# 1 安装

TestNG++使用[cmake](http://www.cmake.org/)作为构建系统。因此，为了构建TestNG++，你需要确保cmake 2.6以上版本已经安装在你的系统上。

到TestNG++的[下载页面](http://code.google.com/p/test-ng-pp/downloads/list)下载最新的发布版本。尽管你可以直接从源代码库checkout最新的源代码，但它的稳定性无法得到保证。

## 1.1 Linux/Mac OS

随后将源代码包解压到某个目录。假社在Linux平台上，我们解压的目录是：/home/arthur/testngpp. ``` arthur> cd /home/arthur/testngpp

arthur> mkdir build

arthur> cd build ```

然后运行cmake，此时，你可以通过变量 **CMAKE\_INSTALL\_PREFIX** 指定 TestNG++的安装路径。如果你不指定，则默认的安装路径是/usr/local。

假设，我们想把TestNG++安装到目录/home/arthur/myproject : ```

arthur> cmake -DCMAKE\_INSTALL\_PREFIX=/home/arthur/myproject ..

-- The C compiler identification is GNU -- The CXX compiler identification is GNU -- Checking whether C compiler has -isysroot -- Checking whether C compiler has -isysroot - yes -- Check for working C compiler: /usr/bin/gcc -- Check for working C compiler: /usr/bin/gcc -- works -- Detecting C compiler ABI info -- Detecting C compiler ABI info - done -- Checking whether CXX compiler has -isysroot -- Checking whether CXX compiler has -isysroot - yes -- Check for working CXX compiler: /usr/bin/c++ -- Check for working CXX compiler: /usr/bin/c++ -- works -- Detecting CXX compiler ABI info -- Detecting CXX compiler ABI info - done -- Found PythonInterp: /usr/bin/python2.5 -- Configuring done -- Generating done -- Build files have been written to: /home/arthur/testngpp/build

arthur> make

```

在成功构建之后。执行:

arthur> make install

然后，在安装目录/home/arthur/myproject，你可以看到一个这样的目录结构：

```

|- include |- bin |- lib |- testngpp |- generator |- listener

```

下表列出了各个目录所放置的内容:

| 目录 | 内容 | |:---|:---| | include | 测试用例需要包含的头文件 | | bin | Test Runner等可执行文件 | | lib | 测试模块需要链接的库文件 | | testngpp/generator | 测试代码产生器 | | testngpp/listener | 测试结果监听器 |

## 1.2 Windows

Windows的安装和配置请参考文档[TestNG++ Installation & Configuration Guide for MSVC Users](http://test-ng-pp.googlecode.com/files/testngpp-msvc.pdf)

# 2 编写测试

在你编写测试之前，你需要配置并安装TestNG++。

## 2.1 测试文件

为了使用TestNG++，你首先要创建放置测试代码的源代码文件，尽管并不是强制的约束，但建议使用".h"作为这些文件的扩展名。

比如，我们要测试的类名为DLModuleLoader，所以，我们可以将我们的测试文件命名为TestDLModuleLoader.h。

## 2.2 头文件

然后，你需要包含TestNG++的头文件 ***testngpp.hpp*** ；像这样： ```

# include

```

为了能够让编译通过，不要忘记把 **TestNG++** 的安装路径的include放到编译选项里。假如 **TestNG++** 的头文件被安装到/home/arthur/myproject下，则你需要确保/home/arthur/myproject/include被指定在你的编译选项里。对于windows用户而言，假设TestNG++被安装在c:\testngpp，则你需要将路径c:\testngpp\include放到你的编译选项里。

## 2.3 FIXTURE

下面，我们需要定义一个 **fixture**:

```

# include

# include "DLModuleLoader.h"

FIXTURE(DLModuleLoader) { };

```

**FIXTURE()** 的第一个参数为 **Fixture ID** ，它可以是任何C/C++标识符（Identifier）。一般而言，将其命名为 **CUT** (Class Under Test)的名字即可。

**FIXTURE()** 还有第二个可选参数：**Fixture Name** 。它可以是任何\_宏参数\_所允许的字符。如果你的编译器支持多语言，你可以使用任何其它字符集所支持的字符。比如：

```

# include

# include "DLModuleLoader.h"

FIXTURE(DLModuleLoader, 类DLModuleLoader的测试, 你可以在fixture name中使用这些字符：(){}[]''~!@#$%^&\*+-=\_?><"") { };

