

25 | 两个单元测试库：C++里如何进行单元测试？

2020-01-22 吴咏炜

现代C++实战30讲

[进入课程 >](#)



讲述：吴咏炜

时长 11:27 大小 10.50M



你好，我是吴咏炜。

单元测试已经越来越成为程序员工作密不可分的一部分了。在 C++ 里，我们当然也是可以很方便地进行单元测试的。今天，我就来介绍两个单元测试库：一个是 Boost.Test [1]，一个是 Catch2 [2]。

Boost.Test

单元测试库有很多，我选择 Boost 的原因我在上一讲已经说过：“如果我需要某个功能，在标准库里没有，在 Boost 里有，我会很乐意直接使用 Boost 里的方案，而非另外去查



找。”再说，Boost.Test 提供的功能还挺齐全的，我需要的都有了。作为开胃小菜，我们先看一个单元测试的小例子：

 复制代码

```
1 #define BOOST_TEST_MAIN
2 #include <boost/test/unit_test.hpp>
3 #include <stdexcept>
4
5 void test(int n)
6 {
7     if (n == 42) {
8         return;
9     }
10    throw std::runtime_error(
11        "Not the answer");
12 }
13
14 BOOST_AUTO_TEST_CASE(my_test)
15 {
16     BOOST_TEST_MESSAGE("Testing");
17     BOOST_TEST(1 + 1 == 2);
18     BOOST_CHECK_THROW(
19         test(41), std::runtime_error);
20     BOOST_CHECK_NO_THROW(test(42));
21
22     int expected = 5;
23     BOOST_TEST(2 + 2 == expected);
24     BOOST_CHECK(2 + 2 == expected);
25 }
26
27 BOOST_AUTO_TEST_CASE(null_test)
28 {
29 }
```

我们从代码里可以看到：

我们在包含单元测试的头文件之前定义了 `BOOST_TEST_MAIN`。如果编译时用到了多个源文件，只有一个应该定义该宏。多文件测试的时候，我一般会考虑把这个定义这个宏加包含放在一个单独的文件里（只有两行）。

我们用 `BOOST_AUTO_TEST_CASE` 来定义一个测试用例。一个测试用例里应当有多个测试语句（如 `BOOST_CHECK`）。

我们用 `BOOST_CHECK` 或 `BOOST_TEST` 来检查一个应当成立的布尔表达式（区别下面会讲）。

我们用 `BOOST_CHECK_THROW` 来检查一个应当抛出异常的语句。

我们用 `BOOST_CHECK_NO_THROW` 来检查一个不应当抛出异常的语句。

如 [\[第 21 讲\]](#) 所述，我们可以用下面的命令行来进行编译：

```
MSVC: cl /DBOOST_TEST_DYN_LINK /EHsc /MD test.cpp
```

```
GCC: g++ -DBOOST_TEST_DYN_LINK test.cpp -  
lboost_unit_test_framework
```

```
Clang: clang++ -DBOOST_TEST_DYN_LINK test.cpp -  
lboost_unit_test_framework
```

运行结果如下图所示：

```
Running 2 test cases...  
test.cpp:23: error: in "my_test": check 2 + 2 == expected has failed [2  
+ 2 != 5]  
test.cpp:24: error: in "my_test": check 2 + 2 == expected has failed  
  
*** 2 failures are detected in the test module "Master Test Suite"
```

我们现在能看到 `BOOST_CHECK` 和 `BOOST_TEST` 的区别了。后者是一个较新加入 `Boost.Test` 的宏，能利用模板技巧来输出表达式的具体内容。但在某些情况下，`BOOST_TEST` 试图输出表达式的内容会导致编译出错，这时可以改用更简单的 `BOOST_CHECK`。

