单元测试规范

一、可衡量：单测的编写应该是可以用具体的指标衡量的

- 增量代码单测通过率要求100%，行覆盖率要求30%（保底）
- 代码有逻辑变更时，单测也应该做相应的变更（回归）
- 单测CASE分级标准
- - Level1：正常流程可用，即一个函数在输入正确的参数时，会有正确的输出
  - Level2：异常流程可抛出逻辑异常，即输入参数有误时，不能抛出系统异常，而是用自己定义的逻辑异常通知上层调用代码其错误之处
  - Level3：极端情况和边界数据可用，对输入参数的边界情况也要单独测试，确保输出是正确有效的
  - Level4：所有分支、循环的逻辑走通，不能有任何流程是测试不到的
  - Level5：输出数据的所有字段验证，对有复杂数据结构的输出，确保每个字段都是正确的

P3用例请做到Level2（暂不考虑）、P2用例请做到Level3（保底30%）、P1用例请做到Level4（保底 60%）

二、独立性：单测应该是独立且相互隔离的

- 一个单测只测试一个方法
- 单元测试之间决不能互相调用，也不能依赖执行的先后次序
- 单测应在最小测试环境运行（不要启动Spring相关容器）
- 单测如果涉及到数据变更，必须进行回滚（或使用内存数据库）
- 单测应该测试目标方法本身的逻辑，对于被测试的方法内部调用其他所有方法都应进行Mock

三、规范性：单测的编写需要符合一定规范

- 单测需写在test目录下，且与原方法保持同路径，命名XXXTest，方法命名为test_targetMethod_normal: desc 或 test_targetMethod_exception: desc
- 单测注释需编写完整，至少包含：@see 被测试方法、给定了什么数据、触发了什么动作、预期什么结果
- 单测应该是无状态的，即可以重复执行
- 单测应覆盖所有提供了逻辑的类，Service层、Domain层及部分包含了逻辑的Controller层等，而不覆盖各种POJO（DO，DTO，VO...）
- 私有方法、静态方法也应编写单测
- 单测应使用断言、异常测试工具、verify等验证预期结果、异常结果与Mock方法签名、调用次数、参数，而不能根据输出进行人肉验证
- 单测应在相应方法开发完后立即编写

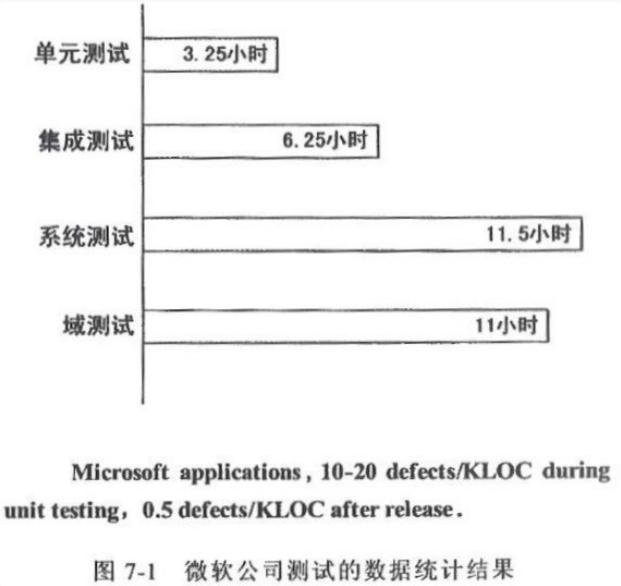
单元测试实践

0 写在最前面

XML之父Tim Bray最近在博客里有个好玩的说法：“代码不写测试就像上了厕所不洗手……单元测试是对软件未来的一项必不可少的投资。”具体来说，单元测试有哪些收益呢?

