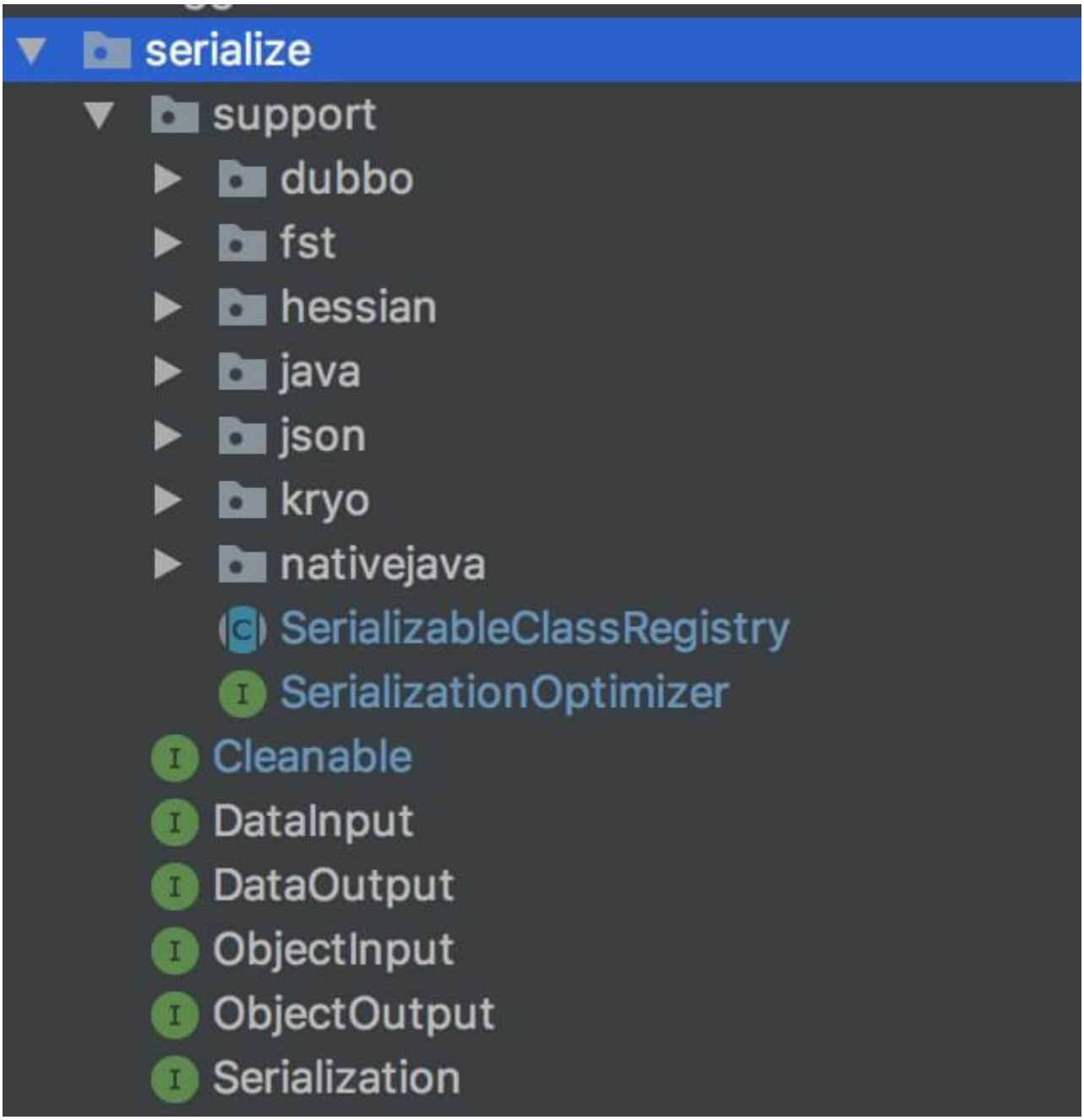
1. 概述

从本文开始，我们来分享 Dubbo 的序列化的实现。在 《Dubbo 开发指南 —— 序列化扩展》 ，对序列化定义如下：

> 将对象转成字节流，用于网络传输，以及将字节流转为对象，用于在收到字节流数据后还原成对象。
>
> 所以，序列化实际上包含两部分。
> 有一个概念，我们需要强调一下：协议和序列化，是两件事情。举个例子，HTTP 是一种协议，可以有 XML 和 JSON 等等序列化（ 数据交换 ）的方式。同时，XML 和 JSON 不仅仅可以用在 HTTP 协议，也可以用在 HTTPS 等等协议中。所以，协议和序列化不是包含的关系，而是组合的关系。

