Escrevendo testes automatizados

Diogo Silveira Mendonça

Slides Baseados no Capítulo 11 do Rust Book
https://doc.rust-lang.org/book/ch11-00-testing.html

Introdução

- Neste capítulo, veremos como:
  - Criar testes;
  - Fornecer mensagens personalizadas para testes que falharam;
  - Controlar como os testes serão executados;
  - Organizar os testes em testes unitários e de integração;

- Os exemplos podem ser encontrados nesse link.

Como escrever testes

- Testes são funções do Rust que verificam a funcionalidade do código.

- O corpo desses testes normalmente realizam 3 ações:
  - Configura todos os dados e estados necessários.
  - Roda o código que deseja testar.
  - Verifica se os resultados são equivalentes ao esperado.

- Vamos ver os recursos que o Rust utiliza para realizar essas ações.
  - O atributo “test”
  - Alguns macros
  - O atributo “should_panic”

A anatomia de uma função de teste

- Um teste em Rust é uma função que é anotada com o atributo de teste.
- Atributos são metadados sobre pedaços de código do Rust.
- Sempre quando é criado um novo projeto de biblioteca com Cargo, também é criado um módulo de teste com um template de uma função de teste.
- Vamos criar uma biblioteca chamada addr, que irá adicionar dois números.

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

A anatomia de uma função de teste

- Ao criar, teremos o arquivo src/lib.rs, com um código similar ao do lado.
- A anotação #[test]: este atributo indica que esta é uma função de teste.
- it_works é o nome do teste.
- O macro assert_eq! é utilizado para verificar se a variável result é igual a 4

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

A anatomia de uma função de teste

- Utilizando o comando “cargo test”, vai compilar e executar os testes.

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
       Finished test [unoptimized + debuginfo] target(s) in 0.57s
       Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

A anatomia de uma função de teste

- A mensagem anterior informa quantos testes passaram, falharam, ignorados, filtrados e medidos.
  - Os testes medidos são para o benchmark, disponível apenas para a versão nightly do Rust.

- Doc-tests adder é um teste para a documentação, onde o Rust pode compilar qualquer exemplo de código que aparece na documentação da nossa API.

- Isso ajuda a manter a documentação do código atualizada.

A anatomia de uma função de teste

- Mudamos o nome do teste it_works para exploration.
- Adicionamos o teste another, que irá falhar chamando o macro panic!.
- Na mensagem de erro, podemos ver que um teste falhou e onde esse teste entrou em pânico.

Exemplo:

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

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

A anatomia de uma função de teste

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
       Finished test [unoptimized + debuginfo] target(s) in 0.72s
       Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)
running 2 tests
test tests::another ... FAILED
test tests::exploration ... ok

failures:
---- tests::another stdout ----
thread 'tests::another' 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; finished
in 0.00s

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

Verificando os resultados com o marco assert!

- O macro assert! é útil quando queremos garantir que uma condição para um teste seja verdadeira.
- Damos ao macro assert! um argumento que verifica um Booleano.
- Se o valor for verdadeiro, nada acontece e o teste é aprovado.
- Se o valor for falso, a macro assert! chama panic!.
- No capítulo 5, usamos uma struct Rectangle e um método can_hold, que retorna um booleano.

Exemplo:

```
#[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
        }
}
```

Verificando os resultados com o marco assert!

- O código ao lado, instância dois Retângulos, um maior 8x7 e um menor 5x1.
- A linha “use super::*;” utiliza um glob, permitindo utilizar coisa que foi definido no módulo externo no módulo de testes.
- Assert! é utilizado para verificar o resultado de larger.can_hold(&smaller)
- Rodando os teste, ele deverá passar.

Exemplo:

```
#[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));
    }
}
```

Verificando os resultados com o marco assert!

- Adicionamos um outro testes, que espera um resultado falso da função can_hold.
- Para isso, utilizamos o símbolo de negação “!”, antes de passar para assert!.
- Tornando o resultado falso em positivo para a verificação de assert!.
- Com isso os dois testes deverão passar.

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

Verificando os resultados com o marco assert!

- Agora vamos introduzir um bug, modificando a função can_hold.
- Mudamos o sinal de maior por um de menor, na verificação da largura.

