# 单元测试

单元测试的目的是在与其他部分隔离的环境中测试每一个单元的代码，以便于快速而准确的某个单元的代码功能是否符合预期。因此，单元测试与他们要测试的代码共同存放在位于 src 目录下相同的文件中。规范是在每个文件中创建包含测试函数的 tests 模块，并使用 cfg(test) 类属性宏标注模块。

例子：

|  |
| --- |
| 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));  }  } |

* #[cfg(test)] 注解告诉 Rust 只在执行 cargo test 时才编译和运行测试代码，而在运行 cargo build 时不这么做。
* #[test]注解告诉Rust这是一个测试函数。

# 集成测试

在 Rust 中，集成测试对于你需要测试的库来说完全是外部的。同其他使用库的代码一样使用库文件，也就是说它们只能调用一部分库中的公有 API 。为了创建集成测试，你需要在根目录下先创建一个 tests 目录，与 src 同级。接着可以随意在这个目录中创建任意多的测试文件，Cargo 会将每一个文件当作单独的 crate 来编译。

例如：

|  |
| --- |
| // src/lib.rs  pub fn add\_two(a: i32) -> i32 {  internal\_adder(a, 2)  }  fn internal\_adder(a: i32, b: i32) -> i32 {  a + b  }  // tests/integration\_test.rs  use adder;  #[test]  fn it\_adds\_two() {  assert\_eq!(4, adder::add\_two(2));  } |

## 公共测试模块

由于在test目录中的每一个文件都会被编译成一个测试程序，因此在test目录中无法直接添加具有公共函数的rs文件。rust使用了另一种方式可以在test目录下，添加被测试代码调用的公共方法。

可以在test目录下新增模块目录，然后在该目录下增加mod.rs文件，其中为该模块的代码。例如在test目录下新增common目录，并新增mod.rs文件。内容如下：

|  |
| --- |
| pub fn setup() {  // 编写特定库测试所需的代码  } |

然后可以在test下的测试代码中集成公共测试模块。例如集成上面定义的common模块：

|  |
| --- |
| use adder;  mod common;  #[test]  fn it\_adds\_two() {  common::setup();  assert\_eq!(4, adder::add\_two(2));  } |

# 测试函数

在函数fn之前添加 #[test]类属性宏，表明这是一个测试函数，这样测试执行者就知道将其作为测试处理。例如：

|  |
| --- |
| #[cfg(test)]  mod tests {  #[test]  fn it\_works() {  assert\_eq!(2 + 2, 4);  }  } |

rust提供了assert类测试宏、should\_panice类属性宏、Result<T,E>返回值这三种方式，用于指示测试是否成功。

## 测试宏

Rust提供了以下三个测试宏：

* assert：接收至少一个参数，如果为参数为false，则测试失败。
* assert\_eq：接收至少两个参数，如果第一个参数和第二个参数的值不等，则测试失败。
* assert\_ne：接收至少两个参数，如果第一个参数和第二个参数的值相等，则测试失败。

任何在 assert类的必需参数之后指定的参数都会传递给 format!宏，用于在测试失败的时候给出一个失败的原因。

例子：

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

## should\_panic类属性宏

Rust提供了should\_panice类属性宏，来检查代码是否按照期望处理错误。例如：

|  |
| --- |
| 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);  }  } |

另外也可以给 should\_panic 属性增加一个可选的 expected 参数，这样可以指定期望的整个 panic 信息。例子：

|  |
| --- |
| 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);  }  } |

## Result<T, E>返回值

Rust支持使用Result<T,E>返回测试的结果。例如：

|  |
| --- |
| #[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"))  }  }  } |

这样编写测试来返回 Result<T, E> 就可以在函数体中使用问号运算符，如此可以方便的编写任何运算符会返回 Err 成员的测试。

# 运行测试

cargo test 命令在测试模式下编译代码并运行生成的测试二进制文件。cargo test 生成的二进制文件的默认行为是并行的运行所有测试，并截获测试运行过程中产生的输出，阻止他们被显示出来，使得阅读测试结果相关的内容变得更容易。

可以将一部分命令行参数传递给 cargo test，而将另外一部分传递给生成的测试二进制文件。cargo test后面先列出的是传递给 cargo test 的参数，接着是分隔符 --，再之后是传递给测试二进制文件的参数。例如运行 cargo test --help 会提示 cargo test 的有关参数，而运行 cargo test -- --help 可以提示在分隔符 -- 之后使用的有关参数。

## 顺序运行测试

当运行多个测试时， Rust 默认使用线程来并行运行所有的测试。如果不希望测试并行运行，或者想要更加精确的控制线程的数量，可以传递 --test-threads 参数给测试二进制文件。例如：

|  |
| --- |
| $ cargo test -- --test-threads=1 |

## 显示函数输出

默认情况下，当测试通过时，Rust 的测试库会截获打印到标准输出的所有内容。如果希望看到通过的测试中打印的值，截获输出的行为可以通过 --nocapture 参数来禁用。例如：

|  |
| --- |
| $ cargo test -- --nocapture |

## 运行部分测试

默认情况下，cargo test会运行整个测试集。可以向 cargo test 传递任意测试的名称来只运行这个测试。例如：

|  |
| --- |
| // src/lib.rs  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 |

另外可以指定部分测试的名称，任何名称匹配这个名称的测试会被运行。例如：

|  |
| --- |
| // src/lib.rs  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 add |

## 忽略特定测试

可以使用 ignore 属性宏来标记耗时的测试并排除他们。例如：

|  |
| --- |
| #[test]  fn it\_works() {  assert\_eq!(2 + 2, 4);  }  #[test]  #[ignore]  fn expensive\_test() {  // 需要运行一个小时的代码  } |

可以使用 cargo test -- --ignored命令，专门运行被忽略的测试函数。例如：

|  |
| --- |
| $ cargo test -- --ignored |