Chapter 5 第5章 错误处理

错误处理是软件开发中的重要部分，它允许程序员拦截错误和失败的程序，并提供足够的错误信息来帮助程序员定位错误。因此，完善的错误处理能够提高代码的生产效率并提供更好的开发体验。Rust 中的一般错误处理准则如下：

- **可恢复错误**。错误可以被处理恢复：使用 Result、Option 和 enum。

例如：文件/目录不存在、授权失败、输入数字解析错误等。

- **不可恢复错误**。错误不可修复：使用 panic。

例如：内存耗尽、栈溢出、被零除、索引超出边界等。

Rust 并没有提供基于异常(exception)的错误处理机制，虽然 panic!宏在让进程挂掉时也能抛出堆栈信息，同时也可以用 std::panic::catch_unwind 来捕捉 panic，但是极其不推荐用来处理常规错误。

catch_unwind 一般是用来在多线程程序中将挂掉的线程兜住，防止因一个线程挂掉而导致整个进程崩溃，或者通过外部函数接口(FFI)与 C 程序交互时将堆栈信息兜住，防止 C 程序因看到堆栈而不知道如何处理。直接把堆栈信息丢给 C 程序属于 C 语言里的未定义行为(undefined behavior)。

另外，catch_unwind 并不保证能拦截住所有 panic，而只对通过 unwind 实现的 panic 有用。同时，因为 unwind 需要额外记录堆栈信息，对程序性能和二进制程序的大小有影响，所以在一些嵌入式平台上的 panic 并没有通过 unwind 实现，而是直接退出(abort)的，所以 catch_unwind 并不保证能捕捉到所有 panic。例如：

```
// chapter5\catch_unwind.rs
use std::panic;
fn main() {
   panic::catch_unwind(||{
      panic!("Panicking!");
   }).ok();

   println!("Running after panic.");
}
```

程序清单 5.1　catch_unwind 的例子

以上程序的输出如下：

```
Rust Programming\sourcecode\chapter5>.\catch_unwind.exe
thread 'main' panicked at 'Panicking!', .\catch_unwind.rs:7:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
Running after panic.
```

和其他允许抛出异常来进行错误处理的编程语言相比，Rust 能够精确地返回错误。本节

介绍 Rust 编程语言提供的返回值处理以及常用的错误处理机制：

- Option 和 Result 类型
- 基于 Option 和 Result 类型的模式匹配

Rust 编程语言还提供了一些处理错误的帮助方法，示例如下：

- try!宏
- ?运算符
- panic
- 定制错误和 error trait
- 组合子(combinators)
- from trait
- carrier trait

5.1 对象解封

链式调用的缺点是无法在出错时提前返回。如果要提前返回，必须从组合子跳出，然后再跳出外层函数，这是普通函数无法做到的。如果对程序有绝对的自信，可以直接解封(unwrap)函数返回值而不做错误处理，即直接调用 unwrap()方法，示例如下：

```
fn string_to_integer() ->i32{
    let strnumber:&str ="200";

    // 确信 number_str 只包含数字,不包含非法字符,所以直接解封 parse 函数的返回
    let number:i32 =strnumber.parse() .unwrap() ;
    return number;
}
let result =string_to_integer() ;
println!("{} ", result) ; // 200
```

如果函数返回值发生错误，直接解封会导致 panic。

5.2 Expect()

expect()方法类似 unwrap()，但是它允许我们设置一个错误信息。因为 expect()和 unwrap()完全不做错误处理，一旦出错，程序就会因 panic 退出。所以，通常是在调试程序时为了方便才使用 unwrap()和 expect()方法，不推荐在真正运行的代码里使用。例如：

```
fn string_to_integer( strnumber: &str) {
    let strnumber:&str ="a150"                  // 此处字母 a 会导致下面的 parse 失败
    let number:i32 =strnumber.parse() ;

    number.expect(" Invalid digit in string" ) ;// 设置错误信息
}
```

上面的程序会导致一个运行错误(panic)如下：

```
thread 'main' panicked at 'Invalid digit in string: ParseIntError{ kind: InvalidDigit} '
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

5.3　Option 类型

Option 类型的变量可以有值，也可以没有值。这个在调用函数或者函数成功或失败返回时是非常有用的(例如在用 C 语言进行链表查询时，如果找到，则返回对象指针；如果没有找到，则返回 null 指针)。这个特性非常像 C 语言里的联合(union)。Option 的定义如下：

```
enum Option<T>{
    Some(T),
    None,
}
```

下面的例子演示了如何获取向量里的第一个元素，以及可能的错误处理方法：

```
fn first_item<T>(v: &Vec<T>) ->Option<T>
    where T: Clone{          // T 必须是可克隆的
   if v.len() >0{            // 如果向量长度大于 0
      Some(v[0].clone())     // 则克隆第一个元素，返回
   } else{
      None                   // 否则返回 None
   }
}
```

Option 提供了 is_some 函数和 is_none 函数，它们用来快速确定 Option 里是有值还是没有值(或者说是一个坏值)，如表 5.1 所示。

表 5.1　Option 的相关的方法

方　法	描　述	返 回 类 型
and	测试两个 Option 类型是否不是 None	Option＜U＞
and_then	链式 Option	Option＜U＞
expect	如果 Option 为 None，则引发 panic	T
filter	用一个断言来过滤 Option	Option＜T＞
flatten	去除嵌套的 Options	Option＜T＞
is_none	检查 Option 是否是 None	bool
is_some	检查 Option 是否是 Some＜T＞	bool
iter	迭代访问或者空	一个迭代子
iter_mut	可变迭代访问或者空	一个迭代子
map	将值转换成另一个值	Option＜U＞
map_or	将值转换成另一个值	U
map_or_else	将值转换成另一个值	U
ok_or	将 Option 转换为 Result	Result
ok_or_else	将 Option 转换为 Result	Result
or	如果前面的 Option 为 None，则返回 or 后面的新值	Option＜T＞
or_else	Provide a value if None	Option＜T＞
replace	Change the value to a Some while returning previous value	Option＜T＞
take	Change the value to None while returning original	Option＜T＞
transpose	将 Option 的 Result 转换成 Result 的 Option	Result，E＞
unwrap	获取值	T
unwrap_or	获取值	T

续表

方　法	描　述	返回类型
unwrap_or_default	获取值	T
unwrap_or_else	获取值	T
xor	Return one of the contained values	Option<T>
zip	Options 合并	Option<(T,U)>

unwrap_or 提供了一个默认值(default),当值为 None 时返回 default。例如:

```
fn extension(file_name: &str) ->Option<&str>{
    file_name.find('.').map(|i| &file_name[i+1..])      // 如果文件名中有.
}
fn main() {
    assert_eq!(extension("game.exe").unwrap_or("rs"),"exe");
    assert_eq!(extension("game").unwrap_or("rs"),"rs"); // 如果没有后缀名(值为 None),则默
                                                        // 认返回 rs
}
```

and_then 看起来和 map 差不多,不过 map 只是把值 Some(t)重新映射了一遍,and_then 则会返回另一个 Option。如果我们在一个文件路径中找到它的扩展名,这时就会变得尤为重要。例如:

