应用设计模式编写易于单元测试的代码

熊 伟

来源IBM网站

2008-07-24

单元测试是软件开发的一个重要组成部分，通过在软件设计、开发的过程中合理地运用设计模式，不但为系统重构、功能扩展及代码维护提供了方便，同时也为单元测试的实施提供了极大的灵活性，可以有效降低单元测试编码的难度，更好地保证软件开发的质量。

引言

设计模式是对被用来在特定场景下解决一般设计问题的类和相互通信的对象的描述，通过在系统设计中引入合适的设计模式可以为系统实现提供更大的灵活性，从而有效地控制变化，更好地应对需求变更或者按需变更系统运行路径等问题。

单元测试是软件开发的一个重要组成部分，是与编码实现同步进行的开发活动，这一点已成为软件开发者的共识。适度的单元测试不但不会影响开发进度，反而可以为开发过程提供很好的控制，为软件质量、系统重构等提供有力的保障，并且，当后续系统需求发生变更、Bug Fix 或功能扩展时，能很好地保证已有实现不会遭到破坏，从而使得程序更易于维护和修改。 Martin Fowler、Kent Beck、Robert Martin 等软件设计领域泰斗更是极力倡导测试先行的测试驱动开发（Test Driven Development，TDD）的开发方式。

> 请访问 Java 设计模式专题，查看更多关于 Java 设计模式的文章和教程。

单元测试主要用于测试细粒度的程序单元，如类的某个复杂方法的正确性，也可以根据需要综合测试某个操作所涉及的多个相互联系的类的正确性。在很多情况下，相互联系的多个类中有些类比较简单，为这些简单类单独编写单元测试用例往往不如将它们与使用它们的类一起进行测试有意义。

模拟对象（Mock Objects）是为模拟被测试单元所使用的外围对象、设备（后文统一简称为外部对象）而设计的一种特殊对象，它们具有与外部对象相同的接口，但实现往往比较简单，可以根据测试的场景进行定制。由于单元测试不是系统测试，方便、快速地被执行是单元测试的一个基本要求，直接使用外部对象往往需要经过复杂的系统配置，并且容易出现与欲测试功能无关的问题；对于一些异常的场景，直接使用外部对象可能难以构造，而通过设计合适的 Mock Objects，则可以方便地模拟需要的场景，从而为单元测试的顺利执行提供有效的支持。

本文根据笔者经验，介绍了几种典型的设计模式在系统设计中的应用，及由此为编写单元测试带来的方便。

从对象创建开始

由于需要使用 Mock Objects 来模拟外部对象的功能，因此必须修改正常的程序流程，使得被测试功能模块与 Mock Objects，而不是外部对象进行交互。要做到这一点，首先要解决的问题就是对象创建，即在原本创建外部对象的地方创建 Mock Objects，因此在设计、实现业务逻辑时需要注意从业务逻辑中分离出对象创建逻辑。

Factory Method 是一种被普遍运用的创建型模式，用于将对象创建的职责分离到独立的方法中，并通过子类化来实现创建不同对象的目的。如果被测试单元所使用的外部对象是通过 Factory Method 创建的，则可以通过从已有被测试的 Factory 类派生出一个新的 MockFactory，以创建 Mock Objects，并在 setUp 测试中创建 MockFactory，从而间接达到对被测试类进行测试的目的。下面的代码片段展示了具体的做法：

> 关于 setUp
> setUp 是 JUnit 基础类 TestCase 的一个重要方法，每个单元测试在被执行前会调用 setUp 方法做一些必要的预处理，如准备好一些公共的基本输入或创建所需的外部对象。

```
// BaseObjects.java
package com.factorymethod.demo;
public interface BaseObjects {
    voidfunc();
}

// OuterObjects.java
package com.factorymethod.demo;
public class OuterObjects implements BaseObjects {
    public void func() {
        System.out.println('OuterObjects.func');
    }
}
```

热点专题

- 2008--嵌入式技术创新及应用高峰论坛
- 2008飞思卡尔技术论坛
- Altera公司SOPC World 2008专题报道
- 第十届高交会电子展
- 科技闪耀北京奥运
- ADLINK DAY—2008年量测与自动化技术国际高峰研讨会
- 中国电子学会Xilinx杯开放源码硬件创新大赛
- 赛灵思公司Virtex-5系列FPGA
- 3G知识
- IPTV
- 触摸屏技术
- RoHS

