一、序列化是什么

1. 序列化就是将一个对象的状态（各个属性量）保存起来，然后在适当的时候再获得。
2. 序列化分为两大部分：序列化和反序列化。序列化是这个过程的第一部分，将数据分解成字节流，以便存储在文件中或在网络上传输。反序列化就是打开字节流并重构对象。

二、Java序列化机制

1.   Java 序列化简介

Java 对象序列化是 JDK 1.1 中引入的一组开创性特性之一，用于作为一种将 Java 对象的状态转换为字节数组，以便存储或传输的机制，以后，仍可以将字节数组转换回 Java 对象原有的状态。

实际上，序列化的思想是 “冻结” 对象状态，传输对象状态（写到磁盘、通过网络传输等等），然后 “解冻” 状态，重新获得可用的 Java 对象。

java.io.ObjectOutputStream代表对象输出流，它的writeObject(Object obj)方法可对参数指定的obj对象进行序列化，把得到的字节序列写到一个目标输出流中。

java.io.ObjectInputStream代表对象输入流，它的readObject()方法从一个源输入流中读取字节序列，再把它们反序列化为一个对象，并将其返回。

只有实现了Serializable和Externalizable接口的类的对象才能被序列化。Externalizable接口继承自Serializable接口，实现Externalizable接口的类完全由自身来控制序列化的行为，而仅实现Serializable接口的类可以采用默认的序列化方式 。

2.       对象序列化步骤：

1） 创建一个对象输出流，它可以包装一个其他类型的目标输出流，如文件输出流；

2） 通过对象输出流的writeObject()方法写对象。

3.       对象反序列化的步骤：

1） 创建一个对象输入流，它可以包装一个其他类型的源输入流，如文件输入流；

2） 通过对象输入流的readObject()方法读取对象。

4.       示例

**People.java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package org.leo.serialize.domain;  import java.io.Serializable;  public class People implements Serializable{      private static final long serialVersionUID = 6060652797409346110L;        private String color;        public String getColor() {          return color;      }        public void setColor(String color) {          this.color = color;      }  } |

**Man.java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | package org.leo.serialize.domain;  import java.io.Serializable;  public class Man extends People implements Serializable{      private static final long serialVersionUID = 4634741880940129060L;      public static String SEX = "M";        public transient String stran = "ggg";        private String name;        private int age;        ////private String id;        public Man(){      }        public Man(String name, Integer age){          this.name = name;          this.age = age;      }      public String getName() {          return name;      }      public void setName(String name) {          this.name = name;      }      public int getAge() {          return age;      }      public void setAge(int age) {          this.age = age;      }  } |

**JSerializerTest.java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | package org.leo.serialize.java;  import java.io.FileInputStream;  import java.io.FileOutputStream;  import java.io.ObjectInputStream;  import java.io.ObjectOutputStream;  import org.junit.Test;  import org.leo.serialize.domain.Man;  import org.leo.serialize.domain.People;  public class JSerializerTest {        @Test      public void test(){          System.out.println(String.class.getName() + "-" + String.class.getName().length());          System.out.println(Man.class.getName() + "-" + Man.class.getName().length());          System.out.println(People.class.getName() + "-" + People.class.getName().length());      }        @Test      public void serialize() throws Exception {          Man man = new Man("leo", 18);          man.setColor("yellow");          //student.setId(10000);          FileOutputStream os = new FileOutputStream("D:\\Temp\\man.out");          ObjectOutputStream out = new ObjectOutputStream(os);          out.writeObject(man);          os.flush();          os.close();          //os.toByteArray();      }        @Test      public void deserialize() throws Exception {          FileInputStream is = new FileInputStream("D:\\Temp\\man.out");          ObjectInputStream in = new ObjectInputStream(is);          Man man = (Man) in.readObject();          in.close();          //System.out.println("id=" + people.getId());          System.out.println("name=" + man.getName() + ";age=" + man.getAge());          System.out.println("SEX=" + Man.SEX + ";stran=" + man.stran);          System.out.println("Color=" + man.getColor());      }  } |