不管是 `BOOST_CHECK` 还是 `BOOST_TEST`，在测试失败时，执行仍然会继续。在某些情况下，一个测试失败后继续执行后面的测试已经没有意义，这时，我们就可以考虑使用 `BOOST_REQUIRE` 或 `BOOST_TEST_REQUIRE`——表达式一旦失败，整个测试用例会停止执行（但其他测试用例仍会正常执行）。

缺省情况下单元测试的输出只包含错误信息和结果摘要，但输出的详细程度是可以通过命令行选项来进行控制的。如果我们在运行测试程序时加上命令行参数 `--`

log_level=all (或 -l all) , 我们就可以得到下面这样更详尽的输出:

```
Running 2 test cases...
Entering test module "Master Test Suite"
test.cpp:14: Entering test case "my_test"
Testing
test.cpp:17: info: check 1 + 1 == 2 has passed
test.cpp:19: info: check 'exception "std::runtime_error" raised as expected' has passed
test.cpp:20: info: check 'no exceptions thrown by test(42)' has passed
test.cpp:23: error: in "my_test": check 2 + 2 == expected has failed [2 + 2 != 5]
test.cpp:24: error: in "my_test": check 2 + 2 == expected has failed
test.cpp:14: Leaving test case "my_test"; testing time: 262us
test.cpp:27: Entering test case "null_test"
Test case null_test did not check any assertions
test.cpp:27: Leaving test case "null_test"; testing time: 29us
Leaving test module "Master Test Suite"; testing time: 344us

*** 2 failures are detected in the test module "Master Test Suite"
```

我们现在额外可以看到:

在进入、退出测试模块和用例时的提示

BOOST_TEST_MESSAGE 的输出

正常通过的测试的输出

用例里无测试断言的警告

使用 Windows 的同学如果运行了测试程序的话, 多半会惊恐地发现终端上的文字颜色已经发生了变化。这似乎是 Boost.Test 在 Windows 上特有的一个问题: 建议你把单元测试的色彩显示关掉。你可以在系统高级设置里添加下面这个环境变量, 也可以直接在命令行上输入:

```
1 set BOOST_TEST_COLOR_OUTPUT=0
```

 复制代码

下面我们看一个更真实的例子。

假设我们有一个 `split` 函数，定义如下：

 复制代码

```
1  template <typename String,  
2          typename Delimiter>  
3  class split_view {  
4  public:  
5      typedef  
6          typename String::value_type  
7          char_type;  
8      class iterator { ... };  
9  
10     split_view(const String& str,  
11               Delimiter delimiter);  
12     iterator begin() const;  
13     iterator end() const;  
14     vector<basic_string<char_type>>  
15     to_vector() const;  
16     vector<basic_string_view<char_type>>  
17     to_vector_sv() const;  
18 };  
19  
20 template <typename String,  
21          typename Delimiter>  
22 split_view<String, Delimiter>  
23 split(const String& str,  
24        Delimiter delimiter);
```

这个函数的意图是把类似于字符串的类型 (`string` 或 `string_view`) 分割开，并允许对分割的结果进行遍历。为了方便使用，结果也可以直接转化成字符串的数组

(`to_vector`) 或字符串视图的数组 (`to_vector_sv`)。我们不用关心这个函数是如何实现的，我们就需要测试一下，该如何写呢？

首先，当然是写出一个测试用例的框架，把试验的待分割字符串写进去：

 复制代码

```
1  BOOST_AUTO_TEST_CASE(split_test)  
2  {  
3      string_view str{  
4          "&grant_type=client_credential"  
5          "&appid="  
6          "&secret=APPSECRET"};  
7  }
```

最简单直白的测试，显然就是用 `to_vector` 或 `to_vector_sv` 来查看结果是否匹配了。
这个非常容易加进去：

 复制代码

```
1  vector<string>
2      split_result_expected{
3      "",
4      "grant_type=client_"
5      "credential",
6      "appid=",
7      "secret=APPSECRET"};
8  auto result = split(str, '&');
9  auto result_s =
10     result.to_vector();
11  BOOST_TEST(result_s ==
12             split_result_expected);
```

