对编程语言中错误处理机制的思考

作者：宇宙之一粟

2023-06-07 。中国香港
本文字数：4174 字。阅读需约 14 分钟

> 似乎没有一种语言能正确处理错误

错误处理是编程的一个基本技能。除非你写的是“hello world”，否则你将需要处理代码中的错误。在这篇文章中，我将讨论一些各种编程语言使用的最常见的方法。

返回错误码

这是最传统的策略之一：如果一个函数可能失败，它可以简单地返回一个错误代码-通常是一个负数，或 `null` 。这在 C 中非常常见，例如：

复制代码

```c
FILE* fp = fopen("file.txt" , "w");
if (!fp) {
  // some error occurred
}
```

这种方法非常简单，无论是实现还是理解起来。它的执行效率也非常高，因为它只涉及一个带有返回值的标准函数调用，不需要运行时支持或分配。然而，它有几个缺点：

- 函数的使用者很容易忘记错误处理。例如，C 中的 `printf` 可能会失败，但我没有看到很多程序检查它的返回代码！
- 在调用堆栈中传递错误是很烦人的，特别是当你的代码必须处理多个不同的失败时（打开一个文件，向它写入，从另一个文件阅读……）

- 除非你的编程语言支持多个返回值，否则如果你必须返回一个有效值或一个错误，这是很烦人的。这导致 C 和 C++ 中的许多函数必须将“sucess”返回值的存储作为一个指针传递，该指针将由函数填充，这样做：

复制代码

```c
my_struct *success_result;
int error_code = my_function(&success_result);
if (!error_code) {
  // can use success_result
}
```

Go 语言选择了这种方法来处理错误。然而，由于 Go 语言允许一个函数有多个返回值，这种模式变得更符合人体工程学--而且非常常见：

复制代码

```go
user, err = FindUser(username)
if err != nil {
    return err
}
```

```
}
```

该模式的 Go 变体简单、有效，并将错误传递给调用者。另一方面，我觉得它相当重复，有点分散了对实际业务逻辑的注意力。

Exceptions 异常

异常可能是最常用的错误处理模式。`try/catch/finally` 方法工作得很好，使用起来也很简单。异常在 90 年代和 2000 年代变得非常流行，并被许多语言采用，如 Java，C# 或 Python。

与错误代码相比，异常有一些优点：

- 它们自然会导致 "快乐路径 "和错误处理路径之间的分离
- 它们将自动通过调用栈冒泡
- 而且你不能忘记处理错误

然而，它们也有一些缺点：它们需要一些特定的运行时支持，并且通常会产生相当大的性能开销。此外，更重要的是，它们具有“深远”的影响——异常可能会被某些代码抛出并被调用堆栈中非常遥远的异常处理程序捕获，从而影响简洁度。

此外，仅通过查看函数的签名，并不清楚函数是否会抛出任何异常。

C++试图用 `throws` 原因来解决这个问题，这个原因很少使用，以至于它最终在 C++17 中被弃用，并在 C++20 中被删除。从那以后，它试图引入 `noexcept` 。

Java 曾尝试使用“检查异常”，即你必须声明作为签名的一部分的异常-但是这种方法被认为是失败的，像 Spring 这样的现代框架只使用“运行时异常”，而像 Kotlin 这样的 JVM 语言完全摆脱了这个概念。最后，没有好的方法来知道一个方法调用是否会抛出任何异常，因此您最终会有点混乱。

错误回调

另一种方法，在 JavaScript 领域非常普遍，就是使用回调，当一个函数成功或失败时，回调将被调用。这通常与异步编程相结合，在后台进行 I/O，而不阻塞执行流程。

例如，Node.JS I/O 函数通常会使用两个参数 `(error, result)` 进行回调，例如：

复制代码

```
const fs = require('fs');
fs.readFile('some_file.txt', (err, result) => {
  if (err) {
    console.error(err);
    return;
  }

  console.log(result);
});
```

