深刻理解Go语言的结构体

一、本节内容

主要揭秘Go语言之中结构体在使用中的一些细节，有些部分容易被忽视，但却非常重要，比如结构体的字节对齐，匿名化结构体，以及结构体嵌入等知识点。

二、知识点概要

- 字节对齐
- 结构体匿名
- 结构体嵌入

1. 字节对齐

在一门开发语言中，如果了解语法，那么可以开发程序，但是若要成为开发当中的高手，字节对齐的问题则一定要考虑，这涉及到结构体定义时的空间分配问题，对于精细化计算的地方尤其关键，本节也主要探讨各种字节对齐的问题。我的计划是通过一些具体的实例，让大家理解字节对齐的原理是什么。

1.1 单字节分析

通过例子去分析现象是最好的学习过程，我们来看下面的代码，顺便来猜一猜输出结果是什么，很多面试题都喜欢出这样的选择题。不过代码中，我们需要借助unsafe包中的sizeof方法，来计算结构体的占用空间大小。

```
unsafe.Sizeof(any) uintptr
```

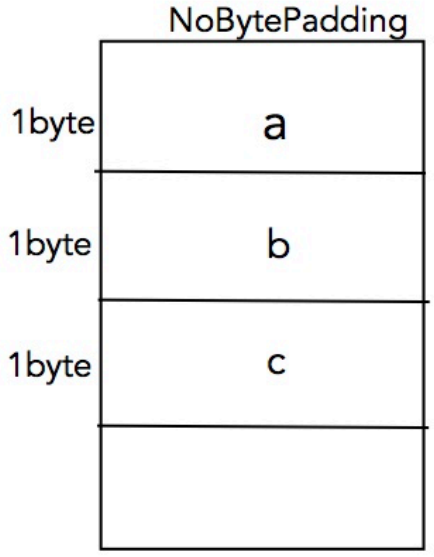
具体来看看代码，猜一猜运行结果。

```
/*
  company: Pdj
  autor  : Yekai
  email  : yekai_23@sohu.com
  file   : 01_padding1.go
*/

package main

import (
  "fmt"
  "unsafe"
)
```

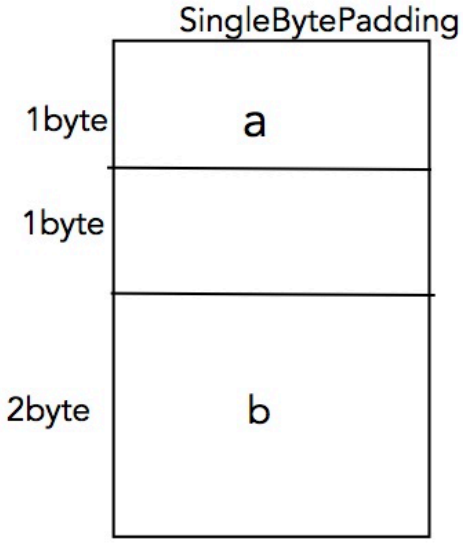
```
//没有补位
type NoBytePadding struct {
  a bool //   1 byte        sizeof 1
  b bool //   1 byte        sizeof 2
  c bool //   1 byte        sizeof 3 - Aligned on 1 byte
}

// 2字节对齐
type SingleBytePadding struct {
  a bool  //  1 byte        sizeof 1
  b int16 //  2 bytes       sizeof 4 - Aligned on 2 bytes
}

// 4字节对齐
type FourBytePadding struct {
  a bool  //  1 byte        size 1
  b int32 //  4 bytes       size 8 - Aligned on 2 bytes
}

func main() {
  var nbp NoBytePadding
  size := unsafe.Sizeof(nbp)
  fmt.Printf("NoBytePadding-SizeOf[%d][%p %p %p]\n", size, &nbp.a, &nbp.b,
&nbp.c)

  var sbp SingleBytePadding
  size = unsafe.Sizeof(sbp)
  fmt.Printf("SingleBytePadding-SizeOf[%d][%p %p]\n", size, &sbp.a, &sbp.b)

  var fbp FourBytePadding
  size = unsafe.Sizeof(fbp)
  fmt.Printf("SingleBytePadding-SizeOf[%d][%p %p]\n", size, &fbp.a, &fbp.b)
}
```