```

**FIXTURE** 本身是一个宏，所以要求\*fixture name\*\*必须遵守宏参数的语法规则，你可以在名字里使用() ' "等符号，但它们必须正确的成对出现，否则编译器将会报错。\*\*

如果你没有提供 **fixture name** ，则 **fixture ID** 将被当作 **fixture name** 。

## 2.4 TEST

在定义了fixture之后，你可以编写你的第一个测试。如下：

```

# include

# include "DLModuleLoader.h"

FIXTURE(DLModuleLoader) { TEST(should be able to load modules whose name does not have suffix ".so") { DLModuleLoader\* loader = new DLModuleLoader();

ASSERT\_THROWS\_NOTHING(loader->load("libMySuite"));

delete loader;

} };

```

**TEST()** 用来定义一个测试用例。括号里面的部分是 **test name** ，和 **fixture name** 一样，它可以使用你的编译器和所允许的任何字符。并且，同样也要遵守宏参数的语法约束。

比如，我可以将上面的测试用例用中文来命名： ```

# include

# include "DLModuleLoader.h"

FIXTURE(DLModuleLoader) { TEST(可以在不需要指明后缀".so"的情况下加载一个模块) { DLModuleLoader\* loader = new DLModuleLoader();

ASSERT\_THROWS\_NOTHING(loader->load("libMySuite"));

delete loader;

} };

```

## 2.5 Assertion

测试用例里面的 **ASSERT\_THROWS\_NOTHING** 是一个测试断言，如果断言失败，则此测试用例将会失败。

TestNG++提供的测试断言如下：

| 断言 | 用法 | 例子 | |:---|:---|:---| | ASSERT\_TRUE(expr) | 断言表达式的求值为true | ASSERT\_TRUE(10 == a) | | ASSERT\_FALSE(expr) | 断言表达式的求值为false | ASSERT\_FALSE(10 != a) | | ASSERT\_EQ(expected, value) | 断言两个表达式相等 | ASSERT\_EQ(10, a) | | ASSERT\_NE(expected, value) | 断言两个表达式不相等 | ASSERT\_NE(10, a) | | ASSERT\_SAME\_DATA(addr1, addr2, size) | 断言两个地址所指向的内存在size范围内完全相同 | ASSERT\_SAME\_DATA(p1, p2, 64) | | ASSERT\_DELTA(x, y, delta) | 断言表达式的差值应该在delta的范围之内 | ASSERT\_DELTA(9.91, 10, 0.1) | | ASSERT\_THROWS(expr, except) | 断言表达式将会抛出except类型的异常 | ASSERT\_THROWS(foo(), std::exception) | | ASSERT\_THROWS\_ANYTHING(expr) | 断言表达式将会抛出异常 | ASSERT\_THROWS\_ANYTHING(foo()) | | ASSERT\_THROWS\_NOTHING(expr) | 断言expr不会抛出任何异常 | ASSERT\_THROWS\_NOTHING(foo()) | | ASSERT\_THROWS\_EQUALS(expr, except, expected, value) | 断言表达式将会抛出except类型的异常，并且两个表达式相等 | ASSERT\_THROWS\_EQUALS(foo(), std::exception& e, "bad alloc", e.what()) |

这些断言的都遵从了 **"快速失败"** 的原则，即一旦断言失败，整个用例将中止执行。

### 2.5.1 EXPECT vs. ASSERT

**”快速失败“** 是个好的原则，但对于C++这种需要程序员自己进行资源管理的语言而言，有时候快速失败可能会带来问题。比如，对于这个例子，如果断言失败将会造成内存泄露。

``` TEST(可以在不需要指明后缀".so"的情况下加载一个模块) { DLModuleLoader\* loader = new DLModuleLoader();

