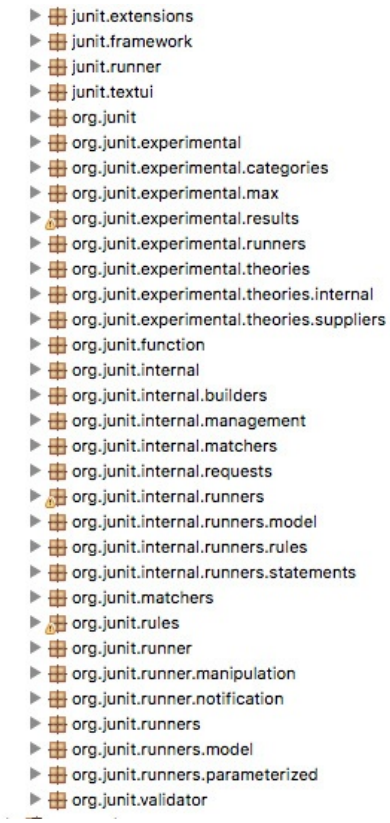
Junit3源码（200@365）

2017-07-19 14:43:02

最近打算看看junit的源码，下载的是：junit4.13-SNAPSHOT 代码里面的包有两种：一种是junit.开头的包（junit3），另一种是org.junit.开头的包（junit4）。



首先简单分析一下junit3。
入口类是junit.textui.TestRunner，首先看一下，该类中的main函数：

```
public static void main(String[] args) {
        TestRunner aTestRunner = new TestRunner();
        try {
            TestResult r = aTestRunner.start(args);
            if (!r.wasSuccessful()) {
                System.exit(FAILURE_EXIT);
            }
            System.exit(SUCCESS_EXIT);
        } catch (Exception e) {
            System.err.println(e.getMessage());
            System.exit(EXCEPTION_EXIT);
        }
    }
```

main函数没有特别复杂的逻辑，就是创建一个TestRunner对象，并调用该对象的start函数，返回TestResult，之后判断测试的状态是否成功。那么，显然，关键应该在start函数里面。

```
    public TestResult start(String[] args) throws Exception {
        String testCase = "";
        String method = "";
        boolean wait = false;

        for (int i = 0; i < args.length; i++) {
            if (args[i].equals("-wait")) {
                wait = true;
            } else if (args[i].equals("-c")) {
                testCase = extractClassName(args[++i]);
            } else if (args[i].equals("-m")) {
                String arg = args[++i];
                int lastIndex = arg.lastIndexOf('.');
                testCase = arg.substring(0, lastIndex);
                method = arg.substring(lastIndex + 1);
            } else if (args[i].equals("-v")) {
                System.err.println("JUnit " + Version.id() + " by Kent Beck and Erich Gamma");
            } else {
                testCase = args[i];
            }
        }

        if (testCase.equals("")) {
            throw new Exception("Usage: TestRunner [-wait] testCaseName, where name is the name of the TestCase class");
        }
```

```
        try {
            if (!method.equals("")) {
                return runSingleMethod(testCase, method, wait);
            }
            Test suite = getTest(testCase);
            return doRun(suite, wait);
        } catch (Exception e) {
            throw new Exception("Could not create and run test suite: " + e);
        }

    }
```

start函数的参数就是main函数的参数，首先使用一个for循环处理参数，可以看出，参数主要可以有-wait，-c ， -m ， 几种，处理完参数之后，可以分为只运行单独一个method和正常的测试用例的运行，第一种就是只运行一个方法，我们主要讲一下正常的测试用例的运行，也就是Test suite = getTest(testCase);return doRun(suite, wait);这两行代码，基本上的意义就是，根据testcase（一个类名）获得Test对象，然后运行该对象获得最终的测试结果。

