序列化和反序列化

为什么要序列化

内存中的map、slice、array以及各种对象，如何保存到一个文件中？ 如果是自己定义的结构体的实例，如何保存到一个文件中?

如何从文件中读取数据，并让它们在内存中再次恢复成自己对应的类型的实例?

要设计一套**协议**，按照某种规则，把内存中数据保存到文件中。文件是一个字节序列，所以必须把数据转换成字节序列，输出到文件。这就是**序列化。** 反之，从文件的字节序列恢复到内存并且还是**原来的**类型，就是**反序列化。**

定义

serialization 序列化：将内存中对象存储下来，把它变成一个个字节。转为 二进制 数据

deserialization 反序列化：将文件的一个个字节恢复成内存中对象。从 二进制 数据中恢复

序列化保存到文件就是持久化。

可以将数据序列化后持久化，或者网络传输；也可以将从文件中或者网络接收到的字节序列反序列化。

我们可以把数据和二进制序列之间的相互转换称为二进制序列化、反序列化，把数据和字符序列之间的相互转换称为字符序列化、反序列化。

字符序列化：JSON、XML等

二进制序列化：Protocol Buffers、MessagePack等

JSON

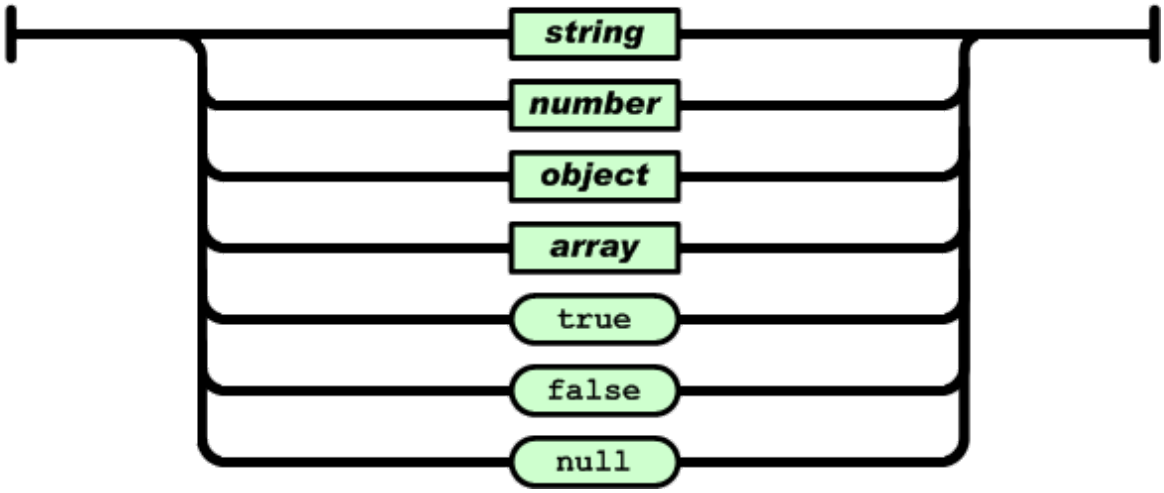
JSON(JavaScript Object Notation, JS 对象标记) 是一种轻量级的数据交换格式。它基于1999年发布的ES3 （ECMAScript是w3c组织制定的JavaScript规范）的一个子集，采用完全独立于编程语言的**文本**格式来存储和表示数据。应该说，目前JSON得到几乎所有浏览器的支持。参看 http://json.org/