执行看一看，然后分析一下为什么，也许就能得到答案了！

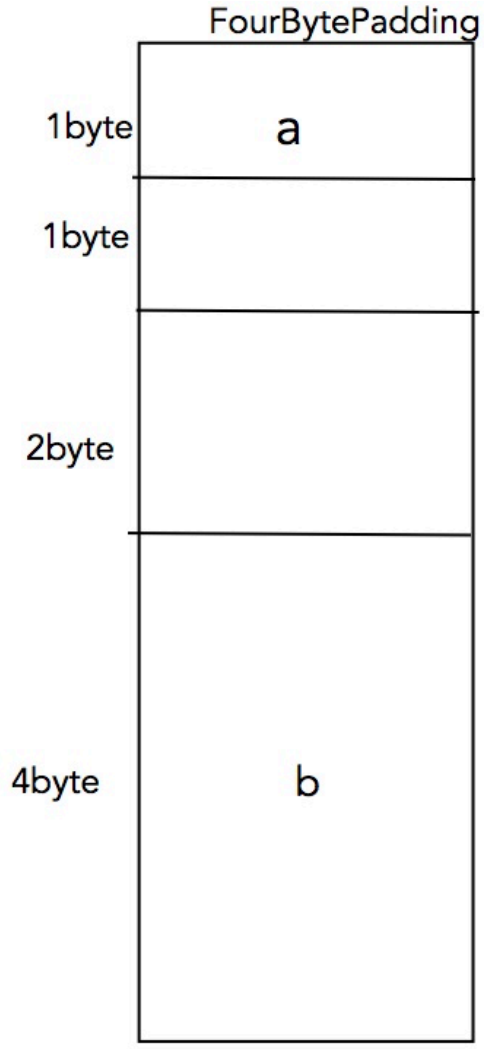
我们还是通过图片来解释来分析一下




```go
//没有补位
type NoBytePadding struct {
    a bool //   1 byte
    b bool //   1 byte
    c bool //   1 byte
}
```




```go
type SingleBytePadding struct {
    a bool  //  1 byte
    b int16 //  2 bytes
}
```




```
type FourBytePadding struct {
    a bool  //  1 byte
    b int32 //  4 bytes
}
```

字节对齐的要点：

- 先找结构体最大字段占用字节数（最大字节单元--N）
- 结构体字节总数一定是 n*N
- 当某字段（或几个字段累加）长度不足最大字节单元时，将会产生补齐现象，直到符合最大字节单元为止
- 最大单元不能超过8字节

1.2 字节对齐验证

经过我们的分析，应该可以对字节对齐有一定的理解，接下来为了验证我们的理解是否正确，我们再来一段代码测试验证一下。

```
/*
  company: Pdj
  autor  : Yekai
  email  : yekai_23@sohu.com
  file   : 02_padding2.go
*/
```

```go
package main

import (
  "fmt"
  "unsafe"
)

// 8字节对齐
type EightBytePadding struct {
  a bool  //  1 byte        size 1
  b int64 //  8 bytes       size 16 - Aligned on 8 bytes
}

// Eight byte padding on 64bit Arch. Word size is 8 bytes.
type ebp64 struct {
  a string // 16 bytes      size 16
  b int32  //  4 bytes      size 20
  c string // 16 bytes      size 40
  d int32  //  4 bytes      size 44
}

// No padding.
type NoPadding struct {
  a string // 16 bytes      size 16
  b string // 16 bytes      size 32
  c int32  //  8 bytes      size 40
  d int32  //  8 bytes      size 48 - Aligned on 8 bytes
}

func main() {
  var ebp EightBytePadding
  size := unsafe.Sizeof(ebp)
  fmt.Printf("EightBytePadding-SizeOf[%d][%p %p]\n", size, &ebp.a, &ebp.b)

  var ebp64 ebp64
  size = unsafe.Sizeof(ebp64)
  fmt.Printf("SizeOf[%d][%p %p %p %p]\n", size, &ebp64.a, &ebp64.b, &ebp64.c,
&ebp64.d)

  var np NoPadding
  size = unsafe.Sizeof(np)
  fmt.Printf("SizeOf[%d][%p %p %p %p]\n", size, &np.a, &np.b, &np.c, &np.d)
}
```

