Golang 错误和异常处理（含生产环境下的解决方案）

作者：areslovecode

原文链接：https://ld246.com/article/1584599072906

来源网站：链滴

最近在维护之前开发的一个项目，发现系统会出现宕机的问题，导致服务不可用。分析报错的异常后现是由于当时对 Golang 的错误和异常处理不够了解造所成
old_sweat ，所以对 Golang 的错误和异常处理做一次总结。按照处理方式分为错误和异常两个大类

1. 错误

无论是系统自带或者是自定义的函数，理论上只要符合前置条件都能够成功返回。但即使在高质量的计和代码编写时，仍然会出现一些不受程序设计者掌握的情况。比如典型的 I/O 操作函数，就可能随会遇到一些可能的错误，导致程序不能够正确的返回。因此对错误的处理是 API 设计或者应用程序接的重要部分，通过对可预料的一些行为进行体提前判断和处理，以保证在发生这种情况时函数依然能按照期望执行并返回。

1.1 错误的处理方式

在进行函数的返回设计时，通常会在遇到错误时返回一个附加参数作为结果参数的一部分，习惯上（考源码中的写法）把错误值作为最后一个结果返回。大部分函数的错误返回值会直接使用 Golang 语内置的错误类型 error，也可以自定义 error 以适应系统的错误响应要求。

1.1.1 内置错误类型 error

Golang 语言内置的错误类型 error 是一个接口类型

使用 Golang 版本为**go1.13.5 darwin/amd64**

```
package builtin
// ...
type error interface {
    Error() string
}
```

error 接口的具体实现是 errors 包，errors 包定义了错误处理的方法

```
// Package errors implements functions to manipulate errors.
package errors

// New returns an error that formats as the given text.
func New(text string) error {
    return &errorString{text}
}

// errorString is a trivial implementation of error.
type errorString struct {
    s string
}

func (e *errorString) Error() string {
    return e.s
}
```

a. errors 包

1. 错误信息使用 errorString 而不是直接 string（可能是为了后续的扩展）
2. errorstring 实现了 error 接口的 Error()，因此 errorstring 是 error 接口类型的
3. 方法 Error()使用的是指针类型*errorString，每次 New 方法返回的 error 类型都不一样

```
// 返回false
fmt.Println(errors.New("EOF") == errors.New("EOF"))
```

4. 使用过程中构建 error 对象时使用 fmt.Errorf 函数更多一些，因为 fmt.Errorf 可以传递更多参数能满足多种需求

```
func errorTest() {
    s := "1.1"
    _, err := strconv.ParseInt(s, 10, 32)
    if err != nil {
        e := useErrorf("strconv.ParseIn", err)
        fmt.Println(e)
    }
}

func useErrorf(name string, err error) error {
    return fmt.Errorf("call %s error: %v", name, err)
}
//  输出：  call strconv.ParseIn error: strconv.ParseInt: parsing "1.1": invalid syntax
```

b. wrap

在 1.13 之后的版本中 errors 包下多了 wrap.go 文件

```
package errors
```

```
import (
    "internal/reflectlite"
)

// Unwrap returns the result of calling the Unwrap method on err, if err's
// type contains an Unwrap method returning error.
// Otherwise, Unwrap returns nil.
func Unwrap(err error) error {
    u, ok := err.(interface {
        Unwrap() error
    })
    if !ok {
        return nil
    }
    return u.Unwrap()
}


func Is(err, target error) bool {
    ...
}

func As(err error, target interface{}) bool {
    ...
}
```

1. wrap 方法作用

使用Unwrap方法可以获取一个错误中包含的另外一个错误，在 fmt.Errorf 方法的注释中有详细的说：当使用 fmt.Errorf 时使用 % w 做占位符，返回的对象会实现Unwrap方法。

```
package fmt

import "errors"

// Errorf formats according to a format specifier and returns the string as a
// value that satisfies error.
//
// If the format specifier includes a %w verb with an error operand,
// the returned error will implement an Unwrap method returning the operand. It is
// invalid to include more than one %w verb or to supply it with an operand
// that does not implement the error interface. The %w verb is otherwise
// a synonym for %v.
func Errorf(format string, a ...interface{}) error {
```

2. Is 方法作用

和指定的错误类型进行比较

3. As 方法作用

类似于断言，对比错误是否是某一类型

1.2 错误的处理策略

1.2.1 将错误传递下去

如果函数发生了错误，尽量的把错误往调用方抛出，这样调用方可以选择是直接返回错误信息还是换种处理方法。当函数向上抛出错误时，可以在外层添加一层信息，尽可能多的返回错误（调用方可以用上面所提到的 Unwrap 获取到原始错误），例如添加一些自定义的错误类型码，以方便后续的判断处理方式。

```
func error1() {
    s := "1.1"
    i, err := strconv.ParseInt(s, 10, 32)
    if err != nil {
        useErrorf("strconv.ParseIn", err)
        fmt.Println(i)
    }
}

// errors.New并不是很常用，更常用的是fmt.Errorf
func useErrorf(name string, err error) {
    fmt.Println(fmt.Errorf("code:101, call %s error: %v", name, err))
}
```

