fastjson 分析报告

中国科学院大学
[姓名] 王晨赳
[学号] 2016K8009929060

一、json 及 fastjson 背景介绍

fastjson 是一个高性能功能完善的 JSON 库,可用于将 Java 对象转换为 JSON 序列，也可以将 JSON 反序列化为 Java 对象。fastjson 是目前解析 JSON 最快的 Java 库。fastjson 是 github 上的开源项目，它的接口简单易用,已经被广泛使用在缓存序列化、协议交互、Web 输出、Android 客户端等多种应用场景。本文将对 fastjson 的序列化和反序列化功能做分析。

既然 fastjson 是解析 JSON，首先认识一下 JSON。JSON 是 JavaScript Object Notation 的缩写，意为 JavaScript 对象表示法，它是一种数据交换格式。在 JSON 出现之前，大家一直用 XML 来传递数据。因为 XML 是一种纯文本格式，所以它适合在网络上交换数据。但 XML 的规则过于复杂，使人很不愉悦。后来在 2002 年的一天，道格拉斯·克罗克福特（Douglas Crockford）为了拯救深陷水深火热同时又被某几个巨型软件企业长期愚弄的软件工程师，发明了 JSON 这种超轻量级的数据交换格式。道格拉斯长期担任雅虎的高级架构师，自然钟情于 JavaScript。他设计的 JSON 实际上是 JavaScript 的一个子集。JSON 的数据类型与 JavaScript 的相差无几。JSON 本质上是一个字符串，要把任何 JavaScript 对象变成 JSON，就是把这个对象序列化成一个 JSON 格式的字符串，这样才能够通过网络传递给其他计算机。如果我们收到一个 JSON 格式的字符串，只需要把它反序列化成一个 JavaScript 对象，就可以在 JavaScript 中直接使用这个对象了。fastjson 即可高效的完成这些工作。

二、fastjson 代码分析

（一）概述

fastjson 的基本功能是序列化和反序列化。序列化指的是将 Java 对象转换为 JSON 字符串。反序列就是相反的过程，即将 JSON 字符串转换为 Java 对象。需要注意的是 JSON 是一种数据交换格式，因此 JSON 一般不会用来传递方法，所谓将 Java 对象转换为 JSON 字符串，是将对象的属性以 JSON 格式保存。而将 JSON 字符串转换为 Java 对象，一般指将其转换为一种称为 JavaBean 的特殊的类。JavaBean 类的定义是，提供一个默认的无参构造函数，需要被序列化并且实现了 Serializable 接口，可能有一系列 private 的可读写属性，可能有一系列的"getter"或"setter"方法，也就是一种十分简单的类。

使用 fastjson 的 API 包含在 com.alibaba.fastjson.JSON 类中，该类提供了有关序列化和反序列化的静态方法，可以直接调用，主要包括以下几个：

1. public static JSONObject parseObject(String text)　//将 JSON 文本转换为 JSONObject 对象

2. public static <T> T parseObject(String text, Class<T> clazz)　//将 JSON 文本转换为指定的类的示例

3. public static JSONArray parseArray(String text)　//将 JSON 文本转换为 JSONArray 对象

4. public static <T> List<T> parseArray(String text, Class<T> clazz)　//将 JSON 文本转换为指定类的对象列表

5. public static String toJSONString(Object object)　//将 Java 对象转换为 JSON 文本

使用 fastjson 时还需要注意，fastjson 并不支持所有类的序列化和反序列化，实际上 fastjson 支持的可序列化和可反序列的类如下：

表 1：fastjson 支持的序列化类型

注册的类型	序列化实例	是否支持序列化	是否支持反序列化
Boolean	BooleanCodec	是	是
Character	CharacterCodec	是	是
Byte	IntegerCodec	是	是
Short	IntegerCodec	是	是
Integer	IntegerCodec	是	是
Long	LongCodec	是	是
Float	FloatCodec	是	是
Double	DoubleSerializer	是	-
BigDecimal	BigDecimalCodec	是	是
BigInteger	BigIntegerCodec	是	是
String	StringCodec	是	是
byte[]	PrimitiveArraySerializer	是	-
short[]	PrimitiveArraySerializer	是	-
int[]	PrimitiveArraySerializer	是	-
long[]	PrimitiveArraySerializer	是	-
float[]	PrimitiveArraySerializer	是	-
double[]	PrimitiveArraySerializer	是	-
boolean[]	PrimitiveArraySerializer	是	-
char[]	PrimitiveArraySerializer	是	-
Object[]	ObjectArrayCodec	是	是
Class	MiscCodec	是	是
SimpleDateFormat	MiscCodec	是	是
Currency	MiscCodec	是	是