序列化在 dubbo-common 项目的 serialize 模块实现。代码结构如下图：

> 在最新版本的 Dubbo 项目中，serialize 模块，已经独立成 dubbo-serialize 项目。



最外层，定义了 API 接口。
`support` 包，提供了多种序列化的实现。

2. API 定义

API 接口，类图如下：

2.1 Serialization

com.alibaba.dubbo.common.serialize.Serialization ， 序列化接口。代码如下：

```
@SPI("hessian2")
public interface Serialization {

    /**
     * get content type id
     *
     * 获得内容类型编号
     *
     * @return content type id
     */
    byte getContentTypeId();

    /**
     * get content type
     *
     * 获得内容类型名
     *
     * @return content type
     */
    String getContentType();

    /**
     * create serializer
     *
     * 创建 ObjectOutput 对象，序列化输出到 OutputStream
     *
     * @param url URL
     * @param output 输出流
     * @return serializer
     * @throws IOException 当发生 IO 异常时
     */
    @Adaptive
    ObjectOutput serialize(URL url, OutputStream output) throws IOException;

    /**
     * create deserializer
     *
     * 创建 ObjectInput 对象，从 InputStream 反序列化
     *
     * @param url URL
     * @param input 输入流
     * @return deserializer
     * @throws IOException 当发生 IO 异常时
     */
    @Adaptive
    ObjectInput deserialize(URL url, InputStream input) throws IOException;

}
```

@SPI("hessian2") 注解，Dubbo SPI 拓展点，默认为 "hessian2" ，即未配置情况下，使用 Hessian 进行序列化和反序列化 。

#getContentTypeId(),#getContentType() 方法，获得内容类型编号和名字。

#serialize(...),#deserialize(...) 方法， 具体看注释。

- 虽然添加了 @Adaptive 注解， 但是实际上，不使用 Dubbo SPI Adaptive 机制，而是代码中，直接获取。例如：

```
// CodecSupport.java
public static Serialization getSerialization(URL url) {
    return ExtensionLoader.getExtensionLoader(Serialization.class).getExtension(
            url.getParameter(Constants.SERIALIZATION_KEY, Constants.DEFAULT_REMOTING_SERIALIZATION));
}
```

  - x
- Serialization 实现类，实现这两个方法，创建对应的 ObjectOutput 和 ObjectInput 实现类的对象。

2.2 DataInput

com.alibaba.dubbo.common.serialize.DataInput ， 数据输入接口。方法如下：

```
boolean readBool() throws IOException;

byte readByte() throws IOException;
short readShort() throws IOException;
int readInt() throws IOException;
long readLong() throws IOException;
float readFloat() throws IOException;
double readDouble() throws IOException;

String readUTF() throws IOException;

byte[] readBytes() throws IOException;
```

从 InputStream 中，读取基本类型的数据。

2.2.1 ObjectInput

com.alibaba.dubbo.common.serialize.ObjectInput ， 实现 DataInput 接口，对象输入接口。方法如下：

```
Object readObject() throws IOException, ClassNotFoundException;
<T> T readObject(Class<T> cls) throws IOException, ClassNotFoundException;
<T> T readObject(Class<T> cls, Type type) throws IOException, ClassNotFoundException;
```

