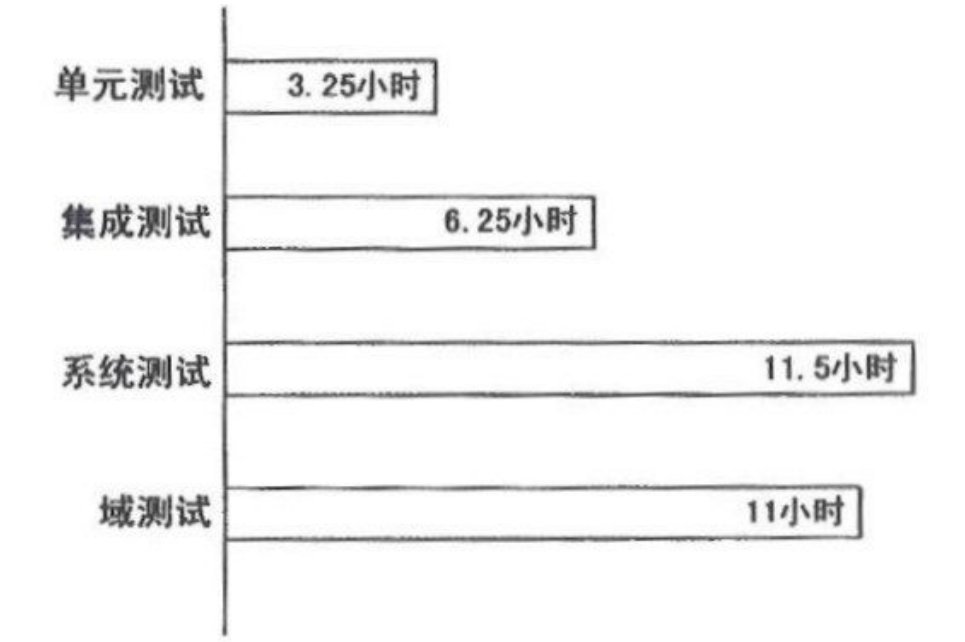
Spock单元测试

概念

单元测试介绍

> 单元测试是一段自动化的代码，这段代码调用被测试的工作单元，之后对这个单元的单个最终结果的某些假设进行检验。单元测试几乎都是用单元测试框架编写的。单元测试容易编写，能快速运行。单元测试可靠、可读，并且可维护。只要产品代码不发生变化，单元测试的结果是稳定的。

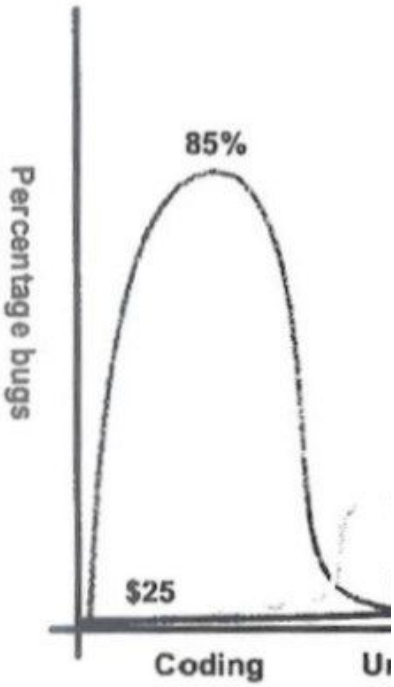
单元测试的意义



Microsoft applications, 10-20 defects/KLOC during unit testing, 0.5 defects/KLOC after release.

图 7-1 微软公司测试的数据统计结果

这张图来自微软的统计数据：Bug在单元测试阶段被发现，平均耗时3.25小时，如果漏到系统测试阶段，要花费11.5小时。



Source:Applied S

图

这张图，旨在说明两个问题：85%级别的增高。

常见框架对比

	Spock	Mockito	PowerMock	Junit
框架独立	是	否	否	是
Mock功能	是	是	是	否
私有方法、静态方法支持	否	是	是	否
代码简洁	高	低	低	一般
学习成本	中	低	低	低

Spock 介绍

介绍

> Spock 是一款基于 Groovy 的测试框架，它结合了 JUnit、Mockito 和 JBehave 的优点，为测试带来了更多的功能和灵活性。Spock 框架支持 BDD（行为驱动开发）和 TDD（测试驱动开发）两种测试风格，并具有易读、易编写和易于维护的特点。

