Modern cpp 30 lectures — unit test

Friday, December 25, 2020 1:54 PM

Boost.Test

```cpp
#define BOOST_TEST_MAIN
#include <boost/test/unit-test.hpp>
#include <stdexcept>

void test(int n)
{
  if (n == 42) {
    return;
  }

  throw std::runtime_error(
    "Not the answer");
}

BOOST_AUTO_TEST_CASE(my_test)
{
  BOOST_TEST_MESSAGE("Testing");
  BOOST_TEST(1+1 == 2);
  BOOST_CHECK_THROW(
    test(41), std::runtime_error);
  BOOST_CHECK_NO_THROW(test(42));

  int expected = 5;
  BOOST_TEST(2+2 == expected);
  BOOST_CHECK(2+2 == expected);
}

BOOST_AUTO_TEST_CASE(null_test)
{
}
```

- 在包含单元测试的头文件前定义了 BOOST_TEST_MAIN.
- 用 BOOST_AUTO_TEST_CASE 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).
- 用BOOST_CHECK或BOOST_TEST来检查一个应当成立的布尔表达式.
- 用 BOOST_CHECK_THROW 来检查一个应当抛出异常的语句.
- 用 BOOST_CHECK_NO_THROW 来检查一个不应当抛出异常的语句.

假设我们有一个split函数,定义如下:

```cpp
template <typename String,
          typename Delimiter>
class split_view{
public:
  typedef
    typename String::value_type
      char_type;
  class iterator {...};

  split_view(const String& str,
             Delimiter delimiter);

  iterator begin() const;
  iterator end() const;

  vector<basic_string<char_type>> to_vector() const;
  vector<basic_string_view<char_type>> to_vector_sv() const;
};

template <typename String,
          typename Delimiter>
split_view<String, Delimiter>
split(const String& str,
      Delimiter delimiter);
```

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).

首先写出一个测试用例的框架,把试验的待分割字符串写进去:

```cpp
BOOST_AUTO_TEST_CASE(split_test)
{
  string_view str{
    "&grant_type=client_credential"
    "&appid="
    "&secret=APPSECRET"};
}
```

最简单直白的测试,就是用to_vector或to_vector_sv来查看结果是否匹配.

```cpp
vector<string>
  split_result_expected{
    "",
    "grant_type=client_",
    "credential",
    "appid=",
    "secret=APPSECRET"};

  auto result = split(str, '&');
  auto result_s = result.to_vector();
  BOOST_TEST(result_s == split_result_expected);
```

如果to_vector实现正确,输出:

```
No errors detected.
```

下一步我们检查to_vector和to_vector_sv的结果是否一致.

```cpp
  auto result_sv = result.to_vector_sv();
  BOOST_TEST_REQUIRE(result_s.size() == result_sv.size());
  {
    auto it = result_sv.begin();
    for (auto& s : result_s) {
      BOOST_TEST(s == *it);
      ++it;
    }
  }
```

最后再测试可以遍历result,并且结果和之前相同.

```cpp
size_t i = 0;
auto it = result.begin();
auto end = result.end();
for (; it != end && i < result_s.size(); ++it) {
    BOOST_TEST(*it == result_s[i]);
    ++i;
}
BOOST_CHECK(it == end);
```

Catch 2

```cpp
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include <stdexcept>

void test(int n)
{
   if (n == 42) {
     return;
   }
   throw std::runtime_error("Not the answer");
}
```

测试用例的参数,第一项是名字,第二项是标签.

```cpp
TEST_CASE("My first test", "[my]")
{
  INFO("Testing");
  CHECK(1+1 == 2);
  CHECK_THROWS_AS(test(41), std::runtime_error);
  CHECK_NOTHROW(test(42));

  int expected = 5;
  CHECK(2+2 == expected);
}

TEST_CASE("A null test", "[null]")
{
}
```

BDD风格的测试:

- 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.

如果要测试一个容器:

```cpp
SCENARIO("Int container can be accessed and modified", "[container]")
{
  GIVEN("A container with initialized items")
  {
    IntContainer c{1,2,3,4,5};
    REQUIRE(c.size() == 5);

    WHEN("I access existing items")
    {
      THEN("The items can be retrived intact")
      {
        CHECK(c[0] == 1);
        CHECK(c[1] == 2);
        CHECK(c[2] == 3);
        CHECK(c[3] == 4);
        CHECK(c[4] == 5);
      }
    }

    WHEN("I modify items")
    {
      c[1] = -2;
      c[3] = -4;

      THEN("Only modified items are changed")
      {
        CHECK(c[0] == 1);
        CHECK(c[1] == -2);
        CHECK(c[2] == 3);
        CHECK(c[3] == -4);
        CHECK(c[4] == 5);
      }
    }
  }
}
```