**5.         Java的序列化算法**

序列化算法一般会按步骤做如下事情：

a)        将对象实例相关的类元数据输出。

b)        递归地输出类的超类描述直到不再有超类。

c)        类元数据完了以后，开始从最顶层的超类开始输出对象实例的实际数据值。

d)        从上至下递归输出实例的数据

6.       示例演示

上面示例，对象man序列后被保存在文件中，我们用十六进制的方式打开文件，内容如下：

AC ED 00 05 73 72 00 1C   6F 72 67 2E 6C 65 6F 2E

73  65 72 69 61 6C 69 7A    65 2E 64 6F 6D 61 69 6E

2E 4D 61 6E 40 51 E9 2F    4F 0F EB 24 02 00 02 49

00 03 61 67 65 4C 00 04      6E 61 6D 65 74 00 12 4C

6A 61 76 61 2F 6C 61 6E     67 2F 53 74 72 69 6E 67

3B 78 72 00 1F 6F 72 67      2E 6C 65 6F 2E 73 65 72

69 61 6C 69 7A 65 2E 64      6F 6D 61 69 6E 2E 50 65

6F 70 6C 65 54 1B C3 9B     C8 8F C2 3E 02 00 01 4C

00 05 63 6F 6C 6F 72   71     00 7E 00 01 78 70 74 00

06 79 65 6C 6C 6F 77   00     00 00 12 74 00 03 6C 65

6F

我们来仔细看看这些字节都代表了啥：

1)        AC ED: STREAM\_MAGIC. 声明使用了序列化协议.

2)        00 05: STREAM\_VERSION. 序列化协议版本.

3)        0x73: TC\_OBJECT. 声明这是一个新的对象.

4)        序列化算法的第一步就是输出对象相关类的描述。例子所示对象为Man类实例，因此接下来输出Man类的描述。见颜色：

5)        0x72: TC\_CLASSDESC. 声明这里开始一个新Class。

6)        00 1C: Class名字的长度，这里28

7)        6F 72 67 2E 6C 65 6F 2E 73 65 72 69 61 6C 69 7A  65 2E 64 6F 6D 61 69 6E 2E 4D 61 6E: Man类名: org.leo.serialize.domain.Man

8)        40 51 E9 2F  4F 0F EB 24: SerialVersionUID, 序列化ID，如果没有指定，则会由算法随机生成一个8byte的ID.

9)        0x02: 标记号. 该值声明该对象支持序列化。

10)    00 02: 该类所包含的域个数。

接下来，算法输出其中的一个域，int age；见颜色：

1)        0x49: 域类型. 49 代表"I", 也就是Int.

2)        00 03: 域名字的长度.

3)        61 67 65: age,域名字描述.

接下来，算法输出其中的一个域，String name；见颜色：

1)        0x4C: 域的类型.

2)        00 04: 域名字的长度.

3)        6E 61 6D 65: name,域名字描述.

4)        0x74: TC\_STRING. 代表一个new String.用String来引用对象。

5)        00 12: 该String长度.

6)        4C 6A 61 76 61 2F 6C 61 6E  67 2F 53 74 72 69 6E 67 3B: Lcontain;, JVM的标准对象签名表示法：Ljava/lang/String;

7)        0x78: TC\_ENDBLOCKDATA,对象数据块结束的标志

接下来算法就会输出超类也就是People类描述了，见颜色

1)        0x72: TC\_CLASSDESC. 声明这个是个新类

2)        00 1F：类名长度

3)        6F 72 67  2E 6C 65 6F 2E 73 65 72 69 61 6C 69 7A 65 2E 64  6F 6D 61 69 6E 2E 50 65 6F 70 6C 65 ：类名描述：org.leo.serialize.domain.People

