



咕泡学院 VIP 课：分布式系统的基石序列化及反序列化

课程目标

1. 了解序列化的意义
2. 如何实现一个序列化操作
3. 序列化的高阶认识
4. 常见的序列化技术及应用
5. Protobuf 实现原理分析
6. 序列化框架的选型

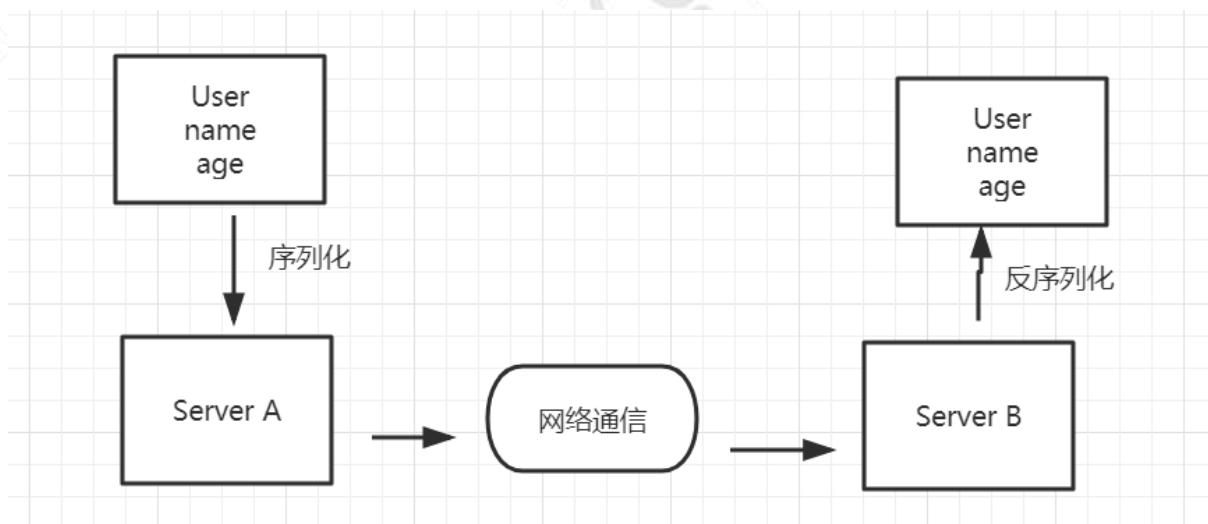
目录

目录

咕泡学院 VIP 课：分布式系统的基石序列化及反序列化.....	1
了解序列化的意义	4
序列化面临的挑战	5
如何实现一个序列化操作	5
定义接口	6
基于 JDK 序列化方式实现.....	6
具体实现	7
序列化的高阶认识	9
serialVersionUID 的作用	9
静态变量序列化.....	10
父类的序列化	10
Transient 关键字	12
绕开 transient 机制的办法	13
序列化的存储规则	14
序列化实现深克隆	14
浅克隆	15
深克隆	16
常见的序列化技术	18
XML 序列化框架	18
JSON 序列化框架	19
Hessian 序列化框架	19
Protobuf 序列化框架	19
下载 protobuf 工具	20
编写 proto 文件	20
生成实体类	21
运行查看结果	21
Protobuf 原理分析	22
varint 编码方式	22
第一步，转化为 2 进制编码	22
第二步，提取字节	22
第三步，继续提取字节	22

第四步，拼接成一个新的字节串	23
varint 压缩小数据	23
第一步，转换为 2 进制编码	23
第二步，提取字节	23
第三步，形成新的字节	24
zigzag 编码方式	24
计算机语言中如何表示负整数？	24
zigzag 原理	26
存储方式	28
总结	29
各个序列化技术的性能比较	30
序列化技术的选型	30
技术层面	30
选型建议	31

了解序列化的意义



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

序列化面临的挑战

评价一个序列化算法优劣的两个重要指标是：序列化以后的数据大小；

序列化操作本身的速度及系统资源开销（CPU、内存）；

Java 语言本身提供了对象序列化机制，也是 Java 语言本身最重要的底层机制之一，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

