java序列化与Json的三个库

如果一个运行时的数据需要被持久化，或者说需要通过网络进行通信，那么就涉及到对象的序列化问题。比如说你在程序中写了一个sql，大概是这个样子：

```
INSERT into student (student_id,`name`) values (121212,null)
```

这个sql要传输到数据库，就必须进行序列化，比如说，null为什么会被识别为Null类型，而不是字符串"null"，121212为什么被识别为一个数字，而不是字符串。这就依赖于序列化和反序列化的约定。

序列化和反序列化在网络编程中更是无处不在。比如现在流行的前后端分离项目，后端接口向前端返回一个对象，在spring boot框架下经常会写成这样代码：

```
@RestController
@RequestMapping("/student")
public class StudentController {

        @GetMapping("/random")
        public Student random(){
        return new Student(121212,"章银莱");
        }
}
```

如果不做其他设置，前端将收到这样的结果：

```
{
  "studentId": 121212
  "name": "章银莱"
}
```

那么就应该好奇，为什么返回了一个json，而不是xml，不是yaml。答案是springboot默认调用jackson，将我们的类对象转换成了一个json字符串。

java对序列化的原生支持

以前，java提供了一个标记接口，java.io.Serializable，任何实现了这个接口的类，都可以使用java自己的序列化机制，实际上，因为这个接口是个标记接口，在大多数情况下，你什么都不用做。如果一个类要完整的序列化，那么他的包含的所有字段都必须实现该序列化接口，不然就会出现异常。

java通过一个序列化序号来识别一个对象，也就是我们经常在实现了java.io.Serializable接口的类里看到这个字段的原因：

```
private static final long serialVersionUID = 1L;
```

如果不提供的话，java会用某种默认生成机制生成。在反序列化时，如果序列化号对不上，反序列化会失败。

java的默认序列化策略有很多问题：

1. 序列化的声明过于繁琐。字段必须也实现该接口，否则序列化会失败；父类必须也实现该接口，否则父类的部分不会被序列化。有时候父类我们是改变不了的，那就只能接受无法序列化的结果。
2. 仅适用于java平台相关语言。其他语言没法直接反序列化，或许通过一些中间件可以支持，但是在有些复杂。在前后端分离的大趋势下，这个限制已经令人无法忍受。
3. 对修改不太友好。假如说我们没有指定序列化号，那么默认的序列号生成策略会考虑到类的现有字段，假如说你对类的字段进行了修改，那序列化号会发生改变。导致反序列化失败。类的改名也有一样的负面影响。

限制众多，显然我们需要更方便、通用性、兼容性更好的序列化方式。

json or xml

序列化并不是某一个具体的行为，而是**一类行为的统称**。我个人认为，序列化的最终目的和最终结果，是**将丰富多样的数据转换成一串二进制流**，毕竟无论是存储在硬盘中以序列化，还是通过网络接口通信，最后的物理设备都只能识别和传输二进制。所以，一个靠谱的序列化方案，显然需要做到二进制层面的一致。那么就有了两种序列化的风格，一种是直接序列化为二进制流，另一种是现将纷繁复杂的数据转化为字符串，再将字符串编码为二进制流。

`java.io.Serializable`采用了第一种方案。但很多更流行的序列化方案都采取了第二种风格。比如最经常使用的json和xml。xml确实有它的优势，但作为资源传递的媒介来说，一堆什么DTD，XSD的规范，可能绕几个月也不一定绕清楚xml的所有规范，另外废话也太多，网络传输中会比json占用更多的宽带。

相比之下，json就更有优势。JavaScript原生支持，规范要求应当使用unicode，编码方式默认使用utf-8，字符串必须用双引号，key只能是字符串，必须用双引号，等等。加上spring默认使用json作为REST的资源表述方式，至少在java web开发领域，json几乎已经一统天下。

json相比`java.io.Serializable`有很多优势：

1. 独立于java语言，可以非常便利地支持多语言。
2. 不怕类名、类字段修改。因为只要修改对不上，这个字段的值将被直接置为null，而多余的字段将会被直接丢弃。至少不会引发异常——其实还可以通过可选字段名的扩展来实现字段名修改前后的兼容。
3. 序列化支持非常广泛。因为是将字段直接映射为`key:value`这种格式，不需要其父类、包含的所有字段实现什么特定接口。

