

27 | 深入浅出之动态测试方法

2018-08-29 茹炳晟

软件测试52讲

[进入课程 >](#)



讲述：茹炳晟

时长 16:26 大小 7.54M



你好，我是茹炳晟，今天我和你分享的主题是：深入浅出之动态测试方法。

相较于，静态测试方法是不需要实际执行代码去发现潜在代码错误的方法，我今天要和你讨论的动态测试方法，则是要通过实际执行代码去发现潜在代码错误的测试方法。

正如我在分享 [《不破不立：掌握代码级测试的基本理念与方法》](#) 这个主题时，将动态测试方法进一步划分为人工动态方法和自动动态方法，今天这次关于动态测试方法的分享，我也会从这两个方面展开。

由于自动动态方法并不能理解代码逻辑，所以仅仅被用于发现异常、崩溃和超时这类“有特征”的错误，而对于代码逻辑功能的测试，主要还是要依靠人工动态方法。

人工动态方法

人工动态方法，可以真正检测代码的业务逻辑功能，其关注点是“什么样的输入，执行了什么代码，产生了什么样的输出”，主要用于发现算法错误和部分算法错误，是最主要的代码级测试手段。

从人工动态方法的定义中，你可以很清楚地看出：代码级测试的人工动态测试方法，其实就是单元测试所采用的方法。所以，下面的分享，我会从单元测试方法的角度展开。

如果有一些代码基础，那么你在学习单元测试框架或者工具时，会感觉单元测试很简单啊，一点都不难：**无非就是用驱动代码去调用被测函数，并根据代码的功能逻辑选择必要的输入数据的组合，然后验证执行被测函数后得到的结果是否符合预期。**但是，一旦要在实际项目中开展单元测试时，你会发现有很多实际的问题需要解决。

我在专栏第 4 篇文章[《什么是单元测试？如何做好单元测试？》](#)中，已经分享过单元测试中的主要概念了，所以今天的分享我不会重复前面的内容，只和你分享前面没有涉及到的部分。如果你有哪些概念已经记不太清楚了，建议你先回顾一下那篇文章的内容。

接下来，我将和你分享单元测试中三个最主要的难点：

1. 单元测试用例“输入参数”的复杂性；
2. 单元测试用例“预期输出”的复杂性；
3. 关联依赖的代码不可用。

单元测试用例“输入参数”的复杂性

提到“输入参数”的复杂性，你应该已经记起了，我在前面的分享中提到过：如果你认为单元测试的输入参数只有被测函数的输入参数的话，那你就把事情想得过于简单了。

其实，这也是源于我们在学习单元测试框架时，单元测试用例的输入数据一般都是被测函数的输入参数，所以我们的第一印象会觉得单元测试其实很简单。

但是到了实际项目时，你会发现单元测试太复杂了，因为测试用例设计时需要考虑的“输入参数”已经完全超乎想象了。

我在[《什么是单元测试？如何做好单元测试？》](#)一文中已经总结了多种常见的单元测试输入数据，但是并没有详细解释每种输入数据的具体含义，你可能也对此感到困惑，那么今天我就结合一些代码示例和你详细聊聊这些输入参数吧。

第一，被测试函数的输入参数

这是最典型，也是最好理解的单元测试输入数据类型。假如你的被测函数是下面这段代码中的形式，那么函数输入参数 a 和 b 的不同取值以及取值的组合就构成了单元测试的输入数据。

 复制代码

```
1 int someFunc(int a, int b)
2 {
3     ...
4 }
```

第二，被测试函数内部需要读取的全局静态变量

如果被测函数内部使用了该函数作用域以外的变量，那么这个变量也是被测函数的输入参数。

下面这段代码中，被测函数 Func_SUT 的内部实现中使用了全局变量 someGlobalVariable，并且会根据 someGlobalVariable 的取值去执行 FuncA() 和 FuncB() 这不同的代码分支。

在做单元测试时，为了能够覆盖这两个分支，你就必须构造 someGlobalVariable 的不同取值，那么自然而然，这个 someGlobalVariable 就成为了被测函数的输入参数。

所以，在这段代码中，单元测试的输入参数不仅包括 Func_SUT 函数的输入参数 a，还包括全局变量 someGlobalVariable。

 复制代码

```
1 bool someGlobalVariable = true;
2 void Func_SUT(int a)
3 {
4     ...
```

```
5  if(someGlobalVariable == true)
6  {
7      FuncA();
8  }
9  else
10 {
11     FuncB();
12 }
13 ...
14 }
```

第三，被测试函数内部需要读取的类成员变量

如果你能理解“被测函数内部需要读取的全局静态变量”是单元测试的输入参数，那么“被测函数内部需要读取的类成员变量”也是单元测试的输入参数就不难理解了。因为，类成员变量对被测试函数来讲，也可以看做是全局变量。