如何实现一个序列化操作

在 Java 中，只要一个类实现了 `java.io.Serializable` 接口，那么它就可以被序列化

定义接口

```
public interface ISerializer {

    /**
     * 序列化
     * @param obj
     * @param <T>
     * @return
     */
    <T> byte[] serialize(T obj);

    /**
     * 反序列化
     * @param data
     * @param clazz
     * @param <T>
     * @return
     */
    <T> T deserialize(byte[] data, Class<T> clazz);
}
```

基于 JDK 序列化方式实现

JDK 提供了 Java 对象的序列化方式，主要通过输出流 `java.io.ObjectOutputStream` 和对象输入流 `java.io.ObjectInputStream` 来实现。其中，被序列化的对象需要实现 `java.io.Serializable` 接口


```
public class JsonSerializer implements ISerializer{

    @Override
    public <T> byte[] serialize(T obj) {
        ByteArrayOutputStream byteArrayOutputStream=new ByteArrayOutputStream();
        try{
            ObjectOutputStream objectOutputStream=new ObjectOutputStream(byteArrayOutputStream);
            objectOutputStream.writeObject(obj);
        }catch (Exception e){
            throw new RuntimeException(e);
        }
        return byteArrayOutputStream.toByteArray();
    }

    @Override
    public <T> T deserialize(byte[] data, Class<T> clazz) {
        ByteArrayInputStream byteArrayInputStream=new ByteArrayInputStream(data);
        try{
            ObjectInput objectInput=new ObjectInputStream(byteArrayInputStream);
            return (T)objectInput.readObject();
        }catch (Exception e){
            throw new RuntimeException(e);
        }
    }
}
```

具体实现

通过对一个 user 对象进行序列化操作

```
public class User implements Serializable{

    private String name;
    private int age;

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

}

public class Demo {
    public static void main(String[] args) {
        ISerializer iSerializer=new JsonSerializer();
        User user=new User();
        user.setAge(18);
        user.setName("Mic");
        byte[] serialByte=iSerializer.serialize(user); //实现序列化

        User dUser=iSerializer.deserialize(serialByte,User.class);
        System.out.println(dUser);
    }
}
```


序列化的高阶认识

serialVersionUID 的作用

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

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

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，方法名等)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。

静态变量序列化

在 User 中添加一个全局的静态变量 num ， 在执行序列化以后修改 num 的值为 10， 然后通过反序列化以后得到的对象去输出 num 的值

```
public class User implements Serializable{

    private static final long serialVersionUID = 454107471257794587L;
    private String name;
    private int age;
    public static int num=5;

    public static void main(String[] args) {
        JsonSerializer iSerializer=new JsonSerializer();
        User user=new User();
        user.setAge(18);
        user.setName("Mic");
        iSerializer.serializeToFile(user); //实现序列化
        user.num=10;

        User duser=iSerializer.deserializeFromFile(User.class);
        System.out.println(duser+"-"+duser.num);
    }
}
```

最后的输出是 10，理论上打印的 num 是从读取的对象里获得的，应该是保存时的状态才对。之所以打印 10 的原因在于序列化时，并不保存静态变量，这其实比较容易理解，序列化保存的是对象的状态，静态变量属于类的状态，因此 序列化并不保存静态变量。

父类的序列化

一个子类实现了 Serializable 接口，它的父类都没有实现 Serializable

接口，在子类中设置父类的成员变量的值，接着序列化该子类对象。再反序列化出来以后输出父类属性的值。结果应该是什么？

```
public class SuperUser {
    String sex;

    public String getSex() {
        return sex;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }
}

public class User extends SuperUser implements Serializable{

    private static final long serialVersionUID = 454107471257794587L;
    private String name;
    private int age;

}

public class Demo {
    public static void main(String[] args) {
        ISerializer iSerializer=new JsonSerializer();
        User user=new User();
        user.setAge(18);
        user.setName("Mic");
        user.setSex("男");
        iSerializer.serializeToFile(user); //实现序列化

        User duser=iSerializer.deserializeFromFile(User.class);
        System.out.println(duser.getSex());
    }
}
```

