# 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**

在编程中，错误和异常是常见的问题。不同的编程语言对于错误和异常的定义也有所不同。在Java中，异常是指程序运行时出现的非预期错误，而在Python中，异常则可以是语法错误、运行时错误、逻辑错误等。在Rust中，我们使用错误统称程序中遇到的非正常情况，而不使用异常(Exception)这个概念。

**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。这些错误通常是可以通过一定的手段进行修复的，在程序中处理这些错误可以让程序更加健壮。在Rust中，我们基于类型的返回值来表示函数调用的成功或失败，并提供了模式匹配语法来对可能出现的错误进行处理。这种方式能够让程序员更加明确地了解哪些函数可能会出错，以及如何处理这些错误。通过对可恢复错误的处理，我们可以提高程序的可靠性和稳定性，同时也能够提高用户的体验。

**不可恢复错误(unrecoverable errors)**: 不可恢复错误(Unrecoverable errors)指的是那些严重的运行时错误，例如下标越界(index out of bounds)或除以0的操作(divide by zero)，这些错误会破坏程序的contracts或者invariants，导致程序无法继续执行。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。当程序panic时，会打印出错误信息并停止运行。与其他语言不同的是，Rust中的panic并不会导致内存泄漏等安全问题，因为Rust的所有权模型能够保证在panic时自动释放所有已分配的资源。

**致命性错误(fatal errors)**: 致命性错误(fatal errors)是超出程序处理能力的错误，它们通常是由于系统故障、资源不足、硬件损坏等原因引起的。这些错误会导致程序无法继续运行，因此需要立即终止程序。当出现致命性错误时，与不可恢复错误不同，程序没有办法主动终止自身的运行，因为这些错误是由程序运行的外部环境造成的。Java使用Error类来标识致命性错误，以表示程序不需要处理这类错误。同样地，Rust也没有提供机制处理致命性错误。

**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，它可以让程序更加稳定和可靠。在现代编程语言中，主流的错误处理机制范式有两种：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。这种方式能够让程序员更加明确地处理可能出现的异常情况，使代码更加清晰、易读和易维护。但它仍然有不足之处，比较突出的问题是开销较大，尤其是在异常被抛出时需要进行栈回退，这可能会内存开销昂贵的场合，例如嵌入式程序的性能产生很大影响。

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。

基于类型的返回代码机制是指函数的返回值是一个可以表示错误的类型，例如Rust提供了Result这个枚举类型，返回值是Ok枚举成员表示函数成功运行，而若是Err枚举成员则表示函数出现错误。Rust不仅提供了例如Option和Result的枚举类型来表示可能的错误，还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，例如在Haskell中使用Maybe和Either类型处理错误。使用基于类型的返回代码范式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，Rust会自动释放已分配的资源，然后终止程序或线程。当然，对不可恢复错误我们也可以显式地指定使用abort处理方式，在这种处理方式下，Rust将直接终止程序，避免释放资源过程的开销。

在接下来的小节中，我们将详细地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

**11.2.1 用option和result处理可恢复错误**

正如前面提到的，Rust采用了基于类型的返回代码机制来处理可恢复错误。这种机制可以帮助程序员更加明确地了解哪些函数可能会出现错误并如何处理这些错误。在Rust中，我们通常使用Option和Result这两个类型来处理可恢复错误。这两个类型是Rust中的枚举类型，它们能够将函数是否出现错误的信息附加在返回值上，相比起直接使用返回值表示错误，这种方式隔离了返回值和错误信息，使得错误变得明晰且方便处理。

**1.Option类型**

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

这段代码用来打开文件名是 ch11\_1.txt的文件，读取其中的前5个字符，然后关闭文件

程序第4行，代码调用fopen打开文件，如果打开成功，返回文件指针，否则，返回空值。因此在程序第5行，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的问题被称为空指针异常。空指针异常在编程中非常普遍，造成了大量的程序安全问题。空值的发明者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”。

很多**程序设计语言**尝试从语言设计层面解决空值问题，例如**Kotlin**程序设计语言通过引入可空类型来解决空指针异常问题。在Kotlin中，使用"?"来表示一个变量可以为空，使用"!!"来表示一个变量不可能为空。

与Kotlin相比，Rust解决空值的方案更加简洁和精巧。Rust去除了表示空值的字面量，作为替代，Rust使用Option类型来表示可能为空的值。Option类型的None枚举成员表示值为空，Some枚举成员表示值不为空，因此Rust可以在编译阶段通过类型检查的方式避免对空值的错误访问。与Kotlin显式标记可空相比，Rust的处理方式使语言的类型系统更加和谐统一。

