

## 28 | 尝试升级（上）：完善测试框架的功能与提示

2020-03-24 胡光

人人都能学会的编程入门课

[进入课程 >](#)



讲述：胡光

时长 14:37 大小 13.40M



你好，我是胡光，欢迎回来。

在上一节课中呢，我们学习了测试框架的主要功能流程，完成了最重要的 `RUN_ALL_TESTS` 函数的功能逻辑。并且在这个学习期间，我们还使用了注册函数的技巧，就是让一些函数先于主函数执行，将测试用例函数信息记录在一个函数指针数组中，为后续的 `RUN_ALL_TESTS` 函数功能的执行作铺垫。

可你有没有发现，我们上节课程所完成的代码，只能让我们的测试框架在整体流程功能上通，然而程序的输出内容却不如 `gtest` 丰富。




今天，我们的主要任务，就是参考 gtest 的输出，逐步完善我们自己测试框架的相关信息输出方面的细节，从而让输出内容更加符合我们想要的信息。来，让我们一起开始吧。

## 温故知新，gtest 的输出结果

我们先来回顾一下 gtest 的输出结果，gtest 的输出内容大体可以分成三个部分。

第一部分，一套单元测试的相关信息：

 复制代码

```
1 [=====] Running 2 tests from 1 test suite.
2 [-----] Global test environment set-up.
3 [-----] 2 tests from test_is_prime
```

这段信息说明这套单元测试中，包含了 2 个测试用例。

第二部分，是每个单元测试运行信息的输出：

 复制代码

```
1 [ RUN      ] test_is_prime.test1
2 [          OK ] test_is_prime.test1 (1 ms)
3 [ RUN      ] test_is_prime.test2
4 gtest_test.cpp:25: Failure
5 Expected equality of these values:
6   is_prime(4)
7     Which is: 1
8   0
9 gtest_test.cpp:26: Failure
10 Expected equality of these values:
11   is_prime(0)
12     Which is: 1
13   0
14 gtest_test.cpp:27: Failure
15 Expected equality of these values:
16   is_prime(1)
17     Which is: 1
18   0
19 [ FAILED   ] test_is_prime.test2 (0 ms)
```

如上所示，第一个单元测试 `test_is_prime.test1` 运行结果正确，所用时间是 1ms；第二个单元测试 `test_is_prime.test2` 中，有三个判等 `EXPECT` 断言的结果是错误的，也就是 `is_prime` 函数的返回值，和测试用例中期望的返回值不符，这说明 `is_prime` 函数存在 Bug。

第三部分，就是这套单元测试的总结信息，以及整个程序单元测试结果的汇总信息。这段信息，有兴趣的小伙伴可以自己理解着看一下，由于不是咱们今天课程的重点，就不展开介绍了。

 复制代码

```
1 [-----] 2 tests from test_is_prime (1 ms total)
2
3 [-----] Global test environment tear-down
4 [=====] 2 tests from 1 test suite ran. (1 ms total)
5 [ PASSED ] 1 test.
6 [ FAILED ] 1 test, listed below:
7 [ FAILED ] test_is_prime.test2
8
9 1 FAILED
```

好了，关于 `gtest` 的输出内容，我大致说清楚了。

今天呢，我们先忽略 `gtest` 输出内容的第一部分和第三部分，主要关注 `gtest` 输出内容的第二部分，也就是每个单元测试运行信息的输出部分。通过第二部分的输出内容，你能想出我们应该从哪些方面来完善测试框架？


这里呢，我给出我的想法：通过观察第二部分的输出，我们基本要从三个方面完善测试框架的输出信息。

1. 在每个测试用例运行之前，要先行输出相关测试用例的名字；
2. 每个测试用例运行结束以后，要输出测试用例的运行时间与运行结果（OK 或者 FAILED）；
3. 若测试用例中的 `EXPECT` 断言出错，需要输出错误提示信息。

好了，优化的方向找到了，那么接下来，我们就开始测试框架改装行动吧！

## 测试用例的名字输出

首先是如何输出测试用例的名字。我们先回忆一下上节课设计的注册函数，如下所示：

 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()
```


注册函数是随着 TEST 展开的，从展开的代码逻辑中可以看到，它只是将测试用例的函数地址记录在了函数指针数组中。要想 RUN\_ALL\_TESTS 函数后续能够输出测试用例的函数名称的话，我们只需要修改注册函数的功能逻辑即可，也就是让注册函数在记录函数信息的时候，增加记录对应测试用例的名称。

而这个名称信息，应该记录在哪里呢？有两种代码实现方式：

1. 另外开辟一个记录测试用例名称的字符串数组；
2. 修改 test\_function\_arr 数组中的元素类型，将新增的测试用例名称以及函数地址信息打包成一个数据元素。

显然，相较于第一种实现方式，第二种代码实现方式会使程序具有更好的封装特性。我们采用之前在“语言基础篇”中学习的结构体相关知识，就可以完成这种多种数据类型打包成一种新的数据类型的功能需求。

下面就是我们将函数指针信息和测试用例名称信息，封装成的一个新的结构体类型：

 复制代码