发现父类的 sex 字段的值为 null。也就是父类没有实现序列化

结论:

1. 当一个父类没有实现序列化时，子类继承该父类并且实现了序列化。
在反序列化该子类后，是没办法获取到父类的属性值的
2. 当一个父类实现序列化，子类自动实现序列化，不需要再显示实现 `Serializable` 接口
3. 当一个对象的实例变量引用了其他对象，序列化该对象时也会把引用对象进行序列化，但是前提是该引用对象必须实现序列化接口

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

```
public class User extends SuperClass {  
  
    public static int num=5;  
  
    private String name;  
    private int age;  
  
    private transient String hobby;  
}
```



```
public static void main( String[] args ) {  
    ISerializer iSerializer=new FastjsonSerializer();  
    User user=new User();  
    user.setAge(18);  
    user.setName("Mic");  
    user.setHobby("菲菲");  
    user.setSex("男");  
}
```

绕开 transient 机制的办法

```
public class User extends SuperClass {  
  
    public static int num=5;  
  
    private String name;  
    private int age;  
  
    private transient String hobby;  
|  
    //序列化对象  
    private void writeObject(ObjectOutputStream objectOutputStream) throws IOException {  
        objectOutputStream.defaultWriteObject();  
        objectOutputStream.writeObject(hobby);  
    }  
|  
    //反序列化  
    private void readObject(ObjectInputStream objectInputStream) throws IOException, Class{  
        objectInputStream.defaultReadObject();  
        hobby=(String)objectInputStream.readObject();  
    }  
}
```

注意：有同学在课堂上问了这个问题，我觉得是个好问题，writeObject 和 readObject 这两个私有的方法，既不属于 Object、也不是 Serializable，为什么能够在序列化的时候被调用呢？原因是，ObjectOutputStream 使用了反射来寻找是否声明了这两个方法。因为 ObjectOutputStream 使用 getPrivateMethod，所以这些方法必须声明为 private 以至于供 ObjectOutputStream 来使用

序列化的存储规则

```
public class StoreRuleDemo {  
  
    public static void main(String[] args) throws IOException {  
        ObjectOutputStream out=new ObjectOutputStream(new FileOutputStream( name: "user.txt"));  
        User user=new User();  
        user.setName("mic");  
        out.writeObject(user);  
        out.flush();  
        System.out.println(new File( pathname: "user.txt").length());  
        out.writeObject(user);  
        out.close();  
        System.out.println(new File( pathname: "user.txt").length());  
    }  
}
```

同一对象两次（开始写入文件到最终关闭流这个过程算一次，上面的演示效果是不关闭流的情况才能演示出效果）写入文件，打印出写入一次对象后的存储大小和写入两次后的存储大小，第二次写入对象时文件只增加了 5 字节

Java 序列化机制为了节省磁盘空间，具有特定的存储规则，当写入文件的为同一对象时，并不会再将对象的内容进行存储，而只是再次存储一份引用，上面增加的 5 字节的存储空间就是新增引用和一些控制信息的空间。反序列化时，恢复引用关系。该存储规则极大的节省了存储空间。

序列化实现深克隆

在 Java 中存在一个 Cloneable 接口，通过实现这个接口的类都会具备 clone 的能力，同时 clone 是在内存中进行，在性能方面会比我们直接通过 new 生成对象要高一些，特别是一些大的对象的生成，性能提升相对比较明显。那么在 Java 领域中，克隆分为深度克隆和浅克隆