通过上述例子，我们对我们之前的验证比较坚定了，而且也发现Go语言当中string类型占用16个字节。

1.3 小结

Go语言的结构体对齐方式核心思想就是结构体内最大字段长度的整数倍，不过这个最大长度不会超过8，字节对齐可以让你更好的理解内存与程序的关系。

2. 可以匿名的结构体

2.1 匿名结构体的使用

在结构体的使用上，我们都习惯先定义一个类型，然后再使用。Go语言从实际开发角度出发，认为结构体也不过是多个不同类型组成的一个集合，而有些时候无需定义结构体的名字，因此Go语言之中，可以使用匿名函数一样的方法，使用匿名结构体。参考代码如下：

```
/*
  company: Pdj
  autor  : Yekai
  email  : yekai_23@sohu.com
  file   : 03_anon_struct.go
*/

package main

import "fmt"

func main() {

  // 这个是什么操作?
  var e1 struct {
    flag    bool
    counter int16
    pi      float32
  }

  // 打印变量e1
  fmt.Printf("%+v\n", e1)

  // 定义且初始化一个结构体变量
  e2 := struct {
    flag    bool
    counter int16
    pi      float32
  }{
    flag:    true,
    counter: 10,
    pi:      3.141592,
  }

  // 打印e2的信息
```

```
    fmt.Printf("%+v\n", e2)
    fmt.Println("Flag", e2.flag)
    fmt.Println("Counter", e2.counter)
    fmt.Println("Pi", e2.pi)
}
```

执行一下看看，和我们熟知的结构体没什么区别！

```
root:struct yk$ go run 03-anon_struct.go
{flag:false counter:0 pi:0}
{flag:true counter:10 pi:3.141592}
Flag true
Counter 10
Pi 3.141592
```

2.2 匿名结构体与结构体类型互通

如果定义了匿名结构体变量，那么定义一个非匿名结构体变量时，他们可以互相赋值吗？那么他们之间可以互相赋值吗？答案是肯定的，不过要有三个前提：

- 结构体字段个数相同；
- 结构体字段类型一致；
- 结构体字段名称一致；

不妨来看看示例代码：

```
/*
    company: Pdj
    autor  : Yekai
    email  : yekai_23@sohu.com
    file   : 04_anon_struct2.go
*/
package main

import "fmt"

//定义个不同类型字段的结构体
type example struct {
    flag    bool
    counter int16
    pi      float32
}

func main() {

    //定义一个匿名结构体变量
```

```
    e := struct {
      flag    bool
      counter int16
      pi      float32
    }{
      flag:    true,
      counter: 10,
      pi:      3.141592,
    }

    // 定义一个非匿名结构体变量
    var ex example

    ex = e

    //打印相关信息
    fmt.Printf("%+v\n", ex)
    fmt.Printf("%+v\n", e)

  }
```

执行结果如下：

```
  root:struct yk$ go run 04-anon_struct2.go
  {flag:true counter:10 pi:3.141592}
  {flag:true counter:10 pi:3.141592}
```

2.3 小结

匿名化可以很便利的提高编程效率，结构体应灵活使用。

3. 结构体内嵌

结构体引用另外一个结构体可以有两种方式：内嵌和非内嵌，如果采用内嵌的方式，那么结构体之间会存在一种继承关系，如果采用非内嵌的方式，那么结构体不存在继承关系，但也可以通过结构体字段调用对应的结构体内部函数。

3.1 非内嵌

```
/*
  company: Pdj
  autor  : Yekai
  email  : yekai_23@sohu.com
  file   : 05-no_embed.go
*/
package main
```

```
import "fmt"

//定义一个用户结构体
type user struct {
  name  string
  email string
}

// 定义user的一个通知函数
func (u *user) notify() {
  fmt.Printf("Sending user email To %s<%s>\n",
    u.name,
    u.email)
}

// 定义管理员结构，包含user字段，非内嵌
type admin struct {
  person user // NOT Embedding
  level  string
}

func main() {

  // 创建一个管理员变量并初始化
  ad := admin{
    person: user{
      name:  "yekai",
      email: "yekai@yahoo.com",
    },
    level: "super",
  }

  //我们要调用notify必须使用person
  ad.person.notify()
}
```

