阅读下面单元测试相关工具的介绍（或查阅其它单元测试工具相关资料），学习单元测试工具的使用

JUnit-Tutorial.pdf

junit_tutorial.pdf

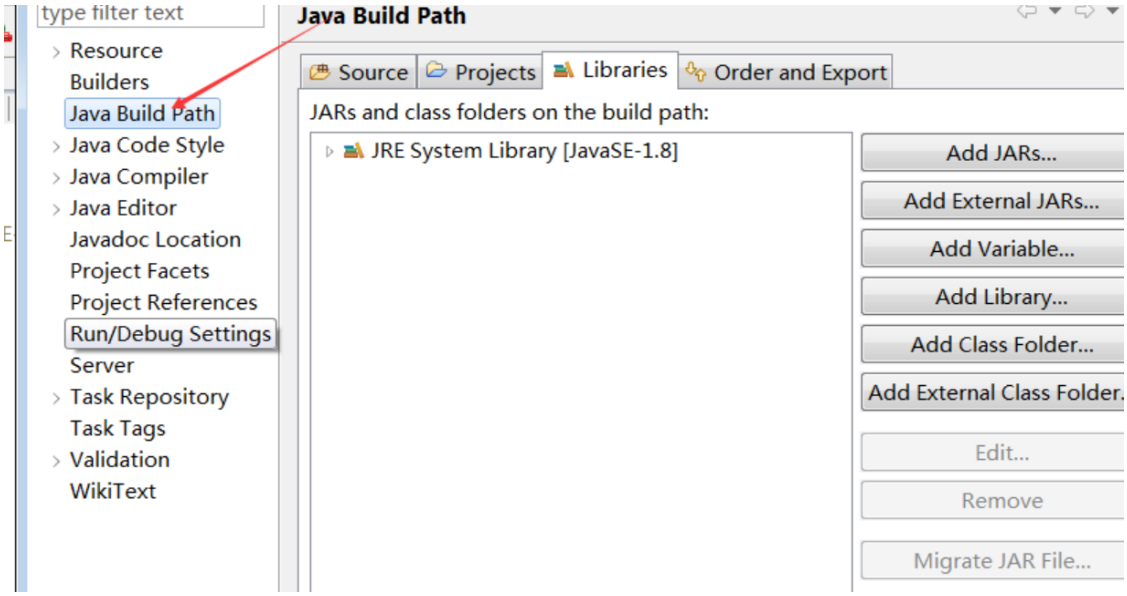
Embunit User Guide.pdf

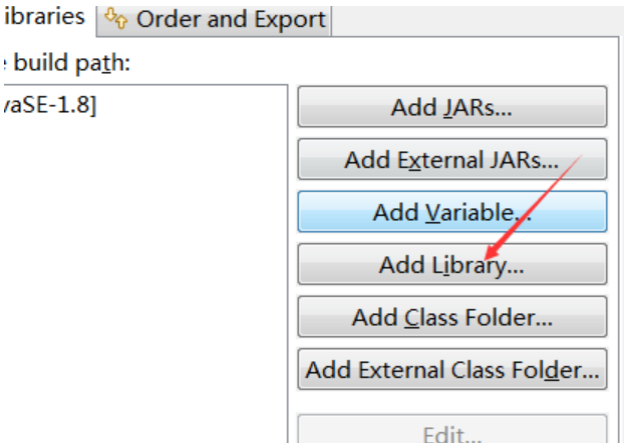
JUnit

介绍：JUnit 是一个Java编程语言的单元测试框架。JUnit 在测试驱动的开发方面有很重要的发展，是起源于 JUnit 的一个统称为 xUnit 的单元测试框架之一。

