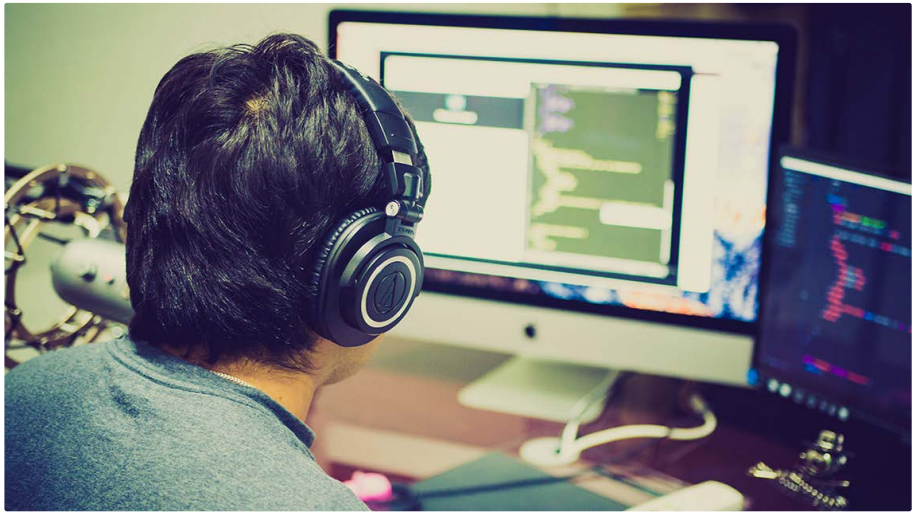
程序中的错误处理（下）：异步编程和最佳实践

2017-11-09 陈皓



上一篇文章中，我们讨论了错误返回码和异常捕捉，以及在不同情况下该如何选择和使用。本文中会接着讲两个有趣话题：异步编程世界里的错误处理方法，以及我在实战中总结出来的错误处理最佳实践。

异步编程世界里的错误处理

在异步编程的世界里，因为被调用的函数是被放到了另外一个线程里运行，这将导致：

- 无法使用返回码。因为函数在被异步运行中，所谓的返回只是把处理权交给下一条指令，而不是把函数运行完的结果返回。所以，函数返回的语义完全变了，返回码也没有用了。
- 无法使用抛异常的方式。因为除了上述的函数立马返回的原因之外，抛出的异常也在另外一个线程中，不同线程中的栈是完全不一样的，所以主线程的 `catch` 完全看不到另外一个线程中的异常。

对此，在异步编程的世界里，我们也会有好几种处理错误的方法，最常用的就是 `callback` 方式。在做异步请求的时候，注册几个 `OnSuccess()`、 `OnFailure()` 这样的函数，让在另一个线程中运行的异步代码来回调过来。

JavaScript异步编程的错误处理

比如，下面这个JavaScript示例：

```
function successCallback(result) {
  console.log("It succeeded with " + result);
}

function failureCallback(error) {
  console.log("It failed with " + error);
}

doSomething(successCallback, failureCallback);
```

通过注册错误处理的回调函数，让异步执行的函数在出错的时候，调用被注册进来的错误处理函数，这样的方式比较好地解决了程序的错误处理。而出错的语义从返回码、异常捕捉到了直接耦合错误出处函数的样子，挺好的。

但是， 如果我们需要把几个异步函数顺序执行的话（异步程序中，程序执行的顺序是不可预测的、也是不确定的，而有时候，函数被调用的上下文是有相互依赖的，所以，我们希望它们能按一定的顺序处理），就会出现了所谓的Callback Hell的问题。如下所示：

```
doSomething(function(result) {
  doSomethingElse(result, function(newResult) {
    doThirdThing(newResult, function(finalResult) {
      console.log('Got the final result: ' + finalResult);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

而这样层层嵌套中需要注册的错误处理函数也有可能是完全不一样的，而且会导致代码非常混乱，难以阅读和维护。

所以，一般来说，在异步编程的实践里，我们会用Promise模式来处理。如下所示（箭头表达式）：