在 DataInput 的基础上，增加读取对象的数据。

2.3 DataOutput

DataOutput 和 DataInput 相反。

com.alibaba.dubbo.common.serialize.DataOutput ， 数据输出接口。方法如下：

```
void writeBool(boolean v) throws IOException;
```

```
void writeByte(byte v) throws IOException;
void writeShort(short v) throws IOException;
void writeInt(int v) throws IOException;
void writeLong(long v) throws IOException;
void writeFloat(float v) throws IOException;
void writeDouble(double v) throws IOException;

void writeUTF(String v) throws IOException;

void writeBytes(byte[] v) throws IOException;
void writeBytes(byte[] v, int off, int len) throws IOException;

// Flush buffer.
void flushBuffer() throws IOException;
```

向 InputStream 中，写入基本类型的数据。

2.3.1 ObjectOutput

com.alibaba.dubbo.common.serialize.ObjectOutput ，实现 DataOutput 接口，对象输出接口。方法如下：

```
void writeObject(Object obj) throws IOException;
```

在 DataOutput 的基础上，增加写入对象的数据。

2.4 Cleanable

com.alibaba.dubbo.common.serialize.Cleanable ，清理接口。方法如下：

```
void cleanup();
```

部分 Serialize 实现类，完成序列化或反序列化，需要做清理。通过实现该接口，执行清理的逻辑。

2.5 Optimizer 相关

2.5.1 SerializationOptimizer

com.alibaba.dubbo.common.serialize.support.SerializationOptimizer ，序列化优化器接口。方法如下：

```
public interface SerializationOptimizer {

    /**
     * @return 需要使用优化的类的集合
     */
    Collection<Class> getSerializableClasses();

}
```

在 Kryo 、FST 中，支持配置需要优化的类。业务系统中，可以实现自定义的 SerializationOptimizer 子类，进行配置。当然，使用文件也是一个选择，Dubbo 在实现考虑取舍的原因如下：

> FROM 类注释
>
> This class can be replaced with the contents in config file, but for now I think the class is easier to write
>
> 这个类可以替换为配置文件中的内容，但是现在我认为这个类更容易编写。

2.5.2 SerializableClassRegistry

com.alibaba.dubbo.common.serialize.support.SerializableClassRegistry ，序列化优化类的注册表。代码如下：

```
public abstract class SerializableClassRegistry {

    private static final Set<Class> registrations = new LinkedHashSet<Class>();

    /**
     * only supposed to be called at startup time
     */
    public static void registerClass(Class clazz) {
        registrations.add(clazz);
    }

    public static Set<Class> getRegisteredClasses() {
        return registrations;
    }

}
```

#registerClass(clazz) 静态方法，注册。在 SerializationOptimizer#getSerializableClasses() 方法，获得的类的集合，会注册到 SerializableClassRegistry 中。
#getRegisteredClasses() 静态方法，获得。在 Kryo 、FST 中，调用该方法，获得需要使用优化的类的集合。

2.5.3 初始化序列化优化器

在 DubboProtocol#optimizeSerialization() 方法中，初始化序列化优化器。代码如下：

```
/**
 * 已初始化的 SerializationOptimizer 实现类名的集合
 */
private final Set<String> optimizers = new ConcurrentHashSet<String>();

private void optimizeSerialization(URL url) throws RpcException {
    // 获得 `"optimizer"` 配置项
    String className = url.getParameter(Constants.OPTIMIZER_KEY, "");
    if (StringUtils.isEmpty(className) || optimizers.contains(className)) { // 已注册
        return;
    }

    logger.info("Optimizing the serialization process for Kryo, FST, etc...");
```

```
        try {
            // 加载 SerializationOptimizer 实现类
            Class clazz = Thread.currentThread().getContextClassLoader().loadClass(className);
            if (!SerializationOptimizer.class.isAssignableFrom(clazz)) {
                throw new RpcException("The serialization optimizer " + className + " isn't an instance of " + Serializat
            }

            // 创建 SerializationOptimizer 对象
            SerializationOptimizer optimizer = (SerializationOptimizer) clazz.newInstance();
            if (optimizer.getSerializableClasses() == null) {
                return;
            }

            // 注册到 SerializableClassRegistry 中
            for (Class c : optimizer.getSerializableClasses()) {
                SerializableClassRegistry.registerClass(c);
            }

            // 添加到 optimizers 中
            optimizers.add(className);
        } catch (ClassNotFoundException e) {
            throw new RpcException("Cannot find the serialization optimizer class: " + className, e);
        } catch (InstantiationException e) {
            throw new RpcException("Cannot instantiate the serialization optimizer class: " + className, e);
        } catch (IllegalAccessException e) {
            throw new RpcException("Cannot instantiate the serialization optimizer class: " + className, e);
        }
    }
```

