程序中的错误处理（上）：错误返回码和异常捕捉

2017-11-07 陈皓

今天，我们来讨论一下程序中的错误处理。也许你会觉得这个事没什么意思，处理错误的代码并不难写。但你想过没有，要把错误处理写好，并不是件容易的事情。另外，任何一个稳定的程序中都是会有大量的代码在处理错误，所以说，处理错误是程序中一件比较重要的事情。这里，我会用两篇文章来系统地讲一下错误处理的各种方式和相关的实践。

传统的错误检查

首先，我们知道，处理错误最直接的方式是通过错误码，这也是传统的方式，在过程式语言中通常都是用这样的方式处理错误的。比如C语言，基本上来说，其通过函数的返回值标识是否有错，然后通过全局的`errno`变量并配合一个 `errdr` 的数组来告诉你为什么出错。

为什么是这样的设计？道理很简单，除了可以共用一些错误，更重要的是这其实是一种妥协。比如：`read()`, `write()`, `open()` 这些函数的返回值其实是返回有业务逻辑的值。也就是说，这些函数的返回值有两种语义，一种是成功的值，比如 `open()` 返回的文件句柄指针 `FILE*` ，或是错误 `NULL`。这样会导致，调用者并不知道是什么原因出错了，需要去检查 `errno` 来获得出错的原因，从而可以正确地处理错误。

一般而言，这样的错误处理方式在大多数情况下是没什么问题的。但是也有例外的情况，我们来看一下下面这个C语言的函数：

```
int atoi(const char *dr)
```

这个函数是把一个字符串转成整型。但是问题来了，如果一个要传的字符串是非法的（不是数字的格式），如"ABC"或者整型溢出了，那么这个函数应该返回什么呢？出错返回，返回什么数都不合理，因为这会和正常的结果混淆在一起。比如，返回 `0`，那么会和正常的对 `"0"` 字符的返回值完全混淆在一起。这样就无法判断什么是出错的情况，什么时候是正确的情况。你可能会说，是不是要检查一下 `errno`，按道理说应该是要去检查的，但是，我们在C99的规格说明书中可以看到这样的描述——

> 7.20.1
>
> The functions atof, atoi, atol, and atoll need not affect the value of the integer expression errno on an error. If the value of the result cannot be represented, the behavior is undefined.

像`atoi()`, `atof()`, `atol()` 或是 `atoll()` 这样的函数是不会设置 `errno`的，而且，还说了，如果结果无法计算的话，行为是`undefined`。所以，后来，libc又给出了一个新的函数`drtol()`，这个函数在出错的时会设置全局变量`errno` ：

```
long drtol(const char *restrict dr, char **restrict endptr, int base);
```

于是，我们就可以这样使用：

```
long val = drtol(in_dr, &endptr, 10);  //10的意思是10进制

//如果无法转换
if (endptr == dr) {
    fprintf(derr, "No digits were found\n");
    exit(EXIT_FAILURE);
}

//如果整型溢出了
if ((errno == ERANGE && (val == LONG_MAX || val == LONG_MIN)) {
    fprintf(derr, "ERROR: number out of range for LONG\n");
    exit(EXIT_FAILURE);
 }

//如果是其它错误
if (errno != 0 && val == 0) {
```

```
    perror("$rtol");
    exit(EXIT_FAILURE);
}
```

虽然，$rtol() 函数解决了 atoi() 函数的问题，但是我们还是能感觉到不是很舒服和自然。

因为，这种用 返回值 + errno 的错误检查方式会有一些问题：

- 程序员一不小心就会忘记返回值的检查，而造成代码的Bug；
- 函数接口非常不纯洁，正常值和错误值混淆在一起，导致语义有问题。

所以，后来，有一些类库就开始区分这样的事情。比如，Windows的系统调用开始使用 HRESULT 的返回来统一错误的返回值，这样可以明确函数调用时的返回值就是成功还是错误。但这样一来，函数的input和output只能通过函数的参数来完成，于是出现了所谓的 入参 和 出参 这样的区别。然而，这又使得函数接入中参数的语义变得复杂，一些参数是入参，一些参数是出参，函数接口变得复杂了一些。而且，依然没有解决函数的成功或失败可以被人为忽略的问题。

