junit设计理念与工作原理(转自selfishman 的博客) [1]

关键词： junit [2] 设计模式 [3] 单元测试 [4]

junit设计理念与工作原理

JUnit是由 Erich Gamma 和 Kent Beck 编写的一个回归测试框架（regression testing framework）,用于帮助Java开发人员编写单元测试。 所谓单元测试也就是白盒测试。单元测试在xp社区极为流行，作为测试驱动开发，junit是java开发使用最为广泛的框架。该框架也得到了绝大多数java IDE和其他工具（例如，ant）的集成支持。同时，junit还有很多的第三方扩展和增强包可供使用。junit最近的变化比较少，现在的最高版本仍是3.8.1。

JUnit的使用很简单，关于JUnit使用的文章和例子也很多，本文着重讲解JUnit的基本概念和其设计理念和工作原理。

一.junit基本概念

JUnit有几个基本概念：TestCase，TestSuite，TestFixtrue。

1.TestCase

代表一个测试用例，每一个TestCase实例都对应一个测试，这个测试通过这个TestCase实例的名字标志，以便在测试结果中指明哪个测试出现了问题。

2.TestSuite

代表需要测试的一组测试用例。

3.TestFixtrue

TestFixtrue代表一个测试环境。它用于组合一组测试用例，这组测试用例需要共同的测试运行环境。

二.junit的设计

JUnit的核心是围绕命令模式和组合模式设计的，当然同时使用了模版方法模式，参数收集模式，适配器模式等。这里只是简单介绍。

JUnit框架中有几个核心的接口和类。

1.Test接口

代表一个测试。它是框架的主接口有两个方法：
int countTestCases();//返回所有测试用例的个数。
void run(TestResult result);//运行一个测试，并且收集运行结果到TestResult。

2.TestCase类

TestCase实现了Test接口，是框架提供的供我们继承的类，我们的所有的测试方法都需要在TestCase的子类中定义，并且符合特定的设计协议。

一个TestCase实例代表一个具体的测试实例，对应一个对某一方法或概念的测试。每个TestCase实例都有一个名字。

一个TestCase类却定义了一个TestFixture。

具体的说就是我们自己定义的TestCase子类中可以定义很多的public 没有参数的testxxx方法。运行时，每个testxxx都在自己的fixture中运行。每个运行的TestCase都有一个名字，如果不指定，一般是TestCase中定义的test方法的名字。

3.TestSuite类

和TestCase一样TestSuite也实现了Test接口。一个TestSuite可以包含一系列的TestCase。把testCase组装入TestSuite有几种方式：A，通过将TestCase的Class参数传入TestSuite的构造函数，TestSuite会自动收集TestCase中所有的public的没有参数的testxxx方法加入TestSuite中。
B，构造空的TestSuite后通过void addTest(Test test)方法添加测试。
C：构造空的TestSuite后通过void addTestSuite(Class testClass) 方法添加测试集。

4.TestResult类

主要通过runProtected方法运行测试并收集所有运行结果。

5.TestRunner类

启动测试的主类，我们可以通过直接调用它运行测试用例，IDE和其他一些工具一般也通过这个接口集成JUnit。

6.Assert类

用于断言，TestCase继承自该类，我们的测试方法通过这些断言判断程序功能是否通过测试。

7.TestListener接口

测试运行监听器，通过事件机制处理测试中产生的事件，主要用于测试结果的收集。

以上是框架的核心接口和类的介绍，通过上面的介绍我们很容易看出来Test，TestCase和TestSuite的设计采用了Composite模式。这样JUnit可以一次运行一个测试用例，也可以一次运行多个测试用例，TestRunner只关心Test接口，而对运行的是单个的TestCase还是同时运行多个TestCase并不在意。

TestCase还使用了Template Method模式：

```
public void run() {
   setUp();
   runTest();
   tearDown();
}protected void runTest() {
}

protected void setUp() {
}

protected void tearDown() {
}
```

JUnit同时使用了Command模式，对于典型的Command模式一般有5种角色：

1) 命令角色（Command）：声明执行操作的接口。有java接口或者抽象类来实现。
2) 具体命令角色（Concrete Command）：将一个接收者对象绑定于一个动

