```
// Output Marshal:
{”ID”:1,”Name”:”Bruce”}

// Output Unmarshal:
{ID:1 Name:Bruce age:0}
```

这里 `age` 被设为了私有变量，于是序列化后的 JSON 串中没有 `age` 这个字段了。同理，从一个 JSON 字符串反序列化为 `Person` 后，也无法正确读取到 `age` 的值。

原因也很简单，如果我们深入 `Marshal` 的源码就能发现，它的底层实际上使用了反射对结构体对象进行动态解析：

```
// .../src/encoding/json/encode.go

func (e *encodeState) marshal(v any, opts encOpts) (err error) {
// ...skip
e.reflectValue(reflect.ValueOf(v), opts)
return nil
}
```

而 Golang 从语言设计的层面上禁止反射访问结构体的私有成员，所以这种反射解析自然是失败的，反序列化同理。

2.2 少用 map

前文里提到，JSON 不仅能操作结构体，还能操作 slice、map 等类型的数据。slice 比较特殊，但 map 和结构体表现在 JSON 格式下其实是一样的：

```
```

```
{
”ID”: 1,
”Name”: ”Bruce”
}
```

这种情况下，除非有特情况或需求，否则，少用 map。因为 map 会带来额外的开销，额外的代码量，以及额外的维护成本。

为什么?

首先，像上面的 Person 例子，由于 ID 和 Name 是不同类型，因此我们如果要用 map 反序列化这个 JSON 数据，就只能申明一个 `map[string]any` 类型的 map。`any`，也就是 `interface{}` ，就意味着我们如果要单独使用 Name 或 ID 时，需要用类型断言来转换类型：

```
var m map[string]any
// ...反序列化 JSON 数据，代码忽略...
// 获取成员
name, ok := m[”Name”].(string)
```

类型断言本身就是一个额外的步骤，为防止 panic，我们还需要判断第二个参数 ok，这无疑增加了开发工作量以及代码负担。

另外，map 本身对数据就是无约束的。结构体中我们能够预先定义各成员字段以及类型，但 map 不行。这就意味着，我们只能通过文档或注释或代码本身来理解这个 map 里到底装了些什么东西。并且，结构体可以限制 JSON 数据的 key 和 value 类型不被乱改，而 map 同样无法约束 JSON 的变更，只能通过业务逻辑代码来检测。这其中的工作量和后期维护成本，想想就知道会有多少。

之所以我会提及这个坑，是因为我在使用 Go 开发之前，主语言是 Python。而 Python 嘛，你们懂的，没有结构体，只有 dict（map）来加载 JSON 数据。在我刚接触 Go 时，我也习惯性用 map 来与 JSON 交互。但因为 Go 是静态类型，必须要显式转换类型（类型断言），不能像 Python 一样直接用，就一度让我很头疼。

总之，少用，或尽量不要用 map 来操作 JSON。

2.3 小心结构体组合

Go 虽然面向对象，但没有 `class` ，只有结构体，并且结构体没有继承。因此 Go 采用了一种组合的方式来复用不同的结构体。很多时候，这种组合给我们带来了极大的便利，我们可以像操作结构体自己的成员一样去操作组合的其他结构体成员，就像这样：

```
type Person struct {
ID    uint
Name string
address
}

type address struct {
Code    int
Street string
}

func (a address) PrintAddr() {
fmt.Println(a.Code, a.Street)
}

func Group() {
p := Person{
ID:    1,
Name: ”Bruce”,
address: address{
Code:    100,
Street: ”Main St”,
},
}
// 用 p 直接访问 Address 的成员和方法
fmt.Println(p.Code, p.Street)
p.PrintAddr()
}

// Output
100 Main St
100 Main St
```

```

很方便对吧，我也这么觉得。但当我们将组合融入到 JSON 的使用当中时，这里会有一个小坑需要注意。来看下面这段代码：

 me:Jim ChildrenCnt:0}

