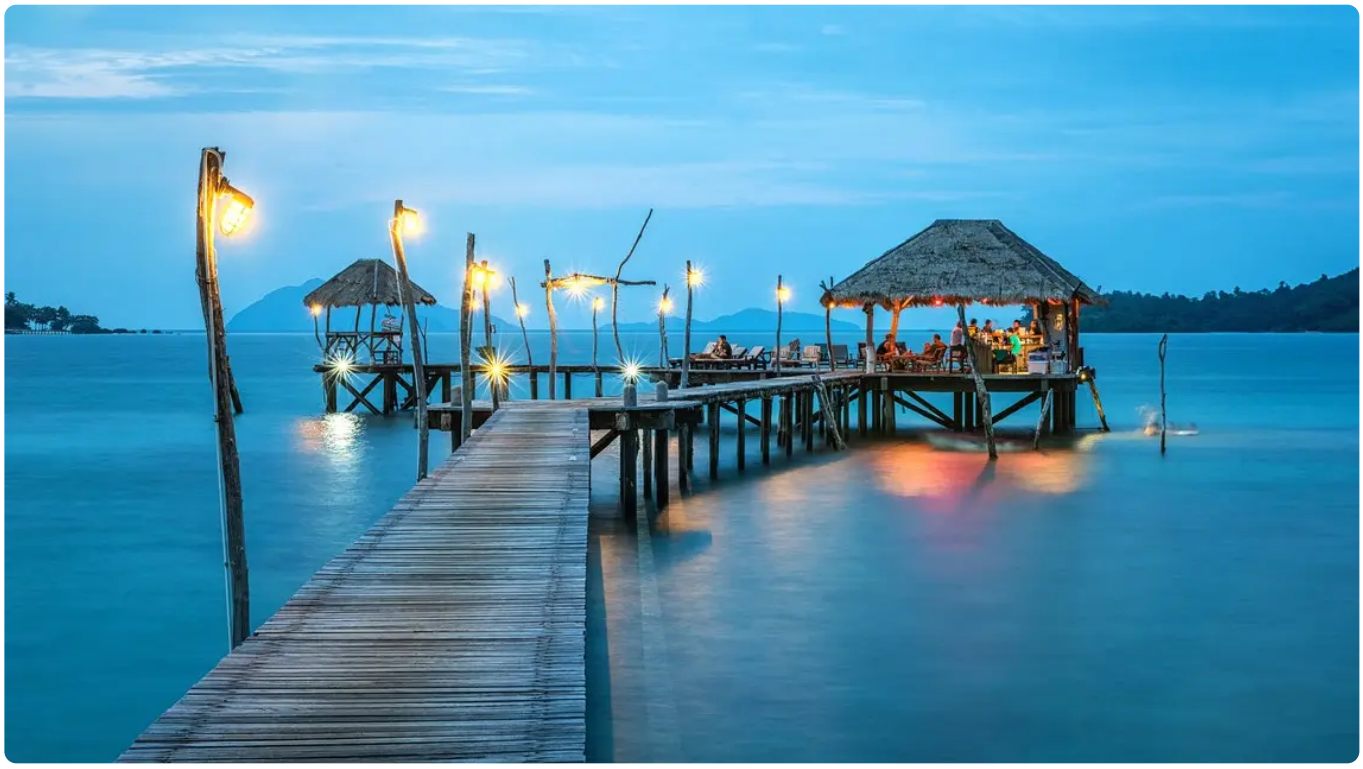
21 | 生产加速：如何使用自动化测试确保 C 项目质量？

2022-02-07 于航

《深入C语言和程序运行原理》 课程介绍 >



讲述：于航

时长 14:36 大小 13.38M

你好，我是于航。

“测试”是每个软件在其开发生命周期（SDLC）中都不可或缺的一个重要阶段。通过对软件进行各种不同类型的测试，我们能够从多个维度验证软件的功能表现，并在出现偏差时及时修正，以确保它们可以按照预期工作。根据实施方式、深入粒度、应用场景及目的等因素的不同，测试可以被分为多种类型。其中，有些测试较为基础和通用，甚至被作为软件开发流程中的必备一环；而有些测试则仅适用于某些特定情况。

