问题

```go
type Part1 struct {
    a bool
    b int32
    c int8
    d int64
    e byte
}
```

在开始之前，希望你计算一下 `Part1` 共占用的大小是多少呢?

```go
func main() {
    fmt.Printf("bool size: %d\n", unsafe.Sizeof(bool(true)))
    fmt.Printf("int32 size: %d\n", unsafe.Sizeof(int32(0)))
    fmt.Printf("int8 size: %d\n", unsafe.Sizeof(int8(0)))
```

```
    fmt.Printf("int64 size: %d\n", unsafe.Sizeof(int64(0)))
    fmt.Printf("byte size: %d\n", unsafe.Sizeof(byte(0)))
    fmt.Printf("string size: %d\n", unsafe.Sizeof("EDDYCJY"))
}
输出结果：
bool size: 1
int32 size: 4
int8 size: 1
int64 size: 8
byte size: 1
string size: 16
```

这么一算，`Part1` 这一个结构体的占用内存大小为 1+4+1+8+1 = 15 个字节。相信有的小伙伴是这么算的，看上去也没什么毛病
真实情况是怎么样的呢？我们实际调用看看，如下：

```
type Part1 struct {
    a bool
    b int32
    c int8
    d int64
    e byte
}

func main() {
    part1 := Part1{}

    fmt.Printf("part1 size: %d, align: %d\n", unsafe.Sizeof(part1), unsafe.Alignof(part1
}
```

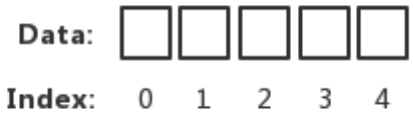
输出结果：

```
part1 size: 32, align: 8
```

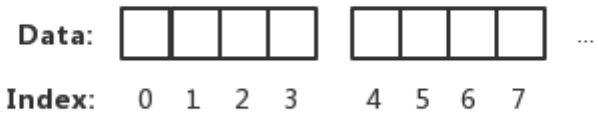
最终输出为占用 32 个字节。这与前面所预期的结果完全不一样。这充分地说明了先前的计算方式是错误的。
为什么呢？
在这里要提到 “内存对齐” 这一概念，才能够用正确的姿势去计算，接下来我们详细的讲讲它是什么

内存对齐

有的小伙伴可能会认为内存读取，就是一个简单的字节数组摆放



上图表示一个坑一个萝卜的内存读取方式。但实际上 CPU 并不会以一个一个字节去读取和写入内存。相反 CPU 读取内存是**一块一块读取**的，块的大小可以为 2、4、6、8、16 字节等大小。块大小我们称其为**内存访问粒度**。如下图：



在样例中，假设访问粒度为 4。 CPU 是以每 4 个字节大小的访问粒度去读取和写入内存的。这才是正确的姿势

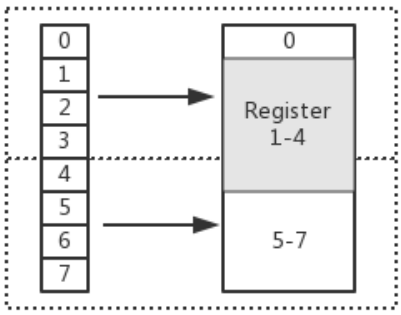
为什么要关心对齐

- 你正在编写的代码在性能（CPU、Memory）方面有一定的要求
- 你正在处理向量方面的指令
- 某些硬件平台（ARM）体系不支持未对齐的内存访问

另外作为一个工程师，你也很有必要学习这块知识点哦 :)

为什么要做对齐

- 平台（移植性）原因：不是所有的硬件平台都能够访问任意地址上的任意数据。例如：特定的硬件平台只允许在特定地址获取特定类型的数据，否则会导致异常情况
- 性能原因：若访问未对齐的内存，将会导致 CPU 进行两次内存访问，并且要花费额外的时钟周期来处理对齐及运算。而本身就对齐的内存仅需要一次访问就可以完成读取动作



在上图中，假设从 Index 1 开始读取，将会出现很崩溃的问题。因为它的内存访问边界是不对齐的。因此 CPU 会做一些额外的处理工作。如下：

1. CPU **首次**读取未对齐地址的第一个内存块，读取 0-3 字节。并移除不需要的字节 0
2. CPU **再次**读取未对齐地址的第二个内存块，读取 4-7 字节。并移除不需要的字节 5、6、7 字节
3. 合并 1-4 字节的数据
4. 合并后放入寄存器

