# 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，我们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

**11.2.1 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。

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。

**11.2.2 用match匹配不同的错误**

在上一小节中，我们提到，在对Option和Result进行处理事要先进行模式匹配。在前面章节[]中，我们介绍了Rust 提供了match关键字来提供模式匹配。我们通过match关键字匹配错误并提取出相应的错误信息进行操作，下面是一个实际的例子。(ch11\_6.rs)

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

在上述例子中，我们使用open函数两次创建ch11\_6.txt这个文件。open的返回类型为Result<File, Error>。由于使用create\_new函数设置了create\_new这个flag，如果需要创建的文件存在，则将返回ErrorKind::AlreadyExists这个错误。ErrorKind是一个枚举类型，AlreadyExists是中间的一个枚举成员，表示文件已经存在这个错误。

在程序第5行，第9行，我们分别使用match 处理 open函数的返回值。在第7行，如果返回值为Ok枚举成员，说明文件被成功创建，我们将它提取到file变量中，并绑定到\_file变量上。如果返回值为Err枚举成员，我们将直接panic并打印错误。在第9行，成员运行到这里时由于ch11\_6.txt已经被创建，open函数将返回ErrorKind::AlreadyExists这个错误。在12行，我们再次使用match对提取出的err类型进行匹配，进行相应处理。

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

在上一节中，我们使用match对Result和Option进行处理，但它可能有点冗长。同时对于操作结果，我们往往使用fail fast模式，即，对于操作成功的返回值，从Ok或Some枚举成员中提取出值；对于操作失败的返回值，提取错误信息并panic。Rust提供了unwarp和expect两个方法来进行上述的常规处理，帮助我们简化处理流程。

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。

Ch11\_6.rs的例子可以用unwarp和expect进行如下简化

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*

**11.2.4 传播错误**

当编写可能出现失败操作的函数时候，除了在这个函数内部处理相关错误，还可以选择返回该错误，并将该错误处理交由函数的调用者处理，这被称为传播错误。传播错误能更好地进行错误处理，因为相比被调用者，调用者可能拥有更多的信息或逻辑去决定该如何处理该错误。

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，create\_file\_write\_string会接受一个字符串，并把结果写入ch11\_8.txt。如果写入成功会返回写入字符串的长度，失败则会返回相关错误。

上述例子使用了处理返回结果的另一个模式，及早返回模式。与上一节fail fast模式不同，在出现错误时我们提前返回错误值，将错误传播给调用方。

**11.2.5 用 ? 操作符简化传播错误**

上一节我们介绍了传播错误以及及早返回模式，事实上Rust提供了 ? 运算符，能够让我们以更简洁的方法去使用及早返回模式。? 运算符被用于Result或Option之后，如果Result或Option的枚举成员是Ok或Some，这个表达式的值是Ok或Some中的值，否则将会令函数返回Err。

下面的例子是使用 ? 运算符简化上一节中的例子。

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)){

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

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

}

}

代码片段ch11\_9.rs

在代码段第8行，我们对open函数的返回结果使用了 ？运算符，open函数的返回结果类型是Result<File, Error>。如果打开成功，返回结果将是Ok(file),？运算符将会提取并返回file，file接着调用write\_fmt将字符串写入文件，如果打开失败将会返回错误。

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或不合法的情况时，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，就有可能破坏程序的状态。这样的错误或不合法的情况就被称为不可恢复错误。

对于不可恢复的错误，我们采用fail-stop的处理逻辑，即立即停止出现该错误对象的执行，并回收与其相关资源。因为这种方式能将错误的影响控制在最小范围，避免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。

Panic机制有两种触发方式，一种是错误地执行一些操作会触发panic，例如访问超过数组末尾的内容；另一种则是通过Rust提供的panic!宏显式地触发panic。

接下来两个例子分别对应两种触发方式：

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ch11\_10.rs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。