```
doSomething()
.then(result => doSomethingElse(result))
```

```
.then(newResult => doThirdThing(newResult))
.then(finalResult => {
  console.log(`Got the final result: ${finalResult}`);
}).catch(failureCallback);
```

上面代码中的 `then()` 和 `catch()` 方法就是Promise对象的方法，`then()`方法可以把各个异步的函数给串联起来，而`catch()` 方法则是出错的处理。

看到上面的那个级联式的调用方式，这就要我们的 `doSomething()` 函数返回Promise对象，下面是这个函数的相关代码示例：

比如：

```
function doSomething() {
    let promise = new Promise();
    let xhr = new XMLHttpRequest();
    xhr.open('GET', 'http://coolshell.cn/....', true);

    xhr.onload = function (e) {
        if (this.status === 200) {
            results = JSON.parse(this.responseText);
            promise.resolve(results); //成功时，调用resolve()方法
        }
    };

    xhr.onerror = function (e) {
        promise.reject(e); //失败时，调用reject()方法
    };

    xhr.send();
    return promise;
}
```

从上面的代码示例中，我们可以看到，如果成功了，要调用 `Promise.resolve()` 方法，这样Promise对象会继续调用下一个 `then()`。如果出错了就调用 `Promise.reject()` 方法，这样就会忽略后面的 `then()` 直到 `catch()` 方法。

我们可以看到 `Promise.reject()` 就像是抛异常一样。这个编程模式让我们的代码组织方便了很多。

另外，多说一句，Promise还可以同时等待两个不同的异步方法。比如下面的代码所展示的方式：

```
promise1 = doSomething();
promise2 = doSomethingElse();
Promise.when(promise1, promise2).then( function (result1, result2) {
    ... //处理 result1 和 result2 的代码
}, handleError);
```

在ECMAScript 2017的标准中，我们可以使用async/await这两个关键字来取代Promise对象，这样可以让我们的代码更易读。

比如下面的代码示例：

```
async function foo() {
  try {
    let result = await doSomething();
    let newResult = await doSomethingElse(result);
    let finalResult = await doThirdThing(newResult);
    console.log(`Got the final result: ${finalResult}`);
  } catch(error) {
    failureCallback(error);
  }
}
```

如果在函数定义之前使用了 `async` 关键字，就可以在函数内使用 `await`。 当在 `await` 某个 `Promise` 时，函数暂停执行，直至该 `Promise` 产生结果，并且暂停并不会阻塞主线程。 如果 `Promise resolve`，则会返回值。 如果 `Promise reject`，则会抛出拒绝的值。而我们的异步代码完全可以放在一个 `try - catch` 语句块内，在有语言支持了以后，我们又可以使用 `try - catch` 语句块了。

下面我们来看一下，一个pipeline的代码。所谓pipeline就是把一串函数给编排起来，从而形成更为强大的功能。这个玩法是函数式编程中经常用到的方法。

比如，下面这个pipeline的代码（注意，其上使用了 `reduce()` 函数）：

```
[func1, func2].reduce((p, f) => p.then(f), Promise.resolve());
```

其等同于：

```
Promise.resolve().then(func1).then(func2);
```

我们可以抽象成

```
let applyAsync = (acc,val) => acc.then(val);
let composeAsync = (...funcs) => x => funcs.reduce(applyAsync, Promise.resolve(x));
```

于是，可以这样使用：

```
let transformData = composeAsync(func1, asyncFunc1, asyncFunc2, func2);
transformData(data);
```

但是，在ECMAScript 2017的 async/await 语法糖下，这事儿就变得简单了。

```
for (let f of [func1, func2]) {
  await f();
}
```

Java异步编程的Promise模式

在Java中，在JDK 1.8里也引入了类JavaScript的玩法 —— CompletableFuture。这个类提供了大量的异步编程中Promise的各种方式。下面例举几个。

链式处理：

