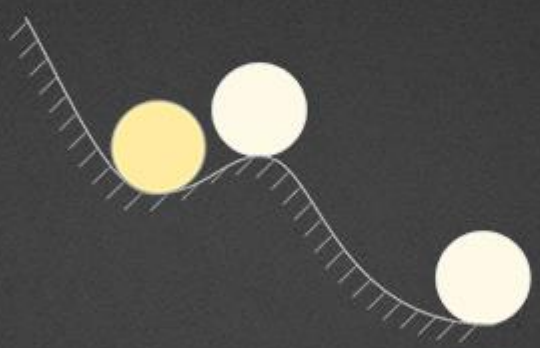
图灵电子书

程序原本



追溯程序原本之书，
走向设计师、架构师的入门之径

周爱民 著

内容提要

在本书中，作者采用反复置疑、层层递进的手法，从一些极其简单的现象或结果出发，为你揭示“程序是什么”这个简单而深刻的命题。本书将创建一个一致性的思维框架，并引领读者在这个框架之下，发现简单代码到系统结构之间不变的那些规则与本质。

本书既是追溯程序原本之书，也是走向设计师、架构师的入门之径。

序 1：程序里的世界

程序，或多或少地反映了这个世界运行的本质。那么，程序的本质又是什么呢？

爱民写了这本《程序原本》，看起来想部分地回答“程序的本质是什么”这个大问题。

在篇一，“计算系统”。他认为，“算”是程序之表，“数”是程序之本。然后，算的过程体现了数之间的“逻辑”。而算与数在人们的认知过程——当然也包括在计算系统——中的体现，则是对“事物”的“抽象”。当然，这个对事物的抽象，是与事物的“具象”相联系的，也只有这样，计算系统才是在反映现实世界的运行。

人们在“计算系统”中，通过“程序”来描述、虚拟现实世界，跟人们在日常生活中，通过讲述、写作来描述与推理现实世界一样，是通过“语言”来达到的。“语言及其面临的系统”，即是本书篇二的内容。

在计算系统中为计算机所理解的程序语言，也有一系列语法、语义与语用的设定，以尽量避免“计算不确定性”带来的副作用。不同计算范式的程序语言或者设计，均力图从不同的角度来减少这种副

作用。比如命令式编程、函数式编程，前者，通过限定计算步骤、命令的严格顺序来避免不确定性；后者，则通过引用透明来避免不确定性。

而系统的构建，即使部件由不同的程序语言写成，只要在交互界面上设计为“可实现的规则集”，以及输出表达为“可计算的数据项”，也可以实现“对话”。或者说，“可实现的规则集”和“可计算的数据项”，实际上也可以归为一套简洁的“语言”。

篇三，“程序设计的核心思想”。作者开始涉及到程序设计中的一些基本要素，包括数据结构、执行体与它在执行过程中的环境、语法树和对象系统等等。这些要素，仍然是从算与数出发，自然延伸出来的。其中“语法树”在本篇提出，我的理解是，“语法树”在很多情况下是可以作为程序设计中的要素加以利用的。

篇四，“应用开发基础”。从本篇起，对“程序”的理解开始上升到“应用”层次，主要是从“软件复用”和“工程化”的角度来陈述“模块/单元”与“项目/工程”这些“应用程序”的组成或者构建过程。并以“模型是一种沟通工具，这是它内在的‘语言’本质特征”来回到“程序语言”这个“程序”最基本的表达方式本身。

篇五，“系统的基础部件”。则在系统的层次来讨论“程序”，认为“系统是‘通过在程序组织上的结构化来解决规模问题’的一种策略，那么程序所解决的问题集‘能否分割’以及‘如何正确地分割’，就是所有系统问题的核心所在”。并提出，“系统应付规模问题的总法则只有两个：运算能力的分布,以及运算对象的分布”。

进一步的，既有了“分布”，则立刻遇到分布的部件之间的“依赖”。而这种依赖，处理的对象则是系统的（分布的）“状态”，

并可以通过“消息”来处理这些“状态之间的依赖”。状态和消息，是处理系统分布与依赖问题的主要数据模式。

最后一篇，篇六，“系统的基本组织方法与原理”。探讨“系统”这个带有“规模化”内涵的程序。而“这一规模的定义本身就是由跨领域引申而来”，以应对“跨领域”的系统级“程序”。另一方面，“服务”则更倾向于设计为带有领域逻辑交互界面的“节点”。那么在这种倾向下，服务要处理的对象，也许应该退化到“数据”本身，并作为“数据”节点来提供它的能力。尤其是面对所谓的“海量”数据时，不同领域的交互，需要数据以更灵活（可变）的方式来存贮、获得和传递。

而“程序”，这个即使在“系统”层面上也仍然是——“算”是程序之表，“数”是程序之本——的“玩意”，在规模化到了系统层面之后，其层间的规划与层间关系的模型，以及如何通过系统化方法来实现这些层之间，亦即是领域间的协作开发，就需要在作者《我的架构思想:基本模型、理论与原则》一书中继续展开了。

邓草原

2017.05.27

序 2：最后一层表象

我曾经写过一本《动态函数式语言精髓》[①]，并称之为我在最终结论面前的“最后一层表象”。而我所追寻的最终结论是关于什么的呢？那个问题从十余年前便困扰着我：作为一个程序员，我到底在做什么呢？换而言之，我一直在追问的问题其实就是：

> 什么是程序？

我曾经认为这个问题是有答案的，比如经典的“算法 + 数据结构 = 程序”[②]。但是在多年之后我对这个答案有了怀疑。因为我看到，《结构程序设计》[③]这本书根本上是在讨论另一个解，即“结构(算法(数据))”。然而即使如此，“结构化”本身，就是程序规模化的唯一求解吗？更深入地说，规模化是“计算的程序化”唯一的问题吗？

---

① 这是《JavaScript 语言精髓与编程实践》一书的电子版（精编版）的书名，事实上这也是前者在正式发行前所使用的书名。

② 《算法+数据结构=程序》，作者尼古拉斯·沃斯（Niklaus Wirth）是 Pascal 语言之父，也是 1984 年图灵奖得主。

③ 《结构程序设计（Structured Programming）》，由 E.W.戴克斯特拉、C.A.R.霍尔和 O.-J.达尔合著，他们分别是 1972 年、1980 年和 2001 年图灵奖得主。

我认为不是。

本书从数据之于计算系统的意义谈起，回顾了我多年前所认识到的那一层（我曾自认为是最后一层）的表象——亦即是“语言”。而后，将所谓真正的问题推及对“程序”的认识和理解，再推及到所谓的“应用”与“系统”，并最终揭示这些表象之下的那个简单的事实：它们不过是“结构化”这一求解的引申。

又是一层表象。

周爱民

2017.02.25

关于本书

> 节选自《大道至易——实践者的思想》（第二版）。

从现在开始，原书（《大道至易》第一版）被分成了三本书，以电子书的形式由图灵出版公司发行。这些书采用了免费共享的授权（是的，你可以很自由地分发它），分别面向各自己的读者群体。它们不但在内容上独立清晰，而且在行文上各持风格，再也不受原书的影响——例如我恢复了所有章节、小节的标题，分别设计了它们的封面等等。

这三本书分别是：

- 《大道至易——实践者的思想》（第二版）

  面向软件工程相关角色。

- 《程序原本》

  本书。面向程序员、软件设计师。

- 《我的架构思想——基本模型、理论与原则》

  面向架构师。

致谢

感谢所有的读者、编者以及一直以来支持我的朋友们。

感谢邓草原先生为本书作序。

感谢图灵出版公司。

感谢我的小学数学老师。

感谢 Joy。En，……我最爱的妻。

目录

引言：简单的本源

接下来，我们将要提出一些现实中的简单问题。这些问题是如此的简单，例如为什么要用一对大括号“{ }”来将代码括起来？之所以说它简单，是因为你会看到每本书都这样忠实地写着，而且你的每一个老师、每一个技术同行，以及每一个代码范本都如此教导着你。而它又无比复杂，以至于到了一种名为 Python 的语言出现时，我们就再也想不明白：为什么代码不再包含在一对括号之中了——对于许多人来说，将这样的一对括号换成一组“begin .. end”都是莫大的挑战。

编程的世界是如此的奇妙，如同我曾说过的一样：会不会编程，甚至成了某些人的智力评测基准。然而也如同上面所谈，解释其中某些看起来习以为常的现象，既大有必须，亦必为挑战。

那么我们有没有办法，不使用那些艰涩的公式或者数理逻辑来证明这个世界的必然性，而仅仅只是去说明它呢？

我想，代码总是需要拿来被“阅读”的吧。先不管读者是谁，一个东西要被阅读，它至少要具有两个性质：能被叙述，以及能被记载。

这其实涉及了两个更深层次的问题，前者是要求有一个语言系统，后者则要求有一个存储系统[①]。语言系统的特性取决于对话的双方，而存储系统的特性则取决于存储的对象，以及存储的条件……

等等类似这些的、将在本编中出现的文字，在我看来就是我们对“代码为什么是这个样子”的本源性的思考。那些完全不了解，也不探知，甚至未能察觉“本源问题”的人，努力地清扫着地毯上的灰尘并一遍又一遍地检视着，点着头发出赞许的啧啧声，而无视于空气中漂浮的粉尘——即使它们在半个小时后又将掉落在地毯上。

所以一个不能思考事物本源的人，是不可能具有开创性的，他既不能解决问题本身，也不能发现解决问题的可能途径。

[①] 这其实是人类文明的根本，通常含义的文明是从有文字记录开始的。

篇一：计算系统

> 计算机本质上来说仍然是一种算具，其基本构造与历来的算具并没有什么不同。其核心仍在于对数和算的表达，即一种表示与存储数的方式，以及一种计算的方法。
>
> 计算本身对工具是不存有依赖的，例如心算。但计算的复杂度，使得我们将计算过程进行切分成为必须，这基本上可以视作函数这一抽象概念的由来。

第 1 章 数，以及对数据的性质的思考

1.1 算数与算术

小的时候，我的数学成绩不错，但每次考完试回家，父亲总是问我：“算数考得怎么样啊？”因此有一个问题让我很是苦恼：为什么我的课本是《数学》，而父亲问我的却是“算数”呢？

很多年之后，当我有足够的资料来追溯这个问题时，发现在我父亲的年代，他们所学的那门功课的确是叫《算数》，后来有些书变成了《算术》，另一些又变成了《数学》。尽管我理解了父亲用这个词所基于的教育背景，但我仍然感兴趣于上面这几个词的演变过程。

再后来，我终于了解到连我学的历史课本也出错了：我国已经证实的最早的数学著作并不是汉代的《九章算术》，而是早了两三个世纪的《算数书》。这两本书的名字都是可查证的，前者为历代记载，后者则写在出土的竹简上。于是我终于了解到一个事实：古人其实最早是将这门学问理解为“算数”的，再后来才退步了，理解为“算术”。

为什么说理解成“算术”就成了退步呢？因为将书名写成“算数”的人，还知道我们“算”的对象是数，而作“算术”者，便只当这是一门为算而算的“术”了。

很多很多年之后，我们开始学计算机。很多人花了许多年功夫，最后尽在“计算”上做足了花样，却忘了我们原本算的仍然不过是“数”。算这些数的那些算法，只是“术”而已。

“算”是程序之表，“数”是程序之本。

1.2 信息是有意义的数据

计算机系统的数与算，是基于数学与电子学而发展起来的。

首先，这里的数（number）并不复杂，我们也都知道是所谓的“ 0 ”与“ 1 ”。在这个“数”上的算法也很简单，是所谓的布尔运算。它们作为二值数的提出与基本运算的确立，基本可以视为是布尔在1847~1848 年间的主要学术贡献。90 年之后（1938 年），香农在提交给麻省理工学院的硕士论文中，展示了布尔逻辑在电子学中的应用。又过了 10 年，香农首次提出使用 bit 这个词来表示二进制数字。[①]

bit 这个词被创造出来，用于表示这样一个二进制数所能代表的信息量。“由于 1bit 表示的是可能存在的最小信息量，那么复杂一些的信息就可以用多位二进制数来表达”[②]。

信息，是有意义的数据。[③]

1.3 我们要操作的数据是什么？

数据（data），并不是数，而是数的系列。数学概念上的布尔数，

---

① 这里涉及 4 篇重要的学术论文，包括乔治•布尔的《逻辑的数学分析——关于演绎推理的一篇随笔》(1847 年)与《逻辑的演算》(1848 年)，以及克劳德•艾尔伍德•香农的《继电器和开关电路的符号分析》(1938 年)和《通信的数学原理》(1948 年)。

② 引自《编码：隐匿在计算机软硬件背后的语言》, Charles Petzold 著。关于二进制数、布尔逻辑以及与此相关的电子学知识，可以参阅该书第 9~14 章。

③ 数据是客观对象的表示；而信息则是数据内涵的意义，是数据的内容和解释。这里选择性地使用了这一释义，其原因在于“数据是表示”适宜作为对程序设计（特别是其中有关数据的表示法）的后续讨论的基本背景。

是纯粹抽象的，可以是黑白点，也可以是正反面；电子学概念中的布尔数，是逻辑门电路中的高低电平信号。而数据，其内聚的特性表明它由一系列存有相互关系的数构成，其外延的特性表明它可以与其他数的系列建立新的抽象关系。

这里的“系列”，只是对数据的构成元素（例如多个二进制位）之间存在着的关系的一个简单称谓。这种关系是什么呢？例如，我们说“A”字符是一个数据，即是说它是由“1000001”这七个二进制数形成的组合关系。这个组合关系是强约束的，如果换一个组合，则不是“A”字符而是别的字符了。再例如，我们既然说“A”是“1000001”的组合关系，则同时也是说“B”必然是“A”的组合关系的一个外延。但这样的外延存在多种可能性，对于这两个字符以及由此构成的**数的系列**这一概念来说，其数学描述方式为：

> B = f(A)

我们知道，在“ASCII 字符集”这个系统中，*f()*所表示的外延关系是 `inc()`[①]，即 `B = A+1`。而在另外的某个系统（例如密码卡）中，*f()*则可能是另外的一个函数。数，与数据之间存有的不确定的函数映射关系，是我在前文中用“数的系列”，而非“数的序列”的真实原因。

我们所面临的硬件系统以及其背后的逻辑基础，都是基于数的概念；而我们的软件系统，例如我们使用某种开发工具、语言，或者操作某个设备的指令集，都是基于数据的概念。因此在软件开发（而非计算机研制）中，第一个要明确的问题是：

> 我们要操作的数据是什么？

[①] 在 Pascal 语言中的一个常用函数，意为 increase()，递增 1。

1.4 可计算数据只有两类，分别面向交互型式和交互行为

总的来说，我们的计算机由五部分构成，即控制器、运算器、存储器、输入设备和输出设备。然而这五个部分所理解的数据并不一样，以个人电脑系统为例，见表 1。

表 1 PC 中的设备与其理解的数据

类　　型	设施或部件	交互手段	数　　据
设备与设备	CPU、GPU、内存(存储器)	总线、端口、数据线	未知数据序列
设备与控制程序	控制器 ↔ CPU(运算器)、程序代码 ↔ 计算机系统	机器代码、汇编代码、中间汇编代码以及脚本等	指令、数值和状态
设备与人	输入输出设备	人机交互行为或反馈	扫描码（Scan Code）、RGB 值、字体等

我们还可以举出更多的计算机组成部件以及它们对数据的不同理解，例如显卡、打印机、硬盘，以及扫描仪等。但归总起来，这些“数据”只有两类：

- 一是在设备与设备之间，我们确立了以传输总线带宽为标准的数据**交互型式(type)**。即，设备与设备之间的数据，明确以如下单位传输：位（bit）、字节（byte）、字（word）、双字（dword）等。
- 二是在控制与被控制对象之间，我们明确了以**交互行为（action）**为视角

的三类数据[①]，即逻辑（logic）、指令（instruction/operator）和操作数（operand）。

正因为**交互行为**本身亦是（可以按照确定**交互型式**来定义的）数据的一种，可以在设备与设备之间传输，所以我们的击键行为最终可以表达为某些 CPU 指令。这也正是“人机交互”的基础：装置，以人可以理解和传递的形式，通过一定介质和中间转换，表达为确定的**设备之间的交换数据**。例如鼠标，人理解和传递的形式是移动与单击，鼠标作为中间的转换装备，最后将位、字节等数据通过设备之间的传输，送至计算系统的其他设施，例如端口。

从这个角度上来说，所谓的人机设备本质上仅仅是一个“数据采集”和“数据反馈”的终端，采集的是人的行为，反馈的则是人的知觉系统所能理解的信息（物理的、生理的、化学的、机械的、光学的等）。包括采集与反馈，以及该装备的内部构造与行为等，也是可以通过计算系统来控制的，那么它又将变成前面所述的、程序员所关注的两种“数据”。

1.5 数据性质的基本子集：标识、值和确定性

我们在程序设计中表达一个数据的时候，总是在描述数据的某些侧面，而不是它的全部。其关键在于，我们事实上不会面对这一数据的全部方面。进而推论：我们既无法、也不必完整地表达现实系统（的种种“数据”）。例如我们说硬币的正反面，可以表达为 0、1，但这一项叙述只表明了 0、1 与硬币两面的映射信息，并没有实际表

① 数据的交互是可以表达为数据的，即在交互双方本质上仍然是通过数据来传递行为指示——“控制与被控制”是两个设备之间的行为关系，“数据”包含了行使行为所需的全部信息，而“（某种）行为”是设备自身的能力。

明：

- 硬币当前是正面亦或反面。
- 旋转中的硬币是正反未确定的。
- 硬币正面明确的称谓是 front？或是“正面”？或是 yes？又或“反面如何称谓”？
- “硬币是正面的”在发生时，直到识别者得到这一结论的过程中，可有变化？

因此在我们约定了**表达数据的规则**——即数与数据之间的映射关系之后，还需要约定**表达数据的方面**。如前所述，我们不需要说明这个数据的全部性质，只需要探求一个最不可或缺的子集。综合目前在计算机语言及其抽象概念上的种种尝试，可确信包含于该子集中的有：

> 标识、值和确定性。

1.6 对数据与行为加以标识，是一个系统“可计算”的基本条件

首先，我们需要一个标识符系统来“**标识**”所有我们要操作的数据（例如值与引用），以及这些数据的操作方法（例如运算符）。现实是：任何一个不被显式地或隐式地标识的数据，都不可能参与运算过程；任何一个不被标识的行为，都不可能在系统中执行操作[①]。

继续思考这些标识对我们而言是相当有意义的。例如说，我们用 `aNum` 标识了一个数据，请问这个数据是指 0 呢，还是指 1？或者我们再设问，即使我们用 `aNum` 标识了数据 1，请问这个数据现在究竟是 0，还

[①] 关于这一点是如何确立的，是我们在“第 4 章 语言”中一切讨论的起始。

是 1 呢？

上面这两个问题看来是文字游戏，但确实是计算机语言和编程中最核心的一些设问。其一，它涉及一个标识是否有其存在价值的问题。亦即是说，数据（包括其操作方法，亦作为数据）是否明确地作为该标识所表明的——内容，亦即数据，亦或者更明确地表述为计算机术语的“值”的——意义而存在[①]。以第一个问题为例，它表明：

```
aNum
```

这样的声明仅只是标识了该数据，但不存有值；而当我们使用下面的代码来声明时：

```
aNum = 0
```

才表明了这个 `aNum` 是存有值的。

其二，它涉及计算环境如何认识上述值的问题。我们继续以上述问题为例，上面的声明并不表明这个值是否存有变化的可能。因此下面的代码声明：

```
var aNum = 0
```

表明 `aNum` 是可能变成 1（以及其他的任何数值）；而以代码：

```
const aNum = 0
```

来声明时，就表明 `aNum` 是确定的，不可能变更为 1 或其他值。

也就是说，程序中有所谓的变量或常量之分，正是计算环境设问数据**“确定与否”**的种种侧象。

[①] 这里说到“值”，是将数据标识的含义，与该含义的内容——一些具体的性质分开。例如说我们谈到“树”这个名词（或作英文的 tree 这个单词），则标识一个自然中存在的事物，但树的内容——例如高度、树龄等这些“值”并没有被叙述。而确定性，便是指“树”这个标识是否包含了这些“值”的一个确定描述。

1.7 数据的确定性总是以其生存周期为背景的

从计算机存储设备的角度来看，所谓的数据其实只是一个个单元格（Cells）[①]，例如内存中的某几个位置。如果我们的数据“永远确定”，则意味着这些单元格要永远存留并且不可变化。这当然不可能，因为不可能有永远的或者无尽的存储。因此任何在计算系统中的数据都存有**生存周期**的问题，我们讨论的确定与不确定——或者你可以（暂时地）看成常量与变量——是以其生存周期为背景的。一旦离开这个背景，则**标识**仍可能存在，但讨论数据**“确定与否”**便不存在意义。

生存周期是一个数据的性质，亦或是运算系统的性质在数据上的投影？这是一个哲学化的命题。但现在，我们可以仅仅将它当成不同视角带来的理解差异。以我们当前所需要的——我的意思是数据的——视角，我们可以认为那仅仅是计算系统的性质。例如一个进程的生存周期，即是该进程中数据的生存周期——对于数据来说，生存周期只是其背景的、自然的规律。

因为生存周期是有限的，所以“数据确定”是可能的。例如我们以一个确定的空间（我们通常称为 Buffer，或 Cache，或特定内存块）来存放数据，那么我们可以在数据填满这个空间之前保证任何数据的确定性。

1.8 不确定性是对机器计算是否有价值的终极拷问

但数据也可能是“不确定”的。分析我们上一项假设，它所包括的

[①] 这一概念(Cells)引自 Peter Van Roy 的著作 *Concepts, Techniques, and Models of Computer Programming*。在本书的附录中，对这一概念的提出以及它与语言范型之间的关系作出了进一步的释义。

约束有二，其一是“确定的空间”，其二是“填满该空间之前”。当这些约束不被满足——数据所必须的单元格(cells)数量不足以表达我们的运算对象，或单元格背景上的生存周期不足以维持数据的确定——的时候，数据将是不确定的。

现实中，二者往往是同时出现的，例如主机接收来自端口的鼠标位置数据时，我们既不能确保它必然在一个生存周期中得以处理，也不能保证有足够的空间来存放这些数据——从开机到关机。