所以，在目前的编程环境下，已经不太常见`java.io.Serializable`这种东西，最多的是看到json大行其道。

扩展阅读：数据类型和Json格式，json的标准定义

look into json

一个**example**类

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
```

```
import java.time.LocalDateTime;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Student {

    private String id;
    private String name;
    private Gender gender;
    private ZonedDateTime birthday;
    private Integer age;
    private Teacher teacher;

    public enum Gender {
        MALE,
        FEMALE,
    }
}
```

常用的三个**java**处理**json**库

google的Gson，alibaba的fastJson，还有springBoot默认的jackson。性能如何，这里就不测试了，对于非极端环境下，作为这种非常优秀且使用广泛的库，基本也不需要我们担心这个问题。

引入的pom如下：

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
    <maven.compiler.target>1.8</maven.compiler.target>
    <maven.compiler.source>1.8</maven.compiler.source>
    <jackson.version>2.9.8</jackson.version>
</properties>

<dependencies>

    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>fastjson</artifactId>
        <version>1.2.55</version>
    </dependency>

    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.8.5</version>
    </dependency>
```

```
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.16.20</version>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-core</artifactId>
        <version>${jackson.version}</version>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>${jackson.version}</version>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-annotations</artifactId>
        <version>${jackson.version}</version>
    </dependency>

</dependencies>
```

可见jackson需要引用的pom最多，这有时候也不太方便。三个库的基本用法如下：

```
Gson gson = new GsonBuilder().create();
ObjectMapper mapper = new ObjectMapper();
//fastJson都是静态方法，所以不需要实例化一个解析器

Student student = new Student("1", "刘", Student.Gender.MALE,
ZonedDateTime.now(ZoneId.of("Asia/Shanghai")), 18, teacher);
String fastJsonStr = JSON.toJSONString(student);
String gsonStr = gson.toJson(student);
String jacksonStr = mapper.writeValueAsString(student);

Student student1 = JSON.parseObject(fastJsonStr, Student.class);
Student student2 = gson.fromJson(gsonStr, Student.class);
Student student3 = mapper.readValue(jacksonStr, Student.class);
```

特殊类型会被怎么处理

- 日期会被怎么处理

  假如我们有一个`LocalDateTime`的类，默认策略下，三个库对其序列化的支持如下：

Gson:{"date":{"year":2019,"month":4,"day":1},"time":{"hour":16,"minute":6,"second":47,"nano":247000000}}

Jackson:{"year":2019,"month":"APRIL","dayOfMonth":1,"dayOfWeek":"MONDAY","dayOfYear":91,"hour":16,"minute":6,"second":47,"monthValue":4,"nano":247000000,"chronology":{"id":"ISO","calendarType":"iso8601"}}

FastJson:2019-04-01T16:06:47

如果我们有一个ZonedDateTime类，默认策略下，三个库对其序列化支持如下：

Gson:{"dateTime":{"date":{"year":2019,"month":4,"day":2},"time":{"hour":17,"minute":25,"second":25,"nano":721000000}},"offset":{"totalSeconds":28800},"zone":{"id":"Asia/Shanghai"}}

JackSon: 请各位自行测试，长到令人崩溃，大概整整两页

FastJson:2019-04-02T17:25:25.721+08:00[Asia/Shanghai]

可以看到gson和jackson对于时间的记录比较完整，而fastJson就有数据的丢失，但绝绝大部分情况下，好像也不会有什么影响，而且fastJson看起来更舒服。

然而，jackson在反序列化自己序列化的时间的时候，会出现错误，解决方法在下面。

- 枚举怎么处理(默认场景)

  三个类十分统一，都将结果转化为："gender":"MALE"

- null字段怎么处理(默认场景)

  Gson和fastJson：直接忽略

  jackson：写入一个null字段。

  到底哪一种处理方式更好。显然是见仁见智的事情了。

面对泛型

根据上面的扩展阅读，一个JSON文本是一个对象（即Map）或者数组（即Array）的序列化结果。因此，我们在任何时候，都可以将一个json文本反序列化一个map或一个array(List, etc...)。比如下面的代码：

```
String fastJsonStr = JSON.toJSONString(student,
SerializerFeature.WriteMapNullValue);
String gsonStr = gson.toJson(student);
String jacksonStr = mapper.writeValueAsString(student);

Object object1 = JSON.parseObject(fastJsonStr, Map.class);
Object object2 = gson.fromJson(gsonStr, Map.class);
Object object3 = mapper.readValue(jacksonStr, Map.class);
```