```
CompletableFuture.supplyAsync(this::findReceiver)
                .thenApply(this::sendMsg)
                .thenAccept(this::notify);
```

上面的这个链式处理和JavaScript中的then()方法很像，其中的 supplyAsync() 表示执行一个异步方法，而 thenApply() 表示执行成功后再串联另外一个异步方法，最后是 thenAccept() 来处理最终结果。

下面这个例子是，合并两个异步函数的结果成一个的示例：

```
String result = CompletableFuture.supplyAsync(() -> {
        return "hello";
    }).thenCombine(CompletableFuture.supplyAsync(() -> {
        return "world";
    }), (s1, s2) -> s1 + " " + s2).join());
System.out.println(result);
```

接下来，我们再来看一下，Java这个类相关的异常处理：

```
CompletableFuture.supplyAsync(Integer::parseInt) //输入: "ILLEGAL"
                .thenApply(r -> r * 2 * Math.PI)
                .thenApply(s -> "apply>> " + s)
                .exceptionally(ex -> "Error: " + ex.getMessage());
```

我们要注意到上面代码里的 exceptionally() 方法，这个和JavaScript Promise中的 catch() 方法相似。

运行上面的代码，会出现如下输出：

```
Error: java.lang.NumberFormatException: For input string: "ILLEGAL"
```

也可以这样：

```
CompletableFuture.supplyAsync(Integer::parseInt) // 输入: "ILLEGAL"
                .thenApply(r -> r * 2 * Math.PI)
                .thenApply(s -> "apply>> " + s)
                .handle((result, ex) -> {
                    if (result != null) {
                        return result;
                    } else {
                        return "Error handling: " + ex.getMessage();
                    }
                });
```

上面代码中，你可以看到，其使用了 handle() 方法来处理最终的结果，其中包含了异步函数中的错误处理。

Go语言的Promise

在Go语言中，如果你想实现一个简单的Promise模式，也是可以的。下面的代码纯属示例，只为说明问题。如果你想要更好的代码，可以上GitHub上搜一下Go语言Promise的相关代码库。

首先，先声明一个结构体。其中有三个成员：第一个 wg 用于多线程同步；第二个 res 用于存放执行结果；第三个 err 用于存放相关的错误。

```
type Promise struct {
    wg  sync.WaitGroup
    res string
    err error
```

```
}
```

然后，定义一个初始函数，来初始化Promise对象。其中可以看到，需要把一个函数 f 传进来，然后调用 wg.Add(1) 对waitGroup做加一操作，新开一个Goroutine通过异步去执行用户传入的函数 f()，然后记录这个函数的成功或错误，并把waitGroup做减一操作。

```
func NewPromise(f func() (&ring, error)) *Promise {
    p := &Promise{}
    p.wg.Add(1)
    go func() {
        p.res, p.err = f()
        p.wg.Done()
    }()
    return p
}
```

然后，我们需要定义Promise的Then方法。其中需要传入一个函数，以及一个错误处理的函数。并且调用 wg.Wait() 方法来阻塞（因为之前被wg.Add(1)），一旦上一个方法被调用了 wg.Done()，这个Then方法就会被唤醒。

唤醒的第一件事是，检查一下之前的方法有没有错误。如果有，那么就调用错误处理函数。如果之前成功了，就把之前的结果以参数的方式传入到下一个函数中。

```
func (p *Promise) Then(r func(&ring), e func(error)) (*Promise){
    go func() {
        p.wg.Wait()
        if p.err != nil {
            e(p.err)
            return
        }
        r(p.res)
    }()
    return p
}
```

下面，我们定义一个用于测试的异步方法。这个方面很简单，就是在数数，然后，有一半的机率会出错。

```
func exampleTicker() (&ring, error) {
    for i := 0; i < 3; i++ {
        fmt.Println(i)
        <-time.Tick(time.Second * 1)
    }

    rand.Seed(time.Now().UTC().UnixNano())
    r:=rand.Intn(100)%2
    fmt.Println(r)
    if  r != 0 {
        return "hello, world", nil
    } else {
        return "", fmt.Errorf("error")
    }
}
```

下面，我们来看看我们实现的Go语言Promise是怎么使用的。代码还是比较直观的，我就不做更多的解释了。

