Dubbo序列化和反序列化模块（二） 主要交互流程分析

曹瀚云

2018K8009907008

本文作为王伟老师OOP课程的大作业——Dubbo序列化和反序列化模块源码分析报告的第二部分，将紧承第一部分结尾所概述的该模块中各个类与接口的关系，结合具体代码剖析序列化/反序列化的交互流程，从中学习dubbo面向对象程序设计的思想，并进一步为报告的第三部分设计模式和高级设计意图做铺垫。

1. Java中序列化相关的基础知识

1. 需要序列化的类必须实现`java.io.Serializable`接口，否则会抛出`NotSerializableException`异常
2. 如果检测到反序列化后的类的`serialVersionUID`和对象二进制流的`serialVersionUID`不同，则会抛出 异常。
3. Java的序列化会将一个类包含的引用中所有的成员变量保存下来（深度复制），所以里面的引用类型必须也要实现`java.io.Serializable`接口。
4. 对于不采用默认序列化或无须序列化的成员变量，可以添加`transient`关键字，但并不是说添加了`transient`关键字就一定不能序列化。
5. 每个类可以实现`readObject`、`writeObject`方法实现自己的序列化策略，即使是`transient`修饰的成员变量也可以手动调用`ObjectOutputStream`的`writeInt`等方法将这个成员变量序列化。

2. 源码分析：交互流程

在报告的第一部分结尾，我附有dubbo-serialization模块（在Github的Dubbo仓库serialization部分里）的文件层次结构以及Dubbo序列化模块关系两张图，在这里为了方便再次展示：