- 它是最容易保证代码覆盖率达到100%的测试。
- 可以大幅降低上线时的紧张指数。
- 单元测试能更快地发现问题（见下图左）。
- 单元测试的性价比最高，因为错误发现的越晚，修复它的成本就越高，而且难度呈指数式增长，所以我们要尽早地进行测试（见下图右）。
- 编码人员，一般也是单元测试的主要执行者，是唯一能够做到生产出无缺陷程序的人，其他任何人都无法做到这一

点。
- 有助于源码的优化，使之更加规范，快速反馈，可以放心进行重构。
- 提升研发效能，无须等待其他依赖模块ready，即可验证方法逻辑

 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。	这张图，旨在说明两个问题：85%的缺陷都在代码设计阶段产生，而发现Bug的阶段越靠后，耗费成本就越高，指数级别的增高。

尽管单元测试有如此的收益，但在我们日常的工作中，仍然存在不少项目它们的单元测试要么是不完整要么是缺失的。常见的原因总结如下：

- 代码逻辑过于复杂。
- 写单测麻烦，一堆重复代码，没有价值、幸福感。
- 写单元测试时耗费的时间较长，没有统一规范。
- 任务重、工期紧，或者干脆就不写了。

基于以上问题，相较于传统的JUnit单元测试，今天为大家推荐Spock + TestableMock的单测方式，希望能够帮助大家提高开发测试的效率。

Spock官网：https://spockframework.org/

TestableMock官网：https://alibaba.github.io/testable-mock/#/

相关代码：https://code.aone.alibaba-inc.com/luoyufeng.lyf/spock-testablemock-demo

1 单测之道

1.1 单测要求-《Java开发手册》规范

【强制】好的单元测试必须遵守AIR原则。 说明：单元测试在线上运行时，感觉像空气（AIR）一样感觉不到，但在测试质量的保障上，却是非常关键的。好的单元测试宏观上来说，具有自动化、独立性、可重复执行的特点。

A：Automatic（自动化）

I：Independent（独立性）

R：Repeatable（可重复）

【强制】单元测试应该是全自动执行的，并且非交互式的。测试用例通常是被定期执行的，执行过程必须完全自动化才有意义。输出结果需要人工检查的测试不是一个好的单元测试。单元测试中不准使用System.out来进行人肉验证，必须使用assert来验证。

【强制】单元测试是可以重复执行的，不能受到外界环境的影响。单元测试通常会被放到持续集成中，每次有代码check in时单元测试都会被执行。如果单测对外部环境（网络、服务、中间件等）有依赖，容易导致持续集成机制的不可用。

【推荐】编写单元测试代码遵守BCDE原则，以保证被测试模块的交付质量。

B：Border，边界值测试，包括循环边界、特殊取值、特殊时间点、数据顺序等。

C：Correct，正确的输入，并得到预期的结果。

D：Design，与设计文档相结合，来编写单元测试。

E：Error，强制错误信息输入（如：非法数据、异常流程、业务允许外等），并得到预期的结果。

【强制】对于单元测试，要保证测试粒度足够小，有助于精确定位问题。单测粒度至多是类级别，一般是方法级别。 说明：只有测试粒度小才能在出错时尽快定位到出错位置。单测不负责检查跨类或者跨系统的交互逻辑，那是集成测试的领域。

【推荐】单元测试的基本目标：语句覆盖率达到70%；核心模块的语句覆盖率和分支覆盖率都要达到100% 说明：在工程规约>应用分层中提到的DAO层，Manager层，可重用度高的Service，都应该进行单元测试。

【补充】单元测试应该是快的。单测不仅仅是给持续集成跑的，跑测试更多的是程序员本身，这样才能尽快的发现问题，单测速度和程序员跑单测的意愿成反比，如果单测只要5秒，程序员会经常跑单测，去享受一下全绿灯的满足感，可如果单测要跑5分钟，能在提交前跑一下单测就不错了。

1.2 下个定义

不依赖任何外部环境，对软件中的最小可测试单元（Method）进行检查和验证，旨在提高软件交付质量及整个生命周期内的研发效能。

2 Spock

2.1 Spock是什么?

Spock是一款国外优秀的测试框架，基于BDD（行为驱动开发）思想实现，功能非常强大。Spock结合Groovy动态语言的特点，提供了各种标签，并采用简单、通用、结构化的描述语言，让编写测试代码更加简洁、高效。官方的介绍如下：