4)        54 1B C3 9B  C8 8F C2 3E： SerialVersionUID, 序列化ID

5)        0x02: 标记号. 该值声明该对象支持序列化

6)        00 01: 类中域的个数

7)        0x4C: 域的类型

8)        00 05： 域名字长度

9)        63 6F 6C 6F 72 ：父类color，域名字描述

10)     71  00 7E 00 01： 参考已写进流对象

11)    0x78 : TC\_ENDBLOCKDATA,对象块结束的标志

12)    0x70: TC\_NULL, 说明没有其他超类的标志。

到此为止，算法已经对所有的类的描述都做了输出。下一步就是把实例对象的实际值输出了。这时候是从父类People的域开始的，见颜色

0x74: TC\_STRING. 代表一个new String.用String来引用对象

00 06:长度，这里是6个字节

79 65 6C 6C 6F 77：域color的值：yellow

下面是Man类域的值：

域age的值（见颜色）：00  00 00 12

域name的值（见颜色）：

0x74: TC\_STRING. 代表一个new String.用String来引用对象

00 03 :长度，这里是3个字节

6C 65 6F：域name的值：leo

7.       小结

1)   java对象增加字段、减少字段不会对序列化产生影响。

2)   改变字段类型，并且不兼容会导致读取问题

3)   serialVersionUID必须一致，如果对象中没有，则会自动根据ClassName、Fields等按某种算法生成，这样的话如果增加、减少字段，则缓存系统的老数据会出现问题。

4)   序列化严格按对象中的真实Fields，和get方法无关

5)   序列化是循环递归调用，相当消耗cpu，也多出很多冗余数据

三、序列化框架简单介绍

除Java序列化之外，还有很多其它的序列框架，这里我们简单介绍几个。

1         Hessian2

**1.1         简单介绍**

Hessian是caucho公司提供的开源协议，基于HTTP传输，服务端不用开防火墙端口。协议的规范公开，可以用于任意语言。Hessian也是一个轻量级的,自定义描述的二进制RPC协议。Hessian的好处是精简高效，可以跨语言使用，我们可以针对任意语言开发对其协议的实现。目前已有实现的语言有：Java、Flash、Python、C++、.NET C#、D、Erlang、PHP、Ruby。还没有Delphi的实现。

**1.2         示例代码**

**HessianSerializerTest**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | package org.leo.serialize.hessian;  import java.io.FileInputStream;  import java.io.FileOutputStream;  import org.junit.Test;  import org.leo.serialize.domain.Man;  import com.caucho.hessian.io.Hessian2Input;  import com.caucho.hessian.io.Hessian2Output;  public class HessianSerializerTest {      @Test      public void serialize() throws Exception {          Man man = new Man("leo", 18);          man.setColor("yellow");          //man.setId("10000");          FileOutputStream os = new FileOutputStream("D:\\Temp\\man-hessian2.out");          Hessian2Output out = new Hessian2Output(os);          out.writeObject(man);          out.flushBuffer();          os.close();      }          @Test      public void deserialize() throws Exception {          FileInputStream is = new FileInputStream("D:\\Temp\\man-hessian2.out");          Hessian2Input in = new Hessian2Input(is);          Man man = (Man) in.readObject();          in.close();          //System.out.println("id=" + man.getId());          System.out.println("name=" + man.getName() + ";age=" + man.getAge());          System.out.println("SEX=" + Man.SEX + ";stran=" + man.stran);          System.out.println("Color=" + man.getColor());      }  } |

**1.3         小节**

1)     有两种方式供选择，JavaSerializer、UnsafeSerializer，默认采用UnsafeSerializer，通过isEnableUnsafeSerializer设置

2)     每个类都会创建对应的Serializer，并缓存在cachedSerializerMap，key是该类的class

3)     JavaSerializer，采用的是Java反射机制获取对象的字段值

4)     UnsafeSerializer，利用Unsafe，获取对象字段的内存址，根据地址获取值。

关于Unsafe请参考：<http://blog.csdn.net/humanbeng/article/details/15813429>

2         Dubbo

**2.1         简单介绍**

Dubbo是阿里巴巴开源出来的一个分布式服务框架，致力于提供高性能和透明化的RPC远程服务调用方案，以及作为SOA服务治理的方案。

Dubbo，默认情况下，使用Hessian2作为其序列化框架，同时也实现了一套自己的序列化框架，我们姑且将该序列化框架叫做Dubbo

**2.2         示例代码**

**DubboSerializerTest.java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | package org.leo.serialize.dubbo;  import java.io.FileInputStream;  import java.io.FileOutputStream;  import org.junit.Test;  import org.leo.serialize.domain.Man;  import com.alibaba.dubbo.common.serialize.support.dubbo.Builder;  public class DubboSerializerTest {      @Test      public void serialize() throws Exception {          Builder<Man> bb = Builder.register(Man.class);            Man man = new Man("leo", 18);          man.setColor("yellow");          //man.setId("10000");          FileOutputStream os = new FileOutputStream("D:\\Temp\\man-dubbo.out");          bb.writeTo(man, os);          os.close();      }          @Test      public void deserialize() throws Exception {          Builder<Man> bb = Builder.register(Man.class);            FileInputStream is = new FileInputStream("D:\\Temp\\man-dubbo.out");          Man man = (Man) bb.parseFrom(is);;          is.close();          //System.out.println("id=" + man.getId());          System.out.println("name=" + man.getName() + ";age=" + man.getAge());          System.out.println("SEX=" + Man.SEX + ";stran=" + man.stran);          System.out.println("Color=" + man.getColor());      }  } |

