

12 错误处理

Option<T>

Rust 中的关键知识讲解了一多半了，到现在我们都没有发现在其他语言中常见的 `null`。那么 Rust 是如何表示一个变量的值尚未初始化呢？抱歉，Rust 不允许这种情况。它提供了另外一种方案：`Option<T>`，这是一个带有范型的枚举类型。我们先来看一下它的定义：

```

1 pub enum Option<T> {
2     None,
3     Some(T),
4 }
```

它表示要么有你预期的、类型为 `T` 的值，要么就没有任何值。假如这样的一个功能，用户登录的时候，需要输入手机号和验证码，我们需要先去数据库中查询用户。但是，可能用户输入的手机号在数据库中存在，也可能是不存在，也就是说，可能会找到对应的用户，也可能找不到。

如果用 JavaScript 写，就类似下面的代码：

```

1 /**
2  * @params {string} phoneNumber 手机号
3  * @returns {User | null} 如果找到，则返回用户对象，否则，返回 null
4 */
5 function getUserByPhoneNumber(phoneNumber) {
6     //...
7 }
8
9 function login(phoneNumber, smsCode) {
10     const user = getUserByPhoneNumber(phoneNumber);
11     // 检查验证码是否匹配。
12 }
```

那么，调用者是否检测返回的 `User` 对象是否存在，那就完全在于调用者了，定义 `getUserByPhoneNumber` 函数的开发者没有办法强制。

那么 TypeScript 会好一些，因为可以在函数声明中表明可能会找不到这个用户，类似下面的代码：

```

1  function getUserByPhoneNumber(phoneNumber: string): User | null
2  {
3      return null;
4  }
5
6  function login(phoneNumber, smsCode): boolean {
7      const user = getUserByPhoneNumber(phoneNumber);
8      if (!user) {
9          return false;
10     }
11 }
```

就需要调用 `getUserByPhoneNumber` 的同学依检查返回值是否为 `null`，避免运行时的错误。

Rust 通过使用 `Option<T>` 告诉使用者，这里的值可能有，也可能没有。无论是否存在，都是程序正常运行可能发生的情况，这个不算是错误，但是需要使用者检测值的存在性。

`Option<T>` 提供了很多方便我们使用的方法：

- `is_none()` 检查 `Option<T>` 是否为 `None`
- `is_some()` 检查 `Option<T>` 是否为 `Some(T)`。`is_none()` 和 `is_some()` 方法其实很少用，通常都是使用 `match` 表达式来检测或者提取内部的值
- `unwrap()` 返回 `T` 的值。如果为 `None` 则会发生异常
- `unwrap_or()` `Option<T>` 为 `Some(T)` 则返回包裹的值，否则返回调用函数时候的默认值，例如：`Some("car").unwrap_or("bike")`，
`None.unwrap_or("bike")`

其他语言，例如 Java 或 JavaScript 中的 `null` 本身并没有错，但是我们在使用过程中往往会忽视对于值的存在性的检查，所以导致了大量的空指针异常。在 Rust 中，强制编写代码的时候进行存在性检查，能够大大减少运行时的问题。

Result<T, E>

`Option<T>` 是用来检查值的存在性的，而 `Result<T, E>` 则是用来检查代码处理的正确性的。

我们知道，几乎任何代码都存在出错的可能，只是概率大小问题罢了。所以，无论是哪种语言，都有一些方案来表示这个函数的处理出现了错误，导致无法正常返回期望的值了。比如在 JavaScript 中，如果代码执行过程中出现错误，则向外抛出 `Error`，或者返回一个特殊的值（例如 `parseInt` 可能返回 `Nan` 表示输入的字符串不是一个有效的整数）；在 Java 中则是抛出 `Exception`；C 语言中，通常返回非零值或者 `NULL` 等特殊的值；PHP 中通常返回 `FALSE`。

下面是 `Result<T, E>` 的定义：

```

1 enum Result<T, E> {
2     Ok(T),
3     Err(E),
4 }
```

如果代码正确执行了，返回的是 `Result::Ok(T)` 包裹的期望值，否则，返回一个 `Result::Err(E)` 包裹的错误。那么调用者就不得不对错误进行处理。假如我们把软件开发的过程分为：需求、设计、开发、测试、上线运行这么几个阶段，有一个常识性的结论就是：问题发现的越早，解决问题所需的成本就越低（实在解决不了的问题就绕过去，比如在需求阶段，离谱的需求先放放 ^_^）。Rust 把发现问题的时机提前到开发阶段，而且带有一定的强制性，如果你不检查返回值就盲目的使用，那么对不起，无法通过编译。

类似 `Option<T>`，在 `Result<T, E>` 枚举中也有一些方便使用的方法：

- `is_ok()` 检查是否成功
- `is_err()` 检查是否发生了错误

- `unwrap()` 尝试获取期望的值，如果此时的结果是错误的，则会发生 panic
- `unwrap_or()` 尝试获取期望的值，如果此时的结果是错误的，则使用传入参数替代

错误处理

程序中的错误可以分为两种：`Result<T, E>` 或者是 `Panic`。前者代表允许发生的错误，可能发生的错误，即使是发生错误，程序还可以继续执行，只不过是和能够正确处理走了不同的分支。后者表示程序就真的无法继续了。例如，一个命令行工具，当输入的参数不是期望的参数的时候，可以通过 `panic!` 宏来中止执行，回想一下我们在猜数字小游戏中，如果用户输入的不是一个数字，我们就通过 `except()` 方法中止程序执行。再比如使用 Rust 开发的后台 API 应用，如果启动的时候，由于配置错误导致无法建立和数据库之间的连接，那么就可以中止启动。

