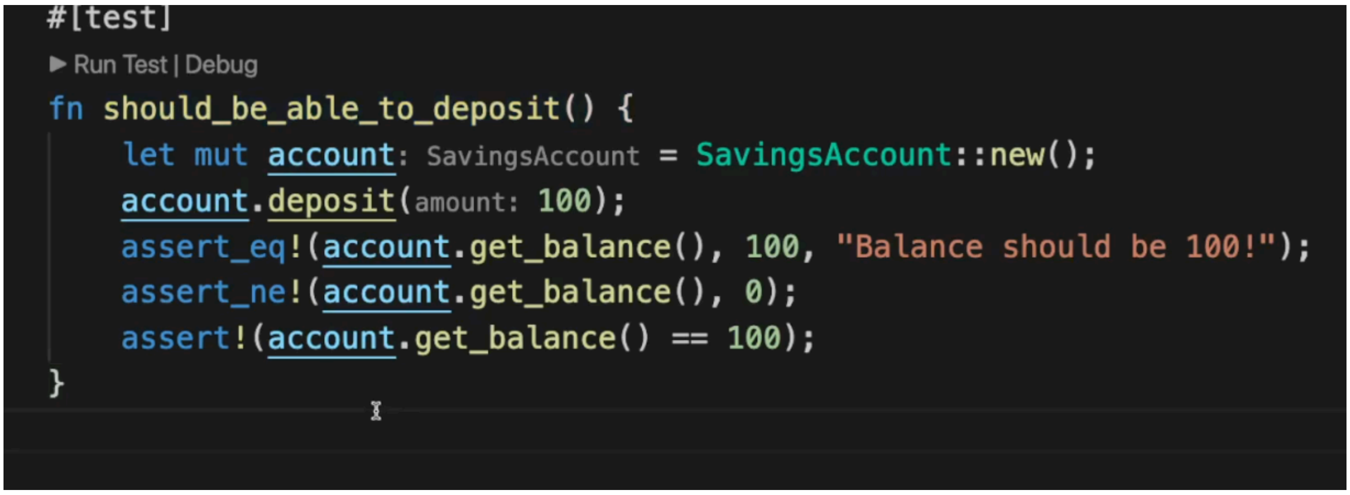
Rust - Tests

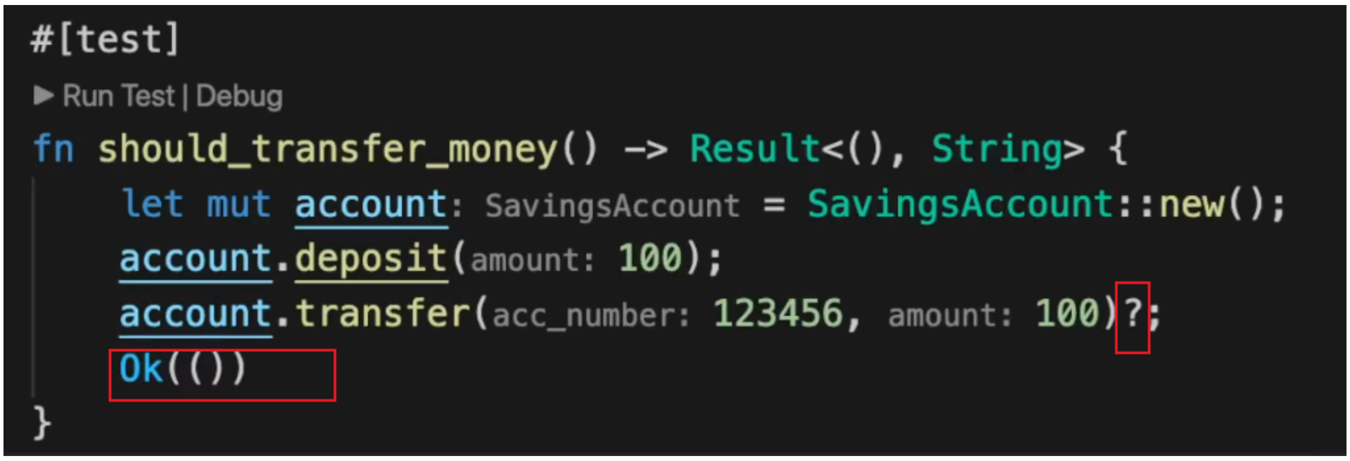
> Rust中 源码与测试代码是写在一块的

- #[cfg(test)]: 只在运行 test 指令时才会执行该部分代码 `cargo test`
- #[test]
- assert 相关宏

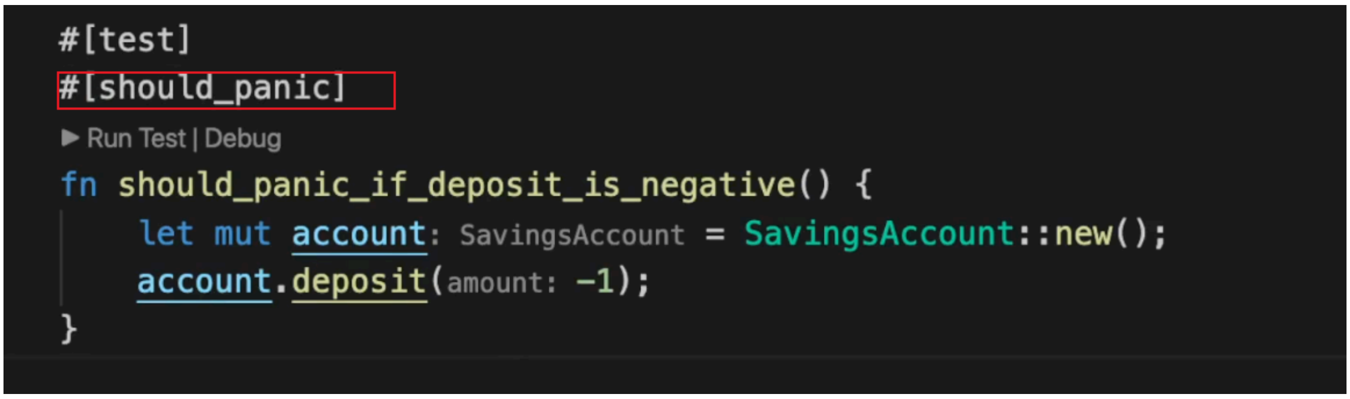
单元测试案例



```
#[test]
▶ Run Test | Debug
fn should_be_able_to_deposit() {
    let mut account: SavingsAccount = SavingsAccount::new();
    account.deposit(amount: 100);
    assert_eq!(account.get_balance(), 100, "Balance should be 100!");
    assert_ne!(account.get_balance(), 0);
    assert!(account.get_balance() == 100);
}
```



