Java 领域的对象如何传输

基于 socket 进行对象传输

先举个简单的例子，基于我们前面几次课程的只是，写一个 socket 通信的代码

User

```
public class User {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

SocketServerProvider

```
public static void main(String[] args) throws
IOException {
    ServerSocket serverSocket=null;
    BufferedReader in=null;
    try{
        serverSocket=new ServerSocket(8080);
        Socket socket=serverSocket.accept();
        ObjectInputStream objectInputStream=
                new
ObjectInputStream(socket.getInputStream());
        User user=(User)objectInputStream.readObject();
        System.out.println(user);
    }catch (Exception e){
        e.printStackTrace();
```

```
    }finally {
        if(in!=null){
            try {
                in.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(serverSocket!=null){
            serverSocket.close();
        }
    }
}
```

SocketClientConsumer

```
public static void main(String[] args) {
    Socket socket=null;
    ObjectOutputStream out=null;
    try {
        socket=new Socket("127.0.0.1",8080);
        User user=new User();
        out=new
ObjectOutputStream(socket.getOutputStream());
        out.writeObject(user);
    } catch (IOException e) {
        e.printStackTrace();
    }finally {
        if(out!=null){
            try {
                out.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
```

```
        }
        if(socket!=null){
            try {
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

运行结果

- 这段代码运行以后，能够实现 Java 对象的正常传输吗？

  很显然，会报错

```
java.io.NotSerializableException: com.gupaoedu.course.User
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1184)
    at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:348)
    at com.gupaoedu.course.Consumer.main(Consumer.java:22)
```