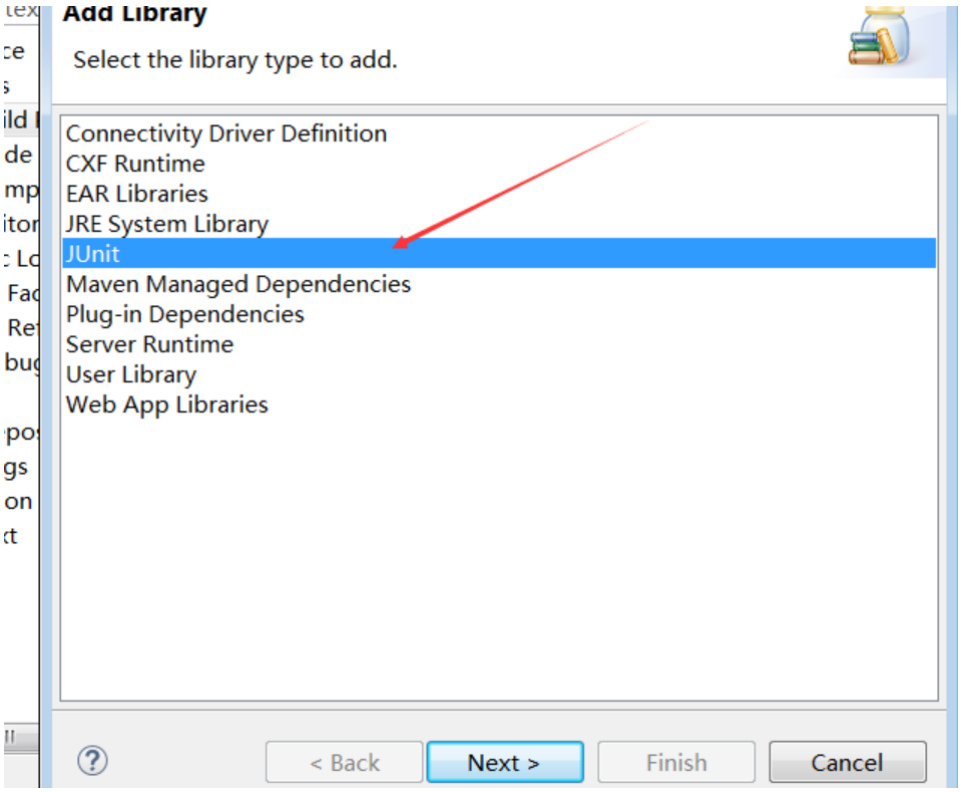
断言机制：将程序预期的结果与程序运行的最终结果进行比对，确保对结果的可预知性

实战步骤：

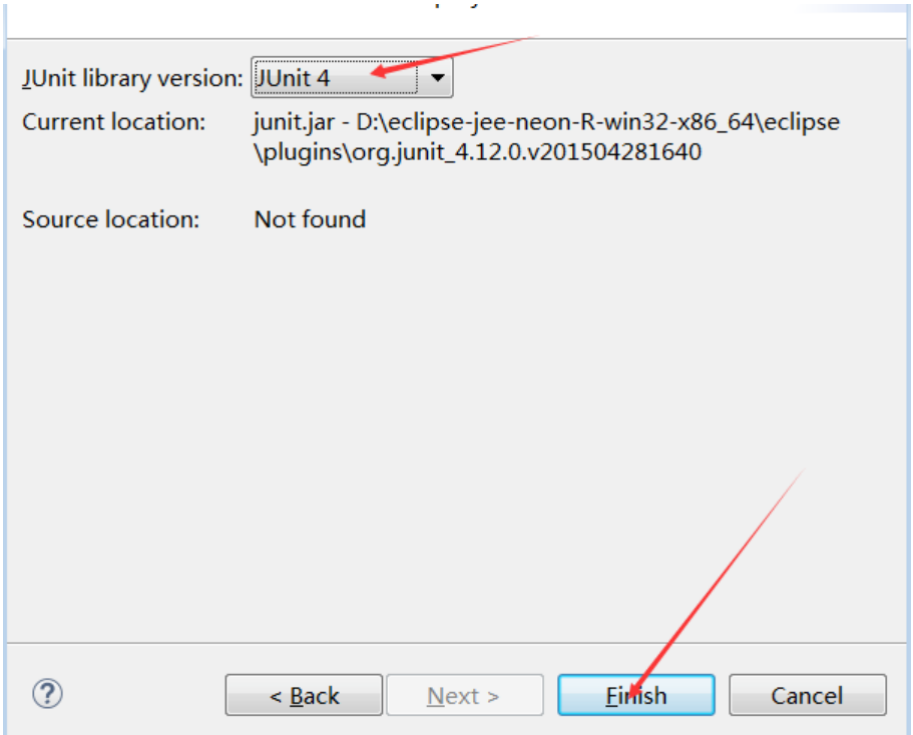
1. 新建一个工程
2. 工程右键，点击Properties
3. 

4. 

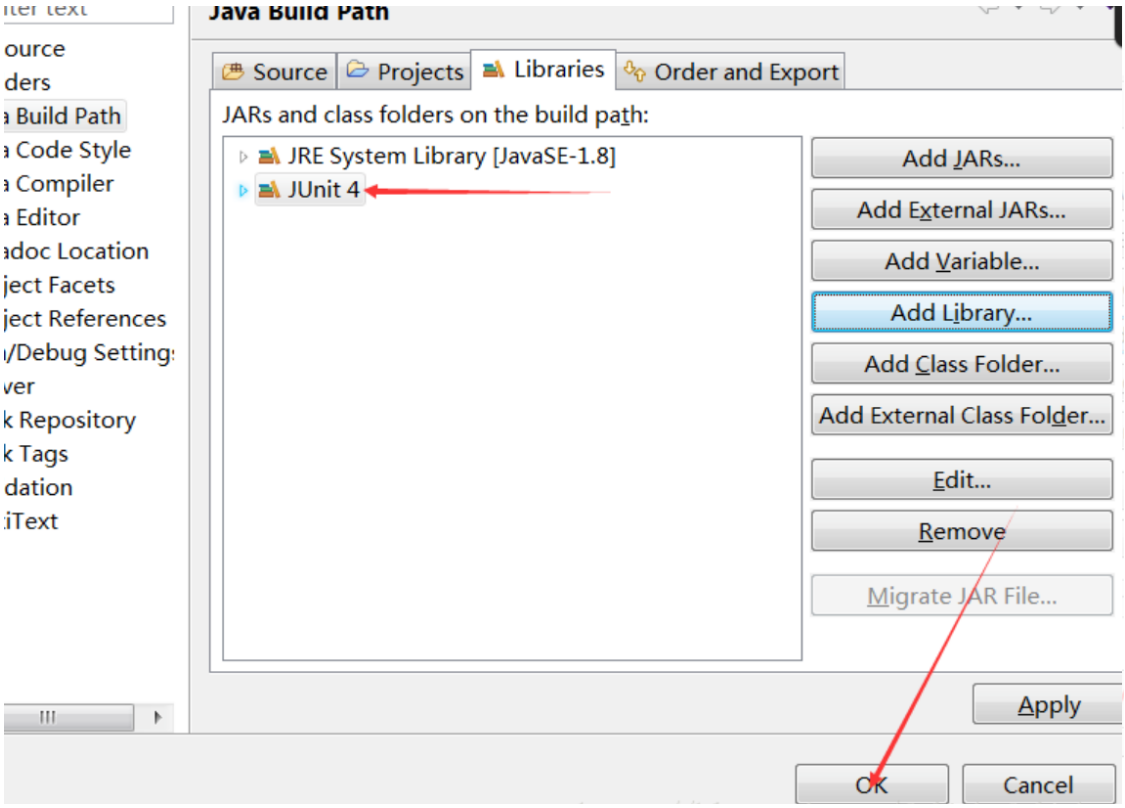
5.


Add Library
Select the library type to add.
Connectivity Driver Definition
CXF Runtime
EAR Libraries
JRE System Library
JUnit
Maven Managed Dependencies
Plug-in Dependencies
Server Runtime
User Library
Web App Libraries
< Back
Next >
Finish
Cancel

6.


