



Fastjson 阅读报告

作者 刘骐鸣

课程名称 面向对象程序设计

教师姓名 王伟



前言

本报告是作为国科大 2019-2020 学年《面向对象程序设计》课程的大作业而撰写。

在一些 Topic 下会出现“FYI”字样开头的网址，其中包括了对 Topic 中所讲述内容的细节与延伸，感兴趣的读者可以结合阅读。

本报告对 Fastjson 这一项目进行分析，主要分为三个部分，第一章-功能分析与基本建模，简单介绍 Fastjson 的起源、功能等；第二章-核心流程设计分析，重点分析了 Fastjson 的两种功能，序列化与反序列化，对两种功能的流程与参与其中的类与对象做了介绍；第三章，高级设计模式分析，结合 23 种高级设计模式，对在 Fastjson 中发现的几种高级设计模式进行分析与介绍，探讨这些设计模式带来的帮助。

撰写报告时学习到了很多关于 Fastjson 的有趣知识，想尽可能地多记录一些，所以报告篇幅稍微超出预期。如果有些地方文段冗长、表意不明，还请多多包涵。

欢迎各位读者指出本报告中的错误与疏漏，若有疑问或指正，可以通过邮箱 liuqiming17@mails.ucas.ac.cn 联系我。

感谢课程主讲王伟老师和助教唐震老师这学期给予我的帮助！

目录

前言	1
Topic 1 功能分析与建模研讨	3
1.1 Fastjson 与基本建模	3
1.1.1 JSON 格式	3
1.1.2 Java Bean 与序列化	4
1.1.3 Fastjson 的优点	5
1.2 Fastjson 主要功能介绍与分析	7
1.2.1 Fastjson 典型功能函数示范	7
1.2.2 Fastjson 用例描述	7
Topic 2 核心流程设计分析	9
2.1 序列化流程	9
2.1.1 序列化入口 JsonSerializer	10
2.1.2 序列化输出容器 SerializeWriter	10

2.1.3 序列化配置器 SerializeConfig.....	10
2.1.4 解析的具体过程.....	11
2.2 反序列化流程.....	11
2.2.1 反序列化入口 DefaultJSONParser	12
2.2.2 反序列化操作控制 JSONLexer.....	13
2.2.3 反序列化配置器 ParserConfig.....	13
2.2.3 反序列化器的查找与创建.....	13
Topic 3 高级设计模式分析	15
3.1 工厂模式.....	15
3.1.1 工厂模式介绍	15
3.1.2 工厂模式在 Fastjson 中的应用举例-deserializer 生成.....	15
3.2 享元模式（Flyweight）	17
3.3 单例模式（Singleton）	18
Singleton 模式的优点	18
Singleton 模式在 FASTJSON 中应用举例-SerializeConfig	19
3.4 外观模式（Facade Pattern）	20
结语 - 为什么我们需要学习面向对象程序设计？	20
参考文献	20



在这个 TOPIC 的内容中，会对 Fastjson 做一些基本的介绍，从 Fastjson 的起源开始，解释该项目的具体功能，并选取其中的一些主要功能进行具体分析。

1.1 FASTJSON 与基本建模

老生常谈的定义问题。这个问题在阿里巴巴官方的 Fastjson wiki 百科中有标准的答案：“Fastjson 是阿里巴巴的开源 JSON 解析库，它可以解析 JSON 格式的字符串，支持将 Java Bean 序列化为 JSON 字符串，也可以从 JSON 字符串反序列化到 Java Bean。”

好了，从这个定义中我们又得到了两个新问题：什么是 JSON 格式？什么是 Java Bean？此外，序列化 (Serialization) 是将对象的状态信息转换为可以存储或传输的形式过程。在序列化期间，对象将其当前状态写入到临时或持久性存储区。以后，可以通过从存储区中读取或反序列化对象的状态，重新创建该对象。

1.1.1 JSON 格式

JSON (JavaScript Object Notation, JS 对象简谱) 是一种轻量级的数据交换格式。它基于 ECMAScript (欧洲计算机协会制定的 js 规范) 的一个子集，采用完全独立于编程语言的文本格式来存储和表示数据。简洁和清晰的层次结构使得 JSON 成为理想的数据交换语言。易于人阅读和编写，同时也易于机器解析和生成，并有效地提升网络传输效率。

FYI: JSON 格式 <https://baike.baidu.com/item/JSON/2462549?fr=aladdin>

JSON 格式之中，又有 JSON 对象、JSON 对象数组与 JSON 字符串三种典型格式，我们用国科大面向对象课程的学生为例子来说明。

```
{
  "ID": 7355608,
  "name": "张伟",
  "age": 20,
  "topic": "fastjson"
}
```

图 1-1

一位国科大面向对象课程的学术，会有自己的学号与姓名、有自己的年龄，同时，根据老师的课堂要求，学生还需要选择一个开源项目进行分析作为大作业。图 1-1 的代码就展示了这样一位学生的数据。这其实就是一个 JSON 对象。

JSON 对象的特点为：

- 1：数据在花括号中
 - 2：数据以“键：值”对的形式出现（其中键多以字符串形式出现，值可取字符串，数值，甚至其他 json 对象）
 - 3：每两个“键：值”对以逗号分隔（最后一个“键：值”对省略逗号）
- 遵守上面 3 点，便可以形成一个 json 对象。

如图 1-2 所示，如果我们把多位学生对象用方括号放在一起，便又形成了一个 JSON 对象数组。

```
[
  {"ID": 7355608, "name": "张伟", "age": 20, "topic": "fastjson"},
  {"ID": 7355708, "name": "张大", "age": 20, "topic": "dom4j"}
]
```

图 1-2

JSON 对象和 JSON 对象数组都可以转换为 JSON 字符串，JSON 字符串用文本表示一个 JSON 对象的信息，本质是一个字符串。

```
var obj = {a: 'Hello', b: 'World'}; //一个对象，键名也是可以使用引号包裹的
var json = '{"a": "Hello", "b": "World"}'; //一个 JSON 字符串，本质是字符串
```