如何解决报错的问题呢？

对 User 这个对象实现一个 Serializable 接口，再次运行就可以看到对象能够正常传输了

```
public class User implements Serializable {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

了解序列化的意义

我们发现对 User 这个类增加一个 Serializable，就可以解决 Java 对象的网络传输问题。这就是今天想给大家讲解的序列化这块的意义

Java 平台允许我们在内存中创建可复用的 Java 对象，但一般情况下，只有当 JVM 处于运行时，这些对象才可能存在，即，这些对象的生命周期不会比 JVM 的生命周期更长。但在现实应用中，就可能要求在 JVM 停止运行之后能够保存(持久化)指定的对象，并在将来重新读取被保存的对象。Java 对象序列化就能够帮助我们实现该功能

简单来说
序列化是把对象的状态信息转化为可存储或传输的形式过程，也就是把对象转化为字节序列的过程称为对象的序列化
反序列化是序列化的逆向过程，把字节数组反序列化为对象，把字节序列恢复为对象的过程成为对象的反序列化

序列化的高阶认识

简单认识一下 Java 原生序列化

前面的代码中演示了，如何通过 JDK 提供了 Java 对象的序列化方式实现对象序列化传输，主要通过输出流java.io.ObjectOutputStream 和对象输入流java.io.ObjectInputStream 来实现。

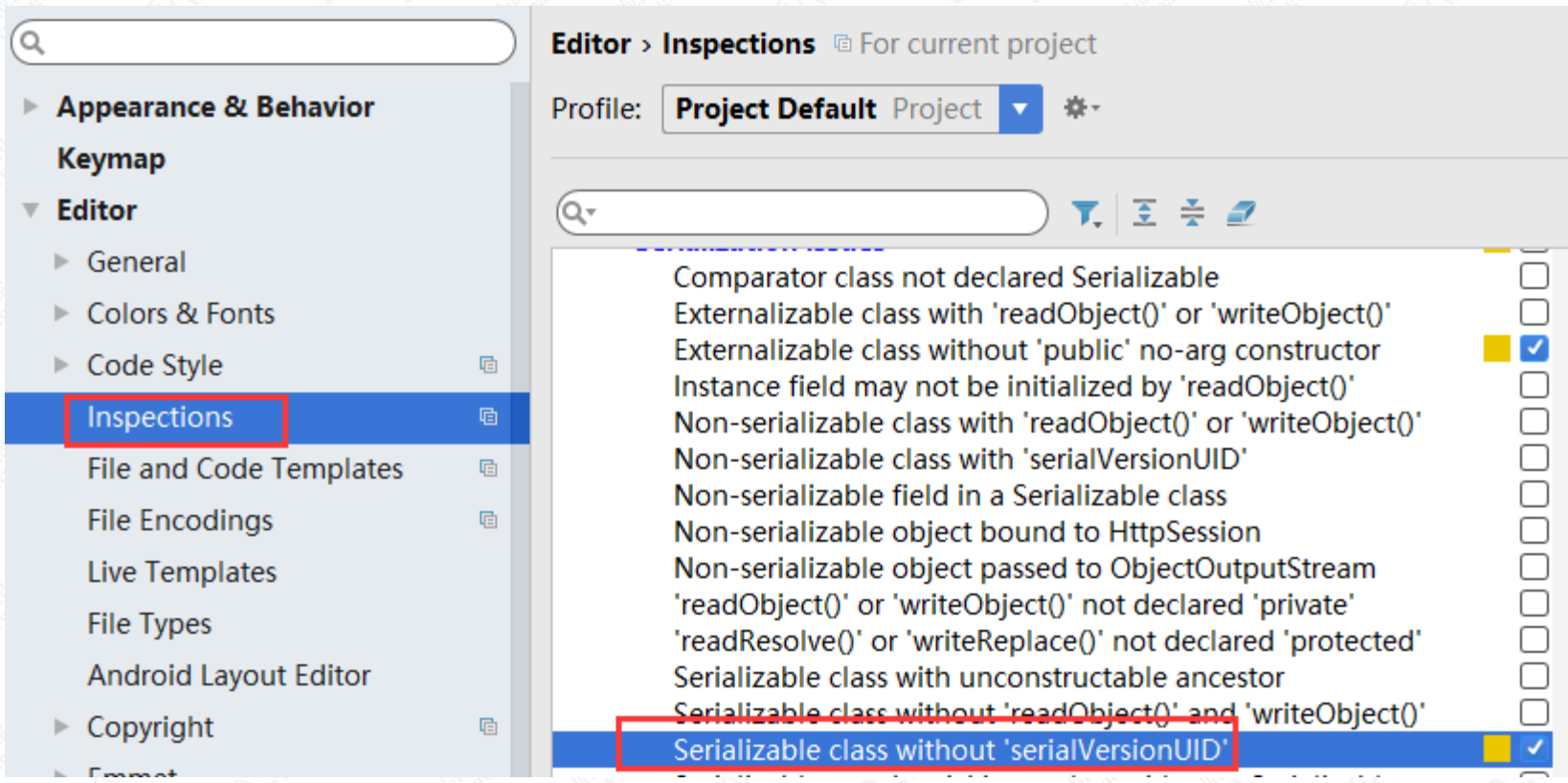
java.io.ObjectOutputStream：表示对象输出流 ，它的 writeObject(Object obj)方法可以对参数指定的 obj 对象进行序列化，把得到的字节序列写到一个目标输出流中。
java.io.ObjectInputStream：表示对象输入流 ,它的 readObject()方法源输入流中读取字节序列，再把它们反序列化成为一个对象，并将其返回

需要注意的是，被序列化的对象需要实现 java.io.Serializable 接口

序列化的高阶认识

serialVersionUID 的作用

在 IDEA 中通过如下设置可以生成 serializeid



字面意思上是序列化的版本号，凡是实现 Serializable 接口的类都有一个表示序列化版本标识符的静态变量

演示步骤

1. 先将 user 对象序列化到文件中
2. 然后修改 user 对象，增加 serialVersionUID 字段
3. 然后通过反序列化来把对象提取出来
4. 演示预期结果：提示无法反序列化

结论

