什么是系统

系统的定义

系统软件是什么？一套计算机应用软件通常是由多个不同的系统组合起来的，这包括操作系统、网络交换系统、数据管理系统等。对软件开发人员而言，他们更多的时间是在定义这些系统应该如何工作，而并不是从底层开始编写程序。从某种意义上讲，应用软件需要的大多数基本功能都可以由这些系统提供。软件开发人员只是调度师，他们编写调度程序，指挥这些系统协同工作，实现软件的高级功能。系统软件在本书中简称为**系统**，它们是一个能个独立运行的模块，是应用软件的组成部分，提供大部分软件共同需要的基本功能。它们最原始的目的是简化软件开发工作，即将繁琐的软件工程过程替换为简单的系统使用过程，从而提升软件开发的效率。

（以上的描述可能和传统的系统软件定义有所区别。按照传统的理解方式，系统软件更像一个平台，用于管理各种硬件设备，同时为上层的应用软件提供资源和服务。操作系统就是典型的例子。但传统的定义并不适合于描述数据管理系统，因为后者并不像平台。简单将系统软件理解为功能部件更利于把握它的要义。）

既然系统是软件的功能部件，而且是通用的，那么它的设计就必须考究。它如果稳定性差或者性能低下，用它构建出来的软件就无法运行。它如果使用起来不方便，就会降低软件开发人员的工作效率。更为重要的是，它如果提供的功能不够通用或者不够齐全，就会为软件开发工作造成困扰，甚至沦为无用。那么，如何衡量一个系统的优劣呢？这是一个很重要的问题，也是我们希望读者在完成本书的学习之后能够解答的问题。但它的答案是复杂的。本书会在不同的章节从不同的角度提供这个问题的答案。我们首先从软件模块化的角度出发，谈一谈如何评价一套系统设计得好还是差。

一个好的系统一定是一个好的模块

所谓软件模块化（modularization），就是将软件的总体功能分解成多个子功能，以便分而治之。模块化是软件开发过程中的一个自然想法。当我们还是程序设计的初学者时，就开始不知不觉使用模块化了。凡是复杂一点的程序，我们都会将其功能分解，然后分别编写各个模块的程序，这是人的思维习惯。在大型软件的开发中，模块化更是一个关键环节，它不仅有利于团队的分工合作，也有利于代码的维护。然而，即使模块化是一种自然想法，我们并不是天生就能将它做好。一个程序的模块化方式很多，不是每一种方式都有利于软件开发。所幸前人已经为我们总结出了好的模块化的标准。

按照前面对系统的初步定义，一个系统实际上就是应用软件的一个模块，只不过它比普通的软件模块更为通用。那么，要衡量一套系统设计得如何，我们首先要考究它是否是一个合理的模块，也就是说，它是否经得起模块化标准的考验。

究竟什么是好的模块化?

衡量模块化的尺子很多。其中的核心标准应该是David L. Parnas教授在上世纪七十年代提出的“信息隐藏”（Information Hiding）标准。这里用一个简单的例子来说明这一项标准。这个例子出现在由David L. Parnas 1972年发表的一篇经典论文中[1]。

KWIC索引问题：

> 输入为一张有序的列表，表中的每一行是一个单词序列，而每一个单词是一个字母序列。我们可以对任意一行单词实施滚动操作。一次滚动是将该行的首个单词取出，再追加到该行的末尾。滚动操作可以不断重复，直到这一行单词恢复初始状态。对一行单词而言，每一次滚动的结果称为该行的一个“位移”。KWIC索引的功能就是对列表中所有行的所有“位移”进行穷举，并将它们按照字母顺序输出。

用一段程序实现KWIC索引并不难。关键是如何将这段程序合理地分解成模块。论文比较了两种方案。

方案一： 将程序分解成五个模块

> 模块1（输入）：将列表的所有行读入，存放在内存里的数据结构中，以便所有模块的程序进行处理。

> 模块2（滚动）：对内存中的每一行单词实施滚动，生成所有的位移。为了节约对内存的消耗，本模块实际上会构建一数组，只将每一行的每一个单词的地址记录在数组中。这样，数组中每一个项就代表了由这个单词起始的那一个“位移”。