图 1-3

JSON 对象与 JSON 字符串的转换也是 Fastjson 的功能之一。

1.1.2 JAVA BEAN 与序列化

JavaBean 是一种 JAVA 语言写成的可重用组件。为写成 JavaBean，类必须是具体的和公共的，并且具有无参数的[构造器](#)。JavaBean 通过提供符合一致性设计模式的公共方法将内部域暴露成员属性，set 和 get 方法获取。

本课程进行三分之一之后，同学们对于“对象”应该已经有了基本概念，Java Bean 就是我们所说的对象的一种常规表示方法。如果读者还记得我

们第一次的作业“设计一个汽车的对象”，想必就可以明白这个汽车的对象就是一个 Java Bean。这里我们也给出一个 student 的 Java Bean 对象示例。

```
public class student {  
    /*  
     * Object  
     */  
    private int ID = 7355608 ;  
    private string name = "张伟";  
    private int age = 20;  
    private string topic= "fastjson";  
  
    public int getID() {  
        return ID;  
    }  
  
    public int gettopic() {  
        return topic;  
    }  
  
    public void settopic(string topic) {  
        this.topic = topic;  
    }  
    public int getname() {  
        return name;  
    }  
    public int getage() {  
        return age;  
    }  
}
```

图 1-4

JavaBean 看上去很好地描述了我们的对象，但在进行信息输送时，我们应尽量选择更轻量级大数据交换格式，最好能完全独立于编程语言，否则我们设计的 Java 的对象，如果要传给以 C 语言或 Python 语言构建的平台，就不能直接适用了。

如之前所说，我们的 JSON 就是一种独立于变成语言的数据交换语言，将 JavaBean 转换为 JSON，其实就是将对象的状态信息转换为可以存储或传输的形式过程，也就是**序列化**。在序列化期间，对象将其当前状态写入到临时或持久性存储区。以后，可以通过从存储区中读取或反序列化对象的状态，重新创建该对象。

而 Fastjson，就是实现了对象到 JSON 格式的序列化与反序列化功能的开源解析库。

1.1.3 FASTJSON 的优点

在 JSON 格式出现后，其实有很多出色的 JSON 库被开发与应用，譬如初期的 json-lib，同为开源框架的 Jackson，还有 Google 公司的 Gson。

json-lib 最开始的也是应用最广泛的 json 解析工具，json-lib 不好的地方确实是依赖于很多第三方包，对于复杂类型的转换，json-lib 对于 json 转换成 bean 还有缺陷在功能和性能上面都不能满足现在互联网化的需求。

相比 json-lib 框架，Jackson 所依赖的 jar 包较少，简单易用并且性能也要相对高些。Jackson 对于复杂类型的 json 转换 bean 会出现问题，一些集合 Map, List 的转换出现问题。且 Jackson 对于复杂类型的 bean 转换 Json，转换的 json 格式不是标准的 Json 格式。

至于 Gson，确实是目前功能最全的 Json 解析库，Gson 当初是为因应 Google 公司内部需求而由 Google 自行研发而来，但自从在 2008 年五月公开发布第一版后已被许多公司或用户应用。Gson 的应用主要为 toJson 与 fromJson 两个转换函数，无依赖，不需要例外额外的 jar，能够直接跑在 JDK 上。在功能上面无可挑剔，但是性能上面比 FastJson 有所差距。

那么我们为什么要选择 Fastjson 呢？这里给出官方文档中 Fastjson 比较突出的几个特点。

1 速度快

Fastjson 相对其他 JSON 库的特点是快，从 2011 年 Fastjson 发布 1.1.x 版本之后，其性能从未被其他 Java 实现的 JSON 库超越。

FYI：为什么 Fastjson 能这么快

https://blog.csdn.net/xf_87/article/details/51872336

2 使用广泛

Fastjson 在阿里巴巴大规模使用，在数万台服务器上部署，Fastjson 在业界被广泛接受。在 2012 年被开源中国评选为最受欢迎的国产开源软件之一。

3 测试完备

Fastjson 有非常多的 testcase，在 1.2.11 版本中，testcase 超过 3321 个。每次发布都会进行回归测试，保证质量稳定。

4 使用简单

```
String text = JSON.toJSONString(obj); //序列化
VO vo = JSON.parseObject("{...}", VO.class); //反序列化
```

fastj
son
的
API

十分简洁。

图 1-5

5 功能完备

支持泛型，支持流处理超大文本，支持枚举，支持序列化和反序列化扩展。

1.2 FASTJON 主要功能介绍与分析

考虑到在第二节我们会对 Fastjson 的核心流程作细致分析，这里我们只着重选择一些案例来说明 Fastjson 的功能。

1.2.1 FASTJON 典型功能函数示范

这里给出 Fastjson 库中几个典型的实现对象类型转换的函数。

```
// 将对象转换为字符串
String str = JSON.toJSONString(infoDo);
// 字符串转换为对象
InfoDo infoDo = JSON.parseObject(strInfoDo, InfoDo.class);
// String 转 Json对象
JSONObject jsonObject = JSONObject.parseObject(jsonString);
// json对象转string
JSONObject jsonObject = JSONObject.parseObject(str);
// json对象转字符串
String jsonString = jsonObject.toJSONString();
```

图 1-6

1.2.2 FASTJSON 用例描述

考虑到 Fastjson 本身针对的只是一个较细节的序列化（反序列化）操作，他的用例描述应当是很简洁的，我们在 Fastjson 库中同样找到了许多用于获取数据的 get 类型函数，所以我们把获取数据这一步也考虑进去，对于一个将 JSON String 类型转化为 JSON 对象的流程，用例描述如下：

【用例描述】

1. GetString（）函数获取我们的字符串类型的 JSON 字符串对象。
2. 调用 parseObject(String text)，这个函数会将 JSON 字符串转化为 JSON 对象（代码见下图）
 - 2.1 内部调用了 parse()方法，调用底层的 DefaultJSONParser 解析类进行转化，在转化失败时，抛出 can not cast to JSONObject 异常。
 - 2.2 返还 JSON 对象


```

public static JSONObject parseObject(String text) {
    Object obj = parse(text);
    if (obj instanceof JSONObject) {
        return (JSONObject)obj;
    } else {
        try {
            return (JSONObject)toJSON(obj);
        } catch (RuntimeException var3) {
            throw new JSONException("can not cast to JSONObject.", var3);
        }
    }
}