因此，为了尽量保证本讲内容的通用性，这里我挑选了 C 项目中最为常用的几种测试类型，主要包括单元测试、集成测试、功能测试与性能测试。接下来，我将为你分别介绍它们的作用，以及它们之间的区别和联系，还有如何进行这些测试。在这一讲的最后，我还会介绍什么是自动化测试，以及如何更进一步地做到“真正”的测试自动化。

单元测试

我们先来看通常会最先接触到的一种测试类型，单元测试（Unit Testing）。顾名思义，单元测试就是对组成程序整体结构的基本单元（也可称为模块）进行功能正确性验证的过程。它的目标是，隔离程序的每个部分，并单独验证这些部分能否按照预期正常工作。**对于 C 程序来说，这里的单元通常为程序使用到的各个函数。**

既然要对这些基本单元的功能进行测试，那便需要编写一些代码来使用这些单元，并为它们提供多种不同输入，来验证相应的输出或副作用变化是否符合单元的正常功能实现。而为了让目标单元在测试过程中正常运作，我们通常还需要为它准备特定的配套测试环境，比如用于替换真实代码（如外部函数调用）的桩（Stub）代码，以及各类 Mock 资源（如 DB 访问层、外部 API 接口）等。这样做的目的在于，隔离单元需要依赖的外部环境，使测试范围可以尽量集中在单元的内部逻辑上。

使用 CUnit 进行 C 单元测试

CUnit 是一个专门用于 C 语言的轻量级单元测试框架。接下来，我就通过它来向你展示一个最基本的单元测试用例是如何编写的。直接来看下面这段代码：

复制代码

```
// ...
int maxi(int x, int y) {  // 被测试函数；
  return (x > y) ? x : y;
}
void test_maxi(void) {  // 测试用例；
  CU_ASSERT(maxi(0, 2) == 2);
  CU_ASSERT(maxi(0, -2) == 0);
  CU_ASSERT(maxi(2, 2) == 2);
}
// ...
```

关于 CUnit 的具体使用方式，这里我不过多介绍了，你可以点击这个链接来查看更多信息。

在这段代码中，我们为函数 maxi 编写了一个基本的单元测试用例 test_maxi。maxi 函数在执行时会接收两个整型参数，并返回两者中值较大的那一个。作为测试用例，test_maxi 方法在它的内部便会按照被测试函数的已知功能逻辑，来验证不同输入下对应输出的正确性。可以看到，我们在用例中分三次调用了 maxi 函数，并将返回值与相应的正确结果值

进行比较（“==”）。而借助 CU_ASSERT 等一系列断言函数，框架可以在测试运行后，帮助我们从整体角度追踪各个具体测试用例的执行情况，并给出相应的测试报告。

当单元测试进行完毕后，我们可以得到如下图所示的测试报告。这里，测试结果会按照测试用例的不同类别分开展示。比如对 CUnit 来说，多条断言语句通常组成一个针对某个具体单元的 Test 函数，而多个相关的 Test 函数则组成一个针对某类功能的 Suite 测试函数集。

```
     CUnit - A unit testing framework for C - Version 2.1-3
     http://cunit.sourceforge.net/


Suite: Sucess
  Test: testSuccess1 ...passed

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      3      3      3      0      n/a

Elapsed time =    0.000 seconds

Tests completed with return value 0.
```

当然，除了可用于测试基本逻辑条件的断言外，CUnit 还提供了名为 CU_PASS 和 CU_FAIL 的断言，可用于测试程序的执行流是否符合预期，比如下面这段示例测试代码：

复制代码

```
// ...
static jmp_buf buf;
void foo(void) {}
void test_longjmp(void) {
  int i = setjmp(buf);
  if (i == 0) {
    foo();
    CU_PASS("run_other_func() succeeded.");
  } else {
    CU_FAIL("run_other_func() issued longjmp.");
  }
}
```

```
13 // ...
```