```
use std::path::Path;
fn file_name(file_path: &str) ->Option<&str>{
    let path =Path::new(file_path);
    path.file_name().to_str()
}
fn file_path_ext(file_path: &str) ->Option<&str>{
    file_name(file_path).and_then(extension)    // 如果正常,返回值是 extension 的返回值,否则返
                                                // 回 None
}
```

5.4 Result 类型

Result 类型类似 Option。Result 可能返回一个值,也可能没有返回值。因此 Result 通常用于函数的返回值,但是不同之处在于 Result 不返回 None 值,而是返回一个错误对象,封装了错误的详细信息。

Result 类型 Result<T,E>是一个有两个状态的枚举类型,定义如下:

```
enum Result<T, E>{      // T 可以是任何类型
   Ok(T),               // 正常的情况,返回 T
   Err(E),              // 错误的情况,返回错误信息
}
```

如果函数执行一切正常,则返回 Ok 分支,并附带返回函数的返回值。但是,如果函数执行异常,Result 就返回 Err 分支,并附带返回错误值。请看下面的例子:

```
use std::fs::File;
use std::io::{BufRead, BufReader, Error};

// 返回值 Result<String,Error>意味着在错误的情况下返回 std::io::Error; 在成功的情况下返回 String
// 类型的值
```

```
fn first_line(path: &str) ->Result<String, Error>{ // 正常返回第一行字符串,否则返回 std::
                                                   io::Error
    let f =File::open(path);

    match f{
        Ok(f) =>{                          // 如果打开文件句柄成功
            let mut buf =BufReader::new(f);
            let mut line =String::new();
            match buf.read_line(&mut line) {
                Ok(_) =>Ok(line),          // 如果读取第一行内容成功,则以字符串的形式返回
                Err(e) =>Err(e),           // 否则,返回详细的错误信息
            }
        }
        Err(e) =>Err(e),                   // 否则,返回详细的错误信息
    }
}
```

std::fs::File::open 会返回一个 Result < std::fs::File,std::io::Error >类型。也就是说,如果正常,则返回文件句柄;如果异常,则返回一个 I/O 错误。我们使用 match 语句:如果错误,则直接返回;否则通过 std::io::BufReader 类型读取文件的第 1 行。read_line 方法返回一个 Result < String,std::io: Error >类型,我们再次使用 match 语句:如果返回错误,则直接返回。注意:open 方法和 read_line 方法的返回类型是 std::io::Error。

Result 提供 is_ok 函数和 is_err 函数,它们用来快速确定 Result 里是有值还是没有值(或者说是一个坏值)。

我们使用组合子就可以实现上面的 match 语句的功能,不用取出计算结果,直接参与计算然后再取出。标准库为 Option 和 Result 提供了大量的组合子,这些组合子将计算→解封装→案例解析→计算→解封装的过程抽象出来,从代码上看可以使计算本身显得更紧凑,而不是被各种错误处理打断(如表 5.2 所示)。

表 5.2　Result 的相关的方法

方　法	描　述	返 回 类 型
and	测试两个结果是否都没有错误	Result
and_then	链式处理结果	Result
as_ref	将内部值转换为一个引用并返回封装值	Option < &T > Result < &T,&E >
as_mut	将内部值转换为一个可变引用并返回封装值	Option < &mut T > Result < &mut T,&E >
err	获取错误	Option < E >
expect	如果错误,则引发 panic	T
expect_err	如果成功,则引发 panic	E
is_err	检查 Result 类型返回是否错误	bool
is_ok	检查 Result 类型返回是否成功	bool
iter	迭代访问或者空	一个迭代子
iter_mut	迭代可变访问或者空	一个迭代子
map	将值转换为另一个值	Result
map_err	将值转换为另一个值	Result
map_or	将值转换为另一个值	U
map_or_else	将值转换为另一个值	U
ok	将 Result 转换为 Option	Option < T >

续表

方　法	描　述	返回类型
or	如果错误,则返回一个新的 Result	Result
or_else	如果错误,则返回一个新的 Result	Result
transpose	将 Option 的 Result 转换为 Result 的 Option	Option<Result<T,E>>
unwrap	获取值	T
unwrap_err	获取错误	E
unwrap_or	获取值	T
unwrap_or_default	获取值	T
unwrap_or_else	获取值	T

在 Rust 的标准库中经常会出现 Result 的别名,用来默认确认其中 Ok(T)或者 Err(E)的类型。**这样能减少重复编码**。例如下例中的 Result<T>就是返回类型的别名:

```
use std::num::ParseIntError;
use std::result;
type Result<T>=result::Result<T, ParseIntError>; // 这里默认确定错误类型为 ParseIntError
fn double_number(number_str: &str) ->Result<i32>{
    unimplemented!();
}
```

这样,我们以后使用时就可以使用 Result<T>,而不是 result::Result<T,ParseIntError>。每次都输入 result::Result<T,ParseIntError>太复杂,也容易引入错误。

5.5 访问和变换 Option 和 Result 类型

Option 和 Result 类型都可以被认为是一个有 0 个或者 1 个值的容器,所以可以用与迭代子相关的功能来访问 Option 和 Result。我们可以对 Option 和 Result 使用 map 系列方法:

- map 调用一个函数来将一个正常值转换成另一个类型的正常值。这个组合子可用于简单映射 Some→Some 和 None→None 的情况。多个不同的 map()调用可以更灵活地链式连接在一起。

请看一个标准库里的 map 方法定义:map 将一个封装值(Option<T>类型的 self)传入一个函数(F)并返回一个新的、可能是不同类型的封装值(Some(f(x)))。所以,map 可以将 Option<T>转换为 Option<U>,或者从 Result<T,E>转换为 Result<U,E>。对于错误值,map 方法保留了错误值,不做任何转换。

Option 中,map 是 std 库中的方法[①]:

```
pub fn map<U, F>(self, f: F) ->Option<U>
   where
      F: FnOnce(T) ->U,
{
      match self{
         Some(x) =>Some(f(x)),
         None =>None,
      }
}
```

① https://doc.rust-lang.org/src/core/option.rs.html

Result 中的 map 方法签名[①]如下：

```
pub fn map<U, F: FnOnce(T) ->U>(self, op: F) ->Result<U, E>{
   match self{
      Ok(t) =>Ok(op(t)),
      Err(e) =>Err(e),
   }
}
```

- map_or 和 map_or_else 通过给错误值提供一个默认值，扩展了 map 的功能。
- map_err 经常用来将系统标准错误转换为自定义错误。例如：

```
let maybe_file: Result<File, std::io::Error>=std::fs::File::open("foo.txt");
let new_file: Result<File, MyError>=maybe_file.map_err(|error| Err(MyError::BadStuff));
```

5.5.1　用 map 替换 match

假如我们要在一个字符串中找到文件的扩展名，例如 game.rs 文件的后缀名为 rs，我们可以使用 std::str::find 方法来找到目标字符串在原字符串中的位置(如 rs 在 game.rs 中的位置)，其函数签名[②]如下：