What is it? Spock is a testing and specification framework for Java and Groovy applications. What makes it stand out from the crowd is its beautiful and highly expressive specification language. Thanks to its JUnit runner, Spock is compatible with most IDEs, build tools, and continuous integration servers. Spock is inspired from JUnit, RSpec, jMock, Mockito, Groovy, Scala, Vulcans, and other fascinating life forms.

Spock是一个Java和Groovy`应用的测试和规范框架。之所以能够在众多测试框架中脱颖而出，是因为它优美而富有表现力的特定语言。Spock的灵感来自JUnit、RSpec、jMock、Mockito、Groovy、Scala、Vulcans。

简单来讲，Spock主要特点如下：

- 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 ……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。

2.2 和JUnit、jMock有什么区别?

总的来说，JUnit、jMock、Mockito都是相对独立的工具，只是针对不同的业务场景提供特定的解决方案。其中JUnit单纯用于测试，并不提供Mock功能。

我们的服务大部分是分布式微服务架构。服务与服务之间通常都是通过接口的方式进行交互。即使在同一个服务内也会分为多个模块，业务功能需要依赖下游接口的返回数据，才能继续后面的处理流程。这里的下游不限于接口，还包括中间件数据存储比如DB、MQ、配置中心等等，所以如果想要测试自己的代码逻辑，就必须把这些依赖项Mock掉。因为如果下游接口不稳定可能会影响我们代码的测试结果，让下游接口返回指定的结果集（事先准备好的数据），这样才能验证我们的代码是否正确，是否符合逻辑结果的预期。

尽管jMock、Mockito提供了Mock功能，可以把接口等依赖屏蔽掉，但不能对静态方法Mock。虽然PowerMock、jMockit能够提供静态方法的Mock，但它们之间也需要配合（JUnit + Mockito PowerMock）使用，并且语法上比较繁琐。工具多了就会导致不同的人写出的单元测试代码“五花八门”，风格相差较大。

Spock通过提供规范性的描述，定义多种标签（given 、 when 、 then 、 where 等），去描述代码“应该做什么”，“输入条件是什么”，“输出是否符合预期”，从语义层面规范了代码的编写。

Spock自带Mock功能，使用简单方便（也支持扩展、集成其他Mock框架，比如PowerMock、TestableMock），再加上Groovy动态语言的强大语法，能写出简洁高效的测试代码，同时能方便直观地验证业务代码的行为流转，增强工程师对代码执行逻辑的可控性。

2.3 使用Spock解决单元测试开发中的痛点

2.3.1 多分支逻辑

如果在 (if/else) 分支很多的复杂场景下，编写单元测试代码的成本会变得非常高，正常的业务代码可能只有几十行，但为了测试这个功能覆盖大部分的分支场景，编写的测试代码可能远不止几十行。

尽管使用JUnit的 @Parametered 参数化注解或者DataProvider方式可以解决多数据分支问题，但不够直观，而且如果其中某一次分支测试Case出错了，它的报错信息也不够详尽。

这就需要一种编写测试用例高效、可读性强、占用工时少、维护成本低的测试框架。首先不能让业务人员排斥编写单元测试，更不能让工程师觉得写单元测试是在浪费时间。而且使用JUnit做测试工作量不算小。据初步统计，采用JUnit的话，它的测试代码行和业务代码行能到3:1。如果采用Spock作为测试框架的话，它的比例可缩减到1:1，能够大大提高编写测试用例的效率。

