# 第11章 **错误处理（黑体 ，三号）**

本章要点：理解rust错误处理概念与机制，通过例子学习如何运用错误处理应对rust实践中的挑战。（黑体，小五，300字以内）

本章导图：

在前面的几章中，我们学习了 Rust 语言的基本概念和语法。通过掌握这些知识，我们可以构建出软件。然而，需要注意的是，错误是软件中不可避免的情况。因此，Rust 提供了一些错误处理的特性。在本章中，我们将深入了解 Rust 的错误处理概念，并学习如何使用错误处理机制来使程序更加健壮。

## 11.1 Rust错误处理概述（节标题宋体，四号）

**1. Rust错误处理的概念**

不同的语言对于错误(error)，异常(exception)有不同的定义。在Rust 语言中，程序遇到的非正常情况统称为错误 (error)，Rust中不使用exception这个概念。在Rust 官方文档中，将错误分为可恢复错误和不可恢复错误两大类。在有些资料中（例如 mastering rust…）,又增加了fatal error这一类错误。该3种错误如下：

**可恢复错误(Recoverable errors)**：可恢复错误(Recoverable errors)是指在用户、环境和程序交互过程中预期发生的异常情况，例如文件未找到、网络通信故障等。对于这类错误，我们通常会向用户报告问题并尝试修复异常情况。在 Rust 中，我们使用基于类型的错误处理方式来处理这类错误。

**不可恢复错误(unrecoverable errors)**: 不可恢复错误(Unrecoverable errors)指的是破坏程序 contracts 或者 invariants 的异常行为，例如下标越界(index out of bounds)或除以0的操作(divide by zero)。对于这类错误，我们需要立即停止程序以防止进一步的损害。在 Rust 中，我们使用被称为panic的机制来处理这类错误。

**致命性错误(fatal errors)**: 致命性错误(Fatal errors)是指会立即终止程序运行的异常情况，例如内存不足或堆栈溢出。Java 中定义了Error类来表示这一类错误，Java程序不应处理这类错误。同样地，在 Rust 中，程序也不会处理这类错误。

**2.多种语言的错误处理机制的对比**

错误处理机制是一种在程序中处理异常情况的方法，而两种主流的错误处理机制范式是返回代码（return codes）和异常（exceptions）。

C语言采用返回代码范式，即通过函数的返回值来表示是否出现错误。例如，C标准库的fopen函数使用返回值NULL或文件指针来表示文件是否成功打开。这种错误处理方式简单、灵活且没有额外的开销，但会造成一些问题。其一是返回值的错误检查不是强制性的，有可能被开发者疏忽，其二是错误处理代码和功能代码交织在一起，易造成逻辑混乱。

相比之下，C++、Java、Python等高级语言引入了异常处理机制，它利用栈回退(Stack Unwind)或栈回溯(Stack Backtrack)机制自动处理异常。于此同时，在语言层面引入了专门的语法例如try-catch来分离功能和错误处理的逻辑。但它仍然有不足之处，比较突出的问题是开销较大。

Rust对于可恢复的错误处理采用了 C 语言的返回代码机制，而没有选择异常处理机制。这是由于后者所带来的较大开销与Rust的零运行时成本(zero runtime costs)哲学相违背。另一方面，Rust改进了C语言的返回代码机制，使用了基于类型的返回代码机制。在Rust中，函数的返回值采用适当的错误类型，例如Option和Result，并支持用户自定义错误类型。这和函数式编程语言的处理方式类似，例如Haskell中使用Maybe和Either类型处理错误。这种设计的好处是可以在编译阶段通过类型检查确保没有错误被忽略处理。

Rust对于不可恢复的错误采用了称为 panic的机制，当做错误处理的兜底。这是一种"fail fast"机制，在这种机制下，程序会立即停止运行，以避免进一步的损害。

在接下来的小节中，将分别介绍可恢复错误和不可恢复错误的处理。

11.2 可恢复错误的处理

**12.2.1 用option和result处理可恢复错误**

如前所述，Rust对于可恢复错误采用了基于类型的返回代码机制。绝大多数的 Rust 程序使用Option和Result这两个类型来处理可恢复错误，它们充当包装器类型，将错误信息附加在返回值上。

**1.Option类型**

Option类型用来包装可能为空值的值。

在支持空值(null)即值缺失的语言中，程序员使用防御式代码来处理可能为空值的值。下列代码片段展示了在C语言中处理文件打开函数(fopen)的返回值：

#include<stdio.h>

int **main**(){

**FILE**\* fp = **fopen**("ch11\_1.txt", "r");

if (fp != **NULL**){

char buffer[10];

