**Fastjson源码阅读报告**

**张朔铭 2017K8009929050**

**摘要：**

这是本人于中国科学院大学大三学期面向对象程序设计课程中的课程实践，不得不说，这是一个极其繁忙的学期，因而用在该源码阅读上的时间可能不够充足，但是每一点阅读的收获我都会详实地记录在本报告中。

本人想试图从一个独特的角度来记录我，一个初步接触面向对象程序设计者的收获，记录一切思维的转换与改变的过程。特别是各种工业软件较个人作业代码的各种差别，都让我对面向对象思想产生进一步的深化理解。其内容会充斥本报告，略显傻萌，请读者见谅。

回到主题，本报告对 fastjson 这一开源项目进行学习分析，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，在之前示例用法基础上进一步理解核心任务：序列化与反序列化，对两种功能实现的流程与参与其中的类与对象做了介绍；第三章，高级设计意图分析，结合23 种高级设计模式，以及面向对象程序设计的5大核心思想，对在 fastjson 中发现的几种设计模式进行分析理解，探讨这些设计模式对于项目的意义。

最后，特别感谢面向对象程序设计课程的主讲王伟老师和助教唐震老师这学期的精彩课程与大量帮助。要是能够拿到高分就更好了!

**第一部分：功能分析与建模**

开始阅读一个源码，首先得知道，它是干什么的，fastjson的官方定义，我从阿里的官方文档中选取了其wiki描述的一部分来介绍：

“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。”

作为这个领域的初接触者，这又抛给了我两个新的问题（套娃的开始），什么是JSON格式字符串，什么又是JavaBean？我们必须先了解fastjson解析器所经手的这两类数据结构。

**JSON:**

首先看百科定义：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。

简而言之，JSON本身只是一种不依赖具体程序语言的数据交换格式，JSON格式又可具体分为JSON对象，JSON对象数组，JSON字符串三种典型格式。

我们分析JSON格式的特点，它比较简洁，且是对象（内部由键值对表示），数组，数，字符串和字面值的组合。简单举例一个如下：

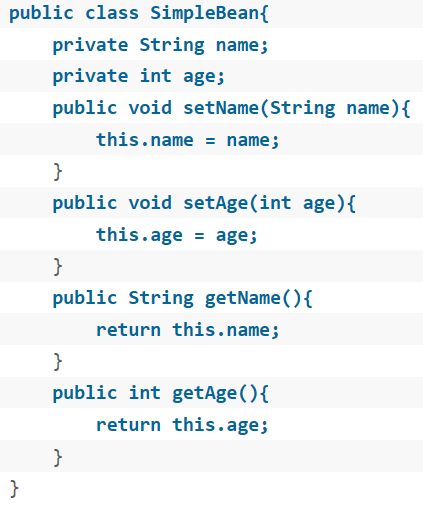
{"a": 1, "b": [1, 2, 3]}

包含一个对象，其属性分别为a,b; 其中a是一个整型数,其值为1，b是一个数组，其内容为[1,2,3]。在作为toJSONString输出时，其输出即为”{"a": 1, "b": [1, 2, 3]}”。

**JavaBean:**

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。如果在一个类中只包含属性、setter、getter方法，那么这种类就成为简单JavaBean。

我们来看一个极为简单的JavaBean的例子，基本初学者可以把JavaBean就理解为某个class的对象就好。



继续之前JSON的例子，JavaBean对象本身也与之非常类似，一个JavaBean对象本身可以很复杂，但无非是只包括属性，方法，事件的公共非抽象类。因此与之也非常相似。

与上面例子中的示例JSON对应的JavaBean为：

Public class vo{

Int a=1;

Int[] b=[1,2,3];

}

由于我们的重点不是去深入分析JavaBean本身的使用和特点(虽然它非常有用且特别面向对象），我们不去深入探讨复杂的JavaBean，其本身就可以作为一门课程了。

**功能需求：为何需要将JavaBean序列化**

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，JSON 就是一种独立于编程语言的数据交换格式，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的过程，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。在完成传输后，传输的接收方如果要重新创建该对象，则需要将JSON反序列化。

而fastjson的核心功能，也就是处理JSON数据传输的这两个步骤。

我们先不去看fastjson的实现做法，从一个初学者的想法，我来回顾一下第一次接触fastjson时琢磨其实现方法的心路历程。一种naive的method：

fastjson在其json解析处理中，不是流程式的写了无数种针对不同任务的解析过程算法。如果我之前没有学过面向对象设计的来解析json，我可能也会这么做：（以下纯属胡想，没有真正实现）

1.输入一个object（或者json string vise versa）

2.剖析object的结构，将object按类型逐一切片

3.每个片段放入一个switch case中，如数字类型，如果type==“int”,

则这个片段的输出为name

4.将每个片段拼接起来，中间用“，”连接,产生一个output string

5.system.out.println（string）；

这样一个代码，可以很简单，几百小一千行就可以完成，它本身就是单纯的处理流程，维护的也只有输入或者过程中产生的与输出相关的变量，针对简单的object的json解析，可能足够了。

问题是：为什么这样的代码只会是我们学生写任务写出来的，而不会是工业级的代码，fastjson又是如何做的，高明在何处？

很快我就能从fastjson源码中找到这样的答案，先说几点重要的：

我们学生自己写的问题：

1.异常处理流程的缺失：

2.高数据泄露的风险：想想为什么360会把我自己编译的C代码当做病毒？？？

3.各种漏掉的case，debug又困难（完全不关心开闭原则啥的)

4.个人的力量总是有限的，但像这样写，叫一个外人来看如何能看懂，如何能帮你补充？

总的来说：可扩展性差，过程式思维，各种处理流程混在一块，缺少异常维护...

我们当然知道自己写的是rubbish，但是先得知道自己的不足，才能知道fastjson是如何做的：

1.模块化编程设计：

2.API的简化，向外暴露的接口简单，使用者不需要了解fastjson具体的处理的流程，也不需要掌握输入限制等。

3.子类的不断继承，方法的不断重载，从而出错时定位出错位置变得更加简单

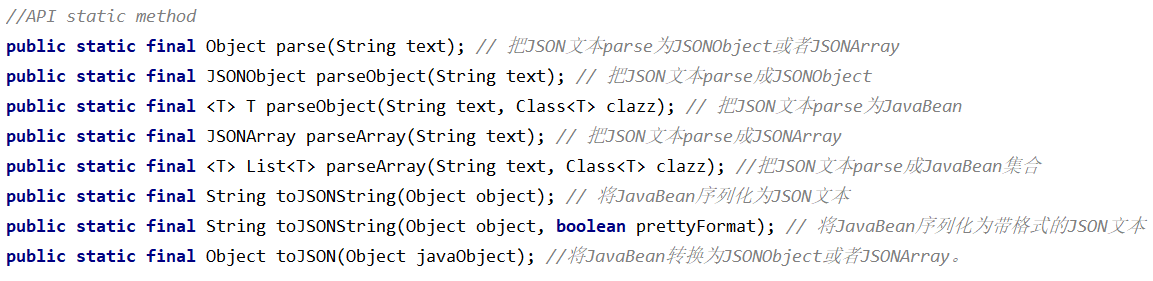
这样做的好处，我们暂时还不知道，但是这给了我们对于fastjson的一大期待，这也是让我，一个只会C和verilog的程序猿所难以理解的。下面我们开始讲fastjson了-\_-

