从C++到Rust，错误处理Result，第二集

原创 Ajonbin AJonbin的杂货铺 2024年01月11日 23:04 美国

上一集讲了Result的基本概念，以及怎么匹配和处理结果。

这集讲讲我们自己创建和返回一个Result。

这次我们和讲panic时的例子类似，创建一个divide函数，返回一个Result。这个Result中会包含两种不同的错误。

首先在上集hello_result的工程里创建个lib crate和新加一个bin crate，顺便复习下之前创建crate的内容。

先看看Cargo.toml

```
[package]
name = "hello_result"
version = "0.1.0"
edition = "2021"

[dependencies]

[lib]
name = "libdivide"
path = "src/lib/lib.rs"

[[bin]]
name = "my_result"
path = "src/my_result.rs"
```


我们增加了libdivide和my_result。强化下记忆，怎么增加lib和bin。

再贴上代码

```rust
pub fn divide(left: u32, right: u32) -> Result<f32, std::io::Error>{
  if right == 0 {
    Err(std::io::Error::new(std::io::ErrorKind::InvalidData, "Cant divide by ZERO"))
  }else if left == 0 {
    Err(std::io::Error::new(std::io::ErrorKind::Unsupported, "Sorry, Don't support ZERO"))
  }else{
    Ok(left as f32/right as f32)
  }
}
```


来详细讲解下

```rust
pub fn divide(left: u32, right: u32) -> Result<f32, std::io::Error>{
```

第1行，定义了divide函数，它有两个参数，left和right，都是无符号32位整型。返回值是*Result<f32, std::io::Error>*。

由于divide()是一个库函数，加上 *pub* 关键字，确保外部可以调用。

回一下上集的内容，*Result<T,E>*

这个T是f32， 32位浮点数，也就是成功情况下返回的实际类型；

这里E是*std::io::Error*，这是标准IO库中定义的类型。虽然除法返回IO的错误不太合适，这里我们为了简单先借用一下。也就是说，如果divide发生了错误，返回的就是一个标准IO库的Error。

```rust
  if right == 0 {
    Err(std::io::Error::new(std::io::ErrorKind::InvalidData, "Cant divide by ZERO"))
```

第2-3行，先检查下right的值，如果right等于0，那么这是一个除0错误。我们创建一个新的Error来表示这个错误。

创建一个新的*std::io::Error*需要两个参数，第一个参数是错误类型*std::io::ErrorKind*，这个我们选了InvalidData。第二个参数可以是任何类型（严格来说，第二个参数应该是实现了Debug和Display特性的类型... 现在不明白就先忽略），这里我们用字符串类型，提示下错误信息。

注意我们新建的std::io::Error被当作参数来创建一个Result::Err()。

Err() 是divide()函数的返回值，通过匹配Err()类型，我们可以得到真正的错误*std::io::Error*。

有点拗口，但是必须搞清楚区别。

```rust
  }else if left == 0 {
    Err(std::io::Error::new(std::io::ErrorKind::Unsupported, "Sorry, Don't support ZERO"))
```

第4-5行，是我硬凑了个错误，为了演示下不同的错误种类。当left是0时，我们也认为出错，创建一个新的Error，第一个参数ErrorKind选择Unsupported，第二个参数也是个字符串。

```
6   }else{
7     Ok(left as f32/right as f32)
8   }
```

第6-8行，这个分支返回成功的值，left除以right。由于left是无符号整型，通过 *as f32* 转换类型位32位浮点数。

同样算术结果被封装在Ok()中返回。在处理函数返回值时，通过匹配Ok()，就可以得到真正的f32结果。

好了，看完怎么定义Result之后，再看看怎么调用和匹配Result。

还是先上完整代码

```
1 use libdivide::divide;
2 fn main() {
3     println!("Hello, world!");
4     let ret = divide(10,4);
5     match ret {
6       Ok(r) => println!(" 10 / 4 = {}", r),
7       Err(e) => println!("{}", e),
8     }
9     let mut ret_e = divide(10,0);
10     match ret_e {
11       Ok(r) => println!(" 10 / 0 = {}", r),
12       Err(e) => match e.kind() {
13         std::io::ErrorKind::InvalidData => println!("We got expected erro -- {:?}", e),
14         std::io::ErrorKind::Unsupported => println!("We got expected erro -- {:?}", e),
15         other_error => println!("This error is no expcted -- {:?}", other_error),
16       }
17     }
18     ret_e = divide(0,10);
19     match ret_e {
20       Ok(r) => println!(" 0 / 10 = {}", r),
21       Err(e) => match e.kind() {
22         std::io::ErrorKind::Unsupported => println!("We got expected erro -- {:?}", e),
23         _ => (),
24       }
25     }
26 }
```



```
1 use libdivide::divide;
```

第1行，将libdivide中的divide通过关键字*use*包含进来。

```
4     let ret = divide(10,4);
5     match ret {
6       Ok(r) => println!(" 10 / 4 = {}", r),
7       Err(e) => println!("{}", e),
8     }
```


第4-8行，我们计算10除以4，由于left和right都不是0，那么就会得到正确的计算结果。

我们将divide()的结果赋值给变量ret，再通过*match*关键字对ret的类型进行匹配。

如果是Ok(r)，就可以通过变量r得到实际的除法结果；

如果是Err(e)，就可以通过变量e得到一个*std::io::Error*;

```
9     let mut ret_e = divide(10,0);
10     match ret_e {
11       Ok(r) => println!(" 10 / 0 = {}", r),
12       Err(e) => match e.kind() {
13         std::io::ErrorKind::InvalidData => println!("We got expected erro -- {:?}", e),
14         std::io::ErrorKind::Unsupported => println!("We got expected erro -- {:?}", e),
15         other_error => println!("This error is no expcted -- {:?}", other_error),
16       }
17     }
```


第9-17行，我们用10除以0，这是right=0的情况。

根据divide的实现，我们应该得到一个Err(std::io::Error)，并且错误种类是InvalidData。

可以看到这里我们用了两次匹配。

第一次用match匹配divide的结果。应该匹配到第12行Err(e)。

第二次用match匹配错误e的种类e.kind()。由于right=0是，ErrorKind是InvalidData，那么第12行的匹配结果应该是第13行 std::io::ErrorKind::InvalidData。

```
18     ret_e = divide(0,10);
19     match ret_e {
20       Ok(r) => println!(" 0 / 10 = {}", r),
21       Err(e) => match e.kind() {
22         std::io::ErrorKind::Unsupported => println!("We got expected erro -- {:?}", e),
23         _ => (),
24       }
25     }
26 }
```

第18-26行，我们用0除以10， 这是left=0的情况。这是返回的应该是ErrorKind是Unsupported的std::io::Error。

我们在22行匹配我们关心的错误种类。

注意第23行的写法，由于，我们不关心其他的值，可以用“_”作为一个变量。这是个特殊的变量，表明之后并不会用到这个变量，此变量会被忽略。如果匹配成功后，我们什么代码都不想执行，可以用“()”，表示一个空操作。

解释完代码，来看看实际运行情况。

注意由于我们有多个bin，所以运行时要加上--bin 来指定bin的名字。

```
hello_result$ cargo run --bin my_result
   Compiling hello_result v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in 0.26s
     Running `target/debug/my_result`
Hello, world!
 10 / 4 = 2.5
We got expected erro -- Custom { kind: InvalidData, error: "Cant divide by ZERO" }
We got expected erro -- Custom { kind: Unsupported, error: "Sorry, Don't support ZERO" }
```


完美。

请看下集。