这里，我们用上述两个断言测试了函数 foo 在执行时，是否调用 longjmp 改变了程序的执行流程。代码的逻辑十分简单，你可以结合我在 12 讲 中介绍的非本地跳转执行方式，来理解这个测试用例的具体执行过程。

测试框架的其他重要能力

关于 C 项目可以使用的更多单元测试框架，你可以点击这个链接查阅相关信息。

作为测试框架，CUnit 仅提供了与单元测试相关的最基础，也是最核心的功能，即通过设置断言来追踪目标单元（函数）在不同输入下的执行结果是否正确。但除此之外，TAP、Fixture、Generator，以及生成 Code Coverage 等能力，也是用于完善单元测试，甚至是支持其他类型测试进行的重要功能。接下来我将为你逐一介绍它们的基本作用。

其中，TAP（Test Anything Protocol）是一种标准协议，该协议定义了一种独立于编程语言的特定文本格式，可用于表示测试用例的执行结果。你可以观察下面的文本，对它的格式有个大致印象：

复制代码

```
1 1..4
2 ok 1 - Input file opened
3 not ok 2 - First line of the input valid
4 ok 3 - Read the rest of the file
5 not ok 4 - Summarized correctly # TODO Not written yet
```

TAP 通过一种可移植的方式，将测试结果从不同的测试框架中“抽离”出来，并将其展示格式统一化。这样做的好处在于，对测试结果进行统一的收集、分析和处理将变得更加容易。你可以点击这个链接，来了解有关 TAP 的更多信息。

Fixture 为我们提供了可以在测试用例执行前后，对测试环境进行准备和清理的能力。在简单的场景中，我们可能需要在测试进行前创建一些必要的模拟数据（比如通过 malloc 创建的堆对象），并且在测试结束后再对它们进行清理。而在一些复杂场景中，我们还可能需要提前将准备好的测试数据集填充到数据库中，然后在测试完成后，及时清空数据库内容，并断开连接。

在支持 Fixture 功能的 C 测试框架中，相关能力可能会通过宏函数，或通过实现带有特定名称回调函数的形式被提供。用户可以通过这些方式，来指定整个测试过程在不同时刻需要执行的不同任务。

至于 Generator，它的功能很直观地体现在它的名字上。借助 Generator，我们可以让测试代码根据指定规则来自动生成测试用例所需要使用的各类数据。通过这种方式，测试将变得更具动态性，且流程也更加自动化。

而 Code Coverage，即代码覆盖率，是用于衡量一个项目测试用例完备性的重要指标。该指标的值为当测试用例运行时，其所能够测试到的不同程序逻辑的比例。按照维度的不同，这个覆盖率可以被细分为函数覆盖率、指令覆盖率、判断覆盖率，等等。总的来看，代码覆盖率一定是越高越好。但现实情况中，并非所有类型的代码覆盖率都能到全覆盖，即 100% 覆盖。

最后，我们需要知道，并不是所有测试框架都支持我们上面提到的这些测试辅助功能，它们中的某些功能可以由另外的独立框架或库单独提供。

C 项目常用的其他测试

接下来，我们再来看看除单元测试外的其他几种常用测试类型。

集成测试

当组成程序的各个单元都能够正常工作时，我们便可以将测试的粒度进一步扩大，来看看当不同单元或模块被整合在一起时，它们是否也可以很好地协同工作。而这类测试通常被称为“集成测试（Integration Testing）”。

我给你举一个简单的例子：对于一个在线购物系统，我们通常会提供“用户注册”的功能，以便让系统使用者可以用合法的方式建立账户。假设这个功能对应于代码中名为 signUp 的函数，由于注册的流程涉及到接收网络请求、更新数据库记录等一系列操作，因此，该函数在内部实现时，便需要调用与网络和数据库操作，以及注册逻辑计算等功能相关的另外一些函数。而在这种情况下，当我们对 signUp 函数使用来自于真正外部依赖项的资源进行测试时，便是进行了一次集成测试。