Java 的序列化机制是通过判断类的 serialVersionUID 来验证版本一致性的。在进行反序列化时，JVM 会把传来的字节流中的 serialVersionUID 与本地相应实体类的 serialVersionUID 进行比较，如果相同就认为是一致的，可以进行反序列化，否则就会出现序列化版本不一致的异常，即是 InvalidCastException。

从结果可以看出，文件流中的 class 和 classpath 中的 class，也就是修改过后的 class，不兼容了，处于安全机制考虑，程序抛出了错误，并且拒绝载入。从错误结果来看，如果没有为指定的 class 配置 serialVersionUID，那么 java 编译器会自动给这个 class 进行一个摘要算法，类似于指纹算法，只要这个文件有任何改动，得到的 UID 就会截然不同的，可以保证在这么多类中，这个编号是唯一的。所以，由于没有显指定 serialVersionUID，编译器又为我们生成

了一个 UID，当然和前面保存在文件中的那个不会一样了，于是就出现了 2 个序列化版本号不一致的错误。因此，只要我们自己指定了 serialVersionUID，就可以在序列化后，去添加一个字段，或者方法，而不会影响到后期的还原，还原后的对象照样可以使用，而且还多了方法或者属性可以用。

tips: serialVersionUID 有两种显示的生成方式：

一是默认的 1L，比如：private static final long serialVersionUID = 1L;

二是根据类名、接口名、成员方法及属性等来生成一个 64 位的哈希字段

当实现 java.io.Serializable 接口的类没有显式地定义一个 serialVersionUID 变量时候，Java 序列化机制会根据编译的 Class 自动生成一个 serialVersionUID 作序列化版本比较用，这种情况下，如果 Class 文件(类名，方法明等)没有发生变化(增加空格，换行，增加注释等等)，就算再编译多次，serialVersionUID 也不会变化的。

Transient 关键字

Transient 关键字的作用是控制变量的序列化，在变量声明前加上该关键字，可以阻止该变量被序列化到文件中，在被反序列化后，transient 变量的值被设为初始值，如 int 型的是 0，对象型的是 null。

绕开 transient 机制的办法

虽然 name 被 transient 修饰，但是通过我们写的这两个方法依然能够使得 name 字段正确被序列化和反序列化

writeObject 和 readObject 原理

writeObject 和 readObject 是两个私有的方法，他们是什么时候被调用的呢？从运行结果来看，它确实被调用。而且他们并不存在于 Java.lang.Object，也没有在 Serializable 中去声明。我们唯一的猜想应该还是和 ObjectInputStream 和 ObjectOutputStream 有关系，所以基于这个入口去看看在哪个地方有调用