```

图 1-7

该方法不仅能实现 json 字符串向 json 对象的转化,经过重载之后,还能实现 json 字符串向 javabean 对象的转化。

这一章完成了对 Fastjson 的功能与作用的基本介绍,同时以较为低层的视角对一些功能作了介绍。但这距离体现面向对象思想的本质还是远不够的。譬如我们上面提到的 `parseObject`,它在 Fastjson 库中可不止上面一个函数(见下图)。为什么会设计这么多种 `parseObject`?他们之间的关系是什么?是如何实现的?有趣的问题很多,在从课堂学到更多知识后,我们会进入下一部分,《核心流程设计分析》,将视角放在 Fastjson 的类、接口、层次结构上进行分析。

```

● S parseObject(byte[], int, int, Charset, Type, Feature...) <T> : T
● S parseObject(byte[], int, int, CharsetDecoder, Type, Feature...) <T> : T
● S parseObject(byte[], Type, Feature...) <T> : T
● S parseObject(char[], int, Type, Feature...) <T> : T
● S parseObject(InputStream, Type, Feature...) <T> : T
● S parseObject(InputStream, Charset, Type, Feature...) <T> : T
● S parseObject(String) : JSONObject
● S parseObject(String, Feature...) : JSONObject
● S parseObject(String, TypeReference<T>, Feature...) <T> : T
● S parseObject(String, Class<T>) <T> : T
● S parseObject(String, Class<T>, ParseProcess, Feature...) <T> : T
● S parseObject(String, Class<T>, Feature...) <T> : T
● S parseObject(String, Type, ParseProcess, Feature...) <T> : T
● S parseObject(String, Type, Feature...) <T> : T
● S parseObject(String, Type, ParserConfig, ParseProcess, int, Featu

```

图 1-8

TOPIC 2 核心流程设计分析

在这一章的内容中，我们重点介绍 Fastjson 的两大核心功能：序列化与非序列化。相对于在 TOPIC 1 中对两种功能所作的简要介绍。在这一章里我们会相对深入的在代码结构层面（但没有细节到代码具体设计层面）分析这两种功能的流程与实现。同时，这里也会给出一个相对完整的实现序列化与反序列化的流程。

2.1 序列化流程

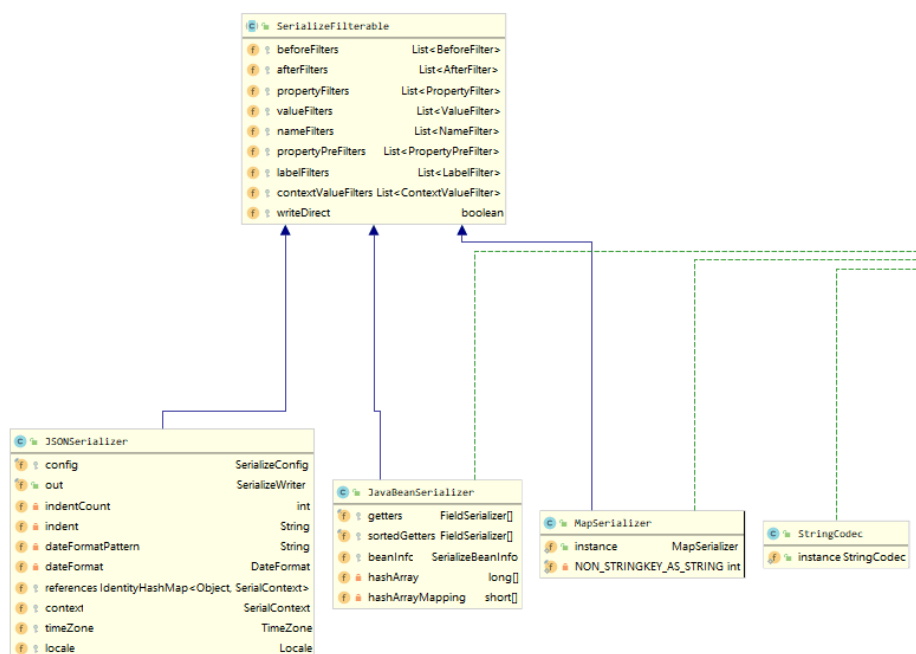


图 2-1

图 2-1 是运用 IntelliJ 插件生成的 Serializer 包的类图的一小部分。除了图上的内容，该插件还可以展现各类的方法，可以说蛮方便。

FYI: 类图插件用法: <https://blog.csdn.net/ywb201314/article/details/88029517>

回顾一下，序列化指的是将各种对象转换为 json 字符串的过程。平常我们经常用到的是 `JSON.toJSONString()` 这个静态方法来实现序列化。其实 `JSON` 是一个抽象类，该类实现了 `JSONAware`（转为 json 串）和 `JSONStreamAware`（将 json 串写入 `Appendable` 中）的接口，同时又是 `JSONArray`（内部实现就是个 `List`）和 `JSONObject`（内部实现就是个 `Map`）的父类。

