Error Handling in Rust

Hitesh Kumar Sharma
Instructor, Pluralsight

PLURALSIGHT

Agenda

- Error in Rust
- Types of Error in Rust
- Panic Macro
- Recoverable Errors
- Unrecoverable Errors
- Unwrap()
- expect()

Error in Rust Programming

An error is an unexpected behavior or event that could cause a program to create unwanted output or to suddenly end. Everyone wants their program to function without errors. We can try to identify and examine the program's potential error-causing components. Once those components have been identified, we may specify how they ought to respond in the event of a mistake. Error handling is the process of identifying and specifying cases for a certain block of code. One thing to remember is that while we cannot totally eliminate errors, we can aim to limit them or at the very least lessen the impact they have on our program.

Types of Error in Rust Programming

Errors in Rust can be divided into ***recoverable*** and ***unrecoverable*** categories.

- **Recoverable Errors:** Errors that can be fixed prevent a program from ending abruptly. Attempting to get a file that is either not available or that we are not authorized to open is an example.

- **Unrecoverable Errors:** Programs that encounter unrecoverable errors end abruptly. Example: Attempting to access an array index that is larger than the array's size.

Rust utilizes a data type Result <R,T> to handle recoverable errors and a panic! macro to terminate the execution of the program in the case of unrecoverable errors. Most languages do not distinguish between the two errors and use an Exception class to overcome them.

Panic Macro

Using the panic! macro, a program can end right away and give the caller of the program feedback. When a program reaches an unrecoverable condition, it should be used.

Syntax

```
fn main() {
   panic!("Hello from PluralSight");
   println!(“Next Statement after Panic"); //unreachable statement
}
```

Output:

```
thread 'main' panicked at Hello from PluralSight ', main.rs:3
```

In this example, when the panic! macro is encountered, the program will end instantly.

Panic Macro (Example)

In this example, we have demonstrated that Panic can be invoked if any rule/logic is violated.

```
fn main() {
  let age = 17;
  //Checking age for driving license
  if age >=18 {
    println!(“Great! You are eligible for Driving License");
  } else {
    panic!(“Sorry! You are not eligible for Driving License ");
  }
  println!("End of main emthod");
}
```

The above coding example returns an error if we pass an age value less than 18 Years.

Recoverable Errors

Recoverable errors do not cause the program to terminate abruptly. Recoverable errors can be handled with Enum Result <T,E>.

```
enum Result<T,E> {
   OK(T),
   Err(E)
}
```

It has two sections :

- **`OK`**
- **`Err`**

`T` and **`E`** are two parameters. These are generic parameters.

- In **`OK`** section, T stands for the type of the value that will be returned in a successful scenario.
- In **`Err`** Section, E Stands for the type of the error that will be returned in a failure case.

Recoverable Errors (Example)
(Unhandled Exception)

In the following code Snippet, We have demonstrated that if a recoverable error is unhandled than what will be the behavior of the code.

The program returns *OK(File)* if the file already exists and *Err(Error)* if the file is not found.

```
use std::fs::File;
fn main() {
   let PS_file =
File::open(“PluralSight_File.txt");
   // PluralSight_File.txt file does not exist
   println!("{:?}", PS_file );
}
```

```
Output:

Err(Error { repr: Os {
code: 2, message: "No such
file or directory" } })
```

As per the output, we can see that Err(Error) section is called, and a default error message is displayed.

Recoverable Errors (Example)
(Handled Exception)

In the following example we have introduced match statement to handle an error that was returned while opening a file. It is used to handle the exception and avoid abnormal termination.

```
use std::fs::File;
fn main() {
   let PS_file = File::open("PluralSight_File.txt ");
match PS_file {
      Ok(f)=> {
         println!("Success! file found {:?}", PS_file );
      },
      Err(e)=> {
         println!("Sorry! file not found \n{:?}",e); }
   }
   println!("This is end of main");
}
```

```
Output:

Sorry! file not found Os {
code: 2, kind: NotFound,
message: "The system
cannot find the file
specified." }
This is end of main
```

The program prints end of the main event though file was not found. This means the program has handled error gracefully.

Matching on Different Errors

Depending on the cause of the failure, we may wish to perform different steps. For example, if File::open failed because the file was not found, we may want to create the file and return the handle to it. We still want the code to panic if File::open failed for any other reason, such as because we lacked authorization to open the file. We include an inner match phrase for this purpose.

```
use std::io::ErrorKind;
use std::fs::File;
fn main() {
    let PS_file = File::open(" PluralSight_File.txt ");

    let my_file = match PS_file {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("PluralSight_File.txt") {
                Ok(fc) => fc,
                Err(e) => panic!(“Sorry! Issue in creating file: {:?}", e),
            },
            other_error => {
                panic!(“Sorry! Issue in opening file: {:?}", other_error);
            }  },  };
}
```

Unrecoverable Errors

Unrecoverable errors are mistakes that, as the term implies, a programmer is unable to correct. The eventual outcome of any unrecoverable error is that the program terminates (quits). The first panic is the whole procedure! macro is executed, the error message and its location are printed, and then the program is ended. The majority of the time, it results from errors that the programmer left in the code.

Example for unrecoverable error in Rust:

```
// Rust program for unrecoverable error

fn main() {
let PluralSight=["html",“dbms",“cn", "css"];
// in this array index 7 does not exist
// So it will trigger panic! macro
println!("{}", PluralSight[7]);
}
```

In above example, we have defined a single `PluralSight` array with four strings in it. Now when we were trying to display 7 elements from the array, panic! macro will be triggered because the length is 4 [0-3] and we are seeking for the 5th index. The error message, location, and stack trace will then be printed.

Unwrap() Method

Rust has two methods, unwrap() and expect(), to make the process simpler if we wish to terminate the program once it encounters a recoverable error.
Following is an example of unwrap() method.

Example unwrap():

```
use std::fs::File;

fn main() {
let PS_file =   File::open(" PluralSight_File.txt ").unwrap();
}
```

If the file cannot be located, the unwrap() function triggers the panic! macro; otherwise, it returns the file handler instance. Although unwrap() shortens the program, it can be difficult to tell which unwrap() method is responsible for the panic! macro when there are too many unwrap() methods present in the program. Therefore, we require a system that can generate the personalized messages. The expect() method saves the day in that circumstance.

Expect() Method

The program can return a custom error message in case of a panic. This is shown in the following example.

Example Expect():

```
use std::fs::File;
fn main(){
   let file = File::open(" PluralSight_File.txt ").expect(“Sorry! File is not
   available");
   println!("end of main");
}
```

Unwrap() and expect() are related functions. The main distinction is that expect allows for the display of a unique error message.

End of Module