한 번에 끝내는
블록체인 개발 A to Z

Chapter 2

Rust Advanced

Chapter 2

Rust Advanced

Writing Tests

How to Write Tests

The Anatomy

At its simplest, a test in Rust is a function that's annotated with the test attribute. Attributes are metadata about pieces of Rust code. To change a function into a test function, add #[test] on the line before fn. When you run your tests with the cargo test command, Rust builds a test runner binary that runs the annotated functions and reports on whether each test function passes or fails.

```
$ cargo new adder --lib
     Created library `adder` project
$ cd adder
```

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

Result

```rust
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }

    #[test]
    fn another() {
        panic!("Make this test fail");
    }
}
```

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.72s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test tests::another ... FAILED
test tests::exploration ... ok

failures:

---- tests::another stdout ----
thread 'main' panicked at 'Make this test fail', src/lib.rs:10:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::another

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;

error: test failed, to rerun pass '--lib'
```

assert! Macro

The **assert!** macro, provided by the standard library, is useful when you want to ensure that some condition in a test evaluates to **true**. We give the **assert!** macro an argument that evaluates to a **Boolean**. If the value is **true**, nothing happens and the test passes. If the value is **false**, the **assert!** macro calls panic! to cause the test to fail.

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

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

    #[test]
    fn larger_can_hold_smaller() {
        let larger = Rectangle {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 1,
        };

        assert!(larger.can_hold(&smaller));
    }
}
```

assert! Macro (2)

The **assert!** macro, provided by the standard library, is useful when you want to ensure that some condition in a test evaluates to **true**. We give the **assert!** macro an argument that evaluates to a **Boolean**. If the value is **true**, nothing happens and the test passes. If the value is **false**, the **assert!** macro calls panic! to cause the test to fail.

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

    #[test]
    fn larger_can_hold_smaller() {
        // --snip--
    }

    #[test]
    fn smaller_cannot_hold_larger() {
        let larger = Rectangle {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 1,
        };

        assert!(!smaller.can_hold(&larger));
    }
}
```

assert_eq! assert_ne!

These macros compare two arguments for equality or inequality, respectively. They'll also print the two values if the assertion fails, which makes it easier to see why the test failed; conversely, the assert! macro only indicates that it got a false value for the == expression, without printing the values that led to the false value.

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

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

Custom Failure Message

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

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

    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(result.contains("Carol"));
    }
}
```

```rust
#[test]
fn greeting_contains_name() {
    let result = greeting("Carol");
    assert!(
        result.contains("Carol"),
        "Greeting did not contain name, value was `{}`",
        result
    );
}
```

Check Panics with should_panic

```rust
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);
    }
}
```

```rust
// --snip--
impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!(
                "Guess value must be greater than or equal to 1, got {}.",
                value
            );
        } else if value > 100 {
            panic!(
                "Guess value must be less than or equal to 100, got {}.",
                value
            );
        }

        Guess { value }
    }
}

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

    #[test]
    #[should_panic(expected = "Guess value must be less than or equal to 100")]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Controlling How Tests Are Run

Parallel Tests

When you run multiple tests, by default they run in parallel using threads, meaning they finish running faster and you get feedback quicker. Because the tests are running at the same time, you must make sure your tests don't depend on each other or on any shared state, including a shared environment, such as the current working directory or environment variables.

If you don't want to run the tests in parallel or if you want more fine-grained control over the number of threads used, you can send the --test-threads flag and the number of threads you want to use to the test binary.

```
$ cargo test -- --test-threads=1
```

Function Output

By default, if a test passes, Rust's test library captures anything printed to standard output. For example, if we call println! in a test and the test passes, we won't see the println! output in the terminal; we'll see only the line that indicates the test passed. If a test fails, we'll see whatever was printed to standard output with the rest of the failure message.

```
fn prints_and_returns_10(a: i32) -> i32 {
    println!("I got the value {}", a);
    10
}

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

    #[test]
    fn this_test_will_pass() {
        let value = prints_and_returns_10(4);
        assert_eq!(10, value);
    }

    #[test]
    fn this_test_will_fail() {
        let value = prints_and_returns_10(8);
        assert_eq!(5, value);
    }
}
```

Function Output (2)

If we want to see printed values for passing tests as well, we can tell Rust to also show the output of successful tests at the end with --show-output.

