11. 自動テストを書く

- Kenta Arai

Copyright

- The Rust trademark is owned and protected by The Rust Foundation
- The Rust Programming Language is licensed under Apatch 2.0 and MIT terms
- This slide is a derivative work of TRPL, and is licensed under Apatch 2.0 and MIT terms

11.1. テストの記述法

- テストでは，テスト対象のコードが想定通りに動作することを検証する
- Rustのテストコードは一般に，下記の3つを実行する
  - 必要なデータと状態を準備する
  - テスト対象のコードを動作させる
  - 結果が想定通りであることを表明する(assert)

テストの構成

- `test` 属性で注釈された関数がテストターゲット
- ソースコードと同じファイル内にテストを記述する

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

テストを実行する

- テスト関数は単一のバイナリとしてコンパイルされる
- `cargo test` でテストプログラムが実行される

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.57s
     Running 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

   Doc-tests adder

running 0 tests

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

テストのお約束

- `Running` の後にテストの結果が表示される
  - テストがパスしたら `ok`，失敗すると `FAILED`
- `test result:` の行では，テストの結果が要約される
  - `passed` 及び `failed` はテストがパス／失敗した数
  - `ignored` は無視したために実行されなかったテスト（後述）
  - `measured` はベンチマークテスト向け（nightlyのみ）
  - `filtered out` はフィルタされたテストの数（後述）

assert!マクロでテストする

- `assert!`マクロは論理型を評価する

```
#[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_eq!/assert_nq!マクロでテストする

- 値の等値性を`assert_eq!`及び`assert_ne!`マクロで評価できる
- `assert`と異なり，非同値の場合に値を出力する

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

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

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

非同値な値をassert_eq!で検証する

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

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

failures:

---- tests::it_adds_two stdout ----
thread 'main' 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.


failures:
    tests::it_adds_two

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

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

カスタムの失敗メッセージを追加する

- `assert!`の第2引数，`assert_eq!`及び`assert_nq!`の第3引数に文字列を追加すると，エラーメッセージとして出力される

```
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
        );
    }
}
```

カスタムの失敗メッセージを確認する

```
$ cargo test
   Compiling greeter v0.1.0 (file:///projects/greeter)
    Finished test [unoptimized + debuginfo] target(s) in 0.93s
     Running target/debug/deps/greeter-170b942eb5bf5e3a

running 1 test
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----
thread 'main' panicked at 'Greeting did not contain name, value was `Hello!`', src/lib.rs:12:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace.


failures:
    tests::greeting_contains_name

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

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

panicする関数をテストする

- panicする関数は，戻り値から結果を判定できない
- `should_panic`属性によって，panicした場合にテストが成功として判定する

```
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); }
}
```

Resultでテストする

- `Result` 型で判定することもできる
  - `Ok` ならテストをパス
  - `Err` ならテストが失敗

```
#[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"))
        }
    }
}
```

11.2. テストの実行のされ方を制御する

テストを逐次に実行する

- `cargo test` は複数のテストを並列に実行する
- 共通の資源を利用する機能をテストする場合，注意が必要
- テストを逐次に実行したい場合は `--test-threads` を使用する

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

関数の出力を表示する

- テスト実行時は標準出力に出力されたものを全てキャプチャされる
- テストに失敗した場合にのみキャプチャされた内容が出力される
- 成功した場合も標準出力の内容を確認したい場合は `--nocapture` を使用する

```
$ cargo test -- --nocapture
```

単独のテストを実行する

- 名前を指定することで，特定のテストを実行できる
- `one_hundred` を実行したい場合は `cargo test one_hundred` を実行

```
$ cargo test one_hundred
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/adder-06a75b4a1f2515e9

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

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

複数のテストを実行する

- テスト名の一部を指定することで，その名前を含む関数を実行できる

```
$ cargo test add
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/adder-06a75b4a1f2515e9

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

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

テストを無視する

- テストしたくない関数がある場合，`ignore`属性で除外できる

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

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished dev [unoptimized + debuginfo] target(s) in 0.24 secs
     Running target/debug/deps/adder-ce99bcc2479f4607

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
```

無視したテストだけ実行する

- `ignore`属性を指定したテストのみ実行することも可能

```
$ cargo test -- --ignored
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/adder-ce99bcc2479f4607

running 1 test
test expensive_test ... ok

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

11.3. テストの体系化

単体テスト

- 各ファイルに`tests`モジュールを作成し，テスト関数を含ませ，`cfg(test)`で注釈する
- `cargo test`を実行した際にテスト関数が実行される
- 非公開関数もテストできる！

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

結合テストを作成する

- `src`ディレクトリと同じ階層に`tests`ディレクトリを作成すると，`Cargo`はそこに結合テストのファイルが置かれると認識する

```
// tests/integration_tests.rs
extern crate adder;

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

結合テストを実行する

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
     Running target/debug/deps/adder-abcabcabc

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

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

     Running target/debug/deps/integration_test-ce99bcc2479f4607

running 1 test
test it_adds_two ... ok

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

   Doc-tests adder

running 0 tests

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

結合テストだけを実行する

- 結合テストだけを実行したい場合，`--test` でテスト名を指定する

```
$ cargo test --test integration_test
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/integration_test-952a27e0126bb565

running 1 test
test it_adds_two ... ok

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

結合テスト内のサブモジュール

- 結合テスト用のサブモジュールは，さらに1つ深い階層に置くことで利用できる

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

// tests/integration_test.rs
extern crate adder;

mod common;

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

まとめ

- Rustのテスト機能は，機能が想定通りに動作することを確認する手段を提供する
- `Cargo` を用いてテストプログラムのコンパイルから実行までをサポートする