JUnit library version:
JUnit 4
Current location:
junit.jar - D:\eclipse-jee-neon-R-win32-x86_64\eclipse\plugins\org.junit_4.12.0.v201504281640
Source location:
Not found
< Back
Next >
Finish
Cancel

7.



这是使用Junit最先要完成的 在完成了这个就进入下一步吧!

来简单使用一下Junit!

新建一个类：Calculate

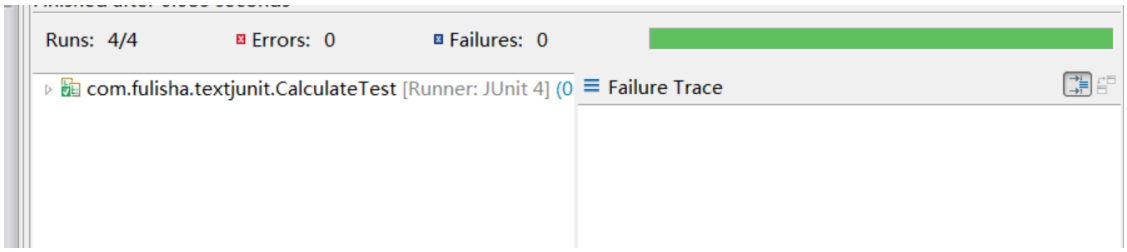
```
package com.fulisha.textjunit;

public class Calculate {
public int add(int a,int b){
    return a+b;
}
public int substract(int a , int b){
    return a-b;
}
public int cheng(int a,int b){
    return a*b;
}
public int chu(int a, int b){
    return a/b;
}

}
```

再新建一个测试类

```
package com.fulisha.textjunit;

import static org.junit.Assert.*;

import org.junit.Test;

public class CalculateTest {

    @Test
```

```
    public void testAdd(){
        assertEquals(6,new Calculate().add(3, 3));
    }

    @Test
    public void testsubstract(){
        assertEquals(2,new Calculate().substract(5, 3));
    }

    @Test
    public void testcheng(){
        assertEquals(15,new Calculate().cheng(5, 3));
    }
    @Test
    public void testchu(){
        assertEquals(2,new Calculate().chu(6, 3));
    }
}
```