```
1 struct test_function_info_t {
2     test_function_t func; // 测试用例函数指针，指向测试用例函数
3     const char *name; // 指向测试用例名称
4 } test_function_arr[100];
5 int test_function_cnt = 0;
```

如代码所示，我们定义了一种新的数据类型，叫做 `test_function_info_t`。这种结构体类型包含了指向测试用例的函数指针 `func` 字段，与指向测试用例名称的字符串指针 `name` 字段，并且我们将这种结构体类型，作为 `test_function_arr` 数组新的元素类型。

既然测试用例信息的存储区 `test_function_arr` 的数据类型发生了改变，那么负责存储信息的注册函数，与使用信息的 `RUN_ALL_TESTS` 函数的相关逻辑都需要作出改变。

首先，我们来看注册函数的改变。想要修改注册函数的逻辑，就是修改 `TEST` 宏，从功能上来说，注册函数中需要额外记录一个测试用例名称信息，示例代码如下：

 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].func = test_name##_##func_name; \
6     test_function_arr[test_function_cnt].name = #func_name "." #test_name; \
7     test_function_cnt++; \
8 } \
9 void test_name##_##func_name()
```

代码中主要是增加了第 6 行的逻辑，这一行的代码将 `TEST` 宏参数的两部分，拼成一个字符串，中间用点 (.) 连接，例如 `TEST(test1, test_is_prime)` 宏调用中，拼凑的字符串就是 `test_is_prime.test1`，和 `gtest` 中的输出的测试用例名称信息格式是一致的。

改完了注册函数的逻辑以后，最后调整一下 `RUN_ALL_TESTS` 中使用 `test_function_arr` 数组的逻辑代码即可：

 复制代码

```
1 int RUN_ALL_TESTS() {
2     for (int i = 0; i < test_function_cnt; i++) {
3         printf("[ RUN      ] %s\n", test_function_arr[i].name);
4         test_function_arr[i].func();
5         printf("RUN TEST DONE\n\n");
6     }
7     return 0;
8 }
```

代码中的第 3 行，是仿照 gtest 的输出格式进行调整的，在输出测试用例名称之前，先输出一段包含 RUN 英文的标志信息。

至此，我们就完成了输出测试用例名字的框架功能改造。

## 输出测试用例的运行结果信息

接下来，就让我们进行第二个功能改造：输出测试用例的运行结果信息。

以下是我们示例代码中的 2 个测试用例，在 gtest 框架下的运行结果信息输出：

```
1 [ OK ] test_is_prime.test1 (1 ms)
2 [ FAILED ] test_is_prime.test2 (0 ms)
```

 复制代码

根据输出的信息，我们可知 gtest 会统计每个测试用例运行的时间，并以毫秒为计量单位，输出此时间信息。不仅如此，gtest 还会输出与测试用例是否正确相关的信息，如果测试用例运行正确，就会输出一行包含 OK 的标志信息，否则就输出一行包含 FAILED 的标志信息。

根据我们自己测试框架的设计，这行信息只有可能是在 RUN\_ALL\_TESTS 函数的 for 循环中，执行完每一个测试用例函数以后输出的信息。

由此，我们面临的是两个需要解决的问题：

1. 如何统计函数过程的运行时间？
2. 如何确定获得每一个测试用例函数的测试结果是否正确？

说到如何统计函数过程的运行时间，我这里就需要介绍两个新的知识点，一个是函数 clock()，另一个是宏 CLOCKS\_PER\_SEC。下面我会对它们详细讲解。

我们先说函数 clock()。它的返回值代表了：从运行程序开始，到调用 clock() 函数时，经过的 CPU 时钟计时单元。并且，这个 clock() 函数的返回值，实际上反映的是我们程序的



运行时间。那这个 CPU 时钟计时单元究竟是什么呢？你可以把 1 个 CPU 时钟计时单元，简单的理解成是一个单位时间长度，只不过这个单位时间长度，不是我们常说的 1 秒钟。

