Result<T> 模式: 错误处理的未来方向

黄嘉诚

2024 年 12 月 11 日

摘要

因为程序中错误的多发性和程序所追求的稳定性，一个好的错误处理机制非常重要。许多语言如Lisp, c, c++都提供了各种错误处理方式。但是, 他们的处理方式 (如try-catch-finally) 都各有问题。本文分析了编程语言进化之路上各种错误处理方式的弊端, 并展现了新的语言在新的编程范式和技术下对于错误处理机制的新理解– Result<T>, 探讨了这种新模式的相较于try-catch-finally等方法的进步之处。

1 引言

错误处理是指在软件开发中识别、管理和响应程序运行过程中可能出现的异常或意外行为的机制。它是为了确保程序在面对不可预期的错误时, 能够以一种可控、明确的方式继续运行或安全退出，而不会导致崩溃、数据丢失或错误结果。一个良好的程序需要在各种条件下都能以恰到的方式完成任务, 因此需要判断各种错误如网络异常, 内存不足, 用户输入错误等来执行不同的逻辑进行提示或是其他渠道的信息获取. 越稳健和复杂的程序, 越需要一种良好的错误处理模式

找到错误的根源很困难。第一，随着现代程序的越发复杂, 内部函数的调用嵌套更多。在报错时, 程序员必须跟踪复杂调用栈, 来找到出错的根源。而在多人项目中, 代码内部实现往往是未知的, 使得发现错误的位置和原因更加困难。第二，在某些特殊情形如异步代码和递归代码中, 因为各种调用的顺序复杂或未知, 调试找到错误的原因更加困难。

c，c++推崇的try-catch-finally的模式确实极大的方便了错误的处理, 然而随着时代的进步和新的编程模式的崛起存在着很多不足 (简要解释

不足)。而在更加年轻的现代语言里 (如 Rust), 新的模式代替了这种语法–`Result<T>` 泛型, 对错误和正确返回一起打包, 使得正确处理的逻辑和错误处理的逻辑更加紧密, 语法上更为优雅。本文将在第 3 节介绍`Result`的实现, 并更加详细地介绍这样的实现带来的`Result<T>`的优点。

2 错误处理机制的进化之路

2.1 错误处理的诞生

早期语言`Lisp` 在设计时就发明了一系列复杂的错误处理 api, 如`assert`, `handler`, `case`等等, 错误处理方式复杂。而后来的`c`语言不加任何错误处理, 利用`return 0;`代表正确处理,`return -1;`代表程序报错等方式。然而过于简陋, 通过设定好的数字代表错误类型很难获得其他信息, 而且使得代码多了很多莫名其妙的魔法数字, 难以读懂。

2.2 try-catch

作为立志于补充和扩展`c`语言的`c++`, 作者开创了一套新的错误处理机制: `try-catch-finally`, 利用`try`块包裹会出错的代码, 然后在`catch`中进行了错误处理, 并在`finally` 中进行资源释放之类的操作。它规范错误处理的行为, 让错误处理和正常运行逻辑直接分离, 同时简化了错误的抛出行为, 我们可以在任何时候任何位置直接`throw error`, 而在任何时候任何位置进行捕获, 这极大的方便了错误的处理, 也被后续的`Python`,`Java`等语言学习。

然而, 带来方便的同时, 它也有很多不足: 一, 随意的抛出和捕获使得上层代码和下层代码完全无法预测相互的行为。二, 错误和正确结果的逻辑被割裂, 使得构建整个项目的逻辑会更加复杂。第三, 程序并不是完全的错误和正确逻辑组合, 在错误逻辑里我们可以也包含正确结果。并且随着协程 (程序内部控制多任务的微线程) 等新型概念的出现, 这种错误处理机制也限制了他们的发挥。

2.3 新机制的初步探索

21 世纪出现的`Go`语言作为异步优先的语言, 它在设计之处就考虑到了错误处理的不便, 它再一次选择了将错误处理和正确代码放在一个步骤里进行处理的方式: 允许代码返回两个值: 而获取结果的时候只需要

`result, err := divide(a, b)`。他昭示了一种思想: 让错误处理平凡化, 正确和错误的代码归一化。缺点是: 这需要语法上的支持; 将结果和错误一起暴露了，不利于封装。因此，我们还需要更进一步。