下面借用《编程珠玑》中一个计算税金的例子。

```
package com.github.pbetkier.spockdemo.chapter2_3;

import java.math.BigDecimal;

/**
 * 税金计算逻辑
```

```java
 *
 * @author liangche, luoyufeng.lyf@alibaba-inc.com
 * @since 2021/12/18
 */
public class TaxCalculation {

    /**
     * Some codes have many branches for test
     *
     * @param income Income used to calculate taxes
     * @return Taxes
     */
    public static double calc(double income) {
        BigDecimal tax;
        BigDecimal salary = BigDecimal.valueOf(income);
        if (income ⩽ 0) {
            return 0;
        }
        if (income > 0 && income ⩽ 3000) {
            BigDecimal taxLevel = BigDecimal.valueOf(0.03);
            tax = salary.multiply(taxLevel);
        } else if (income > 3000 && income ⩽ 12000) {
            BigDecimal taxLevel = BigDecimal.valueOf(0.1);
            BigDecimal base = BigDecimal.valueOf(210);
            tax = salary.multiply(taxLevel).subtract(base);
        } else if (income > 12000 && income ⩽ 25000) {
            BigDecimal taxLevel = BigDecimal.valueOf(0.2);
            BigDecimal base = BigDecimal.valueOf(1410);
            tax = salary.multiply(taxLevel).subtract(base);
        } else if (income > 25000 && income ⩽ 35000) {
            BigDecimal taxLevel = BigDecimal.valueOf(0.25);
            BigDecimal base = BigDecimal.valueOf(2660);
            tax = salary.multiply(taxLevel).subtract(base);
        } else if (income > 35000 && income ⩽ 55000) {
            BigDecimal taxLevel = BigDecimal.valueOf(0.3);
            BigDecimal base = BigDecimal.valueOf(4410);
            tax = salary.multiply(taxLevel).subtract(base);
        } else if (income > 55000 && income ⩽ 80000) {
            BigDecimal taxLevel = BigDecimal.valueOf(0.35);
            BigDecimal base = BigDecimal.valueOf(7160);
            tax = salary.multiply(taxLevel).subtract(base);
        } else {
            BigDecimal taxLevel = BigDecimal.valueOf(0.45);
            BigDecimal base = BigDecimal.valueOf(15160);
            tax = salary.multiply(taxLevel).subtract(base);
        }
        return tax.setScale(2, BigDecimal.ROUND_HALF_UP).doubleValue();
    }

}
```

能够看到上面的代码中有大量的 `if-else` 语句，Spock提供了where标签，可以让我们通过表格的方式来测试多种分支。

```
package com.github.pbetkier.spockdemo.chapter2_3

import spock.lang.Specification
import spock.lang.Unroll

/**
 * 计算税金 Test
 *
 * @author liangche, luoyufeng.lyf@alibaba-inc.com
 * @since 2021/12/18
 */
class TaxCalculationTest extends Specification {

    /**
     * <code>@Unroll</code> 表示每一行数据为单独的 case
     */
    @Unroll
    def "个税计算,收入:#income, 个税:#result"() {
        expect: "when + then 的组合"
        TaxCalculation.calc(income) == result

        where: "表格方式测试不同的分支逻辑"
        income || result
        -1     || 0
        0      || 0
        2999   || 89.97
        3000   || 90.0
        3001   || 90.1
        11999  || 989.9
        12000  || 990.0
        12001  || 990.2
        24999  || 3589.8
        25000  || 3590.0
        25001  || 3590.25
        34999  || 6089.75
        35000  || 6090.0
        35001  || 6090.3
        54999  || 12089.7
        55000  || 12090
        55001  || 12090.35
        79999  || 20839.65
        80000  || 20840.0
        80001  || 20840.45
    }

}
```

使用Spock写的单元测试代码，语法简洁，表格方式测试覆盖分支场景更加直观，开发效率高，更适合敏捷开发。

2.3.2 单元测试代码的可读性和可维护性

我们微服务场景很多时候需要依赖其他接口返回的结果，才能验证自己的代码逻辑。Mock工具是必不可少的。但jMock、Mockito的语法比较繁琐，再加上单元测试代码不像业务代码那么直观，又不能完全按照业务流程的思路写单元测试，这就让不少同学对单元测试代码可读性不够重视，最终导致测试代码难以阅读，维护起来更是难上加难。甚至很多同学自己写的单元测试，过几天再看也一样觉得“云里雾里”的。也有改了原来的代码逻辑导致单元测试执行失败的；或者新增了分支逻辑，单元测试没有覆盖到的；最终随着业务的快速迭代单元测试代码越来越难以维护。