ASSERT\_THROWS\_NOTHING(loader->load("libMySuite"));

delete loader; // 如果上面的断言失败，此语句将得不到执行

} ```

所以，TestNG++提供了另外一套Assertion，以让断言失败后，测试能够继续得以执行。这些Assertion和之前的ASSERT断言一一对应，但以EXPECT开头：

| 断言 | 用法 | 例子 | |:---|:---|:---| | EXPECT\_TRUE(expr) | 断言表达式的求值为true | EXPECT\_TRUE(10 == a) | | EXPECT\_FALSE(expr) | 断言表达式的求值为false | EXPECT\_FALSE(10 != a) | | EXPECT\_EQ(expected, value) | 断言两个表达式相等 | EXPECT\_EQ(10, a) | | EXPECT\_NE(expected, value) | 断言两个表达式不相等 | EXPECT\_NE(10, a) | | EXPECT\_SAME\_DATA(addr1, addr2, size) | 断言两个地址所指向的内存在size范围内完全相同 | EXPECT\_SAME\_DATA(p1, p2, 64) | | EXPECT\_DELTA(x, y, delta) | 断言表达式的差值应该在delta的范围之内 | EXPECT\_DELTA(9.91, 10, 0.1) | | EXPECT\_THROWS(expr, except) | 断言表达式将会抛出except类型的异常 | EXPECT\_THROWS(foo(), std::exception) | | EXPECT\_THROWS\_ANYTHING(expr) | 断言表达式将会抛出异常 | EXPECT\_THROWS\_ANYTHING(foo()) | | EXPECT\_THROWS\_NOTHING(expr) | 断言expr不会抛出任何异常 | EXPECT\_THROWS\_NOTHING(foo()) | | EXPECT\_THROWS\_EQUALS(expr, except, expected, value) | 断言表达式将会抛出except类型的异常，并且两个表达式相等 | EXPECT\_THROWS\_EQUALS(foo(), std::exception& e, "bad alloc", e.what()) |

这样，上面用例就可以改写为：

``` TEST(可以在不需要指明后缀".so"的情况下加载一个模块) { DLModuleLoader\* loader = new DLModuleLoader();

EXPECT\_THROWS\_NOTHING(loader->load("libMySuite"));

delete loader; // 即使上面的断言失败，此语句仍将得到执行

} ```

对于这个例子，EXPECT工作的很好。但在另外一些情况下，它将会导致晦涩复杂的测试。比如：

``` TEST(应该能够根据名字找到相应的test case) { DLModuleLoader\* loader = new DLModuleLoader();

Suite\* suite = 0;

EXPECT\_THROW\_NOTHING(suite = loader->load("libMySuite"));

EXPECT\_TRUE( 0 != suite);

if(suite != 0)

{

Fixture\* fixture = suite->findFixture("MyFixture");

EXPECT\_TRUE(0 != fixture);

if(fixture != 0)

{

TestCase\* testcase = fixture->findTestCase("MyTestCase");

EXPECT\_TRUE(0 != testcase);

EXPECT\_EQ("MyTestCase", testcase->getName());

}

}

delete loader;

} ```

### 2.5.2 用例级别的资源管理

TestNG++提供了另外一种机制来帮助编写简单的测试。

``` TEST(应该能够根据名字找到相应的test case) { DLModuleLoader\* loader = new DLModuleLoader();

\_\_DO\_\_ // 从这里开始是可能失败的部分

Suite\* suite = 0;

ASSERT\_THROW\_NOTHING(suite = loader->load("libMySuite"));

ASSERT\_TRUE( 0 != suite );

Fixture\* fixture = suite->findFixture("MyFixture");

ASSERT\_TRUE( 0 != fixture );

TestCase\* testcase = fixture->findTestCase("MyTestCase");

ASSERT\_TRUE( 0 != testcase );

ASSERT\_EQ("MyTestCase", testcase->getName());

\_\_CLEANUP\_\_ // 无论\_\_DO\_\_里面的断言失败与否，\_\_CLEANUP\_\_里面的代码都会得到执行

delete loader;

\_\_DONE\_\_ // 不要忘了写它

} ```

由于\_\_DO\_\_里面的每一个断言都用的是ASSERT——“快速失败”断言，所以任何一个断言失败都会导致\_\_DO\_\_的部分立即中止，但\_\_CLEANUP\_\_的部分却无论成功与失败都肯定会得到执行。这种机制也可以被理解为用例级别的setup/teardown。

### 2.5.3 特别的Assertion

另外，TestNG++还提供了另外几个Assertion用于特殊的目的。

| 断言 | 用法 | 例子 | |:---|:---|:---| | FAIL(str) | 让用例立即失败，并给出失败的原因 | FAIL("not implemented yet") | | WARN(str) | 给出一个警告，但用例并不失败 | WARN(“got a non-fatal null pointer.”) | | INFO(str) | 给出一个信息，用例并不失败 | INFO(“resource has been released.”) |

## 2.6 SETUP和TEARDOWN

在测试成功运行之后，我们来编写第二个用例。