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

```
running 2 tests
test tests::larger_can_hold_smaller ... FAILED
test tests::smaller_cannot_hold_larger ... ok

failures:
---- tests::larger_can_hold_smaller stdout ----
thread 'tests::larger_can_hold_smaller' panicked at 'assertion failed:
larger.can_hold(&smaller)', src/lib.rs:28:9
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

Testando a Igualdade com as Macros assert_eq! e assert_ne!

- Uma maneira comum de testar é verificar se um valor resultante do código é igual ao valor esperado.

- Você pode fazer isso usando a macro assert! e passando a ela uma expressão usando o operador “==”.

- No entanto, este é um teste tão comum que a biblioteca padrão fornece os macros:
    - assert_eq! que verifica se um valor é igual ao esperado.
    - assert_ne! que verifica se o valor é diferente do esperado

- Essas macros comparam dois argumentos quanto à igualdade ou desigualdade, respectivamente.

- Diferentemente do macro assert!, os macros assert_eq! e assert_ne! imprimem os dois valores, quando o teste falha. Auxiliando no entendimento da falha do teste.

Testando a Igualdade com as Macros assert_eq! e assert_ne!

- A função add_two, está recebendo um número e adicionando 2.
- assert_eq! verifica se o valor de add_two(2) será igual a 4.
- O teste deve passar.

código de src/lib.rs:

```
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));
    }
}
```

Testando a Igualdade com as Macros assert_eq! e assert_ne!

- Agora vamos introduzir um bug. A função, em vez de somar 2 estará agora somando 3.

```
pub fn add_two(a: i32) -> i32 {
    a + 3
}
```

```
running 1 test
test tests::it_adds_two ... FAILED

failures:

---- tests::it_adds_two stdout ----
thread 'tests::it_adds_two' panicked at 'assertion failed: `(left == right)`
  left: `4`,
 right: `5`', src/lib.rs:11:9
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

Testando a Igualdade com as Macros assert_eq! e assert_ne!

- Na mensagem de erro podemos ver que assert_eq! utiliza os valores (left == right).

- Por conta disso, a ordem dos valores não importa, então poderíamos escrever:

  assert_eq!(add_two(2), 4).

- A macro assert_ne! funciona de forma similar a assert_eq!, porém ela espera que os valores sejam diferentes (left != right).

Testando a Igualdade com as Macros assert_eq! e assert_ne!

- Os valores sendo comparados devem implementar os traços PartialEq e Debug.
- Todos os tipos primitivos e a maioria dos tipos da biblioteca padrão implementam esses traços.
- Para structs e enums que você define, será necessário implementar PartialEq para afirmar a igualdade desses tipos.
- Você também precisará implementar Debug para imprimir os valores quando a assertiva falhar.
- Como ambos os traços são traços deriváveis, isso geralmente é tão simples quanto adicionar a anotação #[derive(PartialEq, Debug)] à definição de sua struct ou enum.

Adicionando mensagens de falha personalizadas.

- Podemos adicionar argumentos opcionais para os macros assert!, assert_eq! e assert_ne! exibir mensagens de erro personalizadas.

- Qualquer argumentos especificados após os argumentos obrigatórios são repassados para o macro format!. Então você pode passar uma forma de string, que contenha um espaço reservado {} e valor para preencher esse espaço reservado.

Adicionando mensagens de falha personalizadas.

- O código ao lado, já tem um erro introduzido. Já que greeting("Carol") retornará somente “Hello” e o assert! espera que a string contenha “Carol”.
- Rodando o teste, poderemos perceber que foi printado a mensagem “panicked at 'Greeting did not contain name, value was `Hello!`'”

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

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

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

Verificando por Panic com should_panic

- Considere o tipo Guess que criamos no Capítulo 9.
- Podemos escrever um teste que garanta que tentar criar uma instância de Guess com um valor fora do intervalo 1-100 resulta em um pânico. Link.

```
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);
    }
}
```

Verificando por Panic com should_panic

- Utilizamos o #[should_panic] para indicar que o teste deve entrar em pânico.
    - Se o teste entrar em pânico ele passa.
    - Se o teste não entrar em pânico ele falha.

- Se rodarmos o teste, ele irá passar, mostrando que ele entrou em pânico.

```
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

```
running 1 test
test tests::greater_than_100 - should panic ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s
```

Verificando por Panic com should_panic

- Agora vamos remover a verificação de “value > 100”, para o código não entrar em pânico.
- Nesse caso a mensagem de erro, não auxilia muito no entendimento do erro.