Spock提供多种语义标签，如：given、when、then、expect、where、with、and等，从行为上规范了单元测试代码，每一种标签对应一种语义，让单元测试代码结构具有层次感，功能模块划分更加清晰，也便于后期的维护。

Spock自带Mock功能，使用上简单方便。我们可以再看一个样例，对于如下的代码逻辑进行单元测试：

```
package com.github.pbetkier.spockdemo.chapter2_3;

import java.util.List;

import com.github.pbetkier.spockdemo.model.StudentDTO;
import com.github.pbetkier.spockdemo.model.StudentVO;
import org.apache.commons.lang3.StringUtils;

/**
 * Test mock ability
 *
 * @author liangche, luoyufeng.lyf@alibaba-inc.com
 * @since 2021/12/19
 */
public class StudentService {

    private StudentDao studentDao;

    /**
     * 1. Find Students
     * <p>2. Set the corresponding properties
     * <p>3. Return VO
     *
     * @param id ID
     * @return StudentVO
     */
    public StudentVO getStudentById(int id) {
        List<StudentDTO> students = studentDao.getStudentInfo();
        StudentDTO studentDTO = students.stream()
            .filter(u -> u.getId() == id)
            .findFirst()
            .orElse(null);

        StudentVO studentVO = new StudentVO();
        if (studentDTO == null) {
            return studentVO;
```

```
        }
        studentVO.setId(studentDTO.getId());
        studentVO.setName(studentDTO.getName());
        if ("上海".equals(studentDTO.getProvince())) {
            studentVO.setPostCode("200000");
        }
        if ("北京".equals(studentDTO.getProvince())) {
            studentVO.setPostCode("100000");
        }

        return studentVO;
    }
}
```

比较明显，左边的JUnit单元测试代码冗余，缺少结构层次，可读性差，随着后续的迭代，势必会导致代码的堆积，维护成本会变得越来越高。右边的单元测试代码Spock会强制要求使用 given 、 when 、 then 这样的语义标签（至少一个），否则编译不通过，这样就能保证代码更加规范，结构模块化，边界范围清晰，可读性强，便于扩展和维护。而且使用了自然语言描述测试步骤，让非技术人员也能看懂测试代码：

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。

每个标签后面的双引号里可以添加描述，说明这块代码的作用（非强制），如 when: "获取信息" 。因为Spock使用Groovy作为单元测试开发语言，所以代码量上比使用Java写的会少很多，比如given模块里通过构造函数的方式创建请求对象。

实际上 StudentDTO.java 这个类并没有3个参数的构造方法，是Groovy帮我们实现的。Groovy默认会提供一个包含所有对象属性的构造方法。而且调用方式上可以指定属性名，类似于key:value的语法，非常人性化，方便在属性多的情况下构造对象，如果使用Java写，可能就要调用很多的 setXxx() 方法，才能完成对象初始化的工作。

```
    and: "mock studentDao返回值"
    studentDao.getStudentInfo() >> [student1, student2]
```

Spock自带的Mock语法也非常简单： dao.getStudentInfo() >> [student1, student2]

当调用studentDao.getStudentInfo()方法时返回一个List。List的创建也很简单，中括号[]即表示List，Groovy会根据方法的返回类型，自动匹配是数组还是List，而List里的对象就是之前given块里构造的user对象，其中 >> 就是指定返回结果，类似Mockito的when().thenReturn()语法，但更简洁一些。

