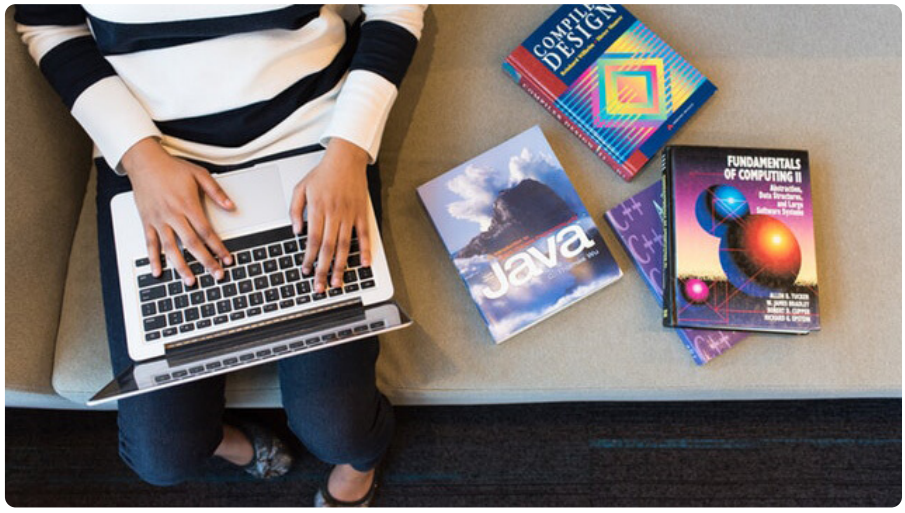
目录

03 Java序列化引发的血案

更新时间：2019-12-16 20:01:57



> 先相信你自己，然后别人才会相信你。
>
> ——屠格涅夫

1、前言

《手册》第 9 页 “OOP 规约” 部分有一段关于序列化的约定 1：

> 【强制】当序列化类新增属性时，请不要修改 serialVersionUID 字段，以避免反序列失败；如果完全不兼容升级，避免反序列化混乱，那么请修改 serialVersionUID 值。
> 说明：注意 serialVersionUID 值不一致会抛出序列化运行时异常。

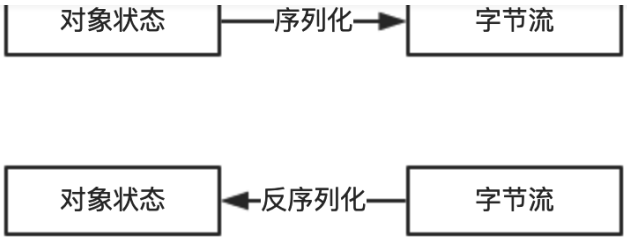
我们应该思考下面几个问题：

- 序列化和反序列化到底是什么？
- 它的主要使用场景有哪些？
- Java 序列化常见的方案有哪些？
- 各种常见序列化方案的区别有哪些？
- 实际的业务开发中有哪些坑点？

接下来将从这几个角度去研究这个问题。

2. 序列化和反序列化是什么？为什么需要它?

序列化是将内存中的对象信息转化成可以存储或者传输的数据到临时或永久存储的过程。而**反序列化**正好相反，是从临时或永久存储中读取序列化的数据并转化成内存对象的过程。



那么为什么需要序列化和反序列化呢？

希望大家能够养成从本源上思考这个问题的思维方式，即思考它为什么会出现，而不是单纯记忆。

> 大家可以回忆一下，平时都是如果将文字文件、图片文件、视频文件、软件安装包等传给小伙伴时，这些资源在计算机中存储的方式是怎样的。
>
> 进而再思考，Java 中的对象如果需要存储或者传输应该通过什么形式呢？

我们都知道，一个文件通常是一个 m 个字节的序列：B0, B1, ⋯, Bk, ⋯, Bm-1 。所有的 I/O 设备（例如网络、磁盘和终端）都被模型化为文件，而所有的输入和输出都被当作对应文件的读和写来执行。2

