

## 27 | 牛刀小试（下）：实现一个自己的测试框架

2020-03-21 胡光

人人都能学会的编程入门课

[进入课程 >](#)



讲述：胡光

时长 14:08 大小 12.95M



你好，我是胡光，欢迎回来，今天呢，我们继续学习测试框架的相关内容。

上节课中，我们讲到了软件开发一般分为前中后三个部分，提到作为技术人员的我们，一般主要负责在软件开发中期的编码与测试阶段。还有，我还讲到我们一般会综合运用白盒测试与黑盒测试这两种方法进行程序测试。

更主要的是，我们还介绍了 Google 的单元测试框架 gtest，并对测试框架代码进行了一番解读。其中提到代码中的 TEST 是一个宏，那它展开后被替换的内容是什么呢？还有，☆ RUN\_ALL\_TESTS 函数是如何依次执行程序中的所有测试用例函数的？

今天呢，我们就一个一个地来解决这些问题，并最终实现一个咱们自己的测试框架。

## 初步实现 TEST 宏

今天我们实现的所有代码呢，都会写在一个名字为 `geek_test.h` 的头文件中。当然我们也知道，将声明和定义写在一起，在大型工程中是会出现严重的编译错误，在实际的工程开发中，我们并不会这么做。

今天把声明和定义写在一起，只是为了课程内容的讲解需要，而你也完全没有必要担心，这不会影响你对主要内容的学习。


我们先回到上节课中的源代码：

 复制代码

```
1 #include <stdio.h>
2 #include "geek_test.h" // 替换掉原 gtest/gtest.h 头文件
3
4 // 判断一个数字 x 是否是素数
5 int is_prime(int x) {
6     for (int i = 2; i * i < x; i++) {
7         if (x % i == 0) return 0;
8     }
9     return 1;
10 }
11
12 // 第一个测试用例
13 TEST(test1, test_is_prime) {
14     EXPECT_EQ(is_prime(3), 1);
15     EXPECT_EQ(is_prime(5), 1);
16     EXPECT_EQ(is_prime(7), 1);
17 }
18
19 // 第二个测试用例
20 TEST(test2, test_is_prime) {
21     EXPECT_EQ(is_prime(4), 0);
22     EXPECT_EQ(is_prime(0), 0);
23     EXPECT_EQ(is_prime(1), 0);
24 }
25
26 int main() {
27     return RUN_ALL_TESTS();
28 }
```

我们的目的，是在不改变这份源代码的前提下，通过在 `geek_test.h` 中添加一些源码，使得这份代码的运行效果，能够类似于 `gtest` 的运行效果。

想要完成这个目标，我们就要先来思考 TEST 宏这里的内容，请你仔细观察这段由 TEST 宏定义的测试用例的相关代码：

 复制代码

```
1 TEST(test1, test_is_prime) {  
2     EXPECT_EQ(is_prime(3), 1);  
3     EXPECT_EQ(is_prime(5), 1);  
4     EXPECT_EQ(is_prime(7), 1);  
5 }
```

TEST(test1, test\_is\_prime) 这部分应该是在调用 TEST 宏，而这部分被预处理器展开以后的内容，只有和后面大括号里的代码组合在一起，才是一段合法的 C 语言代码，也只有这样，这份代码才能通过编译。

既然如此，我们就不难想到，TEST 宏展开以后，它应该是一个函数定义的头部，后面大括号里的代码，就是这个展开以后的函数头部的函数体部分，这样一切就都说得通了。

在实现 TEST 宏之前，我们还需要想清楚一个问题：由于程序中可以定义多个 TEST 测试用例，如果每一个 TEST 宏展开都是一个函数头部的话，那这个展开的函数的名字是什么呢？如果每一个 TEST 宏展开的函数名字都一样，那程序一定无法通过编译，编译器会报与函数名重复相关的错误，所以，TEST 宏是如何确定展开函数的名字呢？


不知道你有没有注意到，TEST 宏需要传入两个参数，这两个参数在输出信息中与测试用例的名字有关。那我们就该想到，可以使用这两个参数拼接出一个函数名，只要 TEST 传入的这两个参数不一样，那扩展出来的函数名就不同。最后，我们就可以初步得到如下的 TEST 宏的一个实现：

 复制代码

```
1 #define TEST(test_name, func_name) \  
2 void test_name##_##func_name()
```

如代码所示的 TEST 宏实现，我们将 TEST 宏的两个参数内容使用 ## 连接在一起，中间用一个额外的下划线连接，组成一个函数名字，这个函数的返回值类型是 void，无传入参数。