**Fastjson API:**

然后我们再来关注fastjson的使用，在众多企业json解析库中，不乏有gson，jackson等

fastjson API 的入口类是com.alibaba.fastjson.JSON

常用的序列化操作都可以使用JSON类上的静态方法直接完成。



fastjson的一大优势就是其简洁容易上手的API，从而对于有JSON解析需求的用户，其不需要繁杂的格式调整，也不需要进行预处理，对于简单的JSON解析，其操作直接使用最常用的parseObject和toJSONString方法即可。

同样的，fastjson也有其支持复杂功能的能力，其支持泛型，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。

从JSON主类出发的模块调用：



JSON本身是个抽象类，有子类JSONObject，JSONArray继承，根据里式替换原则，这些类可以替代JSON类，其本质上也是对JSON类的扩展，主要是对对象和数组的特别功能的支持。在此次报告中，我们的关注点在主类JSON中。

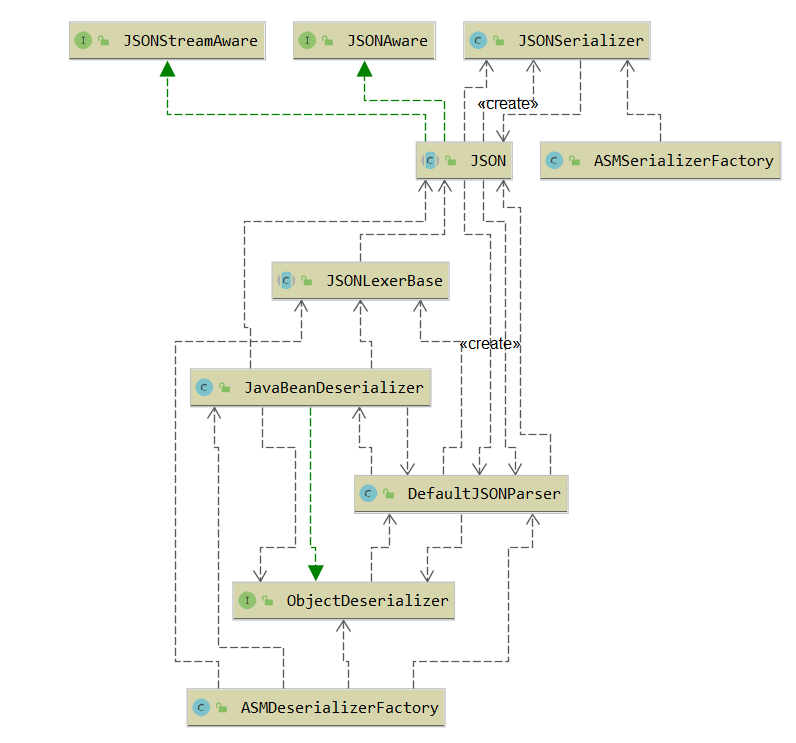
我们来看fastjson的主类JSON的类依赖图。

这里省去了很多细节的类与接口，抓住主要矛盾：

JSON主类实现的Aware接口是类型转换的接口，主要用于配置输入输出流。

序列化的主类是JSONSerializer

反序列化的主类是DefaultJSONParser。



我们试图先简单的跑一边fastjson的序列化流程，首先我们脑子里过一遍人是怎么做的:

即人是如何将Object序列化为JSON字符串的:

我们显然是逐级分析Object的类属性有哪些，我们序列化时忽略掉了其强类型，但是其值本身仍然可以反映出其强类型（如1对应int,1.对应double，“hello”对应字符串等，这在之后反序列化还原出Object时显然是会用到的），我们逐一记下其名称和值，即作为了键值对记录，然后以逗号隔开，而每个类属性可以是数，字符串，数组，字面值，或者又一个对象（进一步往里面嵌套即可），因此一个序列化的过程即是一个机械的按层深入，按属性名-属性值的键值对逐一按照JSON格式存储的过程。

如此，我们获悉了解析JSON并将对象输出到JSON的manual的方法，这会很有利于我们理解fastjson又是如何做的。而显然，这个manual的输出方法对人而言可以很简单的递归（至少对于简单的几层的示例是如此），那对于机器，fastjson是如何处理的呢？

我们后面会知道，刚才的各种想法，在fastjson中，只是一个解码译码器做的事情，但是在codec做这样的事情之前，fastjson需要先做很多事情，才能把对于人来说很好解开看的各种括号拆开，把各种类型给分开处理。

跳过剧透，我们从其中一个API：toJSONString()开始，我们分析其类与函数的调用：



在多次重载之后，我们在JSON类中找到真正开始处理序列化过程的toJSONString方法，

之前的都是在补充配置信息，因为配置并不是必要的步骤所以这里将其多次重载。



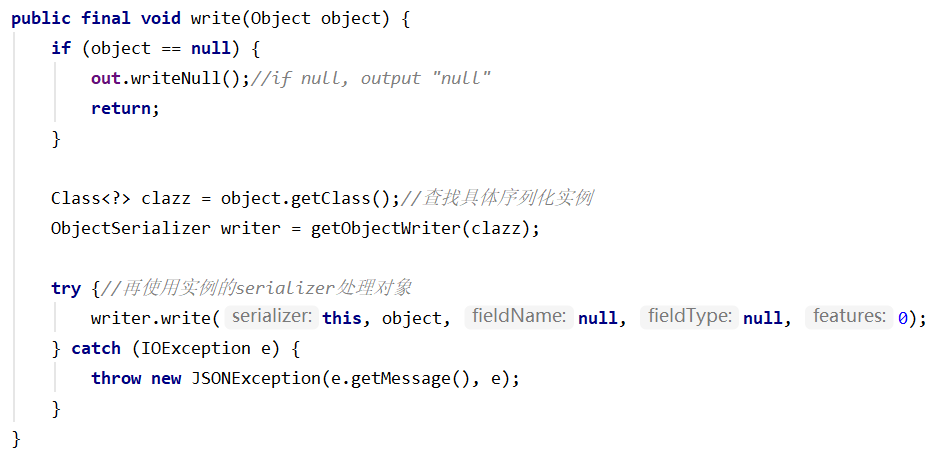
toJSONString创建了JSONSerializer对象，将之前存储的config等信息传入，并初始化好了filter

此后在SerializeWriter中，我们可以将其视为序列化的输出容器

之后的序列化使用ObjectSerializer接口，然后在JSONSerializer实例化：

根据object的类，为此类的object实例化一个writer，然后调用此writer的write方法，就完成了序列化的过程

具体的针对不同的object，writer是如何创建的，我们在核心流程分析中进一步解读：



**小结一下第一部分：初探fastjson**

刚才我们是从简单示例的用例来进行的，以最简单的case为例：倘若输入的object的句柄没连接上（即输入为null，对于JAVA萌新的我，这也是第一次尝试调试走的路径),刚才的流程会在object==null判断后输出null然后返回，甚至连实例化具体class的serializer都省去了

更一般的object（比null的复杂点）输入在序列化时，在之前的getObjectWriter方法时会首先从内部已经注册查找特定class的序列化实例（从缓存中加载（使用当前线程类加载器，JSON加载器等）），对于简单的object，这样的已实现的静态方法可以直接被调用，从而其可以省去诸多的序列化配置操作，进而大大增快JSON解析时间，而这些静态方法使用字节码优化等方式已经被深度优化，After all： fastjson needs fast.