浅克隆

被复制对象的所有变量都含有与原来的对象相同的值，而所有的对其他对象的引用仍然指向原来的对象。

实现一个邮件通知功能，告诉每个人今天晚上的上课时间，通过浅克隆实现如下

```
public class Email {  
    private String content;  
  
    public String getContent() {  
        return content;  
    }  
  
    public void setContent(String content) {  
        this.content = content;  
    }  
}  
  
public class Person implements Cloneable {  
    private String name;  
  
    private Email email;  
  
    @Override  
    protected Person clone() throws CloneNotSupportedException {  
        return (Person) super.clone();  
    }  
}
```

```
public class CloneDemo {  
  
    public static void main(String[] args) throws CloneNotSupportedException {  
        Email email=new Email();  
        email.setContent("今天晚上20:00有课程");  
        Person p1=new Person( name: "Mic");  
        p1.setEmail(email);  
  
        Person p2=p1.clone();  
        p2.setName("黑白");  
        p2.getEmail().setContent("今天晚上是20:30上课");  
  
        System.out.println(p1.getName()+"->" +p1.getEmail().getContent());  
        System.out.println(p2.getName()+"->" +p2.getEmail().getContent());  
    }  
}
```

但是，当我们只希望，修改“黑白”的上课时间，调整为 20:30 分。通过结果发现，所有人的通知消息都发生了改变。这是因为 p2 克隆的这个对象的 Email 引用地址指向的是同一个。这就是浅克隆

深克隆

被复制对象的所有变量都含有与原来的对象相同的值，除去那些引用其他对象的变量。那些引用其他对象的变量将指向被复制过的新对象，而不再是原有的那些被引用的对象。换言之，深拷贝把要复制的对象所引用的对象都复制了一遍

```
public class Person implements Cloneable, Serializable{

    private String name;

    private Email email;

    public Person deepClone() throws IOException, ClassNotFoundException {
        ByteArrayOutputStream bos=new ByteArrayOutputStream();
        ObjectOutputStream oos=new ObjectOutputStream(bos);
        oos.writeObject(this);
        ByteArrayInputStream bis=new ByteArrayInputStream(bos.toByteArray());
        ObjectInputStream ois=new ObjectInputStream(bis);
        return (Person) ois.readObject();
    }
}
```

```
public class Email implements Serializable{
    private String content;

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }
}
```

```
public static void main(String[] args) throws CloneNotSupportedException {
    Email email=new Email();
    email.setContent("今天晚上20: 00有课程");
    Person p1=new Person( name: "Mic");
    p1.setEmail(email);

    Person p2=p1.clone();
    Person p2=p1.deepClone();
    p2.setName("黑白");
    p2.getEmail().setContent("今天晚上是20: 30上课");

    System.out.println(p1.getName()+"->" +p1.getEmail().getContent());
    System.out.println(p2.getName()+"->" +p2.getEmail().getContent());
}
```

这样就能实现深克隆效果，原理是把对象序列化输出到一个流中，然后在把对象从序列化流中读取出来，这个对象就不是原来的对象了。

常见的序列化技术

使用 JAVA 进行序列化有他的优点，也有他的缺点

优点：JAVA 语言本身提供，使用比较方便和简单

缺点：不支持跨语言处理、性能相对不是很好，序列化以后产生的数据相对较大

XML 序列化框架

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

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/fastjson>)
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

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 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

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;
```



```
}
```

proto 的语法

1. 包名
2. option 选项
3. 消息模型(消息对象、字段 (字段修饰符-required/optional/repeated))

字段类型 (基本数据类型、枚举、消息对象)、字段名、标识号)

生成实体类

在 protoc.exe 安装目录下执行如下命令

```
.\protoc.exe --java_out=./ ./user.proto
```

运行查看结果

将生成以后的 UserProto.java 拷贝到项目中

```
public static void main(String[] args) throws InvalidProtocolBufferException {
    //fluent
    UserProto.User user=
        UserProto.User.newBuilder()
            .setName("Mic").setAge(18).build();

    ByteString bytes=user.toByteString();

    System.out.println();