```

# include

# include "DLModuleLoader.h"

FIXTURE(DLModuleLoader) { TEST(可以在不需要指明后缀".so"的情况下加载一个模块) { DLModuleLoader\* loader = new DLModuleLoader();

ASSERT\_THROWS\_NOTHING(loader->load("libMySuite"));

delete loader;

}

TEST(可以在指明后缀".so"的情况下加载一个模块) { DLModuleLoader\* loader = new DLModuleLoader();

ASSERT\_THROWS\_NOTHING(loader->load("libMySuite.so"));

delete loader;

} };

```

然后我们发现，两个用例都需要使用一个loader实例，所以我们把这个实例定义移到fixture里面，并使用fixture的 **SETUP** 和 **TEARDOWN** ：

```

# include

# include "DLModuleLoader.h"

FIXTURE(DLModuleLoader) { DLModuleLoader\* loader;

SETUP() { loader = new DLModuleLoader(); }

TEARDOWN() { delete loader; }

TEST(可以在不需要指明后缀".so"的情况下加载一个模块) { ASSERT\_THROWS\_NOTHING(loader->load("libMySuite")); }

TEST(可以在指明后缀".so"的情况下加载一个模块) { ASSERT\_THROWS\_NOTHING(loader->load("libMySuite.so")); } };

```

像任何正确实现的xUnit框架一样，TestNG++会为每个测试用例创建一个Fixture实例，并执行其 **SETUP** 和 **TEARDOWN** 。对于本例中的两个用例，其执行过程分别如下：

1. **new** FixtureDLModuleLoader()
2. **SETUP**()
3. **TEST**(可以在不需要指明后缀".so"的情况下加载一个模块)
4. **TEARDOWN**()
5. **delete** FixtureDLModuleLoader;
6. **new** FixtureDLModuleLoader()
7. **SETUP**()
8. **TEST**(可以在指明后缀".so"的情况下加载一个模块)
9. **TEARDOWN**()
10. **delete** FixtureDLModuleLoader;

另外，可以保证的是，无论测试失败与否，**TEARDOWN** 都肯定会得到执行。

## 2.7 Annotation

C/C++语言本身并没有Annotation的机制，但TestNG++利用C/C++的注释实现了类Annotation机制。Test Generator的解析器会来解释这些Annotation，并生成相应的测试代码框架。

### 2.7.1 语法

最简单的annotation的语法为: // @ annotation

当前TestNG++支持的annotation有两种：

// @ test void shouldBeAbleToParseFile() { }

annotation **"test"** 用来标示下面的方法是一个测试用例。

``` // @fixture struct TestCFoo : public TESTNGPP\_NS::TestFixture { ... };

```

annotation **"fixture"** 用来标示下面的类是一个Test Fixture。

注意：annotation前面的单行注释符号只能是//。其它形式的注释，比如///，////，/\* \*/等都被认为是普通的注释。

Annotation还可以有可选的 **key=value** list；它们需要遵从如下规则： \* **key=value** 需要放在小括号里面。 \* 如果存在多个 **key=value**, 则它们之间应该用逗号隔开; \* value可以用双引号引起来; \* 如果value的内容存在空格，必须用双引号引起来。

比如：

// @test (id=startup, tags="ut it")

### 2.7.2 测试依赖

随后我们编写针对unload的测试，传统的做法如下：

```

# include

# include "DLModuleLoader.h"

FIXTURE(DLModuleLoader) { DLModuleLoader\* loader;

SETUP() { loader = new DLModuleLoader(); }

TEARDOWN() { delete loader; }

TEST(可以在不需要指明后缀".so"的情况下加载一个模块) { ASSERT\_THROWS\_NOTHING(loader->load("libMySuite")); }

TEST(可以在指明后缀".so"的情况下加载一个模块) { ASSERT\_THROWS\_NOTHING(loader->load("libMySuite.so")); }

TEST(可以在加载后卸载模块) { ASSERT\_THROWS\_NOTHING(loader->load("libMySuite.so")); ASSERT\_THROWS\_NOTHING(loader->unload()); } };

```

由于卸载的前提是先加载模块，而加载模块的测试我们已经编写过，所以我们不想重复这段代码。另外一个选择是：使用测试依赖。改写结果如下：