多返回值

于是，有一些语言通过多返回值来解决这个问题，比如Go语言。Go语言的很多函数都会返回 result, err 两个值，于是：

- 参数上基本上就是入参，而返回接口把结果和错误分离，这样使得函数的接口语义清晰；
- 而且，Go语言中的错误参数如果要忽略，需要显式地忽略，用 _ 这样的变量来忽略；
- 另外，因为返回的 error 是个接口（其中只有一个方法 Error()，返回一个 $ring ），于是你在扩展自定义的错误处理。

比如下面这个JSon语法的错误：

```
type SyntaxError $ruct {
    msg    $ring // description of error
    Offset int64  // error occurred after reading Offset bytes
}

func (e *SyntaxError) Error() $ring { return e.msg }
```

在使用上会是这个样子：

```
if err := dec.Decode(&val); err != nil {
    if serr, ok := err.(*json.SyntaxError); ok {
        line, col := findLine(f, serr.Offset)
        return fmt.Errorf("%s:%d:%d: %v", f.Name(), line, col, err)
    }
    return err
}
```

上面这个示例来自Go的官方文档 《Error Handling and Go》

多说一句，如果一个函数返回了多个不同类型的 error，你也可以用下面这样的方式：

```
if err != nil {
    switch err.(type) {
        case *json.SyntaxError:
            ...
        case *ZeroDivisionError:
            ...
        case *NullPointerError:
            ...
        default:
            ...
    }
}
```

但即便像Go这样的语言能让错误处理语义更清楚，而且还有可扩展性，也有其问题。如果写过一段时间的Go语言，你就会明白其中的痛苦—— if err != nil 这样的语句简直是写到吐，只能在IDE中定义一个自动写这段代码的快捷键……而且，正常的逻辑代码会被大量的错误处理打得比较凌乱。

资源清理

程序出错时需要对已分配的一些资源做清理，在传统的玩法下，每一步的错误都要去清理前面已分配好的资源。于是就出现了 goto fail 这样的错误处理模式。如下所示：

```
#define FREE(p) if(p) { \
                    free(p); \
                    p = NULL; \
                }
main()
{
    char *fname=NULL, *lname=NULL, *mname=NULL;
    fname = ( char* ) calloc ( 20, sizeof(char) );
    if ( fname == NULL ){
        goto fail;
    }
    lname = ( char* ) calloc ( 20, sizeof(char) );
    if ( lname == NULL ){
```

```
        goto fail;
    }
    mname = ( char* ) calloc ( 20, sizeof(char) );
    if ( mname == NULL ){
        goto fail;
    }

    ......


fail:
    FREE(fname);
    FREE(lname);
    FREE(mname);
    ReportError(ERR_NO_MEMORY);
}
```

这样的处理方式虽然可以，但是会有潜在的问题。最主要的一个问题就是——你不能在中间的代码中有 `return` 语句，因为你需要清理资源。在维护这样的代码时需要非常小心，因为一不注意就会导致代码有资源泄漏的问题。

于是，C++的RAII（Resource Acquisition Is Initialization）机制使用面向对象的特性可以容易地处理这个事情。RAII其实使用C++类的机制，在构造函数中分配资源，在析构函数中释放资源。下面看个例子。

我们先看一个不好的示例：

```
std::mutex m;

void bad()
{
    m.lock();                   // 请求互斥
    f();                        // 若f()抛异常，则互斥绝不被释放
    if(!everything_ok()) return; // 提早返回，互斥绝不被释放
    m.unlock();                 // 若bad()抵达此语句，互斥才被释放
}
```