1.2.2 重试

对于不固定或者不可预测的错误，可以在短暂间隔后对操作进行重试。常见的业务场景就是在网络请的过程中，当遇到错误时经常会重新发起连接，直到成功为止，或者重试一定的次数。例如下面的测程序就是一直重连直到成功为止

```
func GetConn(url string) *websocket.Conn {
    var conn *websocket.Conn
    var err error
    for {
        conn, _, err = websocket.DefaultDialer.Dial(url, nil)
        if err != nil {
            log.Type(SERVICE_NAME).Infof("WS connect failed: %v %v", err, url)
            time.Sleep(time.Second * time.Duration(2))
            continue
        } else {
            break
        }
    }
    return conn
}
```

1.2.3 停止程序

如果依然不能顺利进行下去，调用者可以打印错误，优雅地停止程序。在有些场景下，如果程序运行现出错导致无法继续往下执行，那么可以在打印日志以后，直接让程序退出运行。比如项目启动时连数据库失败，连接redis失败等类型的错误，可以让程序打印错误后终止运行。

1.2.4 继续执行

在一些情况下，可以记录下错误信息以后，程序继续执行。如果遇到了不影响主流程执行的错误，可直接进行打印或者调用写日志API以便事后排查，然后主程序继续执行即可

1.2.5 忽略错误

也有一些错误是不需要处理的，直接忽略掉，程序继续执行即可。例如从一个string类型数组里面读所有的数字类型的字符串时，当遇到转换错误的元素时，可以直接忽略这个元素，继续下一个元素的断。

2. 异常

2.1 panic

除了前面提到的那些在开发阶段预料到的错误，还有一些错误是在运行时进行检查的，例如数组越界者引用了空指针的方法。此时程序无法继续执行就会发生宕机，像下面的例子：当程序执行http的GE方法请求时，对方没有响应，此时resp为空，执行resp.Body.Close()时就会发生宕机错误：

```
resp, err := http.Get(realURL)
defer resp.Body.Close()
```

此时系统报了一个panic错误：

```
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x40 pc=0xb55819]
```

在Go源码中有对panic的详细说明：

内置的函数panic会停止当前协程的正常运行。当方法F触发了panic时，F的正常执行流程会立即终止常规方式调用或者依赖F函数的方法会被停滞，此时F的结果会返回到他的调用方那里。对于F的调用方来说，对F函数的调用就像是直接调用了panic，此时G会终止执行，然后运行延迟函数。这个过程会直到协程内的所有延迟函数以和定义时相反的顺序执行完。这时程序会被终止，并且返回一个非0的出码。程序终止的一系列过程就做panic过程，这一过程可以被另一个内置的函数recover捕获处理。

```
package builtin
// The panic built-in function stops normal execution of the current
// goroutine. When a function F calls panic, normal execution of F stops
// immediately. Any functions whose execution was deferred by F are run in
// the usual way, and then F returns to its caller. To the caller G, the
// invocation of F then behaves like a call to panic, terminating G's
// execution and running any deferred functions. This continues until all
// functions in the executing goroutine have stopped, in reverse order. At
// that point, the program is terminated with a non-zero exit code. This
// termination sequence is called panicking and can be controlled by the
// built-in function recover.
func panic(v interface{})
```

也可以手动触发panic使程序进入panci宕机，例如项目启动过程中如果数据库连接失败或者配置文件取失败，会导致后续程序无法正常执行，那么在判断错误类型后可以手动调用panic。

```
func main() {
    flag.Parse()
    if err := config.Init(); err != nil {
        panic(err)
    }
```

```
    ...
}
```

2.2 recover

当程序发生意料之外的错误进入了上面说的panic状态时，退出程序是正常的处理方式。但此时如果希望程序崩溃，也可以使用recover函数使程序从宕机状态重新进入执行状态。源码中对recover的解是：