**fread**(buffer, 1, 5, fp);

}

**fclose**(fp);

}

代码片段ch11\_1.c

在fopen函数调用后，我们需要检查文件指针 fp 是否为空值，只有在 fp 不是空值时才能对 fp 进行操作。如果我们忘记了对文件指针的空值检查，操作空值，就可能会引起segmentation fault。

对空值的忽略或错误处理造成了大量的程序安全问题。因此，Tony Hoare 将他在程序语言中引入空值的行为称为“价值10亿美元的错误”。

Rust中没有表示空值的字面量，作为替代，Rust使用Option类型来包装可能为空值的值。

Option在标准库中定义，并包含在prelude中，因此不需要将它显式引入作用域，同时它的成员也不需要使用Option:: 前缀来使用。

Option的类型签名如下：

pub enum **Option**<**T**> {

*// 值缺失*

**None**,

*// 值不确实，关联类型为T的值*

**Some**(**T**)

}

代码片段ch11\_2.rs

Option包含None和Some(T)两个成员(variants)。 None成员表示空值，Some(T)成员表示值存在；T是泛型类型参数，表明值的类型为T。回忆一下2.2节的内容，枚举(enum)的成员(variants)可以关联数据。在Option中 Some(T)成员关联了一个T类型的值。

下面是Option的一个使用实例：

fn **main**(){

*//表示类型为int的值1*

let some\_int:Option<int> = **Some**(1);

*//表示类型char的值 a*

let \_some\_char:Option<char> = **Some**('a');

*//表示值缺失*

let \_null\_char:**Option**<**char**> = **None**;

*//提取some\_int中的值 并 +1*

let \_some\_int\_plus\_one = some\_int.**unwrap**() + 1;

*//编译错误:cannot add `{integer}` to `Option<{integer}>`*

let \_some\_int\_plus\_one\_err = some\_int + 1;

}

代码片段ch11\_3.rs

注意在12行，some\_int+1这个表达式会产生编译错误。some\_int的类型是Option<int>，1的类型是int，Rust不知道如何将Option<int>和int相加，因为他们类型不同。

这说明在Rust中，如果要使用Option<T>的值，必须显式地将值提取出来，如片段第9行所示。这个显式转换要求，可以提示程序员对可能的空值进行处理。反之，如果一个值不是Option<T>类型，那么程序员可以安全地认定这个值永不为空。Rust通过这个机制来避免空值造成的程序安全问题。

**2.Result类型**

在上一小节中，我们介绍了使用Option来处理出现空值的情况。在本节，我们将使用Result来处理任意错误的情况，而不仅仅是出现空值。

Result类型签名如下所示：

enum **Result**<**T**, **E**> {

*//操作成功，关联类型为T的结果*

**Ok**(**T**),

*//操作失败，关联类型为E的错误*

**Err**(**E**),

}

代码片段ch11\_4.rs

Result 包含Ok(T) 和 Err(E) 两个成员，T和E为泛型类型参数，表明Ok(T)和Err(E)所关联值的类型。Ok(T)成员表示操作成功，没有错误，关联类型为T的返回值，Err(E)成员表示操作失败，关联类型为E的错误信息。

值得注意的是，和Option一样，Result及其成员也被导入到了prelude中，可以在程序中直接使用。

下面是Result的一个使用实例：

use **std**::**fs**::**File**;

fn **main**() {

let file\_result = **File**::**open**("ch11\_5.txt");

let \_file = match file\_result{

**Ok**(file) => file,

**Err**(error) => **panic!**("{}", error),

};

}

代码片段ch11\_5.rs

在上面的实例中, 我们在调用File::open函数试图打开文件，该函数的返回值类型是Result<File, Error>。若文件打开成功，则Result成员为Ok(File)，关联该类型为File的该文件句柄。若打开失败，则Result的成员为Err(Error)，关联类型为Error的错误信息。

与Ch11\_1在C语言中打开文件不同，我们不需要使用防御式代码来处理文件打开错误情况，而是使用模式匹配等方法直接对返回值进行操作。在上面的实例中，我们使用match（模式匹配）的方法对返回值进行操作。如果匹配为Ok(File)，那么文件打开成功，提取出file这个文件句柄给下文进行操作，如第6行所示。反之，文件打开失败，此时错误信息记录在error中，需要进行错误处理。如第7行所示，该片段中错误处理方式为使用Panic！宏输出错误信息。

下面总结一下Option和Result。他们都是枚举类型，通过对返回值的包装来附加错误信息。在处理这两个类型时，我们首先需要对模式进行匹配，然后进行相应的操作，例如对于Option::Some(T)提取关联的值，对Result::Err(Error)进行错误处理。

那么如何模式匹配并进行处理呢，我们将在下面几节中进行详细介绍。