图 1. Dubbo-serialization下文件层次结构示例

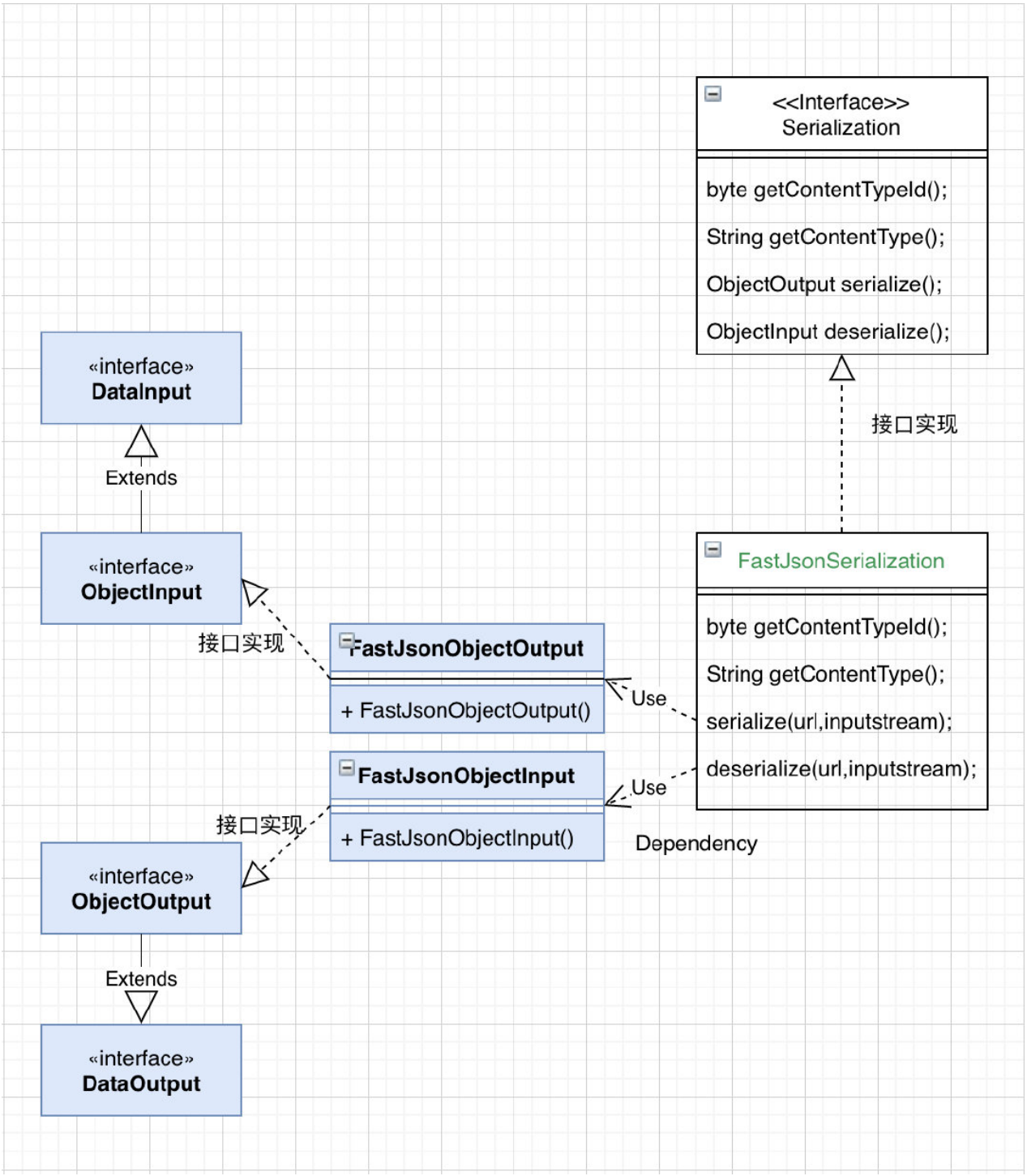

图 2. Dubbo序列化模块关系的UML图（以FastJson方法为例）

从图2可以看出，以FastJson方法为例，要实现FastJsonObjectInput/Output以及FastJsonSerialization这三个核心模块，涉及到Dubbo提供的如下抽象接口：

- `org.apache.dubbo.common.serialize.Serialization`
- `org.apache.dubbo.common.serialize.ObjectInput`
- `org.apache.dubbo.common.serialize.ObjectOutput`

因为我们重点在剖析Dubbo-serialization部分的面向对象思想，因此以下源码分析主要只以较为常见的FastJson作为序列化/反序列化api的实现方法。

(1)与执行序列化相关的DataInput & DataOutput接口

如官方注释所说，DataInput是“Basic data type input interface”，即基础数据类型输入接口，用于从输入流中读取各种类型的数据。该接口里的方法的具体实现在FastJsonObjectInput中。

下面是`DataInput`类的代码：

```
public interface DataInput {

    /**
     * Read boolean.
     *
     * @return boolean.
     * @throws IOException
     */
    boolean readBool() throws IOException;

    /**
     * Read byte.
     *
     * @return byte value.
     * @throws IOException
     */
    byte readByte() throws IOException;

    /**
     * Read short integer.
     *
     * @return short.
     * @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;
}
```

DataInput是“Basic data type output interface”，即基础数据类型输出接口，用于将各种类型的数据转化为Json格式（对于FastJson而言），输出到OutputStream中。

```
public interface DataOutput {

```

```
    /**
     * Write boolean.
     *
     * @param v value.
     * @throws IOException
     */
    void writeBool(boolean v) throws IOException;

    /**
     * Write byte.
     *
     * @param v value.
     * @throws IOException
     */
    void writeByte(byte v) throws IOException;

    /**
     * Write short.
     *
     * @param v value.
     * @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;

    void flushBuffer() throws IOException;

}
```

(2)与执行序列化相关的ObjectInput & ObjectInput接口

ObjectInput & ObjectInput接口与上面Data之类的接口不同，他们是处理object类的数据的序列化和反序列化。因为这些接口只是声明了抽象方法，并未实现。因此只是简单的在下面附上代码，供读者了解该接口包含了那些抽象方法，方法的具体实现在下面——（3）最核心的Serialization接口这部分中的FastJsonObjectInput & Output中实现（包括上面的DataInput & Output）。