内置的recover函数可以使程序接管panic状态的协程。在延迟函数defer中调用recover（并不是所有况下都需要调用）可以通过重新加载回正常程序和拦截传递给panic的错误的形式阻止panic状态的程继续执行。如果recover函数没有在延迟函数中调用，则无法阻止panic状态的继续执行。这种情况下或者是协程并没有发生panic，或者提供给panic的参数是nil，recover都会返回nil。所以recover的回值能起到判断程序是否是panic的左右。

```
// The recover built-in function allows a program to manage behavior of a
// panicking goroutine. Executing a call to recover inside a deferred
// function (but not any function called by it) stops the panicking sequence
// by restoring normal execution and retrieves the error value passed to the
// call of panic. If recover is called outside the deferred function it will
// not stop a panicking sequence. In this case, or when the goroutine is not
// panicking, or if the argument supplied to panic was nil, recover returns
// nil. Thus the return value from recover reports whether the goroutine is
// panicking.
func recover() interface{}
```

当多个协程都在运行时，一个协程中的panic如果没有使用recover拦截，那么会导致整个程序宕机。此，在一些特殊的场景下需要添加recover以避免整个程序的宕机。在文章开始的地方提到的问题，我在生产环境下遇到的。在系统中，使用定时程序进行数据抓取的时候，每一家交易所都使用N个工协程去执行HTTP请求和数据处理，测试阶段没有遇到请求失败的情况，疏忽的没有使用recover进行个协程的panic拦截。结果在上线以后的N天，出现了数据抓取全都停止的状态，最后排查到是由于中一家交易所的接口不稳定，偶发性的出现一次无法请求的状态，导致了整个程序的崩溃：

```
// 在执行请求失败后resp为空。在没有进行resp的判断的情况下
// 直接调用了resp.Body.Close()导致了panic
func get(){
    ...
    resp, err := http.Get(realURL)
    defer resp.Body.Close()
    ...
}
```

2.3 derfer

golang，defer代码块会在函数调用链表中添加一个函数调用，一个defer语句就是在一个普通的函数者方法调用前加上defer。

defer语句正常情况下是在return执行之后执行。当发生宕机时函数正常流程中断，但defer仍然会执，defer执行完毕才意味着函数执行结束。defer的使用可以通过三条规则来学习：

2.3.1 defer声明时实时解析参数

```
// defer声明时，参数就被解析
func demo1() {
```

```
    a := 0
    defer fmt.Println("defer a=", a)
    a++
    fmt.Println("return a=", a)
}
// 输出:
return a= 1
defer a= 0
```

在defer生命时，a的值就被传递到defer语句中，后面即使a变量发生变化，也不会再对defer再产生响

2.3.2 先进后出的执行顺序

```
// 先进后出的执行顺序
func demo2() {
    for i := 0; i < 4; i++ {
        defer fmt.Println(i)
    }
}
//  输出:
3
2
1
0
```

当函数中有多个defer时，执行顺序是和定义顺序相反的。

2.3.3 可以访问函数返回值

```
func demo3() (i int) {
    defer func() {
        i++
    }()
    return 2
}
//  输出
3
```

defer在return语句之后、函数结束之前执行，因此defer可以对return后的变量进行操作。

2.4 异常的处理

2.4.1 处理方式

在一个协程中当出现panic错误，又不希望程序发生宕机（一个协程发生panic，没有recover拦截的，整个程序都会宕机），可以通过以下方式进行处理

```
func demo() {
    defer func() {
        if e := recover(); e != nil {
            fmt.Println(e)
```

```
        }
    }()
    ...
}
```

在函数中使用recover函数捕获协程中发生的错误，recover要定义在defer中才能生效。有几个需要意的点：

```
func demo() {
    defer fmt.Println("函数结束")
    defer func() {
        if e := recover(); e != nil {
            fmt.Println(e)
            // debug.PrintStack()
        }
    }()

    fmt.Println("a")
    fmt.Println("b")
    panic("此处发生panic")
    fmt.Println("c")
    defer fmt.Println("panic后面的panic")
}
// 输出：
a
b
此处发生panic
函数结束
```