值得注意的是，Option类型及其枚举成员已经在标准库中定义，并包含在prelude中，因此使用它之前不需要使用use将它显式地引入作用域，同时枚举成员None和Some也不需要添加Option路径限定符 :: 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类型是一个枚举类型，包含None和Some(T)两个枚举成员(variants)。 None成员表示空值，Some(T)成员表示值存在，类型为T。枚举类型的成员可以关联数据，Option中的Some(T)成员关联了一个T类型的值，因此我们可以使用用于枚举类型的模式匹配语法将Some关联的值提取出来。

值得注意的是，T是一个泛型参数。泛型是对具体类型或属性的抽象替代，它可以将用于使用数据的类型参数化，从而在定义时不需要指定具体的数据类型，在使用时再指定。

当我们要用Option表示一个可能为空的值的时，我们只要将T替换为这个值的类型，便得到了的用于该值的Option类型。例如，我们使用Option<i32>来表示一个可能为空的i32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<i32> = **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

在这个例子中，some\_int 变量被绑定为Some(1)，它的类型为Option<i32>,表示类型为i32的非空值1。\_some\_char表示类型为char的非空值a，\_null\_char 表示一个空值。

注意在12行，some\_int+1这个表达式会产生编译错误。这是因为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，我们将学习使用Result来处理任意错误。

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, 我们调用File::open函数打开文件，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)，关联类型为File的文件句柄。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。

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。

**11.2.2 用match匹配不同的错误**

在上一小节中，我们了解到需要对Option和Result进行模式匹配来处理错误。在前面章节中，我们介绍了Rust使用match语法来进行模式匹配。

match语法是一个灵活且强大的模式匹配工具，用于匹配一个值并根据其匹配到的模式执行相应的代码分支。match语句由一系列的分支组成，每个分支包含一个模式和对应的代码块。当match语句被执行时，Rust会从上到下依次尝试匹配每个分支，直到模式匹配成功，并执行对应的代码块。在通过match语法处理Result和Option时，模式是枚举成员，对应的代码块是错误处理逻辑。

下面我们结合一个文件操作中错误处理的例子来学习如何使用match匹配不同的错误。

use **std**::**fs**::**OpenOptions**;

use **std**::**fs**::**OpenOptions**;

use **std**::**io**::**ErrorKind**;

fn **main**(){

**OpenOptions**::new

let \_file = match **OpenOptions**::**new**().**write**(true).**create\_new**(true).**open**("ch11\_6.txt"){

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

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

};

let \_file\_dup = match **OpenOptions**::**new**().**write**(true).**create\_new**(true).**open**("ch11\_6.txt"){

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

*//match this*

**Err**(err) => match err.**kind**(){

**ErrorKind**::**AlreadyExists** => **panic!**("File exists!"),

\_ => **panic!**("{}", err)

}

};

}

代码片段ch11\_6.rs

这个例子我们将尝试重复创建两次ch11\_6.txt这个文件。假设在程序执行之前，所在

我们使用OpenOptions来设置文件的打开方式，在程序第6行我们设置了write为true, create\_new 为 true，因此open函数将会使用新建ch11\_6.txt，并以写入模式打开该文件。

open函数可能会返回多种错误；如果程序没有权限创建文件，则错误信息为**ErrorKind::PermissionDenied，**如果ch11\_6.txt已经存在，则新建文件失败，错误信息为ErrorKind::AlreadyExists。

在程序第5行，我们第一次新建并打开该文件，open函数将返回Result类型的结果，我们使用match进行处理。如果匹配的是Ok枚举成员，说明文件被成功创建，我们将提取出文件句柄，并绑定到\_file变量上。如果匹配的是Err枚举成员，说明文件创建失败，这时我们使用panic！宏对错误信息即err中的信息进行处理。panic！宏会输出该错误信息并停止程序。

在程序第9行，我们使用open函数尝试第二次新建文件。如果程序能执行到这个程序点，说明我们已经成功新建了ch11\_6.txt, 此时open函数再次新建将出现错误并返回ErrorKind::AlreadyExists这个错误信息。在12行，我们使用嵌套匹配对错误信息进行处理。外层的match将匹配Err枚举成员，并提取出错误信息绑定到err**变量**上，内层的match将匹配错误信息的类型，并使用panic！宏输出错误信息。如果错误信息的类型是AlreadyExists，那么我们输出的错误信息是File exists!，其他类型的错误信息我们将原样输出。

**11.2.3 用unwarp，expect简化Result和Option的处理**

在前面的章节中，我们已经了解了如何使用match语句来处理Rust中的Result和Option类型，这是一种非常灵活的处理方式，因为我们可以通过match匹配对应的代码段实现丰富的错误处理逻辑。

虽然这种方法非常灵活，但并不是很方便，对于每一个Result和Option类型，我们至少需要使用三行代码进行处理。同时实际开发时我们为了提高开发速度而采用"fail fast"的错误处理模式。这是一种非常简便的处理方式，错误处理方式是固定的，即对于成功的操作提取返回值，对于失败的操作则立即使用panic！打印错误信息并停止程序。