然而，这种方法通常会导致所谓的“回调地狱”问题，因为回调可能需要调用更多的异步 I/O，这反过来又需要更多的回调等等，最终导致混乱和难以遵循的代码。

现代版本的 JavaScript 试图通过引入 promise 来使代码更具可读性：

```
fetch("https://example.com/profile", {
      method: "POST", // or 'PUT'
})
  .then(response => response.json())
  .then(data => data['some_key'])
  .catch(error => console.error("Error:", error));
```

promises 模式的最后一步是 JavaScript 采用了由 C# 推广的 `async/await` 模式，这使得异步 I/O 最终看起来很像带有经典异常的同步代码：

```
async function fetchData() {
  try {
    const response = await fetch("my-url");
    if (!response.ok) {
      throw new Error("Network response was not OK");
    }
    return response.json()['some_property'];
  } catch (error) {
    console.error("There has been a problem with your fetch operation:", error);
  }
}
```

使用回调进行错误处理是一个需要了解的重要模式，不仅在 JavaScript 中-例如，人们在 C 中使用它已经很久了。尽管如此，它不再是非常常见的-机会是你将使用某种形式的 `async/await` 。

思考

当我们编写代码时，在调用其他函数时，函数内部会发生错误：

```
fn f() {
// Error can happen when b()
// returns an error
 a = b()
 ...
}

```

由此产生的问题是:

- 有时我们不想处理错误，只是从函数返回
- 有时候我们想减少错误
- 有时候我们希望晚点处理错误-例如，处理其他错误。相应的让正常控制流继续。

每种编程语言都找到了不同的解决方案来应对这三个挑战。

Java 是第一批通过 Exceptions 提升到更高错误管理状态的大众语言之一。 `b()` 可以在错误时抛出异常。然后调用函数什么也不能做，在这种情况下，调用函数 `f()` 返回给它的调用者，并带有异常。或者它可以稍后通过将调用包装在 `try/catch` 中来处理异常。Java 方法的缺点是在错误发生后我们不能有正常的控制流。要么我们处理，要么让它冒出来。

Java 异常机制的缺点之一是声明已检查的异常。如果我们的函数 `f()` 声明了它的异常，而函数 `b()` 抛出了不同的异常，我们需要以任何一种方式处理异常，因为它不会冒泡。

Rust 找到了一个解决方案，它有一种机制，可以自动将一个错误（ `b()` ）转换为另一个错误（ `f()` ）。这样我们就可以让错误冒出来而不处理它。Rust 使用 `?` 符号：

```
fn f() {
 // Let function f() return
 // error autoconvert and bubble up
 a = b()?
 ...
}

```

复制代码

这种方法的优点是，它使错误处理既显式又类型安全，因为编译器确保处理每个可能的结果。

一些编程语言通过在值旁边返回错误代码来处理这三个挑战。其中之一就是 Go :

```
a, err := b()

```

复制代码

现在我们可以处理错误了

```
if err != nil { .... }

```

复制代码

或者从我们的函数返回。我们可以在错误发生后有正常的程序流程-在错误情况下-除非我们想对一个操作：

```
a = a + 1

```

复制代码

如果有错误并且 `a` 为 nil，则不工作。

我们现在可以每次检查 `a` 的存在：

```
if a != nil { .... }

```

但这变得麻烦且快速不可读。

一些编程语言使用 Monads 处理错误后的控制流问题。它允许组合可能失败的函数，而不必使用 try/catch 块或嵌套的 if 语句。

```
// a is of type Result<A,E>
a = b()

```

有了 Result Monad，我就可以处理方法的错误或返回。如上所述，对于返回 Rust 有一些特殊的语法:

```
a = b()?
```

带问号，函数将在 `b()` 返回错误时返回该行，并且错误会随着自动转换而冒泡。

我们也可以在错误的情况下使用正常的控制流，但仍然使用 `a.` 魔法！

```
a = b()
c = a.map(|v| v + 1)

...
// Deal with error later

```

在错误的情况下， `c` 也将是错误，否则 `c` 将包含 `a` 的值加 1。这样，无论错误发生与否，我们都可以在错误发生后拥有相同的控制流。

这使得代码的推理更加容易。

Zig 通过用 `!` 注释类型，有一个简短的 `Result<A,E>` 概念。

```
// Returns i32
fn f() i32 {
```

```
...
}

// Returns i32 or an error
fn f() !i32 {
...
}