因此本质上讲，文本文件，图片、视频和安装包等文件底层都被转化为二进制字节流来传输的，对方得文件就需要对文件进行解析，因此就需要有能够根据不同的文件类型来解码出文件的内容的程序。

大家试想一个典型的场景：如果要实现 Java 远程方法调用，就需要将调用结果通过网路传输给调用方，如果调用方和服务提供方不在一台机器上就很难共享内存，就需要将 Java 对象进行传输。而想要将 Java 中的对象进行网络传输或存储到文件中，就需要将对象转化为二进制字节流，这就是所谓的序列化。存储或传输之后必然就需要将二进制流读取并解析成 Java 对象，这就是所谓的反序列化。

序列化的主要目的是：**方便存储到文件系统、数据库系统或网络传输等**。

实际开发中常用到序列化和反序列化的场景有：

- 远程方法调用（RPC）的框架里会用到序列化。
- 将对象存储到文件中时，需要用到序列化。
- 将对象存储到缓存数据库（如 Redis）时需要用到序列化。
- 通过序列化和反序列化的方式实现对象的深拷贝。

3. 常见的序列化方式

常见的序列化方式包括 Java 原生序列化、Hessian 序列化、Kryo 序列化、JSON 序列化等。

3.1 Java 原生序列化

Serializable 的源码非常简单，只有声明，没有属性和方法：

```
// 注释太长，省略
public interface Serializable {
}
```

在学习源码注释之前，希望大家可以站在设计者的角度，先思考一个问题：如果一个类序列化到文件之后，类的结构发生变化还能否保证正确地反序列化呢？

答案显然是不确定的。

那么如何判断文件被修改过了呢？ 通常可以通过加密算法对其进行签名，文件作出任何修改签名就会不一致。但是 Java 序列化的场景并不适合使用上述的方案，因为类文件的某些位置加个空格，换行等符号类的结构没有发生变化，这个签名就不应该发生变化。还有一个类新增一个属性，之前的属性都是有值的，之前都被序列化到对象文件中，有些场景下还希望反序列化时可以正常解析，怎么办呢？

那么是否可以通过约定一个唯一的 ID，通过 ID 对比，不一致就认为不可反序列化呢？

实现序列化接口后，如果开发者不手动指定该版本号 ID 怎么办？

既然 Java 序列化场景下的 “签名” 应该根据类的特点生成，我们是否可以不指定序列化版本号就默认根据类名、属性和函数等计算呢？

如果针对某个自己定义的类，想自定义序列化和反序列化机制该如何实现呢？支持吗？

带着这些问题我们继续看序列化接口的注释。

Serializable 的源码注释特别长，其核心大致作了下面的说明：

Java 原生序列化需要实现 Serializable 接口。序列化接口不包含任何方法和属性等，它只起到序列化标识作用。

一个类实现序列化接口则其子类型也会继承序列化能力，但是实现序列化接口的类中有其他对象的引用，则其他对象也要实现序列化接口。序列化时如果抛出 NotSerializableException 异常，说明该对象没有实现 Serializable 接口。

每个序列化类都有一个叫 serialVersionUID 的版本号，反序列化时会校验待反射的类的序列化版本号和加载的序列化字节流中的版本号是否一致，如果序列化号不一致则会抛出 InvalidClassException 异常。

强烈推荐每个序列化类都手动指定其 serialVersionUID，如果不手动指定，那么编译器会动态生成默认的序列化号，因为这个默认的序列化号和类的特征以及编译器的实现都有关系，很容易在反序列化时抛出 InvalidClassException 异常。建议将这个序列化版本号声明为私有，以避免运行时被修改。

实现序列化接口的类可以提供自定义的函数修改默认的序列化和反序列化行为。

自定义序列化方法：

自定义反序列化方法：

```
private void readObject(ObjectInputStream in)
  throws IOException, ClassNotFoundException;
```

通过自定义这两个函数，可以实现序列化和反序列化不可序列化的属性，也可以对序列化的数据进行数据的加密和解密处理。

3.2 Hessian 序列化

Hessian 是一个动态类型，二进制序列化，也是一个基于对象传输的网络协议。Hessian 是一种跨语言的序列化方案，序列化后的字节数更少，效率更高。Hessian 序列化会把复杂对象的属性映射到 Map 中再进行序列化。