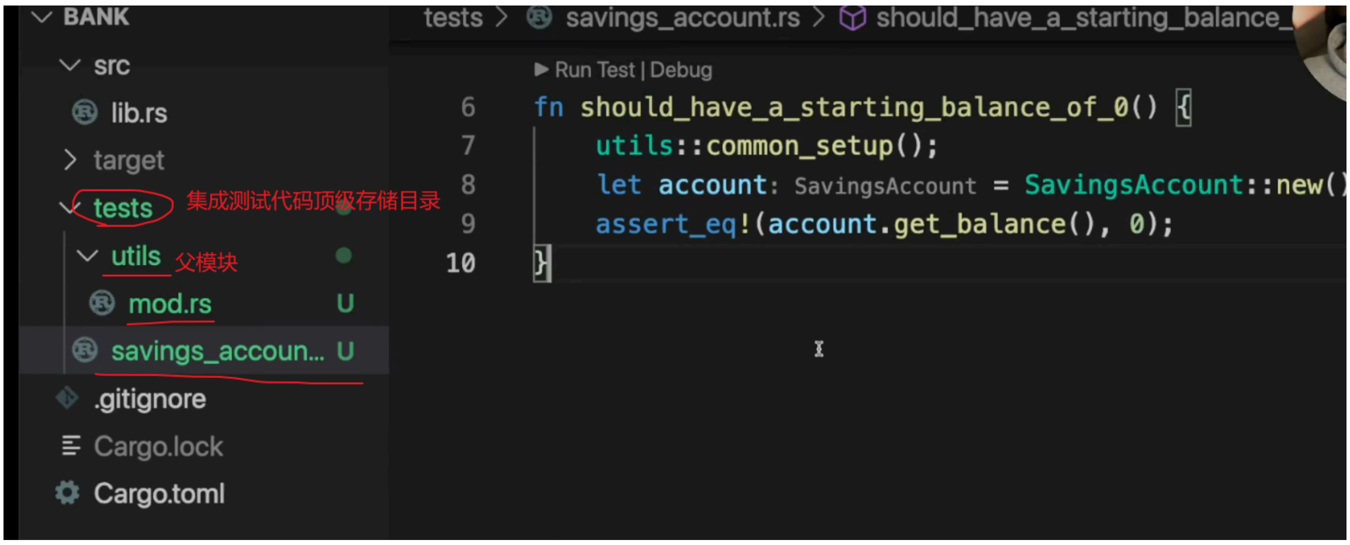
```
#[test]
▶ Run Test | Debug
fn should_transfer_money() -> Result<(), String> {
    let mut account: SavingsAccount = SavingsAccount::new();
    account.deposit(amount: 100);
    account.transfer(acc_number: 123456, amount: 100)?;
    Ok(())
}
```



```
#[test]
#[should_panic]
▶ Run Test | Debug
fn should_panic_if_deposit_is_negative() {
    let mut account: SavingsAccount = SavingsAccount::new();