如果要指定返回不同值的话，可以使用3个右箭头 >>>

```
studentDao.getStudentInfo() >>> [[student1,student2],[student3,student4],
[student5,student6]]

studentDao.getStudentInfo() >> [student1,student2] >> [student3,student4] >>
[student5,student6]

2 * studentDao.getStudentInfo() >> [student1,student2] >> [student3,student4]
```

每次调用studentDao.getStudentInfo()方法返回不同的值。

```
// _ 表示匹配任意类型参数
List<StudentDTO> students = studentDao.getStudentInfo(_);

// 如果有同名的方法，使用as指定参数类型区分
List<StudentDTO> students = studentDao.getStudentInfo(_ as String);
```

2.3.3 结果与异常验证

单元测试不仅仅是为了统计代码覆盖率，更重要的是验证业务代码的健壮性、业务逻辑的严谨性以及设计的合理性，在结果校验方面，Spock表现也是十分优异的。

then模块作用是验证被测方法的结果是否正确，符合预期值，所以这个模块里的语句必须是boolean表达式，类似于JUnit的assert断言机制，但不必显示地写assert，这也是一种约定优于配置的思想。then块中使用了Spock的with功能，可以验证返回结果response对象内部的多个属性是否符合预期值，这个相对于JUnit的assertNotNull或assertEquals的方式更简单一些。

```
/**
 * Exception catch and verify ability demo
 *
 * @author liangche, luoyufeng.lyf@alibaba-inc.com
 * @since 2021/12/19
 */
public class StudentService {

  /**
     * 异常测试
     *
```

```
     * @param student studentVo
     * @throws BusinessException 业务异常
     */
    public void validateStudent(StudentVO student) throws BusinessException {
        if (null == student) {
            throw new BusinessException("10001", "student is null");
        }
        if (null == student.getId()) {
            throw new BusinessException("10002", "student id is null");
        }
        if (StringUtils.isBlank(student.getName())) {
            throw new BusinessException("10003", "student name is null");
        }
        if (StringUtils.isBlank(student.getPostCode())) {
            throw new BusinessException("10004", "student postCode is null");
        }
    }
}
```

在then标签里用到了Spock的thrown()方法，这个方法可以捕获我们要测试的业务代码里抛出的异常。thrown()方法的入参expectedException，是我们自己定义的异常变量，这个变量放在where标签里就可以实现验证多种异常情况的功能（Intellij Idea格式化快捷键，可以自动对齐表格）。expectedException类型调用validateUser方法里定义的BusinessException异常，可以验证它所有的属性，code、message是否符合预期值。

2.3.4 强大的 Where

下面的业务代码有2个if判断，是对邮编处理逻辑：

```
if ("上海".equals(studentDTO.getProvince())) {
    studentVO.setPostCode("200000");
}
if ("北京".equals(studentDTO.getProvince())) {
    studentVO.setPostCode("100000");
}
```

如果要完全覆盖这2个分支就需要构造不同的请求参数，多次调用被测试方法才能走到不同的分支。在前面，我们介绍了Spock的where标签可以很方便的实现这种功能，代码如下所示：

where模块第一行代码是表格的列名，多个列使用|单竖线隔开，||双竖线区分输入和输出变量，即左边是输入值，右边是输出值。格式如下：

输入参数1 | 输入参数2 || 输出结果1 | 输出结果2

表格的每一行代表一个测试用例，即被测方法执行了2次，每次的输入和输出都不一样，刚好可以覆盖全部分支情况。这个就是where+with的用法，更符合我们实际测试的场景，既能覆盖多种分支，又可以对复杂对象的属性进行验证，其中在定义的测试方法名，使用了Groovy的字面值特性：