TimeZone	MiscCodec	是	是
InetAddress	MiscCodec	是	是
Inet4Address	MiscCodec	是	是
Inet6Address	MiscCodec	是	是
InetSocketAddress	MiscCodec	是	是
File	MiscCodec	是	是
Appendable	AppendableSerializer	是	-
StringBuffer	AppendableSerializer	是	-
StringBuilder	AppendableSerializer	是	-
Charset	ToStringSerializer	是	-
Pattern	ToStringSerializer	是	-
Locale	ToStringSerializer	是	-
URI	ToStringSerializer	是	-
URL	ToStringSerializer	是	-
UUID	ToStringSerializer	是	-
AtomicBoolean	AtomicCodec	是	是
AtomicInteger	AtomicCodec	是	是
AtomicLong	AtomicCodec	是	是
AtomicReference	ReferenceCodec	是	是
AtomicIntegerArray	AtomicCodec	是	是
AtomicLongArray	AtomicCodec	是	是
WeakReference	ReferenceCodec	是	是
SoftReference	ReferenceCodec	是	是
LinkedList	CollectionCodec	是	是

注：该表转载自网页 https://zonghaishang.gitbooks.io/fastjson-source-code-analysis/content/

在实现具体的序列化和反序列化的时候会使用第二列的序列化示例，详细过程见下文。

下面给出使用 fastjson 的示例，不失一般性，示例定义了几个被称为 JavaBean 的特殊的类。

```
1  public class TestFastJson {
2      static class Person{
3          private String name;
4          private int age;
5
6          public Person(){
7          }
8          public Person(String name,int age){
9              this.name=name;
10             this.age=age;
11         }
12         public String getName() {
13             return name;
14         }
15         public void setName(String name) {
16             this.name = name;
17         }
18         public int getAge() {
19             return age;
20         }
21         public void setAge(int age) {
22             this.age = age;
23         }
24     }
25     public static void main(String[] args) {
26         method1();
27         method2();
28     }
29
```

```
30      static void method1(){
31          System.out.println("javabean转化示例开始----------");
32          Person person = new Person("Jack",18);
33
34          //这里将javabean转化成json字符串
35          String jsonString = JSON.toJSONString(person);
36          System.out.println(jsonString);
37          //这里将json字符串转化成javabean对象，
38          person =JSON.parseObject(jsonString,Person.class);
39          System.out.println(person.toString());
40
41          System.out.println("javabean转化示例结束----------");
42      }
43
44      static void method2(){
45          System.out.println("List<javabean>转化示例开始----------");
46
47          Person person1 = new Person("Mike",19);
48          Person person2 = new Person("James",20);
49          List<Person> persons = new ArrayList<Person>();
50          persons.add(person1);
51          persons.add(person2);
52          //将对象列表转换为JSON字符串
53          String jsonString = JSON.toJSONString(persons);
54          System.out.println("json字符串:"+jsonString);
55
56          //json字符串转换为Person类的对象列表
57          List<Person> persons_t = JSON.parseArray(jsonString,Person.class);
58          //输出解析后的person对象
59          System.out.println("person1对象: "+persons_t.get(0).toString());
60          System.out.println("person2对象: "+persons_t.get(1).toString());
61
62          System.out.println("List<javabean>转化示例结束----------");
63      }
64  }
65
```

图 1：使用 fastjson 的示例

接下对 fastjson 的序列化和反序列化功能做进一步的分析。

（二）fastjson 的序列化过程

将 Java 对象转换为 JSON 文本需要使用的类主要有 JSON 类，JSONSerializer 类，SerializeWriter 类。

使用 fastjson 的 API 包含在 JSON 类中，这是一个抽象类。在《Java 语言程序设计》一书中说，“在继承的层次结构中，随着每个新子类的出现，类会变得越来越明确和具体。如果从一个子类型追溯到父类，类就会变得更通用、更加不明确。类的设计应该确保父类包含它的子类的共同特征。有时候，一个父类设计得非常抽象，以至于它都没有任何具体的实例。这样的类称为抽象类。”也就是说使用 JSON 类时不需要为其构造具体的实例。而且该类提供了有关序列化和反序列化的静态方法，可以直接调用。