考虑到上述两个因素，Rust在Option和Result中提供了unwrap和expect方法，调用这两个方法就可以使用 ”fail fast” 错误处理范式。

unwarp方法在处理成功时会返回Ok枚举成员中的值，在失败时会自动调用panic!并输出错误信息。

expect与unwarp方法类似，但在失败时使用的错误信息是我们传递给expect的参数。

下面是使用unwarp和expect简化代码片段ch11\_6.rs的例子。

use **std**::**fs**::**OpenOptions**;

fn **main**(){

let \_file= **OpenOptions**::**new**()

.**write**(true).**create\_new**(true)

.**open**("ch11\_6.txt")

.**unwrap**();

let \_file\_dup = **OpenOptions**::**new**()

.**write**(true).**create\_new**(true)

.**open**("ch11\_6.txt")

.**expect**("File Exists!");

}

代码片段ch11\_7.rs

部分运行结果如下图所示

*thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Os { code: 17, kind: AlreadyExists, message: "File exists" }', ch11\_6.rs:3:80*

这个例子与上个例子功能逻辑与错误处理逻辑基本一致。但我们使用了unwrap和expect方法来简化代码。在第6行，我们使用unwarp方法替代ch11\_6.rs中**第6-10行**用match处理错误的代码，在第10行使用expect方法替代ch11\_6.rs中**第11-16行**使用嵌套match处理错误的代码。与ch11\_6.rs相比，这段代码显然更加短小、清晰。

**11.2.4 错误传播**

在编写可能发生错误操作的函数时，除了在函数内部处理相关错误，还可以选择通过传播错误将错误处理交由函数的调用者处理，这种方式被称为错误传播。错误传播能够更好地进行错误处理，因为调用者可能拥有更多的信息或逻辑去决定如何处理该错误，尤其是当函数被用于不同的上下文时。此外，通过错误传播，可以将错误处理的责任从被调用者转移到调用者，这有助于提高被调用函数的通用性并减少代码的耦合性。但是，在使用错误传播时，要清晰并全面的设计错误返回的信息，使用准确的错误类型结合程序API文档等多种手段，确保调用者正确理解错误信息并进行错误处理。

下面我们通过一个例子来介绍如何使用错误传播。

use **std**::**io**::{**Error**, **Write**};

use **std**::**fs**::{**OpenOptions**};

fn **create\_file\_write\_string**(st:&**str**)->**Result**<**usize**, **Error**>{

let file\_result = **OpenOptions**::**new**()

.**write**(true).**create**(true)

.**open**("ch11\_8.txt");

let mut file = match file\_result{

**Ok**(file)=>{

file

},

**Err**(open\_err) => return **Err**(open\_err)

};

match file.**write\_fmt**(**format\_args!**("{}", st)){

**Ok**(\_) => **Ok**(st.**len**()),

**Err**(write\_err) => **Err**(write\_err)

}

}

代码片段ch11\_8.rs

在上述例子中，我们定义了一个名称为 create\_file\_write\_string的函数。这个函数会创建ch11\_8.txt这个文件，并将传入的字符串写入该文件。函数如果写入成功会返回写入字符串的长度，失败则会返回Error类型的错误信息。

在程序第**4**行，我们通过OpenOptions设置了新建文件且以写入模式打开文件的标志，接着使用open函数创建文件并打开该文件。。如果打开文件成功，则将该文件对象存储在file变量中，否则直接返回打开文件失败的错误信息。

接着，file使用write\_fmt方法，将传入的字符串写入文件。如果写入操作成功，则将返回写入字符串的长度，用Ok枚举成员包装，否则将返回错误的信息，用Err枚举成员包装。

调用create\_file\_write\_string的函数可以根据所在的上下文及其他信息选择如何处理create\_file\_write\_string函数返回的错误信息。

上述例子使用了处另外一种错误处理的范式，即及早返回范式。与上一节所介绍的fail fast范式不同，该模式在出现错误时会立即返回错误值，以将错误传播给调用方。

**11.2.5 用 ? 操作符简化传播错误**

在上一节中，我们学习了错误传播和及早返回模式。然而，Rust语言提供了更为简洁的语法，即"?"运算符，使得我们可以更加轻松地使用及早返回模式。当在Result或Option类型后使用"?"运算符时，如果其枚举成员为Ok或Some，则该表达式的值将是Ok或Some中的值；否则，函数将会立即返回一个Err。

使用 “?” 运算符的好处是可以使代码更加简洁，加快开发速度，同时也避免了冗余的错误处理代码破坏了功能逻辑代码的整体性。下面一个例子展示了如何在Rust中使用”?” 运算符。

use **std**::**io**::{**Error**, **Write**};

use **std**::**fs**::{**OpenOptions**};

fn **create\_file\_write\_string**(st:&**str**)->**Result**<**usize**, **Error**>{

match **OpenOptions**::**new**()

.write(true).create(true)

.**open**("ch11\_8.txt")?

