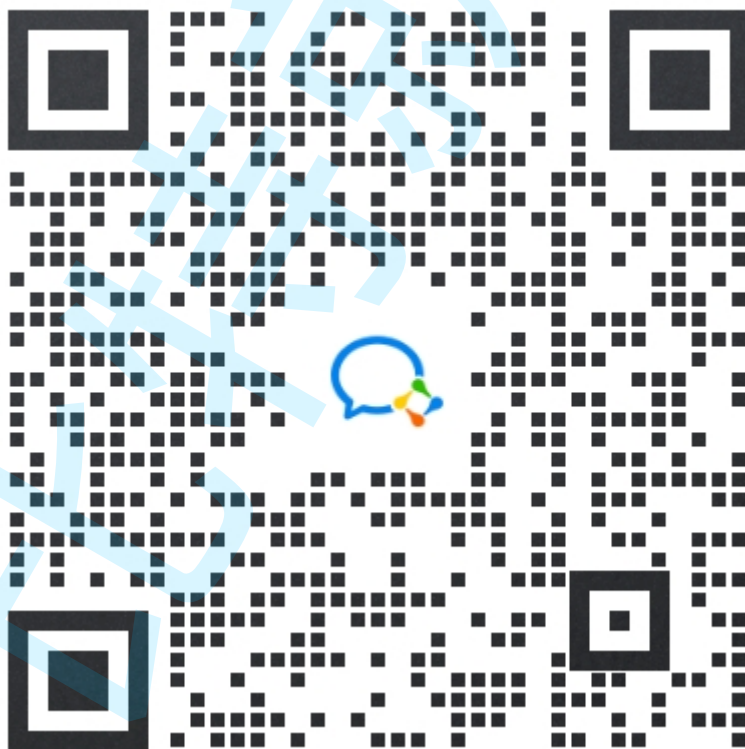


gtest 框架安装与使用

版权说明

本“比特就业课”项目（以下简称“本项目”）的所有内容，包括但不限于文字、图片、音频、视频、软件、程序、数据库、设计、布局、界面等，均由本项目的开发者或授权方拥有版权。我们鼓励个人学习者使用本项目进行学习和研究。在遵守相关法律法规的前提下，个人学习者可以下载、浏览、学习本项目的内容，并为了个人学习、研究或教学目的而使用其中的材料。但请注意，未经我们明确授权，个人学习者不得将本项目的内容用于任何商业目的，包括但不限于销售、转让、许可或以其他方式从中获利。此外，个人学习者也不得擅自修改、复制、传播、展示、表演或制作本项目内容的衍生作品。任何未经授权的使用均属侵权行为，我们将依法追究法律责任。如果您希望以其他方式使用本项目的内容，包括但不限于引用、转载、摘录、改编等，请事先与我们联系，获取书面授权。感谢您对“比特就业课”项目的关注与支持，我们将持续努力，为您提供更好的学习体验。特此说明。比特就业课版权所有方

对比特项目感兴趣，可以联系这个微信。



代码 & 板书链接

<https://gitee.com/bitedu-tech/cpp-chatsystem>

安装

命令安装：

```
C++
dev@dev-host:~/workspace$ sudo apt-get install libgtest-dev
```

介绍

GTest 是一个跨平台的 C++单元测试框架，由 google 公司发布。gtest 是为了在不同平台上为编写 C++单元测试而生成的。它提供了丰富的断言、致命和非致命判断、参数化等等测试所需的宏，以及全局测试，单元测试组件。

头文件包含：

```
C++
#include <gtest/gtest.h>
```

框架初始化接口：

```
C++
testing::InitGoogleTest(&argc, argv);
```

调用测试样例：

```
C++
RUN_ALL_TESTS();
```

TEST 宏

```
C++
//这里不需要双引号，且同测试下多个测试样例不能同名
TEST(测试名称, 测试样例名称)
```

```
TEST_F(test_fixture, test_name)
```

- **TEST**: 主要用来创建一个简单测试，它定义了一个测试函数，在这个函数中可以使用任何 C++ 代码并且使用框架提供的断言进行检查
- **TEST_F**: 主要用来进行多样测试，适用于多个测试场景如果需要相同的数据配置的情况，即相同的数据测不同的行为