我们一起看一段代码。这段代码中，变量 `someClassVariable` 是类 `someClass` 的成员变量，类的成员函数 `Func_SUT` 是被测函数。`Func_SUT` 函数，根据 `someClassVariable` 的取值不同，会执行两个不同的代码分支。

同样地，单元测试想要覆盖这两个分支，就必须提供 `someClassVariable` 的不同取值，所以 `someClassVariable` 对于被测函数 `Func_SUT` 来说也是输入参数。


 复制代码

```
1 class someClass{
2     ...
3     bool someClassVariable = true;
4     ...
5     void Func_SUT(int a)
6     {
7         ...
8         if(someClassVariable == true)
9         {
10             FuncA();
11         }
12         else
13         {
14             FuncB();
15         }
16         ...
17     }
```

```
18    ...
19 }
```

第四，函数内部调用子函数获得的数据

“函数内部调用子函数获得的数据”也是单元测试的输入数据，从字面上可能不太好理解，那我就通过一段代码，和你详细说说这是怎么回事吧。

 复制代码

```
1 void Func_SUT(int a)
2 {
3     bool toggle = FuncX(a);
4     if(toggle == true)
5     {
6         FuncA();
7     }
8     else
9     {
10        FuncB();
11    }
12 }
```

函数 `Func_SUT` 是被测函数，它的内部调用了函数 `FuncX`，函数 `FuncX` 的返回值是 `bool` 类型，并且赋值给了内部变量 `toggle`，之后的代码会根据变量 `toggle` 的取值来决定执行哪个代码分支。

那么，从输入数据的角度来看，函数 `FuncX` 的调用为被测函数 `Func_SUT` 提供了数据，也就是这里的变量 `toggle`，后续代码逻辑会根据变量 `toggle` 的取值执行不同的分支。所以，从这个角度来看，被测函数内部调用子函数获得的数据也是单元测试的输入参数。

这里还有一个小细节，被测函数 `Func_SUT` 的输入参数 `a`，在内部实现上只是传递给了内部调用的函数 `FuncX`，而并没有在其他地方被使用，我们把这类用于传递给子函数的输入参数称为“间接输入参数”。

这里需要注意的是，有些情况下“间接输入参数”反而不是输入参数。

就以这段代码为例，如果我们发现通过变量 a 的取值很难控制 FuncX 的返回值（也就是说，当通过间接输入参数的取值去控制内部调用函数的取值，以达到控制代码内部执行路径比较困难）时，我们会直接对 FuncX(a) 打桩，用桩代码来控制函数 FuncX 返回的是 true 还是 false。

这样一来，原本的变量 a 其实就没有任何作用了。那么，此时变量 a 虽然是被测函数的输入参数，但却并不是单元测试的输入参数。

第五，函数内部调用子函数改写的数据

理解了前面几种单元测试的输入参数类型后，“函数内部调用子函数改写的数据”也是单元测试中被测函数的输入参数就好解释了。

比如，当被测函数内部调用的子函数改写了全局变量或者类的成员变量，而这个被改写的全局变量或者类的成员变量又会在被测函数内部被使用，那么“函数内部调用子函数改写的数据”也就成为了被测函数的输入参数了。

第六，嵌入式系统中，在中断调用中改写的数据

嵌入式系统中，在中断调用中改写的数据有时候也会成为被测函数的输入参数，这和“函数内部调用子函数改写的数据也是单元测试中的输入参数”类似，在某些中断事件发生并执行中断函数时，中断函数很可能会改写某个寄存器的值，但是被测函数的后续代码还要基于这个寄存器的值进行分支判断，那么这个被中断调用改写的数据也就成为了被测函数的输入参数。

其实在实际工程项目中，除了这六种输入参数，还有很多输入参数。在这里，我详细分析这六种输入参数的目的，一来是帮你理解到底什么样的数据是单元测试的输入数据，二来也是希望你从本质上认识单元测试的输入参数，那么在以后遇到相关问题时，你也可以做到触类旁通，不会再踌躇无措。

理解了“输入参数”的复杂性，接下来我们再一起看看“预期输出”的复杂性表现在哪些方面。

单元测试用例“预期输出”的复杂性

同样地，单元测试用例的“预期输出”，也绝对不仅仅是函数返回值这么简单。通常来讲，“预期输出”应该包括被测函数执行完成后所改写的所有数据，主要包括：被测函数的返回值，被测函数的输出参数，被测函数所改写的成员变量和全局变量，被测函数中进行的文件更新、数据库更新、消息队列更新等。