胖友，直接看代码注释。

3. Dubbo 实现

在 《精尽 Dubbo 源码分析 —— 序列化（二）之 Dubbo 实现》 中，详细解析。

4. Kryo 实现

在 《精尽 Dubbo 源码分析 —— 序列化（三）之 Kryo 实现》 中，详细解析。

5. FST 实现

> FST fast-serialization 是重新实现的 Java 快速对象序列化的开发包。序列化速度更快（2-10倍）、体积更小，而且兼容 JDK 原生的序列化。要求 JDK 1.7 支持。

文末，有性能相关测试的分享。

4.1 FstFactory

com.alibaba.dubbo.common.serialize.support.fst.FstFactory ，FST 工厂。代码如下：

```
public class FstFactory {

    /**
     * 单例
     */
    private static final FstFactory factory = new FstFactory();

    /**
     * 配置对象
     */
    private final FSTConfiguration conf = FSTConfiguration.createDefaultConfiguration();

    public static FstFactory getDefaultFactory() {
        return factory;
    }

    public FstFactory() {
        // 注册
        for (Class clazz : SerializableClassRegistry.getRegisteredClasses()) {
            conf.registerClass(clazz);
        }
    }

    public FSTObjectOutput getObjectOutput(OutputStream outputStream) {
        return conf.getObjectOutput(outputStream);
    }

    public FSTObjectInput getObjectInput(InputStream inputStream) {
        return conf.getObjectInput(inputStream);
    }

}
```

`factory` 静态属性，单例。

`conf` 属性，FST 配置对象。在构造方法中，将 SerializableClassRegistry 注册表需要使用优化的类，注册到 FSTConfiguration 中。`SerializableClassRegistry#registerClass(Class ... c)` 方法，注释如下：

```
    /**
     *
     * Preregister a class (use at init time). This avoids having to write class names.
     * Its a very simple and effective optimization (frequently > 2 times faster for small objects).
     * 预注册一个类(在初始化时使用)。这样可以避免编写类名。
     * 它是一种非常简单有效的优化(对于小对象来说，通常是>的2倍)。
     *
     * Read and write side need to have classes preregistered in the exact same order.
     * 客户端和服务端需要预先以完全相同的顺序注册。
     *
     *
     * The list does not have to be complete. Just add your most frequently serialized classes here
     * to get significant gains in speed and smaller serialized representation size.
     *
     * 这个列表并不一定要完整。只需在这里添加最常见的序列化类，以获得速度和较小的序列化表示大小的显著提高。
     */
```

`#getObjectOutput()` 方法，获得 `org.nustaq.serialization.FSTObjectOutput` 对象，被 FstObjectOutput

调用。

#getObjectInput() 方法，获得 org.nustaq.serialization.FSTObjectInput 对象，被 FstObjectInput 调用。

4.2 FstSerialization

com.alibaba.dubbo.common.serialize.support.fst.FstSerialization ，实现 Serialization 接口，FST 序列化实现类。代码如下：

```
public class FstSerialization implements Serialization {

    @Override
    public byte getContentTypeId() {
        return 9;
    }

    @Override
    public String getContentType() {
        return "x-application/fst";
    }

    @Override
    public ObjectOutput serialize(URL url, OutputStream out) throws IOException {
        return new FstObjectOutput(out);
    }

    @Override
    public ObjectInput deserialize(URL url, InputStream is) throws IOException {
        return new FstObjectInput(is);
    }

}
```

"x-application/fst" ，类似 HTTP 协议 的 Content-Types 的 Header 。在 『6. JSON 实现』 类中，返回的是 "text/json" 。