因此数据即使被标识后是确定的，这种确定也必止于它从存储（这个背景）中消亡——例如数值 0 变成 1 ，则 0 即已消亡了，而之前它是确定的。从存储的角度来看，由于单元格(cells)有限，所以必然发生数据更替，进而导致上述消亡的出现。因此，数据的不确定性，首先是由存储背景的有限导致的，而并非来自于数据自身。

数据一旦存在“不确定”的可能，则计算系统的严密性就受到了挑战——计算的不确定性是对机器计算是否有价值的终极拷问。

第 2 章 逻辑

2.1 正确的“数”与正确的“算”，为什么不能得到正确的结果？

在计算系统中，数据基本上来说有两个方面的性质，一是指它的标识，二是指它的内容，亦即是值。出于计算环境的限制，在以“标识与值”这样的方式描述的数据上，也存有第三个方面的性质，即值可能是确定的，或不确定的。

既然确定性是数据与使用这些数据的计算系统的终极问题，那么**我们权且认为数据是确定的**。说“**权且认为**”，是因为我们晚一些会讨论到它“不确定”的一面，而且为了使之可以被整个计算系统接受，我们会试图将“不确定”作为“确定”的一种特例，由此使整个计算系统的行为确定，进而让我们的计算有意义。

> 当我们说“**数据是确定的**”，是指我们可以假定标识 `aNum` 从开始作为数据标识，一直到它失效，其内容都是一个确定值。

当我们确定“数”是什么的时候，才能确定地描述基于该数的“算”是什么。例如说我们确定了二进制数，因而确定了基于 0、1 的加减乘除等。正是数的抽象与算的规则这种紧密的绑定关系，决定了我们不能将这一“算”的规则应用于十进制数。但这并不是说，将**正确的数**与**正确的算**耦合在一起，就可以得到一个正确的计算系统。

在我很小很小的时候，邻居的大人总喜欢出一些数学题来考我。他们问：嗨，小子，你说说“三加二减五”等于多少啊，我回答：是零。然后他们又问，那“三加二乘以五”又等于多少啊。

我回答：是二十五。

然后大人们就开心地哈哈大笑。我在 3+2、5-5、5×5 几个运算中，都正确地得到了结果“数”，并正确地应用了基于十进制数的“算”的规则，但答案是错的。

2.2 正确逻辑：顺序、分支，与循环

现在我们都知道原因：在“三加二乘以五”中应该**先计算**“二乘以五”。对于 3+2×5 这个题目来说，单纯地做：

```
3+2=5
5×5=25
```

这样的计算是不行的，因为上面的解题中出现了“先”计算什么，与“后”计算什么的问题。

由此可见：无论是 3+2 还是 5×5 等，都是数值的计算；这些计算要正确地表述一个解题过程，还需要一个正确的逻辑描述，例如“先后”——即是指，我们要按某种**顺序逻辑**来应用“算”的规则。然而这样“正确的逻辑描述”有哪些呢？

这倒不需要我们再逐一列举，或像我一样回顾数学知识的点滴来源。Dijkstra（戴克斯特拉）[①]对这个问题有过非常严谨的数学论证，他指出：（我们）有三种思维方法用来理解一个程序，即枚举法、数学归纳法和抽象。除**顺序逻辑**之外，他指出枚举法、数学归纳法分

[①] 艾兹赫尔•戴克斯特拉（Edsger Wybe Dijkstra），荷兰计算机科学家，结构程序设计之父。1972 年图灵奖获得者。本书对 Dijkstra 的观点的引用，均出自《结构程序设计》的第一篇，即“结构程序札记”。于此，后文中不再复述。

别可以用程序中的**分支逻辑**和**循环逻辑**来表达[①] [②] [③]。例如，枚举法的基本思维是，对于一个条件集，

- 如果条件 n 不成立，则条件 n+1 可能成立；若条件 n+1 仍不成立，则条件 n+1+1 可能成立……
- 如此非此即彼，则当所有条件不成立时，条件集中没有可成立的条件；否则，
- 条件之一成立，则该集中有成立条件。

对于上述思维过程，就可以用**分支逻辑**（分支以及多重分支语句）来表达。

2.3 我们要记住：Dijkstra 说他只讨论了“顺序机器”上的正确的逻辑

我们已经触及到计算系统的构成，以及在这样的计算系统上正确计算的诸多要素。一些看起来很明显的要素包括：数、数据（标识、值）和正确的逻辑描述，另一些不太明显的要素包括：算、思维方法和确定性。

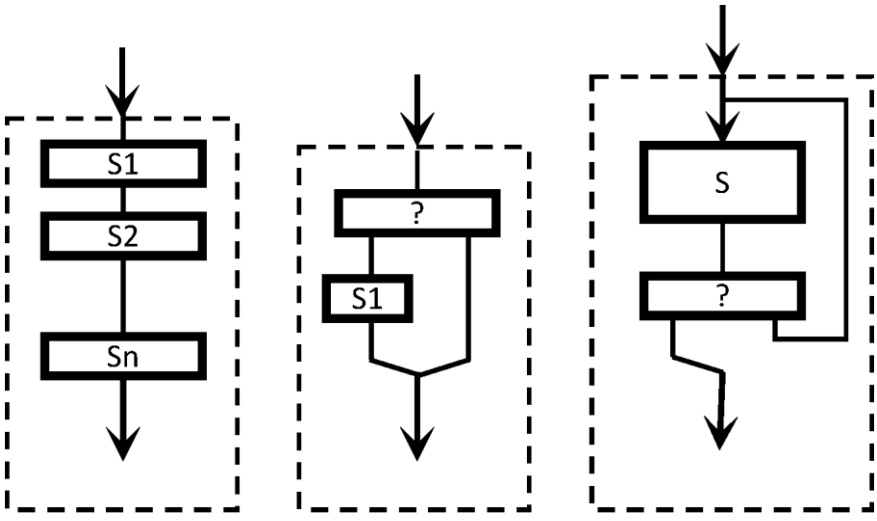
Dijkstra 接下来用顺序逻辑统一了分支与循环逻辑，使之成为“顺序机器”上的最基本的**正确的逻辑描述**。他说明，我们可以从形式上将上述三种逻辑表示为图 1 所示的图形。

[①] Dijkstra 是用数学方法来证明分支与循环（和递归）逻辑的正确性，我们同样可以把这个证明过程看成一个映射关系，即上述逻辑可以视为思维方法在计算系统中的映射。

[②] Dijkstra 提到了抽象，但没有对抽象在逻辑证明中的作用给出类似的数学证明——尽管他事实上在后面（不太明显地）给出了一个抽象逻辑证明的实例。

[③] Dijkstra 并没有用同样的方法来证明顺序逻辑，而顺序逻辑正是图灵机的本质设定。因此，在 Dijkstra 的论证中，他强调“只讨论关于‘顺序机器’的程序”。

图 1 三种逻辑表示



很自然地可以发现：分支逻辑与循环逻辑其实都只有一个入口和一个出口，因此它们也自然可以作为顺序逻辑中的 *S1...Sn*，而不会破坏顺序逻辑的基本规则。更进一步，既然分支逻辑与循环逻辑是可被证明为正确的，并可以作为“顺序机器”本质所设定的顺序逻辑的一部分，那么由这些逻辑构成的“程序”也就必然是正确的。

这里的所谓正确，包括三个意思：一是程序能**正确地描述**人的思维；二是程序可以由机器**正确地执行**；三是机器执行的结果**正确地符合**人的思维的预期。不过这所有的“正确”仍然依赖两项前提：一是计算系统是一个“顺序机器”，二是在每一个用于顺序计算的阶段（S1...Sn）中的数据，是确定的。

最后这一点——数据的确定性，正是顺序逻辑的必然结果：对于一个确定的逻辑而言，一个确定的输入，必有一个确定的输出。所谓输入与输出，若是数据，则在“第 1 章 数，以及对数据的性质的思考”中所述数据的内聚与外延的性质保障了这一结果；若是逻辑，则如上的形式化证明便保障了这一结果。

第 3 章 抽象

3.1 “程序设计的精华”

抽象是人们理解已知与未知事物的基本能力。例如你给旁边的同事甲介绍说：这段程序是张三写的啊。这时甲知道了“张三”，但并不知道张三的年龄身高、衣着打扮，所以这“张三”便是一个抽象。如果此时你把李四拉来旁边，说：不过这个人也出了些主意。这时，甲看到了活生生的，有年龄身高、衣着打扮的一个具象的人，却不知道这个人是李四。

抽象与具象是我们对事物的全部认识。只有当你指着那个人说“这是李四”的时候，同事甲才能把一个具象与抽象联系起来。所以事实是我们作为具象存在，而又用抽象来表明自己存在。这既构成了我们的人类世界，也同样构成了我们的计算世界。而这样的关系，在程序中不过是一行代码：

```
var aNum = 0
```

在此前的讨论中，我们说`aNum`是一个标识，上述代码声明了`aNum`的两项性质：一，它是变量；二，它指代数据 0。从抽象的标识`aNum`，到它作为具象的上述两项性质，我们事实上已经看到了（并非物理上的）计算系统的绝大部分构成。无怪乎 Dijkstra 说：“人们一旦了解在程序设计中如何使用变量，他就掌握了程序设计的精华。”

3.2 理解了“=”号，就理解了“命令式”计算范式

但是在这一行代码中，除了表达上述两项性质所必须的内容之外，还有一个“等号”（=）。在不同的语言中，这个等号可能存在两种

含义[①]：其一，它是仅仅叙述 0 与 `aNum` 之间的指代关系；其二，它表示将 0 这个值存放到 `aNum` 这个标识所指示的存储(cells)。

一些语言中的等号同时包含上述两个含义，另一些却只有其中一个含义。而我们知道，一个标识——例如符号“=”仅仅是一个抽象。那么上述观点是说：语言在等号上的抽象含义并不相同。

拿这三种可能含义之一——“表示将 0 这个值存放到 `aNum` 这个标识所指示的存储(cells)”来讲，这是一个具体的行为，它表明计算机器的各个部件间配合完成的一个操作，即**运算单元**将一个数 0 置入**存储单元**。从人的角度来看，这类似于我们向机器（包括运算单元与存储单元）发送了一个命令“`aNum = 0`”，于是机器就执行了这一命令。

这就是“命令式”计算范式的基本抽象。

3.3 “函数式”计算范式的核心，正好是消除这个“=”号

然而我们知道，“将数置于存储单元”这样的事情并不是计算所必须的——它只是“计算结果的表示法”。因此我们将这个部分从代码：

```
var aNum = 0
```

中抽去。则运算部分可以表示为：

```
= 0
```

这个“运算部分”——作为整体来理解——的具体内容有两种可能：

[①] 事实是可能存在无限种的含义。为了简化我们的讨论，我们这里只讨论常见的两种，即 `aNum` 与 0 之间存在赋值关系的情况。至于将“=”用作等值比较运算等类似情况，我们暂不讨论。

对于计算来说，表明“数 0 ”的方法，就是一个 0 ，或将之表示为一个运算 *f()*的结果。设 *f* 用 `equ0` 来标识——既然“运算部分”整体可以理解为一个标识，那么我们也可以换一个标识来表示这个抽象：

```
function equ0() {
  return 0
}
```

既然上述 `equ0` 是一个计算过程，那我们可能通过有限的步骤来完成这一计算[①]。亦即是说，`equ0` 仅仅关注该标识与其含义的确切关系，其具体指代为 `return 0`，或者是某个耗时有限的计算过程，但并不是这个抽象本身所关注的。

现在，我们提出最后一个推论。既然 `equ0` 可以指代任意有限的计算步骤，那么它必等价于图灵机的“顺序逻辑”中所有步骤；既然 `equ0` 可以指代图灵机所有的顺序步骤，则必然能指代顺序步骤的两种特例：分支与循环；既然 `equ0` 既可以指代计算的要素“数”，又可以指代计算的要素“算”，还可以指代描述正确计算所必须的逻辑，那么 `equ0` 本身——在概念抽象上——必然等同于一个完整的计算系统。

这就是“函数式”计算范式的基本抽象。

3.4 本质上相同的抽象系统，其解集的抽象本质上也是相同的

关于函数，这里再做最后一点点补充。Dijkstra 说“在命名一个运算和使用一个运算之间也存在着一种抽象”，这里的“命名一个运

[①] 这里是无限可能的计算过程，但就顺序机器计算的必要性来说，适用的过程都必然是有限步骤的。

算”即是函数的本意。

基于此，Dijkstra 提出在使用中只注意“（函数）做什么”而不必问“它如何做”。他强调这一过程与使用定理而不必问定理如何证明是一样的。他用这种“偷懒”的法子来证明：使用函数与使用定理一样可行，因而由函数构建起来的计算系统也有着如同用公理、定理构建起来的数学系统一样的正确性。

这一证明的关键假设是：本质上相同的抽象系统，其解集的抽象本质上也是相同的。而我们人类在自然科学领域中的全部知识，皆来自于这一假设的正确性。

篇二：语言及其面临的系统

> 一个和尚挑水喝，是函数的功能问题；两个和尚抬水喝，是运算能力的分布问题；三个和尚没水喝，就是系统问题了。

第 4 章 语言

4.1 语言不能由外在形式来定义，是形式无关的

就程序设计语言来说，它涉及两个事物，其一是程序设计，这个词多少有些美化的成分在里面，因为它原本的意思仅仅是编程（programming），而后来才被演译成了程序设计（program design）；其二是语言（language）。

什么是语言呢？解释这个问题跟说清“什么是我”一样地困难，因为如同“我在解释我”一样，我们也正是在“用语言来解释语言”。语言影响了我们全部的生活，是我们全部的知识：我们在口头上讲的，在书本上写的，以及在头脑中思维的，无一不是语言。但是大多数时候，我们只是“被学会了”一门语言，而并没有去认识它是什么。

我们口头上讲的、书面上写的以及头脑中形成映像的三个东西被统称为“语言”，那么显然，上述三个事物仅仅是“语言”在不同载体上的表现而已——用我们此前的一贯措辞，就是三者都不过是语言的不同侧面。因而真正决定了语言之为语言的，决不是书面上的字符，或口头的发音，或头脑中的那个意象。就某一门语言来说（例如汉语），其一，如果书面上的字符决定了它（是这一门语言），那么它无法包容该语言的、古今文字的差异；其二，如果口头上的发音决定了它，那么它无法包容地方口音；其三，如果头脑中的意象决定了它，那么它无法包容任何还不存在的事物。

所以我们讨论的语言，是不能用其外在形式来定义的。通常我们对语言的定义，仅仅是说明它的功能，即语言是一种事物与事物之间

沟通的工具[①]。由这个定义方式来看，程序设计语言，是指计算机与人（亦即是程序的使用者与定义者）之间沟通的工具。

4.2 什么叫“会编程”？

程序设计语言——这种工具有什么性质呢？或，究竟是什么决定了一种语言称为 Java，而另一种叫做 C#呢？它们之间存有何种不同，又存有哪些渊源呢？有趣的是，通过分析现有的种种程序设计语言，（正因为这些语言是我们人类自己创造的，所以）我们发现如同人类的自然语言一样，程序设计语言也总是有着三种基本性质：**语法**、**语义**与**语用**[②]。正是这三种性质，使得它区别与其他语言，而又能从其他语言那里有所借鉴以及沟通。

语法是指我们表达内容的形式。这一形式首先与不同的表达手段有关，例如同一个意思，我们的口头表达和书面表达是不同的。其次，即使表达手段相同，也会因为介质的材料性质存有差异而导致形式不同，例如钟鼎文和白话文都用于书写，但显然钟鼎文不能像白话文那样冗长。类似地，在我们的程序设计语言中，早期的程序输入就是电子开关的开合，因此代码会是一些操作命令，而现在我们可以将之输入为接近自然语言的程序文本；早期的运行环境限制要求代码必须尽量精少，而现在我们则考虑通过规整而冗长的代码来降低工程整体的复杂性。所以，语法是对语言的**表达手段**，以及对该表达手段的**条件限制**加以综合考虑而设定的一种**形式化的规则**。

---

① 又例如，古文中对语言的定义是自言为言，与别人说则为语，进而有食不语、寝不言之说。

② 莫里斯(C.W.Morris)在他于 1938 年出版的《符号理论基础》一书中，最早将语法学(syntaotics 或 syntax)、语义学(semantics)、语用学(pragmatics)明确地作为符号学的三个分支。从而构成了现代语言学研究的三个主要平面。

语义是指我们表达内容的逻辑含义。语义有两项基本性质：一，必须是一个含义[①]；二，该含义必须能够以某种基本的逻辑形式予以阐述。语义还有一项非必须的性质，即：三，上述的逻辑所表达的含义可以为语言的接受者所知。

略为讨论一下第二项性质。为何语义必须可以阐述为一种基本逻辑呢？因为语义定义为内容的含义，而这种含义可以由多种形式来表现，因此如果它不能用一种基本逻辑来表达，也就没有办法在多种表现形式之间对它互作验证。例如不能用书写的方式来确定口头转述的正确性，或反之也不能通过口传心授来传播书本知识。自然语言中的这种性质（部分地）可以由基本逻辑的矛盾律来约束，即“一概念不能既是该概念，而又非该概念”。正是我们的文字记录与对话交流等内容中存在着这样的一些基本逻辑，所以它才可能科学、严谨以及正确。

第三项性质对自然语言来说是非必须的——如果一个人自言自语，那么他的言语可能仍然是有语义的，只是这语义不为他人所知。但这一点对于程序设计语言来说却是必须的，因为我们设计这样一门语言的目的，正是要让我们所要表达的含义为计算机所知。而正是这第三项性质，加入了对“语言的接受者的理解能力”的限制。出于语义的前两项基本性质，这种**理解能力**也必然由两个方面构成，一是指**含义**，二是指**逻辑**。

我们回到了上一章所讨论的内容：计算系统的要素，包括数、数据和逻辑，以及在此基础上**进行正确计算的方法**的抽象，即计算范式。只有通过这些组织起来的语义，才可能被（我们此前所述的）计算

[①] 概念或实体。即，在表达者的意识中需要表达的对象。

系统理解。这些语义与其表现形式（即语法）是无关的，有其基本逻辑存在。

我们所谓的“会编程”是指对这种语义的理解，而非对某种语法的熟悉。正因如此，我们才可以在 Java 上实现某个程序，又在 C#上同样实现它，在（使用这些语言的）类似的仿制过程中[①]，

> 不变的是语义。

4.3 语用限定：试图避免“计算的不确定性”带来的副作用

我们来稍稍讨论一下语用的问题。

从对自然语言的观察来讲，同一句话——即语法和语义都严格相同——在不同的场合（语境）中出现，却可能有微妙的甚至是迥异的差别。但在讨论“差别”这个问题时，我们要先将语法从中别开，例如：

> “这难道不是吗”与“难道，这不是吗”

在后一种表达中用语法带来的强调效果；也要将语义从中别开，例如我们既使用

```
1 + 1
```

来表达算术，也用它来引申为人与人的合作，这两种语义都是确切且又不同的。最后，我们还需要将某些语言因其不严谨以及使用习惯所导致的歧义从中别开，例如：

[①] 这也意味着，语义上的表达能力决定了一门语言是否真正有别于其他语言。语义能力上等价的语言，除了开发人员的喜好或运行平台的限制之外，所谓有益的价值仅是开发库的丰富与社区的活跃等。而所有这些，都是与语言的本质无关的。

> 早起的鸟儿有虫吃

既可以理解为“有虫吃鸟”，也可以理解为“鸟能吃到虫”。

在这几种情况区别开之后[①]，语用讨论的是语言背景的因素下的差别。例如：

> “去死！”

用在战场中，表示愤怒、诅咒与呐喊；而在情人间即使连标点都不变，也可以表达亲昵。这种在语义的组织与逻辑上，以及在语法的构造与表述上都没有任何的不同，但因为场合而含义有别的情况，是语用的问题。

显然，如同我们此前所说的“计算的不确定性是对机器计算是否有价值的终极拷问”，我们并不希望在使用一种语言与计算机沟通的时候表达出上述的不确定的含义，或者反过来，计算机给出我们一个不确定的结果。因此事实上我们在设计计算机（软件与硬件）系统之初，就在尽力避免与之沟通时存在的语用问题。亦即是说，在严格的计算系统中，语用——这一语言的背景因素被限制在计算机的初始环境中，从而使“语义+语法”能够描述确定的计算及其结果**成为可能**。

但是在计算机的应用中，领域特定语言（DSL，Domain Specific Languages）其实是基于对语用学的研究与实践。所谓**领域特定**，即重设了“严格的计算系统”这样的背景。所以在这类语言中，我们可能看到与此前讨论的“计算系统的要素”不同的内容与逻辑。但是从语言的性质来看，它仍然是基于语法和语义，并且限定语用

[①] 这事实上也意味着计算机语言需要：语法明确，无情调修饰、无语义引申、无歧义。

（领域环境）的。

4.4 绑定

语法与语义是语言的两个基本性质，分别指代语言的两个方面：形式与内容。就经验来说，可以想见的：形式与内容不一致——亦即是所谓的“辞不达义”的情况，就必然会出现。在现实中，我们可以通过对同一事物反复地[①]、从不同侧面[②]和用不同方法[③]描述来解决“辞穷”的困境。而我们显然不可能在程序设计中这样做，因为计算机对事物的**理解形式**很单一，此其一。

其二则是我们不必这样做。因为此前我们讨论过，计算机的**理解能力**是有限的，只包括数、数据和逻辑以及在此基础上进行正确计算的抽象，所以我们只需要**约定**语法与这些计算机理解能力范围内的东西之间的唯一关系，那么计算机所理解的东西与我们描述的东西，就有了唯一的映射关系。不过换一个角度来看，我们也必须按照这样的约定来设计我们的语法，使之唯一对应一种计算机理解能力范围内的语义。