.write\_fmt(**format\_args!**("{}", st)) match {

**Ok**(\_) => **Ok**(st.**len**()),

**Err**(write\_err) => **Err**(write\_err)

}

}

代码片段ch11\_9.rs

这个函数与我们在上一节中实现的函数的功能完全一致，只不过我们使用？运算符代替match组合return的编码方式。

在代码段第8行，我们对open函数的返回结果使用了 ？运算符。从前面的介绍我们知道，open函数会返回一个Result类型的值。如果文件创建并打开成功，返回结果将是枚举成员Ok,？运算符将会提取从中提取file，并将file作为表达式参与接下来的计算。在程序第**7**行，我们调用write\_fmt将字符串写入文件。而如果文件创建或打开失败则将返回错误信息。

让我们总结一下Rust语言的可恢复错误处理。

可恢复错误是指在用户、环境和程序交互过程中预期发生的异常情况。一般而言，在代码实现阶段程序员应尽可能将出现的错误视为可恢复错误进行处理，这一方面提高了程序的健壮性，另一方面也提高了程序的使用体验，因为相比于不可恢复错误直接终止线程或程序，可恢复错误可以在处理完错误后回到程序正常的执行流，使用户不需要重启程序就可以接着使用。

Rust使用基于类型的返回代码范式，常见的类型有Option与Result，这是两个枚举类型。通过对枚举成员类型的匹配就可以区分是函数正常的返回值还是错误信息。编译器可以通过类型检查确保程序员没有遗漏对错误信息的处理。

可恢复错误的处理有多种处理范式，我们可以使用match语法灵活的对错误进行处理。常用的处理范式有fail fast范式和及早返回范式，fail fast范式对错误使用panic机制进行处理，而及早返回范式对错误进行错误传播。Rust分别提供了unwarp,expect以及？运算符帮助程序员快速实现这两种范式。

11.3 不可恢复错误的处理

**11.3.1 使用panic!处理不可恢复错误**

当处于执行阶段的代码遇到某个错误时，如果没有有效的办法能够处理这个错误以恢复代码的执行；而忽略它们则有可能破坏程序的状态。那么这样的错误就是不可恢复错误。

对不可恢复的错误的处理我们采用fail-stop的处理方式，即立即停止出现该错误对象的执行，并回收与其相关资源。采用这种方式是因为这样能将错误的影响控制在最小范围，不影响其他对象的执行。

Rust为我们提供了Panic机制来处理不可恢复错误。Panic机制会停止发生错误的对象的执行，回收与对象相关的资源，并使用Backtrace记录错误发生的过程。作为一门系统编程语言，Panic机制相对于其他高级语言的异常处理机制而言是轻量级的，同时Panic机制提供了只终止对象执行而不回收资源这一选项，这使得Rust适用于不同的平台和应用场景。

触发Panic机制有两种方式，第一种是一些错误的对象操作会触发panic，例如访问超过数组末尾的内容；另一种可以通过Rust提供的panic!宏显式地触发panic。

接下来两个例子分别展示了两种触发方式：

fn **main**(){

let s = **vec!**[1, 2, 3];

//index out of bounds

**println!**("{}", s[3]);

}

代码片段ch11\_10.rs

在这个例子中，我们新建了一个具有3个元素的变长数组(Vec)。在程序第4行，我们试图访问第4个元素，这造成了访问越界。

在C语言中，尝试读取越界的值是未定义行为，我们可能会读取到其他变量在这个内存位置的值。这种情况被称为缓冲区溢出。缓冲区溢出可能会造成安全漏洞，例如攻击者可以利用溢出获取内存中被保护的数据。

为了避免可能的安全漏洞，Rust标准库在实现对可索引对象进行访问时会首先检查索引是否越界，并将越界行为当作不可恢复错误进行处理。

因此第4行的操作将会触发不可恢复错误，我们可以在控制台获得如下输出：

thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 3', ch11\_10.rs:4:20

fn **main**() {

let s = **vec!**[1, 2, 3];

let index = 3;

if (index >= s.**len**()){

**panic!**("index{} out of bounds!", index);

}else{

**println!**("{}", s[index]);

}

}

代码片段ch11\_11.rs

在上面这段代码中，我们模拟了上一个例子中触发index out of bounds的逻辑。在程序第5行，我们使用panic!宏手动触发了index out of bounds。

