> 本文由 简悦 SimpRead 转码， 原文地址 kaiwu.lagou.com

错误处理是软件开发中不可回避的问题，**Go 中主要通过 error 和 panic 分别表示错误和异常**，并提供了较为简洁的错误异常处理机制。作为一门并发性能优越的语言，Go 虽然降低了协程并发开发的难度，但也存在一些**并发陷阱**，这就需要我们在开发时额外注意。

在本课时，我们就来介绍 Go 中的一些错误处理机制，并讲解如何去规避一些常见的并发陷阱。

Errors are values

错误处理是每个开发人员都需要面对的问题，在我过去接触的编程语言中，大多是通过 **try-catch 的方式**对可能出现错误的代码块进行包装：**程序运行 try 中代码，如果 try 中的代码运行出错，程序将会立即跳转到 catch 中执行异常处理逻辑。**

与其他的编程语言不同，Go 中倡导 “Errors are values!” 的处理思想，它将 error 作为一个返回值，来迫使调用者对 error 进行处理或者忽略。于是，在代码中我们将会编写大量的 if 判断语句对 error 进行判断，如下所示：

```
result, err := dothing("work")

if err != nil {

    log.Fatal(err)

}
```

当项目的代码快速增长起来时，我们会发现代码中到处都是类似 err != nil 的判断片段。虽然这会使代码变得很烦琐，但是这种设计和约定也会鼓励开发人员明确检查和确定错误发生的位置。

在 Go 中，error 接口定义如下：

```
type error interface {

    Error() string

}
```

最常用的 error 实现是 Go 标准库 errors 包中内置的 errorString，它是一个仅包含错误信息的 error 实现，可以通过 errors.New 和 fmt.Errorf 函数创建。内置的 error 接口使得开发人员可以为错误添加任何所需的信息，error 可以是实现 Error() 方法的任何类型，比如我们可以为错误添加错误码和调用栈信息，如下所示：

```
type Error struct {

    Msg string
```

```
    Code int32

    St []uintptr

}

func callers() []uintptr {

var pcs [32]uintptr

    n := runtime.Callers(3, pcs[:])

    st := pcs[0:n]

return st

}

func New(code int32, msg string) error {

return &Error{

        Code: int32(code),

        Msg:  msg,

        St: callers(),

    }

}

func (e *Error) Error() string {

if e == nil {

return "OK"
```

```
    }

return fmt.Sprintf("code:%d, msg:%s", e.Code, e.Msg)

}
```

通过**断言**的方式可以将 error 转化为特定的类型从而进行特异化处理，如下所示：

```
if e, ok := err.(*Error); ok {

    st := e.st

}else {

}
```

在 Go 1.13 版本之后，errors 包中添加了 errors.Is 和 errors.As 函数：errors.Is 方法用来比较两个 error 是否相等，而 errors.As 函数用来判断 error 是否为特定类型。

由于 error 是一个值，因此我们可以对其进行编程，简化 Go 错误处理的重复代码。在一些管道和循环的代码中，只要其中一次处理出现错误，就应该退出本次管道或者循环。寻常的做法是在每次迭代都检查错误，但为了让管道和循环的操作显得更加自然，我们可以将 error 封装到独立的方法或者变量中返回，以避免错误处理掩盖控制流程，如 gorm 中的 DB 设计所示：

```
err := DB.Where(queryString, queryValue...).

        Table("table_name").

        Updates(map[string]interface{}{...}).Error

if err != nil{

}
```

这里 error 是从 gorm.DB 的 Error 成员变量中获取的。在数据库请求执行结束之后，程序才从 DB 中获取执行错误，这样的写法使得错误处理不会中断执行流程。但需要注意的是，无论如何简化 error 的设计，程序都要检查和处理错误，错误是无法避免的。

defer、panic 和 recover

错误一般是一些开发人员 “意料之内” 的错误，比如获取数据库连接失败等，这些都是在 Go 中通过 error 表达并可控。但当程序出现异常，如数组访问越界这类 “意料之外” 的错误时，它能够导致程序运行崩溃，此时就需要开发人员捕获异常并恢复程序的正常运行流程。

接下来我们就介绍 defer、panic 和 recover 如何组合恢复运行时执行异常的 Go 程序。

defer 是 Go 中提供的一种延迟执行机制，每次执行 defer，都会将对应的函数压入栈中。在函数返回或者 panic 异常结束时，Go 会依次从栈中取出延迟函数执行。