    account.deposit(amount: -1);
}
```

集成测试

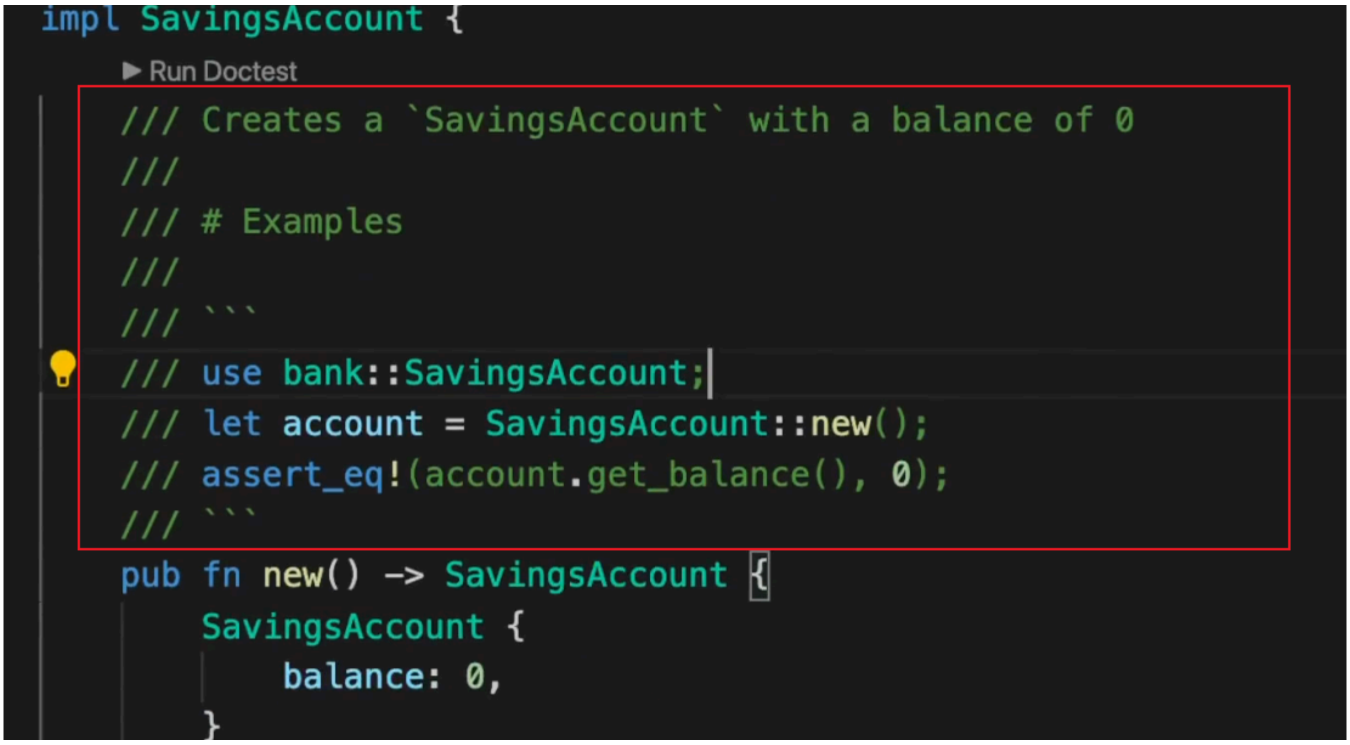
> 集成测试应该只测试公开API
>
> 测试多个单元之间的交互逻辑
>
> Rust中集成测试代码在顶级的 tests目录
>
> 集成测试 tests 目录下的每个 .rs 会被编译为独立的 crate
>
> 集成测试不能从一个 binary crate中导入内容



```rust
fn should_have_a_starting_balance_of_0() {
    utils::common_setup();
    let account: SavingsAccount = SavingsAccount::new(
    assert_eq!(account.get_balance(), 0);
}
```