甚至在默认情况下，比如下面的代码：

```
Object object2 = gson.fromJson(gsonStr, Object.class);
Object object3 = mapper.readValue(jacksonStr, Object.class);
```

这两个类都会直接将结果转化为一个map，如果要将其转为我们需要的类，则必须给他传入一个类型信息。如：

```
Student student1 = JSON.parseObject(fastJsonStr, Student.class);
Student student2 = gson.fromJson(gsonStr, Student.class);
Student student3 = mapper.readValue(jacksonStr, Student.class);
```

这样对简单对象是可以，但当面对泛型时，json往往无能为力，比如将`List<Student>`序列化后，很可能是这样：

```
[
  {
    "id": "1",
    "aaaName": "刘",
    "gender": "MALE",
    "birthday": "2019-04-03T07:23:07.623Z",
    "age": 18,
    "teacher": {
      "students": null
    }
  },
  {
    "id": "2",
    "aaaName": "马",
    "gender": "MALE",
    "birthday": "2019-04-03T07:23:07.623Z",
    "age": 18,
    "teacher": {
      "students": null
    }
  },
  {
    "id": "3",
    "aaaName": "张",
    "gender": "FEMALE",
    "birthday": "2019-04-03T07:23:07.623Z",
    "age": 18,
    "teacher": {
      "students": null
    }
  }
]
```

将其反序列化时，如果想使用gson.fromJson(gsonStr, List<Student>.class)就会出现问题。编译器会提示无法获得List<Student>的类型，因为java的泛型是java5才加入的，为了兼容老代码，采取运行时擦除类型信息的方式实现。所以List<String>在运行时和List<Integer>是一种类型，这造成了一些有意思(也令人困惑)的现象，其中之一就是：

> 泛型类并没有自己独有的Class类对象。比如并不存在List<String>.class或是List<Integer>.class，而只有List.class；

为解决这个问题，三个库都提供了自己的解决方案，主要是运行时获得Type。

- Gson解决方案

```
Type studentListType = new TypeToken<List<Student>>() {
}.getType();

List<Student> students2 = gson.fromJson(gson.toJson(students),
studentListType);
```

- FastJson解决方案

```
import com.alibaba.fastjson.TypeReference

List<Student> students2 =
JSON.parseObject(JSON.toJSONString(students),
                                 new TypeReference<List<Student>>()
{});
```

- jackson解决方案

```
import com.fasterxml.jackson.core.type.TypeReference

List<Student> students2 =
mapper.readValue(mapper.writeValueAsString(students),
                                       new TypeReference<List<Student>>
() {});
```

上述方法对于复杂泛型嵌套一样是可用的，如：

```
import lombok.Data;

import java.util.HashMap;
import java.util.List;

@Data
public class Result<T> {
```

```
    private String result = "result";
    private HashMap<String, List<T>> data = new HashMap<>();

}
```

```
Student student = new Student("1", "刘", Student.Gender.MALE,
ZonedDateTime.now(ZoneId.of("Asia/Shanghai")), 18, teacher);
Student student22 = new Student("2", "马", Student.Gender.MALE,
ZonedDateTime.now(), 18, teacher);
Student student33 = new Student("3", "张", Student.Gender.FEMALE,
ZonedDateTime.now(), 18, teacher);

List<Student> students = Arrays.asList(student, student22, student33);
Result<Student> studentResult = new Result<>();
studentResult.getData().put("students", students);

Result<Student> students1 = gson.fromJson(gson.toJson(studentResult), new
TypeToken<Result<Student>>() {
}.getType());
Result<Student> students2 =
JSON.parseObject(JSON.toJSONString(studentResult), new
com.alibaba.fastjson.TypeReference<Result<Student>>() {
});
Result<Student> students3 =
mapper.readValue(mapper.writeValueAsString(studentResult), new
com.fasterxml.jackson.core.type.TypeReference<Result<Student>>() {
});
```