作；调用接收者相应的操作，以实现命令角色声明的执行操作的接口。

3) 客户角色（Client）：创建一个具体命令对象（并可以设定它的接收者）。

4) 请求者角色（Invoker）：调用命令对象执行这个请求。

5) 接收者角色（Receiver）：知道如何实施与执行一个请求相关的操作。任何类都可能作为一个接收者。

对于JUnit的设计，不能明显的区分出这5种角色，因为它的设计相对复杂，同时参杂了其他模式。

Test接口可以认为是命令模式中的命令角色Command接口，void run(TestResult result)接口方法定义了需要执行的操作；TestCase可以看作是具体命令角色，但又不全是，因为我们还需要自己通过继承TestCase类定义测试方法，这样的每一个测试方法都回被包装在一个TestCase实例中。TestResult可以看作请求者角色（Invoker），它会通过protected void run(final TestCase test) 运行测试并收集结果。我们自己写的Test方法可以认为是接收者角色（Receiver），因为我们的方法才具体执行这个命令。TestRunner就是客户角色（Client），它通过TestResult result= createTestResult()构造TestResult，并通过suite.run(result)运行测试用例（suite是一个Test接口的具体实例，可以是TestCase也可以是TestSuite，但客户端不关心它是什么，这就是组合模式的好处。同时，suite.run（result）又调用result.run（test），如果不仔细分析，就会被这种设计搞迷惑）。

三.junit的工作原理

知其然还要知其所以然，JUnit使用比较简单，但为什么要这样使用，什么是best practices，就需要了解JUnit的内部工作原理了。

一个简单的测试的例子：

```
/**
 * A sample test case, testing <code>java.util.Vector</code>.
 *
 */
public class VectorTest extends TestCase {
 protected Vector fEmpty;
 protected Vector fFull;

 public static void main (String[] args) {
  junit.textui.TestRunner.run (suite());
 }
```

```
protected void setUp() {
 fEmpty= new Vector();
 fFull= new Vector();
 fFull.addElement(new Integer(1));
 fFull.addElement(new Integer(2));
 fFull.addElement(new Integer(3));
}
public static Test suite() {
 return new TestSuite(VectorTest.class);
}
public void testCapacity() {
 int size= fFull.size();
 for (int i= 0; i < 100; i++)
  fFull.addElement(new Integer(i));
 assertTrue(fFull.size() == 100+size);
}
public void testClone() {
 Vector clone= (Vector)fFull.clone();
 assertTrue(clone.size() == fFull.size());
 assertTrue(clone.contains(new Integer(1)));
}
public void testContains() {
 assertTrue(fFull.contains(new Integer(1)));
 assertTrue(!fEmpty.contains(new Integer(1)));
}
public void testElementAt() {
 Integer i= (Integer)fFull.elementAt(0);
 assertTrue(i.intValue() == 1);

 try {
  fFull.elementAt(fFull.size());
 } catch (ArrayIndexOutOfBoundsException e) {
  return;
 }
 fail("Should raise an ArrayIndexOutOfBoundsException");
}
public void testRemoveAll() {
 fFull.removeAllElements();
 fEmpty.removeAllElements();
 assertTrue(fFull.isEmpty());
```

```
 assertTrue(fEmpty.isEmpty());
 }
 public void testRemoveElement() {
  fFull.removeElement(new Integer(3));
  assertTrue(!fFull.contains(new Integer(3)) );
 }
}
```

1. 主入口。

```
public static void main (String[] args) {
 junit.textui.TestRunner.run (suite());
 }
```

调用了TestRunner.run（Test）方法，该方法执行的操作的结果就是构造TestResult，然后调用TestResult.run（Test）方法。

当然，TestRunner也有main方法，可以在命令行下执行，main方法执行的结果和TestRunner.run 相似。

2.suite（ ）

```
    public static Test suite() {
        return new TestSuite(VectorTest.class);
    }
```

这是JUnit框架使用TestSuite规定的模式，尤其是在命令行或图形界面下，只有定义了public static Test suite() 方法，框架才会按照我们的定义运行框架。

3.new TestSuite(VectorTest.class)

TestSuite有三个构造函数一个没有参数，一个以Class为参数还有一个以Class和名字字符串作为参数。Class必须是实现了Test接口的子类，一般继承自TestCase，并且该类中定义了以Test开头没有参数的测试方法。

```
//构造函数

 while (Test.class.isAssignableFrom(superClass)) {
  Method[] methods= superClass.getDeclaredMethods();
  for (int i= 0; i < methods.length; i++) {
```

```
    addTestMethod(methods[i], names, theClass);
   }
   superClass= superClass.getSuperclass();
  }

 //addTestMethod

  if (! isPublicTestMethod(m)) {
   if (isTestMethod(m))
    addTest(warning("Test method isn't public: "+m.getName()));
   return;
  }
  names.addElement(name);
  addTest(createTest(theClass, name));

 //isTestMethod

 private boolean isTestMethod(Method m) {
  String name= m.getName();
  Class[] parameters= m.getParameterTypes();
  Class returnType= m.getReturnType();
  return parameters.length == 0 && name.startsWith("test") &&
returnType.equals(Void.TYPE);
  }
```