对于萌新如我，自己设计的再复杂的object，都没能让fastjson为我新构造一个没有的writer实例（我太菜了）

阅读官方中文文档，可以得知fastjson针对很多的常用对象，都已经注册实现了其序列化方案（反序列化--与之类似）



在做完后面的部分后回看，我在这里要补充说明：已注册方案不代表fastjson一开始就把这些方法都实例化了。如果这么做，fastjson的大小可就大了去了，内存吃不起，速度也快不起来，fastjson不可能在创建时对一个输入同时实例化所有的解决方案。

讲个笑话：为了保证体育课有鞋穿，我有必要带着鞋柜去上体育课吗？（穿一双适合的运动鞋出去就vans了^\_^）

当然，在使用过一次后，其被加载到缓存了，再使用时就不需要再重新new一个了。这里的这种处理也是享元模式的一种体现

**第二部分：核心流程设计分析**

与之前一样，一如既往地复述我的**naive method：**

紧接着JSON主类的类图，我们可以看见，从JSON主类出发，序列化与反序列化分别走的是两条线，从功能上来说，二者也是相反方向的一对功能，因此其功能实现上显然是较为独立的，不恰当地说，序列化面向的是调制的功能需求，而反序列化面向的是解调的功能需求。

在之前的序列化示例中，其实已经将大部分的序列化流程讲完了，但是只是针对一个toJSONString方法而言所讲的，序列化，还得支持多种多样的需求，如常用的JSON格式如JSONObject,JSONArray，还得有配套的toJSONObject,toJSONArray等API，而这些方法与toJSONString相比会怎么开发设计？

从一个惯性思维的C语言使用者（上面向对象课之前的我）而言，针对不同的JSON格式，再重写一个感觉上差不多的toJSONObject啥的方法不就vans了吗？中间我知道是复用代码，ctrl+c,ctrl+v的事情嘛。

企业级，工业级的程序开发能这么干么？fastjson会这么干么？显然不会。如果某个测试用户反馈告诉我们某一个方法出bug了，那我可能不仅要改这一个方法，我得把每个方法都扫一遍，确保之前的错误方法没有被我复制粘贴到别处。（从面向对象的角度看：这是耦合太强了，违背了单一职责和开放封闭的原则）

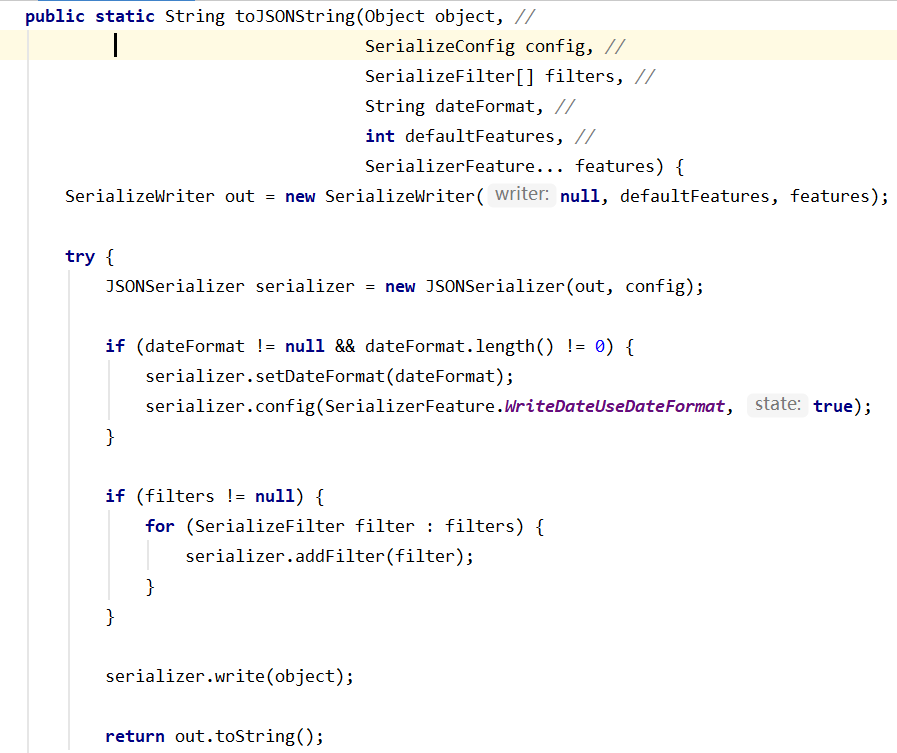
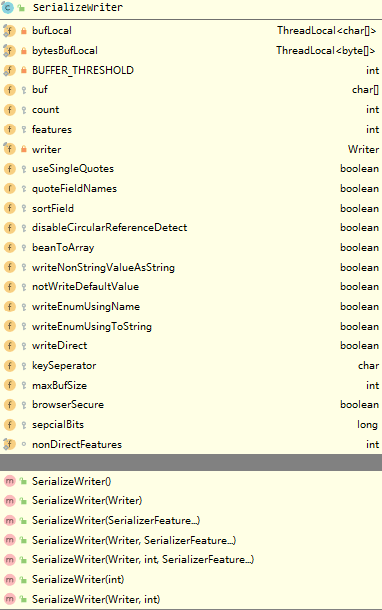
进一步深入了解fastjson的设计流程，我们紧接着看看fastjson的做法，这里我们不再去深究具体代码级的实现（一个解码译码器的内容是我们需要关心其正确性的吗？我认为不是，再说我也看不懂-\_-）

**序列化：**

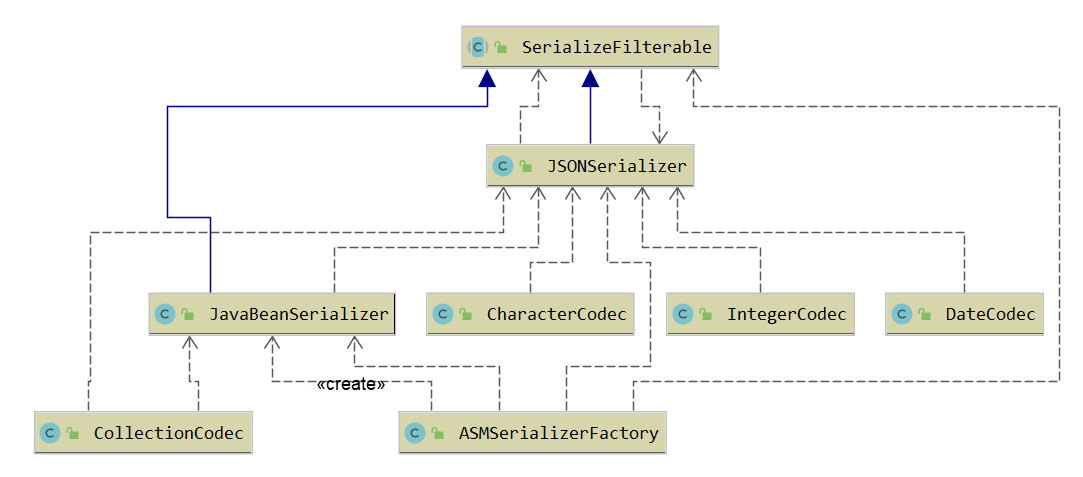
再回顾一下：Serialize过程从开始的简单API开始层层重构这个toJSONString方法，在方法中不断新添加config,filters,dataformat,features等信息和其默认构造。

我认为其核心过程是在其实例化SerializeWriter后开始真正的序列化操作过程，它首先创造一个 SerializeWriter 对象 out，SerializeWriter 类继承自 java.io.Writer 类，内部通过组合一个 Writer 对象来实现。这使得 JSON 对 OutputStream 和 Writer 的支持成为可能，尽管这并不是 SerializeWriter 设计的主要目的。