测试后的结果：



对此进行总结：

1.测试方法上必须使用@Test

2.测试方法必须使用 public void进行修饰

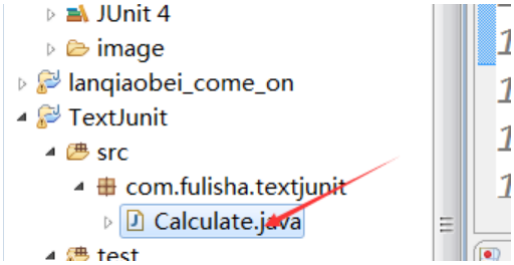
3.新建一个源代码目录来存放测试代码

4.测试类的包应该和被测试类的包一样

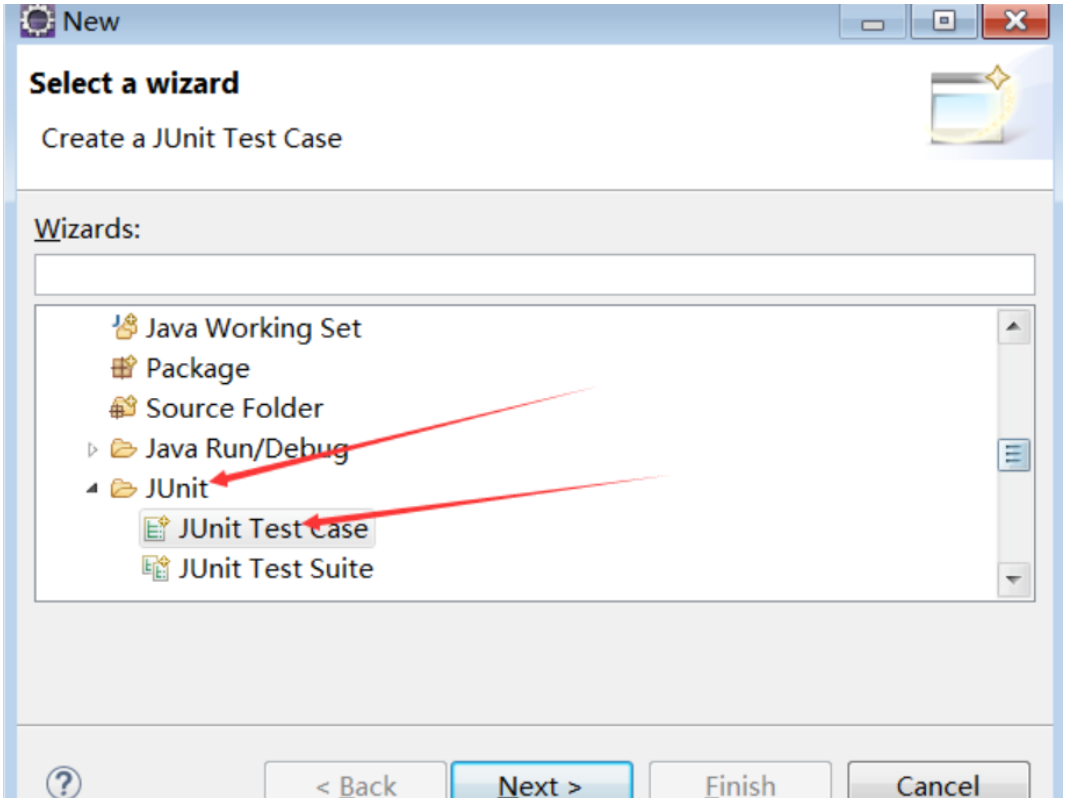
5.测试单元中的每个方法一定要能够独立测试，其方法不能有任何依赖

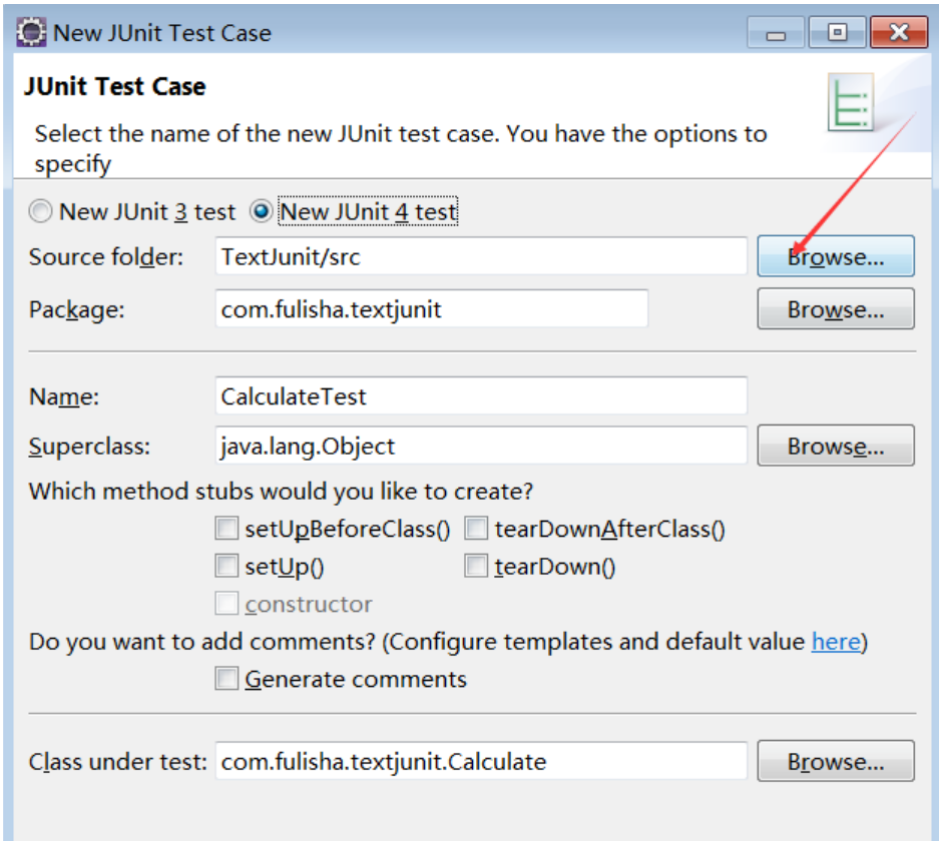
如果，测试的方法多，不想一个个的建立测试方法那么：

- 选择要测试的方法，右击New 选择other

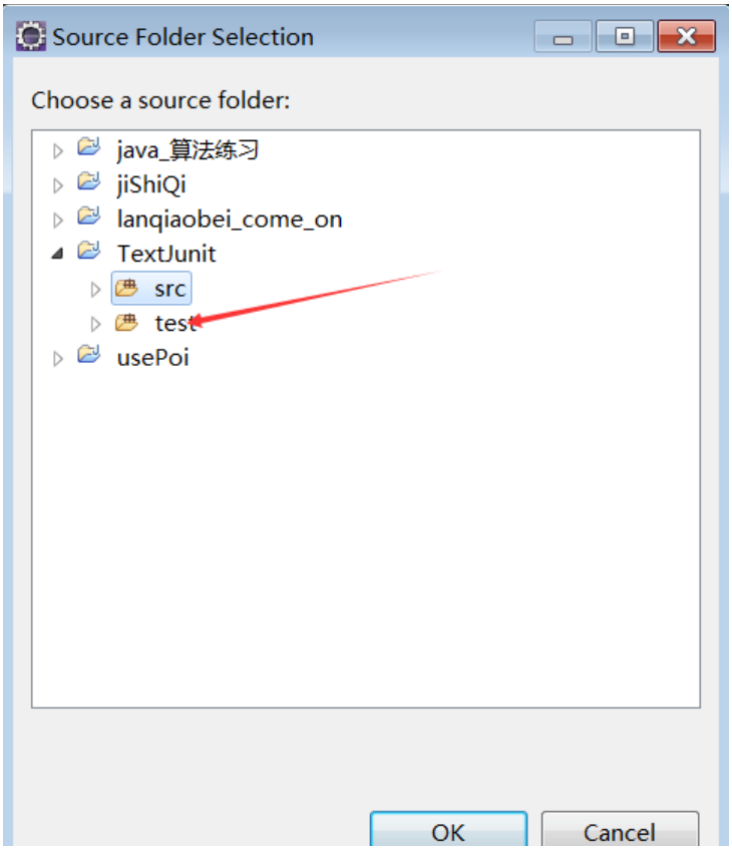


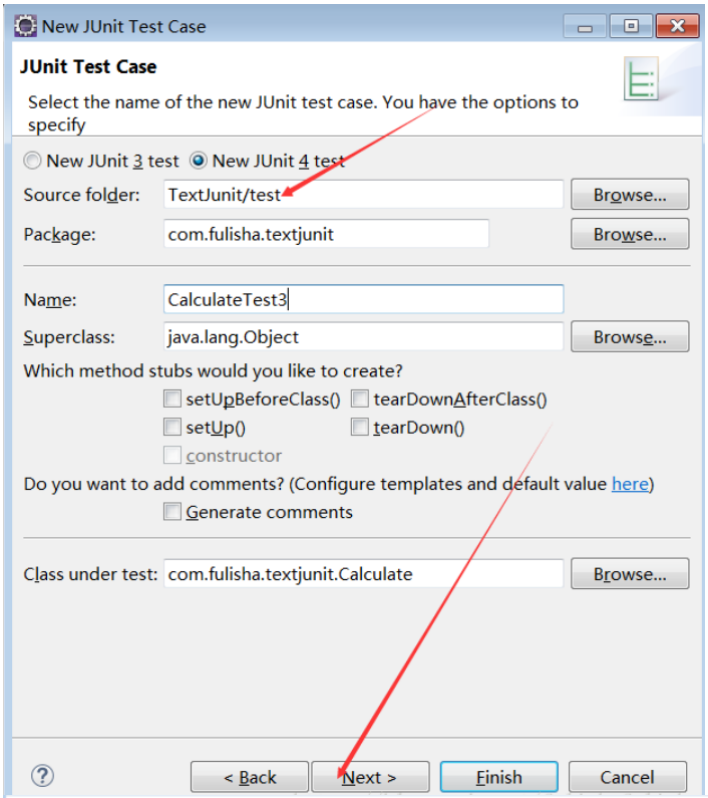
-

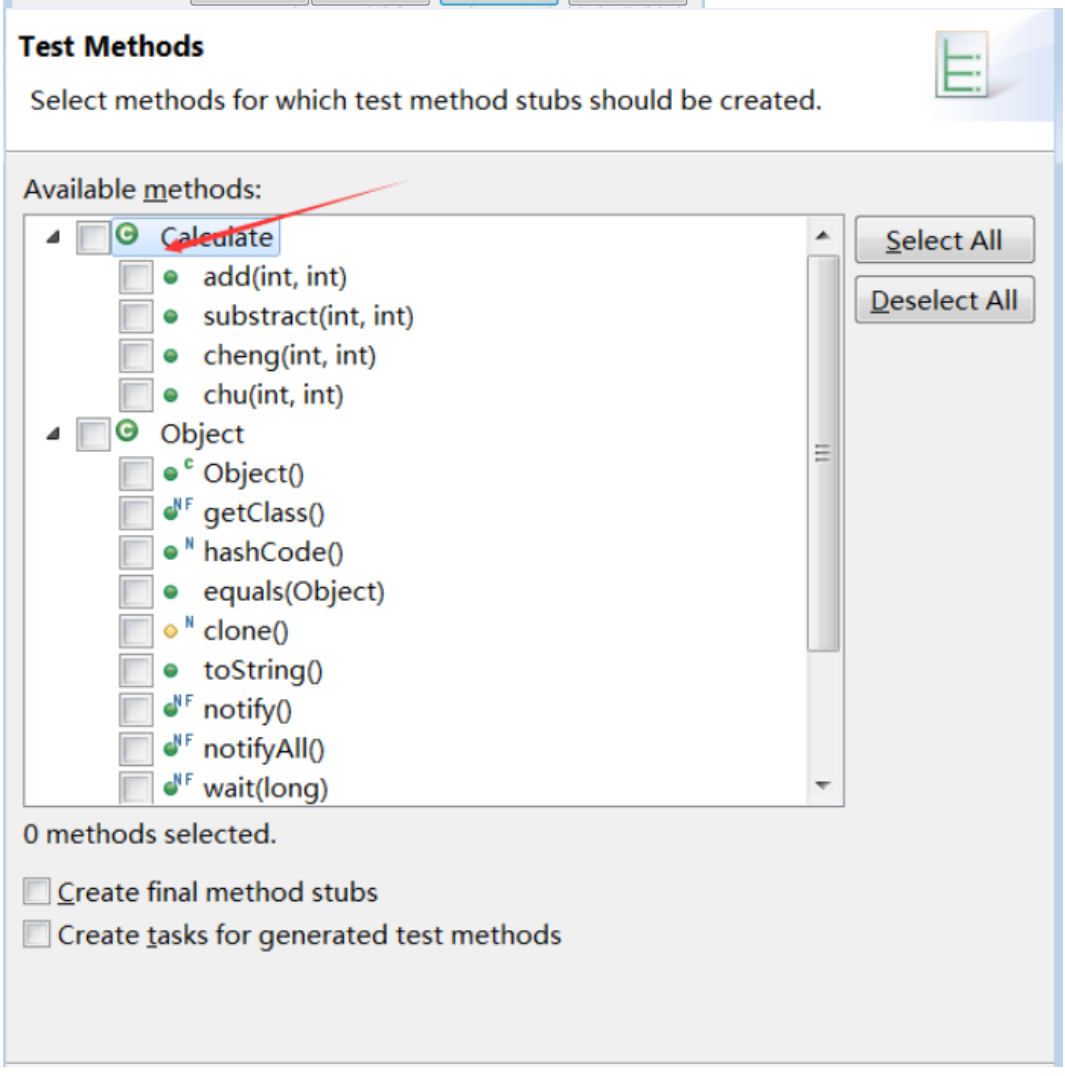

New
Select a wizard
Create a JUnit Test Case
Wizards:
Java Working Set
Package
Source Folder
Java Run/Debug
JUnit
JUnit Test Case
JUnit Test Suite
< Back
Next >