```

我们在 `Person` 结构体中添加了一个 `ChildrenCnt` 字段，用于统计该人物的子女数量。由于零值的存在，当 `p` 加载的 JSON 数据里没有 `ChildrenCnt` 数据时，该字段被赋予 0。此时就产生了误解：**我们无法将这种数据缺失的对象，与子女数确实为 0 的对象区分开**。如例子里的 Bruce 和 Jim，一个是数据缺失导致的子女数为 0，另一个是本来就为 0。而实际上 Bruce 的子女数量应该是“未知“，我们如果真当作 0 处理，在业务上可能就会产生问题。

这样的混淆在一些对数据要求严格的场景下是非常致命的。那么有没有什么办法能避免这种零值的干扰？还真有，就是上一节最后遗留的指针的使用场景。

我们把 `Person` 的 `ChildrenCnt` 类型改为 `*int` ，看看会发生什么：

```
type Person struct {
Name        string
ChildrenCnt *int
}

// Output
{Name:Bruce ChildrenCnt:<nil>}
{Name:Jim ChildrenCnt:0xc0000124c8}
```

区别产生了。Bruce 没有数据，所以 `ChildrenCnt` 是个 nil，而 Jim 则是一个非空指针。此时就能明确地知晓，Bruce 的子女数量是未知了。

本质上这种方式还是利用了零值，指针的零值。这也算是用魔法打败魔法吧（大笑）。

2.7 标签的坑

终于讲到了标签。标签也是 Golang 中一个非常重要的特性，并且常与 JSON 相伴。而且其实用过 Go 标签的读者们应该知道，标签其实是一个非常灵活、好用的东西。那这样的好特性，在使用上会有什么坑要注意呢?

一个是名称问题。Tag 可以指定 JSON 数据中字段的名称显示，这点很灵活且实用，但它同时也容易出错，并且一定程度上对程序员本身增加了一些职业素养的要求。

譬如某个程序员有意或无意地定义了这么一个结构体：

```
type PersonWrong struct {
FirstName string `json:”last_name”`
LastName  string `json:”first_name”`
}
```

Tag 对调了 FirstName 和 LastName。遇到这样的代码你会不会想把这个程序员打一顿? 别说，我还真在生产环境的代码中遇到过类似的。当然那次是无意的，属于某次代码变更时的失误。然而真遇到这种情况的时候，这样的 bug 通常也不太容易定位。主要是因为，这谁特么能想到?

反正各位读者千万别这么干，写的时候还是得多加留意。

另一个问题则与 `omitempty` + 零值的组合有关，看代码：

```
type Person struct {
Name        string `json:”person_name”`
ChildrenCnt int    `json:”cnt,omitempty”`
}

func TagMarshal() {
p := Person{
Name:        ”Bruce”,
ChildrenCnt: 0,
}
```

```
output, _ := json.MarshalIndent(p, "", "  ")
println(string(output))
}

// Output
{
  "person_name": "Bruce"
}
```

看出问题了么？我们在新建结构体对象 `p` 时，为 `ChildrenCnt` 赋值为 0。而因为 `omitempty` 标签的存在，**它使得 JSON 被序列化或反序列化时，忽略空（empty）值。在序列化时的表现就是，输出的 JSON 数据里不包含 `ChildrenCnt`，看上去就像是没有这个数据。什么是空值？对了，就是零值**。

于是熟悉的混淆又产生了：Bruce 的子女数量为 0，并非没有数据。而输的 JSON 则表示 Bruce 的子女数据不存在。

反序列化存在同样的问题，就不举例了。

这种 `omitempty` 的问题又该怎么解决呢？由于本质上还是零值惹得祸，所以，用指针。

3 总结

本文列举了 7 个使用 `encoding/json` 库时容易犯的错，这些问题我自己在工作中基本上都遇到过。如果你还没有遭遇过它们，那么恭喜你！同时也提醒你今后要小心对待 JSON；如果你也遇到过这些问题，并且为其感到困惑，希望这篇文章能够帮助到你。

本人技术有限，文章若有任何错误或不清晰的地方，还请各位不吝之处，感谢！