```
 */
void invokeReadObject(Object obj, ObjectInputStream in)
    throws ClassNotFoundException, IOException,
           UnsupportedOperationException
{
    requireInitialized();
    if (readObjectMethod != null) {
        try {
            readObjectMethod.invoke(obj, new Object[]{ in });
        } catch (InvocationTargetException ex) {
            Throwable th = ex.getTargetException();
            if (th instanceof ClassNotFoundException) {
                throw (ClassNotFoundException) th;
            } else if (th instanceof IOException) {
                throw (IOException) th;
            } else {
                throwMiscException(th);
```

从源码层面来分析可以看到，readObject 是通过反射来调用的。

其实我们可以在很多地方看到 readObject 和 writeObject 的使用，比如 HashMap。

Java 序列化的一些简单总结

1. Java 序列化只是针对对象的状态进行保存，至于对象中的方法，序列化不关心
2. 当一个父类实现了序列化，那么子类会自动实现序列化，不需要显示实现序列化接口
3. 当一个对象的实例变量引用了其他对象，序列化这个对象的时候会自动把引用的对象也进行序列化（实现深度克隆）
4. 当某个字段被申明为 transient 后，默认的序列化机制会忽略这个字段
5. 被申明为 transient 的字段，如果需要序列化，可以添加两个私有方法：writeObject 和 readObject

分布式架构下常见序列化技术

初步了解了 Java 序列化的知识以后，我们又得回到分布式架构中，了解序列化的发展过程

了解序列化的发展

随着分布式架构、微服务架构的普及。服务与服务之间的通信成了最基本的需求。这个时候，我们不仅需要考虑通信的性能，也需要考虑到语言多元化问题

所以，对于序列化来说，如何去提升序列化性能以及解决跨语言问题，就成了一个重点考虑的问题。

由于 Java 本身提供的序列化机制存在两个问题

1. 序列化的数据比较大，传输效率低
2. 其他语言无法识别和对接

以至于在后来的很长一段时间，基于 XML 格式编码的对象序列化机制成为了主流，一方面解决了多语言兼容问题，另一方面比二进制的序列化方式更容易理解。以至于基于 XML 的 SOAP 协议及对应的 WebService 框架在很长一段时间内成为各个主流开发语言的必备的技术。

再到后来，基于 JSON 的简单文本格式编码的 HTTP REST 接口又基本上取代了复杂的 Web Service 接口，成为分布式架构中远程通信的首要选择。但是 JSON 序列化存储占用的空间大、性能低等问题，同时移动客户端应用需要更高效的传输数据来提升用户体验。在这种情况下与语言无关并且高效的二进制编码协议就成为了大家追求的热点技术之一。首先诞生的一个开源的二进制序列化框架-MessagePack。它比 google 的 Protocol Buffers 出现得还要早。

简单了解各种序列化技术

XML 序列化框架介绍

XML 序列化的好处在于可读性好，方便阅读和调试。但是序列化以后的字节码文件比较大，而且效率不高，适用于对性能不高，而且 QPS 较低的企业级内部系统之间的数据交换的场景，同时 XML 又具有语言无关性，所以还可以用于异构系统之间的数据交换和协议。比如我们熟知的 Webservice，就是采用 XML 格式对数据进行序列化的。XML 序列化/反序列化的实现方式有很多，熟知的方式有 XStream 和 Java 自带的 XML 序列化和反序列化两种

JSON 序列化框架

JSON (JavaScript Object Notation) 是一种轻量级的数据交换格式，相对于 XML 来说，JSON 的字节流更小，而且可读性也非常好。现在 JSON 数据格式在企业运用是最普遍的

JSON 序列化常用的开源工具有很多

1. Jackson (https://github.com/FasterXML/jackson)
2. 阿里开源的 FastJson (https://github.com/alibaba/fastjon)
3. Google 的 GSON (https://github.com/google/gson)

这几种 json 序列化工具中，Jackson 与 fastjson 要比 GSON 的性能要好，但是 Jackson、GSON 的稳定性要比 Fastjson 好。而 fastjson 的优势在于提供的 api 非常容易使用

Hessian 序列化框架

Hessian 是一个支持跨语言传输的二进制序列化协议，相对于 Java 默认的序列化机制来说，Hessian 具有更好的性能和易用性，而且支持多种不同的语言

实际上 Dubbo 采用的就是 Hessian 序列化来实现，只不过 Dubbo 对 Hessian 进行了重构，性能更高

Avro 序列化

Avro 是一个数据序列化系统，设计用于支持大批量数据交换的应用。它的主要特点有：支持二进制序列化方式，可以便捷，快速地处理大量数据；动态语言友好，Avro 提供的机制使动态语言可以方便地处理 Avro 数据。

kyro 序列化框架

Kryo 是一种非常成熟的序列化实现，已经在 Hive、Storm）中使用得比较广泛，不过它不能跨语言. 目前 dubbo 已经在 2.6 版本支持 kyro 的序列化机制。它的性能要优于之前的 hessian2

Protobuf 序列化框架

Protobuf 是 Google 的一种数据交换格式，它独立于语言、独立于平台。Google 提供了多种语言来实现，比如 Java、C、Go、Python，每一种实现都包含了相应语言的编译器和库文件，Protobuf 是一个纯粹的表示层协议，可以和各种传输层协议一起使用。
Protobuf 使用比较广泛，主要是空间开销小和性能比较好，非常适合用于公司内部对性能要求高的 RPC 调用。 另外由于解析性能比较高，序列化以后数据量相对较少，所以也可以应用在对象的持久化场景中
但是要使用 Protobuf 会相对来说麻烦些，因为他有自己的语法，有自己的编译器，如果需要用到的话必须要去投入成本在这个技术的学习中

protobuf 有个缺点就是要传输的每一个类的结构都要生成对应的 proto 文件，如果某个类发生修改，还得重新生成该类对应的 proto 文件

Protobuf 序列化的原理

那么接下来着重分析一下 protobuf 的序列化原理，前面说过它的优势是空间开销小，性能也相对较好。它里面用到的一些算法还是值得我们去学习的

protobuf 的基本应用

使用 protobuf 开发的一般步骤是

1. 配置开发环境，安装 protocol compiler 代码编译器
2. 编写.proto 文件，定义序列化对象的数据结构
3. 基于编写的.proto 文件，使用 protocol compiler 编译器生成对应的序列化/反序列化工具类
4. 基于自动生成的代码，编写自己的序列化应用

Protobuf 案例演示

下载 protobuf 工具

https://github.com/google/protobuf/releases 找到 protoc-3.5.1-win32.zip

编写 proto 文件