**2.3         小结**

1)   static，transient类型的字段不被序列化

2)   不强制要求实现接口：Serializable

3)        对象序列前，需调用Builder<Man> bb = Builder.register(Man.class)，该方法利用Javassist将Man类生上成对应的Builder，并保存在Map中，对Man的实例序列化时，直接从该Map中查找对应的Builder，对实例进行实例化操作。

4)        使用Google group varint算法，将数据转成byte

关于varint可参考：<http://www.cnblogs.com/smark/archive/2012/05/03/2480034.html>

3         Protobuf

**1.1         简介**

ProtoBuf，全称是Protocol Buffers, 它是谷歌内部用的一种高效的、可扩展的对结构化数据进行编码的格式规范。谷歌自己内部很多程序之间的通信协议都用了ProtoBuf。

a)     结构化数据存储格式

b)     用于通信协议、数据存储等

c)     高效的序列化和反序列化

d)     语言无关、平台无关、扩展性好

e)     官方支持C++, Java, Python三种语言，但可以找到大量的几乎涵盖所有语言的第三方拓展包

f)      消息格式定义采用proto文件

官网地址：[https://developers.google.com/protocol-buffers/](https://developers.google.com/protocol-buffers/docs/overview)

**1.2         示例代码**

**Man.proto**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | package protobuf;  option java\_package = "org.leo.serialize.protobuf.domain";  option java\_outer\_classname = "ManProtos";  message People {    optional string color = 1;  }  message Man {    required string name = 1;    required int32 age = 2;    optional string color = 3;  } |

**ProtobufSerializerTest.java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | package org.leo.serialize.protobuf;  import java.io.FileInputStream;  import java.io.FileOutputStream;  import org.junit.Test;  import org.leo.serialize.protobuf.domain.ManProtos.Man;  public class ProtobufSerializerTest {      @Test      public void serialize() throws Exception {          Man.Builder man = Man.newBuilder();          //Man man = new Man("leo", 18);          man.setName("leo");          man.setAge(18);          //man.setColor("yellow");          //man.setId(10000);          FileOutputStream os = new FileOutputStream("D:\\Temp\\man-protobuf.out");          man.build().writeTo(os);          os.close();      }          @Test      public void deserialize() throws Exception {          FileInputStream is = new FileInputStream("D:\\Temp\\man-protobuf.out");          Man man = Man.parseFrom(is);          is.close();            //System.out.println("id=" + people.getId());          System.out.println("name=" + man.getName() + ";age=" + man.getAge());          //System.out.println("SEX=" + Man.SEX + ";stran=" + man.stran);          //System.out.println("Color=" + man.getColor());      }  } |

**1.3         小结**

优点

1)       二进制消息，性能好/效率高（空间和时间效率都很不错）

2)       proto文件生成目标代码，简单易用，有eclipse插件

3)       序列化反序列化直接对应程序中的数据类，不需要解析后在进行映射(XML,JSON都是这种方式)

4)       支持向前兼容（新加字段采用默认值）和向后兼容（忽略新加字段），简化升级

5)       支持多种语言（可以把proto文件看做IDL文件）

6)       Netty等一些框架集成

缺点

1)       官方只支持C++,JAVA和Python语言绑定

2)       二进制可读性差（貌似提供了Text\_Fromat功能）

3)       二进制不具有自描述特性

4)       默认不具备动态特性（可以通过动态定义生成消息类型或者动态编译支持）

只涉及序列化和反序列化技术，不涉及RPC功能（类似XML或者JSON的解析器）

4         Avro

**1.1         简介**

Avro是Hadoop中的一个子项目，也是Apache中一个独立的项目，Avro是一个基于二进制数据传输高性能的中间件。在Hadoop的其他项目中例如HBase([**Ref**](http://www.javabloger.com/article/category/hbase))和Hive([**Ref**](http://www.javabloger.com/article/category/hive))的Client端与服务端的数据传输也采用了这个工具，Avro可以做到将数据进行序列化，适用于远程或本地大批量数据交互。