```
pub fn find<'a, P>(&'a self, pat: P) ->Option<usize> where P: Pattern<'a>
```

下面我们通过 match 语句来从 Option 类型中取值：

```
fn main() {
    let file_name ="game.rs";
    match file_name.find('.') {
        None =>println!("找不到文件扩展名."),
        Some(i) =>println!("文件扩展名:{}", &file_name[i+1..]),
    }
}
```

对所有的 Option 都使用 match 语句会使得程序很臃肿，也容易引入错误。我们可以使用 map 简化上面的 match 语句：

```
// chapter5>.\mapmatch.exe
fn main() {
  let file_name ="game.rs";
  println!("文件扩展名:{}", extension(file_name).unwrap());
}

// 使用 map 取代 match
fn extension(file_name: &str) ->Option<&str>{
    file_name.find('.').map(|i| &file_name[i+1..])          // 如果文件名中有.,
}
Rust Programming\sourcecode\chapter5>.\mapmatch.exe
文件扩展名: rs
```

程序清单 5.2　map 取代 match 的例子

① https://doc.rust-lang.org/src/core/result.rs.html

② https://doc.rust-lang.org/std/primitive.str.html#method.find

5.5.2 逻辑组合子

Option 和 Result 都提供 and 方法和 or 方法,它们能以特定的逻辑关系连接两个值。

and 方法要求两个值都是正常值,否则结果就是错误值。例如:

```
// 如果 get_name(old_person) 返回正常值,我们才进行第二步 clone_person
let new_person =get_name(old_person) .and( clone_person(old_person,"Brad") );
// new_person 的值要么是 None,要么是 Some(Person)
```

and_then 允许将相关操作连接在一起,并把第一个操作的结果传送给下一个函数。例如:

```
use std::result::Result;
use std::fs::File;
fn read_first_line(file: &mut File) ->Result<String>{ ...}
let first_line: Result<String>=File::open("foo.txt") .and_then(|file|{
        read_first_line(&mut file)});
```

or 方法会检查第一个结果,如果它是正常值,则直接返回;否则返回第二个结果。

5.5.3 在 Option 和 Result 类型之间互相转换

ok_or 方法通过将错误值作为 Result 类型的第二个参数,从而将一个 Option 值转换成一个 Result 类型值。相似的还有 ok_or_else 方法。例如:

```
let res =opt.ok_or(MyError::new());            // Some(T) ->Ok(T)
let res =opt.ok_or_else(||MyError::new());     // None ->Err(E)
```

上面的语句演示了如何将 Option 类型的 Some(T)转换成 Result 类型的 Ok(T),将 Option 类型的 None 转换成 Result 类型的 Err(E)。

ok 方法通过消耗 self 丢弃 Err 值,从而将一个 Result 类型值转换成一个 Option 类型值,可以使用 match 语句来进行转换。例如:

```
match res{
   Ok(t) =>Some(t),  // Ok(T) ->Some(T)
   Err(_) =>None,    // Err(E) ->None
}
```

Result 类型还可以使用 ok 方法转换为 Option 类型。例如:

```
let opt =res.ok();
```

5.6 try!宏

Option 和 Result 类型通过组合子提供的链式调用无法在出错时提前返回。如果需要提前返回,则必须从组合子中跳出,然后再跳出外层函数,这是普通函数无法做到的。Rust 提供了比普通函数更原始的抽象——宏,它直接操作语法单元作为模板在编译时展开,使得我们可以将 return 塞到宏里面。而宏本身不是函数,所以可以直接提前返回。

try!宏的定义如下:

```
macro_rules! try{
    ( $ expr:expr) =>( match $ expr{
       $ crate::result::Result::Ok( val) =>val,
    $ crate::result::Result::Err( err) =>{ return
  $ crate::result::Result::Err( $ crate::convert::From::from( err) )
        }
    } )
}
```

可以看到，try!宏包括 convert::From::from 函数调用，所以在进行 try!宏调用时，错误类型只要实现了 From trait，就可以自动进行类型转换，而不用显式地通过 map_err 进行错误类型的转换。

对于下面的错误处理程序：

```
fn bytestring_to_string_with_match( str: Vec<u8> ) ->Result<String, FromUtf8Error>{
    let ret =match String::from_utf8( str) {
        Ok( str) =>str.to_uppercase() ,
        Err( err) =>return Err( err)
    };

    println!(" Conversion succeeded: {} ", ret) ;
    Ok( ret)
}
```

用 try!宏重写上面的程序如下：

```
fn bytestring_to_string_with_try( str: Vec<u8> ) ->Result<String, FromUtf8Error>{
    let ret =try!( String::from_utf8( str) ) ;// 直接使用 try!宏，错误直接返回，成功则执行下面的 println
    println!(" Conversion succeeded: {} ", ret) ;
    Ok( ret)
}
```

try!宏不能在 main()程序里使用。注意，在 Rust Edition 2018 以后，try 变成了一个保留关键字，从而 try!()也作废了。下面是用 cargo build 命令编译本书资源包中 7.6.2 节的 chrono 项目时的错误信息：

```
  Compiling chrono v0.1.0 ( E:\projects\Rust Programming\sourcecode\chapter7\chrono)
error: use of deprecated `try` macro
  -->src\main.rs:22:5
   |
22 |     try!( f.write_all( string.as_bytes() ) ) ;  // 我们使用了 try!
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   =note: in the 2018 edition `try` is a reserved keyword, and the `try!() ` macro is deprecated
```

在这种情况下，Rust 编译器建议我们用提早返回的“?”操作符(请参照 5.9 节)来代替。

5.7　panic!宏

在面对一个无法恢复的错误时，可以用 panic!宏来处理。类似于 C/C++和 Go 语言的 os.exit()。当执行 panic!宏时，程序会打印出一个错误信息，展开(unwind)并清理栈数据，然后会终止程序运行并退出。举例如下：

```
// chapter5\panicmacro.rs
use std::num::ParseIntError;
fn main() ->Result<(), ParseIntError>{
  let strnumber ="10a";                          // 字符串中带有字母,所以 parse 时会触发错误
  let number:i32 =match strnumber.parse() {
       Ok(number) =>number,
       Err(_) =>panic!("字符串中包含非数字字符")     // 异常退出程序
  };
  println!("{}", number);
  Ok(())
}
```

程序清单 5.3　panic!宏的例子

以上程序的输出如下：

```
Rust Programming\sourcecode\chapter5>.\panicmacro.exe
thread 'main' panicked at '字符串中包含非数字字符', .\panicmacro.rs:6:19
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

在上面的程序中,如果 strnumber 的 parse()返回错误,则 panic!("字符串中包含非数字字符")将被调用。

Rust 执行 panic!宏时会有两种模式。

- **栈展开与终止(unwind)**。

当出现 panic!时,程序默认会开始展开(unwinding),这意味着 Rust 会回溯栈并清理它遇到的每个函数的数据；会保存 RAII 不变量；运行析构函数(若实现了 Drop trait)；解开栈里的所有内容,保证内存被清除。这个回溯和清理的过程有很多工作；在此过程中,如果又发生 panic,程序将直接终止并退出。我们在 Cargo.toml 文件里定义 unwind 模式如下：

```
[profile.release]
panic ="unwind"
```