```

Zig 还解决了 Java 中通过流分析声明异常的繁琐问题。它会检查你的函数 `f()` 并找出它可以返回的所有错误。然后，如果您检查调用代码中的特定错误，它会确保它是详尽的。

带有 `?` 的 Rust 有一个特殊的语法来当场返回。Java 有特殊的语法 `try/catch` ，如果我们不写额外的代码，就不会当场返回并返回给函数的调用者。

问题是：我们经常做什么？返回错误或继续？我们经常使用的，应该有较少冗长的语法。对于 Rust 中的 `?` case，我们应该需要一个 `?` 来返回，还是需要一个 `?` 来不返回？

复制代码

```
a = b()?

```

`?` 可以表示“错误返回”。或者行为可以是，如果 `b()` 返回一个错误，而 `?` 阻止了这个错误，那么总是当场返回。

这取决于发生的更频繁。

Golang 可能会给予我们另一条线索。当函数返回时，它有特殊的清理语法:

复制代码

```
f := File.open("my.txt")
// Make sure we close the file
// on exiting the function
defer f.close()

a, err = b()

if err != nil {
  // f.close() is called here
  return
}

```

Java 有一些不那么优雅的东西。看起来人们认为错误应该冒出来，在这种情况下，我们需要一些简单的清理。

从我的经验来看，我也怀疑我们会希望让大多数错误通过自动转换而出现，所以 `?` 可能应该表示我们不希望函数返回，这与 Rust 正在做的相反。

Java 似乎是正确的，例外。没有语法意味着泡沫行为。它错过了自动转换和来自 Rust 的 `Exception<V,E>`，以及一个本地的，简单的 `defer`，如 Go，而不是非本地的，冗长的 Java 中 `finally`。Java 没有解释如何正确使用异常，所以每个人都以错误的方式使用异常。

那么，一个假设的语言，像这样：

复制代码

```
fn f() {
  // b() returns Result<V,E> or !V in Zig,
  // f() returns if b is an error
  // a is of type V
  a = b()

  // do not return on error but
  // a is of type Result<V,E> or !V
  a = b()!

  // compiles to a = a.map(|v| v + 1)
  a = a + 1

  // compiles to c = a.map(|v| v.c())
  // c is of type Result<C,E>
  c = a.c()
  ...
}

```

这具有更高的可读性。

当调用另一个方法时，我们应该怎么做？

复制代码

```
// Does not work if d expects
// C as a parameter type
// and not Result<C,E>
d(c)

```

有些语言有一个特殊的语言语法来处理这个问题。Haskell 有 `do`，Scala 有 `for` 但是你有特殊的代码围绕错误和特殊的上下文。这使得事情更难再读一遍，与本意相反。

所以最好抛出编译器错误。请记住，默认的方式是向上冒泡，并且 `a` 的类型是 `V`。

我们可以通过控制流分析来减轻痛苦。一些编程语言，如 TypeScript，做的是这样的事情:

复制代码

```
a = b()
a = a + 1 // A is still Result<V,E>
if a instanceof Error {
 return
}
// A is now of type V
// because we checked for an error
```

```
8 d(a)
9
```

看起来每种编程语言都有一个最佳错误处理难题。从我所看到的，没有人成功过。

参考链接：

- Musings about error handling mechanisms in programming languages
- Error handling patterns

发布于: 2023-06-07 | 阅读数: 50

错误处理 | 6 月 优质更文活动

宇宙之一粟

+ 关注

宇宙古今无有穷期，一生不过须臾，当思奋争 · 2020-05-07 加入
🏆InfoQ写作平台-签约作者 🏆 混迹于江湖，江湖却没有我的影子 热爱技术，专注于后端全栈，轻易不换岗 拒绝内卷，工作于外企开发，弹性不加班 热衷分享，执着于阅读写作，佛系不水文 同名公众号：《宇宙之一粟》

评论

快抢沙发！虚位以待

发布

· 暂无评论 ·