Panic!与Print!宏类似，我们也可以使用格式字符串(format string) 来自定义Panic传递的消息。在程序第5行，我们使用格式化字符串输出了越界的下标值。 程序的输出如下所示：

thread 'main' panicked at 'index 3 out of bounds!', src/main.rs:5:9

**11.3.2 并发编程中的不可恢复错误处理**

上一小节中，我们介绍了使用Panic机制处理不可恢复错误。Panic机制会停止当前产生错误对象的执行，将错误造成的影响最小化。

作为高性能、高可靠性的系统编程语言，Rust提供了内存安全的并发编程支持。Rust中的并发编程主要通过两种方式实现：线程和异步编程。这里我们简要介绍一下使用线程进行并发编程。

在Rust中，线程是一种轻量级的执行单元，允许程序在多个并发的执行上下文中运行。Rust为了保证内存安全的并发编程设计了很多机制，其中较重要的是使用线程安全类型、所有权机制和线程本地存储机制。线程安全类型可以实现线程之间的互斥访问和共享访问，从而保证数据的线程安全性。所有权机制要求每个值只能有一个唯一的所有者，该所有者在唯一的一个线程中。线程本地存储机制，可以创建线程本地变量，避免不同线程之间共享同一变量的竞争问题。

Rust提供了标准库中的std::thread模块，可以方便地创建、管理线程，并实现线程之间的同步和通信。另外，Rust还提供了std::sync模块，包含了多种同步原语，如互斥锁、条件变量、原子类型等，使得线程之间的操作可以进行同步和协调。

Panic机制可以用于Rust并发编程中，使用Panic可以中止发生不可恢复错误的线程的执行，而不会影响其他线程。具体来说，如果主线程出现了不可恢复错误并调用了Panic，整个程序将立即终止，并返回非零退出码101；如果子线程出现了不可恢复错误，那么该错误会通过panic机制在子线程内部进行处理，即停止该子线程并回收相关资源，而不会将错误传播给父线程，也不会对其他线程造成影响。

下面的例子构造了主线程和子线程分别Panic的情况。

use **std**::**thread**;

fn **trigger\_error**(thread\_name:&**str**){

**panic!**("panic in thread {}", thread\_name);

}

fn **spawn\_child**() -> **thread**::**JoinHandle**<()>{

**thread**::**spawn**(move ||{

**println!**("child thread does something");

**trigger\_error**("child thread");

**println!**("this code should not be executed!")

})

}

fn **main**(){

**println!**("spawn child thread");

let child = **spawn\_child**();

let \_ = child.**join**();

**println!**("main thread does something");

**trigger\_error**("main");

}

代码片段ch11\_12.rs

程序运行的结果如下所示

spawn child thread

child thread does something

thread '<unnamed>' panicked at 'panic in thread child thread', ch11\_12.rs:4:5

main thread does something

thread 'main' panicked at 'panic in thread main', ch11\_12.rs:4:5

exit code:101

在该例子中，主线程(main函数)会生成一个子线程 (程序输出spawn child thread)，并等待子线程执行完毕 (代码第18行 child.join)。子线程会模拟执行一些操作 (程序输出 child thread does something), 并触发panic。此时Rust会停止并销毁该线程，因此程序第11行不会被执行。而主线程并不会退出，主线程继续执行第19行代码(程序输出 main thread does something)，之后主进程触发panic，整个程序退出并返回错误码101。

值得一提的是，虽然Rust精心设计了多种机制来提高并发安全性，但并发编程中的错误处理好比Rust并发编程的阿喀琉斯之踵，稍有不慎便会造成安全问题。对此有兴趣的读者可以查阅官方文档及相关论文获得更多并发编程中错误处理的准则和实践建议。

**11.3.2 panic机制中的栈展开与中断(abort)**

Rust使用panic机制处理不可恢复错误，将立即停止出错线程的运行。然而，在终止线程运行的同时，还需要对线程在堆栈和堆上申请的资源进行回收，这一回收过程被称为栈展开。

Panic机制将栈展开作为默认选项，而我们可以通过修改Cargo.toml文件中增加属性（如下图所示），将立即停止(abort)作为panic的处理方式，即只终止该线程的运行，而将资源回收交给其他对象例如操作系统去处理。

[profile.release]

panic = "abort"

Rust将栈展开作为默认选项，是为了防止出现内存泄漏等安全问题，而可以选择中断(abort)，则为应对栈展开导致内存开销过于昂贵的极端情况，例如使用Rust编写嵌入式系统。

与C++等高级语言的异常处理不同，Rust的栈展开机制不需要查找并调用任何异常处理函数，而使用轻量的方法对资源进行回收。栈展开沿着函数的调用链释放函数的栈帧，回收堆栈上的资源。对于在堆上分配资源的变量，Rust会在释放该变量所在函数栈帧之前先对它执行drop方法，以释放堆上申请的空间。