JSON 类开始定义了几个成员变量，之后是几个由 static 修饰的代码块，说明 JSON 类会首先执行这几段代码。代码主要是为成员变量 feature 赋值，feature 是与 fastjson 进行序列化和反序列化功能的配置相关的变量。这些配置信息包括了一些输出格式等等，这里是设置了一些默认的配置。

之前提到将 Java 对象转换为 JSON 文本调用的方法为 public static String toJSONString(Object object)，这里重点关注该方法。其实在 JSON 类中还有许多关于 toJSONString 的重载方法，其余的这些方法增添了关于各种配置信息的的参数，这些配置信息对分析 fastjson 序列化功能关系不大。跟踪 toJSONString(Object object)方法最终会调用重载方法 public static String toJSONString(Object object, int defaultFeatures, SerializerFeature... features)，该方法源码如下图所示：

```
617     public static String toJSONString(Object object, int defaultFeatures, SerializerFeature... features) {
618         SerializeWriter out = new SerializeWriter((Writer) null, defaultFeatures, features);
619 
620         try {
621             JSONSerializer serializer = new JSONSerializer(out);
622             serializer.write(object);
623             return out.toString();
624         } finally {
625             out.close();
626         }
627     }
```

图 2：JSON 类的 toJSONString 方法

可以看到重载方法多出来的参数是一些配置信息，对我们分析方法功能关系不大。这个方法里构造了一个 SerializeWriter 实例，接下来我们先看看 SerializeWriter 这个类的作用是什么。

SerializeWriter 类定义在 com.alibaba.fastjson.serializer 中，这个类继承了 Writer 类。Writer 是 Java 自带的一个类，是处理与输出流相关的一个类。SerializeWriter 类的主体部分定义了一个字符数组 buf[]，这个数组用来存放转换成功的 JSON 字符串。SerializeWriter 类也定义了一些关于配置信息的成员变量，这些变量和 buf[]数组都是 protected 的，因此只能被该类的子类所访问。阅读源码后发现这个类实现了一些将特定类型变量写入输出流，也就是 buf[]中的方法。以 public void writeInt(int i)方法示意如下：

```
516     public void writeInt(int i) {
517         if (i == Integer.MIN_VALUE) {
518             write("-2147483648");
519             return;
520         }
521 
522         int size = (i < 0) ? IOUtils.stringSize(-i) + 1 : IOUtils.stringSize(i);
523 
524         int newcount = count + size;
525         if (newcount > buf.length) {
526             if (writer == null) {
527                 expandCapacity(newcount);
528             } else {
529                 char[] chars = new char[size];
530                 IOUtils.getChars(i, size, chars);
531                 write(chars, 0, chars.length);
532                 return;
533             }
534         }
535 
536         IOUtils.getChars(i, newcount, buf);
537 
538         count = newcount;
539     }
```

图 3：SerializeWriter 类的 writeInt 方法

这个方法是将 int 型的值转换为字符串写到了输出流中。

现在我们知道了 SerializeWriter 类主要是实现了一些将不同类型变量写到输出流中的方法。对于不同的类型变量，对应的方法有其具体的实现，有一些方法还需要根据配置信息决定往输出流写入的格式。具体实现不做介绍，

现在只需要理解这个类的功能就行了。

继续回到之前 JSON 类里的 public static String toJSONString(Object object, int defaultFeatures, SerializerFeature... features)方法：

```
617     public static String toJSONString(Object object, int defaultFeatures, SerializerFeature... features) {
618         SerializeWriter out = new SerializeWriter((Writer) null, defaultFeatures, features);
619 
620         try {
621             JSONSerializer serializer = new JSONSerializer(out);
622             serializer.write(object);
623             return out.toString();
624         } finally {
625             out.close();
626         }
627     }
```

图 4：JSON 类的 toJSONString 方法

在实例化了 SerializerWriter 类得到对象 out 之后，又将对象 out 作为参数传入 JSONSerializer 类的构造方法，构造了一个 JSONSerializer 实例 serializer，之后回调用 serializer 的 writer(Object object)方法，最后返回 out.toString()。由之前对 SerializerWriter 类的分析可知 toString()方法就是将输出流缓存 buf[]字符数组输出，于是就得到了 Java 对象 Object object 转换为 JSON 字符串的结果。现在来看看 JSONSerializer 类的实现。

