错误处理

> Rust 的 `Debug` 与 `Display` 的分离设计，实际上是为了**清晰区分"开发者可读信息"和"用户可读信息"**。

- 不可恢复错误：`panic!(message)` or 运行期错误，如索引越界
- 可恢复：利用 Result 枚举表达可能成功或失败场景

  > Result#unwrap() or Result#expect()

- 错误传播：`?`

```rust
fn read_file(filename: &str) -> Result<String, io::Error>
    let file: File = File::open(path: filename)?;
}
```

- 错误类型不匹配时候

  > - 使用trait object 解决返回类型不匹配问题，适用于不关心返回的具体错误类型，类似java中直接使用 Exception 而非具体类型
  > - 若关心具体的错误类型，则可以自定义错误枚举，然后将原始错误包装或转换为自定义错误以达到统一错误类型目的

```rust
fn parse_file(filename: &str) -> Result<i32, io::Error> {
    let s: String = fs::read_to_string(path: filename)?;
    let i: i32 = s.parse()?;    `?` couldn't convert the error to `std::io::Error` the
    Ok(i)
}
```

```rust
fn parse_file(filename: &str) -> Result<i32, Box<dyn error::Error>> {
    let s: String = fs::read_to_string(path: filename)?;
    let i: i32 = s.parse()?;
    Ok(i)
}
```

```rust
0 implementations
enum ParseFileError {
    File,
    Parse(ParseIntError)
}

fn parse_file(filename: &str) -> Result<i32, ParseFileError> {
    let s: String = fs::read_to_string(path: filename): Result<String, Error>
                    .map_err(op: |e: Error| ParseFileError::File)?;
    let i: i32 = s.parse(): Result<i32, ParseIntError>
                    .map_err(op: |e: ParseIntError| ParseFileError::Parse(e))?;
    Ok(i)
}
```

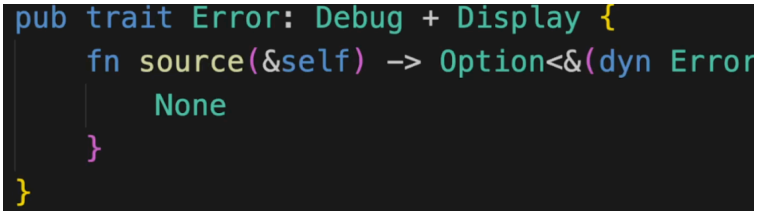
自定义错误枚举，让方法返回自定错误枚举类型，
然后针对方法中不同的错误，将其映射或包装成自定义错误

```rust
fn main() {
    let i: Result<i32, ParseFileError> = parse_file(filename: "example.txt");
    match i {
        Ok(i: i32) => println!("{i}"),
        Err(e: ParseFileError) => {
            match e {
                ParseFileError::File => {
                    // ...
                },
                ParseFileError::Parse(e: ParseIntError) => {
                    // ...
                }
            }
        }
    }
```

如果i是错误变量，那么我们匹配内部错误，这样我们就可以处理文件错误和解析错误。

- Rust中惯用错误：Idiomatic Errors

> 错误应该实现 error trait



注意: 该trait中有 Debug 描述了应该如何报告错误给开发者让其处理，而Display描述了错误应该如何被显示给users（User Facing）

- 自定义错误可以是 struct 或 enum 或 mix二者