Finish
Cancel

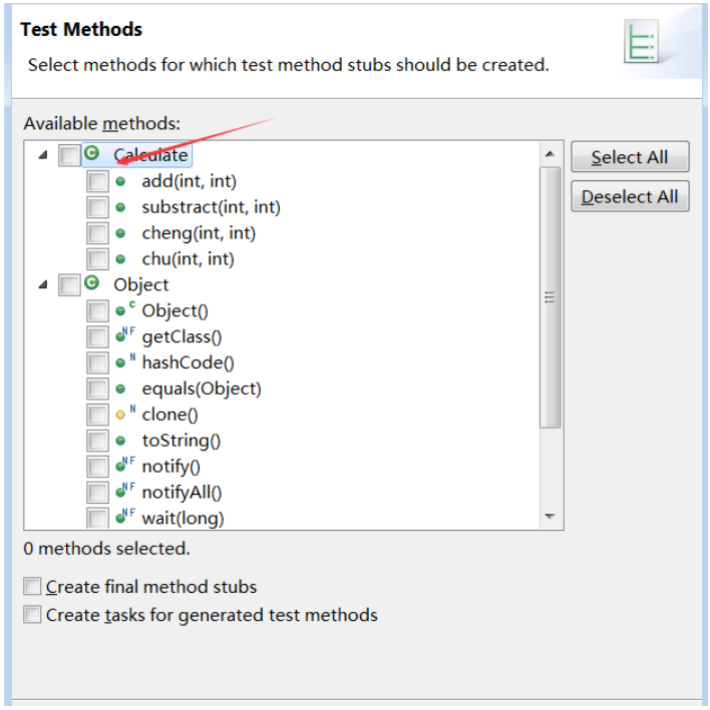
- 
New JUnit Test Case
JUnit Test Case
Select the name of the new JUnit test case. You have the options to specify
New JUnit 3 test
New JUnit 4 test
Source folder:
TextJunit/src
Browse...
Package:
com.fulisha.textjunit
Browse...
Name:
CalculateTest
Superclass:
java.lang.Object
Browse...
Which method stubs would you like to create?
setUpBeforeClass()
tearDownAfterClass()
setUp()
tearDown()
constructor
Do you want to add comments? (Configure templates and default value here)
Generate comments
Class under test:
com.fulisha.textjunit.Calculate
Browse...