JSON的数据类型

值

双引号引起来的字符串、数值、true和false、null、对象、数组，这些都是值

value



字符串

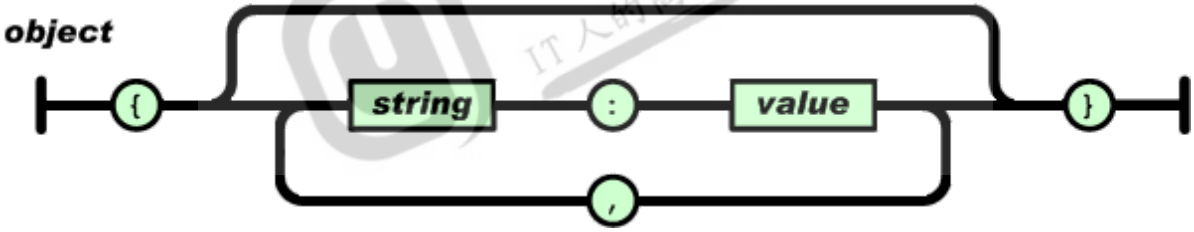
由**双引号**包围起来的任意字符的组合，可以有转义字符。

数值

有正负，有整数、浮点数。

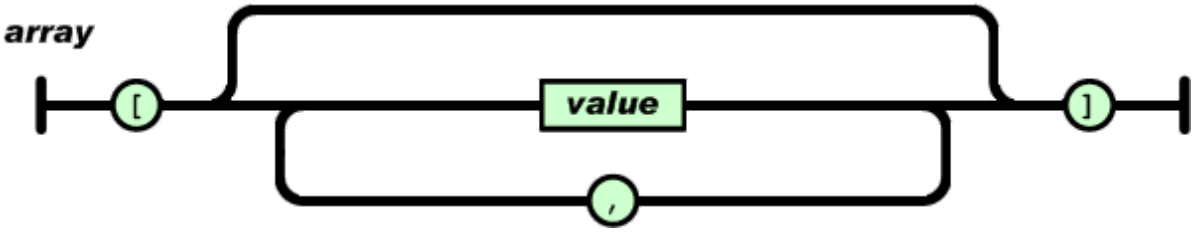
对象

无序的键值对的集合 格式: {key1:value1, ... ,keyn:valulen} **key必须是一个字符串**，需要双引号包围这个字符串。 value可以是任意合法的值。



数组

有序的值的集合 格式：[val1,...,valn]



实例

```
{
  "person": [
    {
      "name": "tom",
      "age": 18
    },
    {
      "name": "jerry",
      "age": 16
    }
  ],
  "total": 2
}
```

特别注意：JSON是字符串，是文本。JavaScript引擎可以将这种字符串解析为某类型的数据。

json包

Go标准库中提供了`encoding/json`包，内部使用了反射技术，效率较为低下。参看 https://go.dev/blog/json

- json.Marshal(v any) ([]byte, error)，将v序列化成字符序列（本质上也是字节序列），这个过程称为Encode
- json.Unmarshal(data []byte, v any) error，将字符序列data反序列化为v，这个过程称为Decode

基本类型序列化

```go
package main

import (
    "encoding/json"
    "fmt"
)

func main() {
    // 序列化
    var data = []any{
        100, 20.5, true, false, nil, "aabb", // 基本类型
        [3]int{97, 98, 99}, // Go array => js array
        []int{65, 66, 67},  // Go slice => js array
        map[string]int{"abc": 49, "aa": 50}, // Go map => js object
    }
    var target = make([][]byte, 0, len(data))
    for i, v := range data { // 一个一个单独序列化，看变化
        b, err := json.Marshal(v)
        if err != nil {
            continue
        }
        fmt.Printf("%d %T: %[2]v => %T %[3]v %s\n", i, v, b, string(b))
        target = append(target, b)
    }
    // fmt.Println(target)
    // 问题，json.Marshal(data)可以吗？

```

```
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")
    // 反序列化
    for i, v := range target { // 一个一个单独反序列化，看变化
        var t any
        err := json.Unmarshal(v, &t)
        if err != nil {
            continue
        }
        fmt.Printf("%d %T: %[2]v => %T %[3]v\n", i, v, t)
    }
}
```