- **直接终止(abort)**。

这种模式会不清理数据就直接退出程序。程序使用的内存需要由操作系统来清理。如果我们需要项目的最终二进制文件尽可能小,则可以终止 panic 时展开的切换。通过在 Cargo.toml 的[profile]部分增加 panic='abort',例如,如果我们想要在发布模式中的 panic 时直接终止,可以这样做：

```
[profile.release]
panic ="abort"
```

我们可以设置 RUST_BACKTRACE 环境变量以得到一个回溯(Backtrace)。Backtrace 是一个包含执行到目前为止所有被调用的函数的列表。Rust 的 Backtrace 和其他语言中的一样：阅读 Backtrace 的关键是从头开始读,直到发现程序员编写的文件。这就是问题发生的地方：这一行往上是代码调用的代码；往下则是调用代码的代码,这些行可能包含核心 Rust 代码、标准库代码或用到的 crate 代码。下例演示了如何获取一个程序的 Backtrace 信息：

```
// chapter5/backtrace.rs
use std::thread;
fn alice() ->thread::JoinHandle<()>{
 thread::spawn(move ||{
       bob();
```

```
})
}
fn bob() {
 malice();
}
fn malice() {
 panic!("malice is panicking!");
}
fn main() {
 let child=alice();
 let _=child.join();
 bob();
 println!("This is unreachable code");
}
```

程序清单 5.4　backtrace 的例子

以上程序的输出如下：

```
rustprogram/sourcecode/chapter5# RUST_BACKTRACE=1 ./backtrace
thread '<unnamed>' panicked at 'malice is panicking!', backtrace.rs:12:2
stack backtrace:
   0: std::panicking::begin_panic
   1: backtrace::malice
   2: backtrace::bob
   3: backtrace::alice::{{closure}}
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace
thread 'main' panicked at 'malice is panicking!', backtrace.rs:12:2
stack backtrace:
   0: std::panicking::begin_panic
   1: backtrace::malice
   2: backtrace::bob
   3: backtrace::main
   4: core::ops::function::FnOnce::call_once
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace
```

5.8　From trait

首先看一下 From trait 的定义，它只有一个函数：

```
pub trait From{ fn from(T) ->Self;}
```

前面提到 try!宏包括调用 convert::From::from。所以，如果用户自己定义了一个错误，然后实现 From trait，将其他错误转换成自定义错误，就可以利用 try!宏实现自动调用，上层调用者得到的就是一个用户自定义的错误类型，而不用再通过 trait 对象间接得到错误类型。例如：

```
use std::fs::File;
use std::io::{self, Read};
use std::num;
use std::io;
use std::path::Path;
// 这里我们使用了派生 Debug 属性.这么做可以提供一个人类比较容易理解的 CliError 的错误描述
#[derive(Debug)]
enum CliError{
     Io(io::Error),
     Parse(num::ParseIntError),
```

```
}
impl From<io::Error> for CliError { // 为 CliError 实现 From trait,将 io::Errror1 转换为 CliError
    fn from(err: io::Error) ->CliError{
        CliError::Io(err)
    }
}
impl From<num::ParseIntError> for CliError{ // 将 num::ParseIntError 转换为 CliError
    fn from(err: num::ParseIntError) ->CliError{
        CliError::Parse(err)
    }
}
fn file_double_verbose<P: AsRef<Path>>(file_path: P) ->Result<i32, CliError>{
    // 下面的 File::open 可能引发的 io::Error 被转换为 CliError
    let mut file =try!(File::open(file_path).map_err(CliError::Io));
    let mut contents =String::new();
    // 下面的 file.read_to_string 可能引发的 io::Error 被转换为 CliError
    try!(file.read_to_string(&mut contents).map_err(CliError::Io));
    // 下面的 contents.trim().parse() 可能引发的 io::Error 被转换为 CliError
    let n: i32 =try!(contents.trim().parse().map_err(CliError::Parse));
    Ok(2 *n)
}
```

这里我们自定义了错误类型 CliError,分别为 io::Error 和 num::ParseIntError,实现了 From 这个 trait。这样,在调用 try!宏时,这两种错误类型都能转换成 CliError。这种做法既利用了 try!宏里面的 from 函数自动转换类型省掉了手写烦琐的 map_err,又可以在遇到错误时提前返回。同时,上层调用者又可以不用反射就得到具体的错误类型,以便根据情况做进一步处理。

5.9 问号(?)操作符

为了让程序的错误处理更便捷,Rust 引入了“?”操作符。下面的程序演示了一个标准的错误处理模式:对于每个函数调用,都要有一段错误处理程序来检查函数运行成功与否,并做相应处理。例如:

```
let x =function_that_may_fail();
let value =match x{
   Ok(v) =>value,
   Err(e) =>return Err(e);
}
```

在 Rust 引入“?”操作符后,程序显然简洁了很多:

```
let value =function_may_fail()?;
```

“?”运算符适用于函数返回类型为 Result<T,E>的场合,它把 Result<T,E>的返回变成了 T。

- **如果结果出错**,当前函数立即退出,返回 Err。这个是编译器自动处理的,不需要程序员写错误处理返回的代码,使程序看起来很简洁,也减少了程序员的编码量。
- **如果结果正确**,函数就会把结果解封装(以上面的程序为例,即从 OK 中解封,直接返回 v),继续执行下面的程序。

下面演示了“?”操作符的使用方法:

```
use std::fs::File;
use std::io::{ BufRead, BufReader, Error} ;
fn readfirstline( path: &str) ->Result<String, Error>{ // 想使用"?"操作符,返回的必须是 Result 类型
    let f =File::open( path) ?;       // 如果出错,直接返回相关错误,否则执行下一行代码
    let mut buf =BufReader::new( f) ;
    let mut line =String::new() ;
    buf.read_line( &mut line) ?;      // 如果出错,直接返回相关错误,否则执行下一行代码
    Ok( line)
}
```

5.10　Carrier Trait

当然,Rust 里面的问号(?)语法糖是通过 Carrier trait 来实现的,其定义如下:

```
pub trait Carrier{
    type Success;
    type Error;
    fn from_success( Self::Success) ->Self;
    fn from_error( Self::Error) ->Self;
    fn translate( self) ->T where T: Carrier;
}
```

只要实现了这个 trait,就可以使用问号语法糖。虽然问号语法糖已经在 stable 版里可用了,但是标准库里只对 Result 类型实现了这个 trait。如果用户要想让用户自定义的类型也能使用这个语法糖,还需要在 nightly 版本下使用。同时,因为标准库里只对 Result 类型实现了这个 trait,根据 Rust 语法的规定,用户是无法在 Option 类型上使用问号语法糖的。

5.11　自定义错误类型

在 Result<T,E>类型里,错误类型 E 可以是任何类型。Rust 允许自定义错误类型,但是使用字符串(String)作为错误类型实际上是存在一些局限的。一般而言,一个"良好"的错误类型具有友好的错误类型标准:

- 使用相同类型来表达不同的错误;
- 给用户提供友好的错误信息;
- 方便和其他类型比较;
- 能够保存详细的错误信息。

字符串(String 类型)可以满足前两条标准,但后两条无法满足。这使得 String 类型错误既难以创建,也难以达到要求。

我们推荐使用实现了 std::error::Error trait 的类型。这样做的好处在于:由于错误类型都基于 std::error::Error,因此用户能更好地处理错误并进而对错误进行聚合处理。trait 可以被认为是一个需要实现相应方法的接口声明。下面是 Error 这个 trait 的定义:

```
trait Error: Debug +Display{
    fn source( &self) ->Option<&( dyn Error +'static) >{ None} ;
    fn backtrace( &self) ->Option<&Backtrace>{ None} ;
}
```

backtrace 方法仅定义在 Rust 编译器的 nightly 版本中。在写这本书时,只有 source 方法

被定义在 stable 版本中。source 用来返回当前错误的前一个错误。如果不存在前一个错误，则其值为 None。返回 None 是 source 方法的默认实现。

实现了 Error trait 的类型必须实现 Debug 和 Display trait。错误可以是一个枚举类型。下面是一个针对读取文件数据库的基于枚举类型的错误处理程序：

```
use std::fmt::{Result, Formatter};
use std::fs::File;

#[derive(Debug)]
enum MyError{                                          // 使用枚举类型作为错误类型
  DatabaseNotFound(String),
  CannotOpenDatabase(String),
  CannotReadDatabase(String, File),
  In(std::io::Error),
  Out(std::io::Error),
}
impl std::error::Error for MyError{}                   // 为 MyError 实现标准的 Error trait

// 必须自己实现 Display trait
impl std::fmt::Display for MyError{
   fn fmt(&self, f: &mut Formatter<'_>) ->Result{
      match self{
         Self::DatabaseNotFound(ref str) =>write!(f,"File `{}` not found", str),
         Self::CannotOpenDatabase(ref String) =>write!(f,"Cannot open database:{}", str),
         Self::CannotReadDatabase(ref String, _) =>write!(f,"Cannot read database:{}",
str),
      }
   }}
```

上面的程序声明了一个可能发生错误的枚举类型(MyError)。枚举的每个错误类型还可以接收参数，如文件名。使用 derive (debug)属性将为 MyError 枚举类型自动引入 Debug trait 功能，非常方便。但是对于 Display trait，就不能简单地用 derive 引入，必须自己实现。为了和其他错误处理代码兼容，上面的程序为 myError 实现了 std::error::Error trait。

下面演示了一个枚举类型的自定义错误类型，并实现了 Display trait：

```
use std::error::Error;
use std::fmt;
use std::convert::From;
use std::io::Error as IoError;
use std::str::Utf8Error;

#[derive(Debug)]                              // 可以使用"{:?}"格式指定符
enum CustomError{
    Io(IoError),
    Utf8(Utf8Error),
    Other,
}

impl fmt::Display for CustomError{            // 可以使用"{}"格式指定符
    fn fmt(&self, f: &mut fmt::Formatter) ->fmt::Result{
        match *self{
            CustomError::Io(ref cause) =>write!(f,"I/O Error:{}", cause),
            CustomError::Utf8(ref cause) =>write!(f,"UTF-8 Error:{}", cause),
            CustomError::Other =>write!(f,"Unknown error!"),
        }
    }
}
```

```
impl Error for CustomError{                    // 支持 Error
    fn description(&self) ->&str{
        match *self{
            CustomError::Io(ref cause) =>cause.description(),
            CustomError::Utf8(ref cause) =>cause.description(),
            CustomError::Other =>"Unknown error!",
        }
    }
    fn cause(&self) ->Option<&Error>{
        match *self{
            CustomError::Io(ref cause) =>Some(cause),
            CustomError::Utf8(ref cause) =>Some(cause),
            CustomError::Other =>None,
        }
    }
}
// 支持将系统标准错误转换成我们自定义的错误
// 可以在 try! 中使用这个 trait
impl From<IoError>for CustomError{          // 将 IoError 转换为 CustomError
    fn from(cause: IoError) ->CustomError{
        CustomError::Io(cause)
    }
}
impl From<Utf8Error>for CustomError{        // 将 Utf8Error 转换为 CustomError
    fn from(cause: Utf8Error) ->CustomError{
        CustomError::Utf8(cause)
    }
}
```

另外，也可以通过结构结合 thiserror trait 来自定义错误类型。例如：

```
use thiserror::Error;
#[derive(Error, Debug)]                              // 自动派生 Error
#[error("{message:}({line:},{column})")]             // 更详细、更精确的错误信息
pub struct JsonError{
    message: String,
    line: usize,
    column: usize,
}
// JsonError 应该能够打印
impl fmt::Display for JsonError{
    fn fmt(&self, f: &mut fmt::Formatter) ->Result<(), fmt::Error>{
        write!(f,"{}({}:{})", self.message, self.line,
        self.column)
    }
}
// JsonError 应该实现 std::error::Error trait,
impl std::error::Error for JsonError{...}
```

总结

- 为自定义的错误类型实现 std::Error::Error trait。
- 自定义错误类型最好支持实现 Send + Sync + 'static。
- 为自定义的错误类型实现 Display 和 Debug trait。
- 可以通过枚举、结构等类型来实现自定义的错误类型。
- 可以通过 trait 对象来实现自定义的错误类型。
- 为自定义的错误类型实现 From 和 Into trait。

自己定义一个错误类型其实比较复杂，但是可以基于一些流行的第三方 crate 来编写错误处理程序。这些 crate 可以让我们更容易地编写和使用自定义错误。

5.12 Error Crates

处理错误时,每次写代码处理并返回 Result 的 Err 分支很麻烦,Error trait 的功能也比较贫乏。本节描述一些可以增强 Error crate 功能的选项。假设我们要实现下面的函数(功能是读取第 1 行的内容):

```
fn first_line(path: &str) ->Result<String, FirstLineError>{ ...}
```

基于上面的案例,下面描述如何用本节介绍的 crate 来实现 FirstLineError。最基本的 FirstLineError 实现是下面的枚举:

```
enum FirstLineError{
    CannotOpenFile{ name: String},
    NoLines,
}
```

5.12.1 failure crate

failurecrate[①] 里有两个概念:Fail trait 和一个 Error 类型。Fail trait 是一个用户定制的错误类型,它包含更丰富的错误信息,可以用来在库包开发中定义新的错误类型。Error trait 是 Fail 类型的封装器,而 Fail 类型可以用来设计高级的错误类型。例如:一个打开文件的错误会导致数据库打开错误,那么可以把打开文件的错误(底层错误)链接到一个数据库打开错误(面向用户的错误类型),用户可以处理数据库打开错误并深入调查,获取程序员所需的最原始的错误信息。一般来说,crate 的开发者会使用 Fail,而 crate 的使用者会使用 Error 类型。