- 让测试代码更规范，内置多种标签来规范单元测试代码的语义，测试代码结构清晰，更具可读性，降低后期维护难度。
- 提供多种标签，比如：`given`、`when`、`then`、`expect`、`where`、`with`、`thrown`……提供结构化语法，帮助我们应对复杂的测试场景，
- 使用Groovy这种动态语言来编写测试代码，可以让我们编写的测试代码更简洁，适合敏捷开发，提高编写单元测试代码的效率。
- 遵从BDD（行为驱动开发）模式，有助于提升代码的质量。
- IDE兼容性好，自带Mock功能。

Spock使用说明

引入说明

> Spock分1.x和2.x两个版本，Spock1.x对应Junit4版本，可以兼容支持PowerMock框架，2.x版本基于Junit5，由于Junit5支持性问题，Spock已经移除Sputnik，不再支持代理运行power mock的方式。
>
> 本次介绍主要介绍Spock2.x版本

Maven 引入

```xml
<!-- Spock引入 -->
<dependency>
    <groupId>org.spockframework</groupId>
    <artifactId>spock-spring</artifactId>
    <version>2.0-M5-groovy-3.0</version>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <artifactId>junit</artifactId>
            <groupId>junit</groupId>
        </exclusion>
    </exclusions>
</dependency>
<!-- Mockito引入 -->
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-inline</artifactId>
    <version>4.3.1</version>
    <scope>test</scope>
</dependency>
```

语法说明

	A	B
1	**Spock**	**JUnit**
2	Specification	Test class
3	setup()	@Before
4	cleanup()	@After
5	setupSpec()	@BeforeClass
6	cleanupSpec()	@AfterClass
7	Feature	Test
8	Feature method	Test method
9	Data-driven feature	Theory
10	Condition	Assertion
11	Exception condition	@Test(expected=…)
12	Interaction	Mock expectation (e.g. in Mockito)

given, when, then,where, and说明

分块	替换	功能	说明
given	setup	初始化函数、MOCK	非必要
when	expect	执行待测试的函数	when 和 then 必须成对出现
then	expect	验证函数结果	when 和 then 可以被 expect 替换
where		多套测试数据的检测	spock的特性功能
and		对其余块进行分隔说明	

- `given`：输入条件（前置参数）。
- `when`：执行行为（`Mock`接口、真实调用）。
- `then`：输出条件（验证结果）。
- `and`：衔接上个标签，补充的作用。
- expect: 相当于when 和then的结合

Where说明：

输入参数| 输入参数 || 输出值| 输出值

groovy语法介绍

对象构建

```
// 用闭包
def resource = new Resource().with {
        setResourceId(1L)
        return it
}

// 用构造传参
def resource = new Resource(resourceId: 1L)
```

字符串

```
// java
String json = "{\"name\":\"tom\",\"age\":18}"
// groovy
def json = '''{"name": "tom","age": 18}'''
```

列表

```
def resourceList = [
    new Resource(resourceId: 2L),
    new Resource(resourceId: 3L),
    new Resource(resourceId: 1L),
    new Resource(resourceId: 4L)
]

// << 可以替代 list.add
resourceList << new Resource(resourceId: 5L) << new Resource(resourceId: 6L)
```