这在程序设计的术语中，就叫做**绑定**。

4.5 你真的理解这行代码吗？

绑定有很多种情况，几乎所有在程序代码中用到的标识，都存有绑

[①] 例如沟通中的“再重申一下我的主张”。

[②] 例如沟通中的“我们换个角度来看这个问题”。

[③] 例如沟通中的“给你画个草图如何”。

定的问题。最常见的例如变量[①]：

```
var aNum = 100;
```

上述一句代码（的语法），对应着三个语义：

（1）有一个标识，记为 `aNum`；

（2）标识所指代的数据，其值是可变的（即，变量）；

（3）该数据当前值为 100。

对于大多数语言来说，第二个语义是一项执行过程中的限制，亦即表明任何可以访问到 aNum 标识的代码都可以改变它的值。为了简便，我们暂不讨论这一个语义的绑定问题。但第一、三个语义所述的：

> 有标识 `aNum`，其数据当前值为 100

就明显作用于我们的计算：如果数据无值，或值不确定，则我们无法进行后续的计算过程。所以对于代码 `aNum = 100`，其在语义上的“值 100”将在何时被绑定到标识 `aNum`，就是一个相当重要的问题。

以一个代码片段来看（相信我，这是实际可运行的代码）：

```
1  // 代码 1
2  function process_part_one() {
3    aNum = aNum * 2;
4    var aNum = 100;
5  }
```

根据我们对这段代码的语法约定，第 2 、 5 行决定了代码片段中的

[①] 请容许我再次指出 Dijkstra 在《结构程序设计》中所说：“人们一旦了解在程序设计中如何使用变量，他就掌握了程序设计的精华。”

标识符的生存周期[①]。但在这个生存周期中，`aNum` 是何时有“值 100”的含义的呢？

对于这个问题，一些语言认为，表达“`aNum` 变量具有初值 100”这样的语义，应该是计算过程的**前设**，即对计算前提条件的声明。而声明并不是计算，因此声明语法与执行语法也应当分别对待。例如 Pascal，就处理为这样的语法（在 `var` 部分做声明，在 `begin...end` 部分处理执行）：

```
// 代码 2，pascal 风格
procedure process_part_one;
var
  aNum = 100;
begin
  aNum := aNum * 2;
end;
```

另一些语言则认为，声明语法可以理解为对执行环境的预置，也有执行含义，因此可以允许“即用即声明”，例如 C。但是语义上，这是因“（需）即用”而进行的声明，不可能出现“已用”而未声明的情况。所以即使 C 语言，也会因为语义上无法解释，而判断上述代码 1 违例。

还有一些语言认为：

```
var aNum
```

是一个语义，它只表明标识 `aNum` 的存在，直到**函数调用**时才绑定另外一个语义：初始操作，亦即是为函数内所有存在的标识进行初始化。在这样的认识下，“代码 1”中的

[①] 如你所知的，这是一个函数。大多数语言的函数，都约定了其（形式上的）函数体内的标识符的生存周期。

```
1  // 代码 3
2  function process_part_one() {
3     var aNum
4  }
```

就具有了完整的语义。这个语义（即“初始操作”）与实际的计算机行为，是在把 `process_part_one()` 作为**函数调用**时，才进行绑定和实施的。其效果是：

- 在生命周期（代码 3 的 2 ~ 4 行，或代码 1 的 2 ~ 6 行）中，记有一个标识 `aNum`；且，
- 在该标识所处生命周期开始时，总会有一个初始化动作，使该标识具有一个初值：**无值**；且，
- 设整个计算环境中，**无值**是一种值，记为 `undefined`。

你可能已经知道，这就是 JavaScript 的实现方法[①]。进一步地，除开上述代码在生命周期上的解释之外，代码 1 被理解为：

```
1  // 代码 4
2  function process_part_one() {
3     aNum = aNum * 2; // aNum 有初值，记为 undefined
4     aNum = 100;
5  }
```

我们看到，代码 1 中的语法：

```
var aNum = 100
```

所具有的两个语义（本小节开始处的语义 1 和语义 3）被分别绑定在**函数调用开始**和代码 4 的第 4 行；且一个标识的初值含义，总是明

[①] 在 JavaScript 的实现中，这里被实现为闭包。当一个闭包被创建时，它的上下文自然被初始化，而非（显式地）进行一个赋以初值的操作。你可以认为闭包是一个内存块，它创造的时候就是一块空的内存 (full by Nil)。

确地被约定为`undefined`。由此一来，上述代码 1 整体的语义就确定了。

4.6 将“计算机程序设计”教成语言课，是本末倒置的

我们所谓的“会编程”是指：将我们的意图表达为计算系统的**理解能力**范围内的语义。而这种语义：

- 由计算系统与程序员共同确知的**数据**与**逻辑**构成；且，
- 最终可以由某种**计算方法**在指定计算系统上实施以得到计算结果。

这里的计算方法并不是指“算法”，而是指对某种计算实施过程的抽象，例如在“第 3 章 抽象”中所讨论到“命令式”和“函数式”这两种计算范式。所以，

> 会编程与掌握某种语言的语法形式是无关的。

编程实质上是一种在语义描述上的能力修养。具备这种能力之后，语法也就无非是一些规则、限制和对不同计算系统的理解能力上的差别了。所以“计算机程序设计”这门功课应该教你编程，而不是教你使用一门具体的语言——我们现在大多把它当成语言课，实在是本末倒置了。

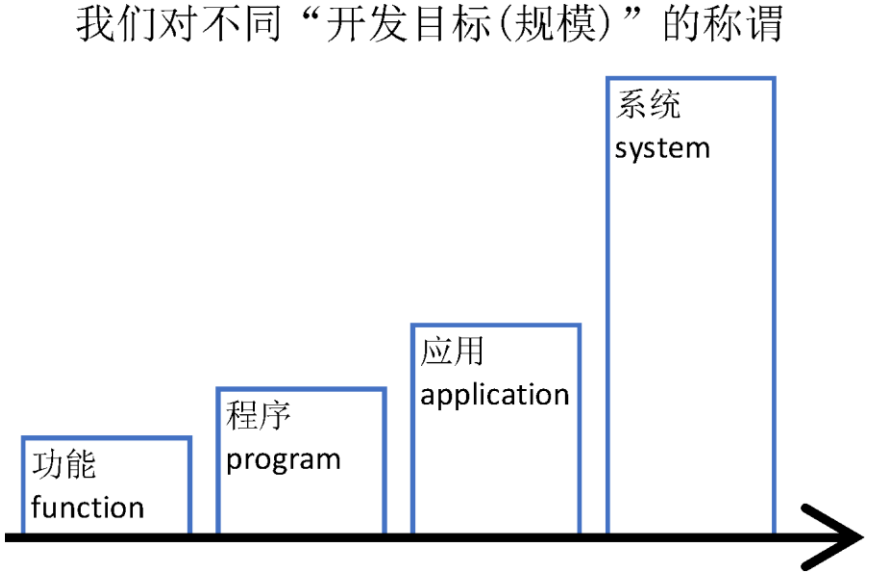
第 5 章 从功能到系统

5.1 软件开发的规模

我一直有一个问题：为什么 JavaScript 不适合大型开发？更进一步的问题是，JavaScript 可以应付何种规模的开发？或者，我们泛义一点来讨论这个问题：**语法与语义上的特性，是否决定了该语言可能适宜的开发规模？**

对于这一系列的问题，我希望尝试通过对第三个问题的分析，回溯至对第一个问题的探讨。首先我们需要界定“开发规模”，而这缺乏一个必要的、科学的分类方法。以我的经验来说，我用四个显著的关键词来区分我们面临的开发规模的不同等级，分别是：功能（function）、程序（program）、应用（application）和系统（system），如图 2 所示。

图 2 开发规模的不同等级



并不是说一段代码中具有了上述某个级别的关键词就有相应的规模，而只能说这种语言具有对这个级别的开发规模的支持。具体语言对这个级别可能支持得好或不好，但“好”的程度并不是我们讨论的问题，因为它是一个与语言进化相关的、变化的状态。

我们需要另外说一下项目（project）这个关键词。“项目”主要是一个开发过程中的、组织上的用词。就其组织的形式而言，更进一步地会有项目组（project group），退一步则可能是一个活动或事件（action/event）。但是，这些都与我们这里讨论的开发规模无关。你可能为一个只有三行代码的微型程序创建一个项目，或者一个项目组中仅仅只有三个空程序，这些说明不了你的开发规模，而只是你对后续开发活动的组织形式的设定。

所以“语言具有什么样的特性”出自语言设计视角，并对“项目组织成多大”有支持作用，但不是后者的全部。那么接下来的问题还有两个，一是**“为什么存在支持作用”**，二是**“有哪些特性，以及它们有何种的支持作用”**？

5.2 结构化：四种等级的本质

我何以将开发规模分为上述四种等级呢？因为这四种等级随计算机应用的演进历史发展而来，具有各自不同的特性和表现。

功能（function）是计算机的本质能力，它包括最初的计算能力，以及计算能力在数学抽象上的、最大粒度的表达。这类语言的目仅仅是“完成计算”，其特性集中在对计算机的计算能力的抽象和实现上。一般来说，一门语言只要具有：

- 数（number）的定义
- 数据（data）的定义

- 计算能力（formula/expression）
- 逻辑能力（顺序、分支与循环）
- 一个计算过程的约定（命令式、函数式或其他计算范式）

那么它就已经是一门这样的语言了。这在个级别上，最大规模以引入函数这一抽象概念为止（即具有将一系列的计算过程定义为函数的能力）。因为函数等同于“数学定理”，所以他可以像一个公式或运算元一样直接使用。而这样的语言被创生的目的，通常是通过计算能力（包括函数）与逻辑能力，对一系列的数——亦即数据进行计算。因此可以将之定义为：

```
p + f(d)
```

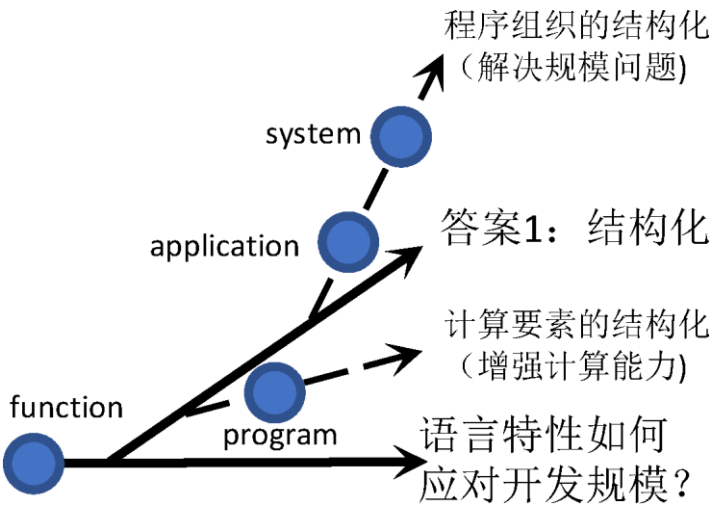
即，“计算范式+计算能力(数据)”：

```
Paradigm + Function(Data)
```

出于表达的需要，我将这一等级上的语言统称为**“计算语言”**（computing language）。需要注意的是，一般来说计算语言并不适合应用软件开发，而只是数学与计算机科学在应用于跨学科研究领域中的工具。

接下来三种等级以**计算语言**为源起，是语言设计在**结构化**这种思想下的、两种方向上的发展结果，如图 3 所示。

> 图 3 “应对开发规模”的结构化求解：四种等级的本质



5.3 方向 1：计算要素的结构化

第一种结构化方向是指：通过在**计算要素**上的结构化来**增强计算能力**。在这一方向上的努力，将我们的开发规模推进到**程序**（program）这一等级。我所见过的，对于程序的最初、最正确和最直观的定义为：

> a + s(d)

即，“算法+数据结构=程序”[①]：

```
Algorithm + Structured(Data)
```

但是这里的结构（structure）主要作用于数据（data），所描述的是对算法(algorithms)中所用数据的抽象。而在更大的计算规模中，计算对象——数据的复杂性固然是一个问题，而计算本身的复杂性

[①] 这个著名的论断出自尼古拉斯•沃斯（Niklaus Wirth）。沃斯是 Pascal 语言之父，1984 年图灵奖获得者。《*Algorithms + Data Structures = Programs*》是他最有名的一本著作。

也是一个问题。因此对于 *a+s(d)*这个定义来说，Dijkstra 在《结构程序设计》中的描述更为完善，即：

```
s(f) + s(d)
```

显而易见，后者用**计算的结构化** *s(f)*来替代了算法 a。原意为[①]：对于一个计算过程，（通过提炼）抽去每次计算所处理的特殊值，余下的不变的东西就是这个“算法”自身。由于对过程中的每一步都进行了“动态提炼（算法）和静态提炼（数据结构）”，因此这里的提炼结果（结构 *s*）在计算过程 *f* 和数据 *d* 上是匹配的、共生的，即[②]

```
s(f(d))
```

其三，程序代码本身在组织上也还存在复杂性的问题。对于这第三个问题，Dijkstra 以“组织和编排程序”为主题加以了讨论。不过在具体背景之下，他所讨论的仅仅是对“函数/过程”[③]进行结构化地组织与编排。

这里需要明确一下在该背景下的**函数**这一概念。在上一小节的**计算语言**中，函数是与数学公理类同的、面向“数”或“数据”的一个计算过程。而在本小节所讨论的程序（program）这个规模之下（以及该规模下的语言之中），函数是算法或算法中的一个子步骤的代

① 原文是“考虑一个算法和它所能引起的各种计算：从这些计算开始，抽去……”。如下一个脚注所谈到的，这里的计算过程是指 procedure，现在则常称为 function，也因此这里使用 *s(f)*这个描述形式。

② 这里确实也可以表达为 *s'(f,d)*，但考虑到这里的 *f* 是必然作用于 *d* 的，因此无疑是说该 *s'()*必然实现为 `return s(f(d))`。

③ 在 Dijkstra 的叙述中，函数是偏向数学计算、有计算结果的；过程只是一段计算机处理，并不表明有计算数据返回。现在我们通常用函数来概含了二者，以 C 语言为例，过程是指返回值声明为 void 的函数。循此惯例，此后的讨论中将仅仅使用**函数**这一个名词。

名词。在 program 这一语境下的函数，与确定的结构有关，既包括计算逻辑的结构，也包括计算数据的结构，是

```
s(f(d))
```

中三者合一的概念。如同《结构程序设计》用**“珍珠”和“项链”**来做的比喻一样，从**珍珠**的角度来看，是可以在整个程序中被替代、被优选的[①]；从**项链**的角度来看，可以是暂时不完整的[②]；从**串起项链**这件事上来看，顺序是确定的[③]。

我将这一等级上的语言统称为**“编程语言”**（programming languages）。我使用这个具有歧义的称谓的原因在于[④]：这一类语言仍然是在**计算要素**上加以增强——尽管对于函数来说也存在着代码的组织与编排上的增强，但归根结底是针对计算要素的，并且是在该抽象上的自然延伸。

在这一阶段中，计算机主要在专业领域中使用，计算量的增加是程序规模变化的主要原因，同时又要求对目标系统有足够的抽象表达能力，因此在数据抽象程度变高的同时仍然强调语言具有完备的计算能力。我们可以将操作系统、网络系统、通信调度、分布式运算环境等绝大多数基础系统的核心部分，以及其中大量利用系统特性的工具软件，都归于这类语言所面向的开发规模。其主要特点是：

① “程序比较（程序验证、测试）”以及渐进优化的思想基础。

② “自顶向下”的结构化设计的思想基础。

③ “架构”以及框架、流程等产出的思想基础。

④ 另外的原因在于，对于许多开发人员来说，在这个级别上的开发才是他们心目中的、（被院校教育所圈定的）计算机科学领域中的“编程”。以其后的例子来说，“写出一个操作系统”可能是许多开发人员心中的巨作。

大量的计算、数据抽象与计算环境相关，主要算法可以在不同的软硬件环境中通用或重现，并且通常用户主要入口是操作系统的约定或直接的硬件系统界面。

5.4 方向 2：程序组织的结构化（从模块化到产品化）

> 问题 3：语法与语义上的特性，是否决定了该语言可能适宜的开发规模？

在第三个问题上的尝试，推动了结构化思想在另一方向上的发展[①]，即通过在**程序组织**上的结构化来**解决规模问题**。这一方向上的主要动力，是程序所应对的需求渐渐扩展到非专业领域，用户的入口变得多样，甚至同一功能涉及到多种用户/角色的、不同时间与时序上的参与。这些也是软件开发工作的成果从专属程序进化到软件产品的主要标志。

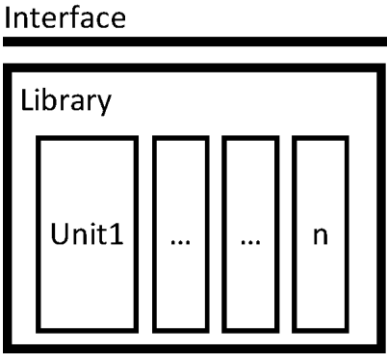
我所观察到的第一个与此相关的语法元素是单元（unit）[②]。因为一段代码可以写得任意长，也可以拆成几个单元，所以这些单元存在与否，就与这段代码要表达的计算功能是完全无关的了——既不是对运算的增强，也不是对数据的增强。接下来，我们发现一些单元可以对许多不同的 function/program 的开发提供支持，由此库（library）的概念就形成了。“库”用于集中管理一些具有相同应用范围、处理相关数据、解决相似问题的单元以及单元对外部声明的界面（interface）。如图 4 所示，这些语法元素之间的关系非常

① 我并不确定 Dijkstra 所述的“组织和编排程序”究竟在何种程度上影响了后两种等级的语言出现，但我确信这种思想是导致它们出现的原因之一。

② 这里指的是 Pascal 语言中的单元(unit)概念。具体到其它语言的实现上，它也可能被称为模块(module)等等，例如 Erlang。

简单。

图 4 单元、库与界面间的关系



大多数情况下，库与库之间的区别仅在于功能集不同、应用范围不同、接口方式不同，以及我们经常关注的效率等的不同。（不考虑这些外在的表现）这些库内在的抽象思想与应用价值是完全相当的。

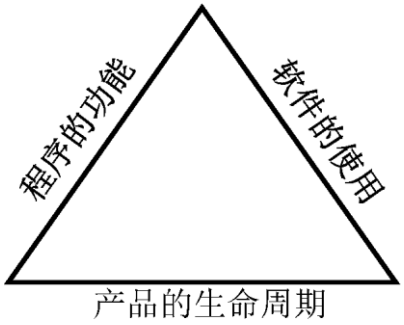
库的出现带来了一些组织程序的策略，例如静态链接库（Lib，Static Link Library）、动态链接库（DLL，Dynamic Link Library）、类库（Class Library）；又例如跨语言的组件库（COM 库、Component 或 Component Library）、公共程序集（Common Assembely）；再例如，远程/跨平台对象（Remoting Object，或 XPCOM）等。这些策略被种种语言实现为不同的、语言内置的关键字，并由此决定了各自不同的一套支持语法。

所有这一切的目的，仅仅是因为那个最原始的发现：总有一些代码与当前处理的业务没有实际关系，可以被隔离到其他代码块（例如 Unit）中去。既然从组织策略来看这种隔离必然会发生，那么我们关注的问题就仅仅是上图的 Library 对外表达的形式（亦即是接口

/Interface），而非 Unit 或类似语法元素的实际实现[①]。

这些变化既有语言进化的内在动力，也有软件规模变化这一外因的推动。当计算机不再仅是用于计算的工具，而变成软件产品的运行平台时，整个软件——传统意义上的程序（program）——的构成已经发生了明显的变化。现在，它必须面临的问题包括三个方面，如图 5 所示。

图 5 面临的问题与背景：Application 在“结构化”上选择不同方向的原因



也就是说，即使程序——我们仍然可以用第二个等级中的“程序”（program）来指代上图中的一个方面——在功能上的规模没有任何变化，那么也将面临如下两类需求：

- 使用人群由专业人员变成普通用户，将导致提出大量的**非功能需求**，即它作为“软件”的使用问题；
- 由于使用人群的变化，维护和发布的工作对程序员变得不可控，因此将导致提出大量的**非当前需求**，即它作为“产品”的生命周期问题。

[①] 这一观点的得来决非易事，它是 20 世纪 70 年代多个程序设计流派之间的主要争端。David Parnas 的“信息隐藏”的观点取得了最终的胜利，即模块内部的数据与过程，应该对不需要了解的信息予以隐藏。Brooks 最初持不同的观点，但在 1995 年的《人月神话》二十周年版中，他坦承“关于信息隐藏的观点，David Parnas 是对的，我是错的”。

这二者，也即是“软件产品”之于“程序、功能”的区别。举例来说，在最初的操作系统中，我们开出一个内存块来使用。

- 首先，我们仅仅是需要在别的计算过程中使用到它，因此它被实现为一个功能（function）[①]；
- 接下来，操作计算机的专业人士认为，其实可以从内存中拿出一块来并将它 mount 成一个虚拟的磁盘，于是将它写成了一个“简单的”程序（program）[②]；
- 之后，某个并不太懂计算机的桌面用户也需要上述功能，但受限于他的计算机操作水平、实际的应用环境[③]，他必然会在上述程序、功能之外，提出前述两类需求。

对于这类（软件产品的）用户，我们称支持非功能需求与非当前需求的程序为**应用（application）**，而将这一等级上的语言统称为**“应用程序设计语言”（application programming language）**。这些需求促使一个应用中包括了多个领域的产品功能，一些显而易见的内容包括：

- 与产品使用相关的界面，例如 3D 操作、游戏界面、数字监控仪表盘等；
- 与应用环境相关的业务逻辑规则，例如企业组织架构、工作流、金融、期货等；
- 与操作平台相关的开发接口，例如 ATL、Android SDK、GTK+等；
- 与发布、维护相关的支持，例如帮助文档、安装包、工程文件管理、代码

[①] 这个功能基本上可以概括为：基于存储地址、用于限制访问边界、纯粹的线性计算（分配）的过程。

[②] 这可能是一个有 150 多个可选配置项的命令行程序，除了`--help`参数就再也没有其他任何称得上文档的东西。

[③] 作为产品的用户，他还期望在个人需要与应用环境都发生变化时，仅支付少量的代价即可应对，而非重新购置。

文档化等；

- ……

这些内容的领域性特点使得任何一个或一类程序员都无法完全胜任它们的开发工作。单元等类似手段以及模块划分时需要隐藏内部信息的思想，都是在这样的背景下产生的。其产生的必然性，是**泛计算领域**[①]本身的纵向隔离的特性所决定的[②]。也就是说，既然这是软件需求从专业计算领域走向泛计算领域带来的规模问题，那么也要求程序在组织上的抽象必须能表达和匹配后者在领域问题上的纵向切分。

