Contents

Octave Vol.1——新手高效 Debug 建议

E-train Liu

2018-05-05

关键词：高效 debug，Octave，新手

“夜不美，代码太危险，总有人黑着眼眶熬着夜”。哦，不对，应该是“总有‘猿’黑着眼眶熬着夜”，程序猿。

上个周末到现在，我的微信里几乎让我帮着 debug 的。很显然，目前对于大多数人，除了没有思路，写出的代码跑不过也是相当大的一个问题。实际上，debug 困难，一是因为开发经验不足，容易掉坑；二是因为没有良好的 debug 技巧，掉进坑里扑棱半天爬不出来。对于第一个问题，解决方法当然是多进行一些编程积累经验，这个肯定需要时间，coursework 的 DDL 就在眼前了，这招估计赶不及。显然，第二种方法对于大部分人来说，是在短时间内快速提升的方法。

这篇很早就开始写了, 零零碎碎写了好几篇，本来想暑假发。但是现在大家拔光了头发也找不到 bug 的情形让我想起我刚入坑时“编程 5 分钟，纠错两小时”的惨痛经历。感同身受

之余，草草把这个合成一篇放了出来，希望解大家 CW 的燃眉之急。文章比较长，大家可以先看目录，再决定看哪一部分。如果发现有问题，欢迎在 Issue 中报给我，感谢不尽。

1 三大类 Bug

虽然程序写的不怎也样，说说行业内部的黑话装个逼还是很必要的。Bug 这个词的发明人是计算机上古时期的大佬 Grace Hopper 和她的团队，他们的计算机电路有一次被飞进去的虫子弄坏了，于是他们就用 bug 表示计算机错误，debug 表示找错。

正如能弄坏电脑的虫子有很多种，能搞乱程序的 bug 也是有分类的。知道 bug 的种类有助于 debug。

第一种叫做语法错误，就是写出了不符合语法规范的代码，导致计算机不认识了。比如说写错变量名、函数名，少写了”`endif`，还有令“生灵有倒悬之危”的赋值语句 `1 = a` 等等。

第二种叫语义错误。简而言之就是逻辑有问题的程序。比如要求把 `a` 和 `b` 两个变量的和赋给 `c`，但是代码写成了 `c = a * b`，这类程序往往计算机能正常运行，但是运行结果不对。

第三种叫做运行错误。这类问题的原因一般是傻逼用户和穷逼系统。如果你写了一个递归法算一个数的迭乘的程序，有人却往里面输了一个字符，或者递归太多系统内存不够了，都属于运行错误。

2 解决语法错误：善用错误和警告

2.1 错误和警告

如果解释器报告错误（error），说明程序中有一个 bug，这个 bug 严重到导致程序不能继续运行。如果发出的是“警告”，说明解释器发现了并不影响运行，但可能有错误的代码。或者是说这样的代码有造成问题的可能。

2.2 善于利用错误和警告

有些人程序一出错，立刻就要去抱大佬的们的大腿。然后各种承诺，请星巴克，求情，卖萌，递女装，自己女装/男装……咳咳，各种方法都用了。然而解释器早已经看透了一切，并且给出了高质量的错误说明。不会看解释器的报错是会严重拉低编程效率的。

例如编写下面这样一个程序

```
1 function result = isEven1(number)
```

```
    % There is an error in this function.
    if mod(number, 2) === 0
        result = true;
    else
        result = false;
    endif
endfunction
```

运行结果如下:

```
>> isEven(4)
parse error near line 3 of file /home/user/octaveFiles/isEven1.m

  syntax error

>>>     if mod(number, 2) === 0
                           ^
```

第一行是输入的命令，不用管。注意看第二行里有“near line 3”的字样，说明错误出在第三行上。倒数第二行是报错的代码，在最下面一行，还用“^”标出了错误的位置。这个报错还是相当贴心而准确的。

还是相似的代码，当然还是有错误的

```
function result = isEven2(number)
    % There is an error in this function.
    if mod(number, 2) = 0
        result = true;
    else
        result = false;
    endif
endfunction
```

运行结果如下

```
>> isEven2(4)
warning: suggest parenthesis around assignment used as truth value
   near line 3, column 23 in file '/home/user/octaveFiles/isEven2.m'
ans = 0
```

WTF？如果你直接看下面的 `ans` 的话，会发现运行结果是 0（`false`），难道 4 不是偶数吗？显然现在程序能运行，但是有 bug。来看一下上面的 warning:

```
2: warning: suggest parenthesis around assignment used as truth
   value near line 3, column 23
```

意思是说，建议在 3 行 23 列处插入能作为“true”的语句，这说明那个原来的语句的值永远为“false”。那么我们滚去源代码的第 3 行看一下

```
3:          if mod(number, 2) = 0
```