在编程的时候，经常需要打开一些资源，比如数据库连接、文件等，在资源使用完成之后需要释放，不然有可能会造成资源泄漏。这个时候，我们可以通过 defer 语句在函数执行完之后，自动释放资源，避免在每个函数返回之前手动释放资源，减少冗余代码。

defer 有三个比较重要的特点。**第一个是按照调用 defer 的逆序执行**，即后调用的在函数退出时先执行，**后进先出**。如下例子所示:

```
func main()  {

defer fmt.Println("I register at first, but execute at last")

defer fmt.Println("I register at middle, execute at middle")

defer fmt.Println("I register at last, execute at first")

    fmt.Println("test begin")

}
```

预期的结果为:

```
test begin

I register at last, execute at first

I register at middle, execute at middle

I register at first, but execute at last
```

第二个特点是 defer 被定义时，参数变量会被立即解析，传递参数的值拷贝。在函数内使用的变量其实是对外部变量的一个拷贝，在函数体内，对变量更改也不会影响外部变量，如下所示:

```
func main()  {

    i := 10

defer fmt.Printf("defer i is %d\n", i)
```

```
    i = 20

    fmt.Printf("current i is %d\n", i)

}
```

预期结果为：

```
current i is 20

defer i is 10
```

然而当 defer 以闭包的方式引用外部变量时，则会在延迟函数真正执行的时候，根据整个上下文确定当前的值，如下示例代码：

```
func main()  {

for i := 0; i < 5; i++ {

defer func() {

            fmt.Println(i)

        }()

    }

}
```

预期的输出结果为：

上述例子为了演示简单，在 for 循环中使用了 defer，但**在日常开发中，我建议你还是不要在循环中使用 defer**。因为相较于直接调用，defer 的执行存在着额外的开销，例如 defer 会对其后需要的参数进行内存拷贝，还会对 defer 结构进行压栈出栈操作。因此，在循环中使用 defer 可能会带来较大的性能开销。

defer 的第三个特点是可以读取并修改函数的命名返回值，如下面的例子所示：

```
func main()  {

    fmt.Println(test())
```

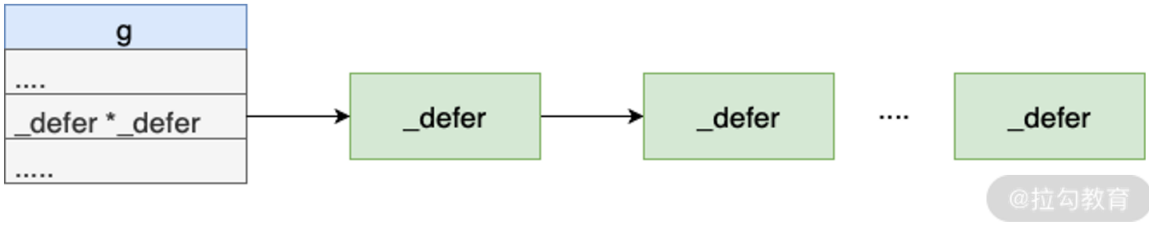
```
}

func test() (i int) {

defer func() { i++ }()

return 1

}
```

预期的返回结果为：

这是因为对于命名返回值，defer 和 return 的执行顺序如下：

- 将 1 赋给 i；
- 执行 i++；
- 返回 i 作为函数返回值。

defer 的内部实现为一个**延迟调用链表**，如下图所示：



defer 延迟调用链表示意图

其中，g 代表 goroutine 的数据结构。每个 goroutine 中都有一个 _defer 链表，当代码中遇到 defer 关键字时，Go 都会将 defer 相关的函数和参数封装到 _defer 结构体中，然后将其注册到当前 goroutine 的 _defer 链表的表头。在当前函数执行完毕之后，Go 会从 goroutine 的 _defer 链表头部取出来注册的 defer 执行并返回。

_defer 结构体中存储 defer 执行相关的信息，定义如下所示：

```
type _defer struct {

    siz       int32

    started   bool

    heap      bool

    openDefer bool

    sp        uintptr
```

```
    pc        uintptr

    fn        *funcval

    _panic    *_panic

    link      *_defer

}
```

panic 是一个内置函数，用于抛出程序执行的异常。它会终止其后将要执行的代码，并依次逆序执行 panic 所在函数可能存在的 defer 函数列表；然后返回该函数的调用方，如果函数的调用方中也有 defer 函数列表，也将被逆序执行，执行结束后再返回到上一层调用方，直到返回当前 goroutine 中的所有函数为止，最后报告异常，程序崩溃退出。异常可以直接通过 panic 函数调用抛出，也可能是因为运行时错误而引发，比如访问了空指针等。

而 **recover 内置函数可用于捕获 panic，重新恢复程序正常执行流程，但是 recover 函数只有在 defer 内部使用才有效。**如下面例子所示：

```
func main() {

    err := panicAndReturnErr()

if err != nil{

        fmt.Printf("err is %+v\n", err)

    }

    fmt.Println("returned normally from panicAndReturnErr")

}

func panicAndReturnErr() (err error){

defer func() {

if e := recover(); e != nil {

            buf := make([]byte, 1024)
```

```
            buf = buf[:runtime.Stack(buf, false)]

            err = fmt.Errorf("[PANIC]%v\n%s\n", e, buf)

        }

    }()

    fmt.Println("panic begin")

panic("panic this game")

    fmt.Println("panic over")

return nil

}
```