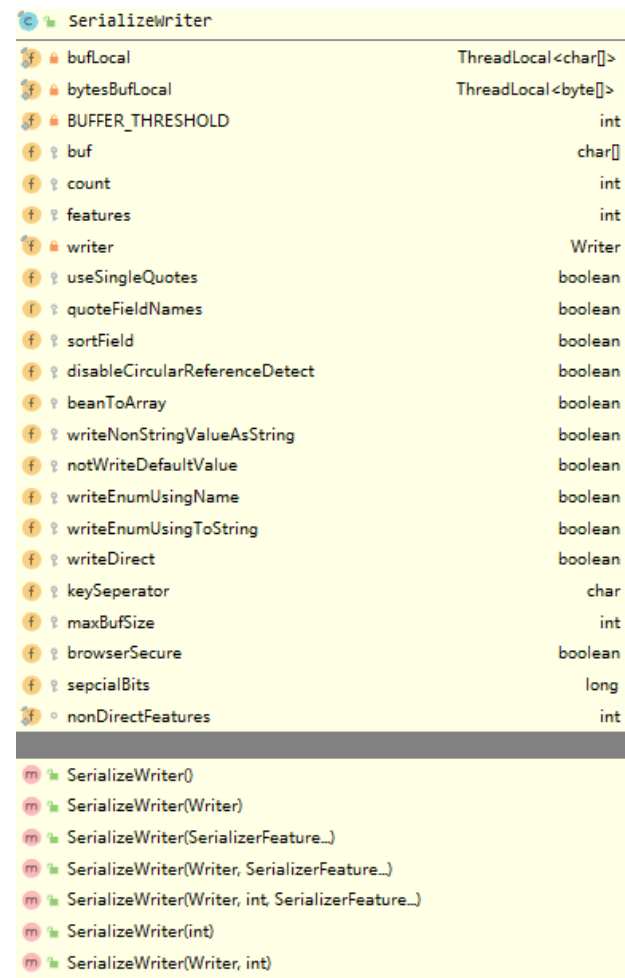
整个序列化过程可以简单概括如下：新产生的一个数据保存器，储存在序列化过程中产生的数据；产生统一的 json 序列化器，其中使用了

outWriter，此类即 json 序列化的统一处理器；调用序列化方法开始序列化对象，以产生 json 字符串信息；最后，返回已经产生并储存的 json 信息。

2.1.1 序列化入口 JSONSERIALIZER

JSONSerializer 类相当于一个序列化组合器，它将上层调用、序列化配置、具体类型序列化实现、序列化字符串拼接等功能组合在一起，方便外部统一调用。该类有几个重要的成员，SerializeConfig、SerializeWriter、各种 Filter 列表、DateFormat、SerialContext 等。某种意义上来说，JSONSerializer 类体现了高级设计模式中的外观模式。

2.1.2 序列化输出容器 SERIALIZEWRITER



SerializeWriter	
bufLocal	ThreadLocal<char[]>
bytesBufLocal	ThreadLocal<byte[]>
BUFFER_THRESHOLD	int
buf	char[]
count	int
features	int
writer	Writer
useSingleQuotes	boolean
quoteFieldNames	boolean
sortField	boolean
disableCircularReferenceDetect	boolean
beanToArray	boolean
writeNonStringValueAsString	boolean
notWriteDefaultValue	boolean
writeEnumUsingName	boolean
writeEnumUsingToString	boolean
writeDirect	boolean
keySeparator	char
maxBufSize	int
browserSecure	boolean
sepcialBits	long
nonDirectFeatures	int
Methods:	
SerializeWriter()	
SerializeWriter(Writer)	
SerializeWriter(SerializerFeature...)	
SerializeWriter(Writer, SerializerFeature...)	
SerializeWriter(Writer, int, SerializerFeature...)	
SerializeWriter(int)	
SerializeWriter(Writer, int)	

SerializeWriter 是一个用于储存在序列化过程中产生的数据信息。方法总共可以分五个部分，第一个部分是针对 writer 基本功能一个扩展，即支持输出 int, 字符，以及字符数组，追加字符数组（包括字符串）等；第二个部分提供了写整形和长整形的基本方法；第三个部分是提供写基本数据类型数组的支持；第四个部分是提供写一个数字+一个字符的形式，比如数据+[,],/,这种格式；第五个部分是提供写数据串，主是是针对字符串追加双引号或单引号（以支持标准 json）。

图 2-2

2.1.3 序列化配置器 SERIALIZECONFIG

SerializeConfig 的主要功能是配置并记录每种 Java 类型对应的序列化类（ObjectSerializer 接口的实现类）；SerializeConfig 是全局唯一的（或者说单例的）；

2.1.4 解析的具体过程

前面提到过，解析方法由统一的接口所定义，为 `write(JSONSerializer serializer, Object object)`，由 `ObjectSerializer` 提供。带两个参数，第一个参数，即为解析的起点类 `jsonSerializer`，此类封装了我们所需要的 `outWriter` 类，需要时只需要从此类取出序列化器即可。第二个参数为我们所要解析的对象，在各个子类进行实现时，可将此对象通过强制类型转换，转换为所需要的类型即可。

```
public static void write(SerializeWriter out, Object object) {  
    JSONSerializer serializer = new JSONSerializer(out);  
    serializer.write(object);  
}
```

图 2-3

那么，有没有可能，当前要处理的对象并没有直接对应的序列化器存在于我们的 `outWriter` 中呢？这是有可能的，此时解析主要做两件事：基于数据类型特点输出所特有的字符包装内容，以及，基于数据类型特点转换为 `outWriter` 所能识别的内容。通过逐步解析来实现对整个对象的转化。

对于复杂的数据类型，在实现过程中，可能需要递归地调用 `JsonSerializer` 来进行序列化。这个时候就体现了上面说的“逐步解析”的想法。比如处理一个对象集合时，除需要处理集合本身之外，还需要处理集合中的每一个对象，这时又是一个新的解析过程。由于使用了同一个 `jsonSerializer`，所以在进行数据处理时，输出的数据会按照在解析过程中的顺序，顺序地写入到 `outWriter` 中，这样即保证了数据的正确性。