```
// --snip--
impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            // --snip--
```

```
running 1 test
test tests::greater_than_100 - should panic ... FAILED

failures:

---- tests::greater_than_100 stdout ----
note: test did not panic as expected

failures:
    tests::greater_than_100
```

Verificando por Panic com should_panic

- Um teste should_panic pode passar mesmo se o teste entrar em pânico por um motivo diferente do que estávamos esperando.
- Para tornar os testes should_panic mais precisos, podemos adicionar um parâmetro opcional expected ao atributo should_panic.
- O mecanismo de teste garantirá que a mensagem de falha contenha o texto fornecido.
- No próximo exemplo, onde a nova função entra em pânico com mensagens diferentes dependendo se o valor é muito pequeno ou muito grande.

Verificando por Panic com should_panic

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

Verificando por Panic com should_panic

- Este teste passará porque o valor que colocamos no parâmetro expected do atributo should_panic é uma parte da mensagem que a função Guess::new retorna ao entrar em pânico.

- Poderíamos ter especificado toda a mensagem de pânico no teste, nesse caso somente esse trecho da mensagem é o suficiente.

Esse código deve ser adicionado junto com o anterior. Link do código.

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

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

Verificando por Panic com should_panic

- Se mudarmos a string que ele retorna. Para ser diferente do esperado.

```
if value < 1 {
    panic!(
        "Guess value must be less than or equal to 100, got {}.",
        value
    );
} else if value > 100 {
    panic!(
        "Guess value must be greater than or equal to 1, got {}.",
        value
    );
}
```

- O código entrará em pânico, porém a mensagem será diferente do esperado, então o teste irá falhar.

Verificando por Panic com should_panic

- Essa mensagem traz mais informação sobre o motivo do erro, além de conseguir diferenciar o motivo pelo qual o código entrou em pânico.

```
running 1 test
test tests::greater_than_100 - should panic ... FAILED

failures:

---- tests::greater_than_100 stdout ----
thread 'tests::greater_than_100' panicked at 'Guess value must be greater than or
equal to 1, got 200.', src/lib.rs:13:13
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
note: panic did not contain expected string
      panic message: `"Guess value must be greater than or equal to 1, got 200."`,
 expected substring: `"less than or equal to 100"`

failures:
    tests::greater_than_100
```

Usando Result<T, E> em testes

- Por enquanto todos os testes que vimos, entravam em pânico quando falhavam.

- Também podemos escrever testes utilizando Result<T, E>, para ele retornar o erro em vez de entrar em pânico.

- A função it_works agora tem o tipo de retorno Result<(), String>.

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() -> Result<(), String> {
        if 2 + 2 == 4 {
            Ok(())
        } else {
            Err(String::from("two plus two does not equal four"))
        }
    }
}
```

Usando Result<T, E> em testes

- No corpo da função, em vez de chamar a macro assert_eq! retornamos:
  - Ok(()) quando o teste passa
  - Err com uma String dentro quando o teste falha.

- Escrever testes para que retornem um Result<T, E> permite que você use o operador de interrogação no corpo dos testes.

- Isso pode ser uma maneira conveniente de escrever testes que devem falhar se qualquer operação dentro deles retornar uma variante Err.

- Não é possível usar a anotação #[should_panic] em testes que usam Result<T, E>.

- Para afirmar que uma operação retorna uma variante Err, não use o operador de interrogação no valor Result<T, E>. Em vez disso, use assert!(valor.is_err()).

Controlando como os testes são rodados

- Agora vamos ver o que está acontecendo quando executamos nossos testes e explorar as diferentes opções que podemos usar com cargo test.

- Assim como “cargo run” compila o código e depois executa o arquivo binário. o “cargo test” compila o código em modo de teste e depois executa o arquivo binário de teste.

- O comportamento padrão do binário produzido pelo cargo test é executar todos os testes em paralelo e capturar a saída gerada durante a execução dos testes, impedindo que a saída seja exibida e facilitando a leitura da saída relacionada aos resultados dos testes.

- É possível especificar opções de linha de comando para alterar esse comportamento padrão.

Controlando como os testes são rodados

- Algumas opções de linha de comando são direcionadas ao cargo test, e outras são direcionadas ao binário de teste resultante.

- Após cargo test, podemos adicionar os argumentos para o cargo test, após separar utilizando “--” podemos adicionar os argumentos para o binário de teste.

  ```
  cargo test argumentos_para_cargo_test -- argumentos_para_o_binário
  ```

- Se digitarmos “cargo test --help” teremos um resultado diferente de “cargo test -- --help”.

Rodando testes em paralelo ou sequenciais

- Por padrão os testes são executados em paralelo, tornando eles mais rápidos.

- Não devemos criar testes dependentes uns dos outros e que compartilhem os mesmos recursos, se formos rodar eles de forma paralela.