杂志精华

- 基于CC2430的无线传感器...
- 无线传感器网络应用系统综述
- 无线传感器网络在野外测量中的...
- 基于竞争的无线传感器网络
- 用于矿井环境监测的无线传感器...
- 具有自适应通信能力的无线传感...
- 基于传感器网络技术的深孔测径...
- 基于无线传感器网络的家庭安防...
- 基于ATmega128L与C...
- 无线传感器网络中移动节点设备...

```
// LogicToBeTested.java, code to be tested
package com.factorymethod.demo;
public class LogicToBeTested {
    public void doSomething() {
        BaseObjects b = createBase();
        b.func();
    }

    public BaseObjects createBase() {
        return newOuterObjects();
    }
}
```

以下则是对应的 MockOuterObjects、MockFactory 以及单元测试的实现：

```
// MockOuterObjects.java
package com.factorymethod.demo;
public class MockOuterObjects implements BaseObjects {
    public void func() {
        System.out.println('MockOuterObjects.func');
    }
}

// MockLogicToBeTested.java
package com.factorymethod.demo;
public class MockLogicToBeTested extends LogicToBeTested {
    public BaseObjects createBase() {
        return new MockOutterObjects();
    }
}

// LogicTest.java
package com.factorymethod.demo;
import junit.framework.TestCase;

public class  LogicTest extends TestCase {
    LogicToBeTested c;
    protected void setUp() {
        c =new MockLogicToBeTested();
    }
    public void testDoSomething() {
        c.doSomething();
    }
}
```

Abstract Factory 是另一种被普遍运用的创建型模式，Abstract Factory 通过专门的 Factory Class 来封装对象创建的职责，并通过实现 Abstract Factory 来完成不同的创建逻辑。如果被测试单元所使用的外部对象是通过 Abstract Factory 创建的，则实现一个新的 Concrete Factory，并在此 Factory 中创建 Mock Objects 是一个比较好的解决办法。对于 Factory 本身，则可以在 setUp 测试的时候指定新的 Concrete Factory ；此外，借助依赖注入框架（如 Spring 的 BeanFactory），通过依赖注入的方式将 Factory 注入也是一种不错的解决方案。对于简单的依赖注入需求，可以考虑实现一个应用专有的依赖注入模块，或者实现一个简单的实现加载器，即根据配置文件载入相应的实现，从而无需修改应用代码，仅通过修改配置文件即可载入不同的实现，进而方便地修改程序的运行路径，执行单元测试。

下面的代码实现了一个简单的 InstanceFactory：

```
// refer to http://www.opensc-project.org/opensc-java/export/100/trunk/
// pkcs15/src/main/java/org/opensc/pkcs15/asn1/InstanceFactory.java
packagecom.instancefactory.demo;

importjava.lang.reflect.InvocationTargetException;
importjava.lang.reflect.Method;
importjava.lang.reflect.Modifier;

public class InstanceFactory {
    private final Method getInstanceMethod;
```

```
    public InstanceFactory(String type) {
        Class clazz =null;
        try {
            clazz = Class.forName(type);
            this.getInstanceMethod = clazz.getMethod('getInstance');
            if(!Modifier.isStatic(this.getInstanceMethod.getModifiers())
            || !Modifier.isPublic(this.getInstanceMethod.getModifiers()))
                throw new IllegalArgumentException(
                    'Method [' + clazz.getName()
                    + '.getInstance(Object)] is not static public.');
        } catch (NoSuchMethodException e) {
            throw new IllegalArgumentException(
                'Class [' + clazz.getName()
                + '] has no static getInstance(Object) method.', e);
        } catch (ClassNotFoundException e) {
            throw new IllegalArgumentException('Class [' + type + '] is not found');
        }
    }

    public Object getInstance() {
        try{
            return this.getInstanceMethod.invoke(null);
        } catch (InvocationTargetException e) {
            if( e.getCause() instanceof RuntimeException )
                throw (RuntimeException)e.getCause();
            throw new IllegalArgumentException(
                    'Method [' +this.getInstanceMethod
                    + '] has thrown an checked exception.', e);
        } catch( IllegalAccessException e) {
            throw new IllegalArgumentException(
                    'Illegal access to method ['
                    +this.getInstanceMethod + '].', e);
        }
    }

    public Method getGetInstanceMethod() {
        return this.getInstanceMethod;
    }
}
```