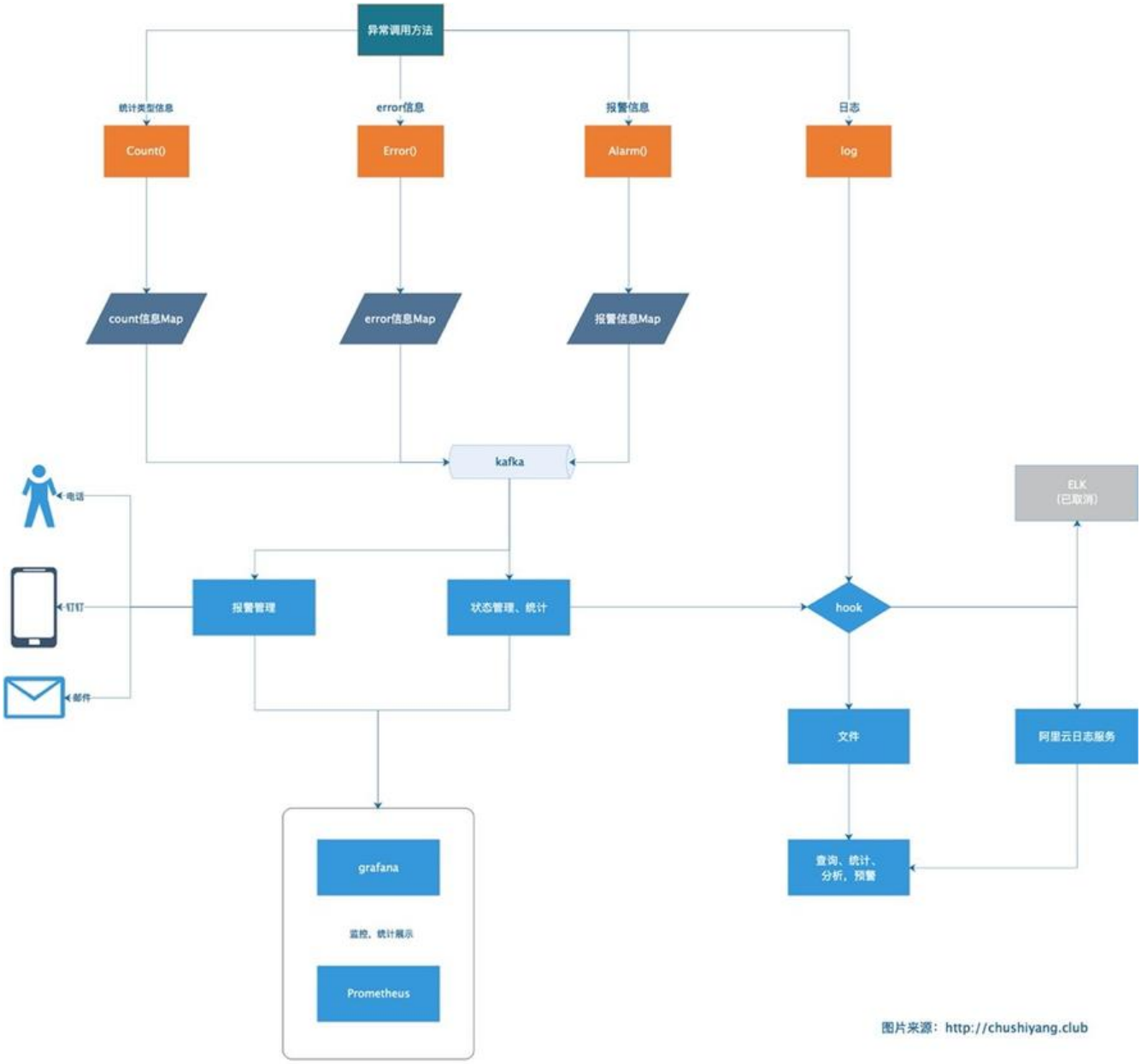
1. 写在panic()后面的语句包括defer的定义都不会被执行到
2. recover()也必须在panic之前定义
3. panic()前定义的defer语句可以正常执行
4. 在recover()后，如果想看到堆栈的错误信息，可以使用 debug.PrintStack()进行打印
5. recover()必须写到defer之后才生效

2.4.2 recover的原理

3. 生产环境下使用metrics系统处理异常

生产环境中异常信息的捕获对于及时发现系统问题有重大的意义。我们期望系统出现重大问题时第一间能够通知到相关人，同时当一般级别的问题频繁出现时也能通知到相关人事后处理，或者是需要对殊信息进行统计，这时候就需要一套完整的系统异常系统。市面上也有很多比较成熟的解决方案，如gkit的异常处理机制，gin中也可以自定义异常处理，gf框架的错误处理。

目前我在使用的一套sdk是参考了go-kit设计和开发的，结构如下图：



1. 项目中引入sdk的概念，按照需要调用Count()完成统计的作用，Error()记录一般异常信息，Alarm(输出需要进行报警的信息

2. 三类信息按照类别存入map中，10秒种向kafka写入一次信息

3. 报警管理和状态管理两个脚本消费kafka中的数据，然后按照设定好的算法进行统计或者报警操作

4. 使用Prometheus和grafana展示信息

5. 其他信息正常写入日志系统中

日志系统最开始使用ELK，后经领导决定改为了阿里云系统，原因是阿里云日志系统中方便进行一些计和查询。

参考资料

Go2 的 error 设计草案：https://go.googlesource.com/proposal/+/master/design/go2draft-error-handling-overview.md

Working with Errors in Go 1.13 :https://blog.golang.org/go1.13-errors