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)。



跳过容器创建的这一行，我们到try语句里面去关心下这个JSONSerializer类，首先看看它的类图：



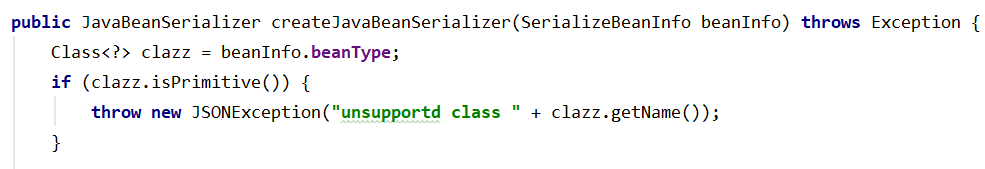
说到刚刚所提及的实例化writer，针对不同的object case，JSONSerializer调了不同的序列化编码解码器（整数对象用IntegerCodec，日期对象用DateCodec，集合对象用CollectionCodec等等 ）

而针对没有已有实例的类，一般来说复杂类型的对象（多层嵌套引用类型，如对象里是对象里是对象里是对象。。。或者是用了泛型等等），可以通过递归地调用JSONSerializer 来进行序列化。

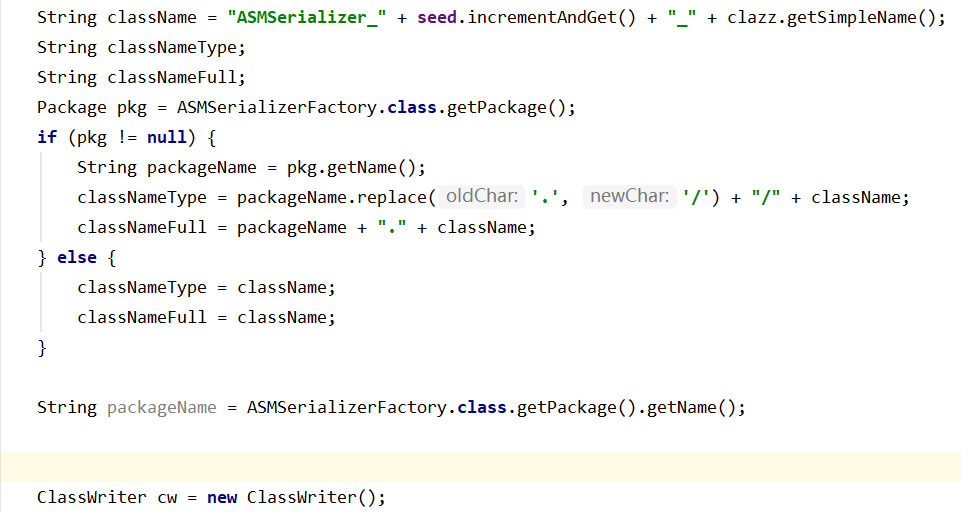
这个时候体现的就是程序设计中常用的“逐步解析”的想法。比如处理一个对象集合时，除需要处理集合本身之外，还需要处理集合中的每一个对象，这时又是一个新的解析过程。由于使用了同一个jsonSerializer，所以在进行数据处理时，输出的数据会按照在解析过程中的顺序，顺序地写入到 outWriter 中，这样即保证了数据的正确性，又实现了关注点的分离（divide and conquer）。

当实在没有办法时，我们还有一个ASMSerializerFactory类，同时从其名称上我们就可以知道其使用的是**工厂模式**（事实上与之相似，fastjson反序列化时也是这么处理无实例的情况的，那就是调工厂根据需求生产一个出来）

ASMSerializerFactory类本身比较复杂（正如ASM明示的，它的方法大量使用字节码进行优化，which I know nothing about）我们暂时忽略某些技术细节（比如asm其实不一定被启动，fastjson还有别的处理方式），只看fastjson在此是如何创建新的serializer的



ASMSerializerFactory中，通过createJavaBeanSerializer方法，根据当前对象的beaninfo参数（或者在其他情况的调用里，直接传入的就是对象的class），首先创建一个新的class，在这个class中有对目标class的序列化方法，然后基于这个class实例化一个instance，作为序列化的处理类对象。

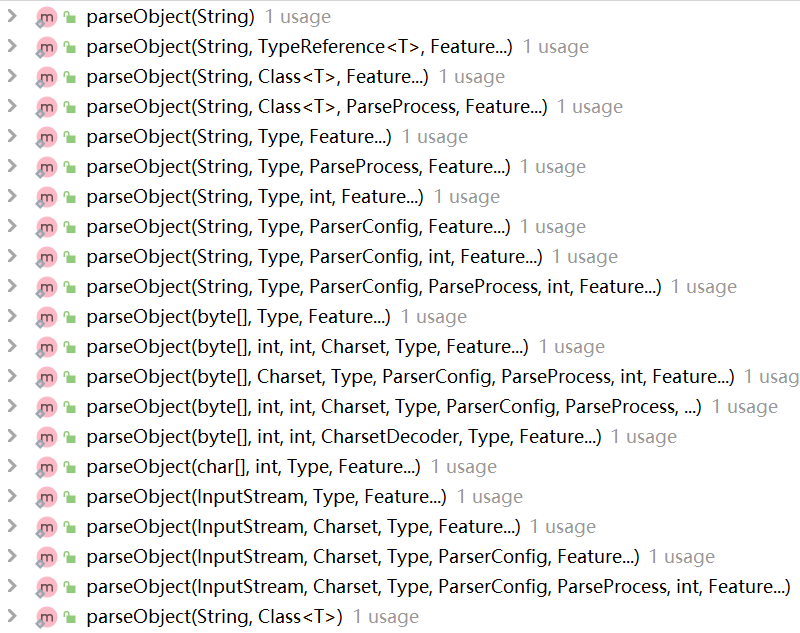


而具体的instance是什么呢，我们都到这一步了，肯定是没有默认注册方法的instance给我们的，因此我们要新实例化一个instance并为其命名，保存，就像搭建了一条全新的工厂流水线，然后只需要将这个需求的产品：针对这个class的serializer生产出来就可以了。而且为了保证其之后如果再用到，我们就不必再重新建立一次该流水线了。

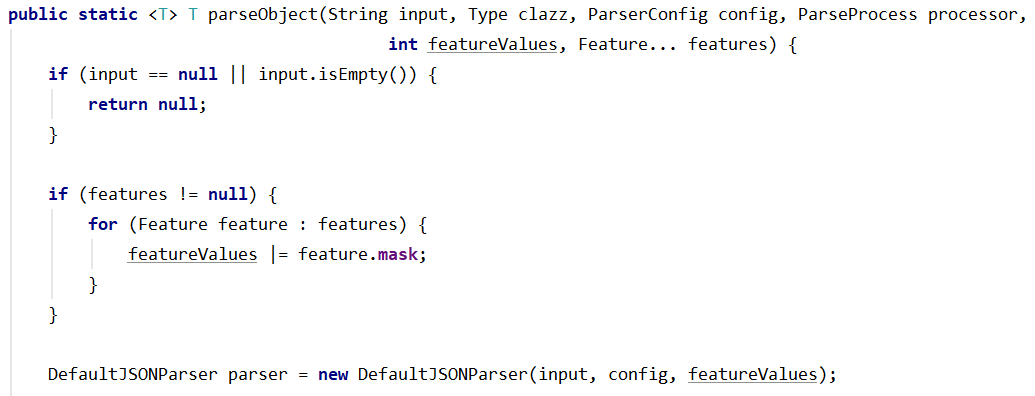
**反序列化：**

之前的各种例子以各种序列化为主，现在我们来看一下反序列化的核心流程。

反序列化的一大入口API是JSON类中的parseObject，根据需求的不同的转换类型，有数十种不同的parseObject方法，但它们都可由简单API根据输入类型与配置参数自动重载进入。



