Recoverable Errors

- 很多情况下，我们希望错误发生的时候不是直接panic，而是能从中recover。比如用户想打开一个不存在的文件的时候，我们希望能提醒用户，并要求用户再输入一遍文件名。这依赖于`Result`这个enum：

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

- 看下面一个简单的例子：

```
fn main() {
    let f = File::open("hello.txt");

    let f = match f{
        Ok(file) => file,
        Err(error) => panic!("Problem opening the file {:?}", error),
    }
}
```

  - 这样便要求我们在打开文件失败或者成功的时候都能有所处理
- 但如果我们想更进一步去处理错误呢？比如针对不同的错误原因采取不同的应对方法。我们可以再嵌套一个match：

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt");

    let f = match f{
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt"){
                Ok(fc) => fc,
                Err(e) => panic!("Problem creating the file {:?}", e),
            },
            other_error => {
                panic!("Problem open the file {:?}", other_error)
            },
        }
    };
}
```

  - 比如，当问题是文件不存在的时候，我们可以新建一个文件。
  - 这里我们用到了`io::ErrorKind`. 它定义了我们再io过程中所有可能遇见的错误。
- 显然，上面连续嵌套了三个match的方法比较繁琐，我们可以用`closure`,在13章也会讲到。

```
let f = File::open("hello_b.txt").unwrap_or_else(|error| {
    if error.kind() == ErrorKind::NotFound {
        File::create("hello_b.txt").unwrap_or_else(|error| {
            panic!("Problem creating the file {:?}", error);
        })
    }else {
        panic!("Problem opening the file {:?}", error);
    }
});
```

- 我们也可以用`unwrap()`这个函数来简化：

```
let f = File::open("hello.txt").unwrap();
```

  - `unwrap()`这个函数的功能是，在返回Ok的时候，它会返回Ok中包裹的值。如果返回时Err，它会panic
- 另外，我们也可以用`expect()`。它的功能和`unwarp()`相似，区别在于panic的时候会返回expect中的信息：

```
let f = File::open("hello.txt").expect("Failed to open hello.txt");
```

? Operation

- 我们可以用`?`来简化很多的事情，先看下面一个打开文件并读取文件内容的例子：

```
fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();

    File::open("hello.txt")?.read_to_string(&mut s)?;

    Ok(s)
}
```

  - `?`的作用是：如果返回是Ok，则将Ok中包裹的值返回。如果是Err，则直接把Errreturn。
  - `read_to_string(&mut s)`是将文件内容读取到到s中
- 当然，由于打开文件并读取到string中是个很常见的操作，Rust提供了`fs::read_to_string()`函数将问题进一步简化了：

```
use std::fs;
use std::io;

fn short_cut() -> Result<String, io::Error> {
    fs::read_to_string("hello.txt")
}
```

? with Option

- `?`除了可以用在Result上，还可以用在Option上，非常常用。比如下面的例子：

```
fn withOPtion(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
```

    - 如果中间next（）返回的是None，那就提前结束，并将None给return了。