断言宏

GTest 中的断言的宏可以分为两类：

- **ASSERT**_系列：如果当前点检测失败则退出当前函数
- **EXPECT**_系列：如果当前点检测失败则继续往下执行

下面是经常使用的断言介绍

```
C++
// bool 值检查
ASSERT_TRUE(参数), 期待结果是 true
ASSERT_FALSE(参数), 期待结果是 false

// 数值型数据检查
ASSERT_EQ(参数 1, 参数 2), 传入的是需要比较的两个数 equal
ASSERT_NE(参数 1, 参数 2), not equal, 不等于才返回 true
ASSERT_LT(参数 1, 参数 2), less than, 小于才返回 true
ASSERT_GT(参数 1, 参数 2), greater than, 大于才返回 true
ASSERT_LE(参数 1, 参数 2), less equal, 小于等于才返回 true
ASSERT_GE(参数 1, 参数 2), greater equal, 大于等于才返回 true
```

样例：

```
C++
#include<iostream>
#include<gtest/gtest.h>

int abs(int x)
{
    return x > 0 ? x : -x;
}
```

```

TEST(abs_test, test1)
{
    ASSERT_TRUE(abs(1) == 1) << "abs(1)=1";
    ASSERT_TRUE(abs(-1) == 1);
    ASSERT_FALSE(abs(-2) == -2);
    ASSERT_EQ(abs(1),abs(-1));
    ASSERT_NE(abs(-1),0);
    ASSERT_LT(abs(-1),2);
    ASSERT_GT(abs(-1),0);
    ASSERT_LE(abs(-1),2);
    ASSERT_GE(abs(-1),0);
}

int main(int argc,char *argv[])
{
    //将命令行参数传递给 gtest
    testing::InitGoogleTest(&argc, argv);
    // 运行所有测试案例
    return RUN_ALL_TESTS();
}

```

```

C++
main : main.cc
      g++ -std=c++17 $^ -o $@ -lgtest

```

```

C++
dev@dev-host:~/workspace$ ./main
[=====] Running 1 test from 1 test case.
[-----] Global test environment set-up.
[-----] 1 test from abs_test
[ RUN      ] abs_test.test1
[      OK  ] abs_test.test1 (0 ms)
[-----] 1 test from abs_test (0 ms total)

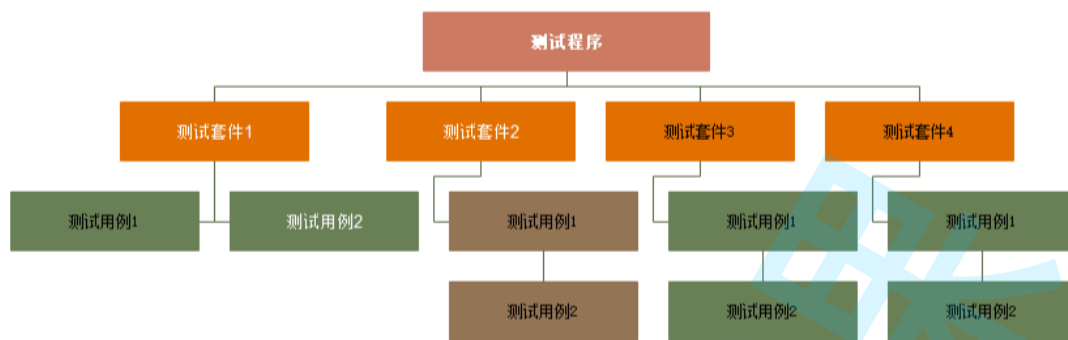
[-----] Global test environment tear-down
[=====] 1 test from 1 test case ran. (0 ms total)
[ PASSED  ] 1 test.

```

事件机制

GTest 中的事件机制是指在测试前和测试后提供给用户自行添加操作的机制，而且该

机制也可以让同一测试套件下的测试用例共享数据。GTest 框架中事件的结构层次：



- 测试程序：一个测试程序只有一个 `main` 函数，也可以说是一个可执行程序是一个测试程序。该级别的事件机制是在程序的开始和结束执行
- 测试套件：代表一个测试用例的集合体，该级别的事件机制是在整体的测试案例开始和结束执行
- 测试用例：该级别的事件机制是在每个测试用例开始和结束都执行

事件机制的最大好处就是能够为我们各个测试用例提前准备好测试环境，并在测试完毕后用于销毁环境，这样有个好处就是如果我们有一端代码需要进行多种不同方法的测试，则可以通过测试机制在每个测试用例进行之前初始化测试环境和数据，并在测试完毕后清理测试造成的影响。