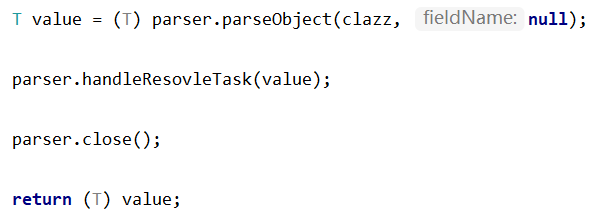
我们跟踪我们的函数调用，发现parseObject实例化了一个DefaultJSONParser



与序列化类似，在实例化parser过程中，查找具体序列化实例同样是委托给config查找的：

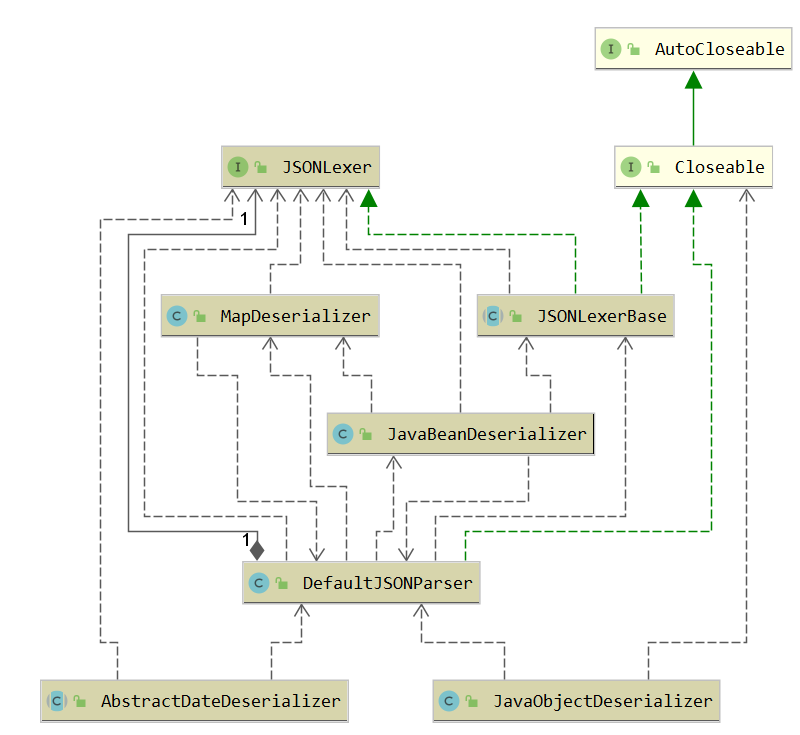


而转换对象的过程，简单地讲是parser创建之后直接调用该实例中注册的parse方法进行转换：



但这里其实等于什么具体的过程都没有讲。

所以我们的关注点要放在JSON反序列化过程的主类DefaultJSONParser中看。



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，如：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，而具体的反序列化在JavaBeanDeserializer等类中处理（根据要parse到的class而定，这也是当不开启ASM方法时fastjson创建反序列化器的方法，序列化也是类似）

其整体的框架如下：

1.创建解析配置ParserConfig对象，包括初始化内部反序列化实例和特性配置等。

2.添加反序列化拦截器

3.根据具体类型查找反序列化实例，执行反序列化转换

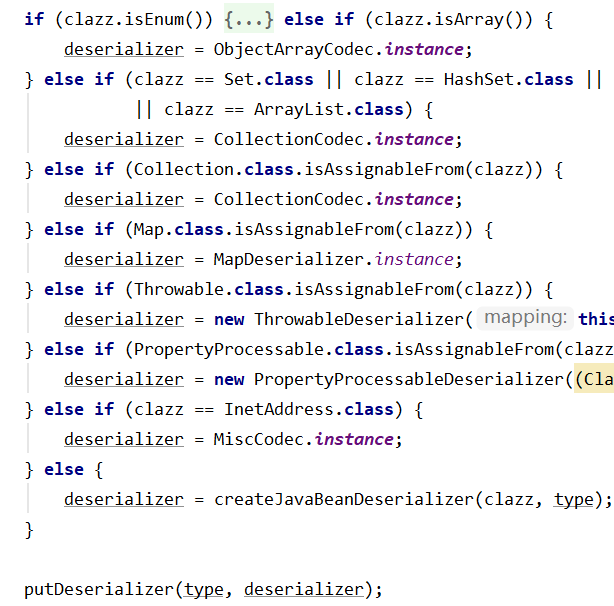
4.解析对象内部引用

我们可以从DefaultJSONParser出发的类图看出这样的关系：

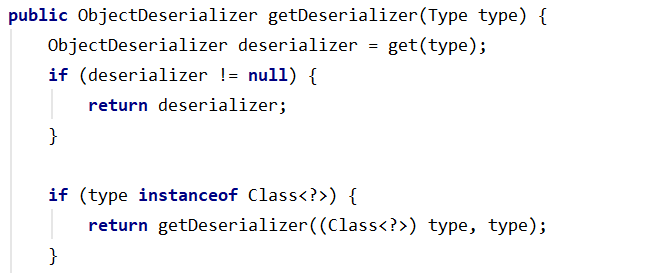
根据不同的反序列化需求，DefaultJSONParser会调不同的Deserializer，如JavaObjectDeserializer，AbstractDateDeserializer等等。而几乎所有从API进入的功能都是被集成在该类中的（即它是对外包装好的盒子）

从设计模式上讲，这里采用的是一种**外观模式**，将所有的功能包装在了DefaultJSONParser中。

这里与序列化时不同点在于，序列化时根据的主要信息来自对象本身，而反序列化时依赖的更多是配置信息，这里也与调制解调的难点相似-\_-

因此反序列化的查找反序列化实例的方式也与序列化时根据对象本身类的查找不太相同。但其核心逻辑是在委托配置查找反序列化实例，与序列化时的过程又具有相似之处。我们分析其中的过程：发现真的很像，反序列化器的创建，也是先从内部已经注册查找特定type的反序列化实例，没有则进一步从内部已经注册查找特定class的反序列化实例，之后逐个判断类型注解，泛型真实类型等等。最后根据特定类型匹配接口或者继承实现类查找。

在最后的情况，也类似，返回默认的JavaBeanDeserializer。而在JavaBeanDeserializer中，如果开启了asm方法，则会与序列化相似，采用工厂方法模式生产一个字节码优化的反序列化器出来。



**小结一下：**

在fastjson的核心流程中序列化与反序列化采取了非常类似的判断策略，以此前讲的比较少的反序列化为例：

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（实则是用工厂方法创建一个新的instance）

相似的判断策略，显然对于程序开发的进度有了很大的帮助，但是反过来看，序列化与反序列化又是两个流程不太一样的过程，二者虽然互为逆过程，但是处理起来的步骤很不一样，只有在Codec时会调同一个Codec去做解码/编码工作，也就是去做逆过程本身的事情，而事实上，我之前设想的C程序也就只做了这么一点点事情，其他的各种功能，完全都没有去想过，也就是说，我写出来的rubbish，充其量也就是一种还不那么高效的解码编码器部件罢了。真正的软件工程开发，还离我非常遥远啊。

而二者的不同，很大程度上在于序列化更多的是直接根据对象本身的特性来做，而反序列化更多是根据用户给出的要求来做，毕竟序列化是个将更多信息的对象简化传输的过程，而反序列化是相反的，将少信息的传输JSON通过外加补充信息从而复原的过程。因此二者又实则是不同的过程。而其难度，显然是反序列化的难度要大很多，但对于fastjson而言，其难点被包装在了DefaultJSONParser类里面，从而减少了对客户处理的对象数目，其使用难度就被简化了，这里颇有些外观模式的体现。