接下来，我们再说说宏 `CLOCKS_PER_SEC`。它实际上是一个整型值，代表多少个 CPU 时钟计时单元是 1 秒。这个值在不同环境中会有所不同，在早年我的 Windows 电脑上，这个值是 1000，也就是 1000 个 CPU 时钟计时单位等于 1 秒。而现在我的 Mac 电脑上，这个值是 1000000，也就是 1000000 个 CPU 时钟计时单位等于 1 秒钟。显然，这个数字越大，统计粒度就越精细。

有了上面这两个工具，我们就可以轻松地统计一个函数的运行时间。在函数运行之前，记录一个 `clock()` 值，函数运行结束以后，再记录一个 `clock()` 值，用两个记录值的差值除以 `CLOCKS_PER_SEC`，得到的就是以秒为单位的函数运行时间，再乘以 1000，即为毫秒单位。

这样呢，我们就解决了刚刚提的第一个问题：统计函数过程的运行时间。

至于如何获得每一个测试用例的测试结果，我们可以采用一个简单的解决办法，那就是记录一个全局变量，代表测试用例结果正确与否。当测试用例中的 `EXPECT_EQ` 断言发生错误时，就修改这个全局变量的值，这样我们的 `RUN_ALL_TESTS` 函数，就可以在测试用例函数执行结束以后，得知执行过程是否有错。

综合以上所有信息，我们可以重新设计 `RUN_ALL_TESTS` 函数如下：

 复制代码

```
1  int test_run_flag;
2  #define EXPECT_EQ(a, b) test_run_flag &= ((a) == (b))
3
4  int RUN_ALL_TESTS() {
5      for (int i = 0; i < test_function_cnt; i++) {
6          printf("[ RUN      ] %s\n", test_function_arr[i].name);
7          test_run_flag = 1;
8          long long t1 = clock();
9          test_function_arr[i].func();
10         long long t2 = clock();
11         if (test_run_flag) {
12             printf("[      OK ] ");
13         } else {
14             printf("[  FAILED  ] ");
15         }
16         printf("%s", test_function_arr[i].name);
```

```
17         printf(" (%.0lf ms)\n\n", 1.0 * (t2 - t1) / CLOCKS_PER_SEC * 1000);
18     }
19     return 0;
20 }
```

代码中的第 8 行是在测试用例运行之前，记录一个开始时间值 t1；代码中的第 10 行是在测试用例函数执行完后，记录一个结束时间值 t2；在代码的第 17 行，根据 t1、t2 以及 CLOCKS\_PER\_SEC 的值，计算得到测试用例函数实际运行的时间，并输出得到的结果。

这段代码中增加了一个全局变量 “test\_run\_flag”，这个变量每次在测试用例执行之前，都会被初始化为 1，当测试用例结束执行以后，RUN\_ALL\_TESTS 函数中，根据 test\_run\_flag 变量的值，选择输出 OK 或者 FAILED 的标志信息。同时，我们可以看到，test\_run\_flag 变量的值只有在 EXPECT\_EQ 断言中，才可能被修改。

## EXPECT\_EQ 断言的实现

最后呢，我们还剩下一个 EXPECT\_EQ 断言的实现，这个就给你留作思考题，请你基于我上述所讲的内容，试试自己实现这个带错误提示输出的 EXPECT\_EQ 断言吧。也欢迎你把你的答案写在留言区，我们一起讨论。

## 课程小结

通过今天的课程呢，我希望你认识到**工程开发中的一个基本原则：功能迭代，数据先行。也就是说，无论我们做什么样的功能开发，首先要考虑的是与数据相关的部分。**更细致的解释，就是你考虑某种功能的实现，要明白这个功能都依赖于哪些数据信息，这些信息在哪里存储，在哪里修改，在哪里读取使用。把数据相关部分设计明白了，你的功能开发也就基本实现了一半了。

就像我们今天改造的第一个功能，输出测试用例的名字。

首先，我们考虑如何存储名字信息，最先被修改的就是 test\_function\_arr 数组的数据类型，我们改造了数据存储的结构。然后，我们修改了注册函数的相关功能逻辑，也就是解决了数据的写入与修改过程。最后，我们修改 RUN\_ALL\_TESTS 中的输出逻辑，也就是解决了数据在哪里读取和使用的事情。



至此，我已经向你演示了基本的功能迭代开发过程。接下来你可以自己试着，给输出的内容加上点儿颜色，以便更清晰地展示测试过程中的测试信息。除此之外呢，你也可以开动你的创造力，给测试框架加些令人惊喜的功能。

好了，今天就到这里了，我是胡光，我们下节课见。

## 学习计划

# 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 27 | 牛刀小试（下）：实现一个自己的测试框架

## 精选留言

💬 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。