第一，被测函数的返回值

这是最直观的预期输出。比如，加法函数 `int add(int a, int a)` 的返回值就是预期输出。

第二，被测函数的输出参数

要理解“被测函数的输出参数”是预期输出，最关键的是要理解什么是函数的输出参数。如果你有 C 语言背景，那么你很容易就可以理解这个概念了。

我们一起来看一段代码。被测函数 `add` 包含三个参数，其中 `a` 和 `b` 是输入参数，而 `sum` 是个指针，指向了一个地址空间。

如果被测函数的代码对 `sum` 指向的空间进行了赋值操作，那么在被测函数外，你可以通过访问 `sum` 指向的空间来获得被测函数内所赋的值，相当于你把函数内部的值输出到了函数外，所以 `sum` 对于函数 `add` 来讲其实是用于输出加法结果的，那么显然这个 `sum` 就是我们的“预期输出”。

如果你还没有理解的话，可以在百度上搜索一下“C 语言的参数传递机制”。

 复制代码

```
1 void add(int a, int b, int *sum)
2 {
3     *sum = a + b;
4 }
5 void main()
6 {
7     int a, b, sum;
8     a = 10;
9     b = 8;
10    add(a, b, &sum);
11    printf("sum = %d \n", sum);
12 }
```

第三，被测函数所改写的成员变量和全局变量

理解了单元测试用例“输入参数”的复杂性，“被测函数所改写的成员变量和全局变量”也是被测函数的“预期输出”就很好理解了，此时如果你的单元测试用例需要写断言来验证结果，那么这些被改写的成员变量和全局变量就是 assert 的对象。

第四，被测函数中进行的文件更新、数据库更新、消息队列更新等

这应该不难理解。

但在实际的单元测试实践中，因为测试解耦的需要，所以一般不会真正去做这些操作，而是借助对 Mock 对象的断言来验证是否发起了相关的操作。

关联依赖的代码不可用

什么是关联依赖的代码呢？

假设被测函数中调用了其他的函数，那么这些被调用的其他函数就是被测函数的关联依赖代码。

大型的软件项目通常是并行开发的，所以经常会出现被测函数关联依赖的代码未完成或者未测试的情况，也就是出现关联依赖的代码不可用的情况。那么，为了不影响被测函数的测试，我们往往会采用桩代码来模拟不可用的代码，并通过打桩补齐未定义部分。

具体来讲，假定函数 A 调用了函数 B，而函数 B 由其他开发团队编写，且未实现，那么我们就可以用桩函数来代替函数 B，使函数 A 能够编译链接，并运行测试。

桩函数要具有与原函数完全相同的原形，仅仅是内部实现不同，这样测试代码才能正确链接到桩函数。**一般来讲桩函数主要有两个作用，一个是隔离和补齐，另一个是实现被测函数的逻辑控制。**

用于实现隔离和补齐的桩函数实现比较简单，只需拷贝原函数的声明，加一个空的实现，可以通过编译链接就可以了。

用于实现控制功能的桩函数是最常用的，实现起来也比较复杂，需要根据测试用例的需要，输出合适的数据作为被测函数的内部输入。

自动动态方法

我们先来回顾一下，什么是自动动态方法。自动动态方法是，基于代码自动生成边界测试用例并执行来捕捉潜在的异常、崩溃和超时的测试方法。


自动动态方法的重点是：如何实现边界测试用例的自动生成。

解决这个问题最简单直接的方法是，根据被测函数的输入参数生成可能的边界值。

具体来讲，任何数据类型都有自己的典型值和边界值，我们可以预先为它们设定好典型值和边界值，然后组合就可以生成了。

比如，函数 `int func(int a, char *s)`，就可以按下面的三步来生成测试用例集。

1. **定义各种数据类型的典型值和边界值。** 比如，`int` 类型可以定义一些值，如 `int` 的最小值、`int` 的最大值、0、1、-1 等；`char*` 类型也可以定义一些值，比如 `""`、`"abcde"`、`"非英文字符串"` 等。
2. **根据被测函数的原形，生成测试用例代码模板，**比如下面这段伪代码：

 复制代码

```
1 try{
2   int a= @@@;
3   char *s = @@s@;
4   int ret = func(a, s);
5 }
6 catch{
7   throw exception();
8 }
```

3. **将参数 @a@和 @s@的各种取值循环组合，分别替换模板中的相应内容，即可生成用例集。**

由于该方法不可能自动了解代码所要实现的功能逻辑，所以不会验证“预期输出”，而是通过 `try...catch` 来观察是否会引发代码的异常、崩溃和超时等具有边界特征的错误。