至此我们已经基本了解了fastjson的组织框架和核心流程序列化和反序列化。

**第三部分：高级设计意图分析**

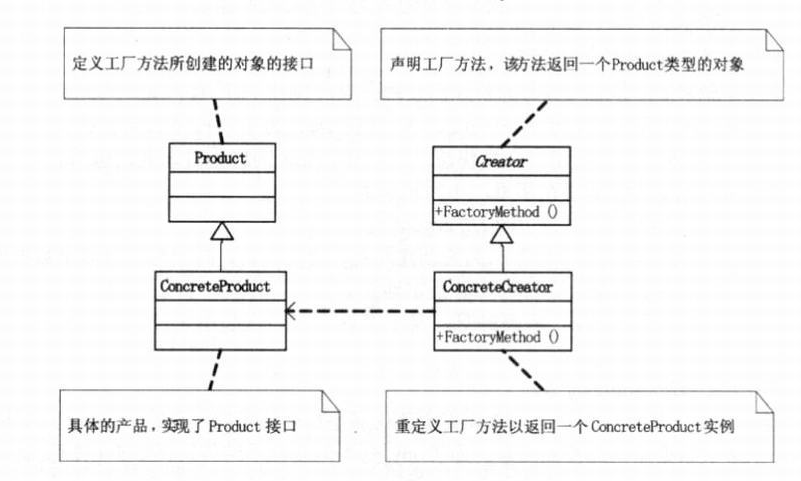
在fastjson的核心流程的解读上，我们已经发现了不少的高级设计模式在序列化与反序列化的过程中的应用，下面来详细地讲讲我在学习这几种设计模式中的收获。这里对各种设计模式的解释图来自程杰著，清华大学出版社出版的《大话设计模式》一书。

**工厂模式：**

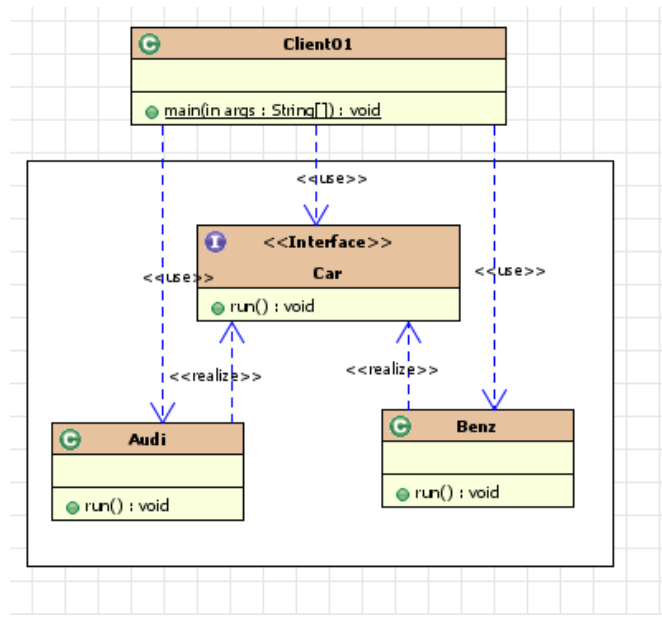
借助GOF中对23种设计模式的定义，我们关注工厂模式的定义：

“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。



一个经典的例子是老师课上所讲的生产 汽车 的工厂，如果得到的需求是生产奥迪，就生成一个 奥迪车对象返回，如果是奔驰，就生成一个 奔驰车对象返回。



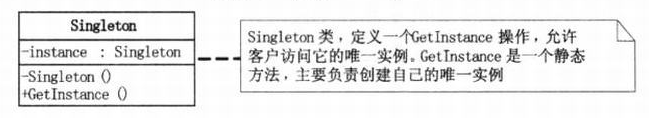
我们回到fastjson的源码中来，我们应该对于之前核心流程解析中的ASMFactory等类有印象，其使用是在序列化与反序列化器的创建过程中：当在各种已注册实例isntance均fail的情况下，我们根据类名和 propertyNamingStrategy 生成 beanInfo，之后利用 asm 工厂类的 createJavaBeanDeserializer 生成处理类。

工厂方法模式的本质是生成基于给定信息的实体对象，在这里只是它所得到的给定信息不只是一个代表汽车需求的字符串，还有更多目标 class 的特征，而其生产的细节，作为创建模式的工厂模式，将其延迟到子类再确定，我认为这样可以达到”因材施教”的效果。举个例子，针对字节码的效率优化，也只有在确定类的属性之后才能进一步做下去，毕竟对于一个具体的生产任务，才能够有更好的更专用的优化方案，这里也是我认为在fastjson中使用工厂模式来生产序列化/反序列化器的一大原因。

Plus:想想程序员熟悉的反例： gcc vs icc, linux vs windows... 通用往往意味着低效，专用往往意味着专业.当然这里就不黑开源了，毕竟fastjson也是开源代码^\_^

**单例模式：**

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。



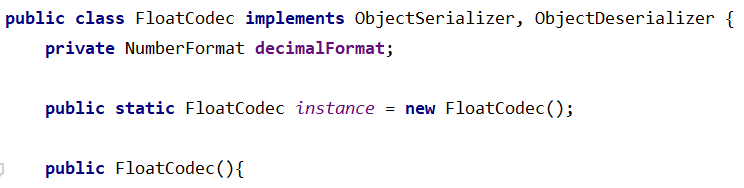
而在单例模式的实现的核心上，是一个比较tricky的语法模式（这显然是java等面向对象语言设计时良定义的），即在我们实例化一个类的对象时，通过的构造方法是先判断是否有显式定义的构造方法，如果没有则使用默认的空的构造方法。而只要显式定义的构造方法存在，默认构造就不能使用了，那么如果我们将这个构造方法设置为private的，则此类的外部类均无法实例化构造这个对象了。此时只需要在类中创建一个public的GetInstance()方法，用于在判断其未实例化时可以调这个private的构造方法来创建唯一的实例，这也就限制死了类的对象的实例化只能在本类中进行，而这就是单例模式所想要实现的。

此前在 FASTJSON 中的序列化与反序列化流程中，有一个 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。



而创建的 SerializeConfig 类的对象 globalInstance，这其实某种程度上说就是 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。我们以FloatCodec类为例：



我们可以很好的看到在Codec解码过程中根据类型只实例化单个解码器，这是明显但不严格的Singleton模式。

只因为：这些instance是public的

严格地说，fastjson所用到的这种单例模式不算是严格的单例模式，我们参照严格的单例模式的关键：

1私有的构造方法。

2仅能有一个实例，且必须由自己创建自己的实例

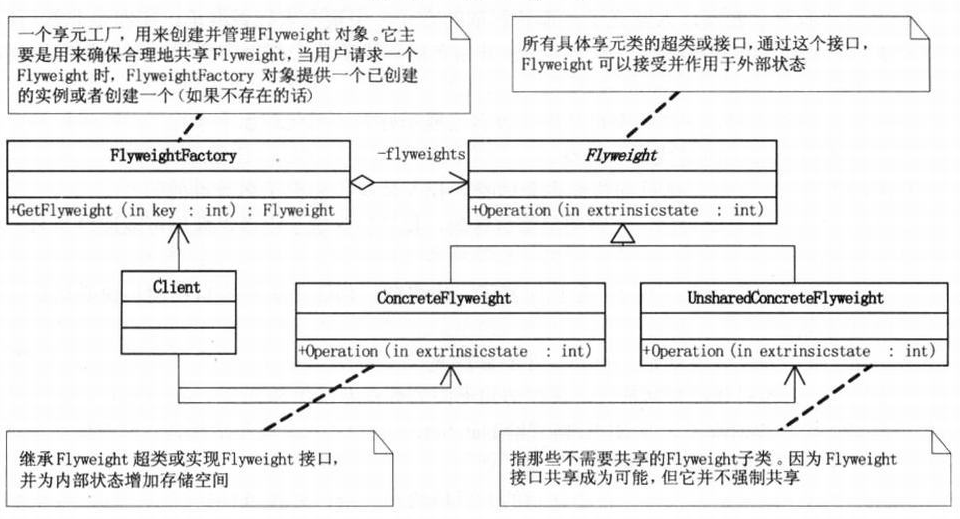
而fastjson在序列化与反序列化中，采用的构造器是公开的，且并没有限制只能有一个实例。