运行结果如下

```
0 int: 100 => []uint8 [49 48 48] 100
1 float64: 20.5 => []uint8 [50 48 46 53] 20.5
2 bool: true => []uint8 [116 114 117 101] true
3 bool: false => []uint8 [102 97 108 115 101] false
4 <nil>: <nil> => []uint8 [110 117 108 108] null
5 string: aabb => []uint8 [34 97 97 98 98 34] "aabb"
6 [3]int: [97 98 99] => []uint8 [91 57 55 44 57 56 44 57 57 93] [97,98,99]
7 []int: [65 66 67] => []uint8 [91 54 53 44 54 54 44 54 55 93] [65,66,67]
8 map[string]int: map[aa:50 abc:49] => []uint8 [123 34 97 97 34 58 53 48 44 34 97 98 99 34 58 52 57 125], {"aa":50,"abc":49}
以上是序列化结果，说明各种类型数据被序列化成了字节序列，也可以说转换成了字符串。转换到这里就行了，下面的事是把字符串交给JavaScript引擎。
**特别注意**，转换的结果都是字符串，但是这些字符串一旦交给JavaScript引擎，它能把它们转换成对应的数据类型。

~~~~~~~~~~~~~~~~~~~~~~~~~~

0 []uint8: [49 48 48] => float64 100
1 []uint8: [50 48 46 53] => float64 20.5
2 []uint8: [116 114 117 101] => bool true
3 []uint8: [102 97 108 115 101] => bool false
4 []uint8: [110 117 108 108] => <nil> <nil>
5 []uint8: [34 97 97 98 98 34] => string aabb
6 []uint8: [91 57 55 44 57 56 44 57 57 93] => []interface {} [97 98 99]
7 []uint8: [91 54 53 44 54 54 44 54 55 93] => []interface {} [65 66 67]
8 []uint8: [123 34 97 97 34 58 53 48 44 34 97 98 99 34 58 52 57 125] => map[string]interface {} map[aa:50 abc:49]
以上是反序列化结果，从字符串（字节序列）反序列化为Go的某类型数据。因为从浏览器发来的数据都是字符串
需要注意的是，JSON字符串中，数值被转换成了Go的float64类型；true、false转成了bool型；null转成了nil；字符串转成了string；数组转成了[]interface{}
```

结构体序列化

```
package main

import (
    "encoding/json"
    "fmt"
)
```

```go

type Person struct {
    Name string
    Age  int
}

func main() {
    // 序列化
    var data = Person{
        Name: "Tome",
        Age:  20,
    }
    b, err := json.Marshal(data)
    if err != nil {
        panic(err)
    }
    fmt.Printf("%+v\n", data)             // 这是Person的实例
    fmt.Printf("%v, %s\n", b, string(b)) // 这是字符串啦
    // 反序列化
    var b1 = []byte(`{"Name": "Tom", "Age": 20}`) // 字符串，增加了些空格，js中的对象也就是键值对
    var p Person                                  // 知道目标的类型
    err = json.Unmarshal(b1, &p)                  // 填充成功，通过指针填充结构体
    if err != nil {
        panic(err)
    }
    fmt.Printf("%T %+[1]v\n", p)

    // 不知道类型
    var i interface{}
    err = json.Unmarshal(b1, &i)
    if err != nil {
        panic(err)
    }
    fmt.Printf("%T %+[1]v\n", i) // 不知道类型，只能理解为键值对
    // map[string]any map[Age:20 Name:Tome]
}
```

切片序列化

```go
package main

import (
    "encoding/json"
    "fmt"
)