Doc tests

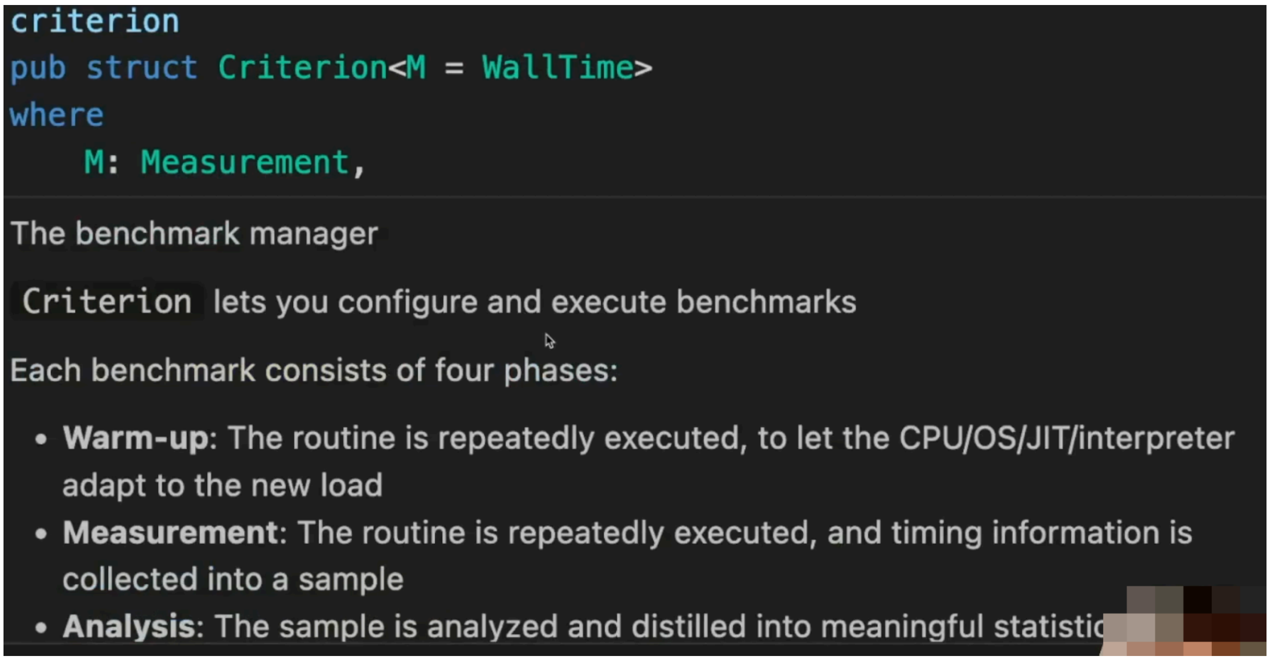
> - 使用 `///` 注释符
> - 支持 `///` 中编写markdown
> - `///` 中可以编写代码，rust 会在运行test时执行该代码片段，console中会出现`Doc-tests`
> - `cargo doc` 生成库文档
> - `cargo doc --open`: 生成文档并打开



```rust
impl SavingsAccount {
    ▶ Run Doctest
    /// Creates a `SavingsAccount` with a balance of 0
    ///
    /// # Examples
    ///
    /// ```
    /// use bank::SavingsAccount;
    /// let account = SavingsAccount::new();
    /// assert_eq!(account.get_balance(), 0);
    /// ```
    pub fn new() -> SavingsAccount {
        SavingsAccount {
            balance: 0,
        }
```