- Se não quisermos rodar em paralelo, ou se quisermos ter um controle maior no número de threads que o sistema vai utilizar, podemos utilizar o comando:

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

Mostrando output da função

- Por padrão, o Rust filtra o que é impresso na saída padrão.
    - Se um teste passa, só é mostrado que ele passou não imprimindo a saída padrão.
    - Se um teste falha, veremos o que foi impresso na saída padrão junto com o restante da mensagem de falha.
- Para também mostrar o que está sendo impresso nos testes que estão passando, usando o comando:

```
$ cargo test -- --show-output
```

Mostrando output da função

- Link do código.

```
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);
    }
}
```

Mostrando output da função

- Rodando o teste com a flag “--show-output” teremos a mensagem imprimindo “I got the value 4”.

```
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
```

Rodando um subconjunto de testes pelo nome

- Você pode escolher quais testes executar passando para o “cargo test” o nome do teste que deseja executar como argumento.

- Nesse exemplo, temos 3 testes para a função add_two.

Exemplo:

```
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));
    }
}
```

Rodando um subconjunto de testes pelo nome

- Se rodarmos “cargo test one_hundred”, será testado somente a função teste one_hundred. Também será mostrado “2 filtered out”.

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

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out;
finished in 0.00s
```

- Se rodarmos “cargo test add”, será testado somente as funções teste com add no nome.

```
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 ignored; 0 measured; 1 filtered out;
finished in 0.00s
```

Ignorando alguns testes, a menos que seja chamado

- Alguns testes podem demorar muito tempo para terminar de ser executado. Então não seria tão interessante rodar ele em todo “cargo test”.

- Podemos utilizar “#[ignore]” para ignorar um teste.

- Output com “cargo test”:

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

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

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

test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Ignorando alguns testes, a menos que seja chamado

- Podemos utilizar o comando “cargo test -- --ignored”, para rodar somente os testes ignorados.

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

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

- Output com “cargo test -- --ignored”:

```
running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out;
finished in 0.00s
```

Organização de testes

- Quando a comunidade Rust pensa em testes, temos duas categorias principais: testes unitários e testes de integração.
    - Testes unitários são pequenos e mais focados, testando um módulo de cada vez em isolamento, e podem testar interfaces privadas.
    - Testes de integração são totalmente externos à sua biblioteca e usam seu código da mesma maneira que qualquer outro código externo, utilizando apenas a interface pública e potencialmente exercitando vários módulos por teste.

Testes unitários

- Testa cada unidade de código isoladamente do restante do código para identificar rapidamente onde o código está ou não está funcionando conforme o esperado.

- Todos os testes que vimos anteriormente são unitários.

- Você colocará testes unitários na pasta src em cada arquivo com o código que estão testando.

- A convenção é criar um módulo chamado tests em cada arquivo para conter as funções de teste e anotar o módulo com cfg(test).

Os módulos de teste e #[cfg(test)]

- O atributo cfg significa configuração e informa ao Rust que o item seguinte só deve ser incluído dado uma certa opção de configuração.
- A anotação “#[cfg(test)]” diz ao Rust, para não compilar quando utilizar “cargo build”, somente quando utilizar “cargo test”.
- Isso inclui qualquer função auxiliar que possa estar dentro deste módulo, além das funções anotadas com #[test].
- Como os testes unitários vão nos mesmos arquivos que o código é importante especificar isso para reduzir o custo computacional do build.

Template de teste:

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

Testando funções privadas

- Existe uma discussão sobre se funções privadas deveriam ser testadas ou não.
- Em Rust é possível testar funções privadas.
- Lembre que itens em módulos filhos podem usar os itens em seus módulos ancestrais.
- use super::*; traz todos os itens do módulo de teste para o escopo. Possibilitando o teste chamar a função privada internal_adder.

Exemplo:

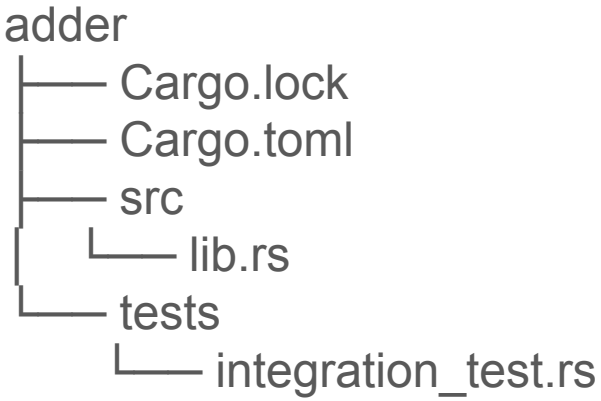
```
pub fn add_two(a: i32) -> i32 {
    internal_adder(a, 2)
}