扩展阅读：java泛型

自定义处理

字段名自定义

```
@com.alibaba.fastjson.annotation.JSONField(name = "aName")
@com.fasterxml.jackson.annotation.JsonProperty("aaName")
@com.google.gson.annotations.SerializedName("aaaName")
private String name;
```

分别对应了三个库的字段名自定义。

日期格式自定义

在讨论这个问题之前，需要注意的是，只有fastJson可以不做任何处理地支持java8新增的各种时间类(虽然处理结果未必是你想要的)，其他两个库都需要对解析器做一定的处理。

其中jackson需要引入新的pom：

```
<dependency>
    <groupId>com.fasterxml.jackson.datatype</groupId>
    <artifactId>jackson-datatype-jsr310</artifactId>
    <version>${jackson.version}</version>
</dependency>
```

同时代码做以下调整：

```
ObjectMapper mapper = new ObjectMapper()
        .registerModule(new JavaTimeModule())
        .setTimeZone(TimeZone.getTimeZone(ZoneId.of("Asia/Shanghai")));
```

而Gson的更复杂一点。

```
Gson gson = new GsonBuilder().registerTypeAdapter(ZonedDateTime.class, new
JsonSerializer<ZonedDateTime>() {
    @Override
    public JsonElement serialize(ZonedDateTime src, Type typeOfSrc,
JsonSerializationContext context) {
        return new JsonPrimitive(src.toInstant().toString());
    }
}).registerTypeAdapter(ZonedDateTime.class, new
JsonDeserializer<ZonedDateTime>() {
    @Override
    public ZonedDateTime deserialize(JsonElement json, Type typeOfT,
JsonDeserializationContext context) throws JsonParseException {
        String datetime = json.getAsJsonPrimitive().getAsString();
        return Instant.parse(datetime).atZone(ZoneId.of("Asia/Shanghai"));
    }
}).create();
```

实际上是直接注册了自定义的类序列化工具，如何注册自定义的类序列化工具，这个在下面再详细分析。

更改过后，新的序列化结果是：

Gson:"2019-04-02T11:33:25.847Z"

Jackson:1554204805.847000000

反序列化都能正常进行。

还可以通过注解简单定制序列化的行为：

```
@JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss.SSSZ", timezone =
"Asia/Shanghai")
```

```
@JSONField(format = "yyyy-MM-dd HH:mm:ss.SSSZ")
private ZonedDateTime birthday;
```

其中Z是时区的意思，缺失的话反序列化就会出现问题，但即使如此，Fastjson在采用这种方式序列化之后，反序列化时会仍然会丢失时区信息。

因为我们刚才的Gson实际上已经完成了完全的自定义序列化和反序列化，所以结果取决于自定义的行为是什么。

不输出某个字段

FastJson：`@JSONField(serialize = false)`

Gson：直接在字段上加transient关键字，如`private transient Gender gender;`

但因为transient是java的关键字，也会影响java的`java.io.Serializable`接口，因此要慎用。更详细的情况可参照Gson之排除字段的常见方式

jackson：`@JsonIgnore`

null值是否输出

Gson和FastJson默认不输出，因此使用下面的语句打开。

Gson:`Gson gson = new GsonBuilder().serializeNulls().create()`

FastJson:`JSON.toJSONString(student, SerializerFeature.WriteMapNullValue);`

Jackson默认输出，因此这样打开：

```
ObjectMapper mapper = new
ObjectMapper().setSerializationInclusion(JsonInclude.Include.NON_NULL);
```

把多个字段映射为一个属性

比如我们从多个源接收数据，想将他们转为反序列化为同一个类，他们的主要字段相同，但字段名却不尽相同。比如我们作为一家app公司，主要是免费加广告的方式发布app，现在有三家广告商，他们提供接口来获取广告收入数据，大概格式都是从哪个时间到哪个时间，点击了多少次，收益多少这样的数据，但字段名不一样，有些叫timeStart，有些叫beginTime，都是json现在传过来了，而我们想统一处理，这里举个例子：

```
{"序号":1,"名字":"小刘","性别":"Male","年龄":22,"生日":"2019-04-14"}
{"age":22,"birthday":"2019-04-14","gender":"Male","name":"小
刘","studentId":1}
```

```
@Data
public class Student {

    private Integer studentId = 1;
```

```
    private String name = "小刘";
    private Gender gender = Gender.Male;
    private LocalDate birthday = LocalDate.now();
    private Integer age = 22;

    public enum Gender {
        Male,
        Female
    }
}
```