如果 `to_vector` 实现正确的话，我们现在运行程序就能在终端输出上看到：

```
*** No errors detected
```

下面，我们进一步检查 `to_vector` 和 `to_vector_sv` 的结果是否一致：

 复制代码

```
1  auto result_sv =
2      result.to_vector_sv();
3  BOOST_TEST_REQUIRE(
4      result_s.size() ==
5      result_sv.size());
6  {
7      auto it = result_sv.begin();
8      for (auto& s : result_s) {
9          BOOST_TEST(s == *it);
10         ++it;
11     }
12 }
```

最后我们再测试可以遍历 `result`，并且结果和之前的相同：

```

1  size_t i = 0;
2  auto it = result.begin();
3  auto end = result.end();
4  for (; it != end &&
5       i < result_s.size();
6       ++it) {
7      BOOST_TEST(*it == result_s[i]);
8      ++i;
9  }
10 BOOST_CHECK(it == end);

```

而这，差不多就接近我实际的 `split` 测试代码了。完整代码可参见：

https://github.com/adah1972/nwa/blob/master/test/boosttest_split.cpp

Boost.Test 产生的可执行代码支持很多命令行参数，可以用 `--help` 命令行选项来查看。常用的有：

`build_info` 可用来展示构建信息

`color_output` 可用来打开或关闭输出中的色彩

`log_format` 可用来指定日志输出的格式，包括纯文本、XML、JUnit 等

`log_level` 可指定日志输出的级别，有 `all`、`test_suite`、`error`、`fatal_error`、`nothing` 等一共 11 个级别

`run_test` 可选择只运行指定的测试用例

`show_progress` 可在测试时显示进度，在测试数量较大时比较有用（见下图）

```
Running 13 test cases...
```

```

0%   10   20   30   40   50   60   70   80   90  100%
|---|---|---|---|---|---|---|---|---|---|
*****

```

```
*** No errors detected
```

我这儿只是个简单的介绍。完整的 Boost.Test 的功能介绍还是请你自行参看文档。

Catch2

说完了 Boost.Test, 我们再来看一下另外一个单元测试库, Catch2。仍然是和上一讲里说的一样, 我要选择 Boost 之外的库, 一定有一个比较强的理由。Catch2 有着它自己独有的优点:

只需要单个头文件即可使用, 不需要安装和链接, 简单方便

可选使用 BDD (Behavior-Driven Development) 风格的分节形式

测试失败可选直接进入调试器 (Windows 和 macOS 上)

我们拿前面 Boost.Test 的示例直接改造一下:

 复制代码

```
1 #define CATCH_CONFIG_MAIN
2 #include "catch.hpp"
3 #include <stdexcept>
4
5 void test(int n)
6 {
7     if (n == 42) {
8         return;
9     }
10    throw std::runtime_error(
11        "Not the answer");
12 }
13
14 TEST_CASE("My first test", "[my]")
15 {
16     INFO("Testing");
17     CHECK(1 + 1 == 2);
18     CHECK_THROWS_AS(
19         test(41), std::runtime_error);
20     CHECK_NO_THROW(test(42));
21
22     int expected = 5;
23     CHECK(2 + 2 == expected);
24 }
25
26 TEST_CASE("A null test", "[null]")
27 {
28 }
```

可以看到，两者之间的相似性非常多，基本只是宏的名称变了一下。唯一值得一提的，是测试用例的参数：第一项是名字，第二项是标签，可以一个或多个。你除了可以直接在命令行上写测试的名字（不需要选项）来选择运行哪个测试外，也可以写测试的标签来选择运行哪些测试。