> 模块3（排序）：将模块2产生的数组作为输入，并生成同样大小和格式的另一个数组。只不过后一个数组是按照“位移”的字母顺序排列的。

> 模块4（输出）：依照模块3生成的数组，去模块1的数据结构中读取相应的“位移”，并将其依次输出到终端。

> 模块5（控制台）：负责启动和控制由前四个模块构成的工作流。

方案二： 将程序分解成六个模块

> 模块1（存储器）：用于管理表格中的数据。提供如下的访问接口：函数CHAR(r,w,c)用于读取表格中第r行的第w个单词的第c个字母；函数SETCHAR(r,w,c,d)用于将第r行的第w个单词的第c个字母设置成变量d的取值；函数WORDS(r)将返回第r行的单词个数；同理，可以有其他函数用于读取每个单词的字母数或者整个表格的行数，等等。

> 模块2（输入）：读取列表，调用模块1的相应函数，将列表存放到存储器中。

> 模块3（滚动）：负责实施滚动操作，主要提供两个访问接口：函数CSCHAR(r,l,w,c)用于读取第r行的第l个“位移”的第w个单词的第c个字母；在首次调用CSCHAR之前，需要调用一次CSSETUP函数，以完成滚动所需的预备工作。实际功能都通过调用模块1的接口实现。

> 模块4（排序）：也提供两个访问接口：函数ALPH用于实现按字母排序，需在其他函数之前调用；函数ITH(i)将返回排序后的第i个“位移”。实际功能都通过调用模块1和模块3的接口实现。

> 模块5（输出）：调用模块4，按字母顺序依次读取“位移”，并输出到终端。

> 模块6（控制台）：负责启动和控制前五个模块的工作。

以上两个方案孰优孰劣可能并非一目了然。第一种方案按照工作步骤划分模块，很自然，也是大部分人首先想到的方式。第二种方案显得比较复杂，但接口定义得很清晰。稍作分析，结论实际上是显而易见的。模块化的目的一方面是为了分工合作，另一方面是让软件易于维护。两者其实都基于同一个想法，就是让模块之间的关联（专业的讲法叫耦合度）越小越好。耦合度越小，就更有利于各个模块的独立开发，分工合作就更容易。同样，关联越小，一个模块的改动对其他模块的影响就越小，软件就更容易维护。在David L. Parnas看来，模块化归根到底是要实现“信息隐藏”，也就是说，用最简单的接口将模块的功能提供给使用者，而将模块的实现细节尽可能藏在模块内部。这样做的目的就是降低模块间的耦合度。

方案二的设计显然遵循了“信息隐藏”的原则，虽然它看上去显得并不十分自然。反观方案一，模块之间的耦合就复杂多了。首先，用于存放表格的数据结构成为了联系各个模块的纽带 - 几乎所有的模块都需要访问这个数据结构以实现自己的功能。其次，模块2和模块3之间以及模块3和模块4之间都需要传递大型数组，这与方案二的简单函数接口形成了鲜明对比。假设现实应用的表格太大，不适合全部存放在内存中。对于方案二而言，只需要对模块1进行调整，其他模块可以不变。而对于方案一，用于存放表格的数据结构将宣告失效，以至于所有模块都需要调整。又假设需求发生了变化，用户只需要输出排在最前面的那部分“位移”。这个变化带来了性能

优化的空间，比如大多数时候只需考虑那些以'a'开头的单词所在的行。为了实现优化，方案一的滚动和排序模块需要推倒重来，而方案二则只需要对模块4进行改动。基于“信息隐藏”原则实现的功能划分优势明显。