3.2 内嵌

将上述代码简单改造，也就是把结构体的引用方式改为内嵌的。

```
/*
  company: Pdj
  autor  : Yekai
  email  : yekai_23@sohu.com
  file   : 06-embed.go
*/
package main
```

```
import "fmt"

//定义一个用户结构体
type user struct {
  name  string
  email string
}

// 定义user的一个通知函数
func (u *user) notify() {
  fmt.Printf("Sending user email To %s<%s>\n",
    u.name,
    u.email)
}

// 定义管理员结构，包含user字段，非内嵌
type admin struct {
  user  // Embed type
  level string
}

func main() {

  // 创建一个管理员变量并初始化
  ad := admin{
    user: user{
      name:  "yekai",
      email: "yekai@yahoo.com",
    },
    level: "super",
  }

  //我们要调用notify可以借助user
  ad.user.notify()
  //我们也可以直接调用内部函数的方式调用notify
  ad.notify()
}
```

3.3 内嵌与接口

当通过内嵌方式引用时，如果父结构体支持某一个接口，那么子结构体也支持该接口。

```
/*
  company: Pdj
```

```
  autor   : Yekai
  email   : yekai_23@sohu.com
  file    : 07-embed_interface.go
*/
package main

import "fmt"

//定义一个接口，包含notify函数
type notifier interface {
  notify()
}

// 定义user结构体
type user struct {
  name  string
  email string
}

// user实现notify
func (u *user) notify() {
  fmt.Printf("Sending user email To %s<%s>\n",
    u.name,
    u.email)
}

// 定义admin结构体，内嵌user
type admin struct {
  user
  level string
}

func main() {

  // 创建admin变量
  ad := admin{
    user: user{
      name:  "yekai",
      email: "yekai@yahoo.com",
    },
    level: "super",
  }

  //调用函数，该函数参数为接口,admin并未实现notify，但也可以作为参数传递
  sendNotification(&ad)
}

//接口作为参数的函数
func sendNotification(n notifier) {
```

```
  n.notify()
}
```

3.4 接口冲突与内嵌

```
package main

import "fmt"

//定义一个接口，包含notify函数
type notifier interface {
  notify()
}

// 定义user结构体
type user struct {
  name  string
  email string
}

// user实现notify
func (u *user) notify() {
  fmt.Printf("Sending user email To %s<%s>\n",
    u.name,
    u.email)
}

// 定义admin结构体，内嵌user
type admin struct {
  user
  level string
}

// admin实现notify
func (a *admin) notify() {
  fmt.Printf("Sending admin Email To %s<%s>\n",
    a.name,
    a.email)
}

func main() {

  // 创建admin变量
  ad := admin{
    user: user{
      name:  "yekai",
      email: "yekai@yahoo.com",
    },
```

```
    level: "super",
  }

  //此时调用哪个notify?
  sendNotification(&ad)

  //调用内嵌user的notify
  ad.user.notify()

  //调用本身的
  ad.notify()
}

//接口作为参数的函数
func sendNotification(n notifier) {
  n.notify()
}
```

3.5 小结

我们可以把内嵌与接口的关系总结如下：

- 内嵌时，父结构体支持接口，子结构体天生支持
- 内嵌时，父子结构体都支持接口，接口调用传入子结构体时调用子结构体的方法
- 内嵌与否，子结构体都可以调用父结构体的方法，只不过方式不同

三、 总结

Go语言是一门工程化的语言，而工程中定面向对象是在所难免的，把结构体的知识掌握好，对于编程很有帮助。我们来总结一下Go语言的结构体：

- 对空间想要精确处理的，要考虑字节对齐
- 可以使用匿名结构体
- 内嵌相当于面向对象的继承
- 内嵌结构体后，天然支持父结构体支持的接口