可以看到，这类测试通常需要使用数据库、网络连接等真实的外部资源，因此在使用测试框架进行测试时，便需要利用框架提供的 Fixture 等功能，来在测试执行前正确地配置相关环境。

另外需要注意的是，**并非每一种测试类型都有与其直接对应的一种专用测试框架**。测试框架的目的只是为你提供进行测试的一系列相关能力。而用例的测试主体（如单个函数或多个函数）、测试的功能范围，以及进行方式（如使用桩代码和模拟资源，或使用真实系统环境）等因素，才综合决定了测试的所属类型以及相应目的。

因此，你可以直接使用支持 Fixture 功能的测试框架来实现 C 项目的集成测试。其中，Fixture 用于初始化测试用例需要使用的相关外部环境依赖，而断言则用于验证测试用例的执行结果。

功能测试

功能测试（Functional Testing）的目的与集成测试十分类似，但也稍有不同。相同点在于，两者在测试时都涉及程序的多个单元，并使用真实的外部依赖来提供测试所需要的资源；而不同点在于，功能测试对测试结果的正确性要求可能会更加严格。通常，这个结果需要满足业务需求中的相应规定。

举个例子：在集成测试中，我们在验证某个测试用例是否通过时，可能仅会关注用例中的函数在调用后，是否返回了某个类型的值。而在功能测试中，相应的用例在返回该类型值时，受限于业务需求的约束，值的具体表现形式（如浮点数值保留的位数、字符串值的固定宽度等）也必须满足相应要求。

功能测试的目标在于验证整个应用程序的全部功能，或部分子功能是否可以按照业务需求正常运作。它是软件在被正式提交或上线前确保其质量的最重要一环。

性能测试

除此之外，性能测试（Performance Testing）也是 C 语言项目必不可少的一个重要测试类型。

我在18 讲 中曾提到过，粗略来看，可以直接使用“运行时间”和“内存使用率”这两个指标来作为程序运行性能的度量单位。但除此之外，由于软件系统的整体架构可能有所不

同，因此，更多场景下的细分性能指标也显得十分重要。

在大多数情况下，我们可以使用 Perf 来进行 Linux 系统上的应用性能测试。Perf 是一个在 Linux 2.6 以上内核版本中添加的程序性能调试工具，功能十分强大。它提供了一系列常用且精细的性能指标，并抽象了不同 CPU 在硬件上的差异，以便更加统一地“评价”不同体系上的程序性能情况。

Perf 以命令行的形式使用，通过指定不同参数，我们可以测量程序在软件和硬件层面的多种性能指标，比如程序运行期间发生的页错误数量、经过的 CPU 时钟数、CPU 分支预测失败的次数，等等。当然，某些硬件指标是否可用，还要看对应 CPU 上的 PMU（Performance Monitoring Unit）单元是否支持。

比如，通过 `perf stats` 命令，我们可以查看某个程序在运行时，有关 CPU 计数器的一些关键信息。可能的输出结果如下图所示：

```
Performance counter stats for './test':

            0.55 msec task-clock                #    0.570 CPUs utilized
               0      context-switches          #    0.000 K/sec
               0      cpu-migrations            #    0.000 K/sec
              46      page-faults               #    0.084 M/sec
   <not supported>    cycles
   <not supported>    instructions
   <not supported>    branches
   <not supported>    branch-misses

       0.000962002 seconds time elapsed

       0.000000000 seconds user
       0.000996000 seconds sys
```

你可以点击这个链接，来查看有关上图中各个指标以及 Perf 工具的更多信息。

在 macOS 与 Windows 平台上，我们也可以相应地使用名为 Instruments 与 WPR 等工具，来进行类似的软件性能测试。当然，它们的具体使用方式和支持的测量指标可能有所不同，你可以点击上文中的相应链接来了解关于它们的更多信息。

其他测试

除了我在上面提到的几种常见测试类型外，其他测试方法还有兼容性测试、安全性测试、无障碍测试、端到端测试等，它们在不同的场景下也同样发挥着重要作用。而上面讲过的功能测试，通常也可以再被细分为冒烟测试、回归测试、健全测试等几种子类型，它们都对软件的功能进行测试，但侧重点却各不相同。

