CppUnit CookBook 中文版

自己翻译的 cppunit cookbook，如有错漏，欢迎指出。可以在**这里**下载到 cppunit 的最新版本源码。

这只是一个 cookbook 的翻译，并没介绍安装方法，你可以在**这里**找到 win32 下的安装方法和例子。不过，这个例子并不清楚，还是建议你看看**这里**的例子，清楚的多。看完这些安装方法和例子之后，再回头看看这篇 cookbook，应该会帮助你理解例子里面那些代码的含义。

1、简单的测试案例

怎么才能知道你的代码是不是能够正常工作？有很多方法可以达到这个目的。通过调试器单步跟踪，或者在你的代码里加入打印输出代码是两个比较简单的办法，但是这两个方法都有缺点。单步跟踪不能自动进行，每次代码稍有调整就要进行调试。打印输出也不错，只是这种方法会增加很多不必要的代码，导致代码臃肿丑陋。

CppUnit 单元测试很容易建立起来，并且可以自动进行，而且，一旦你写完测试用例，就能通过它们保证你的代码质量。

按照下面的流程可以构造一个简单的 test：

1、继承 CppUnit::TestCase 类。

2、重写 runTest()方法。

3、使用 CPPUNIT_ASSERT()和 CPPUNIT_ASSERT(bool)两个宏来检测表达式或值，以判断测试成功与否。

举个例子，如果要测试一个复数类的赋值(=号)运算符，按照上面的步骤，代码如下：

```
01  class ComplexNumberTest : public CppUnit::TestCase
02  {
03  public:
04      ComplexNumberTest( std::string name ) : CppUnit::TestCase( name ) {}
05      void runTest()
06      {
07          CPPUNIT_ASSERT( Complex (10, 1) == Complex (10, 1) );
08          CPPUNIT_ASSERT( !(Complex (1, 1) == Complex (2, 2)) );
09      }
10  };
```

一个简单的 test 就建起来了。但通常来说，我们会在同一个对象里面有很多小的测试用例。这种情况下，我们用 fixture。

2、fixture

fixture 是为一组测试用例提供基础服务的对象，当你边开发边测试时，使用 fixture 非

常方便。我们试着模拟一下这种边开发边测试情况。

假设我们真的在开发一个复数类，首先，定义一个空的 Complex 类：

```
1    class Complex {};
```

现在，创建一个上面的 ComplexNumberTest 类对象，然后编译代码看看会发生什么。我们会得到几个编译错误。因为测试过程中用到了==操作符，但我们并没定义这个运算符。

现在为 Complex 类定义一个：

```
1    bool operator==( const Complex &amp;amp;a, const Complex &amp;amp;b)
2    {
3        return true;
4    }
```

再次编译运行这个测试。这次编译虽然通过了，但测试却是失败的。

要再做一点点事情让==操作符正常工作，重写代码如下：

```
01   class Complex
02   {
03       friend bool operator ==(const Complex&amp;amp; a, const Complex&amp;amp; b);
04       double real, imaginary;
05   public:
06       Complex( double r, double i = 0 ) :real(r) , imaginary(i) {}
07   };
08   bool operator ==( const Complex &amp;amp;a, const Complex &amp;amp;b )
09   {
10       return a.real == b.real   &amp;amp;&amp;amp;   a.imaginary == b.imaginary;
11   }
```

编译运行，测试顺利通过。

好了，现在我们准备增加一些新的操作和新的测试。这个时候，fixture 就体现出方便性了。因为在这个时候，如果使用 fixture 实例化三四个 Complex 对象，并且在测试过程中复用这几个对象，这样不需要重复构建这些实例对象，测试代码会更加好写。

fixture 的使用步骤如下：

- 给每个 fixture 添加成员变量；
- 重写 CppUnit::TestFixture::setUp()以初始化这些变量；
- 重写 CppUnit::TestFixture::tearDown()方法释放 setUp 申请的系统资源。

```
01   class ComplexNumberTest : public CppUnit::TestFixture
02   {
03   private:
```

```
04          Complex *m_10_1, *m_1_1, *m_11_2;
05    public:
06          void setUp()
07          {
08                m_10_1 = new Complex( 10, 1 );
09                m_1_1 = new Complex( 1, 1 );
10                m_11_2 = new Complex( 11, 2 );
11          }
12          void tearDown()
13          {
14                delete m_10_1;
15                delete m_1_1;
16                delete m_11_2;
17           }
18    };
```