type Person struct {
    Name string
    Age  int
}

func main() {
    // 序列化
```

```go
    var data = []Person{
        {Name: "AAA", Age: 20},
        {Name: "aaa", Age: 32},
    }
    b, err := json.Marshal(data)
    if err != nil {
        panic(err)
    }
    fmt.Println(b, string(b)) // 请问序列化后的字符串中，还有类型吗？有什么？
    // 反序列化
    // 不知道类型
    var i interface{}
    err = json.Unmarshal(b, &i)
    if err != nil {
        panic(err)
    }
    fmt.Printf("%T: %+[1]v\n", i)
    // i类型为[]interface{}，值为[map[Age:20 Name:AAA] map[Age:32 Name:aaa]]
    // 知道目标类型
    var b1 = []byte(`[{"name":"AAA","Age":20},{"name":"aaa","Age":32}]`)
    var j []Person
    err = json.Unmarshal(b1, &j)
    if err != nil {
        panic(err)
    }
    fmt.Printf("%T: %+[1]v\n", j)
    // j类型为[]Person，值为[{Name:AAA Age:20} {Name:aaa Age:32}]
}
```

字段标签

结构体的字段可以增加标签tag，序列化、反序列化时使用

- 在字段类型后，可以跟反引号引起来的一个标签，用json为key，value用双引号引起来写，key与value直接使用冒号，这个标签中**不要加入多余空格**，否则语法错误
  - Name string `json:"name"`，这个例子序列化后得到的属性名为name
    - json表示json库使用
    - 双引号内第一个参数用来指定字段转换使用的名称，多个参数使用逗号隔开
  - Name string `json:"name,omitempty"`，omitempty为序列化时忽略空值，也就是该字段不序列化
    - 空值为false、0、空数组、空切片、空map、空串、nil空指针、nil接口值
    - 空数组、空切片、空串、空map，长度len为0，也就是容器没有元素
- 如果使用`-`，该字段将被忽略
  - Name string `json:"-"`，序列化后没有该字段，反序列化也不会转换该字段
  - Name string `json:"-,"`，序列化后该字段显示但名为`"-"`，反序列化也会转换该字段
- 多标签使用空格间隔
  - Name string `json:"name,omitempty" msgpack:"myname"`

JSON序列化的Go实现效率较低，由此社区和某些公司提供大量开源的实现，例如easyjson、jsoniter、sonic等。对于各个Json序列化包的性能对比这里不列出来了，有兴趣的同学自己查看。基本使用方式兼容官方实现。

MessagePack

MessagePack是一个基于**二进制**高效的对象序列化类库，可用于跨语言通信。 它可以像JSON那样，在许多种语言之间交换结构对象。但是它比JSON更快速也更轻巧。 支持Python、Ruby、Java、C/C++、Go等众多语言。宣称比Google Protocol Buffers还要快4倍。

https://msgpack.org/

文档 https://msgpack.uptrace.dev/

安装

```
go get github.com/vmihailenco/msgpack/v5
```

基本使用方法和json包类似

```
package main

import (
    "fmt"

    "github.com/vmihailenco/msgpack/v5"
)

type Person struct {
    Name string `json:"name" msgpack:"myname"`
    Age  int    `json:"age" msgpack:"myage"`
}

func main() {
    // 序列化
    var data = []Person{
        {Name: "Tom", Age: 16},
        {Name: "Jerry", Age: 32},
    }
    b, err := msgpack.Marshal(data) // 方法都和json兼容
    if err != nil {
        panic(err)
    }
    fmt.Println(b, len(b), string(b)) // 二进制
    // 反序列化
    // 知道目标类型
    var j []Person
    err = msgpack.Unmarshal(b, &j)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("%T: %+[1]v\n", j)
}
```

Base64编码

索引	对应字符	索引	对应字符	索引	对应字符	索引	对应字符
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w		
15	P	32	g	49	x		
16	Q	33	h	50	y		

参考 https://en.wikipedia.org/wiki/Base64

简单讲

- 编码过程就是对3个任意字节数据编程4个字节，每个字节的最高2位不用了，只用6位，而6位的变化只有64种，如上图，利用上图查表对应就得出编码了。字节不够3会补齐
- 解码过程是对4个字节的base64编码的数据的每个字节去掉最高2位然后合并为3个字节

主要应用在JWT、网页图片传输等。

参考 https://pkg.go.dev/encoding/base64

```go
package main

import (
    "encoding/base64"
    "fmt"
)

func main() {
    var a = "abc"                                   // 3字节，试试abcd，abcde，abcdef呢？
    s := base64.StdEncoding.EncodeToString([]byte(a)) // 返回字符串
    fmt.Println(len(s), s)                          // 几个字节？
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
    b, err := base64.StdEncoding.DecodeString(s)
    if err != nil {
        fmt.Println(err)
        return
```

```
    }
    fmt.Println(len(b), b, string(b)) // 几个字节？
}
```