    UserProto.User nUser=UserProto.User.parseFrom(bytes);
    System.out.println(nUser);
    //BitMap/BitSet
}
```

Protobuf 原理分析

核心原理：protobuf 使用 varint (zigzag) 作为编码方式，使用 T-L-V 作为存储方式

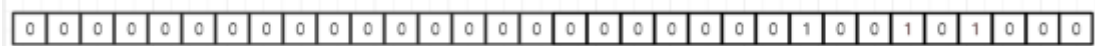
varint 编码方式

varint 是一种数据压缩算法，其核心思想是利用 bit 位来实现数据压缩。

比如：对于 int32 类型的数字，一般需要 4 个字节表示；若采用 Varint 编码，对于很小的 int32 类型数字，则可以用 1 个字节

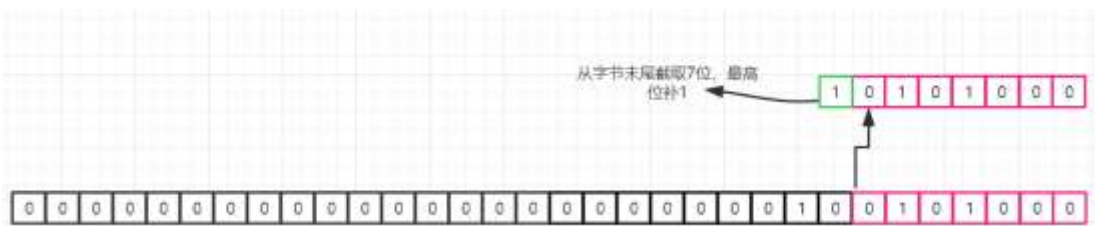
假设我们定义了一个 int32 字段值=296.

第一步，转化为 2 进制编码



第二步，提取字节

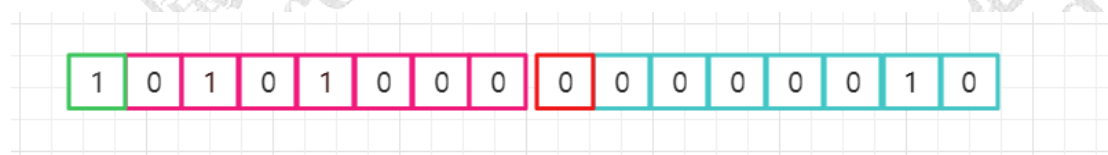
规则：按照从字节串末尾选取 7 位，并在最高位补 1，构成一个字节



第三步，继续提取字节

整体右移 7 位，继续截取 7 个比特位，并且在最高位补 0。因为这个是最后一个有意义的字节了。补 0 不影响结果

将原来用 4 个字节表示的整数，经过 varint 编码以后只需要 2 个字节了。



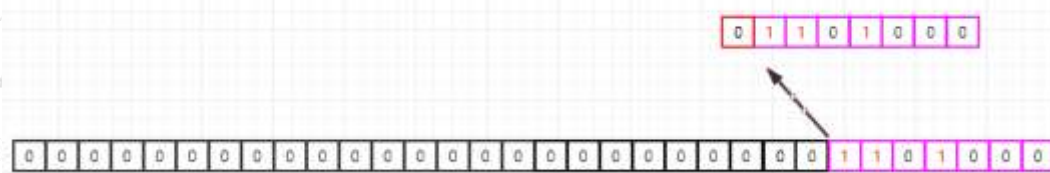
varint 编码对于小于 127 的数, 可以最大化的压缩

比如我们压缩一个 $\text{var32} = 104$ 的数据

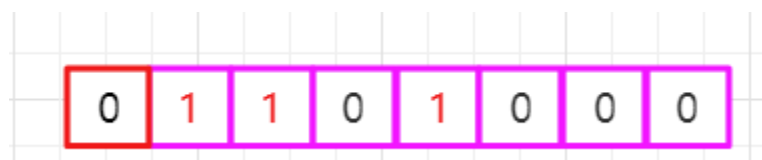
[illegible]

从末尾开始提取 7 个字节

并且在最高位最高位补 0，因为这个是最后的 7 位。



第三步，形成新的字节



也就是通过 varint 对于小于 127 以下的数字编码,只需要占用1个字节。

zigzag 编码方式

对于负数的处理，protobuf 使用 zigzag 的形式来存储。为什么负数需要用 zigzag 算法？

计算机语言中如何表示负整数？

在计算机中，定义了原码、反码和补码。来实现负数的表示。

我们以一个字节 8 个 bit 来演示这几个概念

数字 8 的二进制表示为 0000 1000

原码

通过第一个位表示符号（0 表示非负数、1 表示负数）

(+8) = {0000 1000}

(-8) = {1000 1000}

反码

因为第一位表示符号位，保持不变。剩下的位，非负数保持不变、负数按位取反。那对于上面的原码按照这个规则得到的结果

$(+8) = \{0000\ 1000\}_{\text{原}} = \{0000\ 1000\}_{\text{反}}$ 非负数，剩下的位不变。所以和原码是保持一致

$(-8) = \{1000\ 1000\}_{\text{原}} = \{1111\ 0111\}_{\text{反}}$ 负数，符号位不动，剩下为取反

但是通过原码和反码方式来表示二进制，还存在一些问题。

第一个问题：

0 这个数字，按照上面的反码计算，会存在两种表示