JSONSerialize 类定义在 com.alibaba.fastjson.serializer 中，这个类继承了 SerializeFilterable 类。SerializeFilterable 类是序列化拦截器，主要用于过滤一些无用的信息，这里暂不做讨论。这个类同样有一些配置信息的成员变量，但我们主要关注它的 write(Object object)方法。该方法的源码如下：

```
271     public final void write(Object object) {
272         if (object == null) {
273             out.writeNull();
274             return;
275         }
276 
277         Class<?> clazz = object.getClass();
278         ObjectSerializer writer = getObjectWriter(clazz);
279 
280         try {
281             writer.write(this, object, null, null, 0);
282         } catch (IOException e) {
283             throw new JSONException(e.getMessage(), e);
284         }
285     }
```

图 5：JSONSerialize 类的 write 方法

如果传入参数是 NULL 就输出 out.writeNull()。接下来得到对象 object 的类，这里存在一个泛型的概念。得到 object 的类 clazz 之后，调用 getObjectWriter(clazz)方法返回接口 ObjectSerializer 的实例 writer。接口 ObjectSerializer 源码如下：

```
70        void write(JSONSerializer serializer, //
71                   Object object, //
72                   Object fieldName, //
73                   Type fieldType, //
74                   int features) throws IOException;
75    }
```

图 6：接口 ObjectSerializer

那么 getObjectWriter(clazz)方法返回的是什么呢？这个方法同样是 JSONSerializer 类里的，源码如下：

```
381        public ObjectSerializer getObjectWriter(Class<?> clazz) {
382            return config.getObjectWriter(clazz);
383        }
```

图 7：JSONSerialize 类的 getObjectWriter 方法

config 是 SerializeConfig 的实例，调用其 getObjectWriter(Class<?> clazz)方法，之后会再调用重载方法 getObjectWriter(Class<?> clazz, boolean create)，这个方法的实现较长，这里不再放出来了。但知道它的功能是找到传入参数类 clazz 的序列化实例，这些序列化实例在之前的图中已经列举过了，它们也都在 com.alibaba.fastjson.serializer 中。现在我们假设类 clazz 是 Integer 类，以此举例。于是 getObjectWriter(clazz, true)会找到 Integer 类对应的序列化实例：IntegerCodec，显然这些序列化实例都应该实现 ObjectSerializer 接口。于是原来 JSON 类中返回的对象 writer 就是 IntegerCodec 类的实例，这个类实现了 ObjectSerializer 接口的 write 方法，源码如下：

```
40        public void write(JSONSerializer serializer, Object object, Object fieldName, Type fieldType, int features) throws IOException {
41            SerializeWriter out = serializer.out;
42
43            Number value = (Number) object;
44
45            if (value == null) {
46                out.writeNull(SerializerFeature.WriteNullNumberAsZero);
47                return;
48            }
49
50            if (object instanceof Long) {
51                out.writeLong(value.longValue());
52            } else {
53                out.writeInt(value.intValue());
54            }
55
56            if (out.isEnabled(SerializerFeature.WriteClassName)) {
57                Class<?> clazz = value.getClass();
58                if (clazz == Byte.class) {
59                    out.write('B');
60                } else if (clazz == Short.class) {
61                    out.write('S');
62                }
63            }
64        }
```

图 8：IntegerCodec 类的 write 方法

可见这里调用了之前分析的 SerializerWriter 的 writeLong 和 writeInt 方法，将 Integer 类的变量写到 SerializerWriter 实例 out 的输出流 buf[]中。也就是说在以下代码片段

```
617     public static String toJSONString(Object object, int defaultFeatures, SerializerFeature... features) {
618         SerializeWriter out = new SerializeWriter((Writer) null, defaultFeatures, features);
619 
620         try {
621             JSONSerializer serializer = new JSONSerializer(out);
622             serializer.write(object);
623             return out.toString();
624         } finally {
625             out.close();
626         }
627     }
```

图 9：JSON 类的 toJSONString 方法

第 622 行执行后 out 的 buf[]存储着准 JSON 字符串，执行 623 行后这个就得到了转换完成的 JSON 字符串。然后返回该字符串，Java 对象的序列化到此结束。

（三）fastjson 的反序列化过程

反序列化过程实现主要涉及 JSON 类、DefaultJSONParse 类、词法分析器 JSONLexerBase 类。fastjson 的反序列化功能也为用户提供了友好的接口，