上面这个例子，在函数的第三条语句中提前返回了，导致 `m.unlock()` 没有被调用，这样会引起死锁问题。我们来看一下用RAII的方式是怎样解决这个问题的。

```
//首先，先声明一个RAII类，注意其中的构造函数和析构函数
class LockGuard {
public:
    LockGuard(std::mutex &m):_m(m) { m.lock(); }
    ~LockGuard() { m. unlock(); }
private:
    std::mutex& _m;
}


//然后，我们来看一下，怎样使用的
void good()
{
    LockGuard lg(m);                // RAII类：构造时，互斥量请求加锁
    f();                            // 若f()抛异常，则释放互斥
    if(!everything_ok()) return;    // 提早返回，LockGuard析构时，互斥量被释放
}                                   // 若good()正常返回，则释放互斥
```

在Go语言中，使用defer关键字也可以做到这样的效果。参看下面的示例：

```
func Close(c io.Closer) {
    err := c.Close()
    if err != nil {
        log.Fatal(err)
    }
}

func main() {
    r, err := Open("a")
    if err != nil {
        log.Fatalf("error opening 'a'\n")
    }
    defer Close(r) // 使用defer关键字在函数退出时关闭文件。

    r, err = Open("b")
    if err != nil {
        log.Fatalf("error opening 'b'\n")
    }
    defer Close(r) // 使用defer关键字在函数退出时关闭文件。
```

```
}
```

从上面这三个例子来看，不同语言的错误处理，你喜欢哪个？就代码的易读和干净而言，我更喜欢C++的RAII模式，然后是Go的defer模式，最后才是C语言的goto fail模式。

异常捕捉处理

上面，我们讲了错误检查和程序出错后对资源的清理这两个事。能把这个事做得比较好的其实是 `try - catch - finally` 这个编程模式。

```
try {
    ... // 正常的业务代码
} catch (Exception1 e) {
    ... // 处理异常 Exception1 的代码
} catch (Exception2 e) {
    ... // 处理异常 Exception2 的代码
} finally {
    ... // 资源清理的代码
}
```

把正常的代码、错误处理的代码、资源清理的代码分门别类，看上去非常干净。

有一些人明确表示不喜欢 `try - catch` 这种错误处理方式，比如著名的 Joel Spolsky。

但是，我想说一下，`try - catch - finally` 这样的异常处理方式有如下一些好处。

- 函数接口在input（参数）和output（返回值）以及错误处理的语义是比较清楚的。
- 正常逻辑的代码可以与错误处理和资源清理的代码分开，提高了代码的可读性。
- 异常不能被忽略（如果要忽略也需要catch住，这是显式忽略）。
- 在面向对象的语言中（如Java），异常是个对象，所以，可以实现多态式的catch。
- 与状态返回码相比，异常捕捉有一个显著的好处是，函数可以嵌套调用，或是链式调用。比如：`int x = add(a, div(b,c));` 或 `Pizza p = PizzaBuilder().SetSize(sz)□SetPrice(p)...;` 在需要返回码的情况下，这事儿有点难做。

当然，你可能会觉得异常捕捉对程序的性能是有影响的，这句话也对也不对。原因是这样的。

- 异常捕捉的确是对性能有影响的，那是因为一旦异常被抛出，函数也就跟着return了。而程序在执行需要处理函数栈上的上下文，这会导致性能变得很慢，尤其是函数栈比较深的时候。
- 但从另一方面来说，异常的抛出基本上表明程序的错误。程序在绝大多数情况下，应该是在没有异常的情况下运行的，所以，有异常的情况应该是少数的情况，不会影响正常处理的性能问题。

总体而言，我还是觉得 `try - catch - finally` 这样的方式是很不错的。而且这个方式比返回错误码在诸多方面都更好。

但是，`try - catch - finally` 有个致命的问题，那就是在异步运行的世界里的问题。try语句块里的函数运行在另外一个线程中，其中抛出的异常无法在调用者的这个线程中被捕捉。这个问题就比较大了。