3.3 Kryo 序列化

Kryo 是一个快速高效的 Java 序列化和克隆工具。Kryo 的目标是快速、字节少和易用。Kryo 还可以自动进行深拷贝或者浅拷贝。Kryo 的拷贝是对象到对象的拷贝而不是对象到字节，再从字节到对象的恢复。Kryo 为了保证序列化的高效率，会提前加载需要的类，这会带一些消耗，但是这是序列化后文件较小且反序列化非常快的重要原因。

3.4 JSON 序列化

JSON (JavaScript Object Notation) 是一种轻量级的数据交换方式。JSON 序列化是基于 JSON 这种结构来实现的。JSON 序列化将对象转化成 JSON 字符串，JSON 反序列化则是将 JSON 字符串转回对象的过程。常用的 JSON 序列化和反序列化的库有 Jackson、GSON、Fastjson 等。

4.Java 常见的序列化方案对比

我们想要对比各种序列化方案的优劣无外乎两点，一点是查资料，一点是自己写代码验证。

4.1 Java 原生序列化

Java 序列化的优点是：对对象的结构描述清晰，反序列化更安全。主要缺点是：效率低，序列化后的二进制流较大。

4.2 Hessian 序列化

Hession 序列化二进制流较 Java 序列化更小，且序列化和反序列化耗时更短。但是父类和子类有相同类型属性时，由于先序列化子类再序列化父类，因此反序列化时子类的同名属性会被父类的值覆盖掉，开发时要特别注意这种情况。

> Hession2.0 序列化二进制流大小是 Java 序列化的 50%，序列化耗时是 Java 序列化的 30%，反序列化的耗时是 Java 序列化的 20%。 3

编写待测试的类：

```
    private Long id;
    private String name;
    private Boolean male;
}

@Data
public class Male extends PersonHessian {
    private Long id;
}
```

编写单测来模拟序列化继承覆盖问题：

```
/**
 * 验证Hessian序列化继承覆盖问题
 */
@Test
public void testHessianSerial() throws IOException {
    HessianSerialUtil.writeObject(file, male);
    Male maleGet = HessianSerialUtil.readObject(file);
    // 相等
    Assert.assertEquals(male.getName(), maleGet.getName());
    // male.getId()结果是1，maleGet.getId()结果是null
    Assert.assertNull(maleGet.getId());
    Assert.assertNotEquals(male.getId(), maleGet);
}
```

上述单测示例验证了：反序列化时子类的同名属性会被父类的值覆盖掉的问题。

4.3 Kryo 序列化

Kryo 优点是：速度快、序列化后二进制流体积小、反序列化超快。但是缺点是：跨语言支持复杂。注册模式序列化更快，但是编程更加复杂。

4.4 JSON 序列化

JSON 序列化的优势在于可读性更强。主要缺点是：没有携带类型信息，只有提供了准确的类型信息才能准确地进行反序列化，这点也特别容易引发线上问题。

下面给出使用 Gson 框架模拟 JSON 序列化时遇到的反序列化问题的示例代码：

```
/**
 * 验证GSON序列化类型错误
 */
@Test
public void testGSON() {
    Map<String, Object> map = new HashMap<>();
    final String name = "name";
    final String id = "id";
    map.put(name, "张三");
    map.put(id, 20L);

    String jsonString = GSONSerialUtil.getJsonString(map);
    Map<String, Object> mapGSON = GSONSerialUtil.parseJson(jsonString, Map.class);
    // 正确
    Assert.assertEquals(map.get(name), mapGSON.get(name));
    // 不等  map.get(id)为Long类型 mapGSON.get(id)为Double类型
    Assert.assertNotEquals(map.get(id).getClass(), mapGSON.get(id).getClass());
```