4.3 FstObjectInput

com.alibaba.dubbo.common.serialize.support.fst.FstObjectInput ，实现 ObjectInput 接口，FST 对象输入实现类。

构造方法

```
private FSTObjectInput input;

public FstObjectInput(InputStream inputStream) {
    input = FstFactory.getDefaultFactory().getObjectInput(inputStream);
}
```

input 属性，调用 FstFactory#getObjectInput(inputStream) 方法，获得。

实现方法

每个实现方法，直接调用 FSTObjectInput 对应的方法。比较特殊的是，#readBytes() 方法，代码如下：

```
@Override
public byte[] readBytes() throws IOException {
    int len = input.readInt();
    // 数组为空
    if (len < 0) {
        return null;
    // 数组为零
    } else if (len == 0) {
        return new byte[]{};
    // 数组 > 0
    } else {
        byte[] b = new byte[len];
        input.readFully(b);
        return b;
    }
}
```

[字节数组长度，字节数组内容]

4.4 FstObjectOutput

com.alibaba.dubbo.common.serialize.support.fst.FstObjectOutput ，实现 ObjectOutput 接口，FST 对象输出实现类。

构造方法

```
private FSTObjectOutput output;

public FstObjectOutput(OutputStream outputStream) {
    output = FstFactory.getDefaultFactory().getObjectOutput(outputStream);
}
```

output 属性，调用 FstFactory#getObjectInput(outputStream) 方法，获得。

实现方法

每个实现方法，直接调用 FSTObjectInput 对应的方法。比较特殊的是，#writeBytes(byte[] v) 方法，代码如下：

```
@Override
public void writeBytes(byte[] v) throws IOException {
    // 空，写入 -1
    if (v == null) {
        output.writeInt(-1);
    // 有数组
    } else {
        writeBytes(v, 0, v.length);
    }
}
```

[字节数组长度, 字节数组内容]

6. JSON 实现

基于 FastJSON 实现。

> fastjson 是一个性能很好的 Java 语言实现的 JSON 解析器和生成器，来自阿里巴巴的工程师开发。
>
> 主要特点：
>
> > 快速FAST（比其它任何基于Java的解析器和生成器更快，包括 jackson）
> > 强大（支持普通JDK类包括任意Java Bean Class、Collection、Map、Date 或 enum）
> > 零依赖（没有依赖其它任何类库除了JDK）

代码比较简单，和 『5. FST 实现』 类似，胖友自己查看：

> com.alibaba.dubbo.common.serialize.support.json.FastJsonSerialization
> com.alibaba.dubbo.common.serialize.support.json.FastJsonObjectInput
> com.alibaba.dubbo.common.serialize.support.json.FastJsonObjectOutput

需要注意的是，`FastJsonObjectOutput#writeObject(Object)` 方法的实现，代码如下：

```
@Override
public void writeObject(Object obj) throws IOException {
    SerializeWriter out = new SerializeWriter();
    // 序列化，写入对象
    JSONSerializer serializer = new JSONSerializer(out);
    serializer.config(SerializerFeature.WriteEnumUsingToString, true); // 枚举转字符串
    serializer.write(obj);
    // 写到，输出流
    out.writeTo(writer);
    out.close(); // for reuse SerializeWriter buf
    writer.println(); // 换行
    writer.flush();
}
```

