

22 单元测试之单测举例

更新时间：2020-06-03 15:45:41



一个人追求的目标越高，他的才力就发展得越快，对社会就越有益。——高尔基

1. 前言

前面我们讲到了构造单元测试数据的几种方式，接下来我们将讲述如何编写单元测试。

《手册》第 29 页有对数据库单元测试的规定 1：

【推荐】和数据库相关的单元测试，可以设定自动回滚机制，不给数据库造成脏数据。或者对单元测试产生的数据有明确的前后缀标识。

那么单元测试还有哪些注意事项，除了数据库相关的单元测试外，其它的单元测试又该如何去写呢？

2. 对哪些代码写单测？

实际开发中，主要对数据访问层、服务层和工具类进行单元测试。

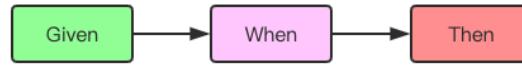
正如前言中所说，**数据库相关的单元测试**，一般要设置自动回滚。除此之外，还可以整合 H2 等内存数据库来对数据访问层代码进行测试。

工具类的单元测试也非常 important，因为工具类一般在服务内共用，如果有 **BUG**，影响面很大，很容易造成线上问题或故障。一般需要构造正常和边界值两种类型的用例，对工具类进行全面的测试，才可放心使用。此时结合注释小节所讲的内容，需将典型的调用和结果添加到注释上，方便函数的使用者。

服务层的单元测试，一般要依赖 `mock` 工具，将服务的所有依赖都 `mock` 掉。其本质是“控制变量法”，将原本依赖的 N 个“变量”都变为“常量”，只观察所要测试的服务逻辑是否正确。

3. 单元测试的结构

大家一定要牢记编写单元测试的核心逻辑，其结构如下：



典型的单元测试可分为三个阶段，分别为准备、执行和验证 [2](#)。

准备阶段（Given） 主要负责创建测试数据、构造 `mock` 方法的返回值，准备环节的编码是单元测试最复杂的部分。需要注意的是 Mockito 库中以 `when` 开头的函数其实是在准备阶段。

执行阶段（When） 一般只是调用测试的函数，此部分代码通常较短。

验证阶段（Then） 通常验证测试函数的执行的结果、准备阶段 `mock` 函数的调用次数等是否符合预期。

4. 单元测试方法命名

早期必须在单元测试函数命名前加入‘`test`’前缀。现在已经不推荐这么使用，一般采用驼峰。

也会有很多人会将太多描述放到测试函数命名中，这也不太推荐，此种情况应该放到函数的注释中。

推荐的命名格式如：`shouldReturnItemNameInUpperCase()`。

5. 单元测试举例

数据访问层测试，只不过是将正常的环境加入了回滚或者采用内存 / 内嵌数据库，难度不大，这里就不给出具体范例。本文将重点讲述工具类的测试和服务层的测试。

5.1 工具类的测试

学习工具类的单元测试，强烈推荐大家参考 [guava](#)、[commons-lang3](#)、[commons-collection4](#) 这三个知名开源工具类项目的单元测试代码。

如 `commons-lang3` 包的 `StringUtils#contains` 源码：

```

// Contains
//-----
/** 
 * <p>Checks if CharSequence contains a search character, handling {@code null}.
 * This method uses {@link String#indexOf(int)} if possible.</p>
 *
 * <p>A {@code null} or empty ("") CharSequence will return {@code false}.</p>
 *
 * <pre>
 * StringUtils.contains(null, *)  = false
 * StringUtils.contains("", *)    = false
 * StringUtils.contains("abc", 'a') = true
 * StringUtils.contains("abc", 'z') = false
 * </pre>
 *
 * @param seq  the CharSequence to check, may be null
 * @param searchChar  the character to find
 * @return true if the CharSequence contains the search character,
 * false if not or {@code null} string input
 * @since 2.0
 * @since 3.0 Changed signature from contains(String, int) to contains(CharSequence, int)
 */
public static boolean contains(final CharSequence seq, final int searchChar) {
    if (isEmpty(seq)) {
        return false;
    }
    return CharSequenceUtils.indexOf(seq, searchChar, 0) >= 0;
}