-


Source Folder Selection
Choose a source folder:
java_算法练习
jiShiQi
lanqiaobei_come_on
TextJunit
src
test
usePoi
OK
Cancel


New JUnit Test Case
JUnit Test Case
Select the name of the new JUnit test case. You have the options to specify
New JUnit 3 test
New JUnit 4 test
Source folder:
TextJunit/test
Browse...
Package:
com.fulisha.textjunit
Browse...
Name:
CalculateTest3
Superclass:
java.lang.Object
Browse...
Which method stubs would you like to create?
setUpBeforeClass()
tearDownAfterClass()
setUp()
tearDown()
constructor
Do you want to add comments? (Configure templates and default value here)
Generate comments
Class under test:
com.fulisha.textjunit.Calculate
Browse...
< Back
Next >
Finish
Cancel


Test Methods
Select methods for which test method stubs should be created.
Available methods:
Calculate
add(int, int)
substract(int, int)
cheng(int, int)
chu(int, int)
Object
Object()
getClass()
hashCode()
equals(Object)
clone()
toString()
notify()
notifyAll()
wait(long)
Select All
Deselect All
0 methods selected.
Create final method stubs
Create tasks for generated test methods



- 创建结果

```
package com.fulisha.textjunit;

import static org.junit.Assert.*;

import org.junit.Test;

public class CalculateTest3 {

    @Test
    public void testAdd() {
        fail("Not yet implemented");
    }

    @Test
    public void testSubstract() {
        fail("Not yet implemented");
    }

    @Test
    public void testCheng() {
        fail("Not yet implemented");
    }

    @Test
    public void testChu() {
        fail("Not yet implemented");
    }

}
```

再在这个基础上进行修改测试类方法

在测试中结果中关于Failure和error的解释

1.Failure 一般由测试单元使用断言方法判断失败引起的，这个报错，说明测试点发现了问题，即程序输出的结果和我们预期的不一样

2.error 是由代码异常引起的，它可以产生代码本身的错误，也可以是测试代码中的一个隐藏bug

Embunit

介绍：Embedded Unit（简称embUnit）是一个针对嵌入式C系统的单元测试框架。它不依赖于标准的C函数库，所有的对象都被静态编译链接。因此，可以比较方便地将其移植到嵌入式平台。

移植思路：

1. 由于embUnit不依赖于标准的C函数库，因此，将我们的编译选项添加到embUnit中的Makefile中，将其代码编译成一个静态库，然后链接到我们原有的程序中；
2. 额外创建一个源文件，用于编写测试代码，该文件也通过编译、链接，将其与原有的程序链接在一起；
3. 在原有程序的main中（即程序入口处），调用embUnit这个框架提供的API函数，执行对函数的单元测试；

实战步骤：

1. 下载的embUnit的源代码解压后，在解压形成的目录里面有一个**embunit**目录，这个目录就是embUnit的源代码所在的目录，首先我们分析这个目录下的**makefile**文件，文件内容如下：

```
View makefile

CC = gcc
CFLAGS = -O
AR = ar
ARFLAGS = ru
RANLIB = ranlib
RM = rm
OUTPUT = ../lib/
TARGET = libembUnit.a
OBJS = AssertImpl.o RepeatedTest.o stdImpl.o TestCaller.o TestCase.o
TestResult.o TestRunner.o TestSuite.o

all: $(TARGET)

$(TARGET): $(OBJS)
    $(AR) $(ARFLAGS) $(OUTPUT)$@ $(OBJS)
    $(RANLIB) $(OUTPUT)$@

.c.o:
    $(CC) $(CFLAGS) $(INCLUDES) -c $<

AssertImpl.o: AssertImpl.h stdImpl.h
RepeatedTest.o: RepeatedTest.h Test.h
stdImpl.o: stdImpl.h
TestCaller.o: TestCaller.h TestResult.h TestListener.h TestCase.h Test.h
TestCase.o: TestCase.h TestResult.h TestListener.h Test.h
TestResult.o: TestResult.h TestListener.h Test.h
TestRunner.o: TestRunner.h TestResult.h TestListener.h Test.h stdImpl.h
config.h
TestSuite.o: TestSuite.h TestResult.h TestListener.h Test.h
```

```
clean:
    -$(RM) $(OBJS) $(TARGET)

.PHONY: clean all
```