这是它在 Windows 下用 MSVC 编译的输出：

```
test is a Catch v2.11.1 host application.  
Run with -? for options
```

```
-----  
My first test  
-----
```

```
test.cpp(14)  
.....
```

```
test.cpp(23): FAILED:  
  CHECK( 2 + 2 == expected )  
with expansion:  
  4 == 5  
with message:  
  Testing
```

```
=====
```

test cases: 2	1 passed	1 failed
assertions: 4	3 passed	1 failed

终端的色彩不会被搞乱。缺省的输出清晰程度相当不错。至少在 Windows 下，它看起来可能是个比 Boost.Test 更好的选择。但反过来，在浅色的终端里，Catch2 的色彩不太友好。Boost.Test 在 Linux 和 macOS 下则不管终端的色彩设定，都有比较友好的输出。

和 Boost.Test 类似，Catch2 的测试结果输出格式也是可以修改的。默认格式是纯文本，但你可以通过使用 `-r junit` 来设成跟 JUnit 兼容的格式，或使用 `-r xml` 输出成 Catch2 自己的 XML 格式。这方面，它比 Boost.Test 明显易用的一个地方是格式参数大小写不敏感，而在 Boost.Test 里你必须用全大写的形式，如 `-f JUNIT`，麻烦！

下面我们通过另外一个例子来展示一下所谓的 BDD [3] 风格的测试。

BDD 风格的测试一般采用这样的结构：

Scenario: 场景, 我要做某某事

Given: 给定, 已有的条件

When: 当, 某个事件发生时

Then: 那样, 就应该发生什么

如果我们要测试一个容器, 那代码就应该是这个样子的:

 复制代码

```
1 SCENARIO("Int container can be accessed and modified",
2         "[container]")
3 {
4     GIVEN("A container with initialized items")
5     {
6         IntContainer c{1, 2, 3, 4, 5};
7         REQUIRE(c.size() == 5);
8
9         WHEN("I access existing items")
10        {
11            THEN("The items can be retrieved intact")
12            {
13                CHECK(c[0] == 1);
14                CHECK(c[1] == 2);
15                CHECK(c[2] == 3);
16                CHECK(c[3] == 4);
17                CHECK(c[4] == 5);
18            }
19        }
20
21        WHEN("I modify items")
22        {
23            c[1] = -2;
24            c[3] = -4;
25
26            THEN("Only modified items are changed")
27            {
28                CHECK(c[0] == 1);
29                CHECK(c[1] == -2);
30                CHECK(c[2] == 3);
31                CHECK(c[3] == -4);
32                CHECK(c[4] == 5);
33            }
34        }
35    }
36 }
```

你可以在程序前面加上类型定义来测试你自己的容器类或标准容器（如 `vector<int>`）。这是一种非常直观的写测试的方式。正常情况下，你当然应该看到：

```
All tests passed (12 assertions in 1 test case)
```

如果你没有留意到的话，在 GIVEN 里 WHEN 之前的代码是在每次 WHEN 之前都会执行一遍的。这也是 BDD 方式的一个非常方便的地方。

如果测试失败，我们就能看到类似下面这样的信息输出了（我存心制造了一个错误）：

```
test is a Catch v2.11.1 host application.
Run with -? for options

-----
Scenario: Int container can be accessed and modified
  Given: A container with initialized items
  When: I modify items
  Then: Only modified items are changed
-----
test.cpp(32)
.....
test.cpp(37): FAILED:
  CHECK( c[3] == -4 )
with expansion:
  0 == -4

=====
test cases: 1 | 0 passed | 1 failed
assertions: 12 | 11 passed | 1 failed
```

如果没有失败的情况下，想看到具体的测试内容，可以传递参数 `--success`（或 `-s`）。

如果你发现 Catch2 的编译速度有点慢的话，那我得告诉你，那是非常正常的。在你沮丧之前，我还应该马上告诉你，这在实际项目中完全不是一个问题。因为慢的原因通常主要是构建 Catch2 的主程序部分，而这部份在项目中只需要做一次，以后不会再有变动。你需要的是分离下面这部分代码在主程序里：

```
1 #define CATCH_CONFIG_MAIN
2 #include "catch.hpp"
```