如果打开 Failure Crate 的回溯属性 Backtrace,并且将 RUST_BACKTRACE 环境变量设置为 1,那么 Failure 也将支持回溯。

下面用 Failure crate 来改写 FirstLineError 错误类型。例如:

```
use std::fs::File;
use std::io::{BufRead, BufReader};
use failure::Fail;
#[derive(Fail, Debug)]
enum FirstLineError{
    #[fail(display ="Cannot open file `{}`", name)]
    CannotOpenFile{ name: String},
    #[fail(display ="No lines found")]
    NoLines,
}
fn first_line(path: &str) ->Result<String, FirstLineError>{
    let f =File::open(path).map_err(
        |_| FirstLineError::CannotOpenFile{    // 闭包将错误替换成一个 FirstLineError 错误值
        name: String::from(path),              // 如果 Result 类型错误,则返回错误的详细信息 path
    }) ?;
    let mut buf =BufReader::new(f);
    let mut line =String::new();
    buf.read_line(&mut line)
        .map_err(|_| FirstLineError::NoLines) ?; // 闭包将错误替换成一个 FirstLineError 错误值
    Ok(line)
}
```

① https://github.com/rust-lang-nursery/failure

derive 宏自动派生为 FirstLineError 实现了 Fail 和 Debug traits。FirstLineError 枚举类型定义中使用了 fail 属性(#[fail(...)]),为 FirstLineError 的每个分支(详细的错误类型)提供了更详细的错误信息。但是,File::open 和 BufRead::read_line 方法返回的是 std::io::Error 类型,而不是 FirstLineError 类型。我们需要使用 Result 的 map_err 方法来将具体的错误类型转换为 FirstLineError 类型,以便 first_line 函数统一返回相同的错误类型:

(1) 如果 File::open 返回结果错误,则 map_err 会调用一个闭包;

(2) 在这个闭包里,我们将错误替换成一个 FirstLineError 错误值;

(3) 通过(1)和(2),我们通过调用 map_err 把 File::open 返回的 Result<(),std::io::Error>类型的值转换成 first_line 函数统一返回所需的 Result<(),FirstLineError> 类型的值。如果返回的 Result 类型是错误的,闭包就可以提供与 Err 分支相关的详细信息;

(4) buf.read_line 的处理类似。

因为 File::open 的返回值是一个 Result 类型,所以我们可以使用“?”操作符在错误发生时立刻返回。外部针对 first_line 函数调用的错误处理代码就可以编写如下:

```
match first_line("foo.txt") {
    Ok(line) =>println!("First line:{}", line),
    Err(e) =>println!("Error occurred:{}", e),  // 直接使用 first_line 函数返回的
                                                // FirstLineError 错误值
}
```

Failure 允许我们在处理过程中使用 ensure!宏生成和 failure::Error 兼容的错误。例如:

```
use failure::{ensure, Error};
fn check_even(num: i32) ->Result<(), Error>{
   ensure!(num % 2 ==0, "num 不是偶数");  // 如果不是偶数,则返回和 failure::Error 兼容的错误
   Ok(())
}
fn main() {
   match check_even(41) {
      Ok(()) =>println!("偶数!"),
      Err(e) =>println!("{}", e),
   }
}
```

使用 failure crate 中的 format_err!宏可以在处理过程中动态生成基于字符串的错误。例如:

```
let err =format_err!("File not found:{}", file_name);  // err 是基于字符串的 failure::Error 类
                                                       // 型值
```

bail!宏等价于 format_err! 加上错误并返回。例如:

```
bail!("File not found:{}", file_name);
```

5.12.2　snafu crate

snafu[①] 和 failure 类似,但是它能更精准地报告实际的错误。上节的程序使用 map_err 将 std::io::Error 转换成自定义的 FirstLineError 值。snafu 提供了 context 方法,让程序员能

① https://docs.rs/snafu/latest/snafu/

够传递更精确的错误信息。下面使用 snafu 来重新定义错误类型。例如:

```
use std::fs::File;
use std::io::{ BufRead, BufReader};
use snafu::{ Snafu, ResultExt};          // 导入 snafu crate 中的类型
#[derive( Snafu, Debug)]
enum FirstLineError{
    #[snafu( display("Cannot open file{} because:{} ", name, source) )]
    CannotOpenFile{
        name: String,
        source: std::io::Error,          // source 字段包含具体的错误类型信息
    },
    #[snafu( display("No lines found because:{} ", source) )] // 加上了 source,提供更详细精准的错
                                                              // 误信息
    NoLines{ source: std::io::Error},
}
fn first_line( path: &str) ->Result<String, FirstLineError>{
    let f =File::open( path) .context( CannotOpenFile{
        name: String::from( path) ,
    }) ?;
    let mut buf =BufReader::new( f) ;
    let mut line =String::new() ;
    buf.read_line( &mut line) .context( NoLines) ?;
    Ok( line)
}
```

为了使用 context(),必须定义一个 source 域。注意,我们直接使用了 CannotOpenFile 而不是 FirstLineError::CannotOpenFile。source 域是自动设定的。如果不希望使用 source 记录真正的错误源,我们可以使用其他名字,只要它和#[snafu(source)]中指定的名字一致即可。另外,如果已经有一个域名叫作 source,而且我们不希望它被认为是 snafu 的 source 域,则可以使用#[snafu(source(false))]来标明。

snafu 也支持 backtrace 域。#[snafu(backtrace)]表明希望能报告错误发生时的 backtrace。同时,我们也可以使用 ensure!宏,其功能和 failure 类似。

5.12.3 anyhow crate

anyhow crate[①] 提供了 anyhow::Result<T>。anyhow::Result<T>是 Result<T, anyhow::Error>的类型别名。可以看到,anyhow::Result<T>不关心具体的错误类型,所以 anyhow::Error 称为不透明错误(Opaque Error)。程序员可以通过使用它来提供动态的错误处理,它可以接受任意类型的错误,并且使用 anyhow!宏可以创建一个特定的错误。例如:

```
let err =anyhow!("File not found:{} ", file_name) ; // error 是基于字符串的 anyhow::Error 类型值
```

同时,anyhow crate 还定义了 bail!宏和 ensure!宏。anyhow 宏的结果还可以链式调用 context()方法。例如:

```
let err =anyhow!("File not found:{} ", file_name)
            .context("Tried to load the configuration file") ;  // 提供一些错误的上下文信息
```

下面的程序演示了如何使用 anyhow。

首先要修改 Cargo.toml:

① https://docs.rs/anyhow/latest/anyhow/

```
[dependencies]
anyhow = "1"
```

程序如下：

```
use anyhow::Result;
#[derive(Debug)]
enum FirstLineError{
   CannotOpenFile{
      name: String,
      source: std::io::Error,
   },
   NoLines{
      source: std::io::Error,
   },}
impl std::error::Error for FirstLineError{}     // 实现 Error trait
impl std::fmt::Display for FirstLineError{      // 实现 Display trait
   fn fmt(&self, f: &mut std::fmt::Formatter<'_>) ->std::fmt::Result{
      match self{
         FirstLineError::CannotOpenFile{ name, source} =>{
            write!(f,"Cannot open file `{}` because:{}", name, source)
         }
         FirstLineError::NoLines{ source} =>{
            write!(f,"Cannot find line in file because:{}", source)
         }
      }
   }
}
fn first_line(path: &str) ->Result<String>{   // Result<String>隐藏了详细的错误类型
   let f =File::open(path).map_err(|e| FirstLineError::CannotOpenFile{
      name: String::from(path),
      source: e,
   })?;
   let mut buf =BufReader::new(f);
   let mut line =String::new();
   buf.read_line(&mut line)
      .map_err(|e| FirstLineError::NoLines{ source: e})?;
   Ok(line)
}
```

anyhow crate 没有定义 display trait。为了打印具体的错误信息，必须自己实现。在不同模块之间传送错误，map_err 组合子需要返回相应的错误信息。上面的例子只返回了错误信息 Result<String>，而没有返回特定的错误类型。

anyhow::Error 是动态错误类型的一个封装。anyhow::Error 提供外在的、额外的上下文信息，以使错误的返回含有丰富的信息，从而帮助程序员快速定位错误。anyhow::Error 非常像 Box<dyn std::error::Error>，但是二者也是有区别的：

- anyhow::Error 要求错误是 Send、Sync 和'static；
- anyhow::Error 保证即使其基于的错误类型并不提供 backtrace，backtrace 也仍然可用；
- anyhow::Error 被实现为一个瘦指针，其大小为 1 个字。

anyhow crate 的用法示例如下(有删节)：

```
use anyhow::Context;
// ...
pub async fn subscribe(/**/) ->Result<HttpResponse, SubscribeError>{
     // ...
     let mut transaction =pool
```

```
            .begin()
            .await
            .context("Failed to acquire a Postgres connection from the pool")?;
                                                        // 如果错误,则返回此信息

        let subscriber_id = insert_subscriber(/**/)
            .await
            .context("Failed to insert new subscriber in the database.")?;
                                                        // 如果错误,则返回此信息
        // ...

        store_token(/**/)
            .await
            .context("Failed to store the confirmation token for a new subscriber.")?;

        transaction
            .commit()
            .await
            .context("Failed to commit SQL transaction to store a new subscriber.")?;

        send_confirmation_email(/**/)
            .await
            .context("Failed to send a confirmation email.")?;  // 如果错误,则返回此信息
        // ...
    }
```

context 方法有以下两个功能:

- 将方法返回的错误转换为 anyhow::Error;
- 给调用者提供丰富的上下文信息,方便错误定位。

5.12.4 thiserror crate

使用 thiserror crate[①] 能让程序员很容易地定义错误类型,并且可以和 anyhow 连用,它使用#[derive(thiserror::Error)]来自动生成 Display 和 std::error::Error 的代码。使用 thiserror crate 中的#[from]属性更容易链接低级错误。例如:

```
#[derive(Error, Debug)]
enum MyError{
    #[error("Everything blew up!")]
    BlewUp,

    #[error(transparent)]
    IoError(#[from] std::io::Error)
}
```

上面的程序自动将 std::io::Error 类型转换为 MyError::IoError。

下面是一个比较完全地结合了 anyhow 和 thiserror crate 的代码示例:

```
use std::fs::File;
use std::io::{BufRead, BufReader};
use anyhow::Result;
use thiserror::Error;
#[derive(Debug, Error)]                    // 为 FirstLineError 自动派生 thiserror::Error
enum FirstLineError{
```

① https://docs.rs/thiserror/1.0.47/thiserror/

```
    // 下面的 error 属性定义详细的错误信息,自动将 std::io::Error 转换为 MyError::IoError
    #[error("Cannot open file `{name}` because:{source}")]
    CannotOpenFile{
        name: String,
        source: std::io::Error,
    },
    // 下面的 error 属性定义详细的错误信息,自动将 std::io::Error 转换为 MyError::IoError

    #[error("Cannot find line in file because:{source}")]
    NoLines{
        source: std::io::Error,
    },}
fn first_line(path: &str) ->Result<String>{   // 此处使用 anyhow::Result
    let f =File::open(path).map_err(|e| FirstLineError::CannotOpenFile{
                    name: String::from(path),
                    source: e,
                }) ?;
    let mut buf =BufReader::new(f);
    let mut line =String::new();
    buf.read_line(&mut line)
        .map_err(|e| FirstLineError::NoLines{ source: e}) ?;
    Ok(line)
}
```

上面的代码用 anyhow 处理 Result 类型,而用 thiserror 处理 error 类型。#[error(...)]用来定义错误信息,非常方便。

5.12.3 节中介绍的 anyhow crate 隐藏了错误的细节,所以称之为不透明的错误。那么什么时候用 anyhow,什么时候用 thiserror 呢? 一言以蔽之,就是编写应用程序时用 anyhow,编写库程序时用 thiserror。但是这有点过于简单化、绝对化。我们需要搞清楚我们的设计意图。

- **基于返回的失败模式,我们希望调用者做不同处理。**
  使用枚举类型的错误类型,并让其匹配不同的错误分支。使用 thiserror 会节省很多模板代码。
- **如果失败发生,我们希望调用者放弃,并向操作员或者用户汇报错误。**
  这种情况下,使用不透明的错误类型不要给调用者通过程序获得错误内部信息的机会。为方便起见,可以使用 anyhow 或者 eyre[①]。

大多数 Rust 库都返回一个枚举类型的错误,而不是 Box<dyn std::error::Error>(如 sqlx::Error)。库的开发者无法假设用户的意图,所以采用枚举类型的错误会给予用户更多的控制权。但是,自由不是没有代价的:如果这么做,接口就会变得复杂,用户需要过滤多个错误分支以找到需要特殊处理的错误分支。所以在设计时要仔细考虑用户案例以及假设,选择合适的错误类型。有时甚至 Box<dyn std::error::Error>或者 anyhow::Error 对库开发而言反而是最适合的。

5.13　Main 函数中的错误返回

在 Rust 2015 版本中,main 函数并不能返回 Result<T,E>。Rust 2018 版本新增了一个功能:允许 main 函数返回 Result 类型。如果 main 函数返回一个 Err 类型,那么程序就将返

① https://docs.rs/eyre/latest/eyre/

回一个非 0 的值,被操作系统用来警示程序执行失败,并使用 Debug trait 打印错误信息。如果需要 Display 功能,就需要将主要功能(如下面的 run 函数)放入 main 函数中运行,并在 main 函数中使用 println! 打印结果。例如:

```
fn main() ->i32{
   if let Err(e) =run() {
      println!("{}", e);
      return 1;
   }

   return 0;
}
fn run() ->Result<(), Error>{ ...}     // Rust 2018 版本
```

如果 Debug trait 满足需要,就可以使用下面的 main 函数声明:

```
fn main() ->Result<(), Error>{ ... }
```

main 函数允许返回 Result 类型。更准确地说,main 函数允许返回任何实现了 Termination trait 的类型。程序通过 Termination trait 获得返回错误号:编译器会对 main 函数返回的类型调用 report()以获得错误号。例如:

```
trait Termination{
       fn report(self) ->i32;
}
```