根据这个实现，预处理器会将源代码中两处 TEST 宏的内容，替换成如下代码所示内容：

 复制代码

```
1 void test1_test_is_prime() {
2     EXPECT_EQ(is_prime(3), 1);
3     EXPECT_EQ(is_prime(5), 1);
4     EXPECT_EQ(is_prime(7), 1);
5 }
6
7 void test2_test_is_prime() {
8     EXPECT_EQ(is_prime(4), 0);
9     EXPECT_EQ(is_prime(0), 0);
10    EXPECT_EQ(is_prime(1), 0);
11 }
```


这样，也就把原来看似不合理的 TEST 宏，转换成了合法的 C 语言代码了。

## \_\_attribute\_\_: 让其它函数先于主函数执行

在继续讲测试框架的设计之前，我们来补充一个知识点。

之前，我们所学习到的程序执行过程，既是从主函数开始，也是从主函数结束。也就是说，在常规的程序执行过程中，其它函数都是在主函数执行之后，才被直接或者间接调用执行。接下来，我就要给你讲一种能够让函数在主函数执行之前就执行的编程技巧。

首先，我们先来看如下代码：

 复制代码

```
1 #include <stdio.h>
2
3 void pre_output() {
4     printf("hello geek!\n");
5     return ;
6 }
7
8 int main() {
9     printf("hello main!");
10    return 0;
11 }
```

代码运行以后，会输出一行字符串 “hello main!” 。

接下来呢，我们对上述代码稍微修改，在 pre\_output 函数前面加上 `__attribute__((constructor))`。这样，pre\_output 函数就会先于主函数执行，代码如下：

 复制代码

```
1 #include <stdio.h>
2
3 __attribute__((constructor))
4 void pre_output() {
5     printf("hello geek!\n");
6     return ;
7 }
8
9 int main() {
10     printf("hello main!\n");
11     return 0;
12 }
```

如上代码执行以后，程序会输出两行内容，第 1 行是 pre\_output 函数输出的内容 “hello geek!”，第 2 行才是主函数的执行输出内容 “hello main!”。

从输出内容可以看出，加了 `__attribute__((constructor))` 以后，pre\_output 函数会先于 main 主函数执行，这种有趣的特性，在接下来的操作中我们还会用得上，你要理解并记住。

其实 `__attribute__` 的作用还很多，你可以上网搜搜，会让你的程序性质变得特别有意思。

## RUN\_ALL\_TESTS 函数设计

好了，准备工作都做完了，接下来让我们来思考一下 RUN\_ALL\_TESTS 函数要完成的事情，以及完成这些事情所需要的条件。

从主函数中调用 RUN\_ALL\_TESTS 函数的方式来看，RUN\_ALL\_TESTS 函数应该是一个返回值为整型的函数。这样，我们可以得到这样的函数声明形式：



```
1 int RUN_ALL_TESTS();
```

从测试框架的执行输出结果中看，RUN\_ALL\_TESTS 函数可以依次性地执行每一个 TEST 宏扩展出来的测试用例函数，这是怎么做到的呢？

我们可以这样认为：在主函数执行 RUN\_ALL\_TESTS 函数之前，有一些函数过程，就已经把测试用例函数的相关信息，记录在了一个 RUN\_ALL\_TESTS 函数可以访问到的地方，等到 RUN\_ALL\_TESTS 函数执行的时候，就可以根据这些记录的信息，依次性地执行这些测试用例函数。整个过程，如下图所示：