总结

代码级测试的动态测试方法，可以分为人工动态测试方法和自动动态测试方法。其中人工动态测试方式，是最常用的代码级测试方法，也是我们在进行单元测试时采用的方法。

人工动态方法，也就是单元测试方法，通常看似简单，但在实际的工程实践中会遇到很多困难，总结来看这些困难可以概括为三大方面：

1. 单元测试用例“输入参数”的复杂性，表现在“输入参数”不是简单的函数输入参数。本质上讲，任何能够影响代码执行路径的参数，都是被测函数的输入参数。
2. 单元测试用例“预期输出”的复杂性，主要表现在“预期输出”应该包括被测函数执行完成后所改写的所有数据。
3. 关联依赖的代码不可用，需要我们采用桩代码来模拟不可用的代码，并通过打桩补齐未定义部分。

而自动动态方法，需要重点讨论的是：如何实现边界测试用例的自动生成。解决这个问题最简单直接的方法是，根据被测函数的输入参数生成可能的边界值。

思考题

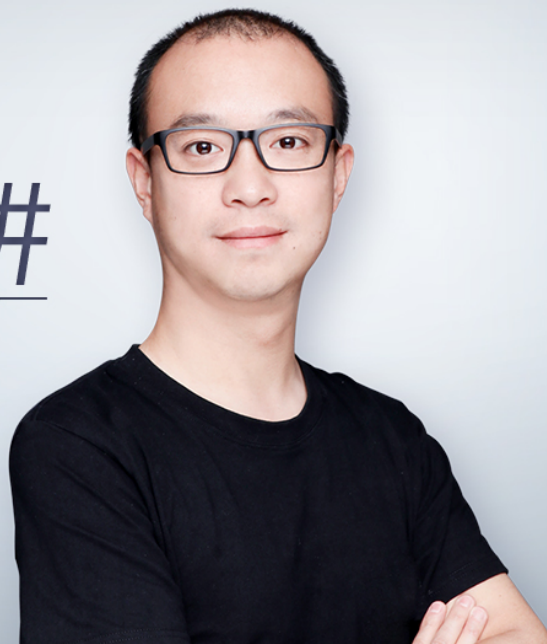
除了我们一起讨论的这些单元测试的难点，还有复杂数据初始化、函数内部不可控子函数的调用、间接输入参数的估算等难点。你在单元测试中是否遇到过这些问题呢，又是如何解决的？

感谢你的收听，欢迎给我留言一起讨论。

软件测试52讲

从小工到专家的实战心法

茹炳晟 eBay中国研发中心
测试基础架构技术主管



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 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 | 深入浅出之静态测试方法

下一篇 28 | 带你一起解读不同视角的软件性能与性能指标

精选留言 (6)

写留言



sylan215

2018-08-29

7

1.单元测试大部分还是需要开发做，如果测试做，那么测试人员的技术要求就比较高了；

2.可以折中的考虑是，测试提供单测需要覆盖的测试点，开发来保证测试的结果；

...

展开



小老鼠

2018-11-07

1

1，如果输入参数或数出参数是一个集合类、哈希类甚至是一个自定义类，作单元测试就复

杂了。

2, 若输出是个随机变量如何作单元测试, 比如一个随机函数的单元测试。

3, 为什么讲了那么多代码级的测试用例, 不介绍语句、条件、分支、条件分支、AC/DC、路径覆盖这些概念?

展开 ∨



口水窝

2019-04-23



感觉看理论都能看懂, 但是没有实践, 自己说也说不出所以然, 所以还是要实践, 转化为自己的想法。



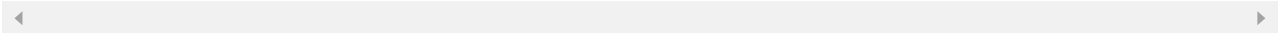
年轻人的瞎...

2018-12-29



感觉有点抽象, 可能要多看几遍。另外打桩这种是需要开发弄好还是需要测试处理?

作者回复: 这里讲的基本都是需要开发来做的测试



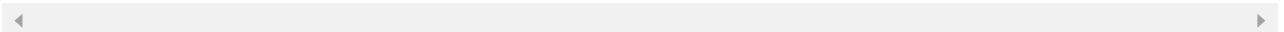
llyl

2018-09-07



如果是遇到程序要等很久才有响应的事件, 比如定时100小时后会发生什么, 那这个时间要怎么办呢? 测试的话不应该要等这么久的时间吧?

作者回复: 很长时间一般是调用了后台的异步处理操作, 这种情况可以考虑用mock解决, 即先只验证是否后台创建了必要的job, 而且相关的参数传递是否都正确, 其次单独的用例在验证job实现的正确性



Struggling

2018-09-03



除了mock还有其他好的方法吗? 期待老师指点

展开 ∨

作者回复: 本质上就是mock了, 但是mock的种类繁多, 有些会提供很多特有的功能