使用Avro可以通过2中方式来实现：  
1.二进制编码，Avro-specific方式依赖代码(文件)生成特定类，并内嵌JSON Schema；  
2.JSON编码，Avro-generic方式通过JSON文件动态加载Schema，不需要编译加载直接就可以处理新的数据源。

avro支持跨编程语言实现（C, C++, C#，Java, Python, Ruby, PHP）

特点

1)        丰富的数据结构类型；

2)        快速可压缩的二进制数据形式，对数据二进制序列化后可以节约数据存储空间和网络传输带宽；

3)        存储持久数据的文件容器；

4)        可以实现远程过程调用RPC；

5)        简单的动态语言结合功能。

**1.2         示例代码**

**man.avsc**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | {"namespace":"org.leo.serialize.avro.domain",   "type":"record",   "name":"Man",   "fields":[       {"name":"name","type":"string"},       {"name":"age", "type":["int","null"]},       {"name":"color","type":["string","null"]}   ]  } |

**AvroSerializerTest.java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101 | package org.leo.serialize.avro;  import java.io.File;  import org.apache.avro.Schema;  import org.apache.avro.file.DataFileReader;  import org.apache.avro.file.DataFileWriter;  import org.apache.avro.generic.GenericData;  import org.apache.avro.generic.GenericDatumReader;  import org.apache.avro.generic.GenericDatumWriter;  import org.apache.avro.generic.GenericRecord;  import org.apache.avro.io.DatumReader;  import org.apache.avro.io.DatumWriter;  import org.apache.avro.specific.SpecificDatumReader;  import org.apache.avro.specific.SpecificDatumWriter;  import org.junit.Test;  import org.leo.serialize.avro.domain.Man;  public class AvroSerializerTest {        @Test      public void genericSerialize() throws Exception {          String avscFilePath = SpecificTest.class.getClassLoader().getResource("avro/man.avsc").getFile();          Schema.Parser parser = new Schema.Parser();          Schema schema = parser.parse(new File(avscFilePath));          GenericRecord man = new GenericData.Record(schema);          man.put("name", "leo");          man.put("age", 18);          man.put("color", "yellow");            DatumWriter<GenericRecord> datumWriter = new GenericDatumWriter<GenericRecord>(schema);          DataFileWriter<GenericRecord> dataFileWriter = new DataFileWriter<GenericRecord>(datumWriter);          dataFileWriter.create(schema, new File("D:\\Temp\\man-avro-generic.out"));          dataFileWriter.append(man);          dataFileWriter.close();      }          @Test      public void genericDeserialize() throws Exception {          String avscFilePath = SpecificTest.class.getClassLoader().getResource("avro/man.avsc").getFile();          Schema.Parser parser = new Schema.Parser();          Schema schema = parser.parse(new File(avscFilePath));            DatumReader<GenericRecord> datumReader = new GenericDatumReader<GenericRecord>(schema);          DataFileReader<GenericRecord> dataFileReader = new DataFileReader<GenericRecord>(new File("D:\\Temp\\man-avro-generic.out"), datumReader);          GenericRecord man = null;          while (dataFileReader.hasNext()) {              man = dataFileReader.next(man);              System.out.println(man.get("name"));              System.out.println(man.get("age"));              System.out.println(man.get("color"));          }      }        @Test      public void specificSerialize() throws Exception {          Man man = new Man();          man.setName("leo");          man.setAge(18);          man.setColor("yellow");            DatumWriter<Man> userDatumWriter = new SpecificDatumWriter<Man>(Man.class);          DataFileWriter<Man> dataFileWriter = new DataFileWriter<Man>(userDatumWriter);          dataFileWriter.create(man.getSchema(), new File("D:\\Temp\\man-avro-specific.out"));          dataFileWriter.append(man);          man.setName("leo-01");          dataFileWriter.append(man);          man.setName("leo-02");          dataFileWriter.append(man);          man.setName("leo-03");          dataFileWriter.append(man);          dataFileWriter.close();      }        @Test      public void specificDeserialize() throws Exception {          DatumReader<Man> userDatumReader = new SpecificDatumReader<Man>(Man.class);          DataFileReader<Man> dataFileReader = new DataFileReader<Man>(new File("D:\\Temp\\man-avro-specific.out"),userDatumReader);          Man man = null;          while (dataFileReader.hasNext()) {              man = dataFileReader.next(man);              System.out.println("name=" + man.getName() + ";age=" + man.getAge());              System.out.println("color=" + man.getColor());          }      }        @Test      public void specificSerialize2() throws Exception {          Man man = new Man();          man.setName("leo");          man.setAge(18);          man.setColor("yellow");            for(int i = 0 ; i < 5 ; i++){              DatumWriter<Man> userDatumWriter = new SpecificDatumWriter<Man>(Man.class);              DataFileWriter<Man> dataFileWriter = new DataFileWriter<Man>(userDatumWriter);              dataFileWriter.create(man.getSchema(), new File("D:\\Temp\\man-avro-specific-" + i + ".out"));              man.setName("leo-" + i);              dataFileWriter.append(man);              dataFileWriter.close();          }      }  } |

