**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 当情况变得艰苦时**

据说肯尼迪总统的父亲，Joseph P. Kennedy在教育他的孩子们时说过：“当情况变得艰苦的时候，只有那些吃苦耐劳的人才能继续前进。”虽然他不曾调试过任何软件。这个小节包含了一些当调试情况变得困难的时候的一些实用的提示。

以下是一些常见的问题，比如说你是否曾经

1. 将参数传递给函数时弄错了顺序；
2. 拼错一个特定的词，比如说在你应该打一个大写字母的时候却不小心打了一个小写字母；
3. 重置变量失败；
4. 检测两个浮点数完全相等而不是约等于（请记住浮点数的算法并不是完全和你在学校所学的其他类型的数的算法一样）；
5. 当你检测两个对象相等时（如：id（L1）==id（L2）），然而你用却使用L1==L2这个比较数值相等的表达式来检测；
6. 忘记了一些内置函数的有副作用；
7. 忘记()会使你对函数类型的对象的引用变成函数的调用；
8. 创建了一个没有意义的对象别名；
9. 或者你自己常犯的典型错误。

不要再继续苦恼为什么程序不按你设想的方式运行下去。而是应该去研究为什么它会以这样的方式运行。后一个问题会比较容易回答，这将会是你怎样去修复程序所迈出第一步。

记住错误可能不在你想当然的地方。如果是的话，你肯定早就找到它了。在不可能发生错误的地方去找错误恰恰是一个实用的找错误在哪里的好方法。就像夏洛克福尔摩斯所说：“当排除所有事实之后，剩下来的那个一定就是事实。”

尝试去与别人交流你的问题，毕竟我们每一个人都有盲点。事实也通常是这样，当你与别人交流你的问题的时候往往会引导发现一些你所忽略的东西。最好的方法就是尝试去解释为什么错误会出现在这个意外的地方。

不要盲目的相信你所阅读的一切资料（特别是一些参考文献）。你自己所写的代码可能不会像文献中那样运行。

停止调试，静下心来记录一些问题。这将会让你从不同的方面发现你问题的解决方法。

暂时放下，明天继续尝试。这也许会比你一直坚持做下去解决这个问题的时间更晚，但是这通常会花费更少的时间来发现问题的所在。也就是说，我们可以牺牲一些东西来换取效率。（同学们，这是一个很棒的理由去更早而不是更晚去做编程错误集的理由!）

**6.2.4 当你找到了问题的所在**

当你认为你已经找到了代码中的错误的时候，产生的编写代码和调试的冲动是无法抗拒的。然而，静下心慢慢来通常会更好。请记住我们的目标并不是单纯的修改这个程序的错误，而是要更快和更有效率地向着程序无错化前进。

问问你自己这个错误能否解释所有的问题还是这仅仅是其中的冰山一角。如果是后者的话，最好在修改错误的时候保持它和其它的变化一致。比如说，你已经发现这个错误是由于你一不小心修改了一个列表所致，你可以局部地回避这个问题（也许是进行列表备份），或者你可以用元组来代替列表（因为元组是保持不变的），这样也许可以消除在代码的其他地方发生类似的错误。

在做出改变之前，尝试和理解怎么样去修复因为改变而衍生出来的其他问题。这样修改会破坏其他的代码吗？它会使程序变得更为复杂吗?这样的调整是否会提供调整其他部分代码的机会？

确保你可以回到你现在所在的地方。没有什么比当你进行了一系列的改变之后让你离目标更远并且你还没有办法回到原来的地方更沮丧了。磁盘的空间很充足，希望你可以利用它来储存你的旧代码。

最后，如果仍有很多解释不了的错误，你也许可以考虑一下逐个找到错误并修复它是否是一个正确的方法。也许思考是否有更好的方式去组织你的程序，或者寻找一些使你的程序更准确和更容易实现的算法会使这个情况变得好一点。

**个人小结**

以前基本上不会接触这些翻译工作，不过感觉这次经历对自己的帮助挺大的，就当为未来的毕业论文的翻译方面做点铺垫。在这次翻译过程中，我发现了自己对于编程这方面的英语单词很不熟悉，虽然基本能够看懂文中叙述的大意，这类英语教材和中文教材的区别还是蛮大的，以后我会争取多看一些英文原版教材。

在编程问题方面，这学期还没有好好的写过一个程序，不过以大一大二时对C++编程时的经历来说，这段文章确实能对自己在修改bug起到引导的作用。那时候也经常会犯一些拼错，比如说拼错自己定义的变量、函数调用时参数的问题、思维固定化不能很好的优化程序。在寻找错误的时候，我们通常会去找那些逻辑方面的错，然而往往这个是没问题的，出现错误的地方经常是那些我们忽略的细小的地方（比如说字母问题，标点问题，=与==搞错等）。这些都告诉了我们要注意细节，细节决定成败，这些小细节往往决定着我们能否有块有准确地写出我们所需要的程序。

保留原代码是一个很必要的行为，要不然在你没有找到正确方向的时候随便改代码之后你会越走越远，越改越乱，所以每次修改时的代码需要保存起来以备不时之需。

有时候全身投入的纠结于某个错误，不肯停下来，或许还不如暂时放下，静下心来，走出去散散步，放空一下脑子之后再回去换一种方式去想想效率来的高。