链滴

序列化和反序列化浅析

作者：andot

原文链接：https://ld246.com/article/1478633580985

来源网站：链滴

简介

序列化和反序列化对于现代的程序员来说是一个既熟悉又陌生的概念。说熟悉是因为几乎每个程序员工作中都直接或间接的使用过它，说陌生是因为大多数程序员对序列化和反序列化的认识仅仅停留在较一下各种不同实现的序列化的性能上面，而很少有程序员对序列化和反序列化的设计和实现有深入研究。

本文将从序列化和反序列化的设计和实现的入手，来简单讲解一下序列化和反序列化。其中包括以下个方面：

1. 序列化和反序列化的作用
2. 什么样的数据是可序列化的
3. 序列化和反序列化的分类
4. 序列化和反序列化的类型映射

本文不会涉及到某几种语言的某几种序列化实现的性能对比之类的内容。

序列化和反序列化的作用

我们在编写程序代码时，通常会定义一些常量和变量，然后再写一堆操作它们的指令。不管是变量还常量，它们表示的都是数据。所以简单的说，一个程序就是一堆指令操作一堆数据。

但是为了更有效的管理这堆数据，现代的程序设计语言都会引入一个类型系统来对这些数据进行分类理，而不是让程序员把所有数据都一股脑的当做二进制串来进行操作。

比如一个常量可能是一个数字，一个布尔值，一个字符串，或者是一个由它们构成的数组。而变量通具有更丰富的类型可以使用。甚至你还可以自定义类型。对于面向对象的语言来说，一个类型表示的仅仅是数据本身，还包括了对这种类型数据的一组操作。

一个程序可以以源码或者可执行的二进制形式保存在磁盘（或者其它存储介质）上。当你需要执行它，它会以某种形式被载入内存，然后执行。

一个程序在执行过程中通常会生成新的数据，这些新的数据一部分是临时的，在内存中，它们转瞬即。还有一部分数据可能需要被保存下来，或者被传递到其它的地方去。在这种情况下，可能就会涉及数据的形式转换的问题。这个把程序运行时的内存数据转换为一种可保存或可传递的数据的过程，我就称它为序列化。

这些保存和传递的数据，可能会在某个时间被重新载入内存，但可能会是不同的进程，或者不同的程，甚至不同的机器上被载入，还原为内存中的具体类型的数据变量，这个从保存的数据还原为具体语具体类型的数据变量的过程，我们称它为反序列化。

什么样的数据是可序列化的

什么样的数据可序列化这是一个相对的问题，而不是一个绝对的问题。因为它会受到各种不同因素的响。

数据的可还原性

被序列化的数据应该是可还原的。可还原的意思是，一个被序列化的数据在被反序列化后仍然是有意的。注意，这里说的是有意义，而不是说被反序列化的数据应该跟序列化之前的源数据相同。为了便

理解，我们来举例说明一下。

指针数据是否可序列化

首先我们来讨论一下指针类型的数据是否可序列化。

通常我们认为指针数据是不可序列化的，因为它表示的是一个内存地址，而如果我们把这个内存地址存下来，下一次我们将这个内存地址还原到一个指针变量中时，这个内存地址所指向的位置的数据可早就不是我们所需要的数据了，甚至指向的是一个完全没有意义的数据。所以，在这种情况下，虽然后两个指针变量的值相同，但是还原之后的指针变量指向的数据已经没有意义了，我们就称它不具有还原性。

那么指针数据真的不可序列化吗？如果我们从需要反序列化的数据有意义这个角度考虑，那么我们也以做到对指针数据的序列化。

指针通常分为指向具体类型数据的指针（例如：int *, string *）和指向不明类型数据的指针（例如 void *）。

对于前者，如果我们希望序列化的数据包含指向的具体类型的数据，并且在反序列化之后，能够还原一个指向该具体类型数据的指针，且指向的数据值跟源值相同的话，那么我们其实是可以做到的。虽，还原之后的指针所指向的内存地址，跟源指针指向的内存地址可能完全不一样，但是它指向的数据有意义的，且是我们期望的，那么这种情况下，我们也可以称这个指针数据是可还原的。

对于后者，如果我们没有所指向数据的具体信息，那就没有办法对指向的数据进行保存。所以这种类的指针也就没办法进行序列化了。

另外，还有一种特殊的指针类型，它保存的并不是一个具体的内存地址，而是一个相对的偏移量，比如 intptr_t 类型就常作此用，这种时候，对它的值序列化和反序列化之后，得到的值仍然是同样的相对移量值，在这种情况下，反序列化后的数据就是有意义的，所以，这种指针数据也具有可还原性。