```

对应的单元测试代码如下：

```

@Test
public void testContains_Char() {
    // 不符合条件的特殊用例
    assertFalse(StringUtils.contains(null, ''));
    assertFalse(StringUtils.contains("", ''));
    assertFalse(StringUtils.contains("", null));
    assertFalse(StringUtils.contains(null, null));
    // 符合条件的用例
    assertTrue(StringUtils.contains("abc", 'a'));
    assertTrue(StringUtils.contains("abc", 'b'));
    assertTrue(StringUtils.contains("abc", 'c'));
    // 不符合条件的正常用例
    assertFalse(StringUtils.contains("abc", 'z'));
}

```

我们可看到，测试时除了选择符合条件的用例外，还要选择不符合条件的用例。其中不符合条件的用例可以还包括常规的用例和特殊用例（边界条件）。

再如 guava 的 `StopWatch#stop` :

```

/**
 * Stops the stopwatch. Future reads will return the fixed duration that had elapsed up to this
 * point.
 *
 * @return this {@code Stopwatch} instance
 * @throws IllegalStateException if the stopwatch is already stopped.
 */
@CanIgnoreReturnValue
public Stopwatch stop() {
    long tick = ticker.read();
    checkState(isRunning, "This stopwatch is already stopped.");
    isRunning = false;
    elapsedNanos += tick - startTick;
    return this;
}

```

根据源码我们可知，调用该函数后 `isRunning` 会被设置为 `false`，如果重复调用会抛出 `IllegalStateException`，

因此，我们要测试已经停止后再次调用停止函数会的效果。

验证调用该函数后 `isRunning` 的确会被设置为 `false`，如果重复调用会抛出 `IllegalStateException`，因此该函数的单元测试源码如下（注意该测试函数命名）：

```

public void testStop_alreadyStopped() {
    stopwatch.start();
    stopwatch.stop();
    try {
        stopwatch.stop();
        fail();
    } catch (IllegalStateException expected) {
    }
    assertFalse(stopwatch.isRunning());
}

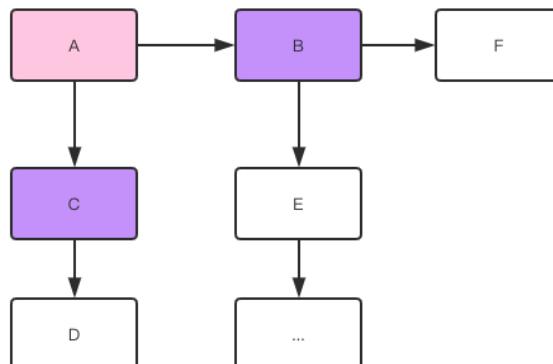
```

5.2 服务层的测试

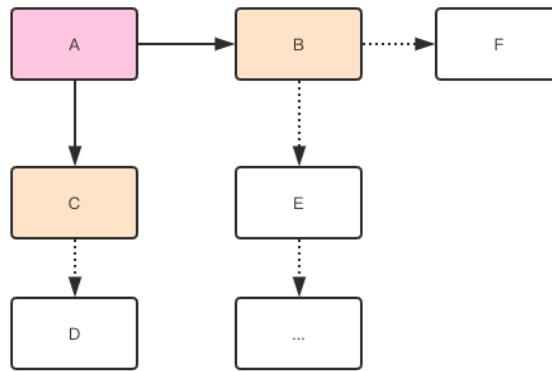
服务层的测试一般将底层的所有依赖都 `mock` 掉，最常用的框架为 `Mockito`、`JMockit`、`Easy Mock`。

本小节的示例采用的是 `Mockito`。

核心场景如：`A` 类的某函数依赖 `B` 类的某函数和 `C` 类的某函数，而 `B` 类又依赖 `E` 类和 `F` 类，`C` 类又依赖 `D` 类，等等。