错误返回码 vs 异常捕捉

是返回错误状态，还是用异常捕捉的方式处理错误，可能是一个很容易引发争论的问题。有人说，对于一些偏底层的错误，比如：空指针、内存不足等，可以使用返回错误状态码的方式，而对于一些上层的业务逻辑方面的错误，可以使用异常捕捉。这么说有一定道理，因为偏底层的函数可能用得更多一些。但是我并不这么认为。

前面也比较过两者的优缺点，总体而言，似乎异常捕捉的优势更多一些。但是，我觉得应该从场景上来讨论这个事才是正确的姿势。

要讨论场景，我们需要先把要处理的错误分好类别，这样有利于简化问题。

因为，错误其实是很多的，不同的错误需要有不同的处理方式。但错误处理是有一些通用的规则的。为了讲清这个事，我们需要把错误来分个类。我个人觉得错误可以被分成三个大类。

- 资源的错误。当我们的代码去请求一些资源时导致的错误，比如打开一个没有权限的文件，写文件时出现的写错误，发送文件到网络端发现网络故障的错误，等等。这一类错误属于程序运行环境的问题。对于这类错误，有的，我们可以处理，有的我们则无法处理。比如，内存耗尽、栈溢出或是一些程序运行时关键性资源不能满足时，我们只能停止运行，甚至退出整个程序。
- 程序的错误。比如：空指针、非法参数等。这类是我们自己程序的错误，我们要记录下来，写入日志，最好触发监控系统报警。
- 用户的错误。比如：Bad Request、Bad Format等这类由用户的不合法输入带来的错误。这类错误基本上是在用户的API层上出现的问题。比如，解析一个XML或JSon文件，或是用户输入的字段不合法之类的。对于这类问题，我们需要向用户端报错，让用户自己处理修正他们的输入或操作。然后，我们正常执行，但是需要做统计，统计相应的错误率，这样有利我们改善软件或是侦测是否有恶意的用户请求。

我们可以看到，这三类错误中，有些是我们希望杜绝发生的，比如程序的Bug，有些则是我们杜绝不了的，比如用户的输入。而对于程序运行环境中的一些错误，则是我们希望可以恢复的。也就是说，我们希望可以通过重试或是妥协的方式来解决这些环境的问题，比如重建网络连接，重新打开一个新的文件。

所以，是不是我们可以这样来在逻辑上分类：

- 对于我们并不期望会发生的事，我们可以使用异常捕捉；
- 对于我们觉得可能会发生的事，使用返回码。

比如，如果你的函数参数传入的对象不是一个null对象，那么，一旦传入后，可以抛异常，因为我们并不期望总是会发生这样的事。而对于一个需要检查用户输入信息是否正确的事，比如：电子邮箱的格式，我们用返回码可能会好一些。所以，对于上面三种错误的种类来说，程序中的错误，可能用异常捕捉会比较合适；用户的错误，用返回码比较合适；而资源类的错误，要分情况，是用异常捕捉还是用返回值，要看这事是不应该出现的，还是经常出现的。

当然，这只是一个大致的实践原则，并不代表所有的事都需要符合这个原则。

除了用错误的分类来判断是否用返回码还是用异常捕捉之外，我们还要从程序设计的角度来考虑哪种情况下使用异常捕捉更好，哪种情况下使用返回码更好。因为异常捕捉在编程上的好处比函数返回值好很多，所以很多使用异常捕捉的代码会更易读也更健壮一些。而返回码容易被忽略，所以，使用返回码的代码需要做良好的测试才能得到更好的质量。

不过，我们也要知道，在某些情况下，你只能使用其中的一个，比如：

- 在C++重载操作符的情况下，你就很难使用错误返回码，只能抛异常；
- 异常捕捉只能在同步的情况下使用，在异步模式下，抛异常这事就不行了，需要通过检查子进程退出码或是回调函数来解决；
- 在分布式的情况下，调用远程服务只能看错误返回码，比如HTTP的返回码。