比如，对于敏捷开发模式下的软件交付，由于新功能会以固定周期的间隔不断迭代，如何合理应用回归测试来确保已有功能的正常运作，便成为软件交付中的重要一环。而在瀑布开发模式中，软件会作为一个整体进行交付，因此，如何通过完备的系统测试来保障用户功能工作正常，就相对更加重要。而在某些需要软硬件结合的项目中，受到软件与硬件开发模式不一致等因素的制约，就需要把更多不同类型的测试组合起来使用，以确保整个系统可以稳定运转。

“软件测试”是一个庞大而复杂的话题，在不同的软件开发策略和软件架构下，需要侧重应用的测试种类都会有所不同。而且，从实际情况来看，不同技术项目、不同团队，乃至不同开发者，通常都会对同一种测试类型有着不同的理解。因此，这里我不想为你输入过多的“测试专有名词”。你可以从这一讲中提到的几种最基本的测试类型开始实践，然后再不断体会它们在整个软件开发流程中的作用。

什么是“真正”的测试自动化？

通常来说，我们谈到的“测试自动化”是指使用脚本或测试框架，以编程的方式代替传统的人工方式，来对软件功能进行测试的过程。因此，**只要涉及了测试框架、软件或脚本的使用，我们就可以称这样的测试为自动化测试**。

但与之相比更有意义的，是如何进一步地把一些重要的测试环节融入到我们的日常开发和功能迭代中，来让整个 DevOps（即软件开发、质量保证以及系统运维）的过程更加自动化。

通常的做法是，让一些针对软件基础功能的测试（如单元测试、功能测试等）成为每一次生产发布前都必须执行的环节，并指定一定的覆盖率作为通过基准。而这一般要求公司提供针对某一类项目（比如 C 语言项目）的统一发布平台，且支持对它们进行集中的持续编译、测试、部署、发布等一系列操作，并让所有相关流程可以按“管道”的方式依次有序执行。这部分内容涉及企业基础 IT 设施架构，这里我就不详细展开了，你可以点击这个链接来获取更多信息。

总结

好了，讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

今天我主要介绍了与 C 语言项目有关的一些常用测试类型，包括单元测试、集成测试、功能测试与性能测试：

> 单元测试通常以函数作为单元，来测试组成程序的各个独立单元自己的功能正确性；
>
> 集成测试将测试的范畴进一步扩大，它测试当多个功能单元组合在一起时，是否还能够正常运作；
>
> 功能测试与集成测试类似，不过，它对测试结果的正确性要求会更加严格，而这些额外的正确性要求通常是与业务需求紧密相关的；
>
> 性能测试则是每一个软件在交付前都需要进行的重要环节，它可以保证软件的运作满足用户对软件响应性等指标的要求。

除了这几种常见的测试类型外，兼容性测试、安全性测试等其他测试类型也在一些特定场景下发挥着重要作用。

最后，我还介绍了什么是自动化测试，以及我认为的“真正的”测试自动化。自动化测试是指借助脚本或测试框架等工具进行软件测试的过程。相较于人工测试，自动化测试使得测试用例可以被频繁多次执行，且大大降低了人工成本。而更进一步，通过结合企业统一的应用发布平台，自动化测试的过程可以与整个 DevOps 结合得更加紧密。

思考题

你所在团队的项目是如何进行测试的，测试过程中又遇到了哪些困难？测试覆盖率会作为生产发布的严格要求吗？欢迎在评论区跟同学们分享你的经历。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

生成海报并分享

赞 3　提建议

上一篇　20 | 生产加速：C 项目需要考虑的编码规范有哪些？

下一篇　22 | 生产加速：如何使用结构化编译加速 C 项目构建？

精选留言 (1)

写留言

2022-02-07

推荐一个单元测试框架，cmocka，支持fixture，官方文档也比较详细

作者回复: 感谢推荐，看了下确实不错！

共 2 条评论 >