下面给出使用 fastjson 模拟 JSON 反序列化问题的示例代码：

```
/**
 * 验证FatJson序列化类型错误
 */
@Test
public void testFastJson() {
    Map<String, Object> map = new HashMap<>();
    final String name = "name";
    final String id = "id";
    map.put(name, "张三");
    map.put(id, 20L);

    String fastJsonString = FastJsonUtil.getJsonString(map);
    Map<String, Object> mapFastJson = FastJsonUtil.parseJson(fastJsonString, Map.class);

    // 正确
    Assert.assertEquals(map.get(name), mapFastJson.get(name));
    // 错误  map.get(id)为Long类型 mapFastJson.get(id)为Integer类型
    Assert.assertNotEquals(map.get(id).getClass(), mapFastJson.get(id).getClass());
    Assert.assertNotEquals(map.get(id), mapFastJson.get(id));
}
```

大家还可以通过单元测试构造大量复杂对象对比各种序列化方式或框架的效率。

如定义下列测试类为 User，包括以下多种类型的属性：


```
@Data
public class User implements Serializable {
    private Long id;
    private String name;
    private Integer age;
    private Boolean sex;
    private String nickName;
    private Date birthDay;
    private Double salary;
}
```

4.5 各种常见的序列化性能排序

实验的版本：kryo-shaded 使用 4.0.2 版本，gson 使用 2.8.5 版本，hessian 用 4.0.62 版本。

实验的数据：构造 50 万 User 对象运行多次。

大致得出一个结论：

- 从二进制流大小来讲：JSON 序列化 > Java 序列化 > Hessian2 序列化 > Kryo 序列化 > Kryo 序列化注册模式；
- 从序列化耗时而言来讲：GSON 序列化 > Java 序列化 > Kryo 序列化 > Hessian2 序列化 > Kryo 序列化注册模式；
- 从反序列化耗时而言来讲：GSON 序列化 > Java 序列化 > Hessian2 序列化 > Kryo 序列化注册模式 > Kryo 序列化；
- 从总耗时而言：Kryo 序列化注册模式耗时最短。

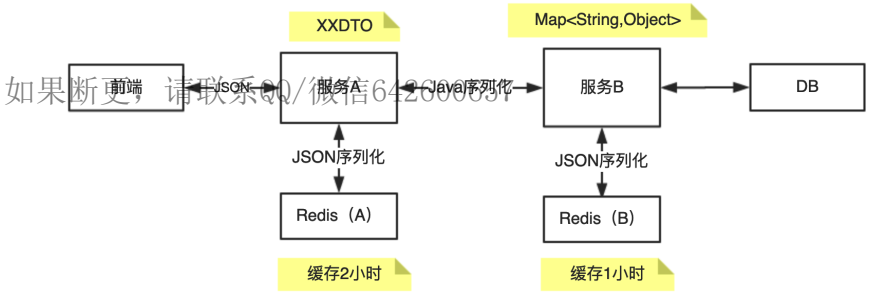
> 因结果可能会有出入。

5. 序列化引发的一个血案

接下来我们看下面的一个案例：

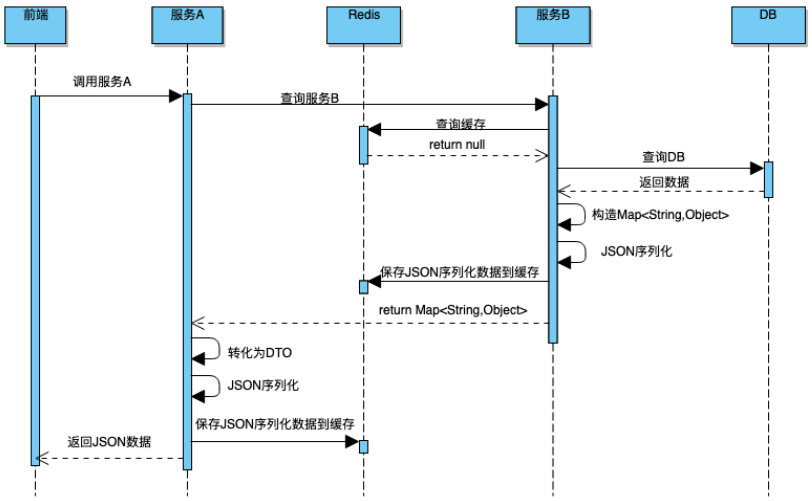
> 前端调用服务 A，服务 A 调用服务 B，服务 B 首次接到请求会查 DB，然后缓存到 Redis（缓存 1 个小时）。服务 A 根据服务 B 返回的数据后执行一些处理逻辑，处理后形成新的对象存到 Redis（缓存 2 个小时）。
>
> 服务 A 通过 Dubbo 来调用服务 B，A 和 B 之间数据通过 `Map<String,Object>` 类型传输，服务 B 使用 Fastjson 来实现 JSON 的序列化和反序列化。
>
> 服务 B 的接口返回的 `Map` 值中存在一个 `Long` 类型的 `id` 字段，服务 A 获取到 `Map`，取出 `id` 字段并强转为 `Long` 类型使用。