所以，在大多数情况下，我们会混用这两种报错的方式，有时候，我们还会把异常转成错误码（比如HTTP的RESTful API），也会把错误码转成异常（比如对系统调用的错误）。

总之，“报错的类型”和“错误处理”是紧密相关的，错误处理方法多种多样，而且会在不同的层面上处理错误。有些底层错误就需要自己处理掉（比如：底层模块会自动重建网络连接），而有一些错误需要更上层的业务逻辑来处理（比如：重建网络连接不成功后只能让上层业务来处理怎么办？降级使用本地缓存还是直接报错给用户，等等）。所以，不同的错误类型再加上不同的错误处理会导致我们代码组织层面上的不同，从而会让我们使用不同的方式（也就是说，使用错误码还是异常捕捉主要还是看我们我们的错误处理流程以及代码组织怎么写会更清楚）。

通过学习今天的内容，你是不是已经对如何处理程序中的错误，以及在不同情况下怎样选择错误处理方法，有了一定的认知和理解呢？然而，这些知识和经验仅在同步编程世界中适用。因为在异步编程世界里，被调用的函数是被放到另外一个线程里运行的，所以本文中的两位主角，不管是错误返回码，还是异常捕捉，都难以发挥其威力。那么异步编程世界中是如何做错误处理的呢？我们将在下篇文章中讨论。同时，还会讲讲我在实战中总结出来的错误处理最佳实践。

Harry

2017-11-07

隐式异常也不要显示的忽略吧，至少记录一条日志，不然会造成异常黑洞。之前帮同事找一个线上小概率问题，所有的日志和异常检测都抓不到，最后对全部进程一起strace，才发现是他catch了异常，啥都没做就return了…

我们后来揍了丫的

majun

2017-11-07

期望听您对进程、线程、多进程、多线程的讲解，谢谢！

左耳朵

2017-11-21

@ stone扎西华丹，是我给极客时间出难题了，我的好些文章里有好多代码，有的会有好些图片，有的会有好些数学公式，这类文章不但都非常难配上语音，而且可能在手机端的排版都会很有问题，难为极客时间的编辑、产品和技术了，这个还望能理解了（后面的文章这样的事你会看到很多）

xfly

2017-11-07

对于上面三种错误的种类来说，程序中的错误，可能用异常捕捉会比较合适；用户的错误，用返回码比较合适；而资源类的错误，要分情况，是用异常捕捉还是用返回值，要看这事是不应该出现的，还是经常出现的。
——
这三种分类和处置方式比较赞同。
但实际在多人协作项目，或大型项目中多方技术人员要在这个层面理解达成一致不是那么容易的事情。如果依赖于解决架构负债，架构升级来优化，驱动力又似乎不足。

yun

2018-03-27

>异常捕捉的确是对性能有影响的，那是因为一旦异常被抛出，函数也就跟着 return 了。而程序在执行需要处理函数栈上的上下文，这会导致性能变得很慢，尤其是函数栈比较深的时候

异常抛出和不抛出，函数栈的深度应该差不多吧？函数栈的上下文会有啥不同？

郎哲

2017-11-07

不用的语言不同的方式处理错误。Elang 虽有catch，几乎用不到，直接返回值。假设传进来的一定正确，霸道一点不正确请修改正确再传，错误非常容易定位用了catch反而多余隐藏bug，勿隐藏应今早发现今早解决。Go 返回error嵌套多了确实蛋疼不得多写好多if，Java 离了try catch 活不下去。

李帅龙

2017-11-07

还有一种rust的方式

stone扎西华丹

2017-11-13

货不对板，我们订的时候，看到的可是一个音频产品。

皮特尔

2017-11-07

赞！迫不及待要看下一篇了。

周孟

2018-06-19

对于返回错误码的方式是否定义结构体或是类同时包含错误码和错误消息会更好一些，特别是一些业务验证或输入问题上

Wilson_qqs

2018-03-08

赞��

Shaoyao·瑶

2017-11-07

早上好～