Panic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。

**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

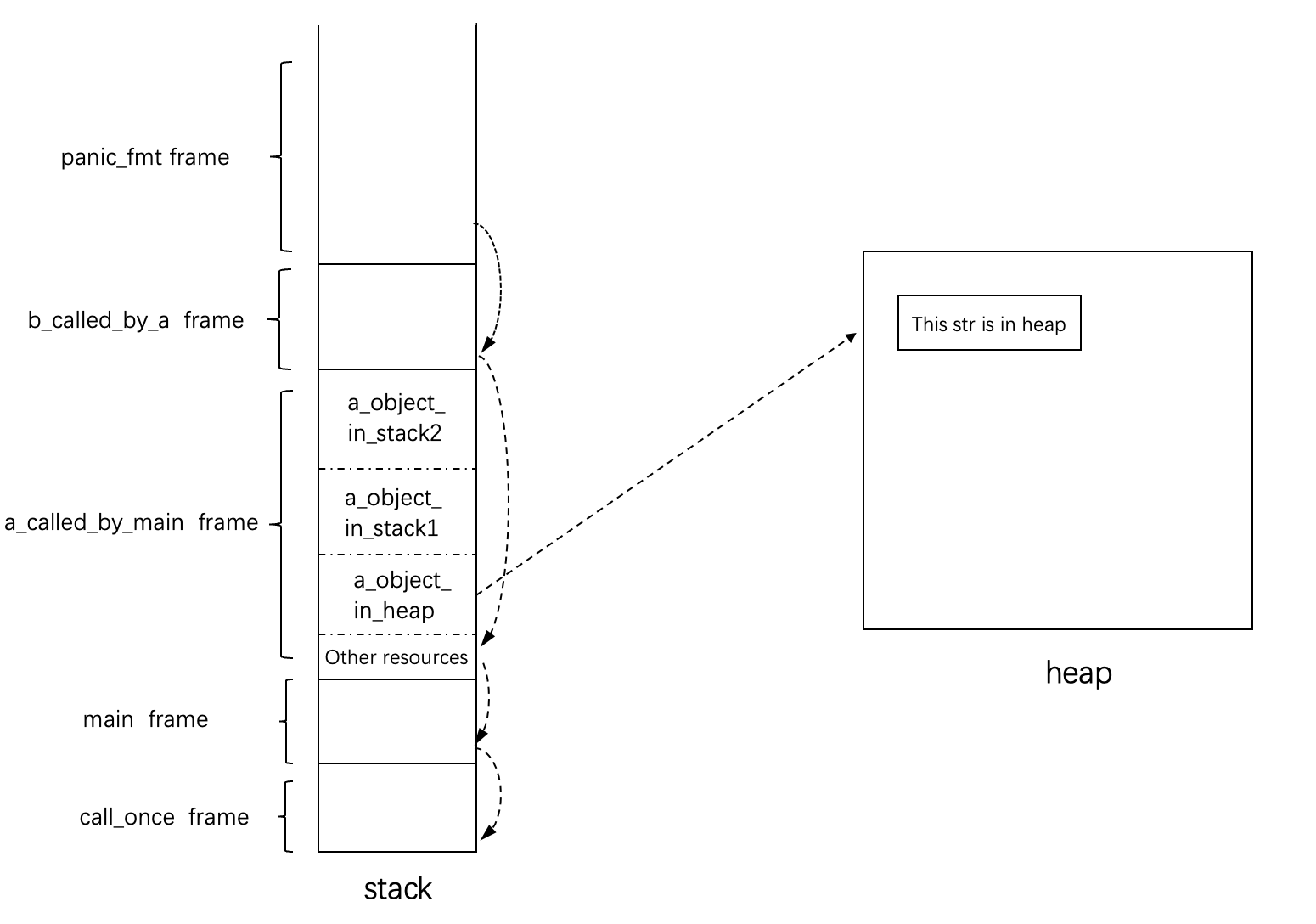
at /private/tmp/rust-20230210-6343-w92jca/rustc-1.67.1-src/library/core/src/ops/function.rs:507:5