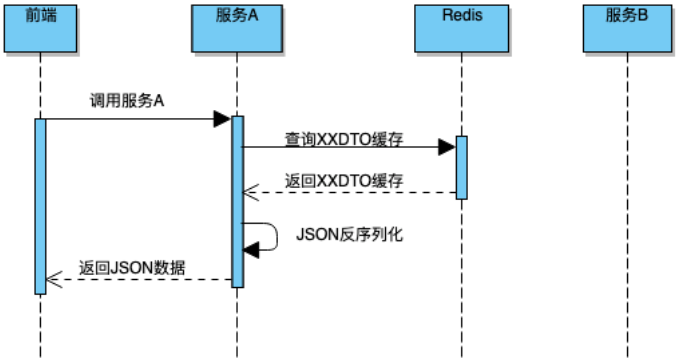
执行的流程如下：



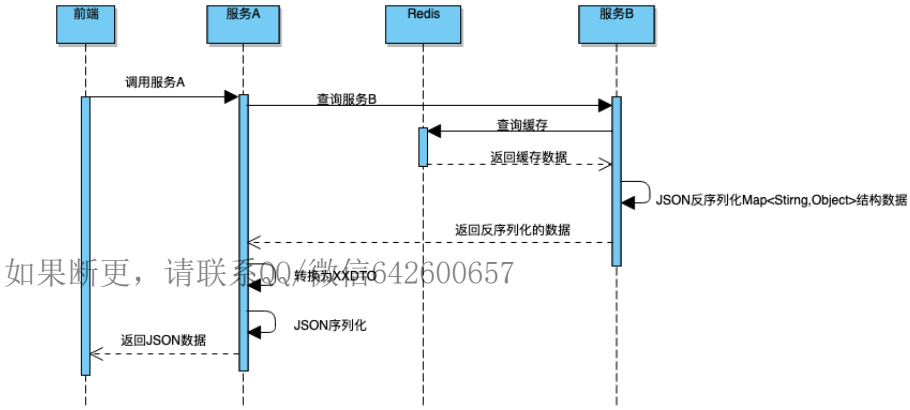
通过分析我们发现，服务 A 和服务 B 的 RPC 调用使用 Java 序列化，因此类型信息不会丢失。

但是由于服务 B 采用 JSON 序列化进行缓存，第一次访问没啥问题，其执行流程如下：





如果服务 A 不开启缓存，服务 A 会请求服务 B ，由于首次请求时，服务 B 已经缓存了数据，服务 B 从 Redis（B）中反序列化得到 Map 。流程如下图所示：



然而问题来了： 服务 A 从 Map 取出此 Id 字段，强转为 Long 时会出现类型转换异常。

最后定位到原因是 Json 反序列化 Map 时如果原始值小于 Int 最大值，反序列化后原本为 Long 类型的字段，变为了 Integer 类型，服务 B 的同学紧急修复。