```
syntax="proto2";
package com.gupaoedu.serial;

option java_package = "com.gupaoedu.serial";
option java_outer_classname="UserProtos";

message User {
    required string name=1;
    required int32 age=2;
}
```

数据类型

string / bytes / bool / int32（4 个字节）/int64/float/double

enum 枚举类

message 自定义类

修饰符

required 表示必填字段

optional 表示可选字段

repeated 可重复，表示集合

1，2，3，4 需要在当前范围内是唯一的，表示顺序

生成实体类

【.\protoc.exe --java_out=./ ./user.proto】

实现序列化

```
<dependency>

<groupId>com.google.protobuf</groupId>
```

```
UserProtos.User user=UserProtos.User.newBuilder().setName("Mic").setAge(18).build();
```

<artifactId>protobuf-java</artifactId> <version>3.7.0</version> </dependency>	ByteString bytes=user.toByteString(); System.*out*.println(bytes); UserProtos.User nUser=UserProtos.User.*parseFrom*(bytes); System.*out*.println(nUser);

protobuf 序列化原理

我们可以把序列化以后的数据打印出来看看结果

```
public static void main(String[] args) {
    UserProtos.User user=UserProtos.User.newBuilder().
            setAge(300).setName("Mic").build();
    byte[] bytes=user.toByteArray();
    for(byte bt:bytes){
        System.out.print(bt+" ");
    }
}
```

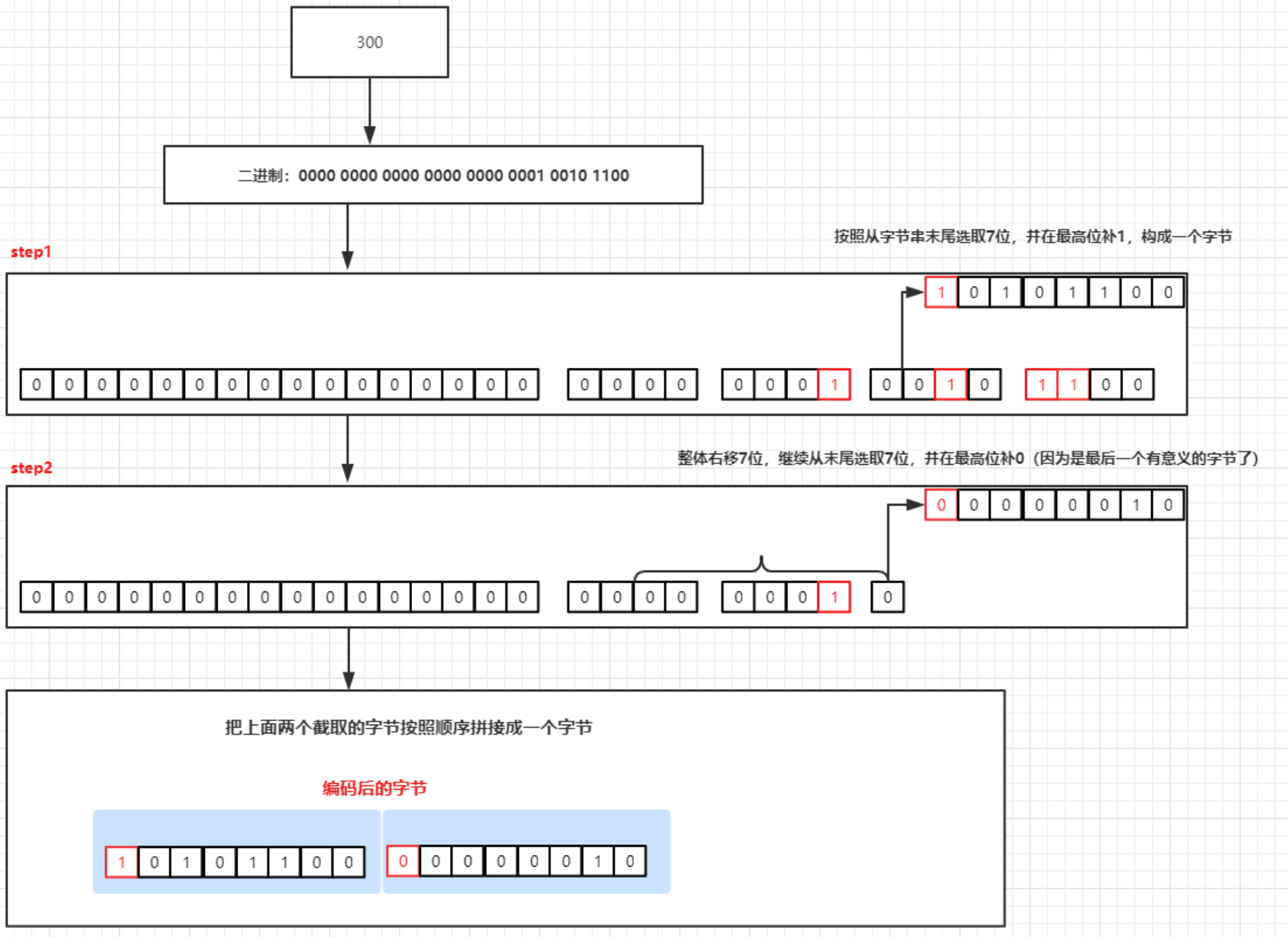
➢ **10 3 77 105 99 16 -84 2**

我们可以看到，序列化出来的数字基本看不懂，但是序列化以后的数据确实很小，那我们接下来带大家去了解一下底层的原理