5.5 方向 2：程序组织的结构化（服务化与系统构建）

另一个带来崭新思考空间的语法元素是服务（service）。一个服务的发布、运行、使用、受益、检测与维护等整个过程都可能由不同的用户/角色来参与。例如，表 2 比较了生活中的邮寄与软件开发中的会话这样两个“服务”。

① 但凡一切可以抽象为可计算对象，并通过计算系统来解决问题的领域。

② 最简单而又直白的说法就是：隔行如隔山。

表 2 服务：比较生活中的邮寄与软件开发中的会话

		主要的用户可见层					
	发布	运行	使用	受益	检测	维护	其他
邮寄	邮局	快递人员	寄件人	收件人	400客服	邮政主管部门	丢件赔付等
会话*	开发包	会话服务	业务1	业务1..*	会话监视服务	会话管理服务	定时重启、转储、镜像服务等

* 这里假设会话服务提供过程中，所有的服务、业务等都有操作人员；即使部分功能是由自动化服务提供的，我们也可以称之为干系人。

服务在操作系统及其应用软件中也有着重要的位置。以一个典型下载软件为例，它可能提供一个后台传输的服务（backTrans），那么该服务在 Windows 中的提供模式可能如表 3 所示。

表 3 服务：后台传输服务在 Windows 中的提供模式

		主要的用户可见层					
	发布	运行	使用	受益	检测	维护	其他
后台传输	下载软件	Back-Trans	桌面用户*	下载软件	Win-dows	Win-dows	安装、卸载等

* 这里假设这是一个桌面用户可选的服务，当不使用该服务时，基本的下载功能不受影响。

而当一个与此功能等价的服务通过网络来提供时（以 Google 账户同步功能在 Gmail 与 Android 通讯簿功能中的使用为参考），那么上述的模式可能改变为表 4 所示的结果。

表 4 服务：与上述服务等价的服务在网络环境下的提供模式

		主要的用户可见层					
	发布	运行	使用	受益	检测	维护	其他
后台传输	同步服务	GMail、Android framework	通讯簿	用户	同步服务	Gmail	接口协议化等

综观上述这样的一个软件需求与实现的模式[①]，是不可能仅仅以Unit以及更高的形式（库/套件）来提供支持的，因为其需求的本质在于“异地实现”。在这个需求下，由于“服务、功能部件，以及功能部件的操作者”这一系列行为完全不可预期，因此服务的调用方已经不能对实现者的语言与运行的环境作出任何限制。在这个级别上的问题，最终总是被归结为两个解决思路：

(1) 交互界面是否可以表达为**可实现的规则集**；

(2) 输出是否可以表达为**可计算的数据项**。

而简化该问题规模的方法也由这两个经验得出，即尽可能简单的界面规则与数据表示，例如 REST 和 JSON[②]。

对于这类需求模式，我们将提供服务集成能力以支持非本地需求的应用称为**系统（system）**，并将这一等级上的语言统称为**“系统程**

[①] 我们可以将能在产品最终用户的环境中实现的需求称为本地需求，将不能在该环境中实现的称为**非本地需求**。

[②] REST（Representational State Transfer，表述性状态转移）是一种面向远程服务提供的架构方法，JSON（JavaScript Object Notation，JavaScript 对象表示法）是一种数据交换语言/规格。从这个角度来看，SOAP 与 XML 并不是复杂的方案。

序设计语言”（system programming languages）。服务的提供能力与其所支持的层次，成为这个级别的语言特性的主要发展方向。由于服务所在的用户领域有着种种差异，因此在这个级别上的语言也需要提供特定领域的部署、维护与交互界面等特性[①]，例如 Java 中的 Beans、Annotations，或 Erlang 中的 Node、Port 等。

5.6 面对规模问题，我们仍未能超越前人的思想：结构化

从计算机应用的历史来看，我们在语言中加入新的元素，其本质的原因正是旧的语言特性在应对规模（而非仅仅是计算）的时候显得力不从心，尤其是在**应用**与**系统**这两个规模级别中，（在语法与语义上的）语言特性体现出来的代码组织能力，相当大的程度上决定了这门语言所适用的开发规模。

最后，我们用表 5 总结一下不同视角对这些规模的认识。

[①] 这里讨论的是语言，因此限定这些特性是通过语法元素来实现的。尽管在开发包中，用第三方工具来提供支持也是解决这一问题的通常手段，但并不是我们主要讨论的话题。

表 5 不同视角对四种开发规模的认识

规模	产　出	用户特点	开发者类型	语言类型	开发行为	规模例举
功能	计算系统的内置功能或基础构件	计算系统的实现者	程序员	计算语言 Computing Languages	编码	排序算法
程序	在特定环境与规则限制下可用的执行体，基本没有产品化过程	计算系统的专业操作人员	软件工程师	编程语言 Prog. Languages	编程	shell脚本；脚本解释器；学术型操作系统
应用	面对某一个独立领域中的单一问题的产品输出	泛计算领域中的某个或某一类用户	应用开发工程师	应用程序设计语言 App. Prog. Languages	开发	MS Office；自动化办公
系统	跨领域、综合性问题的解决方案	多领域的计算需求，以及多类型的用户	系统工程师	系统程序设计语言 System Prog. Languages	构建	Erlang 通用操作系统

* 关于**软件工程师**这个开发者类型：使用这个有争议的名字原因在于，（基于一些历史的、习惯性的因素，）程序、应用和系统被我们统称为软件，因而“软件工程师”它们统一的基础角色。当然，在此前，他还必须是一个程序员。

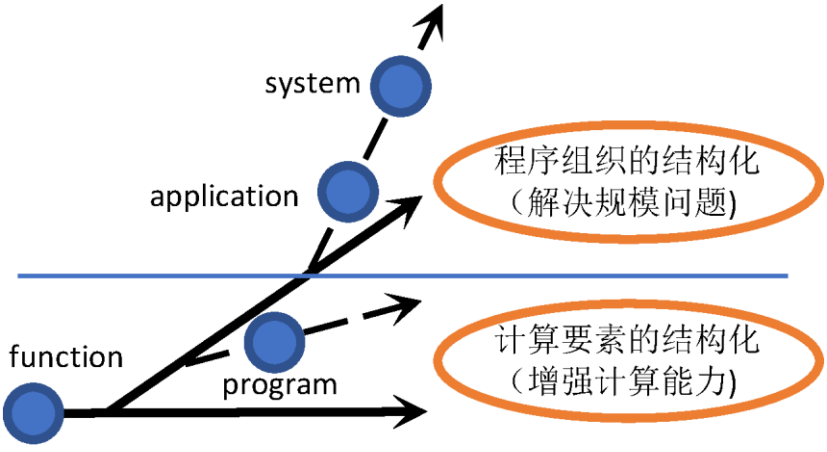
在继续讨论之前，我们先来明确一下将要讨论的对象。

首先，我们要讨论编程（programming），一些场合中也被称为程序设计（program design）。可惜在不太严格的场合下，几乎所有编写软件的活动都被称为程序设计——这是相当可怕的事实。因此，必须明确地对它做一个定义，即我们用**程序设计**来特指在功能与程

序这两个级别上编写软件；相应地，我们将其后的两个规模分别称为**应用开发**与**系统构建**。程序设计、应用开发和系统构建这三种规模下的软件开发活动，是我们后续四篇中将要分别讨论的话题。

这几类活动有着显著的区别。**程序设计**主要应对计算要素问题，产出是一般含义上的**程序**；**应用开发**主要应对程序组织问题，产出是有产品化概念的**软件**；**系统构建**主要应对跨领域问题，产出是可持续进化的**系统**——例如平台，如图 6 所示。

图 6 两种“结构化”方向的本质差异



篇三：程序设计的核心思想

> 但凡与计算机有些关系的领域中，总会有人通过编写一些代码、脚本或批处理来解决问题。这些工作大体上也包含了我们此前讲到的种种计算要素。但是这些只能算作编程，它并不等同于我们在这里谈论的程序设计，它是后者的一个初级阶段。学院式的编程是一种狭义的、升级版本的编码工作，而作为一个软件工程师，程序设计才是我们的必备技能[①]。
>
> 奠定我们如今主要的程序设计方法的那些基础理论和思想，在上个世纪的七十年代就已经成熟了。也正是从那之后，程序设计才从编码工作中脱离出来，成为一个有意义的、独立的名词。接下来的讨论以 Dijkstra 的论述为基础，包括：“计算的结构化 *s(f)*”和“数据的结构化 *s(d)*”两个方面，并且二者存在伴生的关系。

[①] 我这样分类的很大一部分原因在于：有必要将整个软件开发、程序设计阶段需要完成的工作，纳入到我们的讨论范畴。

> 我们应该已经知道：抽象含义上的数是没有类型的。当我们讨论某种数的时候，即是在讨论某一类型的数，而非数的全部。而我们之所以分出这个数的型别，是指该类数具有相同的性质，例如正整数可以由 1 不断复合。同样，数据既然是数的系列，则数据的型别（即数据类型，Data Types）可以理解为系列性质的不同，亦即通过何种系列的抽象来分类数据。何种系列的抽象，即是数据结构[①][②]。

---

① 数据结构与数据类型，前者是方法本身，后者是方法的表示。大多数情况下，也不妨将它们认为是同一个东西。

② 《结构程序设计》的第二篇中，作者霍尔（C.A.R Hoare，1980 年图灵奖获得者）采用数学定义的方式来说明数据类型：设已知有限个的**数**，称其每一个为**基数**，所有基数的集合则为**基型**；由已知类型——含已知结构型和已知基型——构成的类型叫做**结构型**。基型的构造成分唯一，也因此它被称为**非结构型**，它或是计算环境提供的，例如 WORD 与 BYTE；或是程序员定义的有限个数，例如枚举。综上，（数据的）结构是一个过程，或称之为结构化。

第 6 章 数据结构：顺序存储

6.1 规划有限大的空间

所谓“排排座，吃果果，冬冬不在留一个”的故事大概是这样的：小朋友们排成一排或者许多排，然后老师给每个人发一个水果。之所以要先排排座，既是教大家规矩，也是避免老师发漏了或者发重了。大体上，我们的记忆里总会有这么一两个吃果果的场景——老实说，我现在看起来是在讲一个相当无聊的故事。

如果有一个有限大的空间用来放数据，我们能不能也将该空间规划成座位，然后将数据分发上去，从而使得每个局部空间上都有东西（或者还没有，就“留一个”）？这个问题之所以有讨论的必要，是因为只有我们将所有要处理的数据信息都放到计算机可以识别的环境中，计算机才能开始履行我们要它做的事情。

总的来说，我是在讲一个乏味的故事。但在计算机本质的抽象上，的确就是如此乏味。更为“生动”一点的叙述大概是这样：

（1）我们要将所有的数据顺序地放到空间里去；

（2）我们要考虑局部空间上有与没有数据这两种情况。

6.2 用有限大的区域来代表一个待运算的数据

冯•诺依曼体系的计算机，是以控制器、运算器和存储器为核心的，分别映射我们此前讨论的计算（范式）、算和数这三个方面。因此，

如同我们对“算数”的理解一样[①]，**存储器的结构以及它在计算系统中的抽象**，也直接限制了计算及其逻辑。对于这个“抽象”，我们简单地说就是：顺序地址存储。

除了 Bit、Byte、Word、DWord 等与位宽直接相关的、具有数学或物理传输含义的类型——它们显然是顺序表示的值——之外，我们常见的在计算机中用的信息/数据有哪几种形式呢？布尔值算一种，它表示开关、正误等判断，是我们执行逻辑的一个基础；数值算一种，作为数学运算的对象，与我们计算系统的本质设定有关；字符（以及字符串）算是一种，因为它是我们程序员或用户可以正常理解的东西。这三类数据，构成了**计算系统**向（可用的）**计算机**发展的主要条件[②]。而这三种数据中，除了布尔值可以由计算系统的最小存储单元（bit）来表示之外，其他两种都必然面临“如何存储”的问题。

对于这个问题，“顺序地址存储”的核心思想是：设运算器可以通过一个（数值标识的）**地址**，从存储中获得一个（有限大小的）**区域**，则该区域可以表示为一个待运算的**数据**。以 Intel 系统的个人计算机（PC，Personal Computer）中，x86 芯片（CPU，运算器）为例[③]：

- 芯片通过寄存器来读这个地址的值，因此寄存器可以表示的数值大小，决定了地址值的大小；

① 即所谓“算”是程序之表，“数”是程序之本。

② 这里我并不强调它是“必备条件”，因为有些计算机和计算机系统并不是由“人”来使用的，因此与它们交互的数据形式也不相同。关于这一点，应退回到本书“第 1 章 数，以及对数据的性质的思考”中，在与“数据的提出”有关的话题中去讨论。

③ 我们只假设了一种最简单的情况以便于后续讨论。现实是，需要考虑运算器与存储器之间的总线，以及指令与数据缓存、预取装置等，在整个数据的传输过程中的位宽，决定了这些数值的大小以及存取效率。

- 同上，因为芯片也使用寄存器存储计算所需数据，因此上述区域（连续可读取的位数）的大小也是由寄存器可以表示的数值大小来决定的。

那么，首先，待运算的**数据的大小**就被转义成了**区域的大小**，进而变成了运算器与存储器之间的**位宽**问题。所以 32 位的机器上一个（能直接参与 CPU 运算的）值的大小，就是 2^{32}，如果要理解为“有符号整型数”则是 $\pm 2^{31}$，因为符号用掉了一个位（bit）。同样，我们要运算浮点数的话，也要把它“挤”在这 32 位中去表示，例如常用的 IEEE 754 浮点数表示法[①]。再者，我们要表示字符的话，也是通过对 32 位的组合序列做出定义，将它们一一表示为我们的书写字符，例如 ASCII 字符集、GB2312 字符集，以及 UTF8、UTF32 字符集等。

接下来，可能表示的**地址**，也被转义为上述的位宽问题。因为地址是数值标识的[②]，所以它表达的是一个连续空间中的位置（可以想象为数轴上有限区间的点）。根据上述的原则，以 32 位的 CPU 为例，该“顺序地址存储”方案：

- 可以找到的最后的数据的位置索引就是 2^{32}，
- 可以从该位置（以及其他任意位置上）读取的最大可能值是一个无符号的 32 位整型数（DWORD）[③]。

---

① 这样的表示法显然有非常非常多种可能性，而且也确实存在着好几种在用的方案。

② 地址标识与寻址问题是一个复杂而一体的问题，并非此处所说的“整型增量”这样简单。例如，在硬盘读写的寻址方面，就有多套完全独立的体系。这些体系与磁头、磁盘片以及高速马达的惯性等都有关系。地址标识与如何提高设备存取的效能，是计算机系统中“存储”相关的焦点问题。而在此处及后文中，我们仅从软件开发视角上讨论该问题的一个子集，并且事实上“顺序地址”也是软件开发中有关存储的一般抽象。

③ 这里涉及表示的地址索引与存取大小之间的约定。实际上 x86 约定的地址是字节序的，因此在最后的这个地址上只有一个 Byte 能被处理，而其他的位置将因为无法编码地址而溢出。

所以跳开我们熟悉的 x86 芯片去做软件开发的时候（例如单片机），我们常常需要问：寻址空间有多大，基本数据单元是多大。如果没有这样的信息，就无法通过某种语言编程去控制它，因为连一个基本的、与计算机交流的环境都搭建不起来。

6.3 在更大的区域中表示完整含义

上述这些被我们的语言称为“基础数据类型”。这包括两个部分，一是与位宽相关的，例如 Byte，它是计算系统的直接映射；二是与应用环境相关的，例如 Char，它与该系统对外的表示有关[①]。它们事实上本身就是对数据的结构化表达，最明显的就是浮点数的组合型表示方式，又例如字符的顺序型表示方式。但总的来说，这些基础数据类型，总是能“挤”在一个最大位宽的表示单元中去。

换而言之，就是在连续空间中表示**完整含义**。

然而就最初的需求来说，所谓**完整含义**是我们对数据而不仅仅是对于数的假设。例如，下面的数据是有完整含义的：

```
Hello World!
```

但这个数据如何在连续空间中表达呢？从自然形式的理解来看，它是 12 个字符。因此即使以每 8 位表示一个字符而言，它需要 96 个位。对于“顺序地址存储”来说，在 32/64 位机器中，即使能通过一个地址来找到它，也无法一次将它全部取到 CPU 中去运算。但是，它本身又的确是连续而完整的，我们不能孤立地从其中的一个部分来理解它。由此产生了一个问题：我们——程序员以及计算环境——该如何理解超过“（有限大小的）**区域**”的数据呢？

[①] 例如一个图形计算环境，就可以考虑以 RGB 为基础数据类型并建立起基于此的运算体系。

答案是：有一个**起始地址**的**连续**区域。

这里的起始地址与此前的“顺序地址存储”中的地址是同一概念，但“连续”性也必须体现为数值才能为计算系统所理解。因此对于这样的数据，我们需要两个数值来定义：地址、（连续的）长度。现在，由于有了“长度”的概念，因此我们通过“顺序地址存储”可能得到的数据就有了非常大的变化：

- 其一，可以找到的最后的数据的位置索引不变，仍然是 $2^{32}-1$；
- 其二，可以从该位置识别的最大可能值是一个 0…$2^{32}-1$ 长度的连续区域[①]。

或许你现在已经想到了“指针”（pointer）？不，我们现在还没有讨论到它。我们仅仅在讨论一个连续的存储结构，例如数组、字符串或结构体。

6.4 “有一个起始地址的连续区域”思路下的两种数据类型

数组（array type）指的是包括**某种相同数据**（数组元素的类型相同）的连续空间[②]，它是**顺序地址存储**这一概念的自然延伸[③]。首先我们认为 Byte、Word 等基础数据类型是受位宽限制的顺序地址存储，当我们把位宽限制这一条件去掉——或通过长度值来指定连续区域的大小——之后，就得到了数组的概念。由于它自然延伸了（但未改变）**顺序地址存储**的概念，因此它也可以作用于上述这些基础类型。例如：

---

① 与此前讨论的地址索引问题类似，这个连续区域的实际大小也受限于可用的、能被编码的地址大小。

② 我们这里先讨论程序设计中一个狭义的数组概念，之后的讨论中会再进一步完善它。

③ 理解为抽象概念中的“引申”。

- DWORD，等义于长度为 4 的字节数组；
- BYTE，等义于长度为 8 的位数组；
- INT64，等义于长度为 64 的位数组，或长度为 8 的字节数组，或……

基于此，我们也可以用数组这一概念来统一所有的基础类型，这最终可以将任何数据理解为**位数组**（bit array）。当然，需要强调，这里的数组是指一个连续空间中的数组，否则就与我们此前的抽象不一致了。

然而，我们留意上述的“某种相同数据”这一抽象限制条件，也就意味着，我们必然会面临“连续空间中包含**某几种不同数据**”的需求。我们做出这一“必然”判断的原因，是我们的需求总是问题的全集而非某个部分（所谓可能出现的，必将出现）。因此对问题的某一个分类中的所有可能性施以数据抽象，则它必然可以表达问题的全集，以及满足其背后的全部需求。推论上述逻辑：

- 设定：在连续空间（S）中，要么包含同一种数据（m），要么包含不同种数据（n）；
- 如果存有混杂，则可以将它分解为多个连续的连续空间(S_i)，使(S_i)符合上述设定；
- 如上，总是可以用 m 与 n 来表示所有数据，并保持它们在空间上的连续性，亦即是 S。

结构体（struct type）指的就是包括**某几种不同数据**的连续空间[①]。这一概念是对数组的补充，他们一起构成了“用基础数据类型”来

[①] Struct type 一般译作“结构（类型）”，这里用“结构体”以与本书中讨论的、普遍含义上的“（数据）结构”区别开来。一些语言中，结构体也被称为记录，这同样也是数据库中“记录”称谓的源起——本书后文中还将讨论到结构体与数据库之间的抽象关系。

复合其他类型的全部可能性。

6.5 关系型数据库与顺序表

上述的“全部可能性”事实上还应当包括一种泛义的数组，亦即是元素的类型为某种**结构体**的数组。在数据结构上，它通常被称为**顺序表（sequential list，或 list）**。

设数组 `A` 的每一个元素的数据类型为 `T`，基于此前的讨论，元素（结构体）`A[n]`必然有一个确定的长度值 `Size(T)`。由此，数组的长度——顺序表中的记录数 `RecordCount`——决定了整个数据所占用的连续空间的大小：

```
RecordCount * Size(T)
```

在该连续空间中，可以通过数组下标——顺序表中的记录号 `RecordNo`——来访问任意元素，它的地址也是确定的：

```
RecordNo * Size(T)
```

其中 `RecordNo` 的取值空间为一个序列值`[0...RecordCount-1]`。

顺序表具有边界判断简单、能快速存取指定位置的特点，到目前为止仍然是关系型数据库的最基本的、最佳的实现方案。关系型数据库中的表格（table）与顺序表在抽象含义上是相同的：每行——每笔记录——的字段列即是数组元素的结构类型定义，行号（`RowId`）即数组元素的索引下标。因些，结构化查询语言（SQL）中的一行代码：

```
select * from A where RowId = 5
```

与将 `A` 作为数组来存取的时候所采用的操作：

```
A[5]
```

是完全等义的。

6.6 顺序存储的抽象本质：索引数组

综上所述，我们事实上可以用两种方法来统一顺序存储，由此将一个无限大的空间视作空间连续的数据，并使用地址来存取其中任何数据分量。这两种方法是：

- 将包括某种相同数据的连续空间视为数组 A；
- 将包括某几种不同数据的连续空间视为结构 S。

由于 S 本身是连续的，所以：

- S 可以视为有且仅有单个元素的 A。

所以：

- 整个空间可以统一为 A。

在得到这样一种数组概念的同时，我们也找到了在与计算系统交互时，表示数据的基本方式。它引申数列概念以通过下标索引值来定位数据分量，因此也被称为**索引数组（index array）**。

在不讨论存储的、纯粹抽象的数据结构的概念集中，我们将索引数组的概念加上结构体，就看到了顺序表（list）；加上操作方式，就看到了栈（stack，LIFO）与队列（queue，FIFO）。