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。

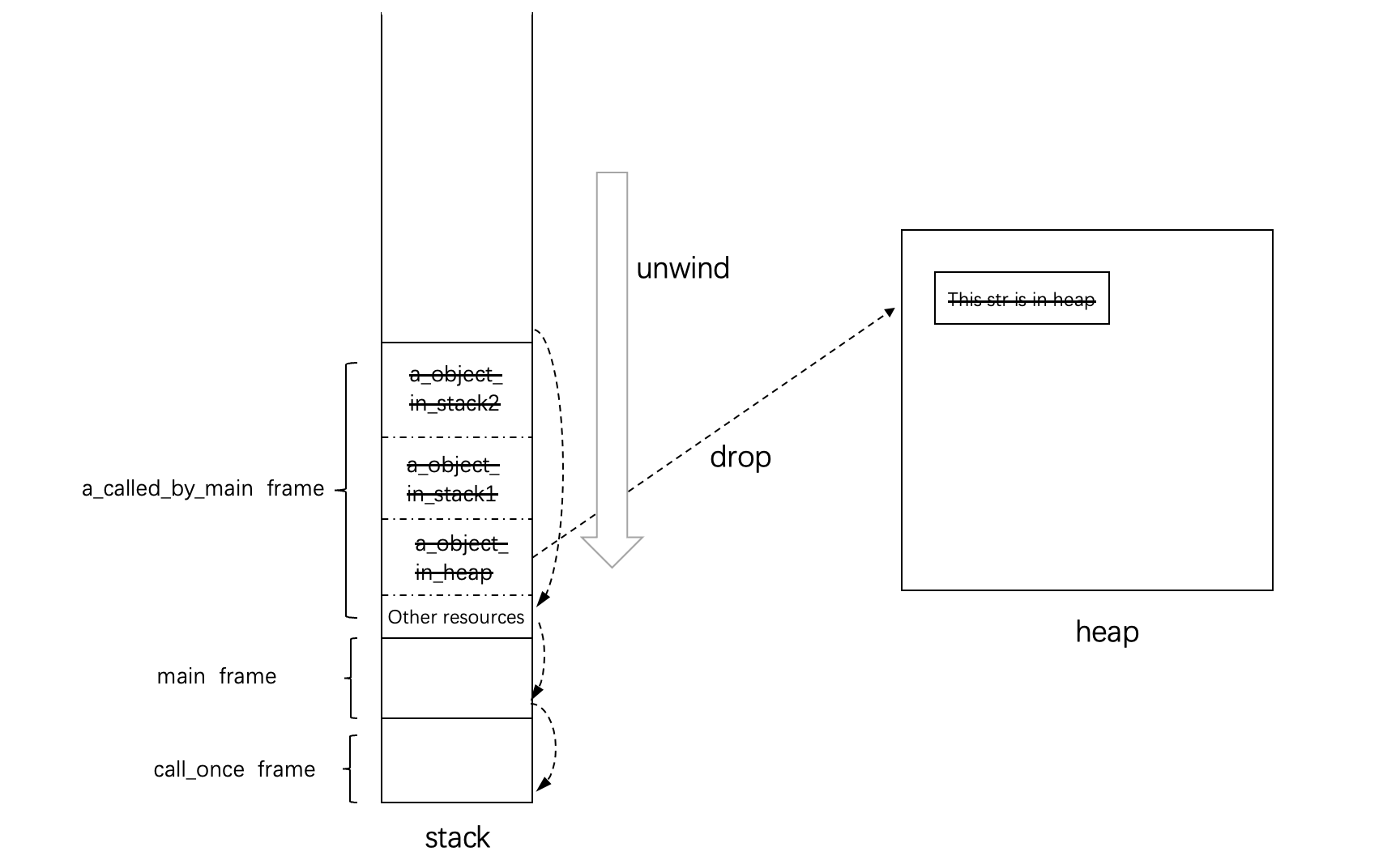
从backtrace的输出第二行开始，是发生panic时的程序栈帧情况，主要内容是函数的调用链。每个被调用的函数有一个编号，编号为 i 的函数被编号为i+1的函数调用。例如在上述例子中，编号为3的函数a\_called\_by main 被编号为 4 的main函数调用。

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。