正常来说，要达到最小的序列化结果，一定会用到压缩的技术，而 protobuf 里面用到了两种压缩算法，一种是 varint，另一种是 zigzag

varint

先说第一种，我们先来看 age=300 这个数字是如何被压缩的



这两个字节字节分别的结果是：-84 、2

-84 怎么计算来的呢？ 我们知道在二进制中表示负数的方法，高位设置为 1， 并且是对应数字的二进制取反以后再计算补码表示（补码是反码+1）

所以如果要反过来计算

1. 【补码】10101100 -1 得到 10101011
2. 【反码】01010100 得到的结果为 84. 由于高位是 1，表示负数所以结果为-84

字符如何转化为编码

“Mic”这个字符，需要根据 ASCII 对照表转化为数字。

M =77、i=105、c=99

所以结果为 77 105 99

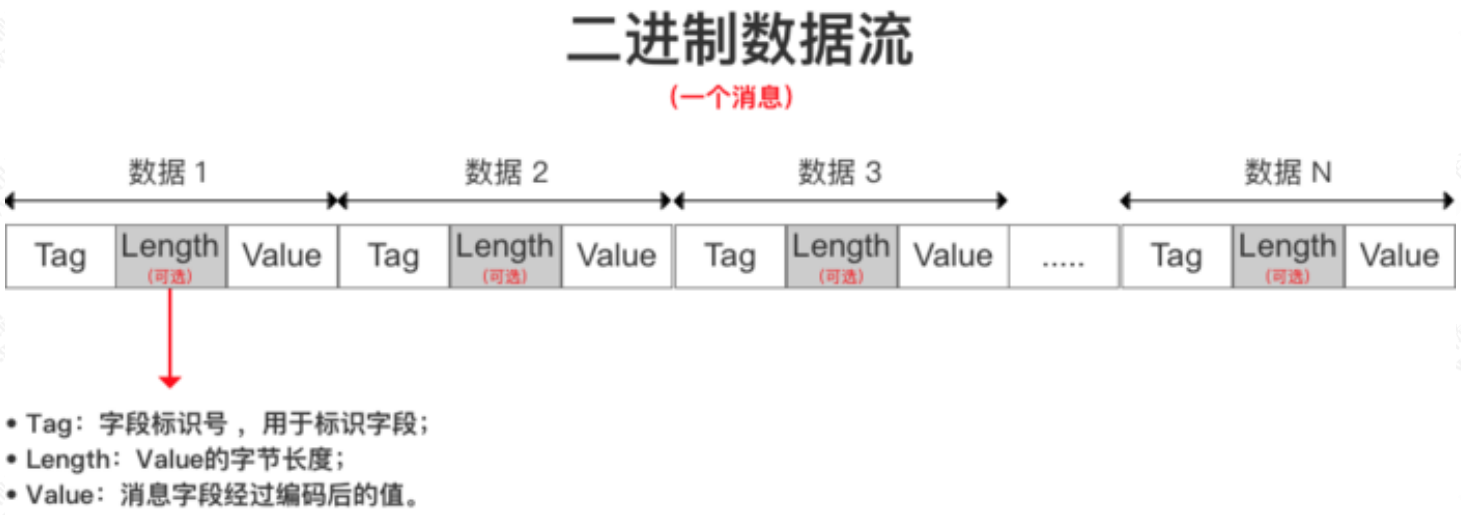
大家肯定有个疑问，这里的结果为什么直接就是 ASCII 编码的值呢？怎么没有做压缩呢？有没有同学能够回答出来

原因是，varint 是对字节码做压缩，但是如果这个数字的二进制只需要一个字节表示的时候，其实最终编码出来的结果是不会变化的

还有两个数字，3 和 16 代表什么呢？那就要了解 protobuf 的存储格式了

存储格式

protobuf 采用 T-L-V 作为存储方式



Wire Type 值	编码方式	编码长度	存储方式	代表的数据类型
0	Varint （负数时以Zigzag辅助编码）	变长（1–10个字节）	T – V	• int32, int64, uint32, uint64,bool, enum • sint32, sint64(负数时使用)
1	64–bit	固定8个字节		fixed64, sfixed64, double
2	Length–delimi	变长	T – L – V	string, bytes, embedded messages, packed repeated fields
3	Start group	已弃用	已弃用	Groups (已弃用)
4	End group			
5	32–bit	固定4个字节	T – V	fixed32, sfixed32, float

tag 的计算方式是　field_number(当前字段的编号) << 3 | wire_type

比如 Mic 的字段编号是 1 ，类型 wire_type 的值为 2　所以 ：　1 <<3 | 2 =10

age=300 的字段编号是 2，类型 wire_type 的值是 0，　所以 ：　2<<3|0 =16

第一个数字 10，代表的是 key，剩下的都是 value。

负数的存储

在计算机中，负数会被表示为很大的整数，因为计算机定义负数符号位为数字的最高位，所以如果采用 varint 编码表示一个负数，那么一定需要 5 个比特位。所以在 protobuf 中通过 sint32/sint64 类型来表示负数，负数的处理形式是先采用 zigzag 编码（把符号数转化为无符号数），在采用 varint 编码。