6.7 指针既是对顺序的结构化存储在运行期的补充，也是天堂与地狱通行证

顺序存储中用到了两种数据结构概念，即数组和结构体。在此前的讨论中，我们预设了两个前提，其一是顺序存储，其二是数组元素

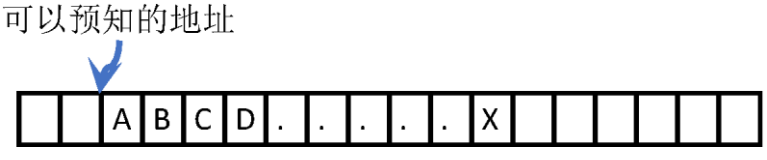
的长度总是能预知的。后面这个条件往往并不能满足，例如我们有一个程序的功能就是“让用户手工输入元素长度”，那么在编写该程序的时候，（程序员）就无法知道如何分配存储了。

这种情况下，程序员在编程的时候知道一个标识，并且要基于该标识来编写后续逻辑，他只是无法将标记与可能的值关联起来而已。例如下面的形式代码：

```
var pArea;  // 无法预知该标识的值
function process(Int len) {
  pArea = allocMem(len);   // 无法预知该标识的值的长度
}
```

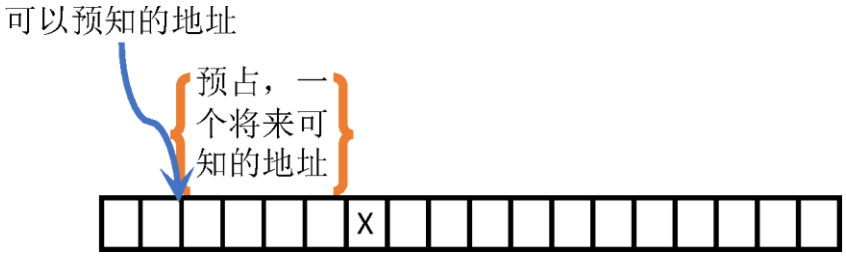
顺序存储的限制条件需要“地址+长度”两个条件，而上面的例子意味着在程序正式运行起来之前只有“地址”是可以预设的。因此我们只能将该地址“预占”起来（记得排排座故事中的“冬冬不在留一个”吧），以便将来用于找到真实地址。下面对比这种情况与一个实际的顺序存储之间的差异。首先是以数组为典型的存储效果，由于长度是已知的，所以程序员可以安排自 X 位置之后存储其他数据（见图 7）：

图 7 已知数据内容的顺序存储



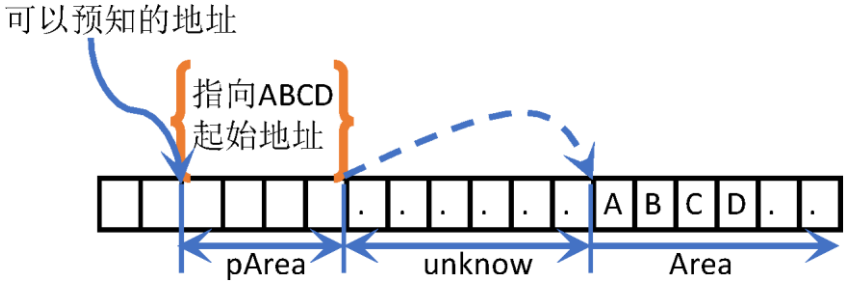
然而，在**待处理的数据**长度未知时，由于“用于存储一个地址值”所需的长度是可知的，所以我们采用“预占”存储效果如图 8 所示：

图 8 数据内容未知时，预占“存储地址值”所需的确定空间



此后程序员仍然可以安排自 X 位置之后的存储——当然，也可以通过编写程序来设计其后的安排。例如，在程序运行过程中，我们在预占位置填写一个**实际的地址值**，以说明**实际的数据**的存储位置，如图 9 所示：

图 9 保证顺序存储的方案：通过填写预占位置，指示实际的存储位置



通过图 9，我们也可以发现，`pArea` 本身和 `Area` 自身都是连续的。如果 unknown 部分也按照上述规则来处理，则 unknown 也将是连续的。如此，我们仍然可以保证整个存储空间是连续的。

现在，我们知道这里所谓的 `pArea` 就是**指针（pointer）**，它是对**顺序的结构化存储**这一方案在运行期的一种补充，满足两个条件：

- 它是一个标识，有一个计算系统访问它的地址（记为 `p`）；

- 它包含数据（值），该值是一个（与它关联的、实际的）数据的地址（记为`p^`）。

在程序中，通过上述方式来设计数据结构时存在两种可能。其一，如果一个地址是可以先期知道的，那么`p^`与值的绑定是静态的、在系统运行之前发生的，因此它也是一种安全的指针。其二，如果一个地址是不可预知的，那么`p^`就需要在运行期再动态地绑定它的值；在正式绑定值之前，`p^`就是一个游离的、无值的标识。

如同我们此前所说过的，如果标识与其值未能绑定，则它的抽象含义——或称之为“计算中的数/数据的含义”——就是不确定的，这也意味着，在这样的模式上建立的计算系统是不安全的。换而言之，指针的动态绑定（以及解除绑定）是一种不安全的机制。

第 7 章 数据结构：散列存储

7.1 哪种情况下，做记号的法子才确保能行得通呢？

排排座的主意其实并不太好：我们总是要尽可能将小朋友排到最前面，否则空缺一旦多了起来，老师们就不大可能记得住了。

于是老师们想起了一个故事。据说，在很久很久以前，有个楚国人在坐船渡河的时候，身上所佩的宝剑掉到水里去了，于是他在船上刻下了一个记号，说：这是我掉剑的地方。当船停在岸边的时候，这个楚国人就跳下船去，沿着这个记号往下找啊找啊……

不对，好像故事讲错了——这个可怜的楚国人好像找不到他的剑了吧？

老师们很快意识到这个问题，于是立即换了另一个著名的故事。又据说，在很久很久以前，有人送给曹操一头大象，但当曹操问这头象有多重时，却难坏了他的官员们——仍然是据说，提出把大象砍成许多段、排成排、逐一称量的那个家伙已经被推出去先砍成许多段了。这时呢，有个叫曹冲的小朋友跳出来说，我们先把大象推到船上去，根据水面位置在船上刻个记号；然后再把大象换成许多石头，直到船沉到记号的位置；最后我们称一下那些石头不就行了吗？

咦！问题解决了！

但新的问题又出现了：为什么同样是做记号，楚人的法子行不通，曹冲的法子却行得通呢？更进一步的问题是：哪种情况下，做记号的法子才确保能行得通呢？

7.2 欢迎来到“名/值”数据的世界

此前我们讨论到，所有的数据最终都可以被抽象为数组，或者说我们此前讨论的所有数据类型都可以视为数组类型的转义。我们也讨论到，如果一个数组能确定长度，则它总能在运行前被顺序存储；如果它不能确知长度，也可以通过指针来保证在运行中实现顺序存储。

但是我们似乎多了一个题设。我们需要设问：顺序存储是必须的吗？

顺序存储的必要性在于：

- 运算器需要通过一种简单的方法来找到数据；
- 可以通过地址的余量来确知存储的占用情况，进而给程序员控制它的可能。

但是这两项条件都是与机器实现有关的，前者是运算器寻址的需求，后者是存储器容量的限制。而这些——我们必须强调在思想上要突破所有的限制——不是“计算”所必须的。换言之，一个计算过程仅仅是**需要数据**，并没有限制**用何种手段来获得数据**。

据此，只要其抽象结果不会影响计算过程，我们就可以将数据的那些**仅仅应用于具体语言实现的**抽象概念一一抽离。仍以下面的代码为例：

```
var
  Number aNum = 100;
```

这一句代码（的语法），对应着四个语义：

- 有一个标识，记为 `aNum`；
- 其数据类型为 `Number`；
- 标识所指代的数据，其值是可变的（即，变量 `var`）；

- 该数据当前值为 100。

当我们把“值可变”与“当前值”这两个——由存储问题推导出来的——概念抽离之后，一个数据就只有三个普遍含义了：

- 名字，即标识：`aNum`；
- 值，即数据：`100`；
- 类型，即数据类型：`Number`。

此前我们也提到过，顺序存储中我们可以将所有数据类型都视为数组的转义，因此我们也可以把“数据类型”从这样的系统抽象中抽离——当一个事物不存在类别含义时，也就无需识别它了。这样，这个数据抽象系统就只剩下了[①]：

```
aNum = 100
```

现在，我们来到了“名/值”（name/value pair）数据的世界。

7.3 解决第一个问题：名字组合的可能性是无穷的

使用“名/值”关系来确立的数据抽象系统，既有可能确切地找到数据，例如曹冲称象；也有可能找不到数据，例如刻舟求剑。但是这一抽象准确地反映了计算系统的真相：找到数据，计算。

本质上来说，索引数组也是这一抽象下的产物，只不过我们是使用**可顺序索引的地址**来找到数据罢了——地址，也可以视为数值体系下的标识或命名。而我们现在，只不过要假设命名的方式不再是地址，而是一个真实的、可以被自然语言理解的“名字”，例如字符串。

[①] 本质上来说，我们是回到了对“数据的性质”的讨论，参考本书第 1 章。

第一个问题。在索引数组中的地址是有限大小的，它与我们的计算系统的存储一一对应，因此使用地址来指示一个数据时，不可能因为越出存储边界而找不到数据。但是我们使用字符串来命名数据的时候，就不能确保数据与存储的这种关系：我们有无穷的方式来生成名字，也就意味着它对应一个无穷尽的数据集合。而这是无法被一个物理的、现实的计算系统——例如计算机——所支持的。

可以用数学方法将一个组合方式无穷的名称集合映射到一个有限的空间中去。这种方法称为**哈希（hash）**。简单地说，它使用一个函数来为每一个名字生成一个有限空间中的索引，例如数字值[①]。这涉及两个问题，其一是映射为数字的索引值；其二是限制在有限空间中。对这两个问题的一种简单的处理，是将字符的编码值求和、取余。例如：

```
// JavaScript Syntax


/**
 * 前设：name 是一个顺序存储的字符串
 *   - 我们可以用 GetStringLength()函数来获得字符串的字节长度；
 *   - name[i]这种语法形式将 name 作为一个字节数组(byte  array)使用；
 *   - name[i]形式可以取得指定下标 i 的数组元素的数值值，范围为[0...255].
 **/
function hash_sum_16(name) {
  var result = 0, len = GetStringLength(name);
  for (var i=0; i<len; i++) {
    result += name[i];
  }
  return result % 16;
}
```

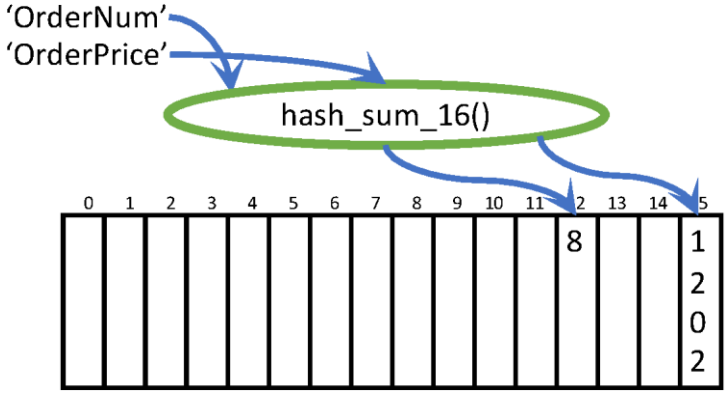
[①] 哈希也称为散列，它仅指这种映射关系而并不要求映射的结果是一个数值，例如用于加密的 Hash 过程，就是将源信息映射为加密指纹信息。

对于某笔订单的基本信息，其字段名的实际运算结果如下：

```
// 计算结果为：12
hash_sum('OrderNum');
// 计算结果为：15
hash_sum('OrderPrice');
```

图 10 表示上述关系在存储上的映射，亦即是上述字段所对应值分别存储在位置 12 和 15 中：

图 10 一个简单的哈希算法在存储上的映射



这表明我们可以通过 `hash_sum_16()` 找到[①]：

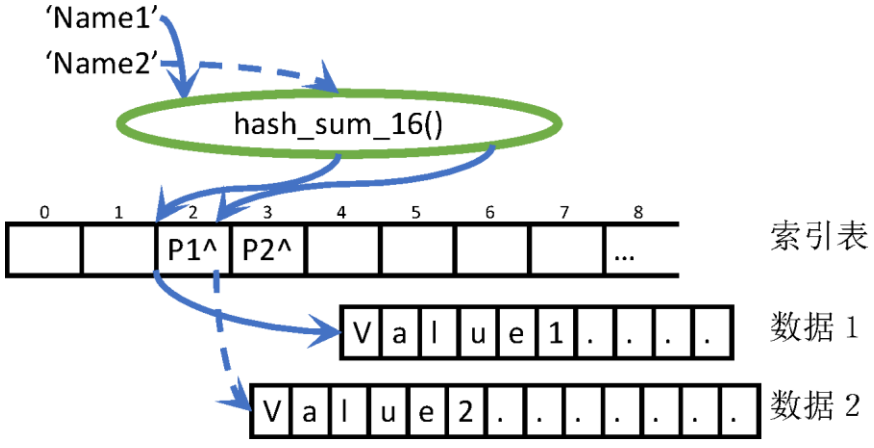
- 名为`'OrderNum'`的数据的值为 8；
- 名为`'OrderPrice'`的数据的值为 1202。

由于求模取余之后，`hash_sum_16()` 的可能值为[0...15]，所以我们能预先分配上述整个索引表区间（16 个基础类型的数据，或者结构体等）。更进一步，我们也可以通过指针来弥补索引表对未定长度数据支持不足的缺憾——这是因为该表本身是一个索引数组，如图 11

① 这些值的计算含义在这里不需要关注，现在只关注如何建立正确的“名/值”关系。

所示。

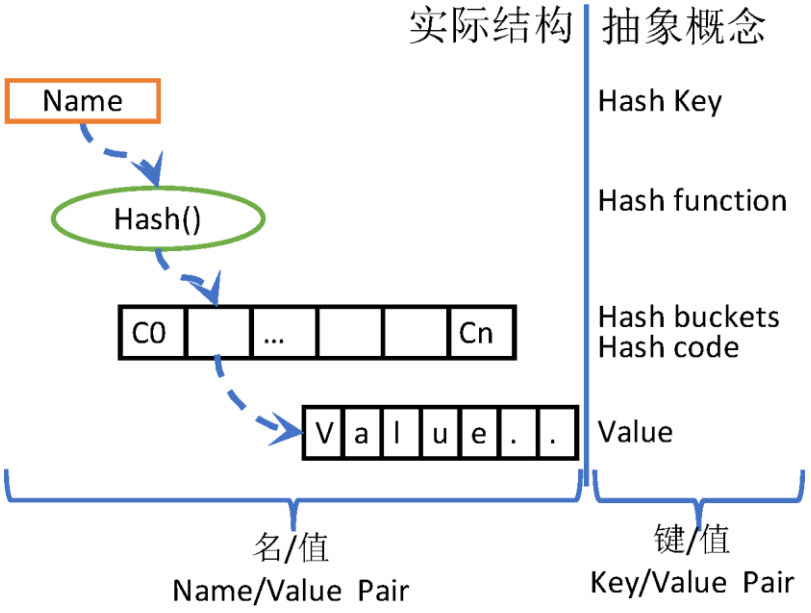
图 11 指针在哈希表上的运用



根据此前的讨论，我们可以保证图 11 中的索引表、数据 1 与数据 2 以及整个空间仍然是顺序存储的，进而不会与计算机的物理实现冲突。

在完整的模型描述中，我们称 `hash_sum_16()`这类函数为哈希算法或哈希函数（hash function），将名字称为键或哈希键（key，hash key），将索引表称为哈希表或桶（hash bucket），将表中的元素 $C_0...C_n$ 称为哈希码（hash code）。图 12 描述了这些抽象关系。

图 12 哈希表的结构与概念间的抽象关系



7.4 Key：对名字不可或缺的验证

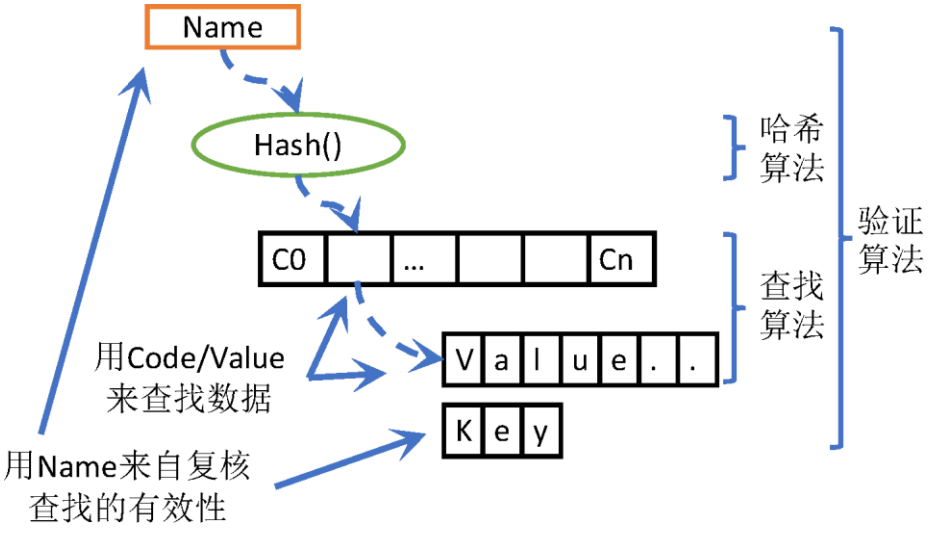
显然，（图 12 右侧抽象概念中的）“键/值”作为抽象系统，的确可以让我们通过标识找到值。我们甚至可以为**索引数组在这个系统上的抽象**定义一个函数——换言之，索引数组也就是这个系统的一个特例：

```
// 索引数组是以下标为标识的、自映射的一个“名/值”数据系统
function hash_index_array(i) {
  return i;
}
```

但是我们也发现正因为索引数组是自映射的，所以它的映射关系是一对一的（传入 `i`，返回 `i`），而此前的 `hash_sum_16()` 却是多对一的，例如 `'OrderNum'` 映射为 12，但如果传入标识 `'Order'`，其映射的结果也会是 12。

因此我们面临要在同一个位置上放两个或者更多数据的情况。这就是**哈希冲突（哈希碰撞，hash collision）**。对此我们可以增大哈希表的长度，但即使它超过可能的哈希键个数，（在不同的算法下）仍然存在冲突的可能。既然我们承认冲突必然存在，因此在找到 Value 之后再做一次验证就是必需的了。这里的验证也有许多种做法，其中之一是用某种新算法再做一次 `Hash()`。尽管两个不同的 Key 在两种不同的 `Hash()`之后仍然相同的机率相当低，但是——仍如此前所述的——还是必然会存在。所以最终的一个步骤通常还是对 Key 的直接识别，例如字符串的逐字符比较，或者数据存储区块的逐字节比较，或者基于顺序存储的、区块地址的比较。这种情况下，数据区必须保留原始的 Name/Key 值，如图 13 所示。

图 13 验证：保留 Name/Key 的原因



有两种特殊情况，其一是元素在系统中不一定完全出现，但其可能值是预知的，例如一周的七天；其二是元素总是会完全出现在系统中，例如键盘上的所有键。这两种情况所面临的数据集都是有限的，对此我们仍有机会设计一个函数来使得每一个 Key 所计算的 Code 都

保持唯一。这样的哈希表是静态大小的，其长度为数据集范围的上限，即：

```
Get_Length(Hash_Buckets)  ==  Get_Element_Count(Data_Set)
```

这其实是为每个元素创建了唯一的哈希码映射，只需要比对哈希码值即可确认数据，也因此就无须再保留原始的 Name/Key 值了。这同时也节省了图 13 中最后一步比对 Name/Key 的开销。

7.5 万法归一：索引数组是关联数组的特例

曹冲称象本质上是通过一个系统将大象的重量映射为石头的重量，而哈希算法正是这样一个系统。对于象与石头的重量来说，这个映射关系是确定的、一对一的，因此刻度也就只需要简单地核对，而无须“原始的象”来参与复核。

但在另一个故事中，楚人为什么就失败了呢？

因为哈希算法的背景变了。需要留意的是，楚人的哈希算法：

- 求掉落位置之于船体的相对位置

没变。算法的结果 HashCode：

- 刻度

在整个过程中也没变，但是楚人计算刻度时的背景是船体，使用刻度（下水找剑）时的背景却是整条河。所以“名/值”数据系统与其存储背景有着相关大的关系，“必须在相同背景下创建与维护哈希表”是一种系统负担，这种负担通常是由语言或某种硬件装置[①]来维

[①] 这是安全令牌、加密盾等实时密码发生装置的基本原理。

护的。

对于在某种特定背景下建立的一种“名/值”关系型的数据系统，我们称为**关联数组（associative array）**。这种抽象的数据结构在基于地址存取的机器中应用时，面临的核心问题是：

> 作为名字的字符串存在无限的组合，这与有限的地址空间是矛盾的。

而哈希机制/算法通过将“名/值”映射为“Key/Code/Value”的关系，从而解决了上述问题。这一映射关系是“哈希算法+存储背景”来约束的，这既是系统负担，但也使系统从根本上避免了刻舟求剑的笑话。

解决了与地址空间的矛盾之后，关联数组得以在顺序机器中实现。它在物理上仍然能够满足顺序存储的需要，但从逻辑上却分离了名字（标识）存储与值存储的关系。而我们知道，标识与值是数据最重要的两个性质。

总的来说，如今我们在计算世界里使用的数据表示方法只有两种，即关联数组和索引数组。无论是在抽象概念还是具体实现上，索引数组都可以视为关联数组的特例，既有存储限制，也有存储限制带来的名称/标识限制。

第 8 章 执行体与它在执行过程中的环境

8.1 总有些知识是可以复制的，反之亦然

河岸。路人甲。

如何过河是一个问题。一般来讲，其他可能的选择是不过河或者绕过去，但前者不是甲当前的选择，后者则充满了变数。对于聪明的路人甲来说，他找到一棵树，挖空了树干，做成了一艘（原始的）船。于是他到达了对岸，离去，留下一个名为“船”的东西[①]。

