11 Writing Automated Test

- Rust功能强大的一点在于：它对coding有严格的限制，并且会进行很多的type checking来尽量保证程序的正确性。但Rust并不能完全做到保证程序是百分百正确的。比如，我想写一个程序将每个传入的参数加2。Rust并不能保证你的程序是加2，而不是加100，200之类的。这种情况就需要用户写tests来确保自己的程序发挥了自己意想中的功能。

How to write tests

一个用于测试的`fn`需要annoted with `test` attribute. 一般来说，是add `#[test]` on the line before `fn`.

想要创建一个test，命令是：`cargo new new_test --lib`

- 另外，test中一个非常常用的macro就是`assert!`, 这个macro是evaluate一个boolean，如果value是true，就do nothing。如果value是false，就触发`panic!` to cause the test to fail.
- 除了`assert!`这个`macro`, 还有`assert_eq!`和`assert_ne!`. 它们可以理解为先对传入的参数进行eq/ne判断，然后就判断的结果进行`assert!`.

Adding Custom Failure Messages

当我们在debug的时候，光是知道有bug其实是不够的，我们最好能知道是哪里出现了什么样的bug。这就需要我们在assert的时候，最好能输出一些信息。

比如我们想测试下面这个`greeting`函数是否正确地带上了人名：

```
pub fn greeting(name: &str) -> String {
    format!("Hello!")
}
```

我们可以这样写测试程序，以让其在failure的时候输出一些信息：

```
    #[test]
    fn test_greeting_function() {
        let name = "David";
        let result = greeting(&name);
        assert!(
            result.contains(&name),
            "Greeting was not containing name, value was {}",
            result
        );
    }
```

Checking for Panics with should_panic

我们可以通过`#[should_panic]`这个修饰，来测试我们程序是否正确地在我们想要的位置（比如error的位置）panic。比如下面这个例子：

我们想要测试，`new()`这个函数，能否在0以下和100以上进行报错。我们便可以用`#[should_panic]`这个修饰告诉这个test，这个函数是应该panic的，如果没有panic则测试不通过。

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess { value }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

另外，我们还可以在`#[should_panic(expected = "...")]`中，指定我们应该接收到的panic的信息。

```
    #[test]
    #[should_panic(expected = "is larger than 100")]
    fn test_init_guess() {
        Guess::new(200);
    }
```

Controlling How Tests Run

- `cargo test`是将所有的tests run in parallel （用很多个threads）。如果你不想这么做，可以用`cargo test -- --test-threads=1`来指定用来跑tests的线程数。
- 另外我们也可以指定出我们想要运行的test：`cargo run $function_name`. 所有名字中带了$function_name的tests都会被运行。
- 另外，我们也可以在test前面加`#[ignore]`这个修饰，这样的话，当我们运行`cargo test -- --ignored`的时候，有这样修饰的tests就会被忽略。

Test Organization

- 在rust中一般把测试分成Unit Test和Intergration Test.
- 写**Unit Test**一般是专门创建一个test module，然后用`#[cfg(test)]`, 来告诉compiler这是一个test，并且只在运行`cargo test`的时候进行编译。
- 而**Integration Test**则是专门创建一个file出来，然后调用在lib中的pub 函数。比如我们在原有的文件结构里新建一个tests文件夹，然后就可以加入任意数量的integration test了。文件结构如下：

```
.
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    ├── integration_tests1.rs
    └── integration_tests2.rs
```

  - 接着，在每个integration tests 文件中，我们只要标注上`#[test]`, 就能告知compiler这是一个integration test，并在`cargo run`的时候将其编译并测试：

```
// integration_test1.rs
use How_To_Write_Test;

#[test]
fn it_adds_two() {
    assert_eq!(4, How_To_Write_Test::add_two(2));
}
```

  - 如果只想运行某一个integration test中的某一个文件，可以直接`cargo test --test integration_test1`