从上面的分析，我们可以看出，指针类型是否可序列化，取决于我们想要什么意义的反序列化数据。

资源类型是否可序列化

对于资源类型，有些语言有明确的定义，比如 PHP，而有些语言则没有明确的定义。但大致上我们可认为一个打开的文件对象，一个打开的数据库连接，一个打开的网络套接字，以及诸如此类跟外部资相关的数据类型，都可以被称作资源类型。

对于资源类型我们通常认为它们都是不可序列化的，哪怕表示该类型的结构体中的所有字段都是可序化的基本类型数据。原因是这些资源类型中保存的数据是跟当前打开的资源相关的，这些数据如果复到其它的进程，或者其它的机器中去之后，这些资源类型中保存的数据就失去了意义。

对于资源类型的一部分属性数据，比如文件名，数据库地址，网络套接字地址，它们可以在不同的进、不同的机器之间传递之后，仍然表示原有的意义。

但是通常的序列化程序是不会对资源类型做这样的序列化操作的，因为序列化程序对资源类型序列化，并不能假定用户需要的仅仅是这些信息，而且如果用户需要的真的就仅仅是这些信息的话，那用户全可以明确的只序列化这些数据，而不是对整个资源类型做序列化操作。

但是有些特殊的资源，比如内存流，文件流等。不同的序列化实现可能对待它们的方式也不同。有些列化实现认为这些资源类型同样不可序列化。而有些序列化实现则认为可以将资源本身一起序列化，如内存流中的数据会被作为序列化数据的主体进行序列化，在反序列化时，被反序列化为另外一个内流对象，虽然是两个不同的资源，但是资源中的数据是相同的。

序列化格式的限制

一个数据能否被序列化，还要看所使用的序列化格式是否支持。

对于基本类型的数据来说，几乎所有的序列化格式都支持。但是对于有些采用代码生成器方式实现的列化来说，它们可能只支持通过 IDL 生成的代码中所定义的类型的序列化，而不支持对语言内置的单原生类型数据变量的序列化，也不支持通过普通方式定义的自定义类型数据的序列化。比如 Protocol uffers 就是这样。

对于复杂类型，比如 map 这种类型，有些序列化格式只支持 Key 为字符串类型的 map 数据的序列。而不支持其它 Key 类型的 map 数据的序列化。比如 JSON 就是这样。

还有一种复杂类型数据是带有循环引用结构的数据，比如下面这个 JavaScript 代码中定义的这个数组 a：

```
var a = [];
a[0] = a;
```

它的第一个元素引用了自己，这就产生了循环引用。对于这种类型的数据，很多的序列化格式也是不持的，比如 JSON，Msgpack 都不支持这种类型数据的序列化。

但是上面所说的情况，并不是所有的序列化格式都不支持，比如 Hprose 对上面所说的所有类型都支。

以上这些限制都是序列化格式本身造成的。

序列化实现的限制

对于同一种序列化格式，即便是在同一种语言中，也可能存在着多种不同的实现，比如对于 JSON 序化来说，它的 Java 版本的实现甚至有上百种。这些不同的实现各有特色，也各有各的限制，甚至互兼容。有些实现可能仅仅支持几种特别定义的类型。有些则对语言内置的类型提供了很好的支持。

还有一些序列化格式跟特定语言有紧密的绑定关系，因此无法做到跨语言的序列化和反序列化，比如 Java 序列化，.NET 的 Binary 序列化，Go 语言的 Gob 序列化格式就只能支持特定的语言。

而且即便是这种针对特定语言的序列化也不是支持该语言的所有类型。比如：Java 序列化对于 class 型只支持实现了 java.io.Serializable 接口的类型；.NET Binary 序列化则只支持标记了 System.SerializableAttribute 属性的类型。

所以，我们不能想当然的认为，一个数据支持某一种序列化，就一定支持其它类型的序列化。这种假是不成立的。

序列化和反序列化的分类

序列化和反序列化的格式多种多样，它们之间的主要区别可以大致分这样几类：

按照可读性分类

首先从可读性角度，大致可分为文本序列化和二进制序列化两种，但是也有一些序列化格式介于两者间，我们将它们暂称为半文本序列化。

文本序列化

XML 和 JSON 是大家最常见的两种文本序列化格式。

文本序列化的数据都是使用人类可读的字符表示的，就像大部分编程语言一样。而且允许包含多余的白，以增加可读性。当然也可以表示为紧凑编码形式，以利于减少存储空间和传输流量。

文本序列化除了可读性还具有可编辑性，因此，文本序列化格式也经常被用于作为配置文件的存储格。这样，使用普通的文本编辑器就可以方便的编辑这种配置文件。

文本序列化在表示数字时，通常采用人类可读的十进制数（包括小数和科学计数法）的字符串形式，除了具有可读性以外，还有另外一个好处，就是可以方便的表示大整数或者高精度小数。