GTest 提供了三种常见的事件：

全局事件：

针对整个测试程序。实现全局的事件机制，需要创建一个自己的类，然后继承 `testing::Environment` 类，然后分别实现成员函数 `SetUp` 和 `TearDown`，同时在 `main` 函数内进行调用 `testing::AddGlobalTestEnvironment(new MyEnvironment);` 函数添加全局的事件机制

```
C++
#include <iostream>
#include <gtest/gtest.h>
//全局事件:针对整个测试程序,提供全局事件机制,能够在测试之前配置测试环境
数据,测试完毕后清理数据
//先定义环境类,通过继承 testing::Environment 的派生类来完成
//重写的虚函数接口 SetUp 会在测试之前被调用; TearDown 会在测试完毕后调用.
std::unordered_map<std::string, std::string> dict;

class HashTestEnv : public testing::Environment {
public:
```

```

        virtual void SetUp() override{
            std::cout << "测试前:提前准备数据!!\n";
            dict.insert(std::make_pair("Hello", "你好"));
            dict.insert(std::make_pair("hello", "你好"));
            dict.insert(std::make_pair("雷吼", "你好"));
        }
        virtual void TearDown() override{
            std::cout << "测试结束后:清理数据!!\n";
            dict.clear();
        }
    };

    TEST(hash_case_test, find_test) {
        auto it = dict.find("hello");
        ASSERT_NE(it, dict.end());
    }
    TEST(hash_case_test, size_test) {
        ASSERT_GT(dict.size(), 0);
    }
    int main(int argc, char *argv[])
    {
        testing::AddGlobalTestEnvironment(new HashTestEnv );
        testing::InitGoogleTest(&argc, argv);
        return RUN_ALL_TESTS();
    }
}

```

运行结果:

```

Shell
[zwc@VM-8-12-centos gtest]$ ./event
[=====] Running 2 tests from 1 test case.
[-----] Global test environment set-up.
测试前:提前准备数据!!
[-----] 2 tests from hash_case_test
[ RUN      ] hash_case_test.find_test
[      OK  ] hash_case_test.find_test (0 ms)
[ RUN      ] hash_case_test.size_test
[      OK  ] hash_case_test.size_test (0 ms)
[-----] 2 tests from hash_case_test (0 ms total)

[-----] Global test environment tear-down
测试结束后:清理数据!!
[=====] 2 tests from 1 test case ran. (0 ms total)

```

```
[ PASSED ] 2 tests.
```

TestSuite 事件：

针对一个个测试套件。测试套件的事件机制我们同样需要去创建一个类，继承自 `testing::Test`，实现两个静态函数 `SetUpTestCase` 和 `TearDownTestCase`，测试套件的事件机制不需要像全局事件机制一样在 `main` 注册，而是需要将我们平时使用的 `TEST` 宏改为 `TEST_F` 宏。

- `SetUpTestCase()` 函数是在测试套件第一个测试用例开始前执行
- `TearDownTestCase()` 函数是在测试套件最后一个测试用例结束后执行
- 需要注意 `TEST_F` 的第一个参数是我们创建的类名，也就是当前测试套件的名称，这样在 `TEST_F` 宏的测试套件中就可以访问类中的成员了。

```
C++
#include <iostream>
#include <gtest/gtest.h>
//TestSuite:测试套件/集合进行单元测试,即,将多个相关测试归入一组的方式进行测试,为这组测试用例进行环境配置和清理
//概念: 对一个功能的验证往往需要很多测试用例,测试套件就是针对一组相关测试用例进行环境配置的事件机制
//用法: 先定义环境类,继承于 testing::Test 基类,重写两个静态函数
//      SetUpTestCase/TearDownTestCase 进行环境的配置和清理
class HashTestEnv1 : public testing::Test {
public:
    static void SetUpTestCase() {
        std::cout << "环境 1 第一个 TEST 之前调用\n";
    }
    static void TearDownTestCase() {
        std::cout << "环境 1 最后一个 TEST 之后调用\n";
    }
public:
    std::unordered_map<std::string, std::string> dict;
};

// 注意,测试套件使用的不是 TEST 了,而是 TEST_F, 而第一个参数名称就是测试套件环境类名称
// main 函数中不需要再注册环境了,而是在 TEST_F 中可以直接访问类的成员变量和成员函数
TEST_F(HashTestEnv1, insert_test) {
    std::cout << "环境 1,中间 insert 测试\n";
```

```

        dict.insert(std::make_pair("Hello", "你好"));
        dict.insert(std::make_pair("hello", "你好"));
        dict.insert(std::make_pair("雷吼", "你好"));
        auto it = dict.find("hello");
        ASSERT_NE(it, dict.end());
    }
    TEST_F(HashTestEnv1, sizeof) {
        std::cout << "环境 1, 中间 size 测试\n";
        ASSERT_GT(dict.size(), 0);
    }

int main(int argc, char *argv[])
{
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}