预期的执行结果为：

```
panic begin

err is [PANIC]panic this game

goroutine 1 [running]:

main.panicAndReturnErr.func1(0xc000062f08)

    /Users/apple/Desktop/micro-go-course/section37/defer_example.go:21 +0xa1

panic(0x10ad640, 0x10eb360)

    /usr/local/go/src/runtime/panic.go:969 +0x166

main.panicAndReturnErr(0x0, 0x0)

    /Users/apple/Desktop/micro-go-course/section37/defer_example.go:26 +0xc2
```

```
main.main()

    /Users/apple/Desktop/micro-go-course/section37/defer_example.go:10 +0x26

returned normally from panicAndReturnErr
```

从这个执行结果可以看出，panicAndReturnErr 函数在 panic 之后将会执行 defer 定义的延迟函数，恢复程序的正常执行逻辑。在上述例子中，我们在 defer 函数中使用 recover 函数帮助程序从 panic 中恢复过来，并获取异常堆栈信息组成 error 返回调用方。panicAndReturnErr 从 panic 中恢复后将直接返回，不会执行函数中 panic 后的其他代码。

在日常开发中，对于可能出现执行异常的函数，如数组越界、操作空指针等，在函数中定义一个使用 recover 函数的 defer 延迟函数，有利提高程序执行的健壮性，避免程序运行时异常崩溃。

常见的并发陷阱

最后我们再来介绍与 Go 并发相关的几个小技巧，帮助你规避 Go 并发开发的一些陷阱。

首先是循环并发时闭包传递参数的问题，如下错误例子所示：

```
func main()  {

for i := 0 ; i < 5 ; i++{

go func() {

            fmt.Println("current i is ", i)

        }()

    }

    time.Sleep(time.Second)

}
```

这段代码极有可能的输出为：

```
current i is 5

current i is 5

current i is 5
```

```
current i is 5

current i is 5
```

这是因为 i 使用的地址空间在循环中被复用，在 goroutine 执行时，i 的值可能在被主 goroutine 修改，而此时其他 goroutine 也在读取使用，从而导致了并发错误。针对这种错误可以通过**复制拷贝**或者**传参拷贝**的方式规避，如下所示：

```
func main()  {

for i := 0 ; i < 5 ; i++{

go func(v int) {

            fmt.Println("current i is", v)

        }(i)

    }

    time.Sleep(time.Second)

}
```

前面介绍 panic 时我们了解到 panic 异常的出现会导致 Go 程序的崩溃。但其实即使 panic 是出现在其他启动的子 goroutine 中，也会导致 Go 程序的崩溃退出，同时 panic 只能捕获 goroutine 自身的异常，因此 ** 对于每个启动的 goroutine，都需要在入口处捕获 panic，并尝试打印堆栈信息并进行异常处理，**从而避免子 goroutine 的 panic 导致整个程序的崩溃退出。**如下面的例子所示：

```
func RecoverPanic() {

if e := recover(); e != nil {

        buf := make([]byte, 1024)

        buf = buf[:runtime.Stack(buf, false)]

        fmt.Printf("[PANIC]%v\n%s\n", e, buf)
```

```
    }

}

func main() {

for i:= 0 ; i < 5 ; i++{

go func() {

defer RecoverPanic()

            dothing()

        }()

    }

}
```

最后一个技巧是要**善于结合使用 select、timer 和 context 进行超时控制。**在 goroutine 中进行一些耗时较长的操作，最好都加上超时 timer，在并发的时候也要传递 context，这样在取消的时候就不会有遗漏，进而达到回收 goroutine 的目的，避免内存泄漏的发生。如下面的例子所示，通过 select 同时监听任务和定时器状态，在定时器到达而任务未完成之时，提前结束任务，清理资源并返回。

```
select {

case msg <- input:

   ....

case <-ctx.Done():

return

case <-time.After(time.Second * 2)

return

default:
```

```
}
```

小结

在本课时我们主要介绍了 Go 中常见的错误处理机制和一些并发开发技巧。

Go 倡导将错误作为返回值返回给调用方，由调用方决定如何处理或者忽略错误。通过 defer 和 recover 内置函数，我们可以轻易地将运行时异常的 Go 程序恢复到正常执行流程。另外，Go 并发开发中存在不少的并发陷阱，这些都需要我们在开发中额外留意并规避。

最后，对于 Go 中的错误处理和并发技巧，你还有哪些经验？欢迎在留言区与我分享。