FastJson:

```
@Data
public class Student {

    @JSONField(alternateNames = "序号")
    private Integer studentId = 1;

    @JSONField(alternateNames = "姓名")
    private String name = "小刘";
    @JSONField(alternateNames = "性别")
    private Gender gender = Gender.Male;
    @JSONField(alternateNames = "年龄")
    private Integer age = 22;

    public enum Gender {
        Male,
        Female
    }

}
```

则以上两个字符串反序列化结果一致。

Gson:

```
@Data
public class Student {

    @SerializedName(value = "studentId", alternate = "序号")
    private Integer studentId = 1;

    @SerializedName(value = "name", alternate = "姓名")
    private String name = "小刘";

    @SerializedName(value = "gender", alternate = "性别")
    private Gender gender = Gender.Male;
```

```
//    因为Gson对Date的自定义格式化比较复杂，为演示去掉了这个字段
//    private LocalDate birthday = LocalDate.now();

    @SerializedName(value = "age", alternate = "年龄")
    private Integer age = 22;

    public enum Gender {
        Male,
        Female
    }

}
```

jackson：

```
@Data
//@JsonSerialize(using = StudentSerializer.class)
//@JsonDeserialize(using = StudentDeserializer.class)
public class Student {

    @JsonAlias("序号")
    private Integer studentId = 1;

    @JsonAlias("姓名")
    private String name = "小刘";

    @JsonAlias("性别")
    private Gender gender = Gender.Male;

    @JsonAlias("年龄")
    private Integer age = 22;

    public enum Gender {
        Male,
        Female
    }

}
```

完全自定义的序列化与反序列化

上面已经示例了Gson的自定义序列化与反序列化，即通过registerTypeAdapter方法来注册

JsonSerializer和JsonDeserializer。其实三个库都差不太多，都是手动实现序列化和反序列化，然后通过某种方式注册到解析器里面去。

FastJson：

因为FastJson的序列化和反序列化都是静态方法，所以需要在需要序列化和反序列化的类上添加注解，或者在每次序列化和反序列化时传入自定义配置。

首先，实现自定义序列化和反序列化类：

```
public class LocalDateSerializer implements ObjectSerializer {

    public void write(JSONSerializer serializer, Object object,
                      Object fieldName, Type fieldType, int features) {
        SerializeWriter out = serializer.getWriter();
        if (object == null) {
            serializer.getWriter().writeNull();
            return;
        }
        out.write("\"活在当下\"");
    }
}
```

```
@SuppressWarnings("unchecked")
public class LocalDateDeserializer implements ObjectDeserializer {

    //不管咋样，我都给你整成当前时间，反正是测试用
    public <T> T deserialze(DefaultJSONParser parser, Type type, Object
fieldName) {
        JSONLexer lexer = parser.getLexer();
        lexer.nextToken(2);     //随意设置一个int都可以，但不设置会出异常，不明原
理
        return (T) LocalDate.now().plusDays(1);
    }

    public int getFastMatchToken() {
        //意义不明，打断点发现从来没被吊用过
        return 0;
    }
}
```

```
public static void main(String[] args) {

    Student student = new Student();

    SerializeConfig config = new SerializeConfig();
    config.put(LocalDate.class, new LocalDateSerializer());
    String jsonStr = JSON.toJSONString(student, config);
    System.out.println(jsonStr);

    ParserConfig parserConfig = new ParserConfig();
    parserConfig.putDeserializer(LocalDate.class, new
LocalDateDeserializer());
```

```
    Student student1 = JSON.parseObject(jsonStr, Student.class);
    System.out.println(student1);

}
```

```
{"age":22,"birthday":"活在当下","gender":"Male","name":"小刘","studentId":1}
Student(studentId=1, name=小刘, gender=Male, birthday=2019-04-15, age=22)
```