从上述流程可得出，不做 “内存对齐” 是一件有点 "麻烦" 的事。因为它会增加许多耗费时间的动作

而假设做了内存对齐，从 Index 0 开始读取 4 个字节，只需要读取一次，也不需要额外的运算。这显然高效很多，是标准的**空间换时间**做法

默认系数

在不同平台上的编译器都有自己默认的 “对齐系数”，可通过预编译命令 `#pragma pack(n)` 进行变更，n 就是代指 “对齐系数”。一般来讲，我们常用的平台的系数如下：

- 32 位：4
- 64 位：8

另外要注意，不同硬件平台占用的大小和对齐值都可能是不一样的。因此本文的值不是唯一的，调试的时候需按本机的实际情况考虑

成员对齐

```
func main() {
    fmt.Printf("bool align: %d\n", unsafe.Alignof(bool(true)))
    fmt.Printf("int32 align: %d\n", unsafe.Alignof(int32(0)))
    fmt.Printf("int8 align: %d\n", unsafe.Alignof(int8(0)))
    fmt.Printf("int64 align: %d\n", unsafe.Alignof(int64(0)))
    fmt.Printf("byte align: %d\n", unsafe.Alignof(byte(0)))
    fmt.Printf("string align: %d\n", unsafe.Alignof("EDDYCJY"))
```

```
    fmt.Printf("map align: %d\n", unsafe.Alignof(map[string]string{}))
}
输出结果：
bool align: 1
int32 align: 4
int8 align: 1
int64 align: 8
byte align: 1
string align: 8
map align: 8
```

在 Go 中可以调用 `unsafe.Alignof` 来返回相应类型的对齐系数。通过观察输出结果，可得知基本都是 `2^n`，最大也不会超过 8。这是因为我手提（64 位）编译器默认对齐系数是 8，因此最大值不会超过这个数

整体对齐

在上小节中，提到了结构体中的成员变量要做字节对齐。**那么想当然身为最终结果的结构体，也是需要做字节对齐的**

对齐规则

- 结构体的成员变量，第一个成员变量的偏移量为 0。往后的每个成员变量的对齐值必须为**编译器默认对齐长度**（`#pragma pack(n)`）或**当前成员变量类型的长度**（`unsafe.Sizeof`），取**最小值作为当前类型的对齐值**。其偏移量必须为对齐值的整数倍
- 结构体本身，对齐值必须为**编译器默认对齐长度**（`#pragma pack(n)`）或**结构体的所有成员变量类型中的最大长度**，取**最大数的最小整数倍**作为对齐值
- 结合以上两点，可得知若**编译器默认对齐长度**（`#pragma pack(n)`）超过结构体内成员变量的类型最大长度时，默认对齐长度是没有任何意义的

分析流程

接下来我们一起分析一下，“它” 到底经历了些什么，影响了 “预期” 结果

成员变量	类型	偏移量	自身占用
a	bool	0	1
字节对齐	无	1	3
b	int32	4	4
c	int8	8	1
字节对齐	无	9	7
d	int64	16	8
e	byte	24	1

字节对齐	无	25	7
总占用大小	-	-	32

成员对齐

- 第一个成员 a
  - 类型为 bool
  - 大小/对齐值为 1 字节
  - 初始地址，偏移量为 0。占用了第 1 位
- 第二个成员 b
  - 类型为 int32
  - 大小/对齐值为 4 字节 --32/8
  - 根据规则 1，其偏移量必须为 4 的整数倍。确定偏移量为 4，因此 2-4 位为 Padding。而当前数值从第 5 位开始填充，到第 8 位。如下：axxx|bbbb
- 第三个成员 c
  - 类型为 int8
  - 大小/对齐值为 1 字节 --8/8
  - 根据规则1，其偏移量必须为 1 的整数倍。当前偏移量为 8。不需要额外对齐，填充 1 个字节到第 9 位。如下：axxx|bbbb|c...
- 第四个成员 d
  - 类型为 int64
  - 大小/对齐值为 8 字节
  - 根据规则 1，其偏移量必须为 8 的整数倍。确定偏移量为 16，因此 9-16 位为 Padding。而当前数值从第 17 位开始写入，到第 24 位。如下：axxx|bbbb|cxxx|xxxx|dddd|dddd
- 第五个成员 e
  - 类型为 byte
  - 大小/对齐值为 1 字节
  - 根据规则 1，其偏移量必须为 1 的整数倍。当前偏移量为 24。不需要额外对齐，填充 1 个字节到第 25 位。如下：axxx|bbbb|cxxx|xxxx|dddd|dddd|e...