```

# include

# include "DLModuleLoader.h"

FIXTURE(DLModuleLoader) { DLModuleLoader\* loader;

SETUP() { loader = new DLModuleLoader(); }

TEARDOWN() { delete loader; }

TEST(可以在不需要指明后缀".so"的情况下加载一个模块) { ASSERT\_THROWS\_NOTHING(loader->load("libMySuite")); }

// @ test (id=load) TEST(可以在指明后缀".so"的情况下加载一个模块) { ASSERT\_THROWS\_NOTHING(loader->load("libMySuite.so")); }

// @ test (depends=load) TEST(可以在加载后卸载模块) { ASSERT\_THROWS\_NOTHING(loader->unload()); } };

```

通过annotation **test** 的key **"id"** 来标注加载测试用例，然后用key **depends** 来引入依赖。

引入测试依赖后，最后一个测试用例的执行过程如下：

1. **new** FixtureDLModuleLoader()
2. **SETUP**()
3. **TEST**(可以在指明后缀".so"的情况下加载一个模块)
4. **TEST**(可以在加载后卸载模块)
5. **TEARDOWN**()
6. **delete** FixtureDLModuleLoader;

在这个过程中，即便 **TEST(可以在指明后缀".so"的情况下加载一个模块)** 里面的断言失败，TestNG++仍然认为是用例 **TEST(可以在加载后卸载模块)** 失败。

### 2.7.3 Tags

Tags是给测试做标记的一种方式。

## 2.8 兼容性

### 2.8.1 cxxtest

TestNG++完全兼容cxxtest，这意味着你可以不用做任何修改，而使用TestNG++来生成，编译，和运行你之前用cxxtest所编写的所有用例：

```

# include

class TestFoo : public CxxTest::TestSuite { CFoo\* foo;

public:

void setUp()

{

foo = new CFoo();

}

void tearDown()

{

delete foo;

}

void test\_should\_be\_able\_to\_add\_2\_integers()

{

TS\_ASSERT\_EQUAULS(4, foo->add(1, 3));

}

};

```

另外，TestNG++消除了cxxtest的一些限制：

1. 你可以在测试代码中自由的使用多行注释（cxxtest不支持用多行注释来注释掉一个用例）
2. 你可以使用#if 0...#endif来diable一个用例（cxxtest不支持，即使你用了它，cxxtest仍然认为被diable的用例是存在的）
3. 你可以使用任何宏控制块 (cxxtest不识别宏控制)

另外，你可以使用annotation: 1. 你可以使用annotation来标注一个测试，这样，你的用例名就不必要以test开始。 1. 你可以使用annotation的所有特性，比如设定测试依赖，指定tags等等.

```

# include

class TestFoo : public CxxTest::TestSuite { CFoo\* foo;

public:

void setUp()

{

foo = new CFoo();

}

void tearDown()

{

delete foo;

}

// @test

void should\_be\_able\_to\_add\_2\_integers()

{

TS\_ASSERT\_EQUAULS(4, foo->add(1, 3));

}

};

```

### 2.8.2 testcpp

除了ResourceCheckPoint之外，TestNG++完全兼容testcpp。

对于ResourceCheckPoint，TestNG++给了一个空的实现。这意味这你的testcpp用例无需做任何修改，使用TestNG++来生成，构建和运行你的testcpp用例。

```

# include

class TestFoo : public TESTCPP\_NS::TestFixture { CFoo\* foo;

public:

void setUp()

{

foo = new CFoo();

}

void tearDown()

{

delete foo;

}

void test\_should\_be\_able\_to\_add\_2\_integers()

{

TS\_ASSERT\_EQUAULS(4, foo->add(1, 3));

}

};