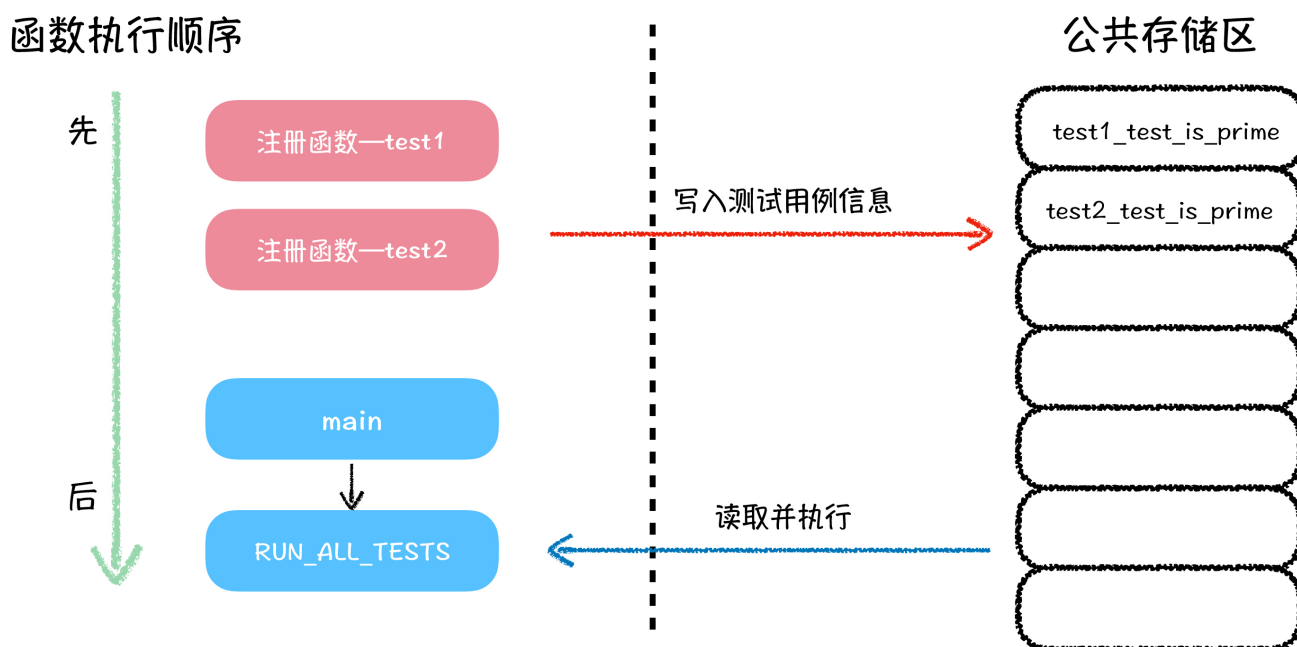



图1: RUN\_ALL\_TESTS 执行流程

图中红色部分，就是我们推测的，某些完成测试用例函数信息注册的函数，它们先于主函数执行，将测试用例的信息，写入到一个公共存储区中。

接下来，我们需要考虑的就是这些注册函数，究竟将什么信息存储到了公共存储区中，才能使得 RUN\_ALL\_TESTS 函数可以调用到这些测试用例？你自己也可以想想是什么。答案就是这个信息是测试用例函数的函数地址，因为只有把函数地址存储到这个存储区中，才能保证 RUN\_ALL\_TESTS 函数可以调用它们。所以，这片公共存储区，就应该是一个函数指针数组。

那如何解决注册函数问题呢？最简单直接的设计方法，就是每多一个由 TEST 宏定义的测试用例，就配套一个注册函数，所以这个注册函数的逻辑，可以设计在 TEST 宏展开的内容中。这就需要对 TEST 宏进行重新设计，这里我一会儿再给你进行说明。

我们先来完成 RUN\_ALL\_TESTS 函数从存储区中，读取并执行测试用例的过程：

 复制代码


```
1  typedef void (*test_function_t)();
2
3  test_function_t test_function_arr[100];
4  int test_function_cnt = 0;
5
6  int RUN_ALL_TESTS() {
7      for (int i = 0; i < test_function_cnt; i++) {
8          printf("RUN TEST : %d\n", i + 1);
9          test_function_arr[i]();
10         printf("RUN TEST DONE\n\n");
11     }
12     return 0;
13 }
```

代码中用到了函数指针相关的技巧，其中 test\_function\_t 是我们定义的函数指针类型，这种函数指针类型的变量，可以用来指向返回值是 void，传入参数为空的函数。

之后，定义了一个有 100 位的函数指针数组 test\_function\_arr，数组中的每个位置，都可以存储一个函数地址，数组中元素数量，记录在整型变量 test\_function\_cnt 中。这样，RUN\_ALL\_TESTS 函数中的逻辑就很简单了，就是依次遍历函数指针数组中的每个函数，然后依次执行这些函数，这些函数每一个都是一个测试用例。

## 重新设计：TEST 宏

根据前面的分析，TEST 扩展出来的内容，不仅要有测试用例的函数头部，还需要有先于主函数执行的注册函数，主要用于注册 TEST 扩展出来的测试用例函数。由此，我们可以得出如下示例代码：