整体对齐

在每个成员变量进行对齐后，根据规则 2，整个结构体本身也要进行字节对齐，因为可发现它可能并不是 `2^n`，不是偶数倍。显然不符合对齐的规则

根据规则 2，可得出对齐值为 8。现在的偏移量为 25，不是 8 的整倍数。因此确定偏移量为 32。对结构体进行对齐

结果

Part1 内存布局：axxx|bbbb|cxxx|xxxx|dddd|dddd|exxx|xxxx

小结

通过本节的分析，可得知先前的 “推算” 为什么错误?

是因为实际内存管理并非 “一个萝卜一个坑” 的思想。而是一块一块。通过空间换时间（效率）的思想来完成这块读取、写入。另外也需要兼顾不同平台的内存操作情况

巧妙的结构体

在上一小节，可得知根据成员变量的类型不同，其结构体的内存会产生对齐等动作。那假设字段顺序不同，会不会有什么变化呢？我们一起来试试吧 :-)

```
type Part1 struct { a bool b int32 c int8 d int64 e byte } type Part2 struct { e byte c int8 a bool b int32 d int64 } func main() { part1 := Part1{} part2 := Part2{} fmt.Printf("part1 size: %d, align: %d\n", unsafe.Sizeof(part1), unsafe.Alignof(part1)) fmt.Printf("part2 size: %d, align: %d\n", unsafe.Sizeof(part2), unsafe.Alignof(part2)) }
```

输出结果：

```
part1 size: 32, align: 8 part2 size: 16, align: 8
```

通过结果可以惊喜的发现，只是 “简单” 对成员变量的字段顺序进行改变，就改变了结构体占用大小

接下来我们一起剖析一下 `Part2`，看看它的内部到底和上一位之间有什么区别，才导致了这样的结果？

分析流程

成员变量	类型	偏移量	自身占用
e	byte	0	1
c	int8	1	1
a	bool	2	1
字节对齐	无	3	1
b	int32	4	4
d	int64	8	8
总占用大小	-	-	16

成员对齐

- 第一个成员 e
  - 类型为 byte
  - 大小/对齐值为 1 字节
  - 初始地址，偏移量为 0。占用了第 1 位
- 第二个成员 c
  - 类型为 int8
  - 大小/对齐值为 1 字节
  - 根据规则1，其偏移量必须为 1 的整数倍。当前偏移量为 2。不需要额外对齐
- 第三个成员 a
  - 类型为 bool
  - 大小/对齐值为 1 字节

    - 根据规则1，其偏移量必须为 1 的整数倍。当前偏移量为 3。不需要额外对齐
- 第四个成员 b
    - 类型为 int32
    - 大小/对齐值为 4 字节
    - 根据规则1，其偏移量必须为 4 的整数倍。确定偏移量为 4，因此第 3 位为 Padding。而当前数值从第 4 位开始填充，到第 8 位。如下：ecax|bbbb
- 第五个成员 d
    - 类型为 int64
    - 大小/对齐值为 8 字节
    - 根据规则1，其偏移量必须为 8 的整数倍。当前偏移量为 8。不需要额外对齐，从 9-16 位填充 8 个字节。如下：ecax|bbbb|dddd|dddd

整体对齐

符合规则 2，不需要额外对齐

结果

Part2 内存布局：ecax|bbbb|dddd|dddd

总结

通过对比 `Part1` 和 `Part2` 的内存布局，你会发现两者有很大的不同。如下：

- Part1：axxx|bbbb|cxxx|xxxx|dddd|dddd|exxx|xxxx
- Part2：ecax|bbbb|dddd|dddd

仔细一看，`Part1` 存在许多 Padding。显然它占据了不少空间，那么 Padding 是怎么出现的呢?
通过本文的介绍，可得知是由于不同类型导致需要进行字节对齐，以此保证内存的访问边界
那么也不难理解，为什么**调整结构体内成员变量的字段顺序**就能达到缩小结构体占用大小的疑问了，是因为巧妙地减少了 Padding 的存在。让它们更 “紧凑” 了。这一点对于加深 Go 的内存布局印象和大对象的优化非常有帮
当然了，没什么特殊问题，你可以不关注这一块。但你要知道这块知识点 😄