实际项目中，我们使用的大部分 crate，调用函数的返回值通常是 `Result<T, E>`，因为以库的方式提供给其他应用使用的代码，必须要考虑到可能出现错误的情况。例如下面的代码：

```

1 fn main() {
2     let s = match foo() {
3         Ok(s) => s,
4         Err(e) => {
5             println!("{}: {:?}", e);
6             e.to_string()
7         }
8     };
9 }
10
11 fn foo() -> Result<String, std::fmt::Error> {
12     match bar() {
13         Ok(s) => Ok(s),
14         Err(e) => {
15             println!("{}: {:?}", e);
16             let fmt_e = std::fmt::Error;
17             Err(fmt_e)
18         }
19     }
20 }
```

```

19     }
20 }
21
22 fn bar() -> Result<String, std::io::Error> {
23     // 读取文件之类的操作，可能会引发 IO 异常的
24 }
```

很显然，每一次调用返回值为 `Result<T, E>` 的函数或者方法都需要使用 `match` 表达式来判断返回值是否正确，会导致代码啰嗦臃肿。那么有没有好办法来解决这个问题呢？我们可以使用 `? 来实现错误向上传播，和 JavaScript 非常类似，不需要每一层调用都使用 try ... catch，只要最外层调用者能够正确处理异常，不要让程序崩溃就可以了。`

上面的解释有点儿抽象，那么我们来看一个真实的例子：我们使用 Redis 实现了一个简单的任务队列，用户发起任务之后，后台将数据导出。操作步骤如下：

1. 从 Redis 读取任务编号
2. 从 MongoDB 获取任务的详细信息
3. 从 MySQL 查询数据
4. 将数据写出到 Excel 文件
5. 将文件上传到阿里云 OSS 存储
6. 修改任务状态，将最终存储的 Bucket 和 Object Key 更新到 MongoDB 的任务详情中

在这个处理过程中，我们可能会遇到的错误有：

- `redis::RedisError`
- `rust_xlsxwriter::XlsxError`
- `mongodb::error::Error`
- `sqlx::Error`
- `aliyun_oss_rust_sdk::error::OssError`

根据这些错误的名字，可以猜出来对应的 crate。首先，我们定义一个这个任务应用的错误枚举，`AppError`，然后借助 `thiserror` 这个 crate 的宏，使得我们的枚举可以包含不同类型的错误：

```

1 #[derive(Error, Debug)]
2 pub enum AppError {
3     #[error("MongoDB operation failed")]
4     MongoError(#[from] mongodb::error::Error),
5
6     #[error("Database operation failed")]
7     SqlxError(#[from] sqlx::Error),
8
9     #[error("Excel error occurred")]
10    XlsxError(#[from] rust_xlsxwriter::XlsxError),
11
12    #[error("OSS error")]
13    OssError(#[from] aliyun_oss_rust_sdk::error::OssError),
14
15    #[error("Redis error")]
16    RedisError(#[from] redis::RedisError),
17
18    #[error("{0}")]
19    MyError(String),
20 }
```

在实际开发中，我们不会把每个方法的返回值都写全为 `Result<T, E>`，这样也会显得啰嗦，所以我们使用 `type` 定义一个类型别名：

```
1 type AppResult<T> = Result<T, AppError>;
```

那么处理任务队列的代码结构类似下面（注，下面的代码中的函数或者方法名称仅供示意）：

```

1 fn get_task_document(oid: &str) -> AppResult<TaskDocument> {
2     // 查询 MongoDB 数据库，可能会抛出 mongodb::error::Error
3     Ok(mongo_client.query_task_document(oid)?)
```

```
4     }
5
6     fn query_rows() -> AppResult<Vec<Row>> {
7         // 查询 MySQL 数据库, 可能会抛出 sqlx::Error
8         Ok(do_sql_query()?)
9     }
10
11    fn output_to_excel(rows: Vec<row>) -> AppResult<()> {
12        // 输出 Excel 文件可能会抛出 rust_xlsxwriter::XlsxError
13        Ok(write_to_excel()?)
14    }
15
16    fn upload_to_oss() -> AppResult<()> {
17        // 把 Excel 文件上传到 oss 可能会抛出
18        aliyun_oss_rust_sdk::error::OssError
19        Ok(upload_file_to_oss()?)
20    }
21
22    fn handle_task(id: &str) -> AppResult<()> {
23        let doc = get_task_document(id)?;
24        let rows = query_rows()?;
25        let _ = output_to_excel(rows)?;
26        let _ = upload_to_oss()?;
27        Ok(())
28    }
29
30    fn main() {
31        loop {
32            let task_id = match redis_client.blpop(key) {
33                Ok(v) => v,
34                Err(e) => {
35                    println!("read value from redis failed");
36                    break;
37                }
38            };
39            match handle_task(task_id) {
40                Ok(_) => {},
41                Err(e) => {
42
```

```
42             println!("handle task {} failed: {:?}", task_id,
43             e);
44         }
45     }
46 }
```

这样以来，代码的机构就比较直观了，不需要每次调用都使用 `match` 表达式处理异常情况，只在最外层处理掉异常情况就可以了。