```
func main() {
    doneChan := make(chan int)

    var p = NewPromise(exampleTicker)
    p.Then(func(result &ring) { fmt.Println(result); doneChan <- 1 },
            func(err error) { fmt.Println(err); doneChan <-1 })

    <-doneChan
}
```

当然，如果你需要更好的Go语言Promise，可以到GitHub上找，上面好些代码都是实现得很不错的。上面的这个示例，实现得比较简陋，仅仅是为了说明问题。

错误处理的最佳实践

下面是我个人总结的几个错误处理的最佳实践。如果你知道更好的，请一定告诉我。

- 统一分类的错误字典。无论你是使用错误码还是异常捕捉，都需要认真并统一地做好错误的分类。最好是在一个地方定义相关的错误。比如，HTTP的4XX表示客户端有问题，5XX则表示服务端有问题。也就是说，你要建立一个错误字典。
- 同类错误的定义最好是可以扩展的。这一点非常重要，而对于这一点，通过面向对象的继承或是像Go语言那样的接口多态可以很好地做到。这样可以方便地重用已有的代码。
- 定义错误的严重程度。比如，Fatal表示重大错误，Error表示资源或需求得不到满足，Warning表示并不一定是个错误但还是需要引起注意，Info表示不是错误只是一个信息，Debug表示这是给内部开发人员用于调试程序的。
- 错误日志的输出最好使用错误码，而不是错误信息。打印错误日志的时候，除了要用统一的格式，最好不要用错误信息，而使用相应的错误码，错误码不一定是数字，也可以是一个

能从错误字典里找到的一个唯一的可以让人读懂的关键字。这样，会非常有利于日志分析软件进行自动化监控，而不是要从错误信息中做语义分析。比如：HTTP的日志中就会有HTTP的返回码，如：404。但我更推荐使用像PageNotFound这样的标识，这样人和机器都很容易处理。

- 忽略错误最好有日志。不然会给维护带来很大的麻烦。
- 对于同一个地方不停的报错，最好不要都打到日志里。不然这样会导致其它日志被淹没了，也会导致日志文件太大，最好的实践是，打出一个错误以及出现的次数。
- 不要用错误处理逻辑来处理业务逻辑。也就是说，不要使用异常捕捉这样的方式来处理业务逻辑，而是应该用条件判断。如果一个逻辑控制可以用if - else清楚地表达，非常不建议使用异常方式处理。异常捕捉是用来处理不期望发生的事的，而错误码则用来处理可能会发生的事。
- 对于同类的错误处理，用一样的模式。比如，对于null对象的错误，要么都用返回null，加上条件检查的模式，要么都用抛NullPointerException的方式处理。不要混用，这样有助于代码规范。
- 尽可能在错误发生的地方处理错误。因为这样会让调用者变得更简单。
- 向上尽可能地返回原始的错误。如果一定要把错误返回到更高层去处理，那么，应该返回原始的错误，而不是重新发明一个错误。
- 处理错误时，总是要清理已分配的资源。这点非常关键，使用RAII技术，或是try-catch-finally，或是Go的defer都可以容易地做到。
- 不推荐在循环体里处理错误。这里说的更多的情况是对于try-catch这种情况，对于绝大多数的情况你不需要这样做。最好把整个循环体外放在try语句块内，而在外面做catch。
- 不要把大量的代码都放在一个try语句块内。一个try语句块内的语句应该是完成一个简单单一的事情。
- 为你的错误定义提供清楚的文档以及每种错误的代码示例。如果你是做RESTful API方面的，使用Swagger会帮你很容易搞定这个事。
- 对于异步的方式，推荐使用Promise模式处理错误。对于这一点，JavaScript中有很好的实践。
- 对于分布式的系统，推荐使用APM相关的软件。尤其是使用Zipkin这样的服务调用跟踪的分析来关联错误。