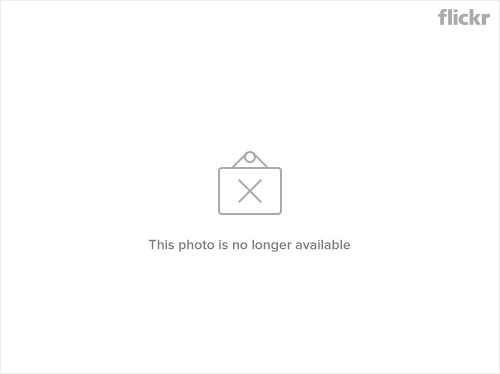
```

另外，testcpp的一些限制也被解除： 1. 你可以使用任何宏控制块 (testcpp不识别宏控制)

你也可以使用annotation: 1. 用annotation来标注一个测试，这样，你的用例名就不必要以test开始。 1. 用annotation的所有特性，比如设定测试依赖，指定tags等等.

# 3. 构建

## 3.1 Test Suite



## 3.2 Test Generator

由于C/C++缺乏反射机制，实现简单的用户界面相对于Java，C#等现代语言更加的困难。为了做到这一点，TestNG++使用\*Test Generator\*\*来扫描用户编写的测试文件，并生成相应的框架代码。\*\*

**Test Generator** 的实现语言是python，在编译和安装之后，**Test Generator** 的python字节码被安装到**${CMAKE\_INSTALL\_PREFIX}/testngpp/generator** 下，其中${CMAKE\_INSTALL\_PREFIX}时TestNG++的安装路径。

**Test Generator** 可以将指定的一个或多个用户编写的测试文件生成一个测试套件源代码文件。

假如TestNG++被安装在/usr/local/testngpp，可以通过下面的命令行来调用\*Test Generator\*\*： python /usr/local/testngpp/testngpp/generator/testngppgen.pyc -e gb2312 -o MyTestSuite.cpp Test1.h Test2.h \*\*

此命令行将会把Test1.h和Test2.h文件里所有的Fixture都生成到测试套件MyTestSuite里面。

testngppgen.pyc的选项有：

| 选项 | 说明 | |:---|:---| | -e | 指定测试文件的编码格式，默认值为utf-8 | | -o | 指定生成的测试套件源文件名 |

## 3.3 编译和链接

当一个测试套件的源代码文件生成之后，就可以进行编译。

TestNG++的运行时机制要求，每个测试套件源代码文件需要被编程和链接成一个 **module** 。在Linux下，一个 **module** 是一个所有符号都得到充分链接的\*共享库(Shared Object File)**；在Mac OS下，一个\*\*module**对应一个**bundle**，而在windows上，一个**module**则是一个**动态链接库(Dynamic Link Library)**。\*\*

### 3.3.1 Linux

在Linux上，如果使用的编译器是g++,为了将一个目标文件链接成共享库，在编译的时候，需要使用选项 **"-fPIC"** 。

在链接的时候，需要指定 **"-shared"** 和 **"Wl,--no-undefined"** 选项，并需要与\*libtestngpp.so\*\*链接。\*\*

比如:

arthur> g++ -fPIC -o MyTestSuite.o -c MyTestSuite.cpp -I/usr/local/include arthur> g++ -shared -Wl,--no-undefined -o libMyTestSuite.so -L/usr/local/lib -ltestngpp -L/home/arthur/MyProject -lMyProject

或者将编译和链接合并为一个命令行：

arthur> g++ -fPIC -shared -Wl,--no-undefined -o libMyTestSuite.so MyTestSuite.cpp -I/usr/local/include -L/usr/local/lib -ltestngpp -L/home/arthur/MyProject -lMyProject

### 3.3.2 Mac OS

在Linux上，为了将一个目标文件链接成共享库，在编译的时候，需要使用选项 **"-fPIC"** 。

在链接的时候，需要指定 **"-bundle"** 选项，并需要与\*libtestngpp.dylib\*\*链接。\*\*

比如:

arthur> g++ -fPIC -o MyTestSuite.o -c MyTestSuite.cpp -I/usr/local/include arthur> g++ -bundle -o libMyTestSuite.so -L/usr/local/lib -ltestngpp -L/home/arthur/MyProject -lMyProject

或者将编译和链接过程合二为一： arthur> g++ -fPIC -bundle -o libMyTestSuite.so MyTestSuite.cpp -I/usr/local/include -L/usr/local/lib -ltestngpp -L/home/arthur/MyProject -lMyProject

### 3.3.3 Windows

在Windows上，在链接DLL的时候，需要与\*testngpp.lib\*\*链接。\*\*