2.2 反序列化流程

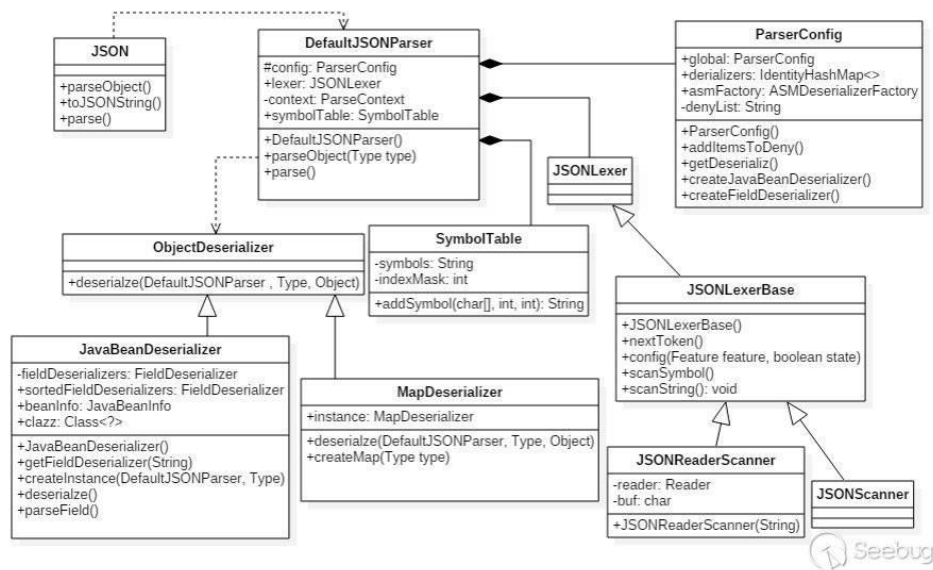


图 2-4

这里用 2017 年的一篇博文中的类图来简单说明反序列化各模块的整体关系。

FYI:图源博文: <http://xxlegend.com/2017/12/06/基于JdbcRowSetImpl的Fastjson%20RCE%20PoC构造与分析/>

反序列化,就是将 json 串转化为对应的 java 对象。反序列化的主要的功能都是在 DefaultJSONParser 类中实现的,在这个类中会应用其他的一些外部类来完成后续操作。ParserConfig 主要是进行配置信息的初始化,JSONLexer 主要是对 json 字符串进行处理并分析,反序列化在 JavaBeanDeserializer 中处理。

2.2.1 反序列化入口 DEFAULTJSONPARSER

反序列化实现了 parse()、parseObject()、parseArray()等将 json 串转换为 java 对象的静态方法,这些方法都在 DefaultJSONParser 中组合等待调用。DefaultJSONParser 类相当于序列化的 JSONSerializer 类,是个功能组合器

DefaultJSONParser 的初始化是在 parseObject 中调用并完成的。初始化过程中还包括了 lexer.token 的设置与其他相关类的实例化。

```
public DefaultJSONParser(JSONLexer lexer) { this(lexer, ParserConfig.getGlobalInstance()); }

public DefaultJSONParser(JSONLexer lexer, ParserConfig config) { this((Object)null, lexer, config); }

public DefaultJSONParser(Object input, JSONLexer lexer, ParserConfig config) {
    this.dateFormatPattern = JSON.DEFAULT_DATE_FORMAT;
    this.contextArrayIndex = 0;
    this.resolveStatus = 0;
    this.extraTypeProviders = null;
    this.extraProcessors = null;
    this.fieldTypeResolver = null;
    this.autoTypeAccept = null;
    this.lexer = lexer;
    this.input = input;
    this.config = config;
    this.symbolTable = config.symbolTable;
    int ch = lexer.getCurrent();
    if (ch == '[') {
        lexer.next();
        ((JSONLexerBase)lexer).token = 12;
    } else if (ch == '[') {
        lexer.next();
        ((JSONLexerBase)lexer).token = 14;
    } else {
        lexer.nextToken();
    }
}
```

图 2-5

2.2.2 反序列化操作控制 JSONLEXER

JSONLexer 是个接口类，定义了各种当前状态和操作接口。JSONLexerBase 是对 JSONLexer 实现的抽象类，类似于序列化的 SerializeWriter 类，专门解析 json 字符串，并做了很多优化。实际使用的是 JSONLexerBase 的两个子类 JSONScanner 和 JSONLexerBase，前者是对整个字符串的反序列化，后者是接 Reader 直接序列化。JSONLexerBase 的成员很多，例如表示字符的 token，扫描字符串用的 sbuf，以及关键的，控制反序列化的 feature。

Feature 关乎反序列化特性的配置，同序列化的 feature 是通过 int 的位或来实现其特性开启还是关闭的。例如默认配置中有要求：AutoCloseSource | UseBigDecimal | AllowUnQuotedFieldNames 等，这意味着在进行反序列化时表示检查 json 串的完整性 and 转换数值使用 BigDecimal and 允许接受不使用引号的 filedName 等。这些参数也是可以通过其他途径配置的。

FYI:feature 指定序列化特性举例：

https://blog.csdn.net/weixin_34249367/article/details/86207059

2.2.3 反序列化配置器 PARSECONFIG

同 SerializeConfig，该类也是全局唯一的解析配置；与 SerializeConfig 不同的是，配置类和对应反序列类的 IdentityHashMap 是该类的私有成员，构造函数的时候就将基础反序列化类加载进入 IdentityHashMap 中。

```
private ParserConfig(ASMDeserializerFactory asmFactory, ClassLoader parentClassLoader, boolean fieldBased) {
    this.deserializers = new IdentityHashMap();
    this.asmEnable = !ASMUtils.IS_ANDROID;
    this.symbolTable = new SymbolTable( tableSize: 4096);
    this.autoTypeSupport = AUTO_SUPPORT;
    this.jacksonCompatible = false;
    this.compatibleWithJavaBean = TypeUtils.compatibleWithJavaBean;
    this.denyHashCodes = new long[] {-8720046426850100497L, -8165637398350707645L, -8109300701639721088L, -80833
    long[] hashCodes = new long[AUTO_TYPE_ACCEPT_LIST.length + 1];

    for(int i = 0; i < AUTO_TYPE_ACCEPT_LIST.length; ++i) {
        hashCodes[i] = TypeUtils.fnv1a_64(AUTO_TYPE_ACCEPT_LIST[i]);
    }
}
```

图 2-6

因此，在进行反序列化时的前一部分操作，和序列化其实是相似的：创建 ParserConfig 对象，包括初始化内部注册反序列化方案和 features 的配置；添加反序列化拦截器；根据具体类型，将反序列化器实例的查找委托给 parser 对象；解析对象内部引用。

