

29 | 尝试升级（下）：“链表”知识在测试框架中的应用

2020-03-26 胡光

人人都能学会的编程入门课

[进入课程 >](#)



讲述：胡光

时长 13:02 大小 11.94M



你好，我是胡光，欢迎回来。

上节课中，我们通过参考 gtest 的输出，完善了我们自己的测试框架的输出信息，也就是添加了测试用例的名称、运行结果以及运行时间。并且，我提到了在一般情况下，项目中的功能开发原则：**功能迭代，数据先行**。就是要开发新的功能之前，我们应该先考虑清楚实现这部分功能相关的数据，在系统中的存储与使用的情况，只有这样，才能更好地完成功能的实现与迭代优化。



今天迎来我们整个测试框架项目的最后一节课。这节课的目的，一是对前几节课内容进行一个总结，二是向你说明我们现在开发的测试框架代码，其实还有很多优化的空间。至于这个

优化空间是什么呢？这次我将带着你结合之前学习的“链表”知识，对测试框架进行一个具体的优化改进。

关于测试框架的优化，是一个不断学习的过程。在这个过程中，你深刻体会到“知不足，然后能自反也”这句话的含义。就是在优化代码的过程中，你会发现自己的不足，然后努力提高自己的能力去弥补不足；当你提升了自己之后，你又会看到自己在其他方面的不足，进而继续提高自己。


好了，废话不多说，我们正式开始今天的学习。

揭晓答案：EXPECT_EQ 宏究竟是如何实现的

在对测试框架进行优化之前呢，我先来回答一下，可能困扰了你两节课的一个问题：就是 EXPECT_EQ 宏究竟是如何实现的？这个问题的答案呢，我给出一个可行的实现，仅供参考。

首先，EXPECT_EQ(a, b) 在 a, b 两部分值相等的时候，不会产生额外的输出信息，而当 a, b 两部分不相等的时候，就会输出相应的提示信息。如下所示：

```
1 gtest_test.cpp:25: Failure
2 Expected equality of these values:
3   is_prime(4)
4     Which is: 1
5   0
```

 复制代码

这段输出信息，对应的是源代码中的“EXPECT_EQ(is_prime(4), 0);”的输出。如你所见，第 1 行的输出内容包含了源文件名 (gtest_test.cpp)，EXPECT_EQ 宏所在的代码位置 (25)，以及一个提示结果 (Failure)。

接下来的信息，你自己就可以看懂了，就是关于 EXPECT_EQ 传入两部分的值。对于函数调用部分，EXPECT_EQ 会输出这个函数的调用形式及返回值信息，也就是输出中的“is_prime(4)”“Which is: 1”这段内容。而对于数值信息，只会输出数值信息本身，也就是输出信息中第 5 行的那个 0。

实际上，要想在宏中实现类似于这种根据传入参数类型，选择输出形式的功能，对于现在的你来说可能有点困难。所以，我们可以重新设计一种输出形式，只要能够清晰地展示错误信息就可以。


重新设计的输出提示，如下所示：

 复制代码

```
1 gtest_test.cpp:25: Failure
2 Expected (is_prime(4) == 0):
3     Which is: (1 == 0)
```

修改完以后的输出信息，你可以看到，第 2 行就是传入 EXPECT_EQ 宏两部分的比较，第 3 行是这两部分实际输出值的比较。

重新设计了输出信息以后，就可以来看看 EXPECT_EQ 宏的实现了：

 复制代码

```
1 #define EXPECT(a, b, comp) { \
2     __typeof(a) val_a = (a), val_b = (b); \
3     if (!(val_a comp val_b)) { \
4         printf("%s:%d: Failure\n", __FILE__, __LINE__); \
5         printf("Expected (%s %s %s):\n", #a, #comp, #b); \
6         printf("    Which is: (%d %s %d)\n", val_a, #comp, val_b); \
7         test_run_flag = 0; \
8     } \
9 }
10 #define EXPECT_EQ(a, b) EXPECT(a, b, ==)
11 #define EXPECT_LT(a, b) EXPECT(a, b, <)
12 #define EXPECT_GT(a, b) EXPECT(a, b, >)
13 #define EXPECT_NE(a, b) EXPECT(a, b, !=)
```