二进制序列化

二进制序列化的的数据不具有可读性，但是通常比文本序列化格式更加紧凑，而且在解析速度上也更优势，当然实际的解析速度还跟具体实现有很大的关系，所以这也不是绝对的。

因为它们本身不具有可读性，所以在实际使用时，如果要想查看这些数据，就需要借助一些工具将它解析为可读信息之后才能使用。在这方面，它们相对于文本序列化具有明显的劣势。

二进制序列化表示数字时，通常会使用定长或者变长的二进制编码方式，这虽然有利于更快的编码和析编程语言中的基本数字类型，但是却不能表示大整数和高精度小数。

Protocol Buffers，Msgpack，BSON，Hessian 等格式是二进制序列化格式的代表。

半文本序列化

半文本序列化格式通常兼具文本序列化的可读性和二进制序列化的性能。

半文本序列化的数据也使用人类可读的字符表示，具有一定的可读性，但是半文本序列化是空白敏感，因此它们不能像文本序列化那样在序列化数据中添加空白。

半文本序列化格式采用紧凑编码形式，而且通常会采用跟二进制编码类似的TLV（Type-Length-Valu）编码方式，因此具有比文本序列化更高效的解析速度，当然实际解析效率也跟具体实现有关。

半文本序列化格式中对原本的二进制字符串数据仍然按照二进制字符串的格式保存，而不会像文本序化格式一样，需要将它们转换为 Base64 格式的文本。对于二进制字符串来说，不管是转为 Base64 式的文本还是原本的样子，都不具有可读性，因此，直接以原格式保存，并不损失可读性，但是却可增加解析效率。

半文本序列化格式在表示字符串时不会像文本序列化那样在字符串中间增加转义字符，或者将原本的符用转义符号表示，因此，半本文序列化格式中的字符串反而比文本序列化的字符串具有更好的可读。

半文本序列化格式在数字编码上具有跟文本序列化格式一样的特点。

Hprose，PHP 序列化格式是半文本序列化的代表。

按照自描述性分类

自描述序列化

如果序列化数据中包含有数据类型的元信息，或者数据的表示形式同时可以反映出它的类型，那么这

序列化格式就是自描述的。自描述的序列化格式，可以在不借助外部描述的情况下，进行解析。

文本序列化和半文本序列化基本上都是自描述的。二进制序列化格式中，大部分也是自描述的。

自描述序列化格式不依赖外描述文件是它的优势，在一些应用场景下，这具有不可替代的优越性。但因为包含了元信息，导致它的数据大小通常要比非自描述序列化的数据大一些。

像 XML，JSON，Hprose，Hessian，Msgpack 都是自描述类型的序列化格式。

非自描述序列化

非自描述序列化的数据在体积上更小，但是因为舍弃了自描述性，使得这种序列化数据在离开外部描之后，就无法再被使用。

Protocol Buffers 是典型的非自描述类型的序列化格式的代表。

按照实现方式分类

序列化和反序列化的很大一部分特征是由它们的实现决定的。关于序列化通常是使用代码生成或者反的方式来实现，而对于反序列化除了这两种方式之外，还有将序列化数据解析为语法树的方式，这种式实际上并不算反序列化，但通常可以更快的查找和获取文本序列化数据中某个节点的值。

基于代码生成器实现的序列化

采用代码生成方式实现序列化的好处是可以不依赖编程语言本身运行时中的元数据信息，这样即使某语言（比如 C/C++）的运行时中本身没有包含足够的元数据时，也可以方便的进行序列化和反序列。

采用代码生成方式实现序列化的另一个好处是，因为不使用反射，序列化和反序列化的速度通常会比于反射实现的序列化反序列化更快一些。

但是采用代码生成方式实现的序列化的缺点也很明显，比如对支持的数据类型限制比较严格，使用起比较麻烦，需要编写 IDL 文件，在类型映射上比较死板，通常只能实现 1-1 的映射（这个我们后面再），类型升级时，会产生兼容性问题等等。

基于反射实现的序列化

基于动态反射来实现序列化和反序列化可以做到更好的类型支持，比如语言的内置类型和普通方式编的自定义类型的数据都可以被序列化和反序列化，而且无需编写 IDL 文件就可以实现动态序列化，类映射也更加灵活，可以实现 n-m 的映射，类型升级时，可以避免产生兼容性问题。