参照雷正宇学长的gitbook，我这里赞同学长的观点，为什么 Fastjson 没有严格地去实现单例，不妨认为这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 Fastjson为了进一步提速要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进行比较等。最后的最后，我们可以考虑到一个解码器本身不需要多个实例。将其构造器公开，也可能是因为fastjson开源代码的调试和用户拓展的方便。

**享元模式：**

享元模式的核心在于享元共用，即共享，共享再共享，目的是减少大量重复的对象的创建，避免带来的高内存占用等资源浪费。

通常使用享元模式，会配套定义一套享元工厂（工厂模式），这个工厂做两件事情，1.在里面维护一个享元池（这个享元池就是保存标准化对象的地方，其实是一张HashMap/HashTable，同时也是一个实例）2.暴露一个方法getFlyweight(key)：在享元池中遍历key，返回匹配的对象。若不存在则创建新对象(key)，保存到享元池，然后返回该对象。（这些标准化对象就是所谓的享元对象）



我们之前在详细将序列化与反序列化过程中，也已经提到了实例化序列化器和反序列化器的过程

JSONSerializer逐一查找注册的享元池中是否有实例，如果没有则由ASM工厂生产一个。

一般来说对于这一系统，也只会有一个ASM工厂，所以可以说，享元模式就是工厂+单例的组合

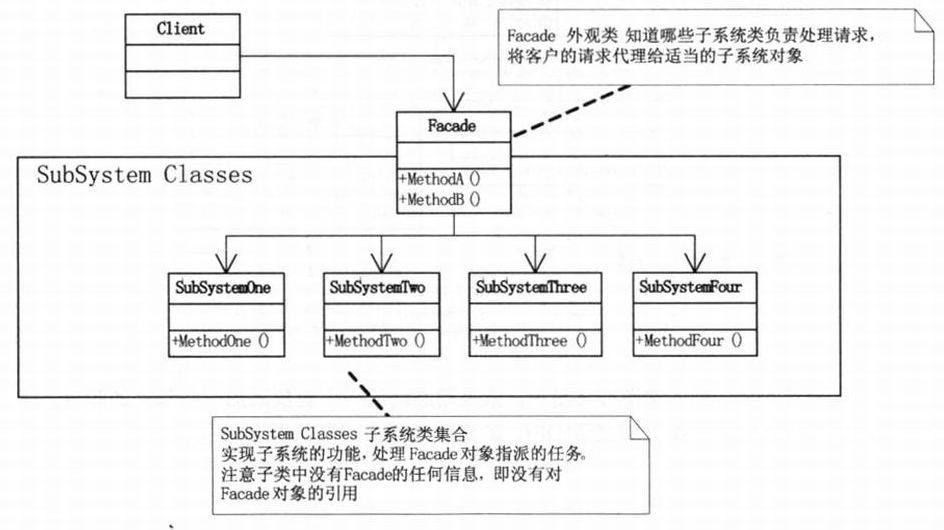
使用享元模式是必要的：fastjson系统中有大量对象，这些对象会消耗大量内存，这些对象的状态大部分可以外部化。使用享元模式，正如此前序列化时所讲到的，实例化每种注册方案都有开销，这些方案通过外部化方法放在缓存中，这样实际运行时只需通过HashMap根据Instance标识码调过来实例化即可。

优点：大大减少了对象的创建，降低了程序内存的占用，提高效率

缺点：增加系统复杂性，增加源码阅读困难度-\_-

**外观模式：**

外观模式的核心是：为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高级接口，这个接口使得这一子系统更加容易使用。

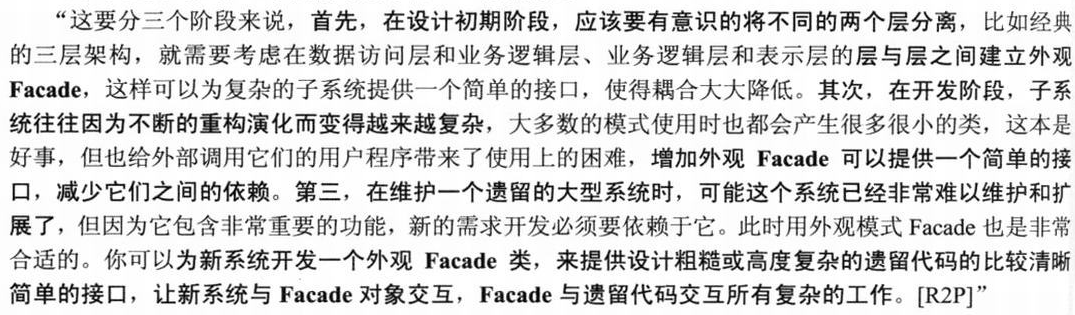


在此前的DefaultJSONParser类中，我们观察到了这样的设计模式，因为对于反序列化过程，

parse()、parseObject()、 parseArray()等将 json 串转换为 java 对象的静态方法都实际托付给了 DefaultJSONParser 类（核心流程中只以parse为例，我们发现在JSON类中它其实啥实事都没做（主要是配置信息和需求））。

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，用一个外观盒子将内部子系统打包，只留下外观盒子的接口供外部统一调用。

由于外观模式较难，我也没有特别深入的理解，这里只能抄一抄介绍外观模式用处的书了：



---摘自《大话设计模式》

Fastjson采用的外观模式，参考了外观模式的常用情形，在这里我认为可能是反序列化本身的难度较大，开发的子系统复杂度太高，小类太多，对于JSON主类不好外部调用，因而采用外观模式的方式预包装好了反序列化的子系统在DefaultJSONParser类中。