**11.3.3 使用backtrace功能回溯不可恢复错误**

为了找到不可恢复错误发生的位置及原因，Rust提供了backtrace功能。Backtrace可以在栈展开之前回溯函数的调用链，记录函数的调用过程。

我们可以在Shell中设置环境变量RUST\_BACKTRACE=1或RUST\_BACKTRACE=all 来在Shell中查看backtrace的结果。

Ch11\_13是一个函数调用过程中发生不可恢复错误的例子，Rust默认使用栈展开作为Panic的处理方式。

我们将借助backtrace定位发生panic的位置，并结合前一小节的知识，简要分析栈展开的过程。

fn **b\_called\_by\_a**(){

**panic!**("panic at fn b");

}

fn **a\_called\_by\_main**(){

let \_a\_obejct\_in\_heap = **String**::**from**("This str is in heap");

let \_a\_object\_in\_stack1 = 0;

let \_a\_object\_in\_stack2 = [1, 2, 3];

**b\_called\_by\_a**();

}

fn **main**(){

**a\_called\_by\_main**();

}

在这个例子中，main函数调用了a\_called\_by\_main函数，a\_called\_by\_main函数在堆栈上和堆上申请了一些资源。a\_called\_by\_main函数接着调用了b\_called\_by\_a函数，并在该函数中触发了panic。

我们首先来看一下backtrace的结果。

thread 'main' panicked at 'panic at fn b', src/ch11\_13.rs:2:5

stack backtrace:

0: \_rust\_begin\_unwind

1: core::panicking::panic\_fmt

2: a::b\_called\_by\_a

at ./src/main.rs:2:5

3: a::a\_called\_by\_main

at ./src/main.rs:9:5

4: a::main

at ./src/main.rs:12:5

5: core::ops::function::FnOnce::call\_once

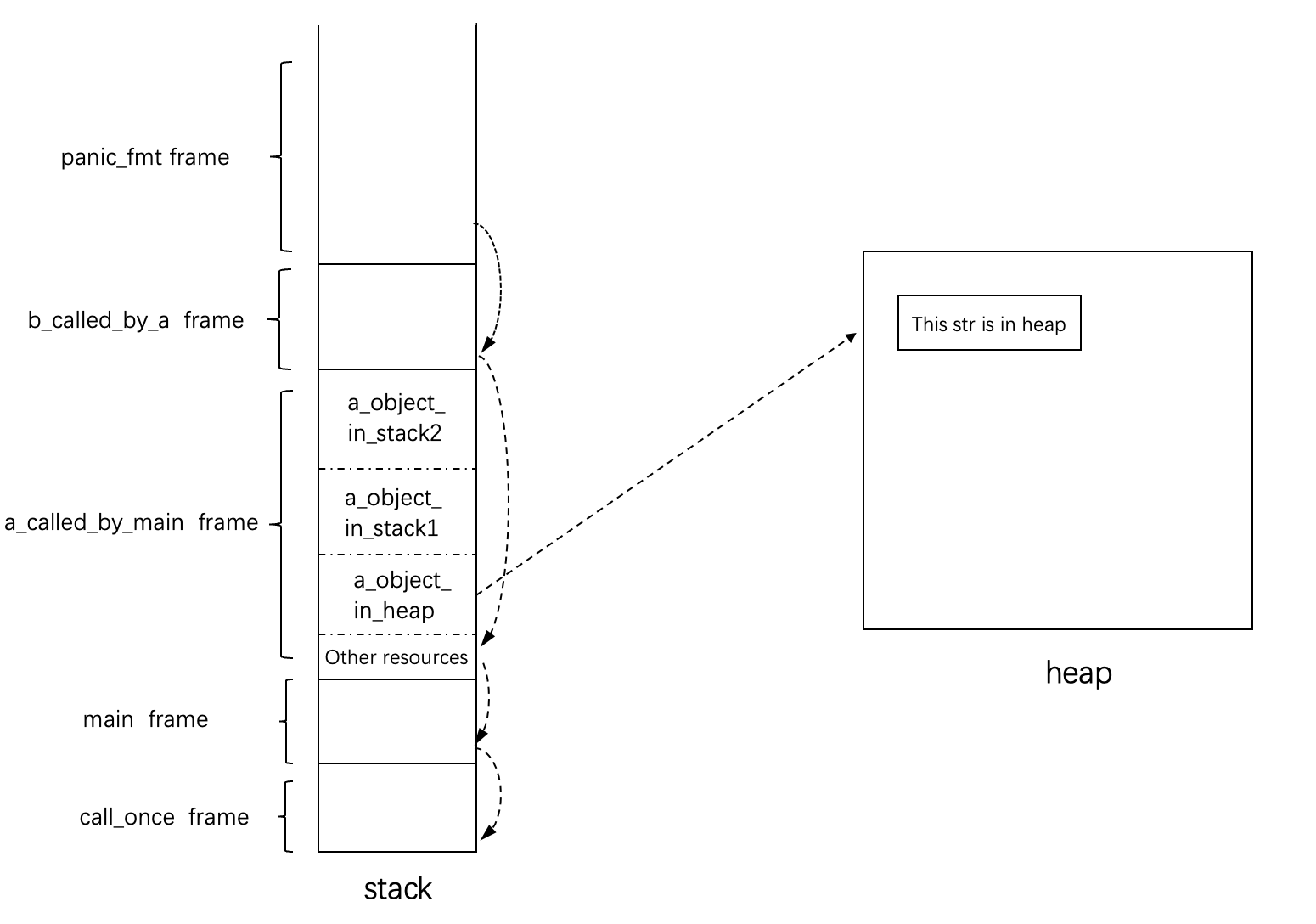
at /private/tmp/rust-20230210-6343-w92jca/rustc-1.67.1-src/library/core/src/ops/function.rs:507:5