那么，我们来看看getTest函数是怎么获得Test对象的。

```
public Test getTest(String suiteClassName) {
    if (suiteClassName.length() <= 0) {
        clearStatus();
        return null;
    }
    Class<?> testClass = null;
    try {
        testClass = loadSuiteClass(suiteClassName);
    } catch (ClassNotFoundException e) {
        String clazz = e.getMessage();
        if (clazz == null) {
            clazz = suiteClassName;
        }
        runFailed("Class not found "" + clazz + """);
        return null;
    } catch (Exception e) {
        runFailed("Error: " + e.toString());
        return null;
    }
    Method suiteMethod = null;
```

```
    try {
        suiteMethod = testClass.getMethod(SUITE_METHODNAME);
    } catch (Exception e) {
        // try to extract a test suite automatically
        clearStatus();
        return new TestSuite(testClass);
    }
    if (!Modifier.isStatic(suiteMethod.getModifiers())) {
        runFailed("Suite() method must be static");
        return null;
    }
    Test test = null;
    try {
        test = (Test) suiteMethod.invoke(null); // static method
        if (test == null) {
            return test;
        }
    } catch (InvocationTargetException e) {
        runFailed("Failed to invoke suite():" + e.getTargetException().toString());
        return null;
    } catch (IllegalAccessException e) {
        runFailed("Failed to invoke suite():" + e.toString());
        return null;
    }

    clearStatus();
    return test;
}
```

代码看起来有点长，其实逻辑很简单。suiteClassName是类名，首先根据类名，加载类，获得Class对象。然后检查该对象中是否存在suite函数，并且该函数必须是static类型，返回一个Test对象，如果存在，直接调用该函数，返回Test对象，否则，返回new TestSuite(testClass)（testClass是类名suiteClassName对应的Class对象）。逻辑很清晰，不过并没有说明Test对象究竟是怎么获得的，为了了解Test对象的构造，我们先了解一下junit中对测试用例部分的设计。junit在测试用例部分主要使用了组合模式，首先定义了基本的接口Test，TestSuite和TestCase都实现了Test接口，其中TestSuite中包含了private Vector<Test> fTests字段，即TestSuite是由TestCase或者其他的TestSuite组成的，TestCase则是基本的测试用例。

然后我们看看suite函数的例子，看看是如何构造Test对象的。

```
public static Test suite() {
        final TestSuite s = new TestSuite();
//添加一个TestSuite
        TestSuite s2 = new TestSuite(new Class<?>[]{Test3.class});
        s.addTest(s2);
//添加一个测试method
        Test3 test = new Test3();
        test.setName("test1");
        s.addTest(test);
//添加一个TestCase
        s.addTestSuite(Test3.class);
        return s;
    }
```

其中，Test3是TestCase的一个子类，test1是Test3中的一个public的方法。我们可以给TestSuite分别添加一个TestSuite，添加一个TestMethod以及添加一个TestCase。其中添加TestSuite没什么好说的，new一个TestSuite，使用addTest添加即可；添加一个TestMethod，其实就是new一个相应的TestCase对象，把该对象里面的一个方法设为该对象的name，然后使用addTest添加该TestCase对象；添加一个TestCase类，addTestSuite(Test3.class)看起来就有点抽象了，总不会直接将Class对象添加到private Vector<Test> fTests字段里面，那么，是怎么通过Class对象构造Test的呢？（注意：addTestSuite中的Class对象必须是TestCase对象或者TestCase的子类对象的Class对象）

主要逻辑在addTestsFromTestCase函数里面，通过Class对象构造TestCase加入Vector<Test> fTests字段里面。经过分析，主要部分在while (Test.class.isAssignableFrom(superClass))这个循环里面，主要过程是遍历Class对象里面的函数，然后调用addTestMethod(each, names, theClass);函数将符合要求的测试函数构造成测试用例添加到数组里面。然后获得Class对象的父类对象的Class对象，重复这个过程。

```
private void addTestsFromTestCase(final Class<?> theClass) {
        fName = theClass.getName();
        try {
            getTestConstructor(theClass); // Avoid generating multiple error messages
        } catch (NoSuchMethodException e) {
            addTest(warning("Class " + theClass.getName() + " has no public constructor
TestCase(String name) or TestCase()"));
            return;
        }

        if (!Modifier.isPublic(theClass.getModifiers())) {
            addTest(warning("Class " + theClass.getName() + " is not public"));
            return;
        }

        Class<?> superClass = theClass;
        List<String> names = new ArrayList<String>();
        while (Test.class.isAssignableFrom(superClass)) {
            for (Method each : MethodSorter.getDeclaredMethods(superClass)) {
                addTestMethod(each, names, theClass);
            }
            superClass = superClass.getSuperclass();
        }
        if (fTests.size() == 0) {
            addTest(warning("No tests found in " + theClass.getName()));
        }
    }
```