1. public static JSONObject parseObject(String text)　//将 JSON 文本转换为 JSONObject 对象
2. public static <T> T parseObject(String text, Class<T> clazz)　//将 JSON 文本转换为指定的类的示例
3. public static JSONArray parseArray(String text)　　//将 JSON 文本转换为 JSONArray 对象
4. public static <T> List<T> parseArray(String text, Class<T> clazz)　//将 JSON 文本转换为指定类的对象列表

同样的，与序列化过程一致，反序列化也列有可支持的反序列化类型，其中一部分类型如下表所示，

表 2：fastjson 支持的部分反序列化类型

注册的类型	反序列化实例	是否支持序列化	是否支持反序列化
SimpleDateFormat	MiscCodec	是	是
Timestamp	SqlDateDeserializer	-	是
Date	SqlDateDeserializer	-	是
Time	TimeDeserializer	-	是
Date	DateCodec	是	是
Calendar	CalendarCodec	是	是
XMLGregorianCalendar	CalendarCodec	是	是
JSONObject	MapDeserializer	-	是
JSONArray	CollectionCodec	是	是
Map	MapDeserializer	-	是
HashMap	MapDeserializer	-	是
LinkedHashMap	MapDeserializer	-	是
TreeMap	MapDeserializer	-	是

注：该表转载自网页 https://zonghaishang.gitbooks.io/fastjson-source-code-analysis/content/

对于 fastjson 反序列化过程的流程，其实它与序列化过程有很多相似之处，其流程可大致分为以下三步，

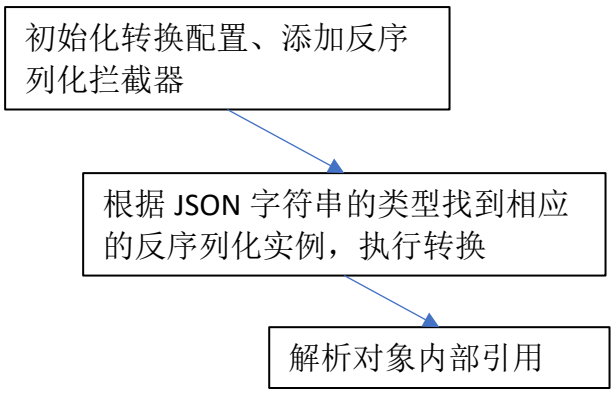

图 10：fastjson 反序列化过程

以实现反序列化的功能而言，第二步是需要分析的重点。

在将 JSON 字符串转换为 Java 对象之前需要经历一些准备阶段，也就是如何识别字符串的各个单词或字符的含义。也就是说，反序列列化的基础是词法分析。词法分析是将字符序列转换为单词（Token）序列的过程。进行词法分析的程序或者函数叫作词法分析器 Lexer，也叫扫描器 Scanner。fastjson 在包 com.alibaba.fastjson.parser 实现了词法分析器。之后在需要进行词法分析的时候，fastjson 就可以调用词法法分析器 Lexer。之前已经列出了反序列化的接口，也就是 parseObject 方法，它实现在 com.alibaba.fastjson.JSON 类中，其代码如下所示，

```
public static <T> T parseObject(String input, Type clazz, ParserConfig config, ParseProcess processor,
                                int featureValues, Feature... features) {
    if (input == null) {
        return null;
    }

    if (features != null) {
        for (Feature feature : features) {
            featureValues |= feature.mask;
        }
    }

    DefaultJSONParser parser = new DefaultJSONParser(input, config, featureValues);

     T value = (T) parser.parseObject(clazz, null);

     parser.handleResovleTask(value);

     parser.close();

     return (T) value;
}
```

图 11：JSON 类的 parseObject 方法

代码在此处实例化了一个 DefaultJSONParser 类的对象 parser，并将 input，也就是 json 字符串作为参数传入构造函数中。接下来在看看 DefaultJSONParser 类中的方法 parseObject 的实现。