```
$ cargo test -- --show-output
   Compiling silly-function v0.1.0 (file:///p
    Finished test [unoptimized + debuginfo] t
     Running unittests (target/debug/deps/sil

running 2 tests
test tests::this_test_will_fail ... FAILED
test tests::this_test_will_pass ... ok

successes:

---- tests::this_test_will_pass stdout ----
I got the value 4


successes:
    tests::this_test_will_pass

failures:

---- tests::this_test_will_fail stdout ----
I got the value 8
thread 'main' panicked at 'assertion failed:
  left: `5`,
 right: `10`', src/lib.rs:19:9
note: run with `RUST_BACKTRACE=1` environment


failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ig

error: test failed, to rerun pass '--lib'
```

Subset of Tests

Sometimes, running a full test suite can take a long time. If you're working on code in a particular area, you might want to run only the tests pertaining to that code. You can choose which tests to run by passing cargo test the name or names of the test(s) you want to run as an argument.

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

    #[test]
    fn add_two_and_two() {
        assert_eq!(4, add_two(2));
    }

    #[test]
    fn add_three_and_two() {
        assert_eq!(5, add_two(3));
    }

    #[test]
    fn one_hundred() {
        assert_eq!(102, add_two(100));
    }
}
```

```
$ cargo test one_hundred
   Compiling adder v0.1.0 (file:/
    Finished test [unoptimized +
     Running unittests (target/de

running 1 test
test tests::one_hundred ... ok

test result: ok. 1 passed; 0 fai
```

```
$ cargo test add
   Compiling adder v0.1.0 (file:///projec
    Finished test [unoptimized + debuginf
     Running unittests (target/debug/deps

running 2 tests
test tests::add_three_and_two ... ok
test tests::add_two_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ig
```

Ignore Some Tests

Sometimes a few specific tests can be very time-consuming to execute, so you might want to exclude them during most runs of cargo test. Rather than listing as arguments all tests you do want to run, you can instead annotate the time-consuming tests using the **ignore** attribute to exclude them.

When you're at a point where it makes sense to check the results of the ignored tests and you have time to wait for the results, you can run **cargo test -- --ignored** instead.

If you want to run all tests whether they're ignored or not, you can run **cargo test -- --include-ignored**

```
#[test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}

#[test]
#[ignore]
fn expensive_test() {
    // code that takes an hour to run
}
```

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/add
    Finished test [unoptimized + debuginfo] tar
     Running unittests (target/debug/deps/adder

running 2 tests
test expensive_test ... ignored
test it_works ... ok

test result: ok. 1 passed; 0 failed; 1 ignored;
```

```
$ cargo test -- --ignored
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out;
```

Test Organization

Test Organization

Unit tests are small and more focused, testing one module in isolation at a time, and can test private interfaces. **Integration tests** are entirely external to your library and use your code in the same way any other external code would, using only the public interface and potentially exercising multiple modules per test.

Unit Tests

The purpose of unit tests is to test each unit of code in isolation from the rest of the code to quickly pinpoint where code is and isn't working as expected. You'll put unit tests in the src directory in each file with the code that they're testing. The convention is to create a module named **tests** in **each file** to contain the test functions and to annotate the module with **cfg(test)**.

The **#[cfg(test)]** annotation on the tests module tells Rust to compile and run the test code only when you run **cargo test**, not when you run **cargo build**.

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

```
pub fn add_two(a: i32) -> i32 {
    internal_adder(a, 2)
}

fn internal_adder(a: i32, b: i32) -> i32 {
    a + b
}

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

    #[test]
    fn internal() {
        assert_eq!(4, internal_adder(2, 2));
    }
}
```

Integration Tests

In Rust, **integration** tests are entirely external to your library. They use your library in the same way any other code would, which means they can only call functions that are part of your library's public API. Their purpose is to test whether many parts of your library work together correctly.

We create a **tests** directory at the top level of our project directory, next to src. Cargo knows to look for integration test files in this directory. We can then make as many test files as we want, and Cargo will compile each of the files as an individual crate.

Filename: tests/integration_test.rs

```
use adder;

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

```
$ cargo test
   Compiling adder v0.1.0 (file://
    Finished test [unoptimized + d
     Running unittests (target/deb

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 faile

     Running tests/integration_tes

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 faile

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 faile
```

```
$ cargo test --test integration_test
   Compiling adder v0.1.0 (file:///projects/a
    Finished test [unoptimized + debuginfo] t
     Running tests/integration_test.rs (targe

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignore
```

Submodule in Integration Tests

Each file in the tests directory is compiled as its own **separate crate**, which is useful for creating separate scopes to more closely imitate the way end users will be using your crate.

To avoid having common appear in the test output, instead of creating **tests/common.rs**, we'll create **tests/common/mod.rs**.

Filename: tests/common.rs

```
pub fn setup() {
    // setup code specific to your library's tests would go here
}
```

Filename: tests/integration_test.rs

```
use adder;

mod common;

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

Integration Tests for Binary Crates

If our project is a binary crate that only contains a **src/main.rs** file and doesn't have a **src/lib.rs** file, we can't create integration tests in the tests directory and bring functions defined in the **src/main.rs** file into scope with a use statement. Only library crates expose functions that other crates can use; binary crates are meant to be run on their own.