7. Hessian2 实现

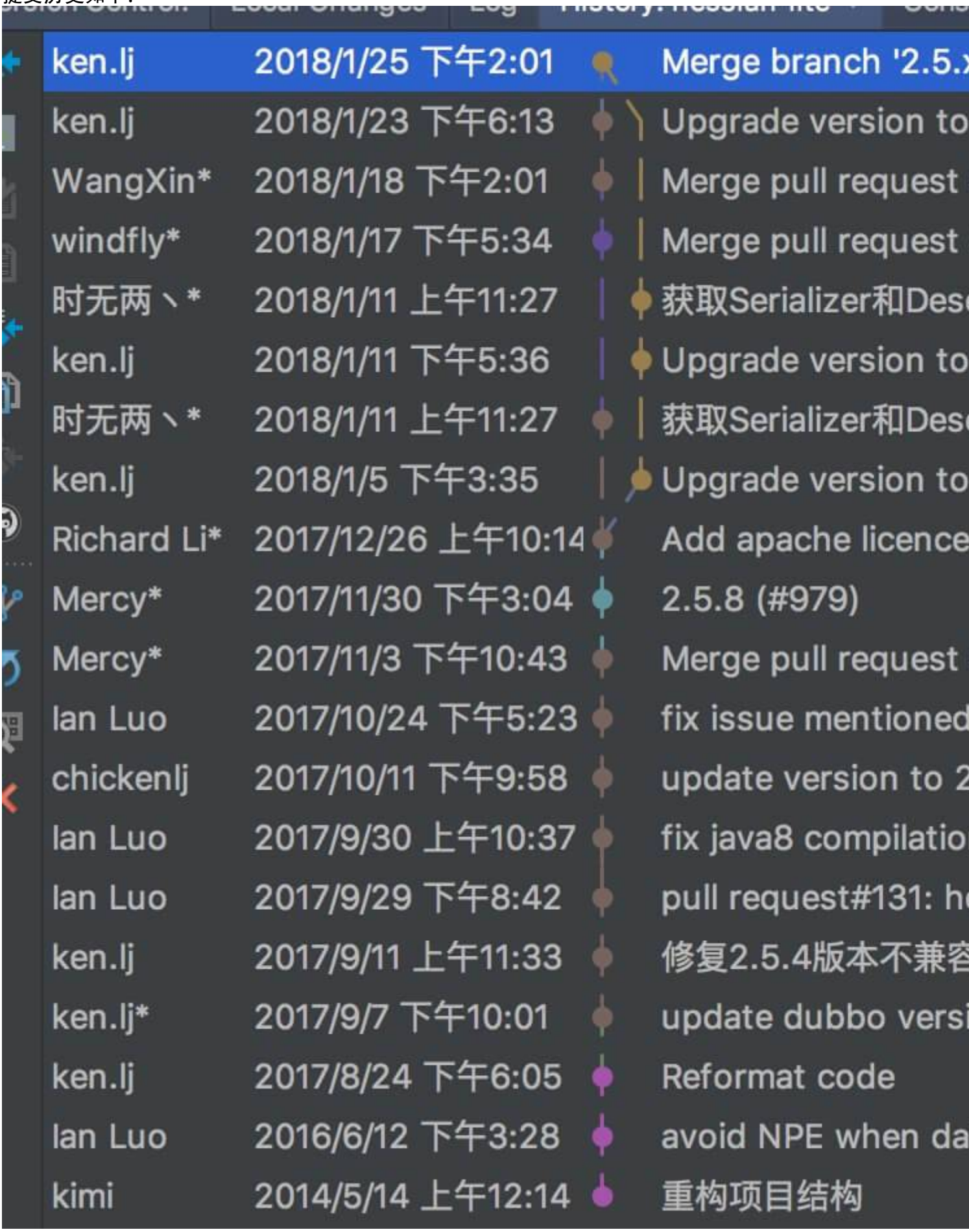
> 和其他 Web 服务的实现框架不同的是，Hessian 是一个使用二进制 Web 服务协议的框架，它的好处在于免除了一大堆附加的API包，例如 XML 的处理之类的 jar 包，这也就是为什么说它是一个轻量级的 Web 服务实现框架的原因，这个原因还在于手机上的应用程序可以通过 Hessian 提供的 API 很方便的访问 Hessian 的 Web 服务。

从介绍中，我们可以看到，Hessian 自己有自己的序列化的实现。但是，Hessian 在实现上，存在一些 Bug 和需要性能优化的点。例如：

> BigDecimal 的反序列化
>
> 使用 Hessian 序列化包含 BigDecimal 字段的对象时会导致其值一直为0，不注意这个bug会导致很大的问题，在最新的4.0.51版本仍然可以复现。解决方案也很简单，指定