即把请求参数值和返回结果值的字符串动态替换掉，#id、#province、#postCodeResult#号后面的变量是在方法内部定义的，实现占位符的功能。

而且如果其中某行测试结果不对，Spock的错误提示信息也很详细，方便进行排查（比如我们把第1条测试用例返回的邮编改成200001）：

3 TestableMock

Spock提供的Mock功能优雅且简单，但对Java代码支持不完整，无法Mock私有、静态等方法，因此采用TestableMock此工具

3.1 主流Mock工具对比

工具	原理	最小Mock单元	对被Mock方法的限制	上手难度	IDE支持
Mockito	动态代理	类	不能Mock私有/静态和构造方法	**较容易**	**很好**
Spock	动态代理	类	不能Mock私有/静态和构造方法	较复杂	一般
PowerMock	自定义类加载器	类	**任何方法皆可**	较复杂	**较好**
JMockit	运行时字节码修改	类	不能Mock构造方法(new操作符)	较复杂	一般
TestableMock	运行时字节码修改	方法	**任何方法皆可**	**很容易**	一般

3.2 TestableMock简介

单元测试中的Mock方法，通常是为了绕开那些依赖外部资源或无关功能的方法调用，使得测试重点能够集中在需要验证和保障的代码逻辑上。

在定义Mock方法时，开发者真正关心的只有一件事："这个调用，在测试的时候要换成那个假的Mock方法"。

当下主流的Mock框架在实现Mock功能时，需要开发者操心的事情实在太多：Mock框架如何初始化、与所用的服务框架是否兼容、要被Mock的方法是不是私有的、是不是静态的、被Mock对象是new出来的还是注入的、怎样把被测对象送回被测类里...这些非关键的额外工作极大分散了使用Mock工具应有的乐趣。

于是，我们开发了TestableMock，**一款特立独行的轻量Mock工具**。

3.3 TestableMock思想

3.4 TestableMock实践

3.4.1 典型注解

`@MockInvoke`

将当前方法标识为待匹配的Mock成员方法。

- 作用于：Mock容器类中的方法

参数	类型	是否必须	默认值	作用
targetClass	Class	否	N/A	指定Mock目标的调用者类型
targetMethod	String	否	N/A	指定Mock目标的方法名
scope	MockScope	否	MockScope.GLOBAL	指定Mock的生效范围

`@MockNew`

将当前方法标识为待匹配的Mock构造方法。

- 作用于：Mock容器类中的方法

参数	类型	是否必须	默认值	作用
scope	MockScope	否	MockScope.GLOBAL	指定Mock的生效范围

3.4.2 QuickStart

3.4.3 Spock + TestableMock

```
package com.github.pbetkier.spockdemo.chapter2_3

import com.alibaba.testable.core.annotation.MockInvoke
import com.github.pbetkier.spockdemo.model.StudentDTO
import com.github.pbetkier.spockdemo.model.StudentVO
import spock.lang.Specification

/**
 * StudentService test demo
 *
 * @author liangche, luoyufeng.lyf@alibaba-inc.com
 * @since 2021/12/19
 */
class StudentServiceTest extends Specification {

    static def studentDTOList

    /**
     * TestableMock Container
     */
    static class Mock {

        @MockInvoke(targetClass = StudentDao.class)
        List<StudentDTO> getStudentInfo() {
            return studentDTOList
        }

    }

    /**
     * Spock + TestableMock
     */
    def "input 学生id:#id, 返回的邮政编码:#postCodeResult"() {
        given: "被测试类初始化"
        studentService = new StudentService()
```

```
        and: "mock data"
        studentDTOList = student

        when: "方法调用"
        def studentVo = studentService.getStudentById(id)

        then: "验证结果"
        studentVo.getPostCode() == postCodeResult

        where: "数据构造"
        id | student                                                  || postCodeResult
        1  | [new StudentDTO(id: 1, name: "test1", province: "上海")] || "200000"
        2  | [new StudentDTO(id: 2, name: "test2", province: "北京")] || "100000"
    }

}
```

4 数据层（DAO）测试

在实际场景中，DAO对数据库的操作依赖于Mybatis的sql mapper 文件或者基于MyBatis Plus的动态sql，在单测中验证所有sql 逻辑的正确性非常重要，在DAO层有足够的覆盖度和强度后，Service层的单测才能仅仅关注自身的业务逻辑。

经分析，其有以下两点核心问题：

1. 如何构造最小测试环境?
2. 如何避免脏数据对实际数据库的影响?