看出问题来了吗？也许你不能明白这个语句的值为什么永远是“`false`”，但你应该能发现，这里赋值符号“=”被误用作了判断相等的“==”。可见，虽然警告不影响运行，但并不代表程序没有错。而这里，利用警告，我们成功找到了错误的点。

那为什么这个语句永远是“`false`”？这里要提到一个补充知识点，赋值语句也是有返回值的。只要赋值成功，那么就会返回等号右边的值。我们可以用下面的代码直接在 octave 的终端里验证一下：

```
>>printf("%i\n", (a = 5));
```

输出结果是

```
5
```

这说明 `(a = 5)` 这整个式子的值是 5。验证了我们刚才的说法

至此，我们可以积累一个经验，**一旦程序报出要“插入真值或假值”那样的警告，我们就应该考虑是不是混用了“=”和“==”。**

2.3 报错位置不一定准确

刚才说过，报错时会显示行号甚至是位置，但这不一定是好事。有时候会帮助我们找到错误，但有时候会误导我们。

例如下面这个程序

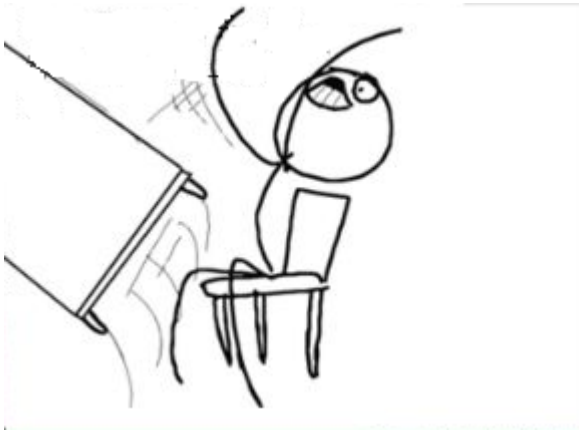
```
function printNumbers(number)
% There is an error in this program.
    for i = 1 : number
        disp(i);
    for j = number : -1 : 1
        disp(j);
    endfor
endfunction
```

简单说一下，这个程序就是从 1 打印到 number，再从 number 打印到 1。运行结果如下：

```
>> printNumbers(12)
parse error near line 7 of file /home/user/octaveFiles/printNumbers
   .m

'for' command matched by 'endfunction'

>>> endfunction
    ^
```



坑爹啊，“'for' command matched by 'endfunction'”？电脑你有病吧，乱划鸳鸯谱把这两个配一对干什么啊！

然后再看报错，说问题出在第 7 行第 1 个字符上，去看一下

```
7: endfunction
```

然而检查了 *n* 遍后，发现这句并没有拼写错误什么的。其实这时，如果你死板的盯着报错里报的第 7 句看的话，即使过了 DDL 你也不会看出任何问题的。但是如果你浏览整篇代码，你倒是有可能发现，第 3 行的 `for` 缺少了对应的 `endfor`，即第 4 和第 5 行间应该加一个 `endfor`。

那为什么不直接在第 4 行报错呢？我们来模拟一下 Octave 解释器逐行读取命令（“»”的位置是读取的位置）。

```
>>  function printNumbers(number)
    % There is an error in this program.
        for i = 1 : number
            disp(i);
        for j = number : -1 : 1
            disp(j);
        endfor
    endfunction
```

第一步，如上面的代码，解释器先读到了一个 `function`。

```
    function printNumbers(number)
    % There is an error in this program.
>>      for i = 1 : number
            disp(i);
        for j = number : -1 : 1
            disp(j);
        endfor
    endfunction
```

接着，解释器读到了第 1 个 `for`。

```
    function printNumbers(number)
    % There is an error in this program.
        for i = 1 : number
            disp(i);
```

```
>>      for j = number : -1 : 1
            disp(j);
        endfor
    endfunction
```

此时，程序读到了第 2 个 `for`。由于我们漏写了 `endfor`，程序没有读到 `endfor` 就会认为第 1 个 `for` 没有结束，而第 2 个 for 是嵌套在第 1 个 for 里的。

```
    function printNumbers(number)
    % There is an error in this program.
        for i = 1 : number
            disp(i);
        for j = number : -1 : 1
            disp(j);
>>      endfor
    endfunction
```

程序读到了一个 `endfor`，所以解释器知道第 2 个 `for` 循环的代码至此结束。

```
    function printNumbers(number)
    % There is an error in this program.
        for i = 1 : number
            disp(i);
        for j = number : -1 : 1
            disp(j);
        endfor
>>  endfunction
```