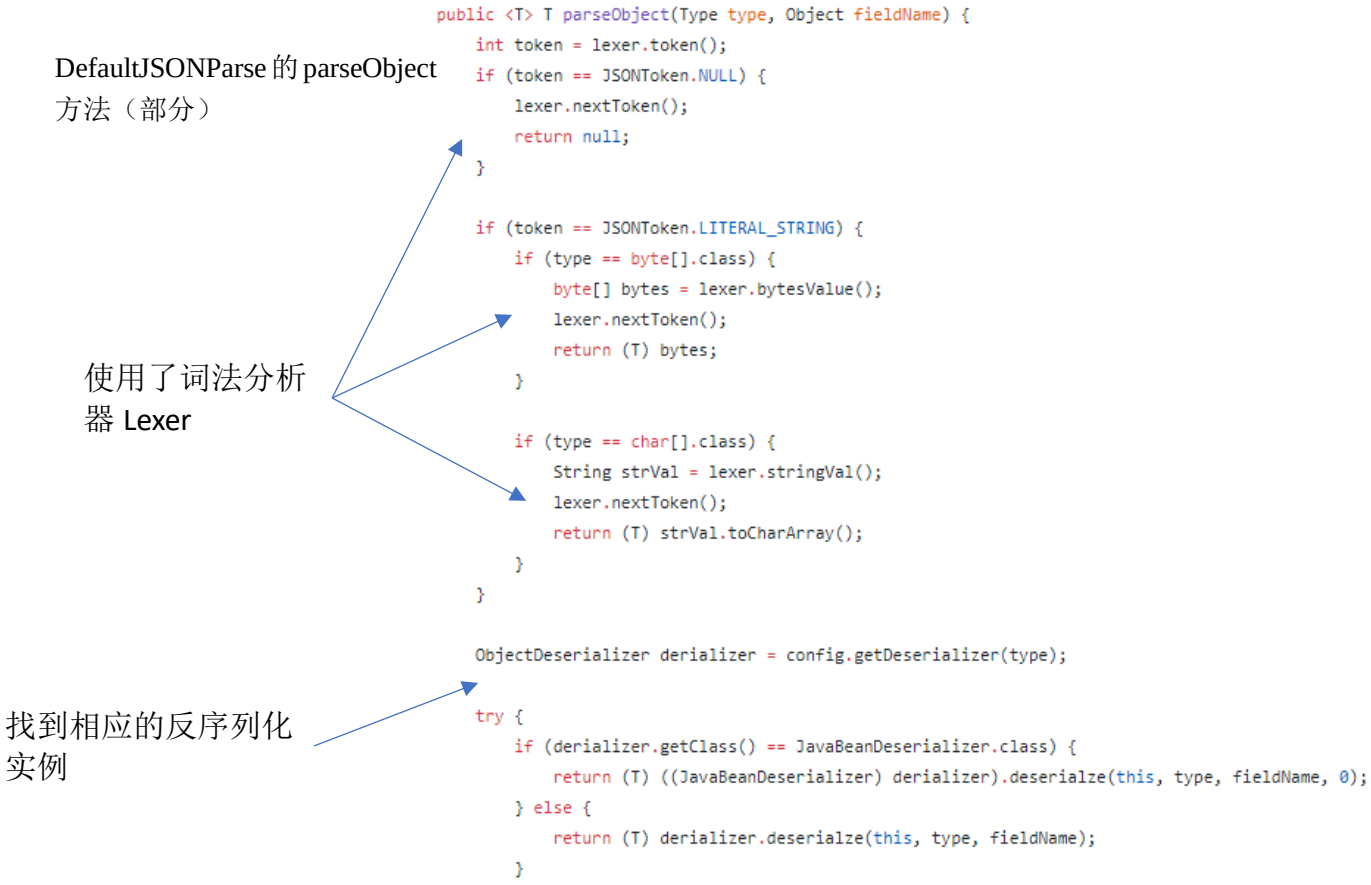

图 12：DefaultJSONParser 类的 parseObject 方法

在这个方法中调用了词法分析器做词法分析。ObjectDeserializer 与序列化时的 ObjectSserializer 类似也是一个接口。语句 ObjectDeserializer derializer = config.getDeserializer(type)，同之前序列化过程类似，就是找到一个相应的反序列化实例。找到这个实例 derializer 之后，之后就会使用这个反序列化器对输入的字符串做反序列化了。于是 parseObject 方法里的的 value 即为类型为 T 的对象，最后返回转换完成的对象 value，反序列化结束。