基准测试

- crate: `criterion`



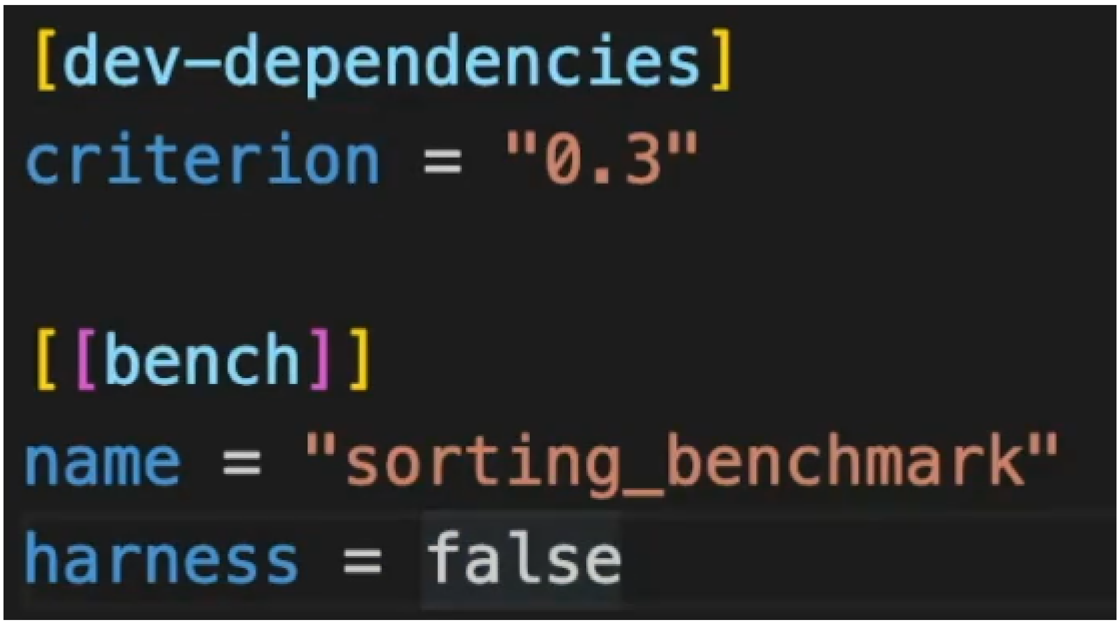
```rust
criterion
pub struct Criterion<M = WallTime>
where
    M: Measurement,
```