程序读到了 `endfunction`。应该来说，这时候程序应该知道 `function` 结束了。但是 Octave 比较特殊，当一个 `*.m` 文件中只有一个 `function` 时，对于 Octave 来说有没有 `endfunction` 都差不多（但我并不建议不写 `endfunction`!）。这时程序认为相比于缺少一个 `function` 的结尾，缺少一个 `for` 的结尾更要命，因此认为代码的问题是把与第一个 `for` 的结尾 `endfor` 写成了 `endfunction`。于是出现了刚才画风清奇的报错结果。

By the way，有些版本的 Octave 对于上面那个程序的报错是“`'endfor' command matched by 'endfunction'`”。我不是特别明白其中的原理，为什么要把两个 `end` 配对，再次不敢妄言。可能是现在社会开放了，连编译器的取向都多样化了 [手动滑稽]。Anyway，如果有大佬知道其中原理的话，欢迎在 Issue 中分享一下。

至此我们又可以涨一个经验：**当程序对某行报错，但该行没有问题时，说明可能是代码中出现了匹配错误。我们此时就不应该仅拘泥于报错的行数，而应该浏览整篇代码中需要匹配 end 的语句。**

这样的经验十分重要。这能让我们更快的找一些坑爹指数高的 bug。

例如下面一个猜数字的游戏。

```
function guessNumber1()
% 一个让用户猜0-10整数的游戏
% 程序中有错误
    answer = round(10 * rand());
    %随机生成一个0-10的整数
    guess = input("Please input an integer in [0,10]: ");
    while guess != answer
        % 获取用户输入的猜测
        if guess < answer
            disp("Too small! Try again.");
        else if guess > answer
            disp("Too large! Try again.");
        endif
        guess = input("Please input another integer in [0,10]: ");
        % 重新再获取一次用户的输入
    endwhile
    disp("Congratulations! Your guess is correct!");
endfunction
```

这个程序运行结果如下

```
parse error near line 16 of file /home/user/octaveFiles/
   guessNumber1.m

    'endif' command matched by 'endwhile'

>>> endwhile
    ^
```

一个没有任何经验的人如果看到这样的结果，第二天可能就有黑眼圈了。但是历史惊人的相似，报错行号不对，且又出现了乱配对的情况。我们已经有了上面的经验，可以直接推断，

程序中出现了匹配错误。这样，我们就只看程序中需要“end”的语句就行了。

其中，function，while 和 if 语句都需要 if。function 和 while 比较简单，仔细检查一下就发现没错。那么下一步就需要好好抠一下这个 if 语句了。

```
1  if guess < answer                    % Line 9
2      disp("Too small! Try again.");   % Line 10
3  else if guess > answer               % Line 11
4      disp("Too large! Try again.");   % Line 12
5  endif                                % Line 13
```

老实说，这个 bug 确实有点难了。对于这样的 bug，一个土方法是根据解释器的报错猜测性的稍微改动一个地方，发现不行的话，再找另一个地方改。虽然这个方法可能很多人不用教都会，但在毫无头绪是也确实管事。当然，当我们对一个地方的错误总看不出来时，可以思考一下是不是我们本身的认知就有问题呢？带着死马当活马医的心态，我们看一下官方 help 中对 if 和 else 的解释。

```
1  >> help if
2
3  -- if (COND) ... endif
4  -- if (COND) ... else ... endif
5  -- if (COND) ... elseif (COND) ... endif
6  -- if (COND) ... elseif (COND) ... else ... endif
7      Begin an if block.
8
9            x = 1;
10           if (x == 1)
11             disp ("one");
12           elseif (x == 2)
13             disp ("two");
14           else
15             disp ("not one or two");
16           endif
17 ...
```

发现官方例子中的 elseif 和我们的的不同之处了吗？官方的是“elseif”，两个单词连写，中间没有任何间隔。而我们的是“else if”，中间加了一个空格。

原来如此！“elseif”本身就是一个完整的命令，它是一个“if”语句的一部分。它应该长这样

```
if condition1
    statement1
elseif condition2
    statement2
else
    statement3
endif
```

当我们加了空格之后，就相当于把它拆成了一个“else”和一个“if”。而拆出的 if 此时相当于被包含在 else 中，它本身是一个独立的 if，还需要一个 endif。就像下面这熊样

```
if condition1
    statement1
else  if condition2
        statement2
      else
        statement3
      endif
endif
```

当然，这样写是不符合代码规范的，小朋友们千万不要学哦，会被其他读代码的人打的哦。正常情况下应该写成下面这样：

```
if condition1
    statement1
else
    if condition2
        statement2
    else
        statement3
    endif
endif
```

另外，那个猜数字的程序里获取用户输入的语句出现了两次，实际是写麻烦了。“do...until”语句了解一下。