**1.3         小结**

1)        Avro依赖于模式(Schema)。通过模式定义各种数据结构，只有确定了模式才能对数据进行解释，所以在数据的序列化和反序列化之前，必须先确定模式的结构。

2)        Schema采用json编写

3)        两种方式序列化生成的数据中，包函Schema

**应用**

Hadoop RPC （http://hadoop.apache.org/#What+Is+Apache+Hadoop%3F）

**优点**

1)        二进制消息，性能好/效率高

2)        使用JSON描述模式

3)        模式和数据统一存储，消息自描述，不需要生成stub代码（支持生成IDL）

4)        RPC调用在握手阶段交换模式定义

5)        包含完整的客户端/服务端堆栈，可快速实现RPC

6)        支持同步和异步通信

7)        支持动态消息

8)        模式定义允许定义数据的排序（序列化时会遵循这个顺序）

9)        提供了基于Jetty内核的服务基于Netty的服务

**缺点**

1)        只支持Avro自己的序列化格式

语言绑定不如Thrift丰富

5         Jackson - Codehaus

**5.1         简介**

Jackson可以轻松的将Java对象转换成json对象和xml文档，同样也可以将json、xml转换成Java对象。功能非常的强悍！

**5.2         示例代码**

**JacksonSerializerTest.java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | package org.leo.serialize.jackson;  import java.io.File;  import java.io.FileOutputStream;  import org.apache.commons.io.FileUtils;  import org.junit.Test;  import org.leo.serialize.domain.Man;  import org.leo.util.JsonUtils;  public class JacksonSerializerTest {      @Test      public void serialize() throws Exception {          Man man = new Man("leo", 18);          man.setColor("yellow");          //man.setId("10000");            byte[] bytes = JsonUtils.writeObjectToBytes(man);          FileOutputStream os = new FileOutputStream("D:\\Temp\\man-jackson.out");          os.write(bytes);          os.close();      }          @Test      public void deserialize() throws Exception {          //FileInputStream is = new FileInputStream("D:\\Temp\\man-jackson.out");          byte[] byteArr = FileUtils.readFileToByteArray(new File("D:\\Temp\\man-jackson.out"));          Man man = JsonUtils.readJsonToObject(Man.class, byteArr);            System.out.println("name=" + man.getName() + ";age=" + man.getAge());          System.out.println("SEX=" + Man.SEX + ";stran=" + man.stran);          System.out.println("Color=" + man.getColor());      }  } |

**5.3         小结**

1)        通过类反射机制获取属性值：accessorMethod.invoke(bean);

6         FastJson

**6.1         简介**

阿里巴巴FastJson是一个Json处理工具包，包括“序列化”和“反序列化”两部分，它具备如下特征：

速度最快，测试表明，fastjson具有极快的性能，超越任其他的Java Json parser。包括自称最快的JackJson；

功能强大，完全支持Java Bean、集合、Map、日期、Enum，支持范型，支持自省；无依赖，能够直接运行在Java SE 5.0以上版本；支持Android；开源 (Apache 2.0)