 复制代码

```
1  #define TEST(test_name, func_name) \
2  void test_name##_##func_name(); \
3  __attribute__((constructor)) \
4  void register_##test_name##_##func_name() { \
```

```
5     test_function_arr[test_function_cnt] = test_name##_##func_name; \
6     test_function_cnt++; \
7 } \
8 void test_name##_##func_name()
```

这个新设计的 TEST 宏，除了末尾保留了原 TEST 宏内容以外，在扩展的测试用例函数头部添加了一段扩展内容，这段新添加的扩展内容，会扩展出来一个函数声明，以及一个以 register 开头的会在主函数执行之前执行的注册函数；注册函数内部的逻辑很简单，就是将测试函数的函数地址，存储在函数指针数组 test\_function\_arr 中，这部分区域中的数据，后续会被 RUN\_ALL\_TESTS 函数使用。

如果以如上 TEST 宏作为实现，原程序中的两个测试用例代码，会被展开成如下样子：

 复制代码

```
1 void test1_test_is_prime();
2
3 __attribute__((constructor))
4 void register_test1_test_is_prime() {
5     test_function_arr[test_function_cnt] = test1_test_is_prime;
6     test_function_cnt++;
7 }
8
9 void test1_test_is_prime() {
10     EXPECT_EQ(is_prime(3), 1);
11     EXPECT_EQ(is_prime(5), 1);
12     EXPECT_EQ(is_prime(7), 1);
13 }
14
15 void test2_test_is_prime();
16
17 __attribute__((constructor))
18 void register_test2_test_is_prime() {
19     test_function_arr[test_function_cnt] = test2_test_is_prime;
20     test_function_cnt++;
21 }
22
23 void test2_test_is_prime() {
24     EXPECT_EQ(is_prime(4), 0);
25     EXPECT_EQ(is_prime(0), 0);
26     EXPECT_EQ(is_prime(1), 0);
27 }
```



这个展开内容，是我给你做完代码格式整理以后的样子，实际展开结果会比这个格式乱一点儿，不过代码逻辑都一样。从展开内容中你可以看到，在展开代码的第 4 行和第 18 行分别就是两个测试用例函数的注册函数。

至此，我们就算是初步完成了测试框架中关键的两个部分的设计：一个是 TEST 宏，另外一个就是 RUN\_ALL\_TESTS 函数。它们同时也是串起测试框架流程最重要的两部分。

关于 EXPECT\_EQ 是如何实现的，我就留作思考题吧，也希望你认真想一想，把你的答案写在留言区中，我们一起讨论。这个实现答案肯定不唯一，你只需要尽量做到最好即可。

## 课程小结

最后，我来给你做一下今天的课程小结：

1. \_\_attribute\_\_((constructor)) 可以修饰函数，使修饰的函数先于主函数执行。
2. RUN\_ALL\_TESTS 之所以可以获得程序中所有测试用例的函数信息，是因为有一批注册函数，将测试用例函数记录下来了。
3. 通过测试框架这个项目，我们再一次看到，宏可以将原本看似不合理的代码，变得合理。

通过这两次课程，我希望你意识到，我们不是在阅读已有的测试框架的源码，而是在根据已有的测试框架，脑补其内部实现过程。

其实，脑补这个能力，往往是项目开发中的重要能力之一。例如，根据产品需要的外在功能描述，脑补后续的开发细节；根据竞品可见的功能表现，脑补其背后的技术细节。能够脑补一个产品的实现细节，可以让我们逐渐掌握，认清相关技术边界的能力，这个能力可以让我们不盲目崇拜某个公司，也不会随意轻视某个产品。

好了，今天就到这里了。下期我将带你继续完善测试框架的相关功能，我是胡光，我们下期见。

# 学习 6 小时， 「免费」领课程！



🕒 3月23日-3月29日

【点击】图片, 查看详情, 参与学习

© 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 26 | 牛刀小试（上）：实现测试框架前的基础准备

下一篇 28 | 尝试升级（上）：完善测试框架的功能与提示

## 精选留言 (1)

💬 写留言



😊 HappyJoo

2020-03-24

老师，自己乱写真的好多奇奇怪怪的问题呀，能不能给个每章节的源代码参考一下呢？我根据章节内容拼凑起来的代码，太多问题了，都不知道该从哪里开始问起了(ಥ\_ಥ)，总不能次次都把我的源码丢出来问你哈哈哈哈哈。有些问题真的搜了两三个小时才可能侥幸找到解决方法，如果老师能给分参考，可能可以减少一些不必要的时间呢~

展开 ▾