addTestMethod函数又调用了其他的函数实现构造的功能，其主要的构造过程如下所示：其中constructor是根据Class对象获得的构造器，name是函数的名字。至于那些函数将被作为TestCase加入测试用例集合呢，规定：以test开头，public，返回值void，参数空的函数将作为测试用例。

```
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});
        }
```

假设有测试用例Test3：那么最终将构成两个测试用例Test3(test1)和Test3(test2)。

```
public class Test3 extends TestCase{

    public  void test1(){
        fail("fail");
    }

    public void test2(){
        throw new RuntimeException("error");
    }

    public void hello(){
        assertEquals(true, true);
    }
}
```

知道了getTest的过程，那么下面就应该是Test的执行过程了，你还记得TestRunner里面的doRun函数吗？

```
public TestResult doRun(Test suite, boolean wait) {
        TestResult result = createTestResult();
        result.addListener(fPrinter);
        long startTime = System.currentTimeMillis();
        suite.run(result);
        long endTime = System.currentTimeMillis();
        long runTime = endTime - startTime;
        fPrinter.print(result, runTime);

        pause(wait);
        return result;
    }
```

逻辑看起来清晰简单，首先创建一个TestResult，存储测试结果，fPrinter负责输出测试结果，runTime统计测试时间，suite.run(result)执行测试用例集合。可以看出关键的执行部分位于Test对象的run函数，TestSuite和TestCase都实现了Test的run方法，我们分别来看看。

TestSuite中的run方法就是遍历所有的Test，然后调用runTest方法，执行测试用例， runTest方法做了什么呢？又重新调用了run方法，做成了递归的效果，看起来没完没了，不过不要忘了，Test也可以是TestCase。

```
public void run(TestResult result) {
        for (Test each : fTests) {
            if (result.shouldStop()) {
                break;
            }
            runTest(each, result);
        }
    }

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

TestCase中的run方法：看起来不能更简洁，直接调用TestResult中的run方法。

```
public void run(TestResult result) {
        result.run(this);
    }
```

那么TestResult中的run方法又干了什么呢？它层层调用了很多方法，综合起来，其实就是调用了TestCase中的runTest方法，该方法的核心就两行代码，其他大多是错误处理：1、runMethod = getClass().getMethod(fName, (Class[]) null);获得测试方法；2、runMethod.invoke(this);调用测试方法。

将我们样例里面的suite方法放入Test4类里面，将TestRunner的命令行参数设为Test4，运行TestRunner，可以看到测试结果：

```
.F.E.F.F.E
Time: 0.002
There were 2 errors:
1) test2(Test3)java.lang.RuntimeException: error
at Test3.test2(Test3.
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.
2) test2(Test3)java.lang.RuntimeException: error
at Test3.test2(Test3.
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.
There were 3 failures:
1) test1(Test3)junit.framework.AssertionFailedError: fail
at Test3.test1(Test3.
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.
2) test1(Test3)junit.framework.AssertionFailedError: fail
at Test3.test1(Test3.
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.
3) test1(Test3)junit.framework.AssertionFailedError: fail
at Test3.test1(Test3.
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.

FAILURES!!!
Tests run: 5，Failures: 3，Errors: 2
```

Junit3的整个运行流程大概就是这样的，至于一些具体的细节我们省略了，如果有兴趣大家可以自己去看看源代码，下回打算研究一个Junit4的执行流程。