 复制代码

只要这两行，来单独编译 Catch2 的主程序部分。你的实际测试代码里，则不要再定义 `CATCH_CONFIG_MAIN` 了。你会发现，这样一分离后，编译速度会大大加快。事实上，如果 Catch2 的主程序部分不需要编译的话，Catch2 的测试用例的编译速度在我的机器上比 Boost.Test 的还要快。

我觉得 Catch2 是一个很现代、很好用的测试框架。它的宏更简单，一个 `CHECK` 可以替代 Boost.Test 中的 `BOOST_TEST` 和 `BOOST_CHECK`，也没有 `BOOST_TEST` 在某些情况下不能用、必须换用 `BOOST_CHECK` 的问题。对于一个新项目，使用 Catch2 应该是件更简单、更容易上手的事——尤其如果你在 Windows 上开发的话。

目前，在 GitHub 上，Catch2 的收藏数超过一万，复刻（fork）数达到一千七，也已经足以证明它的流程度。

内容小结

今天我们介绍了两个单元测试库，Boost.Test 和 Catch2。整体上来看，这两个都是很优秀的单元测试框架，可以满足日常开发的需要。

课后思考

请你自己试验一下本讲中的例子，来制造一些成功和失败的情况。使用一下，才能更容易确定哪一个更适合你的需求。

参考资料

[1] Gennadiy Rozental and Raffi Enficiaud, Boost.Test.

🔗 <https://www.boost.org/doc/libs/release/libs/test/doc/html/index.html>

[2] Two Blue Cubes Ltd., Catch2. 🔗 <https://github.com/catchorg/Catch2>

[3] Wikipedia, “Behavior-driven development” .

🔗 https://en.wikipedia.org/wiki/Behavior-driven_development

点击查看 

打卡学习 C++ 拒绝从入门到放弃



PC 端用户扫码参与



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 24 | Boost: 你需要的“瑞士军刀”

下一篇 26 | Easylogging++和spdlog: 两个好用的日志库

精选留言 (3)

 写留言



承君此诺

2020-01-22

我用的是cmake ctest，个人观点，它很适合测试整个程序，不适合细分到测试某个功能函数。所以，我在看google的gtest。这2个测试框架，您怎么看

展开 

作者回复: 这两个都没有实际使用经验。如果你需要的功能都有，那就没有切换的理由吧。如果不足，就试试其他的。鉴于选择实在太多，重点试几个，满足需求就行了。我一直用 Boost.Test 的理由就是没找到换门的理由，而不是它有多好。当然，至少它是不差的。从下面这个列表里看（几十个 C++ 单元测试框架的概要对比），看起来没有严重缺的东西：

https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#C++



 1



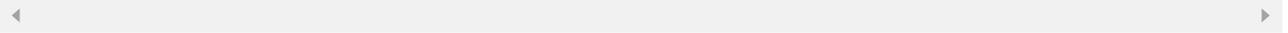
晚风·和煦

2020-01-23

老师，map里面插入成万条数据，如何释放内存呢

展开 ∨

作者回复: 全部清空吗? 用clear()有问题?



3



EricHu

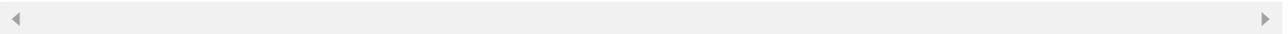
2020-01-22

老师，我想请教一下C++的友元类，它会破坏类的封装性，实际在开发中建议使用吗？要怎么做到类似JAVA反射的效果，访问修改内部静态变量

展开 ∨

作者回复: 尽量不用，但有时候必须用啊。比如我在智能指针的实现里就用了。

跟Java的比较，你说的情况我不熟。泛泛而言，不要在C++里模拟其他语言的做法。退一步，看你要达到的目的是什么，再看语言有什么机制可以满足目标。C++里目前没反射机制，可预见的将来也大概只会有供编译期使用的静态反射。



3