如果需要全局注册，则：

```
SerializeConfig.getGlobalInstance().put(LocalDate.class, new
LocalDateSerializer());
ParserConfig.getGlobalInstance().putDeserializer(LocalDate.class, new
LocalDateDeserializer());
```

如果想只针对特定的字段进行定制，则：

```
@Data
public class Student {

    private Integer studentId = 1;

    private String name = "小刘";
    private Gender gender = Gender.Male;
    @JSONField(serializeUsing = LocalDateSerializer.class, deserializeUsing
= LocalDateDeserializer.class)
    private LocalDate birthday = LocalDate.now();
    private Integer age = 22;

    public enum Gender {
        Male,
        Female
    }
}
```

@JSONField的设置项非常丰富，基本全部是关于自定义的东西，详细了解一下应该很有收获。

另外：FastJson还有一类被称为Filter的类，也可以对结果进行一定的自定义，但并非完全的自定义。大部分是用来判断哪些字段序列化哪些字段不序列化，或者改变序列化后某个字段的名字、值这样的。有兴趣可以了解一下。

Jackson：

套路都一样，首先是实现自己的序列化和反序列化类：

```
public class StudentSerializer extends JsonSerializer<Student> {

    @Override
    public void serialize(Student value, JsonGenerator jsonGenerator,
SerializerProvider provider)
        throws IOException {
        jsonGenerator.writeStartObject();
        jsonGenerator.writeNumberField("序号", value.getStudentId());
        jsonGenerator.writeStringField("名字", value.getName());
        jsonGenerator.writeStringField("性别",
value.getGender().toString());
        jsonGenerator.writeStringField("年龄", value.getAge() + "years-
old");
        jsonGenerator.writeStringField("生日",
value.getBirthday().toString());
        jsonGenerator.writeEndObject();
    }

}
```

```
public class StudentDeserializer extends JsonDeserializer<Student> {

    @Override
    public Student deserialize(JsonParser jp, DeserializationContext ctxt)
{
        return new Student();
    }

}
```

```
Student student = new Student();
SimpleModule module = new SimpleModule();
module.addSerializer(Student.class, new StudentSerializer());
module.addDeserializer(Student.class, new StudentDeserializer());
ObjectMapper mapper = new ObjectMapper().registerModule(module);
String jsonStr = mapper.writeValueAsString(student);
System.out.println(jsonStr);
Student student2 = mapper.readValue(jsonStr, Student.class);
System.out.println(student2);
```

可以看到主要是注册module来实现自定义，更简单的办法是直接在类上添加：

```
@Data
@JsonSerialize(using = StudentSerializer.class)
@JsonDeserialize(using = StudentDeserializer.class)
public class Student {
```

```
    ...
}
```

这两个注解同时可用于字段级别。

再看Gson：

可以看到，另外两个库都有通过注解自定义的方法，gson也有@JsonAdapter()注解，但gson用这个注解同时来序列化与反序列化的自定义工作，鉴于一个类或者字段不可能标注两个同样的注解，因此Gson的序列化和反序列化是写在同一个类里面的，如下：

```
public class StudentTypeAdapter extends TypeAdapter<Student> {

    public void write(JsonWriter out, Student value) throws IOException {
        out.beginObject();
        out.name("序号");
        out.value(value.getStudentId());
        out.name("姓名");
        out.value(value.getName());
        out.endObject();
    }

    public Student read(JsonReader in) throws IOException {
        Student student = new Student();
        student.setGender(Student.Gender.Female);
        in.beginObject();
        String fieldname = null;

        while (in.hasNext()) {

            JsonToken token = in.peek();
            if (token.equals(JsonToken.NAME)) {
                //get the current token
                fieldname = in.nextName();
            }
            if ("序号".equals(fieldname)) {
                student.setStudentId(in.nextInt());
            } else if ("姓名".equals(fieldname)) {
                student.setName(in.nextString());
            } else in.skipValue();
        }
        in.endObject();
        return student;
    }
```

之后用@JsonAdapter(StudentTypeAdapter.class)注解即可。

面对循环引用

设想这样的实体类：

```
@Data
public class Student {

    private Integer studentId = 1;
    private String name = "小刘";
    private Teacher teacher;

}
```

```
@Data
public class Teacher {

    private Integer teacherId = 2;
    private String name = "老王";
    private Student student;

}
```

