# 翻译作业

1. **原文**

**6.2.3 When the Going Gets Tough**

Joseph P. Kennedy, father of President Kennedy, reputedly instructed his children, “When the going gets tough, the tough get going.” But he never debugged a piece of software. This subsection contains a few pragmatic hints about what do when the debugging gets tough.

• Look for the usual suspects. E.g., have you

o Passed arguments to a function in the wrong order,

o Misspelled a name, e.g., typed a lowercase letter when you should have typed an uppercase one,

o Failed to reinitialize a variable,

o Tested that two floating point values are equal (==) instead of nearly equal (remember that floating point arithmetic is not the same as the arithmetic you learned in school),

o Tested for value equality (e.g., compared two lists by writing the expression L1 == L2) when you meant object equality (e.g., id(L1) == id(L2)),

o Forgotten that some built-in function has a side effect,

o Forgotten the () that turns a reference to an object of type function into a function invocation,

o Created an unintentional alias, or

o Made any other mistake that is typical for you.

• Stop asking yourself why the program isn’t doing what you want it to. Instead, ask yourself why it is doing what it is. That should be an easier question to answer, and will probably be a good first step in figuring out how to fix the program.

• Keep in mind that the bug is probably not where you think it is. If it were, you would probably have found it long ago. One practical way to go about deciding where to look is asking where the bug cannot be. As Sherlock Holmes said, “Eliminate all other factors, and the one which remains must be the truth.”

• Try to explain the problem to somebody else. We all develop blind spots. It is often the case that merely attempting to explain the problem to someone will lead you to see things you have missed. A good thing to try to explain is why the bug cannot be in certain places.

• Don’t believe everything you read. In particular, don’t believe the documentation. The code may not be doing what the comments suggest.

• Stop debugging and start writing documentation. This will help you approach the problem from a different perspective.

• Walk away, and try again tomorrow. This may mean that bug is fixed later in time than if you had stuck with it, but you will probably spend a lot less of your time looking for it. That is, it is possible to trade latency for efficiency. (Students, this is an excellent reason to start work on programming problem sets earlier rather than later!)

**6.2.4 And When You Have Found “The” Bug**

When you think you have found a bug in your code, the temptation to start coding and testing a fix is almost irresistible. It is often better, however, to slow down a little. Remember that the goal is not to fix one bug, but to move rapidly and efficiently towards a bug-free program.

Ask yourself if this bug explains all the observed symptoms, or whether it is just the tip of the iceberg. If the latter, it may be better to think about taking care of this bug in concert with other changes. Suppose, for example, that you have discovered that the bug is the result of having accidentally mutated a list. You could circumvent the problem locally (perhaps by making a copy of the list), or you could consider using a tuple instead of a list (since tuples are immutable), perhaps eliminating similar bugs elsewhere in the code.

Before making any change, try and understand the ramification of the proposed “fix.” Will it break something else? Does it introduce excessive complexity? Does it offer the opportunity to tidy up other parts of the code?

Always make sure that you can get back to where you are. There is nothing more frustrating than realizing that a long series of changes have left you further from the goal than when you started, and having no way to get back to where you started. Disk space is usually plentiful. Use it to store old versions of your program.

Finally, if there are many unexplained errors, you might consider whether finding and fixing bugs one at a time is even the right approach. Maybe you would be better off thinking about whether there is some better way to organize your program or some simpler algorithm that will be easier to implement correctly.

1. **译文**

**6.2.3 当事情变得困难的时候**

据说肯尼迪总统的父亲约瑟夫·帕特里克·肯尼迪教导他的孩子们说：“当事情开始变得棘手的时候，那些坚定的人将继续前进。”即使他不曾运行调试过任何软件。下面的部分将在程序调试陷入困境时，为你提供一些有效的解决问题的提示。

注意下面常见的问题。比如：你是否曾经

1．在参数传递给函数的时候，参数的顺序发生了混乱；

2．拼错名字，比如：在你需要输入大写字母时，却输成了小写；

3．重载变量失败；

4．判断两个浮点数是否完全相等(= =)而不是约等于(请记住,浮点运算 不一样你在学校中所学到的算法),

5．在你想要检测两个对象是否相等时（例如：id（L1）==id（L2））， 却做成了检测它们的数值是否相等（例如：通过表达式L1==L2来比 较两列数据）；

6．忘了一些内置函数有副作用；

7．忘记了“（）”的引用一个对象类型的函数变成一个函数调用,

8．创建了一个没有配置过的别名,

9．犯一些你自己经常会犯的错误。

不要问为什么程序不能按照你的意图运行。相反，你要问自己程序为什么那样运行，这应该是一个很容易回答的问题，并且很有可能是修复程序迈出得很好的一步。

请记住错误可能不是你认为它在哪里就在哪。如果它在你想的地方,你可能很久以前就发现它了。一个实用的方法去确定错误在哪就是去看那些错误不可能在的地方。福尔摩斯说:“排除所有其他因素,剩下的一个必然是真相。”

试着向别人解释这个问题。我们都发展盲点。通常情况下,试图向别人解释一个问题往往会使得你看到一些你忽略的事情。试图解释为什么错误不能在某些地方也往往是一件有益的事情。

不要相信任何你看到的的事情，尤其是不要相信一些文件，代码可能不会按照指示运行。停止调试并重新编写文档。这将使得你可以从不同的角度看问题

停下来，明天重新编写。这可能意味着相比于你继续寻找错误，你可能会花更少的时间来改正这个错误。也就是说,它是可能的贸易延迟效率。(学生早开始这样练习编程习题是一个很好的办法!)

**6.2.4 当你发现的问题所在**

当你认为你已经发现了你编码中的问题所在的时候，那种想要重新编码和测试修复的诱惑几乎是不可抗拒的。然而,放慢一点往往是更好的选择。要记住,我们的目标不是解决一个错误,而是能够使没有错误的程序更快更高效的运行。

问问自己这个错误能否解释了所有观察到的异样,还是这只是冰山的一角。如果是后者,你最好考虑到修改这个错误的同时要与其他的变化联系到一起。举个例子，假设你已经发现错误来自于不小心改变了一个列表，你可以局部地回避这个问题（也许是通过将列表进行备份），或者可以考虑用元组来替代列表（因为元组是保持不变的），这样也许能够消除代码中其它部分的相似错误。

在做任何改变之前，尝试并了解这个修改可能衍生出的问题。它会破坏其它部分吗？它会使程序复杂化吗？它能为你整理程序的其它部分提供更好的机会呢？

要永远确保自己能够回到修改之前。意识到一系列的修改让你远离目标越来越远，并没有办法回到你开始的地方。没有什么比这更加让人沮丧的了。磁盘的空间往往是很充足的，记得用它来储存旧版本的程序

最后，如果出现了许多难以解释的错误，你应该思考一下逐个找到错误并加以修改是不是正确的方法。也许你应当思考是否有更好的方法去组织你的程序，或者寻找一些更简单算法，这样将会更容易实现。