在这段实现中，你会发现，我们不仅实现了 EXPECT_EQ，还额外实现了 EXPECT_LT、EXPECT_GT、EXPECT_NE 等用于比较的宏。其中，LT 是英文 little 的缩写，是判断小于关系的；GT 是 great 的缩写，是判断大于关系的；NE 是 not equal 的缩写，是判断不等于关系的。而这些所有的宏，都是基于 EXPECT 宏实现的。


我们将用于比较的运算符，当作参数传递给 EXPECT 宏。有了 EXPECT 宏以后，你就可以参考代码中的第 10 ~ 13 行的内容，轻松地扩展出用于小于等于或者大于等于的宏了。由于

EXPECT 宏的实现，全都是我们之前学习过的知识点，所以在这里我就不再赘述了，你可以自行阅读文稿中的代码。

用链表存储测试用例

看完了 EXPECT 宏的参考实现以后，整个测试框架的基础功能，就算是彻底搭建完成了。

接下来，我们再重新审视下面这段函数指针数组 test_function_arr 的代码设计，来思考一下这个测试框架中还有没有可以优化的地方。

 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;
```

这段代码中，我们使用了数组来定义存储测试函数信息的存储区，这个数组的大小有 100 位，也就是说，最多可以存储 100 个测试用例函数信息。

那我们来思考一个问题：要是程序中定义了 1000 个测试用例，怎么办呢？毕竟，对于中型项目开发来说，定义 1000 个测试用例，可不是什么难事儿。这个时候，你可能会说，那简单啊，数组大小设置成 10000 不就行了。

但是你要明白，这种设计尽管简单粗暴且有效，可它一点儿程序设计的美感都没有。什么意思呢？就是当我们为测试用例准备了 10000 个数组空间的时候，可能在真正的开发过程中，只定义了 2 个测试用例，这就会浪费掉 9998 个数组空间。

更形象地描述这种行为的话，这种设计方式很像计划经济，计划多少用多少。同时，它的弊端也很明显，一旦计划不好，要不是造成空间浪费，要不就是资源紧张。

所以，我们应该尝试着从“计划经济”向“市场经济”转变一下，可不可以转变成想用多少就生产多少。那应该怎么做呢？

我们知道，在程序中数组的空间大小，是需要提前计划出来的。但是有一种结构的空间，是可以动态增加或减少的，那就是我们之前讲过的“**链表**”结构。你想一下，如果我们把一个

一个的测试函数信息，封装成一个一个的链表节点，每当增加一个测试用例的时候，就相当于向整个链表中插入一个新的节点。此时，用链表实现的存储测试函数信息的结构，它所占空间大小就和实际测试用例的数量成正比了。这就是我说的用多少，就生产多少。

下面，我们就来说说具体怎么做。

第一步，我们需要改变 `test_function_info_t` 的结构定义，也就是把原先存储测试用例函数信息的结构体类型，改装成链表结构。最简单的方法，就是在结构体的定义中，增加一个指针字段，指向下一个 `test_function_info_t` 类型的数据，代码如下所示：

 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     struct test_function_info_t *next;
5 };
6 struct test_function_info_t head, *tail = &head;
```

可以看到，我们给 `test_function_info_t` 结构体类型增加了一个链表中的 `next` 字段，除此之外，我们还定义了一个虚拟头节点 `head` 和一个指针变量 `tail`。这里你需要注意，`head` 是虚拟头节点，后续我们会向 `head` 所指向链表中插入链表节点，`tail` 指针则指向了整个链表的最后一个节点的地址。

第二步，在准备好了数据存储结构以后，需要改写的就是函数注册的逻辑了。在改写 `TEST` 宏中的注册函数逻辑之前呢，我们先准备一个工具函数 `add_test_function`，这个工具函数的作用，就是根据传入的参数，新建一个链表节点，并且插入到整个链表的末尾：