如何行船以到达对岸？这个问题的解被路人甲带走了，尽管船还留在那里。如果你能从路人甲——或者其他人、使用手册或者你努力地思考——那里得到这个问题的答案，那么你就会用船过河了。

总有些知识是可以复制的[②]，例如船和行船的方法。复制这些知识，就可以得到一个算法；只要条件合适，就可以得到相同的解。

那么，什么才是合适的条件呢？

8.2 船的原型与知识

下面是“一艘船”：

```
// 代码基于 JavaScript 语法
function ship(people, water, checkShore) {
  do {
```

① 这里用“船”而不是“独木舟”（canoe/boat）的原因，仅出于中文在行文上的方便。

② 这也意味着有些知识是不能复制的。

```
    people.row(water);
  }
  while (!checkShore());
}
```

这艘船造得并不怎么好，但这是一个稍晚一些才会讨论的问题。在这里，我们先关注其中的要素，这包括既存的 `ship`、`people` 与 `water`。其中，`ship()`封装了：

- 知识 1：`row` 是 `people` 的一个行为；
- 知识 2：`row` 这一行为的使用；
- 知识 3：`checkShore` 这一检测行为的使用；
- 知识 4：不断 `row` 直到 `checkShore` 得到正确结果这一过程。

除了 `ship()`自身封装的上述知识之外，以下知识也是存在的：

- 知识 5：如何实施 `row` 这一行为；
- 知识 6：如何实施 `checkShore` 这一检测行为。

更进一步，上面的 `ship()`还隐含了一个不确切的知识：

- 知识 7：`checkShore` 是谁的行为？

如果我们（我的意思是说，作为 `people` 的行船者）正确地理解与支持上述知识，那么 `ship()`可以帮助你到达对岸。

我想已经有程序员开心得大叫起来：总算看到了“面向对象编程”（OOP，Object-Oriented Programming）了。

没那么快。

8.3 行船方法论

因为我们得先讨论行为。如前所述：

- `ship` 本身（包括知识 2、3、4）和知识 5、6，是**定义行为**；
- 知识 1、7 是**找到行为**；
- `ship()`、`row()`、`checkShore()`是**行为**[①]。

行为总是未可知且无穷尽的，例如路人甲面对河岸时可能的选择与解。但是“定义”与“找到”行为的方法则可以确定。对于前者（定义行为），它与声明一个变量本质上没有区别，例如：

```
var ship;
```

对于后者（找到行为），如果 `ship` 可以像变量一样被定义，那么我们找到它的方法也就与（此前我们讨论过的）“通过标识找到值”没有什么不同。换言之，

> 总是可以将变量和方法统一为数据。

8.4 数据（亦或知识）的生存周期

回顾上一节的所有讨论，我们在语言中使用一个数据的方法，根底上只是如下过程：找到它，使之参与运算。而关联数组使“找到数据”这件事变成对一个计算背景的维护。例如，我们有一段代码：

```
var
  a = 100,
  b =′ abc′ ,
  c = false;
```

这些数据的定义可以被理解为一个背景的建立（当然，我们也可以为零个数据建立一个背景），因此我们得到一个关联数组：

---

① 我们将 `ship` 理解为行为的定义，而将 `ship()`理解为行为的能力——这出于程序与抽象表达上的需要，而与英文的惯例是不同的。一旦使用 `ship()`，则说明是指该“划船”作为一个系统的、整体的行为的实施过程。

```
aAssociativeArray = {
  'a' : 100,
  'b' : 'abc',
  'c' : false
}
```

接下来我们在这个背景环境中运算。但——根据语言的不同——我们可能又需要“即用即声明”一个数据，例如：

```
for (var i=0; i<100; i++) {
  ...
}
```

而 `i` 这个数据的出现，意味着我们需要在 `aAssociativeArray` 中添加一个新的 Name。虽然 `i` 的值是可变的，而在整个过程中 `i` 的名字却不变，因此我们对于 `aAssociativeArray` 的 Name 只有添加和删除的需求，不需要因为值的改变而导致 Name 的改变。更进一步，

> 我们事实上是将一个数据的生存周期映射成了一个 Name 的增删。

8.5 关联数组可以维护一个计算过程所需的一切参考

接下来我们讨论与此相关的四个问题。在讨论语言与关联数组的性质的过程中，这些问题分别在以下章节中出现过：

- 第 4 章，第 4.5 节 你真的理解这行代码吗？
- 第 7 章 数据结构：散列存储

其一是增删 Name 的必要性，即如果一个语言认为变量是计算的前设——亦即是它不支持变量的动态声明，则我们只需要一个静态的关联数组来维护其背景，否则我们需要一个动态维护的关联数组。这

意味着我们在静态语言和动态语言[①]中都可以使用关联数组。尤其重要的是，对于静态语言来说，我们可以在编译期使用关联数组，而无需在数据区保存一个用来复查的 Name 列表。

其二是 Name 作为前设，但其值仅在计算过程中可知，这一绑定关系可以变成在上述背景中的一个值的修改。也就是说，

```
a = a * 5
```

可以映射为：

```
aAssociativeArray['a'] = aAssociativeArray['a'] * 5;
```

与此相关的，`a` 的确定性（是否可以改变值）也可以映射为对 `aAssociativeArray` 中的名字 `a` 的访问限制，即 `aAssociativeArray['a']` 是否可写。

其三是如果我们需要在计算过程中新建一个数据，它的含义无非是我们要在上述关联数组中添加一个“名/值”。例如我们在上述背景中执行如下代码：

```
eval('var x=1000')
```

如果这行代码得以执行，它表明需要动态创建 `x` 这个名字，我们只需要将该 Name 添加到 `aAssociativeArray` 中，并置值为 1000 即可。当然，这也相应要求 `aAssociativeArray` 是可以动态维护的。

其四，如果我们试图在运算过程中访问一个并不存在的名字，对于没有这一项需求的静态语言来说，`aAssociativeArray`（可选的）可以不保存 Name 列表；对于动态语言来说，它将表现为在 `aAssociativeArray`

[①] 我们还没有讨论到动态/静态语言这一分类方法。这里简单地从标识角度来定义它们，即静态语言是指标识（例如“变量”）在程序执行前就已经完全预知的，动态语言则允许在程序执行过程中新建它们。

中无法查找到某个 Name。

可见，对于一个计算过程来说，关联数组可以维护它所需的一切参考，所有的数据性质都可以表述为关联数组与其存取的性质。我们称这一运算所需的参考为**上下文环境**（context，context environment）。上下文是保证一个运算具有确定性的主要方式，换言之，它相当于语言中的语用这一要素。对于计算来说，在此前的讨论中我们说过“数据确定，则运算确定”，因此关联数组本质上也只是在这种情况下为计算维护了一些数据而已。

8.6 一门语言与一个程序的区别，仅在于参考环境的差异——后者被称为运行时环境(Runtime)

对于 `ship()` 来说，所谓**合适的条件**就是类似这样的一个上下文环境：

```
_ship_env_0: {
  people: {
    row:       /* 划水 */
  },
  water:       /* 水 */,
  checkShore:  /* 检测岸 */
}
```

我们来考虑类似环境的可能性。例如，假设我们将原始问题中的“水”和“检测岸”确定下来，则 `ship` 将可能有许多种方案。如：

```
_ship_env_1: {
  water:    /* 水 */,
  checkShore:   /* 检测岸 */,
  ship: function(people) {
    // people.row 的可能性是无穷的
  }
}
```

假设我们也将“人”确定下来，那么方案就只剩下一种：

```
_ship_env_2: {
  water :     /*水*/,
  checkShore: /* 检测岸 */,
  people:     /* 人 */,
  ship: function() { ... }
}
```

除非我们——人——能有不同样的 `row()` 方法。

在从 `_ship_env_0` 到 `_ship_env_1` 的演变过程中，我们把 `ship()` 和它相关的环境 `_ship_env` 绑定在了一起。这是一个不小的变化，因为它的含义是语义与语用之间的绑定关系。如果我们将 `_ship_env` 分离出去，则 `ship()` 将是 DSL（领域特定语言，Domain Specific Languages）的一个实现；如果将 `_ship_env` 与 `ship()` 绑在一起，那么 `ship()` 就是一个应用程序中的确定求解。

前者创建了一门语言，后者创建了一个程序。

8.7 所谓操作系统，不过是参考环境更复杂的执行体而已

我们已经讨论四点：

- 关联数组的实质是“名/值”映射；
- 代码中的数据与逻辑可以统一为数据，进而统一为标识；
- 关联数组可以维护一个计算过程所需的参考，亦即上下文环境；
- 对于静态语言来说，上下文环境的实质是一个不需要名字/标识的（或称标识可选的）关联数组——这种情况下，它更像一个索引数组。

综上所述，对于静态的、编译型语言来说，下面的代码：

```
// Pascal Syntax
```

```
var
  x1 : Integer = 100;
  x2 : String = ' Hello.' ;
procedure P(y1: String);
begin
  ...
end;
```

意味着 P()这个计算过程所需的环境有两个，其一是 x1，x2 … xn，它们在 P()进行计算之前就可以预知；其二是 y1...yn，它们在 P()计算的当时才会得知。对于编译型语言来说，它认为：

- X 是 P()计算的前设；且，
- X 与 P 本身是后续其他计算的前设；继续推论之，则，
- 所有在最外层出现的标识是整个系统运算的前设。

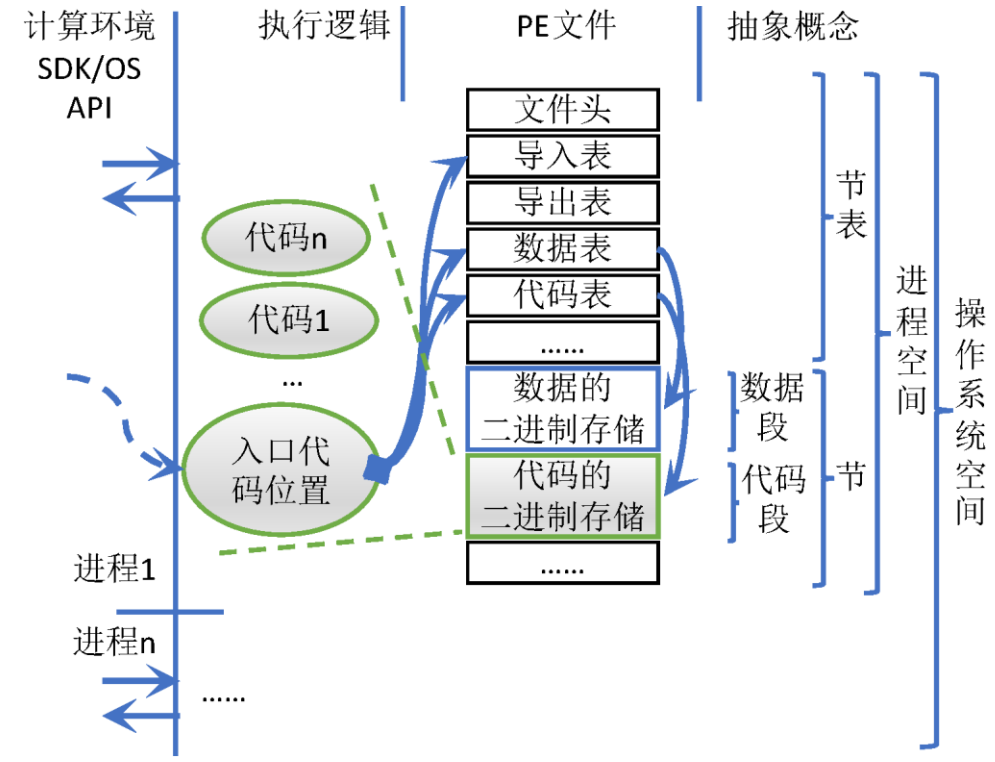
由此，编译型语言通过编译器程序

- 将所有这些标识与其内容——例如数据、函数（逻辑、行为），
- 根据所有计算过程约定的计算环境——例如操作系统，
- 按照确认的规则——例如可执行文件格式，

编排并放在一个实体文件中。最后，操作系统（桌面用户或服务进程）决定何时将该实体文件装载并运行（Launch）起来，以完成所需的全部计算。

我们看到上述过程的限制条件包括：操作系统环境、可执行文件格式和编排方法。这些是程序得以执行的要件，但并非我们需要讨论的内容主体。因此，图 14 以 32 位 Windows 为例，仅大体说明这些要件之间的关系：

图 14 从一个入口开始：操作系统准备的执行环境



参考图 14，对于操作系统来说，仅需要知道一个程序的**“入口代码位置”**。例如：

```
// C Syntax
void function main(int argc,char *argv[]) {
  ...
}
```

随后，入口代码将按照规则在**“代码表”**中查找标识，例如：

```
// C Syntax
function P() {
  ...
}
void function main() {
  P();
```

```
...
```

最后，如果 `P` 或 `main` 需要数据，则使用类似的方法通过**“数据表”**来查找标识。

编译型语言在程序执行前就明确这些标识，因此编译过程可以省去这些名字，而指代以存储中的索引，亦即地址；因为上述规则是操作系统对存储的约定，所以对于编译型语言来说，该可执行文件（PE 文件，Portable Execute）可以置入存储环境中，以应用上述地址；因为操作系统知道上述 PE 文件的入口地址，所以只需要：

- 装载该 PE 文件、
- 分配地址、
- 交出 CPU 的执行权限，

然后就可以完成整个计算过程。

如上图所见，节表是节的索引，是节中的数据与代码[①]（及其标识），是整个计算过程的参考——上下文环境。推而广之，对于整个进程来说，它可以通过导入表来获得整个操作系统的上下文环境；细究之，对于一个内部函数来说，它可以通过入口参数表来得到上下文环境（例如此前讨论的 `y1...yn`），进一步也可以得到进程的、操作系统的上下文环境。

总结这所有的行为，仅仅只有两种[②]：1、找到（包括逻辑在内的）数据；2、计算。

① 这里是特指可以执行的机器指令。

② 这里的推论是相当重要的。综合上述的讨论，我们对于一个环境的依赖，事实上可以看成对“定义与查找（数据）”这样的行为的依赖。再进一步，这也就揭示了我们的计算环境之于“数与算”的抽象本质，以及将数据结构（包括以此为基础的种种算法）作为核心研究领域的真实原因。

那么对于解释型语言来说，上述过程是更复杂，亦或更简单呢？

第 9 章 语法树及其执行过程

9.1 概念笼子：十个或是更多

我们已经在讨论一种具体的语言实现形式，即所谓“静态的、编译型”语言。这事实上涉及到两种语言的分类法，其一是静态与动态的，其二是编译与解释的。

在计算机语言实现中的所谓“编译”，是不同于**翻译**（translation）的。从本质来说，我们所使用的任何计算机语言编写的代码都需要通过**翻译**才能交由计算机执行，因为计算机最终是只能理解开关状态的电子电路。但是在最终抵达目标语言——即机器语言，或称之为电子电路的序列行为——之前，我们事实上会经过不止一次的翻译过程。在这些翻译过程中，**编译**（compile）与**解释**（interpret），都不过是其中的“某一个过程，或包括多个过程的阶段”的代名词而已。

问题是这个翻译过程可能有相当多的步骤，而且今后步骤还将有增减（或因系统复杂而变多，或因优化而变少），所以几年前作为名称的“编译或解释（这些过程）”与今天就可能根本不同。为了避免这类问题，我们采用两种极端的方法来定义它们，即对于一门语言翻译结果的**执行体**：

- 如果是可以直接指示 CPU 行为的指令，则我们称之为**编译型语言**；否则（必然地），
- 它需要被某个软件再经过（至少一次）翻译才能得出上述指令的，我们称之为**解释型语言**。

而“静态与动态”这样的分类法，则是源起于语法与语义之间的绑

定关系。其中所谓的语义包括数、数据与逻辑等。一般来说，语言中会通过“有值但没有标识”的数据来表达**“数”**这一概念，即[①]：

- 对于没有标识的代码（这种数据），我们称为**匿名函数**（或**匿名方法**等）；一般性的没有标识的数据，我们称为**直接量**（或**字面量**、**立即值**）。

接下来，我们前面所谈到的数据结构最终被表示为（语言中的）数据类型，数据类型也因此可以理解为数据在语言中的性质——除非一个数据是无结构的。那么将语义绑定于标识时[②]，事实上语法元素就有了对这些值、数据类型以及对逻辑（代码、执行体）这一特殊数据的考量。作为一个约定，如果在代码执行之前

- 代码与该段代码的标识是绑定的，并且
- 任何上述代码所使用的数据的标识具有确定的数据类型含义[③]，

则我们称该语言是一个**静态语言**。

9.2 从静态到动态

相反地，一个语言环境中也可能存在四种与上述**静态语言**约定不相容的情况。

其一，在执行之前存在标识但没有任何已知的值，代码运行时该标

[①] 对于“数”的支持，在动态或静态语言中并没有根本的不同，但其中的匿名函数在静态语言中实现起来要困难很多——不过这并非无法实现，例如 Delphi 2009 以后的版本。

[②] 与上述“匿名 XX”相对应的，我们称之为“具名”。

[③] 注意这里没有强调值与标识的绑定，这是因为存在变量的缘故。对于 Erlang 这类语言来说，所有**“作为系统计算的前设”的值**都必须是常量性质的，即：不可写且类型确定。

识存有绑定任何值的可能，称为**动态绑定**[①]。例如下面的代码行在 JavaScript 中表示环境中有一个标识 `age`，但直到该行代码执行时，`age` 所指示的存储中才会有值 20：

```
// 示例 1
var age = 20;
```

与此相对，我们称标识在运行之前就具有了值和类型的情况为**静态绑定**。例如在 Delphi/Pascal 中：

```
// 示例 1
Var Age : Integer = 20;
```

其二，对于已经（动态或静态）绑定过的标识，如果通过写值的方式能够使之具有新的数据类型含义，我们称为**动态重写**[②]。对此前的示例 1，我们可以有如下代码：

```
age = "unknown";
```

这会导致 `age` 具有新的数据类型，而这在 Delphi/Pascal 中会被视为违例。

其三，动态绑定或是动态重写导致标识具有不同数据类型含义的情况，称为**动态类型**。示例 1 中的 `age` 在动态绑定发生之后，JavaScript 环境允许使用 `typeof()` 取得其类型为 `Number`，但在绑定发

① 动态/静态绑定这个概念被用在很多的地方，我们这里只明确地阐述其中的一种形式。即一个无类型、无值的变量，可以动态绑定一个值来赋予它类型的概念，即后文的动态类型。除此之外，我们并没有讨论在静态语言中存在的一个有类型变量，在执行期进行绑定（赋值）的情况。

② 重写一个具名函数看起来是一个动态特性，但它其实并没有改变标识上的数据类型，所以不作为这里的动态重写来讨论。从另一个角度来说，具名函数的重写与 `i++` 的性质是一样的。

生前则是未定义的[①]。

其四，如果允许将（任意或特定数据类型的）数据作为代码片段加以执行，则称为**动态执行**。例如：

```
eval("dynamic_code_text");
```

当然，这也意味着多数脚本语言具有动态执行的性质，只要它们能读取文本文件内容并执行。

对于支持上述（“静态绑定”是显然被排除在外的）四种性质中的一种或多种性质的语言，我们就称为**动态语言**。

所以我们此前将部分讨论限定在“静态的、编译型”语言之中，实在是有着严苛的条件：既要能直接编译成机器语言，又要能明确有标识和数据类型含义。接下来，我们的讨论则要广泛得多。不过此前讨论的那些性质，在后续讨论中也仍然是合用的——这本来就是我们构划这样一个讨论路径的原因。

9.3 在计算系统上的实现语言，其本质是：找到数据

编译过程将代码变为机器指令，这使得代码必然面临机器环境的限制，例如存储的使用与 CPU 执行权限的交接就是这类语言中主要的两类复杂问题：内存分配和线程调度。解释型语言将程序的入口由 CPU 指令序列变成代码的自然入口，散列存储使我们避免面临存储地址的限制，这些技术可以让语言从“机器问题”中解放出来。这一结果让我们认识到，机器环境的限制只是语言实现中的负担。

[①] `Undefined` 在 JavaScript 中也是一个类型，但这是概念完备性的一种表现，并非我们这里讨论的数据结构。

因此，我们的问题得以回到此前的讨论：程序的所有行为无外乎两点，其一找到（包括逻辑在内的）数据；其二计算。亦即是说，在一个解释性的，或不以机器问题为焦点的语言中，“找到（包括逻辑在内的）数据”必然会成为本质、核心和关键的问题。表 6 列出了整个系统中的语法元素的性质。

表 6 语法元素的性质

行为	对　　象	示　　例	备注
定义	数据	obj	
	行为(逻辑/运算/过程等)	method	
计算	（无）	foo()	
	数据	obj.method(data)	
	行为	obj.sort(sort_func)	*
	行为+数据	enum(enum_proc , enum_struct)	**
值	（无）	return	
	数据	return result	
	行为	return function() { ... }	
	数据+行为	return {scope}	***

* 可用于指定行为所依赖的行为，或关联项。例如 JavaScript 中的排序，可以指定排序过程。

** 作为通用行为，必须传入行为具体过程与参考数据。例如 Win32 SDK 中的 EnumWindows()的列举。

*** 返回同时包括数据与行为的一个结构。例如 JavaScript 中的闭包。

通过“定义”，我们确定系统中的元素必然只存在数据、行为。这

种定义通常是语法性质的，但最终总是可以表达为一个“名/值”映射——虽然这并非实现上的必需。需要留意的是，尽管表 6 中刻意从“数据”中把“行为”分离出来，并将后者进一步地分解为`计算()→值`这样两种，而事实上它们都可以视作数据，可以用相同的方法**定义**，并施以相同的**行为**。

所以我常说，程序是“被定义出来”的。因而我们通过思维就可以完成全部程序，而它在执行过程中的排错、修正与优化等，是出于我们在工具使用上的、熟练度训练的必需，而非编程思维训练的必需。