但通常基于反射实现的序列化和反序列化的速度要比采用代码生成方式的序列化和反序列化要慢一些但是这也不是绝对的，因为在实现中，可通过一些其它的手段来提升性能。

例如采用缓存的方式，对于那些需要反射才能获得的元信息进行缓存，这样在获取元信息时可以避免射而直接使用缓存的元信息来加快序列化速度。还可以使用动态的字节码生成方式，比如在 Java 中用 ASM 技术来动态生成序列化和反序列化的代码，在 .NET 中使用 Emit 技术也可以实现同样的功能而对于 C、C++、Rust 等语言可以采用宏和模板的方式在编译期生成具体类型的序列化和反序列化代码，对于 D、Nim 等语言则可以采用编译期反射和编译期代码执行功能在编译期动态生成具体类的序列化和反序列化代码，通过这些手段，既可以获得传统的代码生成器方式的序列化和反序列化的能，又可以避免代码生成器的缺陷。

例如 Hprose for .NET 就采用上面提到的元数据缓存 + Emit 动态代码生成的优化手段，使得它的序化和反序列化速度远远超过 Protocol Buffers 的速度。

按照跨语言能力分类

并不是所有的序列化格式都是跨语言的。即使是跨语言的序列化格式，在跨语言的能力上也有所不同。

特定语言专有的序列化

大部分语言内置的序列化格式都属于特定语言专有的序列化。例如 Java 的序列化，.NET 的 Binary 列化，Go 的 Gob 序列化都属于这一种。

但也有特例，比如 PHP 序列化，原本是 PHP 语言专有的序列化格式，但因为它的格式比较简单，因也有一些其它语言上的 PHP 序列化的第三方实现。但终究 PHP 序列化格式跟 PHP 语言的关系更加密，所以在其他语言中使用 PHP 序列化时相对于其它跨语言的序列化格式或多或少的会有一些不方的地方。

跨语言的序列化

文本序列化格式往往具有更好的跨语言特征。比如 XML，JSON 等序列化格式，对于不同的语言都有多的实现来支持。

还有一些半文本或二进制序列化格式也是为跨语言而设计的，比如 Hprose，Protocol Buffers，Msgack 等，它们也具有很好的跨语言能力。

但多数二进制序列化格式在跨语言方面有很多限制。

序列化和反序列化的类型映射

如果编程语言中的数据类型跟序列化格式中的数据类型有且只有唯一的映射关系，我们就把这种类型射关系称为 1-1 映射。

如果在序列化时，编程语言中的多种数据类型被映射为一种序列化格式的类型，并且在反序列化时，种序列化类型可以被反序列化为编程语言中的多种类型，那么这种类型映射关系称为 n-m 映射。

当然还存在其它的情况，比如多种序列化类型被反序列化为编程语言中的同一种类型，再比如编程语中的所有类型跟序列化类型中的某个类型都不存在映射关系，等等。这些其它情况，我们也把它们归到 1-1 映射中。

1-1 映射还是 n-m 映射，除了跟序列化格式有关以外，还跟具体的语言实现有很大的关系。

1-1 映射

语言内置的序列化和反序列化实现一般都是 1-1 映射。这可以保证序列化之前的数据跟反序列化之后数据在类型上的完全一致性。但也由于语言内置类型的丰富性和 1-1 映射的一致性，导致这些语言内的序列化格式几乎无法做到跨语言实现。

我们前面也谈到过一个特例，那就是 PHP 序列化，PHP 序列化之所以能够做到跨语言实现，是因为本身的内置类型非常有限，以至于即使在 PHP 中是 1-1 映射的数据类型还不如其它一些跨语言的序化支持的数据类型更丰富。

而 JSON 格式，如果把它放到 JavaScript 中，它也是 1-1 映射的。而 JSON 序列化在其它语言中的现则是多种多样，有的仅支持 1-1 映射，有的则支持 n-m 映射，即便是同一种语言的不同实现也是此。

1-1 映射最麻烦的问题是，要么支持的类型不够丰富，要么跨语言方面难以实现。

第一个问题对于本来类型就不是很多的脚本语言来说通常不是问题，但对于 Java，C# 之类的语言来，这就是个问题了。

n-m 映射

n-m 映射可以很好的解决这个问题。

比如序列化格式中不需要为 Array，List，Tuple，Set 定义不同的类型，而只需要一种通用的列表类，之后就可以将某种具体语言的 Array，List，Tuple，Set 等具有列表特征的数据都映射为这一种列类型，在反序列化的时候，则直接反序列化为某种指定的类型。

这样做还有一个额外的好处：当你希望类型一致的时候，你就可以实现类型一致，而当你不希望使用致的类型时，可以直接在序列化和反序列化的过程中进行类型的转换。而不需要得到了一致的类型之，再去自己手动转换为另一种类型。

通过反射方式来实现的序列化和反序列化可以更方便的实现 n-m 映射。而通过代码生成器方式实现序列化和反序列化则通常只能实现 1-1 映射。因此，通过反射方式来实现的序列化和反序列化具有更的灵活性。