The benchmark manager

`Criterion` lets you configure and execute benchmarks

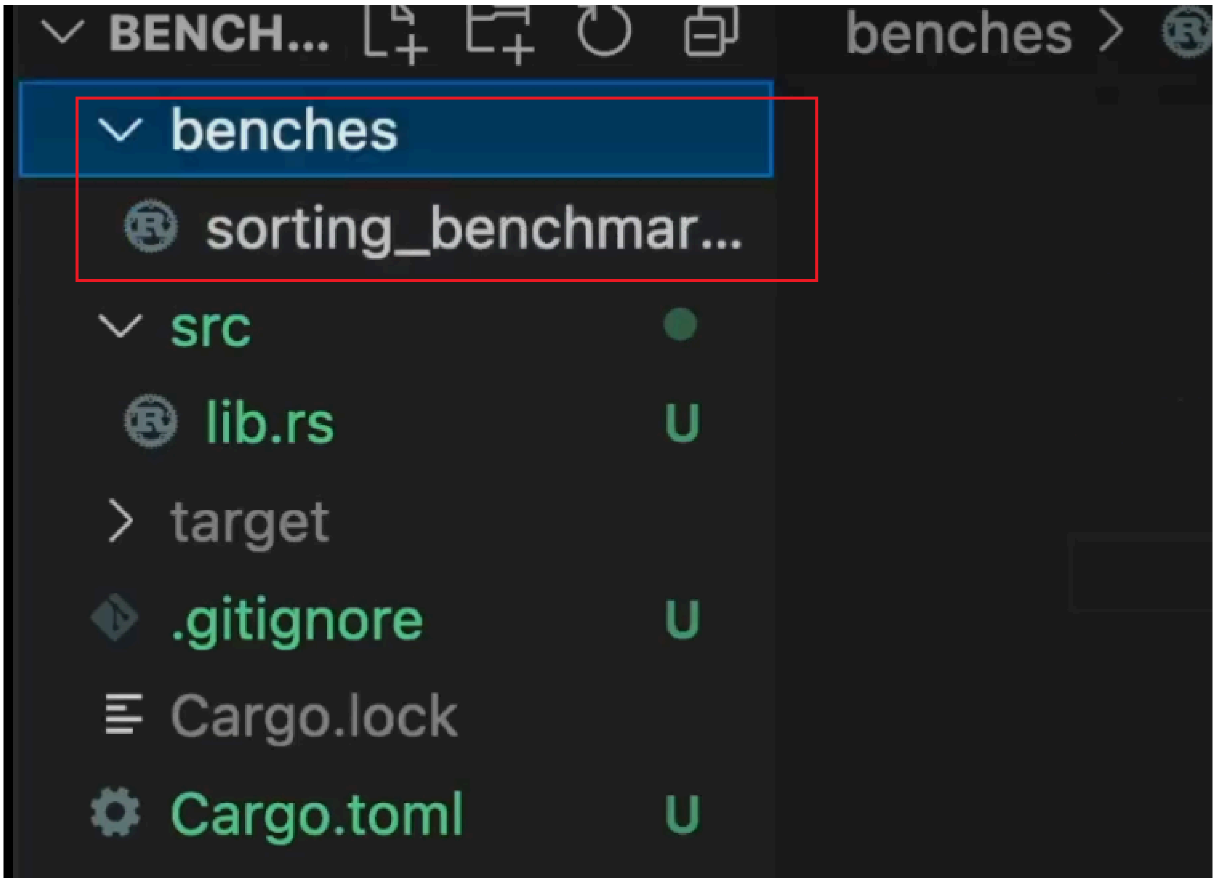
Each benchmark consists of four phases:

- **Warm-up**: The routine is repeatedly executed, to let the CPU/OS/JIT/interpreter adapt to the new load
- **Measurement**: The routine is repeatedly executed, and timing information is collected into a sample
- **Analysis**: The sample is analyzed and distilled into meaningful statistic

- 

```toml
[dev-dependencies]
criterion = "0.3"

[[bench]]
name = "sorting_benchmark"
harness = false
```



```
BENCH...                benches >
∨ benches
    sorting_benchmar...
∨ src                   ●
    lib.rs              U
> target
  .gitignore            U
  Cargo.lock
  Cargo.toml            U
```