分析可知，这个makefile会将当前目录下的源文件编译和链接成一个名为**libembUnit.a**的静态库文件。然后，我们只要修改相应的编译选项，去掉不用的选项，下面是修改的Makefile：

```
View Makefile

include ../build/common.mk

.c.o:
    $(CC) $(CFLAGS) -c -o $*.o $<


embu_OBJS = AssertImpl.o RepeatedTest.o stdImpl.o TestCaller.o TestCase.o
TestResult.o TestRunner.o TestSuite.o
embu_DIR = ../object/common/
TARGET = libembUnit.a

all: $(TARGET)

$(TARGET): $(embu_OBJS)
    $(AR) $(ARFLAGS) $(TARGET) $(embu_OBJS)
    cp -f $(TARGET) $(embu_DIR)

.c.o:
    $(CC) $(CFLAGS) $(INCLUDES) -c $<

AssertImpl.o: AssertImpl.h stdImpl.h
RepeatedTest.o: RepeatedTest.h Test.h
stdImpl.o: stdImpl.h
TestCaller.o: TestCaller.h TestResult.h TestListener.h TestCase.h Test.h
TestCase.o: TestCase.h TestResult.h TestListener.h Test.h
TestResult.o: TestResult.h TestListener.h Test.h
TestRunner.o: TestRunner.h TestResult.h TestListener.h Test.h stdImpl.h
config.h
TestSuite.o: TestSuite.h TestResult.h TestListener.h Test.h

clean:
    -rm -f *.o *.a

.PHONY: clean all
```

仔细分析，修改的内容就是将编译选项修改了，然后将生成的**libembUnit.a**文件复制到一个公共目录下；

2. 修改完Makefile后，如果编译，会提示stdio.h这个头文件找不到。原因是embUnit的源代码中的config.h这个文件include了<stdio.h>这个C库文件，而由于我们的操作系统完全自己实现，并且没有提供stdio.h这个头文件，因此，将其注释掉即可；然后再执行make，就可以编译通过了，虽然会有一些警告，不过可以忽略；
3. 书写测试代码

```
View Code
```

```
/* include local files */

/* include embUnit include */
#include "../embUnit/embUnit.h"
#include "../embUnit/AssertImpl.h"
#include "../embUnit/config.h"
#include "../embUnit/HelperMacro.h"
#include "../embUnit/RepeatedTest.h"
#include "../embUnit/stdImpl.h"
#include "../embUnit/Test.h"
#include "../embUnit/TestCaller.h"
#include "../embUnit/TestCase.h"
#include "../embUnit/TestListener.h"
#include "../embUnit/TestResult.h"
#include "../embUnit/TestRunner.h"
#include "../embUnit/TestSuite.h"

#include "../rts_include/test.h"

int iFile = -1;

static void setUp(void)
{
    /* initialize */
    iFile = open("testFile.txt", O_WRONLY|O_CREAT|O_APPEND|O_NAND);
}

static void tearDown(void)
{
    /* terminate */
    close(iFile);
}

static void testFile(void)
{
    char buff[4] = "abcd";
    TEST_ASSERT_EQUAL_INT(6, write(iFile, buff, 4));
}

/*embunit:impl=+ */
/*embunit:impl=- */
TestRef testFile_tests(void)
{
    EMB_UNIT_TESTFIXTURES(fixtures) {
        /*embunit:fixtures=+ */
        /*embunit:fixtures=- */
        new_TestFixture("testFile", testFile),
    };
    EMB_UNIT_TESTCALLER(test,"test",setUp,tearDown,fixtures);
    return (TestRef)&test;
};
```

首先注意测试代码一方面要引用embUnit的头文件，另一方面也要引用原有程序相应的头文件，这样才能既使用embUnit提供的API函数，又能使用原有程序提供的接口函数；

上述代码的解释如下：

- 上述代码的目标是对write函数进行测试，方法就是先open一个文件，然后写入固定的字节，判断写入的字节数是否正确，最后关闭文件。
- setUp函数的作用是：提供待测试函数的前端输入。譬如，上述代码要测试write函数，必须要先用open打开一个文件，那么就可以在setUp这个函数中调用open函数去创建要write的文件；
- tearDown函数的作用是：提供待测试函数的后端处理。譬如，上述代码，写入文件后，要关闭文件，那么就可以在tearDown这个函数调用close函数关闭文件；
- testFile函数（函数名其实可以自己定义）的作用是：提供测试逻辑。譬如，上述代码，调用embUnit提供的断言宏，进行判断write函数是否执行成功。
- testFile_tests函数的作用是：将上述几个函数整合到一个测试suite中，以供后续程序调用；

对测试代码编译（仍然采用目标平台的编译选项进行编译），使其生成的.a（如test.a）文件，并将.a文件复制到公共的库文件目录下；

4. 在原有函数的入口出执行测试程序

   代码如下：

```
View Code

#include "../embUnit/embUnit.h"
#include "../embUnit/AssertImpl.h"
#include "../embUnit/config.h"
#include "../embUnit/HelperMacro.h"
#include "../embUnit/RepeatedTest.h"
#include "../embUnit/stdImpl.h"
#include "../embUnit/Test.h"
#include "../embUnit/TestCaller.h"
#include "../embUnit/TestCase.h"
#include "../embUnit/TestListener.h"
#include "../embUnit/TestResult.h"
#include "../embUnit/TestRunner.h"
#include "../embUnit/TestSuite.h"

void main(void)
{
    printf("*************************************************************\n");
    printf("**************      Unit Test Start         **************\n");
    printf("*************************************************************\n");

    /* 将测试结果按照编译格式输出 */
    TestRunner_start();
    TestRunner_runTest(testFile_tests());
    TestRunner_end();

    printf("*************************************************************\n");
    printf("**************       Unit Test End          **************\n");
    printf("*************************************************************\n");
}
```

   在上述代码中，可以看到语句**TestRunner_runTest(testFile_tests());**调用的就是在测试代码中声明的**testFile_tests**函数；
   需要的注意的是，在该程序中也要引用embUnit提供的头文件，否则无法编译通过。

5. 将上述代码也编译成.a文件，并且与刚才编译生成的**libembUnit.a**和**test.a**文件，以及原有程序的其他.a文件链接一个可执行文件，并将其烧录到目标机器中，然后即可看到运行结果。下图是本人的运行结果：

```
**************************************************************
***************       Unit Test Start          ***************
**************************************************************
.
test.testFile (test.c 60) exp 6 was 4

run 1 failures 1
**************************************************************
***************        Unit Test End           ***************
**************************************************************
```