三、fastjson 的设计意图分析

（一）概述

软件设计现在正朝着越来越工程化的方向发展。为了提高软件开发的效率，降低软件开发的难度，软件开发需要遵循一些必要的指导，于是人们便提出了一些面向对象软件设计时可以参考依靠的设计原则和设计模式。面向对象主要有七种设计原则，分别为单一原则、开闭原则、里式替换原则、依赖倒转原则、接口分离原则、合成复用原则、迪米特原则。这些原则是为了解决一些面向对象设计时会产生的问题而提出的，主要是为了达到软件设计高内聚，低耦合的目标。设计模式总体来说分为三大类，共 23 种。分别为创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式；结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式；行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。这些设计模式是许多人软件设计经验的总结，使用这些模式可以提高开发效率，降低开发难度。

（二）设计原则分析

在 fastjson 项目中，面向对象的七种原则或多或少都有体现。

单一原则。单一原则说的是一个类应该只负责一项责任，即在设计的时候就应该仔细地将要实现的功能合理地拆分成最小的功能单位。例如将 fastjson 划分为序列化功能和反序列化功能。分别实现在 JSONSerializer 类和 DefaultJSONParse 类中。此外在序列化时，对于每一种序列化的对象，fastjson 都有一个序列化的类与其对应，如 IntegerCodec 类、FloatCodec 类、DateCodec 类等。反序列化时也有对应的反序列化器的类，同时还有类 JSONLexerBase 用于实现词法分析。可以看到 fastjson 对于类的划分基本符合单一原则，每一个类基本上只实现了需要它负责的功能，降低了类之间的耦合。

里式替换原则。里式替换原则说的是若类 B 继承于类 A，将类 A 的对象替换成类 B 的对象时，这个程序不会发生任何的变化。这就要求在程序实现时应该面向父类编程。例如序列化过程里需要使用的类 SerializeWriter 类继承了 writer 类。writer 类时 java 自带的一个抽象类，主要功能是将字符写到输出流中。SerializeWriter 类实现了 writer 类里的 write 方法。JSONSerializer 类继承自 SerializeFilterable 类。后者主要提供过滤一些信息的处理。查看其实现的代码，可以满足里式替换原则。

依赖倒置原则。依赖倒置原则指高层次的模块不应该依赖于低层次的模块，它们都应该依赖于抽象；抽象不应该依赖于具体实现，具体实现应该依赖于抽象。依赖倒置原则的核心在于解耦，为了解决类与类之间的耦合度过高导致类之间的依赖性太强。为达到此目的可以使用接口或抽象类。对于一个类来讲，如果之前它依赖于另一个类，那么现在可以把这两个类里所需要的共同的方法或变量抽象为一个接口。这样不同的类就可以在类自身实现需要的方法，实现了类与类之间的解耦。在 fastjson 中比如那些序列化和反序列化的实例，都实现了接口 ObjectSerializer 或 ObjectDeserializer。最开始的类 JSON 也实现了接口 JSONStreamAware,和 JSONAware。在实例本身实现接口里的抽象方法，降低了类之间的耦合程度，实现了面向接口编程。

接口隔离原则。接口隔离指一个类不应该依赖它不需要的接口。也就是说一个接口不应该实现得太过臃肿，只需要提供该类所需的最小接口即可。这样才是充分利用了接口的灵活性。但同时接口也不宜实现得太小，因为这样会导致接口数量的增多。所以需要设计者的权衡。可以看到 fastjson 里的接口如 ObjectSerializer 和 ObjectDeserializer 里只定义了一个抽象方法，充分实现了接口隔离。而词法分析器的接口 JSONLexer 就显得大多了。不过这个接口只实现在抽象类 JSONLexerBase 中，也符合接口隔离原则。

迪米特原则。迪米特原则说的是一个对象应该对另一个对象保持最小的了解。这也是为了降低类之间的耦合程度，增加内聚。例如序列化是需要使用的针对不同序列化对象的实例。其实将它们都放进类 JSONSerializer 里也可以实现，但是这样的话每次序列化对象时都会存在冗余，因为别的对象的序列化方法不是本次序列化必要的，代码的相当一部分并没有用。所以好的设计方法是将这些序列化方法拆开成一个个类，同时注意降低这些类之间的耦合。

合成复用原则。合成复用原则指的是如果只是想达到代码复用的目的，尽量使用组合与聚合，而不是继承。因为相对于只是引用其他类的方法，继承一个类显然耦合程度更高，而这并不是仅仅想复用一部分代码的类所希望看到的。这个原则其实用到的机会不大，因为想要复用的代码完全可以采用接口或继承抽象类实现，这样同样可以减