```

运行结果:

```

C++
[zwc@VM-8-12-centos gtest]$ ./event
[=====] Running 2 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 2 tests from HashTestEnv1
环境 1 第一个 TEST 之前调用
[ RUN      ] HashTestEnv1.insert_test
环境 1, 中间 insert 测试
[      OK  ] HashTestEnv1.insert_test (0 ms)
[ RUN      ] HashTestEnv1.sizeof
环境 1, 中间 size 测试
event.cpp:81: Failure
Expected: (dict.size()) > (0), actual: 0 vs 0
[  FAILED  ] HashTestEnv1.sizeof (0 ms)
环境 1 最后一个 TEST 之后调用
[-----] 2 tests from HashTestEnv1 (0 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test case ran. (1 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] HashTestEnv1.sizeof

1 FAILED TEST

```

能够看到在上例中，有一个好处，就是将数据与测试结合到同一个测试环境类中了，这样与外界的耦合度更低，代码也更清晰。

但是同样的，我们发现在两个测试用例中第二个测试用例失败了，这是为什么呢？这就涉及到了 `TestCase` 事件的机制。

TestCase 事件:

针对一个个测试用例。测试用例的事件机制的创建和测试套件的基本一样，不同地方在于测试用例实现的两个函数分别是 `SetUp` 和 `TearDown`，这两个函数也不是静态函数

- `SetUp()`函数是在一个测试用例的开始前执行
- `TearDown()`函数是在一个测试用例的结束后执行

也就是说，在 `TestSuite/TestCase` 事件中，每个测试用例，虽然它们同用同一个事件环境类，可以访问其中的资源，但是本质上每个测试用例的环境都是独立的，这样我们就不用担心不同的测试用例之间会有数据上的影响了，保证所有的测试用例都使用相同的测试环境进行测试。

C++

```
//TestCase:测试用例的单元测试,即针对每一个测试用例都使用独立的测试环境
数据进行测试
//概念:它是针对测试用例进行环境配置的一种事件机制
//用法:先定义环境类,继承于 testing::Test 基类,在环境类内重写
SetUp/TearDown 接口
class HashTestEnv2 : public testing::Test {
public:
    static void SetUpTestCase() {
        std::cout << "环境 2 第一个 TEST 之前被调用,进行总体环境配
置\n";
    }
    static void TearDownTestCase() {
        std::cout << "环境 2 最后一个 TEST 之后被调用,进行总体环境
清理\n";
    }
    virtual void SetUp() override{
        std::cout << "环境 2 测试前:提前准备数据!!\n";
        dict.insert(std::make_pair("bye", "再见"));
        dict.insert(std::make_pair("see you", "再见"));
    }
    virtual void TearDown() override{
        std::cout << "环境 2 测试结束后:清理数据!!\n";
        dict.clear();
    }
};
```

```

    }
    public:
        std::unordered_map<std::string, std::string> dict;
};
TEST_F(HashTestEnv2, insert_test) {
    std::cout << "环境 2,中间测试\n";
    dict.insert(std::make_pair("hello", "你好"));
    ASSERT_EQ(dict.size(), 3);
}
TEST_F(HashTestEnv2, size_test) {
    std::cout << "环境 2,中间 size 测试\n";
    auto it = dict.find("hello");
    ASSERT_EQ(it, dict.end());
    ASSERT_EQ(dict.size(), 2);
}

int main(int argc, char *argv[])
{
    testing::InitGoogleTest(&argc, argv);
    RUN_ALL_TESTS();
    return 0;
}

```

运行结果:

```

Bash
[zwc@VM-8-12-centos gtest]$ ./event
[=====] Running 2 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 2 tests from HashTestEnv2
环境 2 第一个 TEST 之前被调用,进行总体环境配置
[ RUN      ] HashTestEnv2.insert_test
环境 2 测试前:提前准备数据!!
环境 2,中间测试
环境 2 测试结束后:清理数据!!
[ OK      ] HashTestEnv2.insert_test (1 ms)
[ RUN      ] HashTestEnv2.size_test
环境 2 测试前:提前准备数据!!
环境 2,中间 size 测试
环境 2 测试结束后:清理数据!!
[ OK      ] HashTestEnv2.size_test (0 ms)
环境 2 最后一个 TEST 之后被调用,进行总体环境清理

```

```
[-----] 2 tests from HashTestEnv2 (1 ms total)
```

```
[-----] Global test environment tear-down
```

```
[=====] 2 tests from 1 test case ran. (1 ms total)
```

```
[  PASSED  ] 2 tests.
```