ObjectInput接口代码如下：

```
/**
 * Object input interface.
 */
public interface ObjectInput extends DataInput {

    /**
     * Consider use {@link #readObject(Class)} or {@link #readObject(Class, Type)} where possible
     *
     * @return object
     * @throws IOException if an I/O error occurs
     * @throws ClassNotFoundException if an ClassNotFoundException occurs
     */
    @Deprecated
    Object readObject() throws IOException, ClassNotFoundException;

    /**
     * read object
     *
     * @param cls object class
     * @return object
     * @throws IOException if an I/O error occurs
     * @throws ClassNotFoundException if an ClassNotFoundException occurs
     */
    <T> T readObject(Class<T> cls) throws IOException, ClassNotFoundException;

    /**
     * read object
     *
     * @param cls object class
     * @param type object type
     * @return object
     * @throws IOException if an I/O error occurs
     * @throws ClassNotFoundException if an ClassNotFoundException occurs
     */
    <T> T readObject(Class<T> cls, Type type) throws IOException, ClassNotFoundException;


    /**
     * The following methods are customized for the requirement of Dubbo's RPC protocol implementation. Legacy protocol
     * implementation will try to write Map, Throwable and Null value directly to the stream, which does not meet the
     * restrictions of all serialization protocols.
     *
```

```
     * <p>
     * See how ProtobufSerialization, KryoSerialization implemented these methods for more details.
     * <p>
     * <p>
     * The binding of RPC protocol and biz serialization protocol is not a good practice. Encoding of RPC protocol
     * should be highly independent and portable, easy to cross platforms and languages, for example, like the http headers,
     * restricting the content of headers / attachments to Ascii strings and uses ISO_8859_1 to encode them.
     * https://tools.ietf.org/html/rfc7540#section-8.1.2
     */
    default Throwable readThrowable() throws IOException, ClassNotFoundException {
        Object obj = readObject();
        if (!(obj instanceof Throwable)) {
            throw new IOException("Response data error, expect Throwable, but get " + obj);
        }
        return (Throwable) obj;
    }

    default Object readEvent() throws IOException, ClassNotFoundException {
        return readObject();
    }

    default Map<String, Object> readAttachments() throws IOException, ClassNotFoundException {
        return readObject(Map.class);
    }
}
```

ObjectOutput接口代码如下：

```
/**
 * Object output interface.
 */
public interface ObjectOutput extends DataOutput {

    /**
     * write object.
     *
     * @param obj object.
     */
    void writeObject(Object obj) throws IOException;

    /**
```

```
     * The following methods are customized for the requirement of Dubbo's RPC protocol implementation. Legacy protocol
     * implementation will try to write Map, Throwable and Null value directly to the stream, which does not meet the
     * restrictions of all serialization protocols.
     *
     * <p>
     * See how ProtobufSerialization, KryoSerialization implemented these methods for more details.
     * <p>
     *
     * The binding of RPC protocol and biz serialization protocol is not a good practice. Encoding of RPC protocol
     * should be highly independent and portable, easy to cross platforms and languages, for example, like the http headers,
     * restricting the content of headers / attachments to Ascii strings and uses ISO_8859_1 to encode them.
     * https://tools.ietf.org/html/rfc7540#section-8.1.2
     */
    default void writeThrowable(Object obj) throws IOException {
        writeObject(obj);
    }

    default void writeEvent(Object data) throws IOException {
        writeObject(data);
    }

    default void writeAttachments(Map<String, Object> attachments) throws IOException {
        writeObject(attachments);
    }

}
```

(3)最核心的Serialization接口

`Serialization`中提供了`serialize`和`deserialize`接口对数据进行序列化和反序列化操作，默认会选择hessian2作为序列化和反序列化接口的实现。

```
/**
 * Serialization strategy interface that specifies a serializer. (SPI, Singleton, ThreadSafe)
 *
 * The default extension is hessian2 and the default serialization implementation of the dubbo protocol.
 * <pre>
 *     e.g. &lt;dubbo:protocol serialization="xxx" /&gt;
 * </pre>
 */
```

```java
@SPI("hessian2")
public interface Serialization {
    /*
    ...
    */
    byte getContentTypeId();
    /*
    ...
    */
    String getContentType();

    /**
     * Get a serialization implementation instance
     *
     * @param url URL address for the remote service
     * @param output the underlying output stream
     * @return serializer
     * @throws IOException
     */
    @Adaptive
    ObjectOutput serialize(URL url, OutputStream output) throws IOException;
    @Adaptive
    ObjectInput deserialize(URL url, InputStream input) throws IOException;

}
```