Spock简单实例

先介绍一个简单的代码示例：

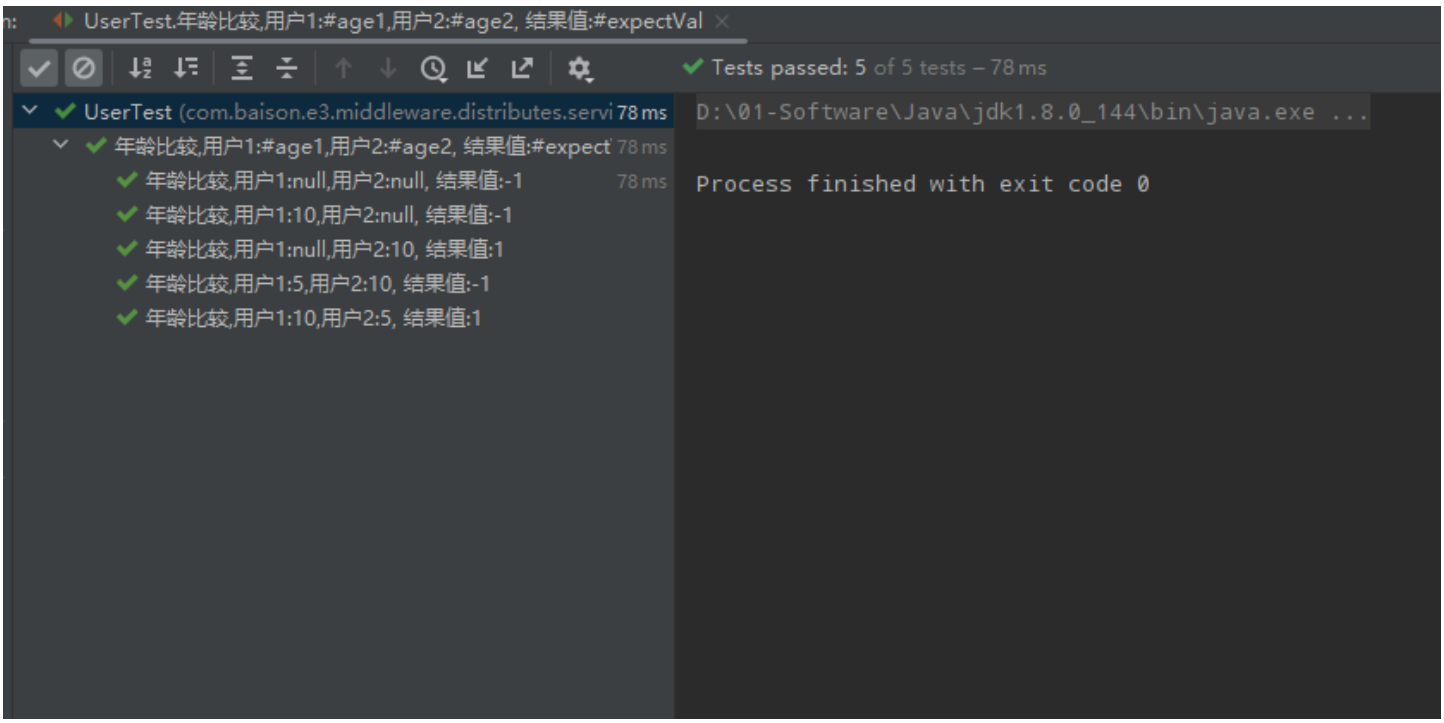
```
public static int compare(User user1, User user2) {
    if(user1.getAge() == null && user2.getAge() == null) {
        return -1;
    }
    if (user1.getAge() == null) {
        return 1;
    }
    if (user2.getAge() == null) {
        return -1;
    }
    return user1.getAge().compareTo( user2.getAge());
}
```

测试示例：

```
def "年龄比较,用户1:#age1,用户2:#age2, 结果值:#expectVal"() {
    expect:
    User.compare(new User(age:age1), new User(age: age2)) == expectVal
    //where    ||双竖杆表示运行后输出值
    where:
    age1 | age2 || expectVal
    null | null || -1
    10   | null || -1
    null | 10   || 1
    5    | 10   || -1
    10   | 5    || 1
```

```
12 }
```

执行结果：



Mock功能

目前大部分服务架构都是微服务模式，在业务实现接口中会调用到大量的rpc或其他接口，Spock对此也提供了强大的Mock功能。

简单功能mock

```
1 def mock(){
2     given:
3     def mockUser = Mock(User)
4     //mock返回blob
5     mockUser.getName() >> "blob"
6     expect:
7     mockUser.getName() == "blob"
8     mockUser.age == ?
9 }
```

链式mock

```
1 def mockMore(){
```

```
    given:
    def mockUser = Mock(User)
    //多次返回用>>> ,第一次返回blob，第二次返回jason，第三次返回tom，并且Mock属性时，可
    mockUser.name >>> ["blob", "jason", "tom"]
    expect:
    mockUser.getName() == "blob"
    mockUser.getName() == "jason"
    mockUser.getName() == "tom"
}

//以上多次mock还可以这么写
def mockMore(){
    given:
    def mockUser = Mock(User)
    //多次返回用>>> ,第一次返回blob，第二次返回jason，第三次返回tom，并且Mock属性时，可
    1 * mockUser.name >> "blob"
    2* mockUser.name >>>  ["jason", "tom"]

    expect:
    mockUser.getName() == "blob"
    mockUser.getName() == "jason"
    mockUser.getName() == "tom"
}
```

