第三章笔记

2019年9月7日 星期六　　下午7:08

One powerful design strategy, which is particularly appropriate to the construtions of programs for modeling physical systems, is to base the structure of programs on the structure of the system being modeled. For each object in the system, we construct a corresponding computational object. For each system action, we define a symbolic operation in our computational model. Our hope in using this strategy is that extending the model to accommodate new objects or new actions will require no strategic changes to the program, only the addition of the new symbolic analogs of those objects or actions. If we have been successfully in our system organization, then to add a new feature or debug an old one we will have to work on only a localized part of the system.

The difficulties of dealing with objects, change, and identity are a fundamental consequence of the need of grapple with time in our computational models. These difficuties become even greater when we allow the possibility of the concurrent execution of programs.

An object is said to "have state" if its behavior is influenced by its history.

Encapusulation reflects the general system-design principle known as the hiding principle: hiding principle: one can make a system more modular and robust by protecting parts of the system from each other; that is, by providing information access only to those parts of the system that have a "need to know".

Soon as we intrduce assignment into our language, substition is no longer an adequete model of procedure application.

We have already used begin implicity in our programs, because in Scheme the body of a procedure can be a sequence of expressions. Also, the <consequent> part of each clause in a cond expression can be a sequence of expressions rather than a single expression.

Two calls to the same procedure with the same arguments always produced the same result.

Actually, this is not quite true. One exception is random number generator. Another exception involved the operation /type tables we introduced in section 2.4.3, where the values of two calls to get the same arguments depended on intervening calls to put. On the other hand, until we introduce assignment, we have no way to create such procedures ourselves.

Set!操作使我们可以去模拟带有局部状态的对象。然而，这一获益也有一个代价，它使我们的程序设计语言不能再用代换模型解释了。进一步说，热河具有“漂亮”数学性质的简单模型，都不可能继续适合作为处理程序设计语言的对象和赋值的框架了。

只要我们不适用赋值，以同样参数对同一过程的两次求值一定产生出同样的结果，因此就可以认为是在计算数学函数。向我们在本书的前两章所作的一样，不用任何赋值的程序设计成为函数式程序设计。

这里的麻烦在于，从本质上来讲，代码的最终基础就是，这一语言里的符号不过是作为值的名字。而一旦引进了set!和变量的值可以变化的想法，一个变量就不再是一个简单的名字了。现在的一个变量索引着一个可以保存值的位置，而存储在哪里的值也是可以改变的。

如果一个语言支持在表达式里“同一的东西可以相互替换”的观念，这样替换不会改变有关表达式的值，这个语言就称为具有引用**透明性**。在我们的计算机语言里包含了set!之后，也就打破了引用透明性，就使确定能否通过等价的表达式代换去简化表达式变成了一个一场错中复杂的问题了。由于这种情况，对使用赋值的程序做推理也将变得极其苦难。

有了前面有关“同一”和“变化”的论述，如果peter和Paul只能检查他们的银行账户，而不能执行修改余额的操作，那么看清这两个账户是否不同的问题就需要仔细讨论了。一般而言，如果我们绝不修改数据对象，那么就可以将一个符合数据对象完全看作是由其片段组成的一个整体。例如，一个有理数是完全由它的分子和分母确定的。如果出现了修改，这以观点也就不再合法了，此时符合数据对象有了一个“标识”，而它又是与组成这一对象的各片段都不同的东西。及是我们通过提款修改了一个账户的余额，这个账户依然是“同一个”账户。于此相反，我们也可能有两个银行账户，它们具有相同的状态信息。这种复杂性是将应工行账户看作一个对象而产生的结果，而不是程序语言的问题。例如，通常我们不将一个有理数看作具有标识的可修改对象，并不像修改其分子并保持“同一个”有理数。