少耦合。

开闭原则。开闭原则指代码对扩展开放，对修改关闭。即对于已经完成的项目，如果产生了新的需求，那么首先应该是想着在原有代码基础上进行增添扩展，而不是修改原来的代码。因为修改原来已经顺利运行的代码可能会产生新的错误，所以为了避免这个问题，应该采取的方式是扩展代码。这就要求代码需要有较好的可扩展性。对于 fastjson 来说，因为它的功能有限，而且需求变化也不大，应该说都是修改代码为主，因为它历经了这么多年很少再需要增添新的类实现新的需求了。

面向对象的设计原则归根结底就是为了使代码达到高内聚，低耦合的水平。在具体设计时需要设计者对内聚和耦合有所权衡，尽量通过应用以上原则达到最好的效果。

（三）设计模式分析

设计模式的使用极大的提高了软件开发的工程化程度，fastjson 里也使用了几种不同的设计模式。针对主要功能，这些设计模式主要有属于创建型模式的工厂方法模式、属于结构型模式的适配器模式、属于行为型模式的中介者模式。其实在具体的代码内容中，有些模式的界限并不是特别清晰，毕竟这些模式也仅仅是提供了参考，没必要死板地完全按照这些模式去走。

工厂方法模式。这个模式出现在 fastjson 创建 JSON 对象的时候。fastjson 创建一个 JSON 对象的功能之前并没有提到过，因为它的实现其实与序列化差不多。工厂方法模式适合应用场景是当需要创建大量产品，且它们具有共同的接口。fastjson 创建一个 JSON 对象好像也符合此条件，这个功能是在 JSONObject 类里实现的，这个类还实现了接口 JSONSerializable。这个类里的方法 toString

```
public String toString() {
    return JSON.toJSONString(this);
}
```

图 13：JSONObject 类的 toString 方法

返回了 JSON 对象，对应于工厂方法模式的生产。工厂方法模式的精髓是用相同的方法生产出大量相同的产品，虽然 fastjson 可能不用产生大量的 JSON 对象，但也是可以采用这个模式的。

适配器模式。适配器模式是将某个接口、对象或类转换成另一个表示，目的是消除由于接口不匹配所造成的类的兼容性问题。fastjson 里在很多地方使用了接口的适配器模式。接口的适配器模式的引入是为了解决以下问题：当一个类里有许多抽象方法，于是该接口的实现类需要实现所有的抽象方法，有时候这样做比较浪费，因为并不是所有的方法都是该类需要的。为了解决此问题，引入了接口的适配器模式，即借助于一个抽象类，这个抽象类实现了这个接口的所有抽象方法。之后，原先需要实现该接口的类改为继承该抽象类，重写需要的方法即可。例如 fastjson 里的接口 JSONLexer 和抽象方法 JSONLexerBase。JSONLexer 是词法分析器接口，里面声明了词法分析时需要的抽象方法。之后这个接口在抽象类 JSONLexerBase 里被实现。随后 JSONScanner 和 JSONReaderScanner 类继承了这个抽象类并重写了一些方法。这个是接口的适配器模式，避免了只使用接口时会造成的麻烦。

中介者模式。中介者模式是为了降低耦合，引入了 Mediator 类，它提供统一接口，具体类与类之间的关系及调

度交给 Mediator 实现。 fastjson 中似乎没有具体的 Mediator，不过这个模式其实和依赖倒置原则类似，与面向接口编程很像，只不过在这个模式中接口由 Mediator 实现了。它们的主要考虑都是为了降低类与类之间的耦合，提高代码的健壮程度。

在大型的软件项目中，选用什么样的设计模式，是很考验一个软件开发人员的。合理地使用设计模式可以增强代码的可重用性、让代码更容易被他人理解、更好地保证代码可靠性。fastjson 中也体现了这些。

（四）结语

fastjson 作为目前最快的 JSON 序列化和反序列化工具，其项目设计必有可借鉴之处。分析过程中可以感受到作者对项目总体、一些特别的类都进行了精心地设计。项目的许多地方都有对为了提高性能而进行的优化，面向对象七种原则也都有体现。在一些地方也有面向对象设计模式的影子。总的来说，作者的设计意图一定是要使 fastjson 变得更快，同时给客户提供友好的接口，以便提高开发人员的开发效率。fastjson 为开源社区的建设做出了很大贡献。