异常mock

```
public User getUserException() {
    return new User();
}

def mockException() {
    given://定义，mock方法
    def mockUser = Mock(User)
    mockUser.getUserException() >> {throw new BusinessException()}

    when: //业务调用执行
    mockUser.getUserException()

    then://断言
    thrown(BusinessException)
}
```

复杂场景Mock

在常规业务代码中，会大量调用dao层或者外部服务接口，正常服务执行时会通过spring注入，调用接口实际返回值。在单元测试场景中，我们会对这种需要调用资源服务或远程接口进行mock，返回一个预期内的值后，再实现自己的代码逻辑。

```
//业务代码
public UserVO findUser(Long id) {
    User user = userDao.findById(id);
    if (user == null) {
        return null;
    }
    UserVO userVO = new UserVO();
    userVO.setName(user.getName());
    userVO.setAge(user.getAge());
    userVO.setTelephone(user.getTelephone());
    if (user.getAge() > 0 && user.getAge() < 18) {
        userVO.setGroup("少年组");
    } else if (user.getAge() >= 18 && user.getAge() <= 30) {
        userVO.setGroup("青年组");
    } else if (userVO.getAge() > 30 && userVO.getAge() < 60) {
        userVO.setGroup("中年组");
    } else {
        userVO.setGroup("老年组");
    }
    return userVO;
}

//单元测试示例
//1、定义mock接口
def userDao = Mock(UserDao)
//新建service服务接口
def userService = new UserServiceImpl(userDao: userDao)
//userService.userDao = userDao

def findById() {
    given:
    // _ 下划线表示任意入参，Mockito.any()
    userDao.findById(_) >> new User(age: 10, name: "tom", telephone: "123456789"

    expect:
    def result = userService.findUser(3L)
    with(result) {
        group == "少年组"
    }
}
```

Spock mock& spy & stub差异点

Spock除了mock功能外，还提供spy, stub功能

- mock对象，除mock方法外，其他方法的返回值都是null
- spy对象，除mock方法外，其他方法都是真实调用
- stub对象，除mock方法外，其他方法返回是无意义的

```
@Data
public class User {

    private Integer age;
    private String name;

    public String getSex() {
        return "男";
    }
 }

    def mock() {
    given:
    def mockPerson = Mock(User)
    def spyPerson = Spy(User)
    def stubPerson = Stub(User)
    mockPerson.getName() >> "bob"
    spyPerson.getName() >> "bob"
    stubPerson.getName() >> "bob"

    expect:
    mockPerson.getName() == "bob"
    spyPerson.getName() == "bob"
    stubPerson.getName() == "bob"
    mockPerson.getSex() == null
    spyPerson.getSex() == "男"
    stubPerson.getSex() != "男" && stubPerson.getAge() != null
}
```

异常验证

以下是一个常见的校验方法，在方法里，如果校验失败，则抛出异常

```
public void validateUser(User user) throws BAPRuntimeException {
```

```
    if(user == null){
        throw new BAPRuntimeException("10001", "user is null");
    }
    if(StringUtils.isBlank(user.getName())){
        throw new BAPRuntimeException("10002", "user name is null");
    }
    if(user.getAge() == null){
        throw new BAPRuntimeException("10003", "user age is null");
    }
    if(StringUtils.isBlank(user.getTelephone())){
        throw new BAPRuntimeException("10004", "user telephone is null");
    }
    if(StringUtils.isBlank(user.getSex())){
        throw new BAPRuntimeException("10005", "user sex is null");
    }
}