$(+0) = \{0000\ 0000\}_{\text{原}} = \{0000\ 0000\}_{\text{反}}$

$(-0) = \{1000\ 0000\}_{\text{原}} = \{1111\ 1111\}_{\text{反}}$

第二个问题：

符号位参与运算，会得到一个错误的结果，比如

$1 + (-1) =$

$\{0000\ 0001\}_{\text{原}} + \{1\ 0000\ 0001\}_{\text{原}} = \{1000\ 0010\}_{\text{原}} = -2$

$\{0000\ 0001\}_{\text{反}} + \{1111\ 1110\}_{\text{反}} = \{1111\ 1111\}_{\text{反}} = -0$

不管是原码计算还是反码计算。得到的结果都是错误的。所以为了解决这个问题，引入了补码的概念。

补码

补码的概念：第一位符号位保持不变，剩下的位非负数保持不变，负数按位取反且末位加 1

$$(+8) = \{0000\ 1000\}_{\text{原}} = \{0000\ 1000\}_{\text{原}} = \{0000\ 1000\}_{\text{补}}$$

$$(-8) = \{1000\ 1000\}_{\text{原}} = \{1111\ 0111\}_{\text{反}} = \{1111\ 1000\}_{\text{补}} \text{ 末位加一(补码)}$$

$$8 + (-8) = \{0000\ 1000\}_{\text{补}} + \{1111\ 1000\}_{\text{补}} = \{0000\ 0000\} = 0$$

通过补码的方式，在进行符号运算的时候，计算机就不需要关心符号的问题，统一按照这个规则来计算。就没问题没问题

zigzag 原理

有了前面这块的基础以后，我们再来了解下 zigzag 的实现原理

比如我们存储一个 $\text{int32} = -2$ 按照上面提到的负数表现形式如下

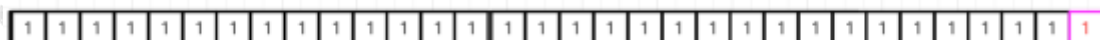
原码 $\{1\ 000\ 0010\}$ \rightarrow 取反 $\{1111\ 1101\}$ \rightarrow 整体加 1 $\{1111\ 1110\}$




zigzag 的核心思想是去掉无意义的 0，最大可能性的压缩数据。但是对于负数。第一位表示符号位，如果补码的话，前面只能补 1。就会导致陷入一个很尴尬的地步，负数似乎没办法压缩。

所以 zigzag 提供了一个方法，既然第一位是符号位，那么干脆把这个符号位放到补码的最后。整体右移。

所以上面这个 -2，将符号位移到最末尾，也就是右移 31 位。得到如下结果（对于负数形式，整体右移 31 位，把符号位移动到最后边；为什么要移动到最后呢，因为对于负数形式，补码位永远是 1，那么如果他站在最高位，就永远没办法压缩。所以做了一个移动）



但是对于上面这个操作，并不能解决压缩的问题，因为值越小，那么前导的 1 越多。所以 zigzag 算法考虑到是否能够将符号位不变，整体取反呢？

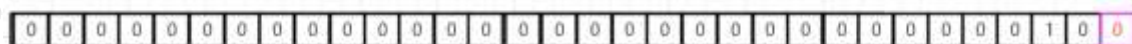