这个抽象接口主要包括了4个方法：`getContentTypeId()`，`getContentType()`，`serialize()`和`deserialize()`。因为serialization这个接口被`FastJsonSerialization.java`里的`FastJsonSerialization`类实现，这个类的代码如下：

```java
public class FastJsonSerialization implements Serialization {

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

    @Override
    public String getContentType() {
        return "text/json";
    }

    @Override
    public ObjectOutput serialize(URL url, OutputStream output) throws IOException {
        return new FastJsonObjectOutput(output);
    }
```

```
17
18      @Override
19      public ObjectInput deserialize(URL url, InputStream input) throws
    IOException {
20          return new FastJsonObjectInput(input);
21      }
22
23  }
```

可以看到类的实现：`getContentTypeId()`就是简单把FASTJSON_SERIALIZATION_ID这个参数值返回；`getContentType()`返回了`"text/json"`这个字符串；`serialize`这个方法返回了`new FastJsonObjectOutput(output)`；`deserialize`这个方法返回了`new FastJsonObjectInput(output)`。

我们详细分析一下serialize：

```
1  @Override
2      public ObjectOutput serialize(URL url, OutputStream output) throws
    IOException {
3          return new FastJsonObjectOutput(output);
4      }
```

`serialize`函数新建了一个`FastJsonObjectOutput(output)`产生的对象并将其return。并且值得注意的是，函数的返回值类型声明为`ObjectOutput`，正是`FastJsonObjectOutput`的父类。也就是说这个函数的内部语句等价于下面两句：

```
1  ObjectOutput A = new FastJsonObjectOutput(output);
2  return A;
```

我们发现，第一行正符合多态的要求——父类引用指向子类对象。而多态作为面向对象思想的三大特征之一，其相关内容如下：

> 多态——同一个行为具有多个不同表现形式或形态的能力，是指***一个类实例（对象）的相同方法***在不同情形有**不同表现形式**。多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，它们（那些操作）可以通过相同的方式予以调用。
>
> **多态的优点：**
>
> - 1. 消除类型之间的耦合关系
> - 2. 可替换性
> - 3. 可扩充性
> - 4. 接口性
> - 5. 灵活性
> - 6. 简化性
>
> 多态存在的**三个必要条件**：
>
> - 继承

> - 重写（子类继承父类后对父类方法进行重新定义）
> - 父类引用指向子类对象

这样可以使得外部接收`serialize`函数返回值的变量类型为`ObjectOutput`，且对于接口`ObjectOutput`和父接口`DataInput`以及和两者的实现类`FastJsonObjectOutput`都定义的方法，调用时直接使用子类中的方法（注：结合上面图2:Dubbo序列化模块关系的UML图可知，`ObjectOutput`这个接口继承了`DataOutput`这个父接口，在Java中，一个类实现子接口，则**在实现类中，既要实现“子接口”中的方法，又要实现“父接口”中的方法**）。

这样，在外部调用`FastJsonSerialization`类的`serialize`方法后，返回的对象中就包含有`FastJsonObjectInput`类中实现的`readBool()`、`readByte()`、`readObject()`等多个类型的序列化方法可供使用。这些read方法把各自的类型（如boolean.class, byte.class等）作为参数传入read方法中，read会调用`JSON.parseObject()`方法将输入流中的Json字符串转化为相应的类的对象。

`read()`方法代码如下所示：

```
private <T> T read(Class<T> cls) throws IOException {
        String json = readLine();
        return JSON.parseObject(json, cls);
    }
```

3.小结

本文作为OOP-源码分析报告的第二部分，主要目的是分析Dubbo-序列化/反序列化部分源码的交互流程，体会面向对象思想的应用。从上面报告部分可以看出，模块之间有着“高内聚，低耦合”的特点，并且通过面向接口编程，我们分析出了多态的体现；此外随处可见的SPI的@Adaptive注解也不时地提醒着我们其“控制反转 & 依赖注入”的特点（但由于本人对Java语言特性了解尚浅，此处可能以后在深入学习后会更详细地展开介绍）。更重要的，在最核心的`serialization.java`文件的第28行出现了"Singleton"这个代表单例模式的单词，这将是我们第三部分报告的重点——分析使用到的设计模式和高级设计意图。我会持续不断地努力学习Java并研究Dubbo的源码，努力丰富自己的认识，完善这份报告。