以下代码演示了 InstanceFactory 的简单使用：

```
// BaseObjects.java
package com.instancefactory.demo;

public interface BaseObjects {
    voidfunc();
}

 // OuterObjects.java

package com.instancefactory.demo;

public class OuterObjects implements BaseObjects {
    public static BaseObjects getInstance() {
        return new OuterObjects();
    }

    public void func() {
        System.out.println('OuterObjects.func');
    }
}

// MockOuterObjects.java
```

```
package com.instancefactory.demo;
public class MockOuterObjects implements BaseObjects {
    public static BaseObjects getInstance() {
        return new MockOuterObjects();
    }

    public void func() {
        System.out.println('MockOuterObjects.func');
    }
 }

// LogicToBeTested.java
packagecom.instancefactory.demo;
public class LogicToBeTested {
    public static final String PROPERTY_KEY= 'BaseObjects';
    public void doSomething() {
        // load configuration file and read the implementation class name of BaseObjects
        // read it from properties to simplify the demo
        // actually, the property file reader can be implemented by InstanceFactory
        String impl = System.getProperty(PROPERTY_KEY);
        InstanceFactory factory = new InstanceFactory(impl);
        BaseObjects b = (BaseObjects)factory.getInstance();
        b.doSomething();
    }
 }

// LogicTest.java
packagecom.instancefactory.demo;
importjunit.framework.TestCase;
public class LogicTest extends TestCase {
    LogicToBeTested c;
    protected void setUp() {
        // set the property file of class map to a file for MockObjects, omitted
        // use System.setProperty to simplify the demo
        System.setProperty(LogicToBeTested.PROPERTY_KEY,
                'com.instancefactory.demo.MockOuterObjects');
        c = new LogicToBeTested();
    }

    public void testDoSomething() {
        c.doSomething();
    }
 }
```

替换实现

通过 Factory Method 替换被创建对象可以满足一些修改程序运行路径的需求，但是，这种方法以子类化为前提，具有很强的侵入性，并且在编写单元测试时，开发人员需要同时负责 Mock Objects 的开发，供 Factory Method 调用，因此，编码量往往会比较大，单元测试开发人员也需对所使用的公共模块的内部结构有十分清楚的认识。即使可以使用公共的 Mock Objects 实现避免代码重复，往往也需要修改业务逻辑中公共服务相关对象的创建代码，这一点对于应用公共模块的业务逻辑的单元测试可能不太适合。

在笔者曾参与设计、开发的某应用系统中，有一个专门的数据库缓冲（Cache）公共服务，该 Cache 负责完成与数据库交互，实现数据的存取，并缓存数据以提高后续访问的效率。对于涉及数据库缓冲的业务逻辑的单元测试，需要一个替代方案来替代已有的数据库缓冲，以避免直接访问实际数据库，但又要保证这个替换不会影响到被测试单元的实现。

为了解决这个问题，我们并没有直接替换 Cache 创建处的代码，因为这些代码遍布在业务代码中，直接替换 Cache 创建代码无疑会侵入业务逻辑，并需要大量使用子类化。为了尽可能降低对业务逻辑的影响，我们维持了原有 CacheFactory 的接口，但是将 CacheFactory 的实现委托（Delegate）给另一个实现类完成，以下是 CacheFactory 实现的伪代码：

```
package com.cachefactory.demo;
public abstract class CacheFactory {
    private static CacheFactoryinstance = new DelegateCacheFactory();
    private static CacheFactorydelegate;
    protected CacheFactory() {
    }

    // CacheFactory is a singletonpublic
    static CacheFactory getInstance() {
```

```
        return instance;
    }

    // the implementation can be changedprotected
    static void setDelegate(CacheFactory instance) {
        delegate= instance;
    }

    public abstract Cache getCache(Object... args);

    // redirect all request to delegateeprivate
    static class DelegateCacheFactoryextendsCacheFactory {
        private DelegateCacheFactory() {
        }

        public Cache getCache(Object... args) {
            return delegate.getCache(args);
        }
    }
}
```

与 CacheFactoryImpl 类似地，我们实现了一个 MockCacheFactory，但与 CacheFactoryImpl 不同的是，这个 MockCacheFactory 所创建的 MockCache 对象虽然与真正的 Cache 实现了相同的接口，但是，它的内部实现却是基于 HashMap 的，因此，可以很好地满足单元测试快速、方便地运行的需要。