那这样就能够实现压缩的需求了？（这里如果是单纯的这么实现，是没办法实现反序列化的。）所以还需要下面这个过程。

所以对于同样 (-2) 的正数形式 (2) ，在二进制中的表现为 $\{00000010\}$

那 zigzag 算法定义了对于非负数形式，则把符号位移动到最右，其他整体往左移动一位。得到如下的效果

（对于非负数形式 2，按照整体左移 1 位，右边补零的形式来表示如下）



这样一来，对于 (2) 这个数字，正负数都有表示的方法了。那么 zigzag 结合了两种表示方法，来进行计算。计算规则是将正数形式和负数形式进行异或运算。按照上面的两种表现形式的异或运算结果是



而在 zigzag 中的计算规则是

将 -2 的二进制形式 $\{1111\ 1110\}$ 按照正数的算法，左移一位，右边补零得到 $\{11111100\}$ ，如下图左边。按照负数的形式，讲符号位移动到最右边，右移 31 位，得到下面右图。再将两者取异或算法。实现最终的压缩。



然后再将两个结果进行“异或”运算

ps:

异或运算是

0 异或 0 = 0

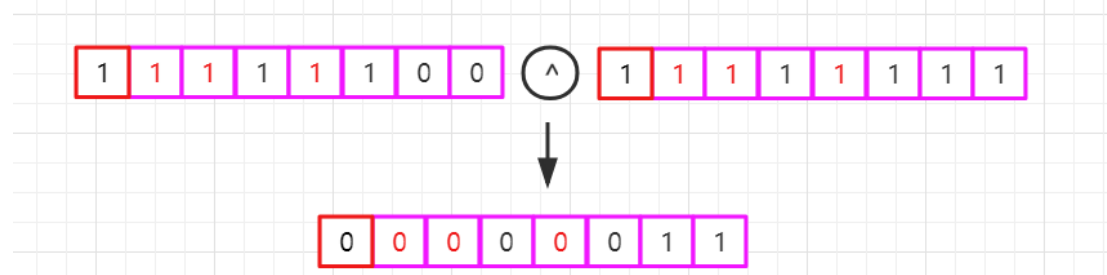
1 异或 1 = 0

1 异或 0 = 1

0 异或 1 = 1



得到



最后，-2 在的结果是 3. 占用一个比特位存储。

存储方式

经过编码以后的数据，大大减少了字段值的占用字节数，然后基于 T-L-V 的方式进行存储

二进制数据流

(一个消息)



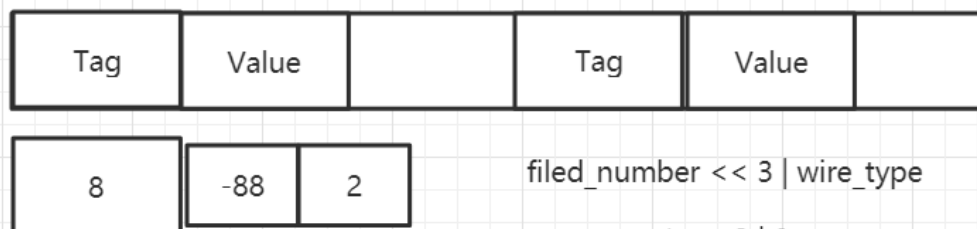
- Tag: 字段标识号，用于标识字段；
- Length: Value的字节长度；
- Value: 消息字段经过编码后的值。

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	<ul style="list-style-type: none">int32, int64, uint32, uint64, bool, enumsint32, sint64 (负数时使用)
1	64-bit	固定8个字节		fixed64, sfixed64, double
2	Length-delimited	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 的取值为 $\text{field_number}(\text{字段数}) \ll 3 \mid \text{wire_type}$

296 被 varint 编码后的字节为 10101000 00000010

296的存储结构



$\text{field_number} \ll 3 \mid \text{wire_type}$

$1 \ll 3 \mid 0$

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

各个序列化技术的性能比较

这个地址针对不同序列化技术进行性能比较：

<https://github.com/eishay/jvm-serializers/wiki>

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