# 4. 运行

TestNG++的所有测试都最终都生成一个个 **module** ，然后通过 **Test Runner** 来运行这些 **module** 。

## 4.1 Test Runner

TestNG++ **Test Runner** 所对应的可执行文件是 **testngpp-runner** (在Windows上，文件名为testngpp-runner.exe)，它被安装在${CMAKE\_INSTALL\_PREFIX}/bin。

它所支持的选项如下:

| 选项 | 说明 | |:---|:---| | -s | 使用sandbox模式来运行测试；如果不指定，则使用flat模式 | | -cNumber | 当使用sandbox模式时，指定最大并发sandbox数量。比如-c10 | | -l | 指定Test Listener。可以指定任意多个Test Listener，并且可以指定Test Listener的参数，在指定参数的时候，需要用引号。比如: -l"testngppstdoutlistener -c" | | -L | 指定 Test Listener的搜索路径，可以指定多个。比如: -L/home/arthur -L/usr/local/testngpp/listener | | -t | 指定tag运行规范。指定tag规范时需要使用双引号。比如-t"ut > it" | | -f | 指定测试过滤规则。可以指定多个，它们之间是或的关系. 比如 -f"TestFoo:testStartup" -f "TestBar::testStartup" |

## 4.2 运行模式

TestNG++提供了两种运行测试的模式，一种是传统的的模式Flat模式，另外一种是Sandbox模式。

### 4.2.1 Flat模式

在Flat模式下，所有的测试在同一个地址空间（进程），一个接一个顺序执行。

在这种模式下，测试之间可能互相影响。比如： 1. 上一个测试运行结束后遗留的环境状态，可能对后面测试的运行造成影响。 1. 一个测试的崩溃，将会造成整个测试的中止。

### 4.2.2 Sandbox模式

当使用Flat模式的时候，尽管你可以通过完善的编写测试来尽力的避免测试之间的相互影响，但既然你在做测试，你的CUT就应可能是存在问题的。即便CUT本身没有任何问题，但为了保证一个测试运行结束后，其遗留的状态对另外一个测试没有任何影响，你需要付出额外的努力。

为了解决这样的问题，TestNG++提供了另外一种运行模式：Sandbox模式。在这种模式下，每一个测试都会运行在独立的地址空间。每一个测试都启动于一个干净的环境，即便一个测试崩溃，其他的测试也完全不受任何影响。

除此之外，Sandbox模式还提供并发运行测试的能力。

在调试状态下，由于多进程调式的困难，建议使用Flat模式来运行测试。

## 4.3 Test Listener

### 4.3.1 Stdout Listener

| 选项 | 说明 | |:---|:---| | -c | 使用彩色模式进行输出，如果你终端不支持彩色模式，请关闭此选项 | | -f | 以fixture为单位进行输出 | | -s | 以suite为单位进行输出 | | -t | 显示当前的tag规范 | | -v | verbose模式，如果不使用verbose模式，则默认是打点模式 | | -lNumber | 信息输出级别，Number的取值范围为0到3；0-输出所有信息；1-关闭INFO; 2-关闭INFO/WARN 3－关闭INFO/WARN/SKIPPED |

例子：

// 以verbose和彩色模式进行输出 -l"testngppstdoutlistener -c -v"

### 4.3.2 XML Listener

xmllistener只有一个参数，即xml报告的输出文件名。比如：

-l"testngppxmllistener result.xml"

如果不指定，则将报告输出到标准输出.

## 4.4 Tag Specification

## 4.5 Test Filter

Testngpp可以通过-f选项指定过滤规则，其规则为：

-f fixturePattern:testPattern

在fixturePattern和testPattern之间以冒号":"进行分割。

如果想过滤符合某种fixturePattern的所有测试，则可以通过如下的过滤规则：

-f fixturePattern

如果想过滤符合某种testPattern的所有测试，则可以通过如下的过滤规则：

-f :testPattern

分号“:”前面为空，则说明fixture可以为任意名字。

testngpp支持根据多种规则过滤，这些过滤规则之间是“或”的关系：

-f fixturePattern1:testPattern1 -f fixturePattern2:testPattern2

而过滤规则符合Posix文件名匹配规则。比如，想过滤fixture名字以abc开头的所有用例，则可以使用如下规则：

-f abc\*

而过滤规则符合Posix文件名匹配规则。比如，想过滤fixture名字由三个字母组成，其中第一个字母为a，第三个字母为c的所有用例，则可以使用如下规则：

-f a?c

而过滤规则符合Posix文件名匹配规则。比如，想过滤fixture名字以abc开头，而test名字里包含efg的所有用例，则可以使用如下规则：

-f abc\*:\*efg\*