从backtrace输出的第一行可以发现，该程序是在程序第2行，即b\_called\_by\_a函数的第一个语句触发panic的，这和我们之前的分析相一致。

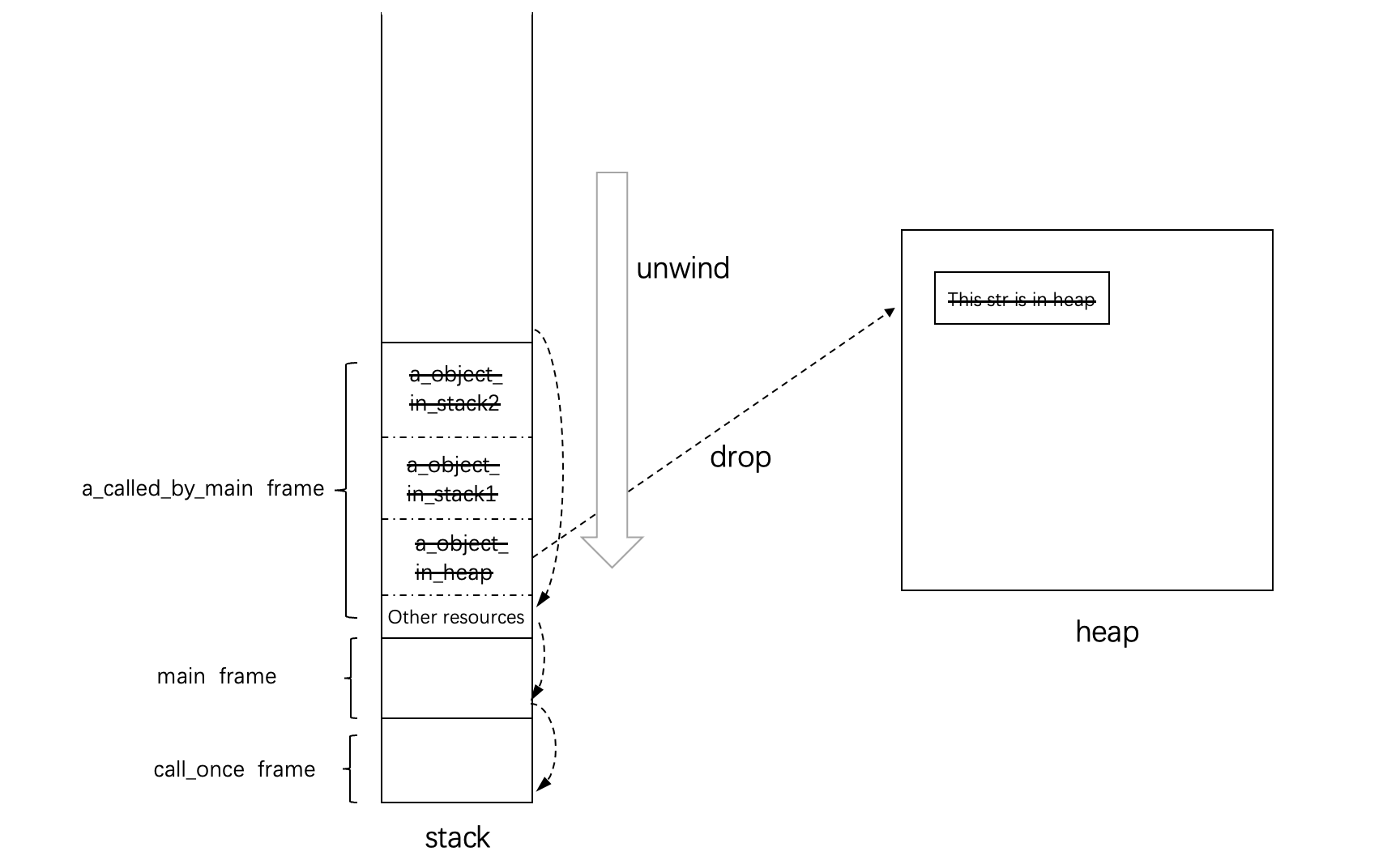
从backtrace的输出第二行开始，是发生panic时的程序栈帧情况，主要内容是函数的调用链。每个被调用的函数有一个编号，编号为 i 的函数被编号为i+1的函数调用。例如在上述例子中，编号为3的函数a\_called\_by main 被编号为 4 的main函数调用。