这是一个简化版的一对一家教的数据库表。常见于@OneToOne的场景下。

之后会有这样的代码：

```
Student student = new Student();
Teacher teacher = new Teacher();
student.setTeacher(teacher);
teacher.setStudent(student);
```

那么序列化时会发生什么问题？我们测试一下：

```
ObjectMapper mapper = new ObjectMapper();
Gson gson = new GsonBuilder().create();

System.out.println(JSON.toJSONString(teacher));
System.out.println(gson.toJson(teacher));
System.out.println(mapper.writeValueAsString(teacher));
```

FastJson的输出是：{"student":{"name":"小刘","studentId":1,"teacher":{"$ref":".."}}}

Gson和Jackson爆栈异常，其实我个人觉得爆栈异常更合理，这种互相持有引用的情况应该特殊处理，fastJson替我们处理了有好有坏，比如其他语言可能无法反序列化这样它的序列化结果，因而引发异常。所以官方也提供手动关闭的功能：

```
//全局关闭
JSON.DEFAULT_GENERATE_FEATURE |=
```

```
SerializerFeature.DisableCircularReferenceDetect.getMask();
//单次关闭
JSON.toJSONString(obj, SerializerFeature.DisableCircularReferenceDetect);
```

那么假设我们真的要处理这样的情况，该如何是好呢？

一种方式是在某个类的对另一个类的引用上忽略另一个类，即不输出这个字段。这样断开循环引用。另外的方式就是通过某种方式来指明引用方式：

jackson下最全的：Jackson – Bidirectional Relationships，其中和FastJson表现形式最接近的是@JsonIdentityInfo方法。要注意的是@JsonIdentityInfo必须提供的参数是generator，其实是指定了ID的生成策略，之后通过ID解除相互引用的问题。随便选择一个可用的即可，比如：

```
@Data
public class Teacher {

    private Integer teacherId = 2;
    private String name = "老王";
    @JsonIdentityInfo(generator =
ObjectIdGenerators.StringIdGenerator.class)
    private Student student;

}
```

```
@Data
public class Student {

    private Integer studentId = 1;
    private String name = "小刘";
    @JsonIdentityInfo(generator =
ObjectIdGenerators.StringIdGenerator.class)
    private Teacher teacher;

}
```

Gson没有查到类似策略，似乎只能通过忽略字段来断开无限循环。

当然了，以上问题都可以通过完全自定义的序列化与反序列化完成工作

三个库与Spring框架整合

目前Spring框架默认使用Jackson来序列化和反序列化json，如果细致一点观察Spring框架，会发现Spring框架正是使用HttpMessageConverter来实现HTTP信息处理的，其他两个Gson库也提供了和Spring框架整合的能力，得益于Spring框架本身的松耦合，替换HttpMessageConverter是轻而易举的：

FastJson：官方文档：集成Spring框架

Gson: Configure gson in spring using GsonHttpMessageConverter

如果已经决定替换掉jackson，那还应该在pom文件中去除jackson的引用，如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
        <exclusions>
        <exclusion>
            <artifactId>jackson-databind</artifactId>
            <groupId>com.fasterxml.jackson.core</groupId>
        </exclusion>
    </exclusions>
</dependency>
```

我该如何选择呢？

整体上讲，三个框架各有特点吧。jackson能被spring官方选中自然有他的道理，对标准的严格执行，强大的可扩展性、稳定性，速度也很快，缺点是接口不那么易用，要引的包也着实有点多。Gson是谷歌的产品，我们单位用的比较多，但被fastJson公然嘲讽速度慢：

> Gson应该叫龟速Json。

fastJson速度快，接口简单，但批评者认为，使用Json本身就不是看中他的速度，而是他的兼容性和灵活性，fastJson硬编码的东西多，灵活性不够，等等。见知乎讨论：fastjson这么快老外为啥还是热衷 jackson，但是它确实很简单易用，速度也挺快。

从功能上来讲，三者可以说都是完备的。如果是我选择，想偷懒应该会用FastJson，想安全稳定面对特别复杂的需求，应该还是用jackson。Gson.....不知道为什么就是不太喜欢它.....