3 Result<T> 的诞生

3.1 新的错误处理机制所需要的优点

我们知道了旧有范式的很多缺点, 我们来总结一下新的范式所需要改进的地方:

1. 新的错误处理机制应该更加简洁明了, 甚至不需要语法级的支持
2. 我们想要让恰当的错误处理者处理错误, 而不是随意的处理, 这需要一种约束, 也就是说我们要处理完错误逻辑才能获取值。
3. 我们可以高度自定义以适应各种需求。

基于以上这种理念,`Result<T>`诞生了。

3.2 Result<T> 的实现

我们先来看使用`Kotlin`语言的一个`Result<T>`的简单实现。

```
sealed class Result<T> {
    data class Success<T>(val data: T) : Result<T>()
    data class Failure<T>(val error: Throwable) : Result<T>()
    // Helper methods
    fun isSuccess(): Boolean = this is Success<T>
    fun isFailure(): Boolean = this is Failure<T>
    fun getOrNull(): T? = if (this is Success<T>) this.data else null
    fun exceptionOrNull(): Throwable? = if (this is Failure<T>)
                                            this.error else null
}
```

这个实现完全不需要语法支持, 在任何支持泛型的语言都可以自己实现, 从而取代现有的错误处理机制, 它的侵入性极小。

3.3 Result<T> 的使用

我们来看这个类如何进行使用:

```
    fun divide(a: Int, b: Int): Result<Int> {
    return if (b == 0) {
        Result.Failure(IllegalArgumentException(
            "Division by zero is not allowed"))
    } else {
        Result.Success(a / b)
    }
}
fun main() {
    val result1 = divide(a, b)
    // 处理情况
    when (result1) {
        is Result.Success -> println(
                        "Result: ${result1.data}")
        is Result.Failure -> println(
                        "Error: ${result1.error.message}")
    }
}
```

可以看到, 和Go语言,Result<T>将正确和错误信息封装了。

3.4 Result<T> 的进步之处

Result<T>完美的符合了上述的要求，第一，它让适当的人进行错误处理。Result<T>作为一个载体, 同时承载了错误和返回值两个信息。但是和Go不同的是, 如果想要获取里面的信息, 我们必须对错误进行处理, 才能把他转为success的子类型获取到信息。而对于不需要这个信息的中间函数函数来说, 这个类型就是个黑箱子, 我们无需关注内部成功与否, 而想要获得值, 我们必须解包, 也就是对错误进行处理。这样,try-catch-finally的逻辑链条被成功合并到了正常的代码里, 并且要求使用返回值的函数处理错误, 确保了错误不会始终无人问津。

第二，这种侵入性小的实现方法让新的技术比如协程也可以更好的进

行错误处理, 而对于之后可能的新架构, 注意到这种实现完全没有任何新的语法设定, 只需要改变返回值的类型就. 可以立刻加入的新的架构中。

第三，这种Result<T> 类有高度的自定义化程度, 完全可以依据业务类型进行拓展, 本文的实现就有两个工具方法。而实际业务中，加入更多错误信息, 加入一些常见的错误逻辑处理, 甚至在Result 内容设置监听器, 在某些特定类型的错误上报给一些记录器或是一些全局的逻辑链条都可以自由做到, 比如 gui 程序里的网络异常错误。

这种高度自定义化, 非侵占性且具有责任制的错误处理方式极大地解决了错误处理的很多弊端, 让错误处理真正的被代码所掌控, 而不是成为一种语言内部的原生机制。

4 未来错误处理机制的展望

必须承认,Result<T>也有很多不足, 比如层层嵌套的时候我们也要层层处理Result<T>, 这相当于把try-catch的报错堆栈进行的一步步处理。但从这个角度, 这样的弊端只不过是由于错误处理本身导致的, try-catch只不过给了开发者一种方便之处, 但是在报错的时候有变成了拖累。而且Result<T> 的高度自定义化也可以构造很多工具方法帮助扁平化嵌套的程序或者堆内部复杂的错误链条进行判断和整理，使得很多追踪和缘由分析自动化，方便调试快速找到重点。

而未来, 我们或许考虑能对于层层嵌套的错误处理做出真正的简化, 同时对于错误处理有更细粒度的方法。这需要新的方法和理论的出现。