|  |
| --- |
| <https://www.cnblogs.com/junneyang/p/6078810.html> |
| <https://blog.csdn.net/hellolingyun/article/details/52433167> |
| <http://xiaorui.cc/2016/03/09/%E5%85%B3%E4%BA%8Egolang%E7%9A%84panic-recover%E5%BC%82%E5%B8%B8%E9%94%99%E8%AF%AF%E5%A4%84%E7%90%86/> |
| https://edu.aliyun.com/a/12168 |

Panic:

1. 内建函数
2. 假如函数F中有panic语句，则会终止其后要执行的代码，在panic所在函数F内如果存在要执行的defer函数列表，按照defer的逆序执行
3. 返回函数F的调用者G，在G中，调用函数F语句之后的代码不会执行，假如函数G中存在要执行的defer函数列表，按照defer的逆序执行
4. 直到goroutine整个退出，并报告错误

Recover:

1. 内建函数
2. 用来控制一个goroutine的panicking行为，捕获panic，从而影响应用的行为
3. 调用建议 在defer函数中，通过recover来终止一个goroutine的panicking过程，从而回复正常代码的执行；可以获取通过panic传递的error

使用panic抛出异常，抛出异常后将立即停止当前函数的执行，并运行所有被defer的函数，然后将panic抛向上一层，直到程序crash。可以使用被defer的recover函数来捕获异常阻止程序的崩溃，recover只有被defer之后才有意义

注意事项：

1. defer需要放在panic之前定义，另外recover只有在defer调用的函数中才有效
2. recover处理异常后，逻辑并不会恢复到panic那个点去，函数跑到defer之后的那个点
3. 多个defer会形成defer栈，后定义的defer语句会被最先调用

Panic 主动抛出异常 与 recover 收集异常

Recover用来对panic 的异常进行捕获，panic用于向上传递异常，执行顺序时在defer之后。

不处理异常

|  |
| --- |
| func f() {      for {          fmt.Println("1")          a := []string{"a","b"}          fmt.Println(a[3])  // 这里slice越界异常了          /\*panic("bug")\*/          fmt.Println("4")          time.Sleep(1 \* time.Second)      }  } |
| 1  panic: runtime error: index out of range    goroutine 1 [running]:  panic(0xda9c0, 0x8201c8090)      /usr/local/go/src/runtime/panic.go:464 +0x3e6  main.f()      /Users/ruifengyun/gg.go:23 +0x33b  main.main()      /Users/ruifengyun/gg.go:16 +0x14  exit status 2 |

处理异常

|  |
| --- |
| package main    import (      "fmt"      "time"  )    func main() {      defer func() { //必须要先声明defer，否则不能捕获到panic异常          fmt.Println("2")          if err := recover(); err != nil {              fmt.Println(err) //这里的err其实就是panic传入的内容，bug          }          fmt.Println("3")      }()      f()  }    func f() {      for {          fmt.Println("1")          panic("bug")          fmt.Println("4") //不会运行的.          time.Sleep(1 \* time.Second)      }  } |
| 1  2  bug  3 |

出现未知异常

|  |
| --- |
| package main    #http://xiaorui.cc    import (      "fmt"      "time"  )    func main() {      f()      fmt.Println("end")  }    func f() {      defer func() { //必须要先声明defer，否则不能捕获到panic异常          fmt.Println("xiaorui.cc start")          if err := recover(); err != nil {              fmt.Println(err) //这里的err其实就是panic传入的内容，"bug"          }          fmt.Println("xiaorui.cc end")      }()      for {          fmt.Println("1")          a := []string{"a", "b"}          fmt.Println(a[3])  // 越界访问，肯定出现异常          panic("bug”)  // 上面已经出现异常了,所以肯定走不到这里了。          fmt.Println("4") //不会运行的.          time.Sleep(1 \* time.Second)      }  } |
| 1  xiaorui.cc start  runtime error: index out of range  xiaorui.cc end  end |

上述可以看出当出现异常会走到defer这一步，defer这里可以打印具体的异常信息，defer运行完之后不能回到原点，控制权会被扔到该函数的外层，也就是调用这个函数的上层，对应上述例子的代码即为main函数。

其他：

1. panic一旦忘记调用recover，退出的不是一个goroutine而是整个golang进程。对于编写lib的人来说，在你调用别人库的时候一定需要处理错误
2. panic更多的时候可以看作是程序控制权的一种转交，除非精心设计，否则最好不要用他来处理异常

panic和recover的执行过程：

panic:

panic是一个内建函数，可以中断原有的控制流程，进入一个令人惊恐的流程中。当函数F调用panic，函数F的执行被中断，但是F中的延迟函数会正常执行，然后F返回到调用它自身的地方。在调用的地方，F的行为就像效用了Panic。这一过程继续向上，直到发生panic的goroutine中所有调用的函数返回，此时程序退出。恐慌可以直接调用panic产生，也可以由运行时错误产生，例如访问越界的数组。

recover:

recover是一个内建函数，可以让进入恐慌流程中的goroutine恢复过来，recover仅在延迟函数中有效。在正常的执行过程中，调用recover会返回nil，并且没有任何效果。如果当前goroutine陷入恐慌，调用recover可以捕获到panic的输入值，并且恢复正常的执行。

Go没有像java那样的异常机制，它不能抛出异常，而是使用了panic和recover机制。但是应该将它作为最后的手段来使用，即代码中应该没有，或者很少的panic。