通过观察函数调用链，我们可以知道，程序第2行的panic！宏展开后调用了core::panicking::panic\_fmt这个函数，从而进入panic机制。而该函数又调用了\_rust\_begin\_unwind，进行栈展开。（事实上\_rust\_begin\_unwind只是一个Rust内部用于链接符号，代表开始栈展开，实际的栈展开功能由与平台相关的**动态链接库**提供）。backtrace记录了开始栈展开时刻的栈帧情况。

通过函数调用链，结合源代码，我们可以画出发生panic时的程序堆栈和堆的示意图，如下图所示。



如上图所示，图的左边为程序栈展开时刻堆栈的情况，右侧为堆的情况。在a\_called\_by\_main函数的栈帧中，我们新建了两个栈上对象 a\_object\_in\_stack1, a\_object\_in\_stack2；一个堆上字符串对象，这个对象由栈上胖指针a\_object\_in\_heap指向。



在栈展开过程中，如图所示，函数栈帧从栈顶向栈底展开，堆上对象通过drop释放。

11.4 错误恢复的实践建议

在前面章节中，我们可以使用Option, Result 进行可恢复处理，panic机制进行不可恢复处理。在实际错误处理中是使用可恢复错误处理还是不可恢复错误处理呢？本节将提供一些实践方面的建议。

让我们首先探讨错误恢复在不同阶段的代码开发中的实践原则。一般来说，代码开发可以分为四个阶段：代码示例、代码原型、代码实现和代码测试。代码示例是用来展示项目中某些概念的一些代码，例如API的实现逻辑。代码原型是一个小型软件系统，满足基本需求，主要用于确认和验证。代码实现是代码开发的主体部分，需要实现一个完整的、满足需求的软件系统，并满足健壮性等要求。代码测试则是通过测试保证所写的代码满足软件需求及健壮性等要求。

在代码示例、代码原型和代码测试阶段，我们建议优先使用panic方法，即不可恢复错误处理。原因如下：

在代码示例阶段，需要突出代码的功能性，而可恢复错误处理需要编写健壮的、可能较为复杂的错误处理代码，这可能会使代码的功能性描述不够清晰。

在代码原型阶段，为了快速构建原型，我们可能会暂时忽略错误处理逻辑。使用panic方法中的unwarp和expect可以确保程序在遇到错误时的安全性。同时，这两个函数也是清晰的标记，提示我们可以在代码实现阶段替换panic方法并实现具体的可恢复错误处理逻辑。

在代码测试阶段，使用panic可以在出现错误时快速结束测试并捕捉错误发生的位置。

而在代码实现阶段，则优先使用Result进行可恢复错误处理。这是因为一旦panic，就没有恢复的可能。为了程序的健壮性，开发者应尽可能细致的考虑错误情况并解决可以恢复的错误；即使被调用者在当前无法对该错误进行处理，它也可以通过错误传播的方式将错误处理交于调用者使用。

下面，我们将从程序开发者的角度来探讨错误处理的原则。

在程序开发中，我们通常将开发对象分为库对象和应用对象。库对象是被调用方，提供库函数给调用方应用对象使用。

对于库对象开发者而言，为了提高库的通用性和灵活性，需要尽量将错误处理的权限交给应用对象开发者。因此，使用Result进行错误传播是首选方法。但是，当出现破坏程序状态的错误时，应该尽量使用panic机制来终止程序或线程的运行，而不是将该不可恢复错误交给应用对象开发者处理。这是因为这种情况通常是由应用对象开发者错误使用库对象造成的，使用panic可以提醒应用对象开发者他们的代码存在缺陷需要修复。

而对于应用对象开发者而言，他们可以通过库函数的返回类型和API文档来确定如何处理库函数传播的错误。与代码实现类似，应优先使用Result处理可恢复错误。

11.5 小结

本章介绍了 Rust 中的错误类型，包括可恢复错误和不可恢复错误。我们学习了如何使用 Option 和 Result 来处理可恢复错误，以及使用 panic 机制来处理不可恢复错误。此外，我们还学习了如何传播错误和选择合适的错误处理方式，并且在学习过程中深刻领会到 Rust 对于错误处理的科学、严谨和高效设计。

通过本章的学习，我们可以更好地利用 Rust 的错误处理机制，编写出更加健壮和可靠的代码。希望读者能够充分掌握这些实践经验，并将其应用于实际开发中。