由于写入的是4个字节，但是测试代码的预期值是6个字节，因此，测试没有通过。

到此，整个移植过程结束。不过，上述测试结果显示，不是那么友好，很幸运的是embUnit提供的源码中还有一个格式化测试结果的工具的源代码，我们就照葫芦画瓢将这个工具也移植过来；

移植embUnit的格式化测试结果工具

在刚才下载的压缩包解压后的根目录下有个**textui**的目录，这个目录提供的就是格式化测试结果的工具的源代码，移植的方法与上面类似，现简介如下：

1. 修改makefile

   textui原有的makefile如下所示：

```
View makefile

CC = gcc
CFLAGS = -O
INCLUDES = ..
LIBS = ../lib
AR = ar
ARFLAGS = ru
RANLIB = ranlib
RM = rm
OUTPUT = ../lib/
TARGET = libtextui.a
OBJS = TextUIRunner.o XMLOutputter.o TextOutputter.o CompilerOutputter.o

all: $(TARGET)

$(TARGET): $(OBJS)
    $(AR) $(ARFLAGS) $(OUTPUT)$@ $(OBJS)
    $(RANLIB) $(OUTPUT)$@

.c.o:
    $(CC) $(CFLAGS) -I$(INCLUDES) -c $<

TextUIRunner.o: TextUIRunner.h XMLOutputter.h TextOutputter.h
CompilerOutputter.h Outputter.h
XMLOutputter.o: XMLOutputter.h Outputter.h
TextOutputter.o: TextOutputter.h Outputter.h
CompilerOutputter.o: CompilerOutputter.h Outputter.h

clean:
    -$(RM) $(TARGET) $(OBJS)

.PHONY: clean all
```

   很明显，原有的makefile是将所有的源文件编译生成一个**libtextui.a**的文件，现将这个makefile修改如下：

View Makefile

```
include ../build/common.mk

.c.o:
    $(CC) $(CFLAGS) -c -o $*.o $<

textui_OBJS = TextUIRunner.o XMLOutputter.o TextOutputter.o
CompilerOutputter.o
textui_DIR = ../object/common/
TARGET = libtextui.a

all: $(TARGET)

$(TARGET): $(textui_OBJS)
    $(AR) $(ARFLAGS) $(TARGET) $(textui_OBJS)
    cp -f $(TARGET) $(textui_DIR)

TextUIRunner.o: TextUIRunner.h XMLOutputter.h TextOutputter.h
CompilerOutputter.h Outputter.h
XMLOutputter.o: XMLOutputter.h Outputter.h
TextOutputter.o: TextOutputter.h Outputter.h
CompilerOutputter.o: CompilerOutputter.h Outputter.h

clean:
    -rm -f *.o *.a


.PHONY: clean all
```

与前述一样，只是修改的编译选项，并且将生成的**libtextui.a**文件复制到一个公共的目录下；

2. 编译textui

   修改完Makefile文件后，此时编译会提示：**stdout未定义**；

   原因是：textui中使用了fprintf将信息输出的stdout（即标准输出，一般指屏幕），但是我们的目标机不支持显示器等标准输出，也没有对stdout进行定义。

   解决办法是：在Outputter.h文件中，重新定义printf，即，忽略stdout，使用printf代替fprintf，具体如下所示：

```
#define fprintf(stdout, formats, args...)    printf(formats, ##args)
```

   解决上述问题后，编译时，仍然会提示错误，主要是头文件找不到的错误；

   原因是：在textui的源文件中，需要引用embUnit的头文件，但是其采用的是**#include <>**方式，当然找不到对应的头文件了；

   解决办法：我们将这些头文件的引用方式改为**#include " "**方式，并且指明对应头文件的相对路径；

3. 修改在程序入口处的调用测试程序的代码，具体如下所示：

View Code

```
void main(void)
{

    printf("*****************************************************************\n");
```

```
    printf("**************      Unit Test Start         **************\n");
    printf("************************************************************\n");

    /* 将测试结果按照编译格式输出 */
    TestRunner_start();
    TestRunner_runTest(testFile_tests());
    TestRunner_end();

    /* 将测试结果按文本格式输出 */
    TextUIRunner_setOutputter(TextOutputter_outputter());
    TextUIRunner_start();
    TextUIRunner_runTest(testFile_tests());
    TextUIRunner_end();

    /* 将测试结果按照XML格式输出 */
    TextUIRunner_setOutputter(XMLOutputter_outputter());
    TextUIRunner_start();
    TextUIRunner_runTest(testFile_tests());
    TextUIRunner_end();

    printf("************************************************************\n");
    printf("**************       Unit Test End          **************\n");
    printf("************************************************************\n");

}
```

从上述代码可知，注册了两种测试结果输出方式：
TextUIRunner_setOutputter(TextOutputter_outputter());
TextUIRunner_setOutputter(XMLOutputter_outputter());

4. 全编译所有代码，并链接成可执行程序，烧到目标机器上，即可看到测试结果，下图就是测试结果：

```
************************************************************
**************       Unit Test Start          **************
************************************************************
.
test.testFile (test.c 60) exp 6 was 4

run 1 failures 1
- test
1) NG testFile (test.c 60) exp 6 was 4

run 1 failures 1
<?xml version="1.0" encoding='shift_jis' standalone='yes' ?>
<TestRun>
<test>
<FailedTest id="1">
<Name>testFile</Name>
<Location>
<File>test.c</File>
<Line>60</Line>
</Location>
<Message>exp 6 was 4</Message>
</FailedTest>
</test>
<Statistics>
<Tests>1</Tests>
<Failures>1</Failures>
</Statistics>
</TestRun>
************************************************************
**************       Unit Test End            **************
************************************************************
----------------heros tcpip 1.0.0  initialize end-------------------------
```

到此，整个移植过程结束。