sint32：(n << 1) ^ (n >> 31)

sint64：(n << 1) ^ (n >> 63)

比如存储一个（-300）的值

-300

原码：0001 0010 1100

取反：1110 1101 0011

加 1 ：1110 1101 0100

n<<1: 整体左移一位，右边补 0 -> 1101 1010 1000

n>>31: 整体右移 31 位，左边补 1 -> 1111 1111 1111

n<<1 ^ n >>31

1101 1010 1000 ^ 1111 1111 1111 = 0010 0101 0111

十进制： 0010 0101 0111 = 599

varint 算法： 从右往做，选取 7 位，高位补 1/0（取决于字节数）

得到两个字节

1101 0111 0000 0100

-41 、 4

总结

Protocol Buffer 的性能好，主要体现在 序列化后的数据体积小 & 序列化速度快，最终使得传输效率高，其原因如下：

序列化速度快的原因：

a. 编码 / 解码 方式简单（只需要简单的数学运算 = 位移等等）

b. 采用 Protocol Buffer 自身的框架代码 和 编译器 共同完成

序列化后的数据量体积小（即数据压缩效果好）的原因：

a. 采用了独特的编码方式，如 Varint、Zigzag 编码方式等等

b. 采用 T - L - V 的数据存储方式：减少了分隔符的使用 & 数据存储得紧凑

序列化技术的选型

技术层面

1. 序列化空间开销，也就是序列化产生的结果大小，这个影响到传输的性能
2. 序列化过程中消耗的时长，序列化消耗时间过长影响到业务的响应时间
3. 序列化协议是否支持跨平台，跨语言。因为现在的架构更加灵活，如果存在异构系统通信需求，那么这个是必须要考虑的
4. 可扩展性/兼容性，在实际业务开发中，系统往往需要随着需求的快速迭代来实现快速更新，这就要求我们采用的序列化协议基于良好的可扩展性/兼容性，比如在现有的序列化数据结构中新增一个业务字段，不会影响到现有的服务
5. 技术的流行程度，越流行的技术意味着使用的公司多，那么很多坑都已经淌过并且得到了

解决，技术解决方案也相对成熟

6. 学习难度和易用性

选型建议

1. 对性能要求不高的场景，可以采用基于 XML 的 SOAP 协议
2. 对性能和间接性有比较高要求的场景，那么 Hessian、Protobuf、Thrift、Avro 都可以。
3. 基于前后端分离，或者独立的对外的 api 服务，选用 JSON 是比较好的，对于调试、可读性都很不错
4. Avro 设计理念偏于动态类型语言，那么这类的场景使用 Avro 是可以的

各个序列化技术的性能比较

这个地址有针对不同序列化技术进行性能比较：https://github.com/eishay/jvm-serializers/wiki