有了这个 fixture 之后，我们就能在开发过程中添加另外的测试用例和我们需要的其他东西了。

3、Test Case

怎么用 fixture 编写和调用单独的测试呢？

可以分两步完成：

- 把要调用的测试写成 fixture 类的一个方法；
- 创建一个运行指定方法的 TestCaller

下面是一个有了测试方法的 test case：

```
01    class ComplexNumberTest : public CppUnit::TestFixture
02    {
03    private:
04          Complex *m_10_1, *m_1_1, *m_11_2;
05    public:
06          void setUp()
07          {
08                m_10_1 = new Complex( 10, 1 );
09                m_1_1 = new Complex( 1, 1 );
10                m_11_2 = new Complex( 11, 2 );
11          }
```

```
12          void tearDown()
13          {
14                  delete m_10_1;
15                  delete m_1_1;
16                  delete m_11_2;
17          }
18          void testEquality()
19          {
20                  CPPUNIT_ASSERT( *m_10_1 == *m_10_1 );
21                  CPPUNIT_ASSERT( !(*m_10_1 == *m_11_2) );
22          }
23          void testAddition()
24          {
25                  CPPUNIT_ASSERT( *m_10_1 + *m_1_1 == *m_11_2 );
26          }
27  };
```

可以像下面这样创建并调用每个 test case:

```
1   CppUnit::TestCaller&lt;ComplexNumberTest&gt;          test(&quot;testEquality&quot;,
&amp;ComplexNumberTest::testEquality );
2   CppUnit::TestResult result;
3   test.run( &amp;result );
```

TestCaller 构造函数的第二个参数是 ComplexNumberTest 类中某个方法的地址。当 TestCaller 运行的时候，指定的方法也就运行了。

但是，这些都是没用的，因为没有显示判断结果。通常，我们使用 TestRunner 来显示测试结果。(后面会讲到)

当你有很多测试用例的时候，可以把它们组织成 test suite。

4、Suite

我们在上面已经知道如何运行单个的 test case，那怎么才能让所有的 Test Case 一次性全部运行起来呢？

CppUnit 提供了一个 TestSuite，可以把很多 Test Case 组织在一起运行。

可以按照下面的方式把两三个 test 组织成一个 test suite：

```
1   CppUnit::TestSuite suite;
2   CppUnit::TestResult result;
3   suite.addTest(                                                    new
```

```
CppUnit::TestCaller&lt;ComplexNumberTest&gt;(&quot;testEquality&quot;,&amp;ComplexNumberTest::testEquality ) );
    4    suite.addTest( new CppUnit::TestCaller&lt;ComplexNumberTest&gt;(&quot;testAddition&quot;, &amp;ComplexNumberTest::testAddition ) );
    5    suite.run( &amp;result );
```

TestSuites 不仅仅可以包含 TestCases 的 TestCaller，它还能包含任何实现了 CppUnit::Test 接口的对象。例如，你可以在你的代码里面创建一个 TestSuite，我也可以在我的代码里面创建一个 TestSuite，我们可以创建一个包含这两个 TestSuite 的新的 TestSuite，然后你我两个 TestSuite 就可以一起执行了。

```
1    CppUnit::TestSuite suite;
2    CppUnit::TestResult result;
3    suite.addTest( ComplexNumberTest::suite() );
4    suite.addTest( SurrealNumberTest::suite() );
5    suite.run( &amp;result );
```

5、TestRunner

怎么运行测试用例并收集结果？

有了一个 TestSuite，你肯定想去运行它。CppUnit 提供了运行 suite 并显示结果的工具-TestRunner。在你的 suite 中添加一个静态方法，获取 test suite，这样你的 suite 就能够被 TestRunner 调用了。