**6.2         示例代码**

**FastJsonSerializerTest.java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | package org.leo.serialize.fastjson;  import java.io.File;  import java.io.FileOutputStream;  import org.apache.commons.io.FileUtils;  import org.junit.Test;  import org.leo.serialize.domain.Man;  import com.alibaba.fastjson.JSON;  import com.alibaba.fastjson.serializer.SerializerFeature;  public class FastJsonSerializerTest {      @Test      public void serialize() throws Exception {          Man man = new Man("leo", 18);          man.setColor("yellow");          //man.setId("10000");            byte[] bytes = JSON.toJSONBytes(man, SerializerFeature.WriteEnumUsingToString,SerializerFeature.DisableCircularReferenceDetect);          FileOutputStream os = new FileOutputStream("D:\\Temp\\man-fastJson.out");          os.write(bytes);          os.close();      }          @Test      public void deserialize() throws Exception {          //FileInputStream is = new FileInputStream("D:\\Temp\\man-jackson.out");          byte[] byteArr = FileUtils.readFileToByteArray(new File("D:\\Temp\\man-fastJson.out"));          Man man = JSON.parseObject(byteArr,Man.class);            System.out.println("name=" + man.getName() + ";age=" + man.getAge());          System.out.println("SEX=" + Man.SEX + ";stran=" + man.stran);          System.out.println("Color=" + man.getColor());      }  } |

**6.3         小结**

1)        利用ASM，生成类的ObjectSerializer

2)        将生成的ObjectSerializer，保存在SerializeConfig，要用时，要怕类的class查找

3)        获取对象属性值时，直接调用该属性的get方法，避免了反射

一、      测试报告

1.       测试环境

jdk7 64bit

2.       测试一

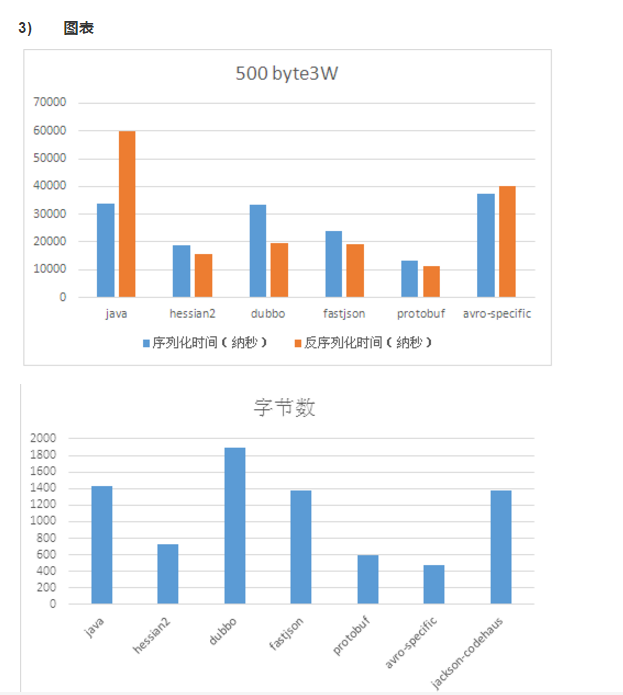
**1)        测试场景**

一个对象数据500 byte，3W个

**2)        测试结果**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **序列化时间（纳秒）** | **反序列化时间（纳秒）** | **字节数** |
| java | 33702 | 59946 | 1427 |
| hessian2 | 18742 | 15724 | 727 |
| dubbo | 33383 | 19566 | 1896 |
| fastjson | 23918 | 19077 | 1382 |
| protobuf | 13233 | 11407 | 590 |
| avro-specific | 37385 | 40332 | 482 |
| jackson-codehaus | 374701 | 444992 | 1382 |

**3)        图表**



3.       测试二

**1)        测试场景**

一个对象数据150 byte，6W个

**2)        测试结果**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **序列化时间（纳秒）** | **反序列化时间（纳秒）** | **字节数** |
| java | 12149 | 36867 | 734 |
| hessian2 | 8187 | 7794 | 362 |
| dubbo | 11276 | 7069 | 573 |
| fastjson | 6750 | 6525 | 423 |
| protobuf | 4445 | 3866 | 177 |
| avro-specific | 4735 | 14433 | 146 |
| jackson-codehaus | 327674 | 397589 | 423 |

**3)        图表**