系统软件的设计也必须遵循“信息隐藏”原则。换而言之，一个系统应该用简单清楚的接口将自己的功能暴露给用户，而将功能的实现方式隐藏在内部。对用户而言，只需要通过接口将自己的需求或者想要达到的效果表达出来，然后将工作全权交给系统。系统负责准确无误并且不出意外地完成用户的任务。至于怎么完成任务，这完全是系统自己的事，用户无需关心。系统内部的工作方式可以不断改进，但只要功能接口不变，用户的程序就不受影响。这个想法看似简单，实现它却不容易。要获得简洁的接口，首先要界定系统的功能范围。如果界定得不好，要么接口难以简化，要么功能缺失。在数据管理系统的发展过程中，人们就在这方面遇到了相当大的挑战，并且至今争论不休。在后面的内容中，我们会介绍不同数据管理系统的设计选择，及其相应的优势和弊端。总之，有一点是毋庸置疑的，一个好的系统一定是一个好的模块。

折衷

系统在设计中还面临另一个棘手的问题，就是取舍。

大数据的热潮带动了分布式系统的发展。对分布式系统有所了解的人大都听说过所谓的CAP理论。在CAP理论中，C代表数据一致性（Consistency），A代表可用性（Availability），P代表分区容忍能力（Partition Tolerance）。三者的具体含义这里暂且不表。CAP理论断言：对于一个分布式系统而言，C、A、P三种性质不可能同时获得，只能选择其中的两种。于是，一个分布式系统的设计者就不得不做权衡和取舍 - 到底放弃哪一种性质呢。类似CAP理论这样的困境其实不是什么新鲜事（not big news）。对于系统设计师而言，取舍就是家常便饭，是工程实践中随时都会碰到。经验告诉他们，完美的系统只存在于想象中，系统设计的目标只能限于找到一个完美的折衷。

作为人之常情，我们希望一个系统是完美的 - 它既有强大的功能，又有极致的性能，还非常易于使用。只在功能、性能和易用性三种粗略性质之间，我们就已经碰到了难以调和的矛盾。如果一个系统功能强大，它的构造就会相对复杂，而复杂的构造势必带来性能损失。同样，一个功能强大的系统需要提供足够丰富的接口，让用户能够使用这些功能。接口的增加又会降低系统的易用性。类似这样的矛盾很多。在后面具体系统的介绍中，我们还会碰到不少。这些矛盾表明完美的系统是很难实现的，设计师必须取舍，找到一个合理的折衷点。同时，这些矛盾让系统设计成为了一件很有意思的事。它们考验人的智慧。好的系统宛如一件艺术品，映射出设计师深邃的洞察力和巧妙的构思。

因为折衷，现实世界才出现了大量各式各样的数据管理系统。这些系统因为选择了不同的折衷点而表现出不同的优势和弊端，也因而获得了各自的生存空间。然而，系统的多样性对软件开发而言并非好事。一方面，一个软件工程师不可能有精力去学习和掌握太多种类的系统。另一方面，一个软件生态也难以容纳过多的系统。站在软件开发者的角度，对同一类工作，大家最好只使用一个通用系统，比如，所有人最好只使用MySQL数据库管理数据，这样会大大简化软件工程的开销。这种“one size fits all”的想法很实用，也成为不少系统开发者的目标。但是，折衷给“one size fits all”的理想带来了巨大的麻烦 - 既然折衷是必须的，那么任何一个系统都有自己的弱点，因此无法做到真正的通用。在数据管理系统的发展历程中，我们看到“one size fits all”和“one size fits a bunch”始终在抗衡，此消彼长。

如今，曾经被认为是“one size fits all”的关系数据库正面临各种NoSQL和NewSQl数据库的挑战。数据管理系统的多样化成为了当前的趋势。这预示着未来的软件开发人员需具备使用不同系统的能力。正因如此，本书选择将多种数据管理系统作为讲解对象，避免造成大家的思维固化，误将某一类系统的折衷选择作为放之四海而皆准的准则。

>>下一页

参考资料

[1] D. L. Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM, v.15 n.12, p.1053-1058, Dec. 1972.