2.2.3 反序列化器的查找与创建

在反序列化过程中，对于相应的目标对象，需要找到对应的反序列化器来进行转换。

```

public <T> T parseObject(Type type, Object fieldName) {
    int token = this.lexer.token();
    if (token == 8) {
        this.lexer.nextToken();
        return null;
    } else {
        if (token == 4) {
            if (type == byte[].class) {
                byte[] bytes = this.lexer.bytesValue();
                this.lexer.nextToken();
                return bytes;
            }

            if (type == char[].class) {
                String strVal = this.lexer.stringVal();
                this.lexer.nextToken();
                return strVal.toCharArray();
            }
        }

        ObjectDeserializer deserializer = this.config.getDeserializer(type);

        try {
            return deserializer.getClass() == JavaBeanDeserializer.class ? ((JavaBeanDeserializer)deserializer).
        } catch (JSONException var6) {
            throw var6;
        } catch (Throwable var7) {
            throw new JSONException(var7.getMessage(), var7);
        }
    }
}

```

图 2-7

查看图 2-7 的中间部分，FASTJSON 在 `parseObject` 中委托 `DefaultJSONParser` 对象进行解析。这里的 `config` 是之前初始化的 `ParserConfig`。Type 是用来查找反序列化器的参数。

1. 生成反序列化器时，优先查找已经创建并存储的反序列化器，如果没有，则重新生成，大致流程如下：从已经注册的方案中查找 `class` 的反序列化器；
2. 如果找不到且目标是引用类型，用 `getDeserializer` 方法递归查找；
3. 如果又找不到，而且目标是泛型，先获取泛型的原始类型，判断其是否为引用类型，再用不同方式递归查找；
4. 进行通配符和限定类型的判断；
5. 最后，如果都不满足，就返回默认的反序列化器 `JavaObjectDeserializer`

这一部分的流程体现了两种重要的高级设计模式，我们接下来会在第三章中重点讲解。

TOPIC 3 高级设计模式分析

在这一章的内容中，我们会结合各种高级设计模型来对 Fastjson 的架构进行探讨。主要分为两块：工厂模式在 Fastjson 中的应用，以及，Singleton 及其他模式在 Fastjson 中的应用。在课堂演示时，工厂部分会作为笔者介绍高级设计模式的例子展示，在第三次汇报中，则会把重点放在 Singleton 及其他模式在 Fastjson 中的运用上。

3.1 工厂模式

3.1.1 工厂模式介绍

“Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.”

在基类中定义创建对象的一个接口，让子类决定实例化哪个类。工厂方法让一个类的实例化延迟到子类中进行。

这是 GOF 中对工厂模式的定义。

FYI: 三种工厂模式介绍: <https://www.jianshu.com/p/6b347d596241>

我们为什么要使用工厂模式呢？一般有三种原因：

- (1) 解耦：把对象的创建和使用的过程分开
- (2) 降低代码重复: 如果创建某个对象的过程都很复杂，需要一定的代码量，而且很多地方都要用到，那么就会有很多的重复代码。
- (3) 降低维护成本：由于创建过程都由工厂统一管理，所以发生业务逻辑变化，不需要找到所有需要创建某个对象的地方去逐个修正，只需要在工厂里修改即可，降低维护成本。

3.1.2 工厂模式在 FASTJSON 中的应用举例-DESERIALIZER 生成

工厂模式作为最常用的设计模式之一，在 Fastjson 中自然也有出现。我们不妨想一想 Fastjson 哪里会用到工厂：Fastjson 的主要功能是序列化与反序列化，那么自然会有对应的方法与处理类来实现各种数据结构到对象或其他数据结构的互相转化过程。然而，对象与各种结构体可以有各种各样的组成，如果我们需要设计一个处理类或方法去实现针对某一种对象或结构体的序列化，这个类或方法的设计可能也会有成千上万种。

为了降低代码重复与维护成本，我们当然可以在生成序列化与反序列化处理类的地方运用工厂模式，即设计一个工厂来生产各种各样的处理类，更严格的说，我们先将需要处理的对象（子类）的特征给予工厂，工厂根据收到的信息与要求来量身定做做一个处理类。

FASTJSON 生成反序列化器时，就是这样的情况。

```

public ObjectDeserializer getDeserializer(Type type) {
    ObjectDeserializer deserializer = (ObjectDeserializer)this.deserializers.get(type);
    if (deserializer != null) {
        return deserializer;
    } else if (type instanceof Class) {
        return this.getDeserializer((Class)type, type);
    } else if (type instanceof ParameterizedType) {
        Type rawType = ((ParameterizedType)type).getRawType();
        return rawType instanceof Class ? this.getDeserializer((Class)rawType, type) : this.getDeserializer(rawType);
    } else {
        if (type instanceof WildcardType) {
            WildcardType wildcardType = (WildcardType)type;
            Type[] upperBounds = wildcardType.getUpperBounds();
            if (upperBounds.length == 1) {
                Type upperBoundType = upperBounds[0];
                return this.getDeserializer(upperBoundType);
            }
        }

        return JavaObjectDeserializer.instance;
    }
}

```

图 3-1

我们对照上图，回顾一下反序列化时的流程：

6. 从已经注册的方案中查找 class 的反序列化器；
7. 如果找不到且目标是引用类型，用 `getDeserializer` 方法递归查找；
8. 如果又找不到，而且目标是泛型，先获取泛型的原始类型，判断其是否为引用类型，再用不同方式递归查找；
9. 进行通配符和限定类型的判断；
10. 最后，如果都不满足，就返回默认的反序列化器 `JavaObjectDeserializer`

如果我们进行到了流程的最后一步，即无论怎么找都找不到对目标 class 进行操作的反序列化器，我们只有自行生产一个：根据类名和 `propertyNameingStrategy` 生成 `beanInfo`，之后利用 asm 工厂类的 `createJavaBeanDeserializer` 生成处理类。这个 `createJavaBeanDeserializer` 便是我们这里的一个工厂例子。

我们来看 `createJavaBeanDeserializer` 的部分代码。

```

public ObjectDeserializer createJavaBeanDeserializer(ParserConfig config, JavaBeanInfo beanInfo)

```

图 3-2

该方法的参数有两个：配置并记录每种 Java 类型对应的反序列化类的 `ParserConfig` 和之前生成的记录类名等信息的 `beanInfo`。通过这些参数，