必须在这里指出的是，无论我们的编程语言如何变化，上述都是语义完整性的需求，即最大可能的集合。当然，其前提在于我们对“定义”的假设，如果所定义的语法元素更多，那么这一组合的空间就会变得更大，在语言的语义表达上会趋于丰富，进而可能无法控制；如果所定义的语法元素变小，例如将数据与行为统一为数据，则语义表达上会更简洁，但也可能出现（用自然语言来理解时的）歧义[①]。

最后，我们也可以推论出：既然“找到（逻辑与数据）”本质上也是一个行为——或称之为计算，那么它必然也满足表 6 中有关“计算和值”这两类行为的全部约束，亦即它可能实现的全部方法[②]。

---

① 也许有人会说“元素更少，则更易理解”，但并不全然如此。抽象概念过少的时候，我们的表达是倾向于重叠指代的，例如“船”，既是名词也作动词。在编程语言中，这通过 `ship` 与 `ship()` 来表达，便已经具备了两个符号（与其组合）——这使得我们在抽象概念上，至少已经分出了数据与行为。

② 在《数据结构》中，这被称为“检索”，它与下一小节要讨论的“排序”是两个非常关键的“（计算）行为”。

9.4 没有更多的技巧——排序，然后顺序执行

所谓翻译（编译与解释）无非是将一系列的源代码文本（所对应的计算行为）排列成一个顺序序列，而最终的执行器——无论是机器还是解释器——只是从**这个顺序序列的起始位置**开始处理，一直到结束[①]。正是因为这个缘故，所以**依据什么排序**才会是翻译中的一个关键问题。

语法树是对语法元素进行排序的一个方案。这一方案将语法元素解析为不同类型的树结点，例如操作符结点、操作数结点等。以下面的代码为例：

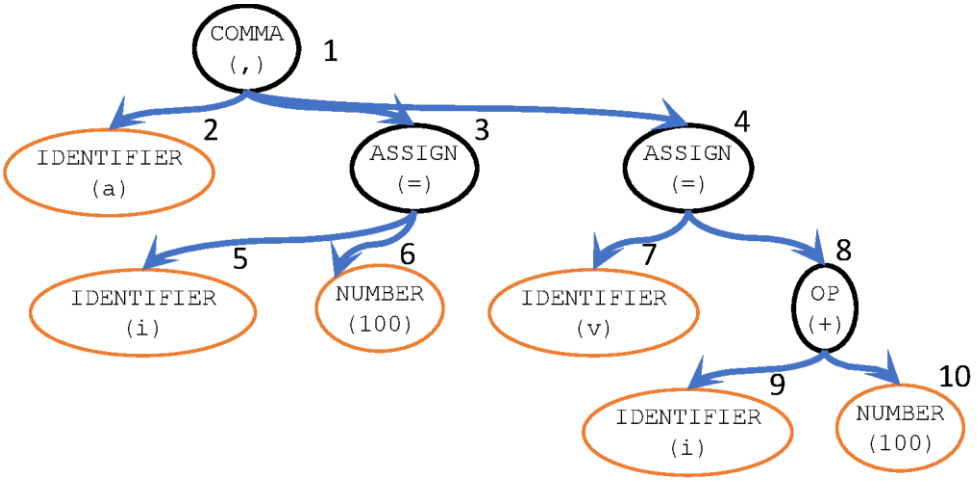
```
a, i = 100, v = i + 100;
```

它可能被解析成如图 15 所示的语法树[②]。

[①] 这种结束有两种可能，其一是序列完全计算完毕，我们通常理解为“退出”；其二是序列进入一个无休止的循环过程，我们通常理解为“等待—就绪”。对于执行器来说，这个顺序的代码序列（静态文本的翻译结果）所代表的执行逻辑（动态的处理）既可能是操作系统所调度的 CPU 执行权限的入口，也可能是一个解释器的执行片段的起始——不幸的是两种情况都被称为 process，只是前者通常译为进程，后者则译为处理。

[②] 本例是 JavaScript 解释器引擎的一个实际解析结果。需要留意的是：不同的语言以及其翻译器在解析细节上可能会有不同，因而可能产生别的结果。

图 15 语法树：对语法元素进行的排序



在该语法树中，所有叶子结点都是标识或直接的值——它们代表数据，而非叶子结点则是操作，亦即逻辑；如果是逻辑，则计算的结果可以作为其他计算所需的数据，例如图中的结点 8 的计算结果是结点 4 操作数。

当如上的语法树形成之后，顺序扫描树中所有结点的方法有两个：深度优先遍历与广度优先遍历——理论上说存在无数种可能的方法，这取决于形成该树时所使用的算法。以使用深度优先遍历算法为例，我们将得到下面这样一个处理顺序：

```
2,5,6,3,7,9,10,8,4,1
```

去除掉标识和数据部分，我们看到的是：

```
_,_,_,3,_,_,__,8,4,1
```

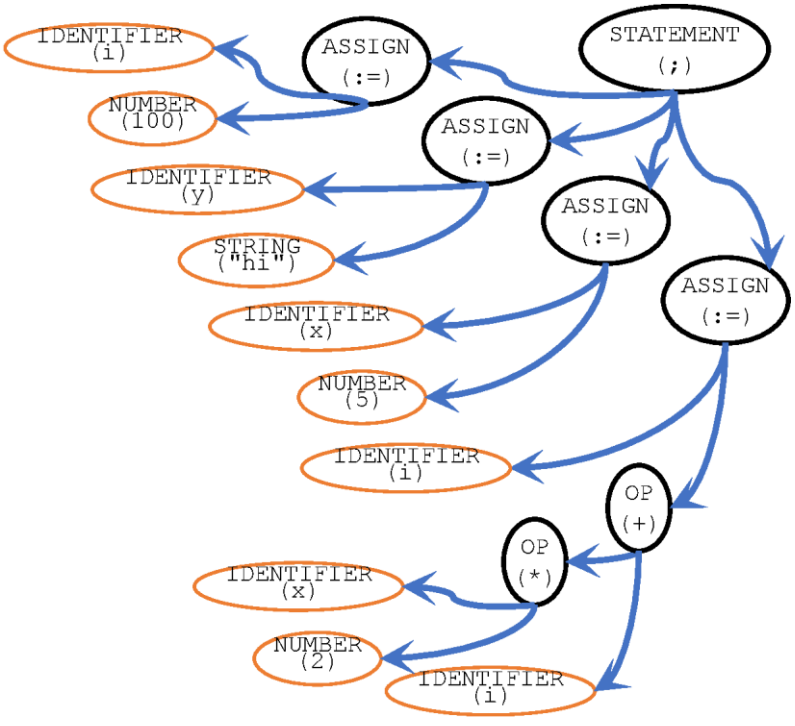
这几个操作的顺序，即我们对“顺序计算（逻辑）”在语法树上的重现。

那么，以数据排序又会是怎样的一种情形呢？例如，下面的代码：

```
i := 100;
y := "hi";
x := 5;
i := x*2 + i;
```

可以产生一个类似的语法树，如图 16 所示。

图 16 生成的语法树（为分析“基于数据的排序”而进行了排版变化）

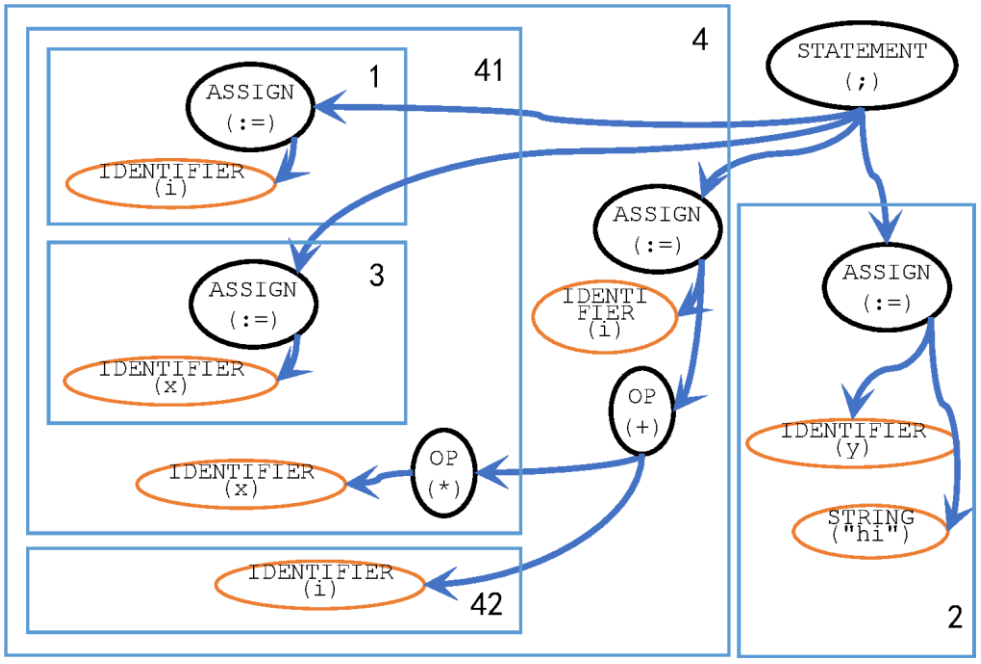


（除了版式上的不同）这个图与此前的图并没有本质的区别，我们仍然可以按上述遍历规则得到一个执行顺序——如果你将这个图理解为**基于运算的排序**的话。但是，我们这里要讨论的是**基于数据的排序**，即：运算影响到哪些数据，以及其影响先后的问题。基于此前的讨论，我们可以将“数据”理解为运算所需的参考——上下文环境，然后

- 将图 6 中树的每一个运算放到它所需的上下文环境中，并
- 将这些环境用框图及其层次表达出来[①]。

如图 17 所示。[②]

图 17 基于数据的排序：将“数据”理解为运算所需的参考



可见，本质上来说，我们要正确处理此前的代码文本，则意味着我们将对上下文环境的相关性进行排序：内层是被依赖的，因此排序优先级更高；同层不存在数据/上下文环境依赖，因此优先级相等

① 内层的环境是外层的环境的一个参考，或称之为历史环境的一个映像。

② 其一，图中的 1~4 为代码行号。其二，为方便绘制，图中省去了直接值结点，只留下表示数据依赖关系的标识。

（例如 1 与 3，以及 2 与 4）[①]。

比较两种方案：以数据为顺序时，标识符的作用域（亦即是上下文环境）问题将会绑定在语法树上；而以逻辑为顺序时，作用域问题与语法树是无关的。

[①] 注意由于优先级相等表明了不存在同层相互间的依赖，也就意味着同层的代码文本是可以并行执行的；当以类似闭包（scope）这样的方式来解析代码并表达它们的层次关系时，是原生支持并行语言的。

第 10 章 对象系统：表达、使用与模式

10.1 抽象本质上的一致性

所有被计算的数据，我们要么看做是顺序存储的值，要么看成是散列存储的“名/值”。但这只是对数据全体的一个认识。针对其中一个局部，例如某几个地址上的连续数据，或某几个“名/值”数据，我们又如何认识它呢？它们在数据概念上又是什么？

在顺序存储中，我们已经为这样的数据找到了一个“看起来合理”的名字：结构类型/结构体。基于这一存储方式的特点，我们对结构体中的“字段名”并不敏感，例如说：

```
// C Syntax
struct info {
       char first[10];
       int age;
};
```

这个结构有`first`和`age`两个字段名，但忽视这两个字段名，我们以下面的结构体来操作它：

```
struct info {
       char data[14];
};
```

也没有什么不同。更进一步，它与一个字节数组也没有什么不同[①]：

```
byte info[14];
```

但是散列存储中的**某几个“名/值”数据**就不可能这样消化掉了。例

[①] 在一个语言的具体执行环境中，还涉及边界对齐问题。

如：

```
// JavaScript Syntax
info = {
  name: 'Tom',
  age: '32'
};
```

其中 `name` 与 `age` 这两个名值对，其内部是有依存关系的——它们分别是 `info` 的不同性质；也有其外延，即可以作为其他的数据的性质，例如：

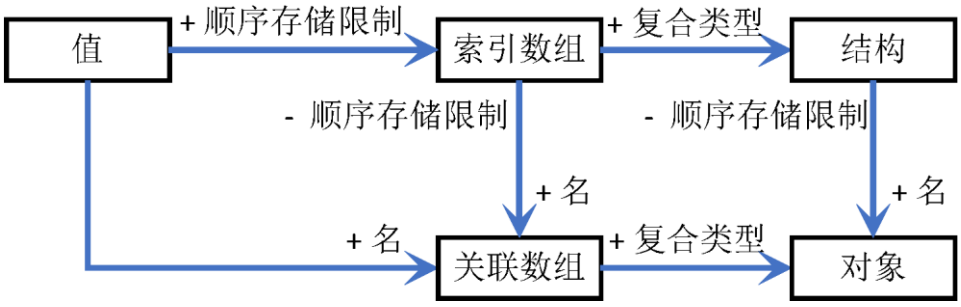
```
aMan = {
  baseinfo: info,
  snsinfo: ...
};
```

如我们此前讨论的，`name`、`age`（以及更多个）这样的名/值构成了一个数据系列，并满足了我们对数据的内涵与外延的定义，因此它必然可以被理解为数据。而其作为数据，其**何种系列的抽象**，亦必将为一种数据结构。

这种系列的抽象，我们称为**对象**。对象是一种数据结构，其每一个分量数据为一个**属性**，属性可以是对象、索引数组、关联数组，或它们基础的、延伸的数据类型之任一。

对象可以看成是关联数组的一个自然延伸。我们知道，关联数组在概念上与“去除存储限制的”索引数组可作类比；与此相似，对象与“去掉存储限制的”结构体也可作类比，图 18 表明了这样的关系。

图 18 “结构”与“对象”在抽象本质上的一致性



可见，数据的最终抽象，究竟是“结构体”还是“对象”，只是缘于我们在“如何找到数据”这个问题上的不同选择[①]而已。除开这一点，二者殊途同归，在其抽象本质上是一样的[②]。

10.2 继承和多态都是多余的概念

对象是带有一系列相关性质的数据。除此之外，它不带有任何必须附加的概念。明确这一点，有利于我们看到对象系统的本质与构建过程。

在《结构程序设计》中，达尔[③]用一种非常奇特的方式来定义了类与对象之间的关系，即：如果一个过程能够产生比调用语句存活得更久的数据实体，则我们称该过程为**类**；这样的数据实体（实例，instance），我们称之为**对象**。更有趣的是，《结构程序设计》将

① 通过地址或名字来找到数据，这是目前计算机发展中对这个问题的两个答案，但并非只有这两个可能答案。

② 所以我们看到最新的 Go 语言只支持所谓结构，而.NET 中的结构与对象之间的界线也相当模糊。

③ 奥利—约翰•达尔（Ole-Johan Dah）与霍尔合写了该书的第三篇。达尔被称为面向对象之父，是 2002 年图灵奖得主。霍尔同时是该书第二篇的作者，是 1980 年图灵奖获得者。

这样构建对象系统的过程称为“层次程序结构[①]”。不过这其中的“层次”却并不是指类的继承层次，而仅仅是指“按层次的方式构造和分析”去处理系统的复杂性问题。达尔在书中强调了“层次方式”是唯一有效的办法，但并没有认为系统的逐层分解与对象的继承之间存有某种必然联系。

事实上在Simula 67最初的“面向对象”观念中，对象只是一种（相对于一般数据类型而言）更为高级的数据抽象形式。直到 1971 年 SmallTalk 才提出了继承性概念。如今，对象的 PME（Property-Method-Event，即属性、方法、事件）模型，以及 EIP（Encapsulation-Inheritance-polymiorphism，即封装、继承、多态）构成了完整的面向对象编程的概念集。

但这些都并不是**对象**这一概念抽象的本义，而是实现**对象系统**过程中的一些实践。例如，我们可以将基于继承的对象系统[②]，视作是通过层次方式来处理系统复杂性的一个实践。准确地说，它通过**类属关系**从开发目标复杂无序的数据中抽取了一部分出来，使它们成为一个可编程的、（在一定程度上）可复用的对象集。正因为这一过程只处理了**具有类属关系的层次数据**，所以——必然地——下面这些问题也就局限了基于继承的面向对象系统的应用：

- 无类属关系的数据；或
- 非层次的数据；或
- 系统中的逻辑需求。

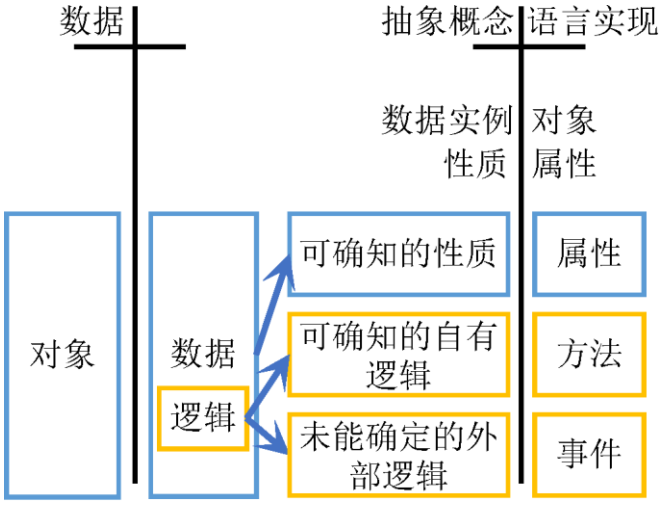
[①] 这里的“结构”更适宜理解为动词。

[②] 注意我们并没有强调是类继承的对象系统，这是因为类只是实现继承的方法之一。其它的继承方式包括原型继承、元类继承等。

10.3 对象是“属性包”：方法与事件可以视为属性的特例

将对象作为“一种数据”来观察，则所谓 PME 中的方法（M）与事件（E）就可以视为是“面向对象逻辑需求的”特殊属性（P）。其中，方法是对象创建之前即可确知的自有逻辑，事件则是在对象创建时仍有未定因素的外部逻辑，这二者本质上也是对象的性质。如图 19 所示。

图 19 PME 作为语言特性所实现的抽象概念

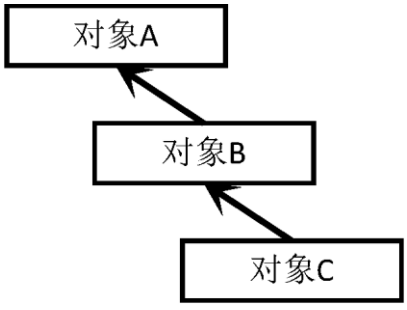


所以 PME 其实只是概括了对象属性，而无助于得到这一对象本身。也就是说，从 PME 的角度上来思考，只是对对象的细节刻画；如何从目标系统中识别到该对象，则是刻画它的前提。

对象的**继承性**则是对“对象如何获得”的一个解释。如同我们说整数作为数据，在它的系列中的关系为“+1”一样，继承性约定了对

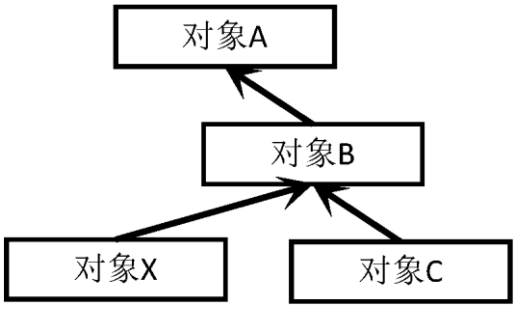
象——这个数据在它的系列中的“关系”为层次继承[①]，如图 20 所示。

图 20 基本的对象继承



若对象 C 继承自 B，B 继承自 A，则 C 必继承自 A，即 C 必具有 A 都有的属性性质。如同人位于生物继承关系的最末端，因而人必然有所有生物的所有基础性质。又如图 21 所示。

图 21 对象继承性的衍化与证明



[①] 在“10.1 抽象本质上的一致性”中我们曾经将“（对象）可以作为其他对象的性质”理解为它作为数据系列时的外延。当时我们只讨论了它结构化的一方面而没有拓展这一外延性质，因为“数据的可引用性”是结构类型（Struct types）的基础，是对象之于结构类型类似之处。然而现在我们约定了系列关系之一为“继承”，其基础是“层次间的相似性”，在这一语境之下可引用性将是实现继承的一个工具，例如“封装”。不过仍然需要强调的是，“继承”是系列关系之一，而非惟一，亦非必须。

基于以上推论，若对象 A 具有某唯一性质，例如“A 是对象”，则 X、B、C 都“必为对象”。因此，对象的继承性保证了对象系统中的每一个数据都必然是对象。

但是对象 X 与对象 C 又各自具有不同性质，例如人与马具有所有陆生动物的性质，但人与马又有不同。这一点，又是通过对象的**多态性**来说明的。多态性意味着对象 X 与对象 C 是在对象 B 的基础上相同的，但在其各自的表达范围中却是不同的（多态）。例如图 22 说明对象 X、C 在范围{B}中是相同的，但在范围{X}和{C}中各自不同。由此，我们可以进一步推论图 23 中的对象 X1、X2 与对象 C 在范围{B}中也是相同的：

图 22 X 与 C 是范围{B}中同类对象的多态

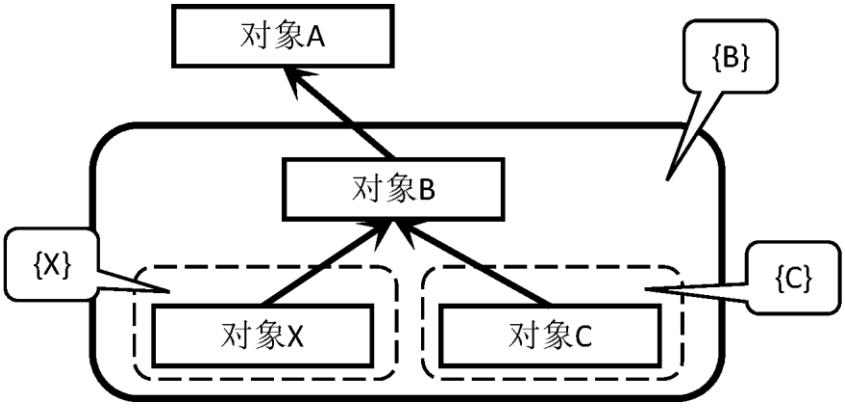
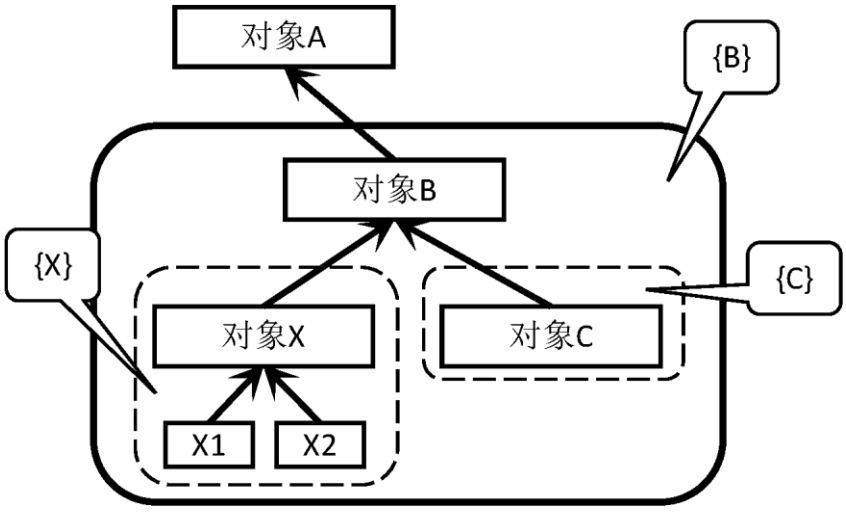