好了。关于程序中的错误处理，我主要总结了这些。如果你有更好的想法和经验，欢迎来跟我交流。

sonnyching

2017-11-12

很喜欢这种方式去分析问题，我都是单从java的角度去看待对于一个问题的处理。耗子叔的文章都是从不同语言的处理方式，然后得出一个稍微通用的解决，或者给出不同处理方式的优劣。看来有时间的确应该多学几门不同的语言，看问题的视野完全不一样啊☺☺为了看懂go的代码，我还特地去简单看了下go的语法。貌似耗子叔很喜欢go(⊙o⊙)哇哈哈哈

杜小琨

2017-12-15

向上尽可能地返回原始的错误。如果一定要把错误返回到更高层去处理，那么，应该返回原始的错误，而不是重新发明一个错误。

这一条如果要作为通用规则我觉得略有争议。我的观点是错误(异常)同样要考虑封装性。

需要考虑到这个错误原因是否应该上层感知。比如在在存储一张图片时，上层进行抽象业务逻辑的单元并不期望知道底层具体存储方式的存在，你用本地磁盘也好，hdfs也罢，这些原始错误暴露到业务流程中反而会让业务茫然不知所措。其实上层业务只需要知道错误发生(图片保存失败)即可，并不关心具体错误的原因。

何小事儿

2017-11-13

老师，如何解决技术团队因需求不断，业务开发繁忙而导致的技术水平没有显著提高机会的问题?

莫名的冲动

2017-11-10

耗子叔，挺喜欢你的文章，篇篇都是经验之谈，而且面向的受众技术人员也是最广的，既不是那种网上一搜就能找到的技术罗列，也不是普通技术人员平常都难遇到的高深技术问题解析，角度和深度都刚刚好！赞！

左耳朵

2017-11-21

@ 何小事儿 后面的《时间管理》会有一些，敬请关注

左耳朵

2017-11-21

@ Chris 哪些地方看不懂?

Chris

2017-11-10

老师好，你的文章很有质量，很好，不过我是属于初级，能否谈谈对于文章内可能看不懂的地方，自己应该如何解决这个问题？希望你能指导一下，谢谢！

渔夫

2018-07-04

异常应分层分类，需要识别分层边界和领域边界

宝爷

2018-06-21

很好奇多个async/await异步调用的try catch怎么实现的，我能想到的就是编译器自动在多个异步的代码处补充try catch，使用同一个异常处理

刘強

2018-06-14

骨灰级程序员，名副其实啊

MarksGui

2018-06-04

想不到一个错误处理能有这么多学问！ 长见识了！ 果然程序是严谨的学问，需要用一生去学习！

站住！我的马甲

2018-04-11

耗子叔，对于dubbo调用怎么能够更好的返回它的异常信息

浪子

2018-04-10

开个脑洞，对于"不要用错误处理逻辑来处理业务逻辑"这条规范，在幂等处理中的用数据库唯一性约束解决是不是恰好是反例?

扫地僧的功夫梦

2018-04-02

实现Callable接口是不是算有返回呢?

laofengfenglao

2018-02-11

耗子叔，请教您一个和异步沾点边的具体问题，最近我在用javaweb做一个图片上传接口，应用服务器是tomcat，用的是NIO，客户端用字符串的形式把base64串给服务端（大约100KB），服务端再传到swift容器，压测时发现瓶颈在读取字符串上。一般是怎么做可以实现这种客户端高效图片上传(我们现在并发500测试时时候吞吐率连每秒50都不到，平均响应时间10秒左右)

云学

2018-01-13

语言大神，能否总结下您接触的各种语言的特色和缺陷，比如该语言的最佳适用场景，吐槽下该语言不好的地方，比如c++的RAII，javascript的promise就是他们的特色

作者回复

2018-01-16

吐槽，可以啊。过两天写一篇

Ray

2018-01-09

有时候会写出在try块里面出现try块的情况，请问这种写法应该避免还是得看具体的业务?

milley

2017-11-30

耗子叔举例讲解了不同语言不同用法，最终得出实践。虽然每种都是略懂，但是看完觉得很有启发。