```

Spock内置`thrown()`方法，可以捕获调用业务代码抛出的预期异常并验证，再结合`where`表格的功能，可以很方便地覆盖多种自定义业务异常

```
def "validate user info: #expectedMessage"() {
        when: "校验"
        def tester = Spy(User)
        tester.validateUser(userCondtion)

        then: "验证"
//        thrown(expectedException)

        def exception = thrown(expectedException)
        exception.code == expectedCode
        exception.message == expectedMessage

        where: "测试数据"
        userCondtion   || expectedException   | expectedCode | expectedMessage
        getUser(10001) || BAPRuntimeException | "10001" | "user is null"
        getUser(10002) || BAPRuntimeException | "10002"      | "user name is nul
        getUser(10003) || BAPRuntimeException | "10003"      | "user age is null
        getUser(10004) || BAPRuntimeException | "10004"      | "user telephone i
        getUser(10005) || BAPRuntimeException | "10005"      | "user sex is null
    }

def getUser(code) {
    def user = new User()
    def condition1 = {
```

```
        user.name = "张三"
    }
    def condition2 = {
        user.age = 20
    }
    def condition3 = {
        user.telephone = "12345678901"
    }
    def condition4 = {
        user.sex = "男"
    }

    switch (code) {
        case 10001:
            user = null
            break
        case 10002:
            user = new User()
            break
        case 10003:
            condition1()
            break
        case 10004:
            condition1()
            condition2()
            break
        case 10005:
            condition1()
            condition2()
            condition3()
            break
    }
    return user
}
```

静态代码mock

Spock2.x版本由于Junit5，无法支持PowerMock组件，Mockito看到了这个机会，实现原有功能的突破，进入PowerMock原先的领域，Mockito自2.1版本之后支持final版本，3.4.0版本之后支持Mock静态方法。因此在涉及静态方法时，我们可以用mockito 替代PowerMock，使用Mockito mock静态代码时，需要手动关闭

```
public UserVO findUser(Long id) {
    User user = userDao.findById(id);
    if (user == null) {
```

```
        return null;
    }
    UserVO userVO = new UserVO();
    userVO.setName(user.getName());
    userVO.setAge(user.getAge());
    userVO.setTelephone(user.getTelephone());
    if (user.getAge() > 0 && user.getAge() < 18) {
        userVO.setGroup("少年组");
    } else if (user.getAge() >= 18 && user.getAge() <= 30) {
        userVO.setGroup("青年组");
    } else if (userVO.getAge() > 30 && userVO.getAge() < 60) {
        userVO.setGroup("中年组");
    } else {
        userVO.setGroup("老年组");
    }

    //调用静态代码，获取权限
    Map<String, String> permission = PermissionUtil.getPermission(user.getName()
    userVO.setPermission(permission);
    return userVO;
}

//单元测试
MockedStatic<PermissionUtil> mockedStatic;

def testStatic() {
    given:
//    mockedStatic = Mockito.mockStatic(PermissionUtil.class)
    userDao.findById(_) >> new User(age: 10, name: "tom", telephone: "123456789"

    Map<String, String> map = new HashMap<>()
    map.put("role1", "role1")
    map.put("role2", "role2")
    Mockito.when(PermissionUtil.getPermission("tom")).thenReturn(map)

    expect:
    def result = userService.findUser(3L)
    with(result) {
        group == "少年组"
        permission.get("role1") == "role1"
    }
    //手动关闭
    mockedStatic.close()
}
```

私有方法测试

有时候一些私有方法有变动，或者从public方法级别测试比较困难，如果想要回归验证，只能对私有方法进行验证。Spock本身是不直接支持测试私有方法，但是可以通过反射机制，间接实现单元测试效果

还是获取用户的示例，这次方法声明是private

```
private UserVO getUserVO(Long id) {
    User user = userDao.findById(id);
    if (user == null) {
        return null;
    }
    UserVO userVO = new UserVO();
    userVO.setName(user.getName());
    userVO.setAge(user.getAge());
    userVO.setTelephone(user.getTelephone());
    if (user.getAge() > 0 && user.getAge() < 18) {
        userVO.setGroup("少年组");
    } else if (user.getAge() >= 18 && user.getAge() <= 30) {
        userVO.setGroup("青年组");
    } else if (userVO.getAge() > 30 && userVO.getAge() < 60) {
        userVO.setGroup("中年组");
    } else {
        userVO.setGroup("老年组");
    }

    //调用静态代码，获取权限
    Map<String, String> permission = PermissionUtil.getPermission(user.getName()
    userVO.setPermission(permission);
    return userVO;
}

//单元测试
def testPrivate() {
    given:
    mockedStatic = Mockito.mockStatic(PermissionUtil.class)
    userDao.findById(_) >> new User(age: 10, name: "tom", telephone: "123456789"

    Map<String, String> map = new HashMap<>()
    map.put("role1", "role1")
    map.put("role2", "role2")
    Mockito.when(PermissionUtil.getPermission("tom")).thenReturn(map)

    expect:
    //通过反射机制拿到method方法，并且反射执行
    Method method = UserServiceImpl.class.getDeclaredMethod("getUserVO", Long.cl
    method.setAccessible(Boolean.TRUE)
```

```
    def result = method.invoke(userService, 3L)

    //def result = userService.findUser(3L)
    with(result) {
        group == "少年组"
        permission.get("role1") == "role1"
    }
    // result.group == "少年组"
    //手动关闭
    mockedStatic.close()
}
```

void方法测试

在项目中，也有很多void方法，我们无法从接口调用的响应值判断接口执行是否成功，常规的断言脚本无法直接验证数据的准确性。但是细思考下，每个接口的执行都是有意义的，一般分以下几种