由于对于此模式的认识不够深入，因此不再言多必失。

**总结：**

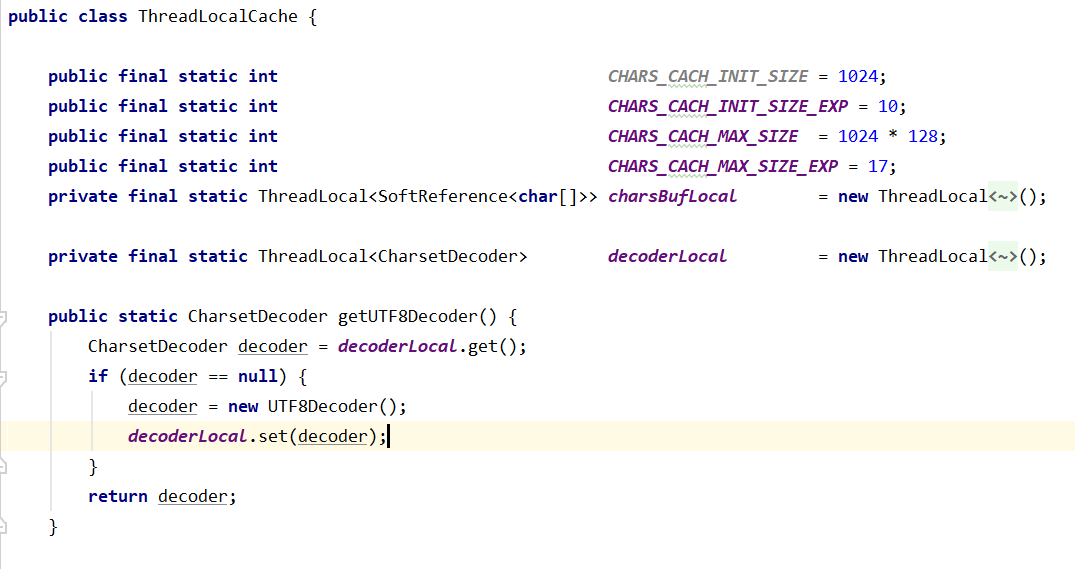
Fastjson的阅读，在这个学期的课程上完之后，我大概将除了字节码的部分都看了一遍，主类和核心流程等不涉及具体编码操作的但是重要的代码大概看了3遍。不能说看懂了fastjson，但是也确实收获了不少东西。

个人对fastjson源码的理解，确实有一个非常明显的点：那就是fastjson真的是为了fast穷尽手段，如之前提及的各种asm方法的效率优化，用HashMap快速查找缓存实例等

我还进一步去查了下和速度相关的一些可能的技术，首先是多线程：

fastjson有良好的多线程支持（又是一个让人看不懂的地方），因此只在网上颇为丰富的fastjson用法中找到了一个多线程优化的例子：

在ThreadLocalCache中缓存了buf，这个办法能够减少对象分配和gc，从而提升性能。SerializeWriter中包含了一个char[] buf，每序列化一次，都要做一次分配，使用ThreadLocal优化，能够提升性能。



作为不会多线程的OS没学好的垃圾，这里并不能说清楚fastjson的多线程支持的源码

但是fastjson确实支持多线程，事实上，一个企业级的开发不能够支持多线程是可笑的。

Fastjson花了大投入在了JSON解析的速度上，那么其达到了想要的效果么？

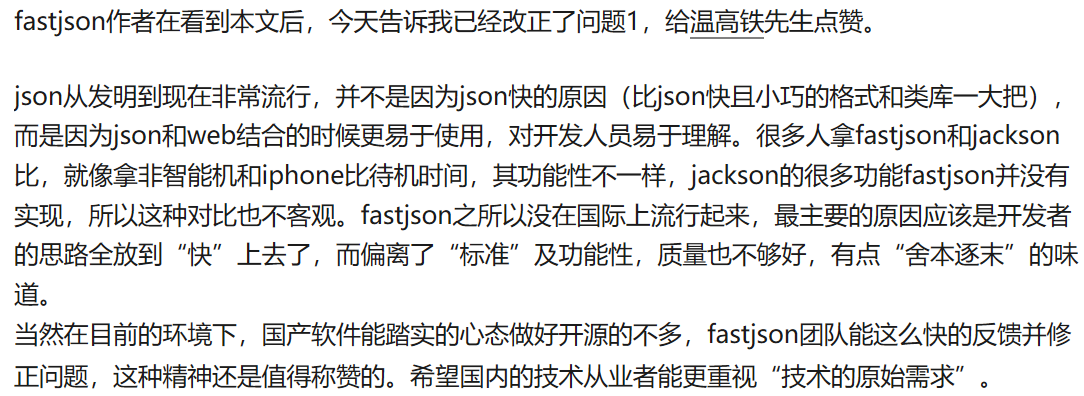
我们从网上找到的关于各种json解析库的功能测评test，确实看得出来，fastjson的性能测试上确实具有显著的优势，无论是序列化时间还是反序列化时间都具有优势，而且大小上也不算特别大（这得感谢设计模式）。



除去benchmark的评测，我在fastjson相关的网站中还发现了不少对fastjson的批评的或者是贬低的博客，而这些不少又是靠贬低fastjson来吹捧jackson和gson的。毕竟与不同公司开发的相同功能产品之间总缺不了口水战，就像可口可乐和百事可乐的广告战的迭起兴衰一般。

作为一个没有大工程开发经验的人，自然没有资格去评价fastjson的是非对错，但是可以看看行业牛人对fastjson的评价，这里是在一个内存泄露的问题上的评价：

详细链接：https://www.zhihu.com/question/44199956?sort=created



fastjson固然不完美，但阿里的程序员们能够快速反馈修正问题，快速迭代版本，并且一直坚持开源，确实也算是业界的一大良心。

以上就是对于fastjson源码阅读的个人体会与收获吧。

-----------------------------------------分割线----------------------------------------------

下面还得感性地谈谈面向对象程序设计这门课对我的影响：

本学期的课程学习，不得不说有一个遗憾：王伟老师的课真的非常有趣生动，我个人对于java的兴趣似乎也比c要高很多（这主要是因为写操作系统写自闭了），但是本学期的课业压力确实很重，导致源码阅读的时间零零散散，对于设计模式的理解，更多地，还只能是学到一个概念，对于fastjson本身的设计模式意图，还不能非常深入地理解。

但面向对象的思想，确实在本学期王老师讲课的过程中，潜移默化地影响了我的编程风格。

一如我在此前部分提到的，我的想法从崇尚简单直接的naive的个人单干，逐渐地被fastjson这样OOP的代码所感染。不得不说，C语言这样的过程式的语言的使用，确实对于思想上面向对象的思想有所抵触，曾经我初步接触java对一来就得定义package，class，要找好半天才能找到一个main颇为头疼。但是逐渐地我理解到这才是真正程序开发时所必需的，毕竟我不能用一个main写完所有要实现的事情（似乎C语言老师讲的其实是可以的？？？就是从头到尾完全不复用代码那种，我的五子棋大概就是那么写的）

阅读这fastjson的开源代码，确实让我对面向对象程序设计有了新的认识，新的见解。这里尤其是以我接触的另一门语言verilog感触颇深，此前我总觉得这样的硬件设计语言与面向对象语言不会有什么联系，但我其实发现它们其实有很多的共通点，尤其是verilog的模块的布线连接，模块间信号的输入输出，其实就是面向对象设计中的模块间继承，接口的实现等等。虽然语法上截然不同，但是思想上确实，也是面向对象的思想，因此鉴于我薄弱的java能力，也许我用的最好的OOP语言是verilog？？？

胡扯就到这里，最后再次感谢王伟老师和唐震老师本学期以来的精彩课程！

张朔铭

2020.1.9

**参考文献：**

【1】fastjson中文手册：

http://www.dba.cn/book/fastjson/

【2】CSDN 博客-为什么Fastjson能够做到这么快? ：https://blog.csdn.net/u012961566/article/details/76944982

【3】CSDN 博客-23 种高级设计模式：

https://www.cnblogs.com/swordfall/p/10742412.html#auto\_id\_16

【4】雷正宇学长的gitbook：

https://842376130.gitbook.io/fastjson-learning-report/di-san-zhang-xiao-cha-qu

【5】百度百科：

https://baike.baidu.com/item/JSON/2462549?fr=aladdin

【6】CSDN 博客 -关于享元模式https://blog.csdn.net/weixin\_36094484/article/details/80517145

【7】知乎 -fastjson这么快老外为啥还是热衷 jackson?

https://www.zhihu.com/question/44199956?sort=created