基于此，我们采用H2内存数据库及Spring而非SpringBoot自动装配，解决DAO层测试问题

4.1 依赖引入

详见代码

4.2 准备初始化SQL

在测试资源目录 src/test/resource 下新建 db/{your_module}.sql ，其中的内容是需要初始化的建表语句，也可以包括部分初始记录的dml语句，可使用DMS导出。如果表结构发生了更改，需要人工重新导出。

4.3 测试环境与测试类

```
package com.github.pbetkier.spockdemo.chapter

import com.github.pbetkier.spockdemo.chapter.dao.StudentMapper
import com.github.pbetkier.spockdemo.chapter.pojo.StudentDO
import com.github.pbetkier.spockdemo.config.DaoTestConfiguration
```

```groovy
import org.springframework.test.context.ContextConfiguration
import spock.lang.Specification
import spock.lang.Title

import javax.annotation.Resource

/**
 * 数据库测试
 * <pre>
 *     1. 引入测试环境上下文
 *     2. 正常流程
 * </pre>
 * @author liangche, luoyufeng.lyf@alibaba-inc.com
 * @since 2021/12/19
 */
@Title("Mapper 测试")
@ContextConfiguration(classes = DaoTestConfiguration.class)
class StudentMapperTest extends Specification {

    @Resource
    private StudentMapper studentMapper

    def "Mapper test"() {
        given: "数据初始化"
        def studentDO = new StudentDO(id: 1, name: "test")
        studentMapper.insert(studentDO)

        when: "查询数据库"
        def result = studentMapper.select(1)

        then: "验证结果"
        result.id == 1
        result.name == "test"
    }

}
```

`@ContextConfiguration(classes = [DaoTestConfiguration.class])`这个注解很关键，他是spring-test模块的注解，通过这个注解可以配置spring容器在启动时的上下文。

```groovy
package com.github.pbetkier.spockdemo.config;

import javax.sql.DataSource;

import org.apache.ibatis.session.SqlSessionFactory;
import org.mybatis.spring.SqlSessionFactoryBean;
import org.mybatis.spring.SqlSessionTemplate;
import org.mybatis.spring.annotation.MapperScan;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
```

```
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.Resource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

@Configuration
@MapperScan({"com.github.pbetkier.spockdemo.chapter.dao"})
public class DaoTestConfiguration {

    @Value("classpath:mybatis-config.xml")
    private Resource configLocation;

    @Bean
    public DataSource dataSource() {
        EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
        return builder.setType(EmbeddedDatabaseType.H2).addScript("classpath:db/student-
h2.sql").build();
    }

    @Bean
    public JdbcTemplate jdbcTemplate() {
        JdbcTemplate jdbcTemplate = new JdbcTemplate();
        jdbcTemplate.setDataSource(dataSource());
        return jdbcTemplate;
    }

    @Bean
    public SqlSessionFactory sqlSessionFactory() throws Exception {
        SqlSessionFactoryBean sqlSessionFactoryBean = new SqlSessionFactoryBean();
        sqlSessionFactoryBean.setConfigLocation(configLocation);
        sqlSessionFactoryBean.setDataSource(dataSource());

        return sqlSessionFactoryBean.getObject();
    }

    @Bean
    public SqlSessionTemplate sqlSessionTemplate() throws Exception {
        return new SqlSessionTemplate(sqlSessionFactory());
    }

}
```

5 覆盖率统计

Jacoco是统计单元测试覆盖率的一种工具，当然Spock也自带了覆盖率统计的功能，这里使用第三方Jacoco的原因主要是国内公司使用的比较多一些。在pom文件里引用Jacoco的插件：jacoco-maven-plugin，然后执行mvn package 命令，成功后会在target目录下生成单元测试覆盖率的报告，点开报告找到对应的被测试类查看覆盖情况。

####