`createJavaBeanDeserializer` 首先创建了一个新的 `class`（见下图），这个 `class` 有对目标 `class` 的反序列化方法，然后基于这个 `class` 实例化一个对象 `instance`，作为反序列化的处理类对象。

```
ClassWriter cw = new ClassWriter();
cw.visit( version: 49, access: 33, classNameType, ASMUtills.type(JavaBeanDeserializer.class), (String[])null);
this._init(cw, new ASMDeserializerFactory.Context(classNameType, config, beanInfo, initVariantIndex: 3));
this._createInstance(cw, new ASMDeserializerFactory.Context(classNameType, config, beanInfo, initVariantIndex: 4));
this._deserialize(cw, new ASMDeserializerFactory.Context(classNameType, config, beanInfo, initVariantIndex: 5));
this._deserializeArrayMapping(cw, new ASMDeserializerFactory.Context(classNameType, config, beanInfo, initVariantIndex: 6));
byte[] code = cw.toByteArray();
Class<?> deserClass = this.classLoader.defineClassPublic(classNameFull, code, off: 0, code.length);
Constructor<?> constructor = deserClass.getConstructor(ParserConfig.class, JavaBeanInfo.class);
Object instance = constructor.newInstance(config, beanInfo);
return (ObjectDeserializer)instance;
}
```

图 3-3

我们回顾工厂模式的特征：定义一个用于创建对象的接口，让子类决定实例化哪一个类；一个类通过其子类来指定创建哪个对象。

一个经典的例子是一个生产 `ball` 的工厂，如果得到的指令是“volleyball”，就生成一个 `volleyball` 对象返回，如果是“basketball”，就生成一个 `basketball` 对象返回。其本质实际上是生成基于给定信息的实体对象。这里的 `createJavaBeanDeserializer` 也是在做一样的事，只是它所得到的给定信息不只是一个代表 `name` 的字符串，还有更多目标 `class` 的特征。

3.2 享元模式（FLYWEIGHT）

上一节中我们说到，`createJavaBeanDeserializer` 在做的是通过传入的参数量身定做做一个类，然后生成需要的对象。这是工厂模式的实现。

我们把视野拉高一点，再重新看一看图 3-1，和反序列化时的流程，就可以发现一个工厂模式的好基友！它叫享元模式！

享元模式的主要目的是实现对象的共享，减少创建对象的数量，从而减少内存占用和提高性能。这种类型的设计模式属于结构型模式，它提供了减少对象数量从而改善应用所需的对象结构的方式。享元模式尝试重用现有的同类对象，如果未找到匹配的对象，则创建新对象。通常与工厂模式一起使用。

图 3-1 的查找反序列化器的流程，其实就是享元模式的例子：方法先根据 `fieldType` 获取已缓存的解析器，如果没有则根据 `fieldClass` 获取已缓存的解析器，否则根据注解的 `JSONType` 获取解析器，否则通过当前线程加载器加载的 `AutowiredObjectDeserializer` 查找解析器，否则判断是否为几种常用泛

型（比如 `Collection`、`Map` 等），最后通过 `createJavaBeanDeserializer` 来创建对应的解析器。在 `createJavaBeanDeserializer` 前的各式查找，其实就是在对象的“共享池”中查找已有的解析器。

我们可以想象我们码代码二十年后赚了大钱开了一家手机制作工厂，里面已经有一些手机的制作流程（也就是相应的反序列化类）。用户想来买一个手机，告诉我们品牌型号和参数（即目标 `class`），我们首先看这种手机我们是不是已经在做了（即查找已有的反序列化器），如果能找到就把现成货给他；如果没有，那我们得首先设计一套制作这款手机的流程（也就是带有相应反序列化器的类），然后生产一个手机（即生成对象）返回给顾客；这当然是一笔亏本生意，但这个流程我们也会保存下来以便之后查找。这样的例子也正反映了工厂模式的本质：用户给工厂一些参数，工厂基于这些参数生成相应的实例对象返回给顾客，利用了面向对象的多态性。工厂不直接生产这个对象，而是通过构造相应的流程与流水线来生产，即依赖子类生产对象。

那么在我们这家手机工厂中，制作一款新手机的过程就是工厂模式，而顾客提出需求时工厂优先查找是否存在已有对象满足条件，实际上就是享元模式。

在反序列化中是这样的模式，在序列化过程中，其实也有相似的流程，就不再赘述了。

3.3 单例模式（`SINGLETON`）

单例模式（`Singleton`）是一种常用的设计模式。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

`SINGLETON` 模式的优点

这样的模式有几个好处：

在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例避免对资源的多重占用，也避免了核心控制类的唯一性，（比如一个军队出现了多个司令员同时指挥，肯定会乱成一团），所以只有使用单例模式，才能保证单例的实例类能独立控制整个流程。

SINGLETON 模式在 FASTJSON 中应用举例-SERIALIZECONFIG

在第二章的内容中我们介绍了 FASTJSON 中的序列化与反序列化，其中提到了 `SerializeConfig` 类。

```
public class SerializeConfig {  
    public static final SerializeConfig globalInstance = new SerializeConfig();  
}
```

图 3-4

一个 `SerializeConfig` 对象用来保存和匹配类型信息，同时维护一个名为 `serializers` 的 `IdentityHashMap`。

而在图 3-4 中我们创建了 `SerializeConfig` 类的对象 `globalInstance`，这其实就是 Singleton 模式的体现。这个 `globalInstance` 是 `Fastjson` 中唯一一个 `SerializeConfig` static 对象，其被用于维护存放待序列化数据和序列化器的 `serializers`。

此外，序列化器类也都维护了一个 `instance` 对象（某种意义上，我们在工厂模式介绍中提到的类也可能是这样的），也可以是单例模式的一种。

```
public class FloatCodec implements ObjectSerializer, ObjectDeserializer {  
    private NumberFormat decimalFormat;  
    public static FloatCodec instance = new FloatCodec();  
  
    public FloatCodec() {  
    }  
}
```