服务 A 开启缓存时， 虽然采用了 JSON 序列化存入缓存，但是采用 DTO 对象而不是 Map 来存放属性，所以 JSON 反序列化没有问题。

因此大家使用二方或者三方服务时，当对方返回的是 Map<String,Object> 类型的数据时要特别注意这个问题。

> 作为服务提供方，可以采用 JDK 或者 Hessian 等序列化方式；
>
> 作为服务的使用方，我们不要从 Map 中一个字段一个字段获取和转换，可以使用 JSON 库直接将 Map 映射成所需的对象，这样做不仅代码更简洁还可以避免强转失败。

代码示例：

```
    Map<String, Object> map = new HashMap<>();
    final String name = "name";
    final String id = "id";
    map.put(name, "张三");
    map.put(id, 20L);

    String fastJsonString = FastJsonUtil.getJsonString(map);
    // 模拟拿到服务B的数据
    Map<String, Object> mapFastJson = FastJsonUtil.parseJson(fastJsonString,map.getClass
    // 转成强类型属性的对象而不是使用map 单个取值
    User user = new JSONObject(mapFastJson).toJavaObject(User.class);
    // 正确
    Assert.assertEquals(map.get(name), user.getName());
    // 正确
    Assert.assertEquals(map.get(id), user.getId());
}
```

6. 总结

本节的主要讲解了序列化的主要概念、主要实现方式，以及序列化和反序列化的几个坑点，希望大家在实际业务开发中能够注意这些细节，避免趟坑。

下一节将讲述浅拷贝和深拷贝的相关知识。

7. 课后题

给出一个 PersonTransit 类，一个 Address 类，假设 Address 是其它 jar 包中的类，没实现序列化接口。请使用今天讲述的自定义的函数 writeObject 和 readObject 函数实现 PersonTransit 对象的序列化，要求反序列化后 address 的值正常。

```
@Data
public class PersonTransit implements Serializable {

    private Long id;
    private String name;
    private Boolean male;
    private List<PersonTransit> friends;
    private Address address;
}

@Data
@AllArgsConstructor
public class Address {
    private String detail;
  }
```

参考资料

1. 阿里巴巴与 Java 社区开发者.《 Java 开发手册 1.5.0》华山版. 2019. 9 ↩

2. [美] Randal E.Bryant/ David O’Hallaron.《深入理解计算机系统》. [译] 龚奕利，贺莲。机械工业出版社. 2016 ↩

3. 杨冠宝。高海慧.《码出高效：Java 开发手册》. 电子工业出版社. 2018 ↩

目录

精选留言 3

欢迎在这里发表留言，作者筛选后可公开显示

慕UI2268095

5的示例代码没写错？user那里怎么是map?

0 回复 4天前

明明如月 回复 **慕UI2268095**

复制有误，已经订正，感谢反馈。 User user = new JSONObject(mapFastJson).toJavaObject(User.class);

回复 1天前

慕无忌4011151

老师，不太会最后的序列化习题，可以提供一个思路吗

0 回复 2019-11-17

明明如月 回复 **慕无忌4011151**

尽量自己动手写，方法都已经告诉你了，要培养自学能力，查下资料。 大家自己动手之后可以参考下我的慕课手记的参考答案：https://www.imooc.com/article/295939

回复 2019-11-18 23:35:07

慕无忌4011151 回复 **明明如月**

好的谢谢

回复 2019-11-18 23:52:41

慕粉3543028

人气有点少，这么好的课，可惜了，最值的专栏

4 回复 2019-11-09

明明如月 回复 **慕粉3543028**

多谢支持。 其实很多人并不懂得方法的价值，并不能意识到自己再走弯路。 其实本专栏所要讲的更多是基于《手册》知识点，然后通过Java语言规范、Java虚拟机规范、反汇编等重要资料和手段养成快速解决问题的习惯。 基于《手册》，然后脱离《手册》，用来学习其他知识。 通过个人的开发经验，总结的好的学习进阶方法，提高大家的进阶速度。 相信会对很多人有帮助，会让很多人少走弯路。 如果有收获，欢迎大家向周围的同学、同事、朋友宣传，感谢。

回复 2019-11-11 12:53:13

千学不如一看，千看不如一练