如果要测试 **A** 类的某个函数，则需要 **mock B** 类 和 **C** 类的对象。测试者可以指定 **B** 的某个函数接受某个参数返回固定的结果，指定 **C** 接受特定参数，返回特定结果，然后调用 **A** 的对应函数，验证 **A** 的返回值是否符合期待。



注：为了简化，此处并没有采用标准的类图方式作图。

此时有些朋友可能会有一个疑问，为什么不 **mock D**、**E** 和 **F** 等其它类呢？

其实这就是本专栏特别强调学习时要重视“是什么”的原因。单元测试从思想上来讲就是“控制变量法”，即将依赖变为“常量”，只有待测试的函数参数是“变量”，通过输入参数推测出结果，和实际的结果去对比，才可以更好地验证其正确性。

因此，我们只需要把它的直接依赖变成“常量”即可，其它的依赖 **mock** 没有意义。

另外，大家一定要注意单元测试和集成测试的区别，不要将单元测试和集成测试混在一起。

下面给出一个简单示例：

待测试的服务接口：

```
public interface ItemService {  
    String getItemNameUpperCase(String itemId);  
}
```

待测试的服务的实现类：

```
@Service  
public class ItemServiceImpl implements ItemService {  
  
    @Resource  
    private ItemRepository itemRepository;  
  
    @Override  
    public String getItemNameUpperCase(String itemId) {  
  
        Item item = itemRepository.findById(itemId);  
  
        if (item == null) {  
            return null;  
        }  
        return item.getName().toUpperCase();  
    }  
}
```

可见该服务依赖数据访问组件 **ItemRepository**。

根据前面的单元测试的结构和命名建议，我们对该函数编写单元测试代码：

```
import org.junit.Before;
import org.junit.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.Mockito.*;

public class ItemServiceTest {

    @Mock
    private ItemRepository itemRepository;

    @InjectMocks
    private ItemServiceImpl itemService;

    @Before
    public void setUp(){
        MockitoAnnotations.initMocks(this);
    }

    /**
     * 如果从存储层查询到一个item，那么它的 name 将被转化为大写。
     */
    @Test
    public void shouldReturnItemNameInUpperCase() {

        // Given
        Item mockedItem = new Item("it1", "Item 1", "This is item 1", 2000, true);
        when(itemRepository.findById("it1")).thenReturn(mockedItem);

        // When
        String result = itemService.getItemNameUpperCase("it1");

        // Then
        verify(itemRepository, times(1)).findById("it1");
        assertThat(result).isEqualTo("ITEM 1");
    }
}
```

测试函数采用驼峰命名并且体现出了该测试函数的核心含义。

可以看出在准备阶段，构造测试对象（数据）并 `mock` 掉底层依赖；在执行阶段直接调用待测试的函数；在验证阶段对结果进行断言。

Mockito 的更多高级用法请参考[官方网站](#)和[框架配套 wiki](#)。如果需要 `mock` 静态方法、私有函数等，可以学习[PowerMock](#)，拉取其源码通过学习单元测试来快速掌握其用法。

5. 总结

本节主要给出单元测试在实际编程中的运用，给出了单元测试的结构、命名建议以及使用范例。希望大家在实际编程中能够举一反三，灵活运用，通过单元测试提高编码的质量。

下一节将给出 Java 学习宝典。

6. 课后作业

- 拉取 PowerMock 的源码，通过源码的单元测试来学习如何 `mock` 私有函数；

- 使用 easy-random 代替 shouldReturnItemNameInUpperCase () 函数构造测试数据部分的代码。

参考资料

阿里巴巴与 Java 社区开发者. 《Java 开发手册 1.5.0》华山版.2019 [□□](#)

[《Given-When-Then in Junit Tests》 \[□□\]\(#\)](#)

}

[◀ 21 单元测试构造数据的正确姿势](#)

23 Java学习宝典

[▶](#)