- 在方法中更改入参对象
- 在方法中实现调用外部接口
- 是否走到某个分支的代码，调用了某一个接口
- for循环中方法调用了几次

从这些视角，我们可以验证方法执行的准确性，以下是一个void方法

> ```
> 验证调用次数说明
> 1 * subscriber.receive("hello")      // exactly one call0 * subscriber.receive
> (1..3) * subscriber.receive("hello") // between one and three calls (inclusive
> (1.._) * subscriber.receive("hello") // at least one call
> (_..3) * subscriber.receive("hello") // at most three calls
> _ * subscriber.receive("hello")      // any number of calls, including zero//
> ```

```
public void updateUser(User user) {
    if (user == null || user.getAge() == null) {
        return;
    }
    int count;
    if (user.getAge() < 18) {
        count = userDao.countGroup("少年");
    } else if (user.getAge() >= 18 && user.getAge() < 30) {
        count = userDao.countGroup("青年");
        System.out.println(count);
    } else if (user.getAge() >= 30 && user.getAge() < 60) {
```

```
12          count = userDao.countGroup("中年");
13      } else {
14          count = userDao.countGroup("老年");
15      }
16      user.setCount(count);
17
18      userDao.updateUser(user);
19  }
20
21  //单元测试，这个单元测试中，发现一个很有意思的问题，userDao.countGroup这个方法，既有mock
22  //如果在断言中，判断  n * userDao.countGroup("青年")，则会报错，但实际上程序中是有真实
23  def testVoid() {
24          given:
25          userDao.updateUser(_ as User) >> 1
26          userDao.countGroup("青年") >> 10
27
28          when:
29          userService.updateUser(userCondition)
30
31          then:
32          verifyAll {
33              //验证方法调用时入参
34              m * userDao.updateUser({it.count == 10})
35  //              n * userDao.countGroup("青年")
36              //验证方法执行次数
37              n * userDao.countGroup("少年")
38          }
39
40          where:
41          userCondition     || n | m
42          new User(age: 25) || 0 | 1
43      }
```

DAO层测试

MockData

dbUnit

> DbUnit 是一个 Java 库，用于在单元测试中管理数据库状态。它的作用是使得单元测试更加可靠和可重复，特别是在需要测试与数据库交互的应用程序时。
>
> 具体来说，DbUnit 可以将测试用例和数据库之间的数据交互封装起来，使得测试用例可以独立于数据库状态进行运行。它提供了一组 API，用于在测试用例执行前准备数据库状态，以及在测试用例执行后还原数据库状态。

> 通过使用 DbUnit，开发人员可以在测试用例中方便地创建、插入、更新和删除数据库中的数据，而无需手动编写 SQL 语句。此外，DbUnit 还支持使用 XML 或 CSV 文件来定义测试数据，以及使用数据库中的数据集合来验证测试结果。

H2说明

> H2是一个开放源码的轻量级Java数据库。它可以嵌入到Java应用程序中或以客户端 - 服务器模式运行。主要是H2数据库可以配置为作为内存数据库运行，这意味着数据不会在磁盘上持久存储。由于嵌入式数据库不适用于生产开发，而是主要用于开发和测试。

总结：

单元测试有很多优点，如下：

1、降低开发bug错误率，提高测试场景覆盖率

2、提升代码质量，通过单元测试写出更模块化，结构的代码

3、是代码重构时的保证利器

4、代码更容易维护

但同时也有一些缺点

1、学习成本高

2、开发时间长

3、推广和运用单元测试需要比较大的投入，同时收效周期比较长

参考文档

Spock: https://spockframework.org/spock/docs/2.3/all_in_one.html

Mockito: https://javadoc.io/static/org.mockito/mockito-core/5.3.0/org/mockito/Mockito.html