我们可以这样写 main 程序:

```
// chapter5/termination.rs
fn main() ->Result<(), &'static str>{
    let s =vec!["coke","7up"];
    let third =s.get(3).ok_or("I got only 2 drinks")?;
    Ok(())
}
```

程序清单 5.2 Termination 的例子

以上程序的输出如下:

```
Rust Programming\sourcecode\chapter5>./termination
Error:"I got only 2 drinks"
```

5.14 错误传递

Rust 的“?”操作符的功能是解封返回值,如果错误则提早返回。“?”操作符是通过 From trait 执行类型转换的。一个函数如果返回 Result<T,E>,而且 E:From<X>(从类型 X 可以生成 E),就可以对任何 Result<T,X>使用问号操作符。

使用“?”操作符做错误提早返回,我们需要通过某种方法来界定错误处理的范围。例如:

```
fn do_the_thing() ->Result<(), Error>{
    let thing =Thing::setup()?;
    // .. code that uses thing and ? ..
    thing.cleanup();
```

```
    Ok( () )
}
```

上面的错误处理是不干净的：在 setup()?和 cleanup()?中间任何导致提早返回的情况都会跳过我们的 cleanup 代码，这也是官方准备引入 try/throw/catch 语法来处理错误的原因。不过和一般语言的 try/throw/catch 机制不同，Rust 的 try/throw/catch 语法仍然是基于 Option/Result 的，这要求 Result 在穿越多层调用时能够将类型转换成相同类型的 Result。目前 try 还不稳定，nightly 里有 try_catch 的特性开关。具体细节这里不再展开，感兴趣的读者可以参考 RFC 说明[①]。

5.15　函数中处理多种错误类型

在编程实践中，一个函数中可能要使用多个外部库包，因此可能和多种 Error 类型打交道。例如一个函数中可能需要读取文件、访问数据库、进行网络通信等。每个功能的异常都会有一个 Error 类型，再加上自定义的 Error 类型，都需要转换为一个通用的 Error 类型，这样做是为了方便外部的统一错误处理。而难题就在于：函数如何整合多种错误类型为一个可以统一返回的错误类型。

传统的面向对象编程语言处理上面的情况的方法是定义一个 Error 的基类，所有其他的 Error 都派生自这个基类。那么，我们在指定函数的错误返回类型时就可以直接指定这个基类。具体的错误内容可以通过函数多态(polymorphism)和重写/覆盖(override)来获取(如图 5.1 所示)。

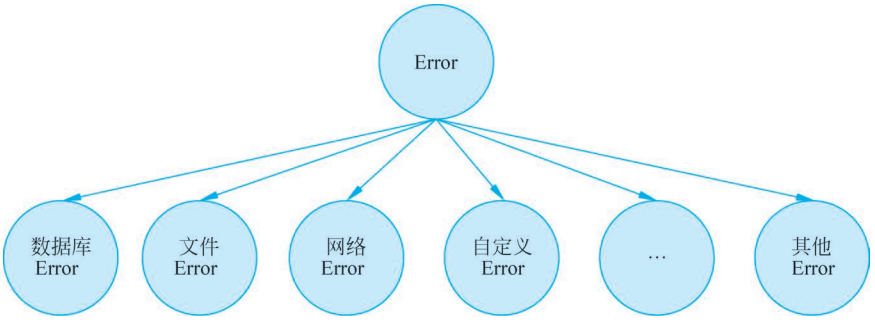

图 5.1　错误类型的继承关系

Rust 语言中，一个比较简单的方案是使用 std::error::Error 作为基类：所有标注库里的错误类型都可以转换成 Box < dyn std::error::Error + Send + Sync + 'static >。其中：

- dyn std::error::Error 标识任何错误；
- Send+Sync+'static 标识该错误类型是跨线程安全的。

```
type Result<T>=Result<T, Box<dyn Any +Send +'static>>;
```

方便起见，还可以定义别名如下：

```
type GenericError =Box<dyn std::error::Error +Send +Sync +'static>;
type GenericResult<T>=Result<T, GenericError>;
```

① https://github.com/rust-lang/rfcs

在函数中,每个功能调用都可能有其专有的错误类型。函数中如果有多个错误类型,返回 Result 就不可能了。这种情况下,我们就需要使用 trait 对象,但是 trait 对象是有潜在的问题的。使用 trait objects 也称为类型消除(type erasure),顾名思义,如果使用了 trait 对象,Rust 编译器就不再知道错误的精准来源。所以,使用 Box < dyn Error >作为错误返回 Result 的分支会导致丧失错误的具体信息。错误起源转换成了同样的类型。

Rust 中使用 Trait 和 Trait Object 实现上面的方案。

5.16 处理特定的错误类型

针对函数返回上节定义的 GenericResult 类型,如果我们只希望处理某个特定的错误,而放行其他错误,则可以使用泛型方法 error. downcast_ref::< ErrorType >()。如果当前错误是我们希望处理的错误类型,error. downcast_ref::< ErrorType >()就会以引用的方式借用错误对象。例如:

```
loop{
   match compile_project() {    // 编译项目
      Ok( () ) =>return Ok( () ),
       Err(err) =>{
         // 我们只想处理 MissingSemicolonError
         if let Some(mse) =err.downcast_ref::<MissingSemicolonError> () {
           insert_semicolon_in_source_code(mse.file(), mse.line() )?; // 如果分号缺失,则加入分号
           continue; // 返回 loop 的第一个语句-compile_project,再编译
         }
         return Err(err);
      }
   }
}
```

downcasting(向下转型)可以将 dyn Error 向下转型成一个实际的错误类型。例如:用户如果收到 std::io::Error,而且具体错误类型是 std::io::ErrorKind::WouldBlock,则进行一些特殊处理。对于其他错误类型,则略过而不作为。上面的程序例子特别处理的错误类型是 MissingSemicolonError。

如果用户收到一个 dyn Error,则用户可以尝试使用 Error::downcast_ref 来试图向下转型为具体的错误类型。downcast_ref 方法返回一个 Option,告诉用户向下转型是否成功。downcast_ref 只有在参数是'static 的情况下才能工作。如果我们返回一个不是'static 的 Error,就失去了对特征消除后的错误进行内部检查(introspection)的功能。

5.17 总结

Rust 错误处理有以下几种情况:

- 在做原型设计和一些快速验证的工作时可以用 unwrap 或 expect 忽略错误;
- 在不需要提前返回错误时可以链式调用各种组合子,一气呵成地处理错误;
- 需要提前返回错误,如果不是写库程序,则直接用 try!宏;
- 作为库的作者,应该自己定义错误类型并实现 From trait,然后借助 try!宏和各种组合子综合灵活处理;

- 依据个人偏好，选择使用 try!宏还是问号(?)语法糖；
- panic!宏代表一个程序无法处理的状态，并且停止执行，而不是使用无效或不正确的值继续处理；
- Result 代表操作可能会在一种可以恢复的情况下失败，可以使用 Result 来告诉代码调用者需要处理潜在的成功或失败。