BigDecimal 的序列化器即可。

所以 Dubbo 维护了自己的 hessian-lite ，对 Hessian 2 的 序列化 部分的精简、改进、BugFix 。
提交历史如下：



代码比较简单，和 『5. FST 实现』 类似，胖友自己查看：

- com.alibaba.dubbo.common.serialize.support.hessian.Hessian2SerializerFactory
- com.alibaba.dubbo.common.serialize.support.hessian.Hessian2Serialization
- com.alibaba.dubbo.common.serialize.support.hessian.Hessian2ObjectInput
- com.alibaba.dubbo.common.serialize.support.hessian.Hessian2ObjectOutput

8. NativeJava 实现

> 旁白君：由于芮芮对 Java 原生的序列化，了解的比较粗浅，本小节更多的是把代码梳理干净。

nativejava ，基于 Java 原生（ 自带 ）的 Java 序列化实现，即使用 java.io.ObjectInputStream 和 java.io.ObjectOutputStream 进行序列化和反序列化。

代码比较简单，和 『5. FST 实现』 类似，胖友自己查看：

- com.alibaba.dubbo.common.serialize.support.nativejava.NativeJavaSerialization
- com.alibaba.dubbo.common.serialize.support.nativejava.NativeJavaObjectInput
- com.alibaba.dubbo.common.serialize.support.nativejava.NativeJavaObjectOutput

8.1 Java 实现

java ，在 『8. NativeJava 实现』 的基础上，实现了对空字符串和空对象的处理。如下是 JavaObjectOutput 对空字符串和空对象的序列化，代码如下：

```
// 【注意】JavaObjectOutput extends NativeJavaObjectOutput ！！！

@Override
public void writeUTF(String v) throws IOException {
    if (v == null) { // 空字符串
        getObjectOutputStream().writeInt(-1);
    } else {
        getObjectOutputStream().writeInt(v.length()); // 长度
        getObjectOutputStream().writeUTF(v); // 字符串
    }
}

@Override
public void writeObject(Object obj) throws IOException {
    if (obj == null) { // 空
        getObjectOutputStream().writeByte(0); // 空
    } else {
        getObjectOutputStream().writeByte(1); // 非空
        getObjectOutputStream().writeObject(obj); // 对象
    }
}
```

代码比较简单，和 『NativeJava 实现』 类似，胖友自己查看：

- com.alibaba.dubbo.common.serialize.support.java.JavaSerialization
- com.alibaba.dubbo.common.serialize.support.java.JavaObjectInput
- com.alibaba.dubbo.common.serialize.support.java.JavaObjectOutput

8.2 CompactedJava

compactedjava ，在 『8.1 Java 实现』 的基础上，实现了对 ClassDescriptor 的处理。如下是 CompactedObjectOutputStream 对 ClassDescriptor 的写入，代码如下：

```
// 【注意】CompactedObjectOutputStream extends ObjectOutputStream ！！！

@Override
protected void writeClassDescriptor(ObjectStreamClass desc) throws IOException {
    Class<?> clazz = desc.forClass();
    if (clazz.isPrimitive() || clazz.isArray()) {
        write(0);
        super.writeClassDescriptor(desc);
    } else {
        write(1);
        writeUTF(desc.getName());
    }
}
```

在 JavaObjectOutput 的创建时，根据 compact = true 时，使用 CompactedObjectOutputStream 输出流。代码如下：

```
public JavaObjectOutput(OutputStream os, boolean compact) throws IOException {
    super(compact ? new CompactedObjectOutputStream(os) : new ObjectOutputStream(os));
}
```

代码比较简单，胖友自己查看：

com.alibaba.dubbo.common.serialize.support.java.CompactedJavaSerialization
com.alibaba.dubbo.common.serialize.support.java.CompactedObjectInputStream
com.alibaba.dubbo.common.serialize.support.java.CompactedObjectOutputStream

666. 彩蛋

推荐阅读：

《在Dubbo中使用高效的Java序列化（Kryo和FST）》
《深入理解RPC之序列化篇 - 总结篇》
《序列化和反序列化》

文章目录

回到首页