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));
    }
}
```

Teste de integração

- Testes de integração são totalmente externos à sua biblioteca.
- O objetivo deles é testar se muitas partes de sua biblioteca funcionam corretamente juntas.
- Unidades de código que funcionam corretamente por conta própria podem ter problemas quando integradas, então a cobertura de testes do código integrado também é importante.

O diretório de testes

- Primeira devemos criar um diretório chamado “tests”, no nível superior do projeto, para os testes de integração.
- O cargo irá procurar pelos testes de integração nesse diretório.
- Cada arquivo no diretório tests é uma crate separada.
- Adicionamos “use adder;” no topo do código, para trazer nossa biblioteca para o escopo da crate de teste.
- Para testes de integração, não precisamos utilizar #[cfg(test)], já que o Rust só compila esses arquivos com cargo test.



Código de integration_test.rs:

```
use adder;

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

O diretório de testes

- Utilizando “cargo test”, poderemos ver os testes de integração, os unitários e os de documentação.

```
     Running unittests src/lib.rs (target/debug/deps/adder-1082c4b063a8fbe6)
running 1 test
test tests::internal ... ok
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

   Running tests/integration_test.rs
(target/debug/deps/integration_test-1082c4b063a8fbe6)
running 1 test
test it_adds_two ... ok
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

  Doc-tests adder
running 0 tests
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

O diretório de testes

- Se algum teste em uma seção falhar, as seções seguintes não serão executadas.
- Por exemplo: Os testes de integração são serão executados se todos os testes unitários passarem.
- Cada arquivo de teste de integração tem sua própria seção.
- Se quisermos rodar somente os testes de integração de um arquivo específico, podemos utilizar o comando “cargo test --test integration_test”

Submódulos em testes de integração

- Podemos criar mais arquivos no diretório de tests, para ajudar na organização.
- Se criarmos um arquivo tests/common.rs, para armazenar algumas funções auxiliares para os testes.
- Ele vai aparecer como uma sessão, mesmo sem ter um teste.

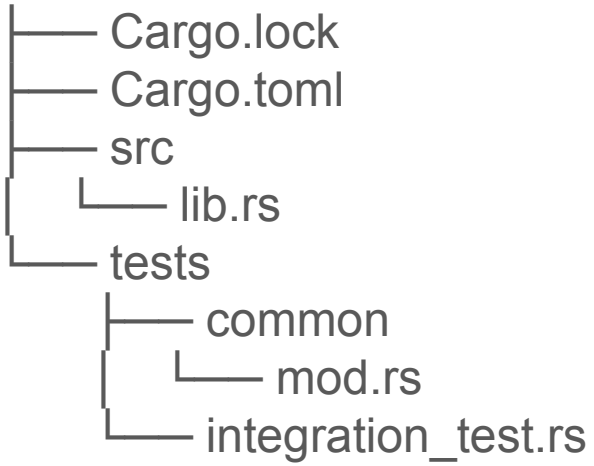
Código de common.rs:

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

```
 Running tests/common.rs
(target/debug/deps/common-92948b65e88960b4)

running 0 tests
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Submódulos em testes de integração

- Para evitar de common aparecer no output, em vez disso, criamos o arquivo tests/common/mod.rs.
- Arquivos em subdiretórios do diretório tests não são compilados como crates separadas nem têm seções na saída do teste.
- Podemos chamar a função setup desse arquivo adicionando “mod common;” ao código. Podemos chamar utilizando common::setup().



Código de integration_test.rs:

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

Código de mod.rs:

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

Testes de integração para crates binárias.

- Apenas crates de biblioteca src/lib.rs expõe funções que outras crates podem usar.
- Crates binárias src/main.rs são destinadas a serem executadas por conta própria.
- Se não tivermos uma crate de biblioteca, não poderemos realizar testes de integração no diretório tests e trazer funções definidas no arquivo src/main.rs
- Normalmente em projetos em Rust, as implementações ficam em crates de biblioteca, e src/main.rs chama essas implementações.
- Então normalmente se os testes das crates de biblioteca estão passando, não terá problemas com main.rs. Então acabam não criando testes para essa única crate binária.

Referências

Slides Baseados no Capítulo 11 do Rust Book
https://doc.rust-lang.org/book/ch11-00-testing.html