图 3-5

雷正宇学长^[7]在其 `Fastjson` 报告中也提出过这一点，他认为虽然严格意义上不完全对，这些类仍可以算作 Singleton 模式。我这里持相同的观点。严格意义上 Singleton 模式不成立的主要原因是这些类的构造器是公开的，且没有通过其他措施来限制类的实例数量（图 3-5）。然而在以“快”为核心目的的 `Fastjson` 中，构造并存放两个相同的序列化器对象，只会让索引序列化器的操作效率变低、同时占据更多空间，这自然是不明智的。当前已有的代码也不存在于其他地方构造相同序列化器的操作。

而单例模式的三种要求，在这里都是被满足的，即：单例类只能有一个实例；必须自己创建自己的唯一实例，以及，必须给其他对象提供这一实例。所以单例模式这一点应当是成立的。

至于为什么 `Fastjson` 没有严格地去实现单例，不妨认为是对设计模式的总原则-开闭原则的遵循，即对扩展开放、对修改关闭。如果只会 `Fastjson` 为了进一步提速考虑并行运行与存储，那么也不是没有可能需要创建多个相同类的实例。

3.4 外观模式（FACADE PATTERN）

外观模式（**Facade Pattern**）隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。这种类型的设计模式属于结构型模式，它向现有的系统添加一个接口，来隐藏系统的复杂性。

在之前提到的反序列化中，实现了 `parse()`、`parseObject()`、`parseArray()` 等将 json 串转换为 java 对象的静态方法。这些方法的实现，实际托付给了 `DefaultJSONParser` 类。那么 `DefaultJSONParser` 类相当于序列化的 `JSONSerializer` 类，其实就是个功能组合器，它将上层调用、反序列化配置、反序列化实现、词法解析等功能组合在一起，相当于设计模式中的外观模式，供外部统一调用。

```
public class DefaultJSONParser implements Closeable {
    public final Object input;
    public final SymbolTable symbolTable;
    protected ParserConfig config;
    private static final Set<Class<?>> primitiveClasses = new HashSet();
    private String dateFormatPattern;
    private DateFormat dateFormat;
    public final JSONLexer lexer;
    protected ParseContext context;
```

图 3-6

出于篇幅问题，这一模式就不再详细探讨了。

结语 - 为什么我们需要学习面向对象程序设计？

著名美国数学家和数学教育家 G.波利亚写过一本书叫《怎样解题》，在这本书中讨论的是数学中发现和发明的方法和规律，但是对在其他任何领域中怎样进行正确思维都有明显的指导作用。很有意思的一点是，这本书虽然叫《怎样解题》，通篇却没有一个公式，既不教求导，也不教积分，而是将解题分为四个步骤进行分析，“弄清问题”、“拟定计划”、“实现计划”和“回顾反思”。

这需要人教么？读者一开始必然会疑惑，这些在他们解题时下意识采用的思维模式，似乎并不需要作为多么重要的经验来讲授，毕竟不管看不看书，大家都知道这么做。

但事实上，这本书确实给了读者们极大的帮助。在解决实际问题时，读者虽然下意识思考过一些解决问题的步骤，但可能又忽略许多解决问题的方法和细节。按波利亚提出的这些问题和建议去寻找解法，其实是把寻找和发现解法的思维过程进行分解，使我们对解题的思维过程看得见，摸得着，易于操作。最终形成可靠严谨的思维模式。

对笔者而言，面向对象程序设计所教授的，也是类似的东西，即教会学生如何去思考程序的设计，而非盲目地下手敲键盘，做出看似当前合格但不具实用性的作品。

举个例子，我们第三章所介绍的高级设计模式，在 23 种模式最初被提出时，人们的评价是褒贬不一的，有人说这些设计模式并没有讲什么实质的东西，太过注重套模式反而会显得走形式，更别说很多设计模式我们下意识就能写出来。

但下意识能写出来，就意味着不需要去掌握么？本报告介绍的单例模式，无非就是创建唯一对象的过程，很容易不经意写出来，但每次设计时都一定能“不经意”么？23 种设计模式都能“不经意”写出么？

答案自然是否定的。

我们学习面向对象、设计模式，是为了把每次设计程序时“该怎么做才能满足需求”的填空题，变成“该选哪一种设计模式才能满足需求”的选择题。每次遇到问题都临时想怎么设计，不仅浪费时间，还不一定能达成效果。我们固然有自己的经验，但如果前人的总结更加完善、全面，自然是要学的。（某种意义上，“调用前人总结的框架与设计模式，如果可以满足需求就使用”，也正是享元模式的体现）。当需要解耦和降低代码重复时，我们知道用工厂模式去创建对象，当需要设计一个大的 Project 时，我们知道遵守高级设计模式可以做到对扩展开放，对修改关闭，立足于比代码更高一级的位置给予我们指导，这就是面向对象思维的价值所在。

闲话就不多赘述了。本学期的 Fastjson 学习让笔者得以把学到的理论设计落实到真切存在的 Project 中，加深理解，帮助很大。王老师的课程也十分 nice，如果给分再 nicer 一点就是 nicest！

再次感谢老师同学们这学期对我的帮助！

刘骐鸣

2019.12.27

参考文献

- [1]百度百科: <https://baike.baidu.com/item/JSON/2462549?fr=aladdin>
- [2]CSDN 博客: <https://blog.csdn.net/srj1095530512/article/details/82529759>
- [3]官方说明: <https://github.com/alibaba/fastjson/wiki/Quick-Start-CN>
- [4]CSDN 博客: <https://www.cnblogs.com/peiyangjun/p/8146381.html>
- [5]官方实践文档: <http://kimmking.github.io/2017/06/06/json-best-practice/>
- [6]CSDN 博客-23 种高级设计模式:
https://www.cnblogs.com/swordfall/p/10742412.html#auto_id_16
- [7]雷正宇学长 gitbook:
<https://842376130.gitbook.io/fastjson-learning-report/di-san-zhang-xiao-cha-qu>
- [8]博客: <http://xxlegend.com/2017/12/06/基于JdbcRowSetImpl的Fastjson%20RCE%20PoC构造与分析/>
- [9]CSDN 博客: https://blog.csdn.net/liu_shi_jun/article/details/95083230