例如，要使 TestRunner 能够调用 ComplexNumberTest 这个 test suite，可以在 ComplexNumberTest 中添加如下代码：

```
1    public:
2    static CppUnit::TestSuite *suite()
3    {
4        CppUnit::TestSuite *suiteOfTests = new CppUnit::TestSuite( &quot;ComplexNumberTest&quot; );
5        suiteOfTests-&gt;addTest( new CppUnit::TestCaller&lt;ComplexNumberTest&gt;(&quot;testEquality&quot;, &amp;ComplexNumberTest::testEquality ) );
6        suiteOfTests-&gt;addTest( new CppUnit::TestCaller&lt;ComplexNumberTest&gt;(&quot;testAddition&quot;, &amp;ComplexNumberTest::testAddition ) );
7        return suiteOfTests;
```

```
8    }
```

下面是使用 test runner 调用 test suite 的代码，以文本界面的 test runner 为例。

要使用文本界面的 TestRunner，请在 main.cpp 中包含头文件：

```
1    #include &lt;cppunit/ui/text/TestRunner.h&gt;
2    #include &quot;ExampleTestCase.h&quot;
3    #include &quot;ComplexNumberTest.h&quot;
```

并且在 main 函数中添加调用 addTest 方法的代码：

```
1    int main( int argc, char **argv)
2    {
3        CppUnit::TextUi::TestRunner runner;
4        runner.addTest( ExampleTestCase::suite() );
5        runner.addTest( ComplexNumberTest::suite() );
6        runner.run();
7        return 0;
8    }
```

这样，TestRunner 就会运行所有的测试了。如果过程中有任何一个测试失败，会得到下面的信息：

- 失败测试的名字；
- 失败测试源文件名；
- 失败测试失败的行数；
- 检测到测试失败的 CPPUNIT_ASSERT 宏内部的所有文本信息。

CppUnit 区分两种错误，failure 和 error。failure 是框架预期的结果，会被断言宏检测到。而 error 则是非预期的，比如除 0，或者其他 C++运行时抛出的异常，你自己的代码抛出的异常等，这些不是框架预期的错误。

6、帮助宏

你可能已经注意到了，实现 fixture 的静态 suite()方法是个重复且容易出错的工作。CppUnit 创建了一系列 WritingTestFixture 宏可以帮助你自动实现静态的 suite()方法。

下面是用这些宏重写之后的 ComplexNumberTest 类代码：

```
01   #include &lt;cppunit/extensions/HelperMacros.h&gt;
02
03   class ComplexNumberTest : public CppUnit::TestFixture
04   {
05       //首先，我们声明 suite，把类名传给宏
06       CPPUNIT_TEST_SUITE( ComplexNumberTest );
```

```
07      //然后，我们声明 fixture 的每个 test case，即添加测试函数到 suite 中
08      CPPUNIT_TEST( testEquality );
09      CPPUNIT_TEST( testAddition );
10      //最后，结束 suite 声明
11      CPPUNIT_TEST_SUITE_END();
12      //这个时候，static CppUnit::TestSuite *suite();这个签名的方法已经实现了
13
14      //剩下的代码原样保留
15  private:
16      Complex *m_10_1, *m_1_1, *m_11_2;
17  public:
18      void setUp()
19      {
20          m_10_1 = new Complex( 10, 1 );
21          m_1_1 = new Complex( 1, 1 );
22          m_11_2 = new Complex( 11, 2 );
23      }
24
25      void tearDown()
26      {
27          delete m_10_1;
28          delete m_1_1;
29          delete m_11_2;
30       }
31
32       void testEquality()
33       {
34           CPPUNIT_ASSERT( *m_10_1 == *m_10_1 );
35           CPPUNIT_ASSERT( !(*m_10_1 == *m_11_2) );
36       }
37       void testAddition()
38       {
39           CPPUNIT_ASSERT( *m_10_1 + *m_1_1 == *m_11_2 );
40       }
```

```
41    };
```

添加到 suite 中的 TestCaller 名称是由 fixture 名称和方法名符合而成的。

例如，在当前的 test case 中，TestCaller 的名称是："ComplexNumberTest.testEquality"和"ComplexNumberTest.testAddition"。