 复制代码


```
1 void add_test_function(const char *name, test_function_t func) {
2     struct test_function_info_t *node;
3     node = (struct test_function_info_t *)malloc(sizeof(struct test_function_i
4     node->func = func;
5     node->name = name;
6     node->next = NULL;
7     tail->next = node;
8     tail = node;
9     return ;
10 }
```


好了，`add_test_function` 工具函数准备好之后，我们正式来改写 `TEST` 宏中注册函数的逻辑。其实难度也不大，也就是要求注册函数调用 `add_test_function` 函数，并且传入相关的测试用例的函数信息即可：

 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     add_test_function(#func_name "." #test_name, \
6                       test_name##_##func_name); \
7 } \
8 void test_name##_##func_name()
```

最后一步，处理完了数据写入的过程以后，来让我们修改一下使用这份数据的代码逻辑，那就是 `RUN_ALL_TESTS` 函数中的相关逻辑。之前，`RUN_ALL_TESTS` 函数中，循环遍历数组中的每一个测试用例，并且执行相关的测试用例函数，对这一部分，修改成针对于链表结构的遍历方式即可，代码如下所示：

 复制代码

```
1 int RUN_ALL_TESTS() {
2     struct test_function_info_t *p = head.next;
3     for (; p; p = p->next) {
4         printf("[ RUN      ] %s\n", p->name);
5         test_run_flag = 1;
6         long long t1 = clock();
7         p->func();
8         long long t2 = clock();
9         if (test_run_flag) {
10             printf("[      OK ] ");
11         } else {
12             printf("[  FAILED  ] ");
13         }
14         printf("%s", p->name);
15         printf(" (%.0lf ms)\n\n", 1.0 * (t2 - t1) / CLOCKS_PER_SEC * 1000);
16     }
17     return 0;
18 }
```

这样，我们就彻底完成了测试用例函数信息存储部分的“链表”改造过程。

对于上面的这份代码实现，你会发现，链表节点空间是通过 malloc 函数动态申请出来的，可在我们的程序中，并没有对这些空间使用 free 进行释放，如果你想让这个程序对空间的申请与回收做到有始有终，变得更加干净，那应该怎么办呢？

这里你可以借助 `__attribute__((destructor))` 的功能，之前我们介绍了一个 `__attribute__((constructor))` 的作用是让函数先于主函数执行，而 `destructor` 就是使函数在主函数结束以后才执行的函数特性设置。有了这个特性设置，你可以实现一个函数，专门用来销毁测试函数链表所占存储空间，这样在逻辑上，你的程序会变得更完美。当然，你即使不这么做，也不会影响到原有的程序功能的正确性。

项目小结

至此，我们关于测试框架开发的内容，就算是告一段落了。

从 26 讲到 29 讲，我们经历了一个项目从 0 到 1 的过程，继而又完成了项目从 1 到 1.5 的升级。所谓从 0 到 1 就是项目从最初的想法变成代码的过程，从 1 到 1.5 就是我们对于代码的优化过程。这是一个追求极致、不断优化项目的过程。

关于测试框架开发的讲解内容虽然结束了，但我希望这几节课可以成为你优化这份代码的一个起点。日后，你可以选择增加额外的功能，修改实现的架构，甚至是使用不同的语言重新进行实现，哪怕是一个小小的改动，都是值得称赞的。

在下节课，我将指导你完成一个简易的计算器程序，也算是给我们这个课程一个圆满的结束。

好了，今天就到这里了，我是胡光，我们下一节课见。

学习计划

打卡 3 道题 「免费」领课程

🕒 3月30日-4月5日



【点击】图片, 立即领取

© 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 28 | 尝试升级 (上) : 完善测试框架的功能与提示

下一篇 30 | 毕业设计: 实现你自己的计算器程序

精选留言 (1)

💬 写留言



一步

2020-03-26

有关 C 语言的知识还有待提高

展开 ∨