所以，我们必须保证我们要在实现Test接口的子类中定义符合以上要求的测试类，否则，框架将不会运行我们的测试方法。

同时我们也看到，当我们将Test类传入TestSuite后，TestSuite将所有的test构造为TestCase实例，每个实例都会有一个名字，和要测试的方法。

```
//createTest  ( addTest(createTest(theClass, name))时调用 )

 static public Test createTest(Class theClass, String name) {
  Constructor constructor;
  try {
   constructor= getTestConstructor(theClass);
  } catch (NoSuchMethodException e) {
   return warning("Class "+theClass.getName()+" has no public constructor
TestCase(String name) or TestCase()");
```

```
  }
  Object test;
  try {
   if (constructor.getParameterTypes().length == 0) {
    test= constructor.newInstance(new Object[0]);
    if (test instanceof TestCase)
     ((TestCase) test).setName(name);
   } else {
    test= constructor.newInstance(new Object[]{name});
   }
  } catch (InstantiationException e) {
   return(warning("Cannot instantiate test case: "+name+"
("+exceptionToString(e)+")"));
  }
  return (Test) test;
 }
```

在createTest(Class theClass, String name)方法中theClass是实现了Test接口的类，一般情况下是TestCase的子类。这样name就是TestCase的名字，用于标志一个测试。

在TestCase的protected void runTest() throws Throwable 方法中：

```
runMethod= getClass().getMethod(fName, null);//获取方法名

runMethod.invoke(this, new Class[0]);//运行测试方法。
```

这也就是上面说的，一个TestCase类中可以定义很多test方法，但一个TestCase实例只对应一个测试方法。

Test.run（TestResult）是实际上调用TestResult.run（Test）。

```
//TestResult.run（Test）

 protected void run(final TestCase test) {
  startTest(test);
  Protectable p= new Protectable() {
   public void protect() throws Throwable {
    test.runBare();
   }
```

```
 };
 runProtected(test, p);

 endTest(test);
}
```

该方法也是一个模版方法，关键是runProtected方法：

```
public void runProtected(final Test test, Protectable p) {
 try {
  p.protect();
 }
 catch (AssertionFailedError e) {
  addFailure(test, e);
 }
 catch (ThreadDeath e) { // don't catch ThreadDeath by accident
  throw e;
 }
 catch (Throwable e) {
  addError(test, e);
 }
}
```

这个方法起到的作用就是当测试出现异常或错误时会在TestResult中记录，如果是ThreadDeath 就继续抛出异常，程序可能会终止，如果是其他错误和异常，程序将继续运行。这也就是protect的含义。因此，我们抛出Assert中的异常时，不会影响下面的测试继续运行。

这个方法中传入的Protectable p的protect方法调用了test.runBare();我们看到，TestRunner调用了Test的run（TestResult）方法后，TestResult实际上是在一个保护的环境下调用TestCase的runBase方法。也就是说如果我们自己的测试类没有继承TestCase，也要定义一个runBase方法，执行基本的操作。基于这一点，我们的测试类应该继承自TestCase。

5.一个比较绕的地方

TestSuite也是一个Test，也有我们的例子中是调用TestSuite的void run(TestResult result)方法。

```
public void run(TestResult result) {
```

```
 for (Enumeration e= tests(); e.hasMoreElements(); ) {
   if (result.shouldStop() )
    break;
  Test test= (Test)e.nextElement();
  runTest(test, result);
 }
}

public void runTest(Test test, TestResult result) {
 test.run(result);
}
```

上面的代码首先是一个循环，其次看起来好像形成了一个环：run(TestResult result) -〉runTest(test, result)-〉test.run(result)-〉run(TestResult result)，这就要求我们必须对java的运行时类型比较熟悉才能比较容易的理解。

tests()是在TestSuite构造时构造的所有的tests，一般情况下这些test都是TestCase子类的实例。不过也有可能是另一个TestSuite，比如通过 public void addTestSuite(Class testClass) 方法提阿加：

```
public void addTestSuite(Class testClass) {
  addTest(new TestSuite(testClass));
}
```

如果test是TestCase，就直接调用按照4中说明的方法运行。如果是TestSuite就会递归调用TestSuite中定义的void run(TestResult result)方法，直到没有了suite中的所有的Test都不是suite(因为suite在构造时就会将suite解析为单个的TestCase的集合，可以通过tests（ ）方法取出所有的test)。

这里不太容易说清楚，仔细看看代码思考一下就明白了。

以上介绍了JUnit的基本概念和设计理念及工作原理。junit当然也有很多不尽人意的地方。不过由于它是java中使用的最广泛的单元测试框架，我们还是需要仔细研究一下的。另外，TestNG是一个比较不错的框架，原先了结果，由于当时没有写笔记，现在差不多都忘了，如果有机会可以了解一下。

1. http://www.cnblogs.com/growing/archive/2010/10/14/1851273.html
2. http://tag.bokee.com/tag/junit

3. http://tag.bokee.com/tag/%C9%E8%BC%C6%C4%A3%CA%BD
4. http://tag.bokee.com/tag/%B5%A5%D4%AA%B2%E2%CA%D4