帮助宏能够帮你编写普通的断言。例如，检查 ComplexNumber 中发生除 0 异常时，是否抛出 MathException 异常：

- 使用 CPPUNIT_TEST_EXCEPTION 宏，指定预期的异常类型，编写测试并添加到 suite 中
- 在 fixture 中编写测试方法

```
01    CPPUNIT_TEST_SUITE( ComplexNumberTest );
02    // [...]
03    CPPUNIT_TEST_EXCEPTION( testDivideByZeroThrows, MathException );
04    CPPUNIT_TEST_SUITE_END();
05    // [...]
06    void testDivideByZeroThrows()
07    {
08          // 下面的代码会抛出 MathException.
09          *m_10_1 / ComplexNumber(0);
10    }
```

若没抛出预期的异常，则会报告一个 assert failure。

7、TestFactoryRegistry

TestFactoryRegistry 解决了两个麻烦：

- 忘记向 test runner 中添加 fixture（由于两者不在同一个文件中，确实很容易忘记）
- 包含所有测试头文件的麻烦

TestFactoryRegistry 是程序初始化时注册 test suites 的地方。

在 cpp 文件中添加下面的代码，注册 ComplexNumber suite：

```
1     #include &lt;cppunit/extensions/HelperMacros.h&gt;
2
3     CPPUNIT_TEST_SUITE_REGISTRATION( ComplexNumberTest );
```

在这句代码背后，声明了一个静态的 CppUnit::AutoRegisterSuite 变量。在构造函数中，它调用 CppUnit::TestFactoryRegistry::registerFactory(TestFactory*)方法注册了一个 CppUnit::TestSuiteFactory 到 CppUnit::TestFactoryRegistry 中。TestSuiteFactory 通过 ComplexNumberTest::suite()(译者注：原文为 ComplexNumber::suite()，疑有误，特改之)方法返回一个 TestSuite。

使用文本模式的 test runner 运行测试，此时不再需要包含 fixture：

```
01  #include &lt;cppunit/extensions/TestFactoryRegistry.h&gt;
02  #include &lt;cppunit/ui/text/TestRunner.h&gt;
03
04  int main( int argc, char **argv)
05  {
06      CppUnit::TextUi::TestRunner runner;
07      //首先，查找的 CppUnit::TestFactoryRegistry 的实例
08                          CppUnit::TestFactoryRegistry  &amp;registry  =
CppUnit::TestFactoryRegistry::getRegistry();
09      //CppUnit::TestFactoryRegistry 中包含所有使用
10      //CPPUNIT_TEST_SUITE_REGISTRATION()宏注册的测试用例，
11      //所以，可以通过 TestFactoryRegistry 获取一个 CppUnit::TestSuite，并将其添加到 test
//runner 中
12      runner.addTest( registry.makeTest() );
13      runner.run();
14      return 0;
15  }
```

8、生成后测试

到现在为止，我们的单元测试可以正常运行了，可是怎样将单元测试整合到我们的构建过程中呢？

首先，应用程序必须返回一个不等于 0 的值，以表明有错误发生。

而 TestRunner::run()方法返回一个布尔值，用来表示测试是否成功。

可以修改主程序如下：

```
01  #include &lt;cppunit/extensions/TestFactoryRegistry.h&gt;
02  #include &lt;cppunit/ui/text/TestRunner.h&gt;
03
04  int main( int argc, char **argv)
05  {
06      CppUnit::TextUi::TestRunner runner;
07                          CppUnit::TestFactoryRegistry  &amp;registry  =
CppUnit::TestFactoryRegistry::getRegistry();
08      runner.addTest( registry.makeTest() );
09      bool wasSuccessful = runner.run( &quot;&quot;, false );
```

```
10          return !wasSuccessful;
11      }
```

现在，编译后运行程序吧。

在 VC++中 ,可以设置生成事件->生成后事件 ,指定命令行参数为“$(TargetPath)”达到将单元测试整合进构建过程中的目的。”$(TargetPath)”将会被展开为此测试工程的可执行文件路径，所以，当编译生成完成后，会自动运行 main.exe 完成测试。这样就完成了单元测试与构建过程的绑定。