单元测试时，只需要在 setUp 时调用执行如下操作：

```
setDelegate(new MockCacheFactory());
```

将 CacheFactory 的实现委托给 MockCacheFactory 即可，所有业务逻辑都无需作任何修改，因此，这种替换实现的方式几乎是没有侵入性的。

这种通过将实现分离到专门的实现类中的做法其实是 Bridge 模式的一个应用，通过使用 Bridge 模式，为替换实现保留了接口，从而使得在不对业务逻辑作任何修改的情况下可以轻松替换公共服务的实现。

除此之外，Strategy 模式也是一种替换实现的有效途径，这种方式与 Factory Method 类似，通过子类化实现新的 Strategy 以替换业务逻辑使用的旧的 Strategy，通过与 Factory Method 或 Bridge 等模式联合使用，在编写应用公共服务的业务逻辑的单元测试时也十分有用。

绕过部分实现

绕过部分实现进行单元测试在大多数情况下是不可取的，因为这种做法极有可能会影响单元测试的质量。但是对于一些特殊的情况，我们可以“冒险”使用这种方式，比如有这样的一个场景：所有请求需经过多级认证，且部分认证处理需要访问数据库，认证结束后为请求分配相应的 sessionId，请求在获得 sessionId 后继续进行进一步的业务逻辑处理。

在保证多级认证模块已被专门的单元测试覆盖的情况下，我们在为业务逻辑编写单元测试的过程中可以考虑跳过多级认证授权模块（对于部分特权用户，也应跳过部分检查），直接为其分配一个 Mock 的 sessionId，以进行后续处理。

对于多级认证问题本身，我们可以考虑采用 Chain of Responsibility 模式将不同的认证逻辑封装到不同的 RequestHandler 中，并通过编码或者根据配置，将所有的 Handler 串联成 Responsibility Chain ；而在单元测试过程中，可以修改 Handler 的串联方式，绕过部分不希望在单元测试中经过的 Handler，从而简化单元测试的运行。

对于这个问题，笔者并不同意为了单元测试的需要去采用 Chain of Responsibility 模式，实际上，上面所阐述的多级认证问题本身比较适合采用这种模式来解决，能够根据需要绕过部分实现，只是应用这种模式的情况下进行单元测试的一种可以考虑的测试途径。

总结

单元测试是软件开发的重要组成部分，而应用 Mock Object 是进行单元测试一种普遍而有效的方式，通过在软件设计、开发的过程中合理地运用设计模式，不但为系统重构、功能扩展及代码维护提供了方便，同时也为单元测试的实施提供了极大的灵活性，可以有效降低单元测试编码的难度，方便地在单元测试中引入 Mock Objects，达到对被测试目标进行单元测试的目的，从而更好地保证软件开发的质量。

参考资料

学习设计模式，请阅读关于设计模式的经典图书：“设计模式：可复用面向对象软件的基础”。

关于单元测试的更多信息，请访问：“JUnit 站点”及“TestNG 站点”。

“使用模仿对象进行单元测试”（developerWorks，2003 年 3 月）：介绍如何使用模仿对象替换合作者以改进单元测试。

“Java 设计模式专题”：查看更多关于 Java 设计模式的文章和教程。

追求代码质量 系列（Andrew Glover，developerWorks）：学习更多关于编写专注于质量的代码的信息。

Java 技术专区：数百篇关于 Java 编程各个方面的文章。

关于作者

熊伟（Wayne Xiong），华中科技大学硕士，曾用网名 Bill David、大卫、大笨熊等。精于 C++，后转入 JAVA 阵营，曾就职于 Lucent、BEA（Oracle）等公司，从事电信及 J2EE 应用平台的设计开发；现为 Adobe 公司高级软件工程师，主要从事 Flash Media Server 及 RIA 相关应用的设计开发。可以通过 billdavidcn@hotmail.com 或博客 http://blog.csdn.net/billdavid 与他联系。

在线联系

添加到收藏夹

关于“应用设计模式编写易于单元测试的代码”，我有如下需求或意向：

用户名： 密码： 验证码： 5829 欢迎注册 提 交

相关应用

- Java 下实现锁无关数据结构
- 一种Java字节码静态调整的方法及应用

地址：北京海淀区清华东路25号电子六所大厦
联系电话：82306084 / 82306085 传真：62311179 京ICP备05053646号
推荐分辨率1024*768 IE6.0版本