图 23 继承性决定了 X1、X2 与 C 在范围{B}中的多态关系



由此可见，多态性是对象的性质在其继承关系上的表现。由于继承关系本身即是附加的性质，因而多态性也是附加的性质。

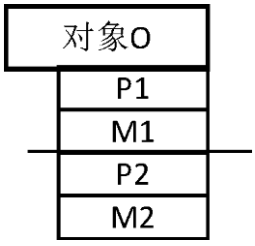
10.4 可见性同样也是多余的：它是对继承性的补充与展现

我们再来略为讨论一下可见性的问题。

所谓可见，是数据相对于计算（的需要）而言的——如果数据不需要参与计算，则讨论它的可见与不可见本身就失去了意义。我们这里所说的**计算**，在对象系统中被称为行为/方法（且方法本身也是对象的性质之一）。这些与计算相关的因素，综合起来有如下几种情况。

其一，以一个对象来说，属性若以“被计算数据（P）”和“计算行为（M）”来区分的话，则 P 本身就有对 M 的可见性问题，如图 24 所示。

图 24 对象的数据之于行为的可见性问题



M1 是否可以访问 P1？若可以，那么 M2 能否访问 P1？这个问题的有趣之处在于，在当前我们对对象系统的设计中，Mx 是能访问 Px 的，因为它们是**属于同一个对象的**性质。但是，如果 P1 只需要被 M1 访问，那么 P1 是否应该仅是 M1 的一个私有变量？例如代码：

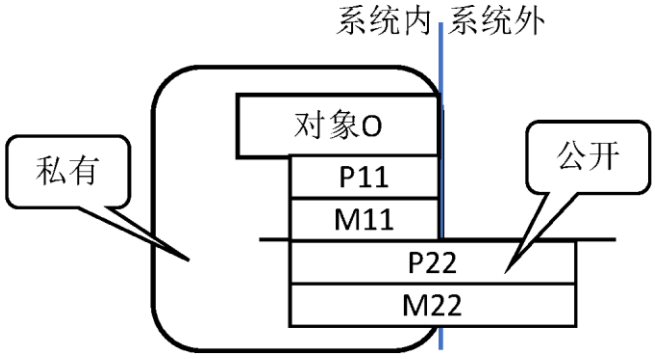
```
aMan = {
  height: 1.75,  //身高
  heighten: function() { ... }, // 增高
  weight: 70, // 体重
  loseweight: function() { ... } // 减肥
};
```

显然，身高与体重都是 `aman` 的性质，但身高与增高有关，与减肥却没什么关系。所以是否应该考虑 `height` 是 `heighten()` 的一个私有的、计算用的数据呢？但如果这样，`height` 难道又不是 `aman` 的性质了吗？

更进一步说，我们的现状是：让任一方法 M1 都必将面对所有的属性 Px。这在一定程度上增加了对象自身构造时的复杂性。

其二，以一个对象作为系统中待处理的数据来说，则对象本身（与其所有属性）就有对系统的可见问题，如图 25 所示。

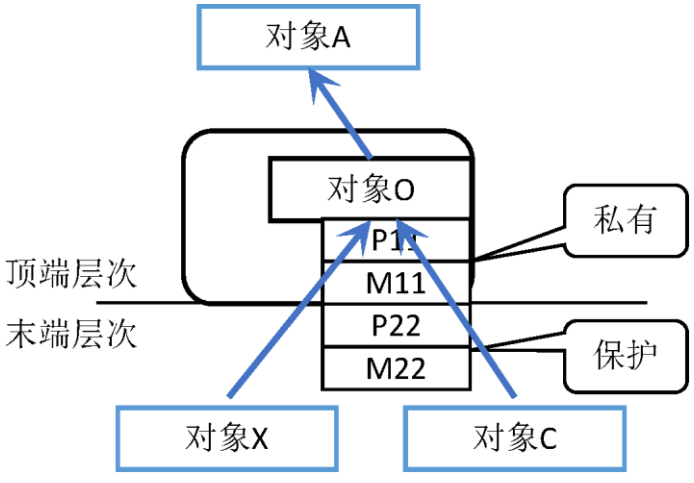
图 25 对象之于系统的可见性问题



因为外部系统可能需要了解 P22 与 M22，而并不需要了解 P11 与 M11，因此 P11 与 M11 就应当对外部不可见（内部私有，internal-private），P22 与 M22 就应该是对外部可见的（公开，public）。再者，与上面的一个问题相关，在 P22 与 M22 中又将涉及到 P22 是否需要被公开的问题。

其三，若一个对象作为继承层次中的一层，则对象有对其他末端层次的可见问题，如图 26 所示。

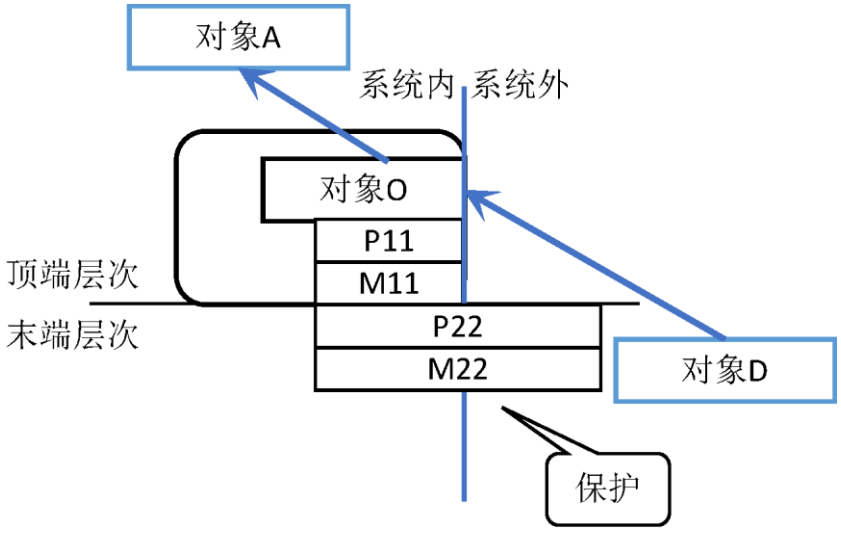
图 26 对象在继承层次上的可见性问题



因为对象 C 的实现可能依赖于一些对象 B 的性质，也可能根本就不依赖某些性质。对于完全不依赖的，就不需要由 B 继承到 C，因此这里的“私有”并不单单表明对象 C 是否可见，而且表明性质 P11/M11 是否要从类继承关系上完全隔离开来，确保对象 C 无法通过继承得到。反之，我们就应该让对象 C 可以继承 P22/M22，以确保其他行为可以访问它们（内部——在继承链上的——保护，internal-protected）。

其四，基于上述问题的进一步设问是：若外部系统中有一个对象是继承自对象 B 而得来的呢？如图 27 所示。

图 27 跨系统继承（对象复用）时的可见性问题



这种情况下，对于 P22/M22，我们仍然需要让外部系统可以在对象 D 中访问到（外部——在继承链上的——保护，protected）。

回顾上一小节的的讨论：多态性是对象的性质在其继承关系上的**表现**，而本节内容则指出这一**表现**的具体方法（即可见性）包括：

- 可见但限于在内部实现的子类的，即 internal-protected；

- 可见并允许包括外部系统实现的所有子类的，即 protected。

上述分析也表明：如果要在对象继承中，利用子类相对于父类的多态特性，则相关的、在继承链上的性质必然是上述两种可见性的。除此之外，其他的可见性还包括：

- 不可见，即 private；
- 可见但与继承性无关，即 internal 和 public。

10.5 更复杂的对象系统：从 GoF 模式来看“对象及其要素之间的关系”

除了以下三种关系，即：

- 对象之间的关系——继承，
- 对象性质在其继承关系上的关系——多态，
- 对象性质分成属性与方法之后带来的关系——可见性。

之外，对象之间、对象的性质之间、对象的性质与对象之间还存在哪些关系呢？

GoF 模式的提出，在一定程度上是对上述问题的回应。GoF 模式的一种基本分类如图 28 所示。

图 28 GoF 模式的基本分类

		目的		
		创建型	结构型	行为型
范围	类	Factory Method	Adapter(class)	Interpreter Template Method
	对象	Abstract Factory Builder Prototype Singleton	Adapter(object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

“目的”这一视角下的三种分类所描述的正是对象之间的三种关系：

- 创建型：基本关系，继承性的定义与获得；
- 结构型：将对象理解为数据时，除继承关系之外的其他关系，包括组合、分类等；
- 行为型：观察对象的行为能力时，对象关于行为的可能关系。

我们先讨论结构型与行为型，因为它们事实上与“面向对象”没有什么关系。结构型讨论的是从系统的外部来看，一组对象应表现为何种属性集（属性组合）的问题。

例如 Adapter 这个模式，我们先假设一个系统中有 N 个对象，N_1与N_x各个不同，但它们都有一部分相同的性质，这些 $N_1...N_x$ 就适合用 Adapter 模式向外表达这种相似性。它们各自实现 IAdapter_N_1...IAdapter_N_x接口，只要 IAdapter_N 接口一致，那么外部系统总是能访问到$N_1...N_x$的每一个对象的上述相同性质。

又例如该系统中另有 M 个对象，且这些对象原本都是继承自相同的基类，它们因此而具有某些相同的性质，那么这个基类就可以被称为一个 Bridge 类。M_1...M_x是该 IBridge_M 的 x 种实现，这个模式以此掩盖了“因为基类相同而具有相同的性质”这一事实。

通过这一分析可见，Bridge 与 Adapter 模式的最终目的，就是向系统外提交“具有一组相似性质”的**对象**中的一个实例，而系统外并不需要关注其内部的转换、继承或组合等实现方式。

那么，如果我们将上述**对象**换成**数据**，是不是也存有类似的模式呢？答案是肯定的。向系统外暴露**一组数据性质**并不是一件复杂的事情。以下述数据（一个数组）为例：

```
data = [["a1","b1"],["a2","b2"],["a3","b3"],["a4","b4"]];
```

设我们为 `data` 绑定了一组行为，分别用于存取 a1...a4 属性：

```
get_ax = function(data, x) { return data[x][1] }
```

那么我们显然可以向系统外暴露一个对象，称之为 Adapter：

```
adapter = {
  get_a1: ...
  get_a2: ...
  get_a3: ...
  get_a4: ...
}
```

该 Adapter 具有 `data` 数组所表达的所有性质，但 Adapter 自身既可以不是对象（例如作为结构体），也与继承无关。

参照如上的分析，可以得出表 7 和表 8。

表 7 GOF 模式：对结构型的分析

考查目标	内部关系	对外表现	结构型（一组性质）
一些对象	具有部分相同性质（P1...Pn）	某个可运算的接口/对象类型，包含左述的P1...Pn性质	桥接与适配器模式(Bridge&Adapter)
一些不同对象	分别具有一系列外部所需性质的部分	尝试复合多个分属不同对象的性质，使之表现为某个可运算的接口/对象类型	复合模式(Composite)
一个对象	具有一部分外部所需性质的部分	可运算的接口/对象类型/对象类型*	装饰者模式(Decorator)

（续表）

考查目标	内部关系	对外表现	结构型（一组性质）
一些不同对象	所有对象中都只有部分性质对外有意义，所有有意义性质被视为一个整体对外表现	说明这些性质的外在表现**	外观模式(Façade)
一个对象	在同一时刻，该对象对外仅有部分性质有意义	仅说明这些有意义的性质	轻量模式(Flyweight)
一个对象	在同一时刻，外部系统需要了解到该对象的全部性质	性质集的一个可获得副本（PP1...Pn）	代理模式(Proxy)

* （关于 Decorator 模式）尝试以一个对象为基础，增加或隐藏性质，使之表现为这样的接口或对象类型。

** （关于 Facade 模式）不依赖于这些对象的个体，因此可以置换对象中的部分或全部，而不改变其上述表现。

表 8 GOF 模式：对行为型的分析

考查目标	内部关系	对外表现	行为型（一类行为）
一些对象A，调用者对象B	所有a都实现visit()行为，该行为返回是否允许调用者B使用对象a中的其他行为	A可以向访问者报告它能完成的处理	访问者模式(Visitor)
一些对象A，外部对象B	所有a都实现handleRequest()行为，该行为被调用时，a尝试处理来自某个B的请求，或不处理	A（可能）可以处理一个已知数据对象	责任链模式(Chain of Responsibility)
一个对象I，一些处理对象A，外部对象B	对象I接收来自B的一系列请求；对象I确知所有a能处理的行为；对象I将请求做预处理并交由a完成具体行为	A可以按指令、指令的序列完成操作	指令模式（Command）

（续表）

考查目标	内部关系	对外表现	行为型（一类行为）
一个对象I，一些处理对象A，外部对象B	B可以向I对象发出指令格式的文本；具有处理该格式文本的能力；I将文本处理为一组逻辑并加以执行	A可以按语法执行逻辑	翻译器模式（Interpreter）
一个对象I，一些对象A	通过I找到所有a	A可以被列举	迭代模式（Iterator）
一个对象I，一些对象A	I表明所有a之间的联动性的行为	A的行为间存在逻辑关系	中介者模式（Mediator）
一个对象I，一个对象a，一些对象A	I知道a的历史A是同一个对象a的不同历史状态（映像）	a可以退回到历史状态	备忘录模式（Memento）

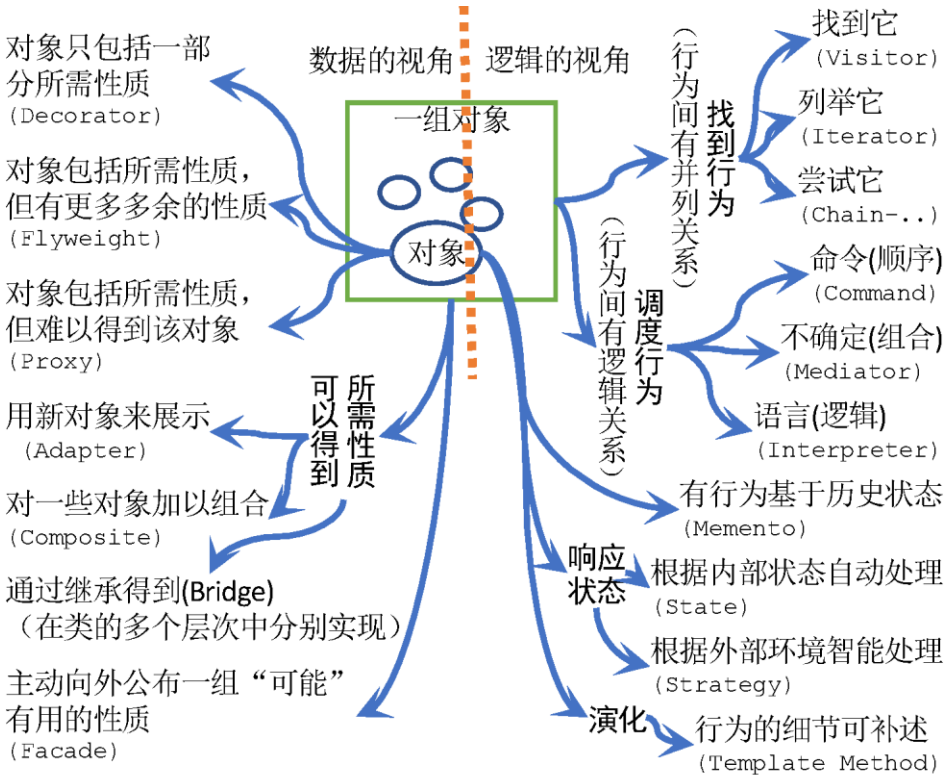
（续表）

考查目标	内部关系	对外表现	行为型 （一类行为）
一个对象I, 一些对象A	I知道所有a的变化	A的变化需要被外部所知	观察者模式 (Observer)
一个对象a	a的行为依赖于其属性变化	a是自动的。a的属性也可能是自变的。	状态模式 (State)
一个对象I, 一些对象A	I接收外部环境的变化，并通知A*	a随外部环境变化(其行为)	策略模式 (Strategy)
一个对象a, 以及对象a的行为a.m	a会按确定逻辑行使a.m；且a.m可以被重定义	a是可补述的	样板方法 (Tpl. Method)

* （关于 Strategy 模式）a 以及 a 的行为是可追加的，并能根据外部环境决定如何行使行为。

图 29 是对上述分析的更进一步归纳。它清晰地阐述了一个事实，即所谓模式的结构型与行为型，其实是从数据与逻辑的视角观察一组对象或一个对象的结果；而所谓模式，是对上述观察所见关系的一种抽象描述。

图 29 模式是对视察所见关系的一种抽象描述



最后，我们也可以更进一步地观察到：（除了 Bridge 模式之外）“对象”与一个一般含义上的、具有一组性质的数据并没有不同。因为我们在图中所说“所需性质”其实就是“具有一组性质的‘名/值’列表”，这既可以是结构体，也可以是（非继承关系下的）对象。亦即是说，这些模式是否要基于“面向对象系统”去实现是无关紧要的。

10.6 再论继承性

接下来我们再回过头来，深入地讨论一下对象系统中的继承性问题。

首先，继承是一种抽象方法，即对事物间延伸关系的一种认识。同

时它也是一种手段，使得对象的一个性质能够“遗传”给子类对象，从而使子类对象可以具有该性质，而无需特别地、显式地说明。

这涉及对象、类与子类的定义与关系。《结构程序设计》将“类”定义为“产生对象（实例）”的过程，这的确带有历史痕迹。除开这一点，这一定义意味着两个事实：

1. 类，必须了解对象是什么样的，即类必须知道对象所有性质的定义；
2. 对象，必然是由一个**构造过程**产生的，这个过程必须知道类，或等同于类本身。

基于这两点，如何来实现继承呢？总结如今在面向对象开发上的实践，所谓继承，有三种实现方式：原型继承、类继承、元类继承。

原型继承是对上述第二项的实践。这种继承方式认为，子类对象对父类对象的相似性可以藉由“抄写（复制）”而得到。这一抄写的过程，即是构造过程。因此，如果一个构造过程 `F` 产生 `b` 对象，则 `F` 只需要知道 `b` 的父类对象 `A` 即可，并不需要通过一个独立的**类**（例如类 B）来获得这一认识。下面的代码反映了这一实现原型继承的基本模式：

```
// JavaScript Syntax
F.prototype = A;
b = F();
```

类继承则显式地要求一个“**类**”来记录对象间的继承关系。类可以是

- A：构造过程（**逻辑**），或
- B：记录上述关系的**数据**，或
- C：仅仅是在源代码期间可知的一种文本**定义**（声明性文本，由翻译程序处理）。

最后一种情况 C 是基本形式。即**类**作为某种**面向对象语言**的一个特性/机制时，仅有翻译程序需要知道代码中**对象之间**的继承关系，这一关系可以用声明性的文本记录在代码中，而完全不需要成为可执行机器代码（即翻译程序处理的结果）的一部分——因为，事实上我们说过，机器代码根本不知道何谓“对象”，更无所谓的“继承”关系[①]。

当这种关系需要为程序员所知——例如试图在代码中知道和处理继承关系，那么它就必须被表达为一种**数据**，即**类**，亦即是上述的情况 B；更进一步，该种数据也必然因其具有一种系列关系而具有数据类型，即**类类型**。可见，类类型本身只描述、记录对象的继承关系；而**类**，根据我们此前的定义，它可以是构造过程本身，也可以仅仅是构造过程执行时了解继承关系所需的一个参考[②]。

那么，从这里也可以了解到，事实上如果构造过程本身记录了上述继承性关系，又可以向外公布这种继承性关系（以便程序员得知），那么**类**也可以是一个过程，即上述的情况 A。这种情况下，类类型也可以是该过程的一个引用[③]。

下面的形式代码（基于 JavaScript 和 Pascal）反映了 A、B 两种实

① 例如不支持 RTTI（RunTime Type Information, 运行期类型信息）、编译性的语言。

② 例如支持 RTTI 的语言。由于在运行期了解类的关系，因此类必须在 RTTI 中登记为一个类型，即类类型。

③ 例如在函数式语言中实现类继承方式的面向对象系统。这时，类信息应该记录在一个构造过程（函数）中，并通过该过程向外公布。

现类继承的基本模式[①]。

```
// B：类是数据，而非执行体
type // <-- 翻译程序可以通过 type 定义来注册 rtti
  TMyObject = { ... };
  TMyObjectClassType = class of TMyObject;
…
Obj = TMyObject.Create();

// A：类是代码，即构造过程
Obj = MyObject();
rtti = MyObject.typeinfo;  // 或 MyObject('typeinfo');
```

现在，我们再一次考察支持**类继承**的系统，便会意识到一个问题：这样的对象系统中既有对象，也有类。既然对象是通过类来得到的，那么“类”又是从哪里来的呢？也就是说，对于最终执行的程序来说，如何了解一个类自何诞生？

而这也就是**元类继承**所要解释的问题了。既然

- 我们可以将**所有数据**视为对象，又，
- 上述的**类**也是一种数据，那么，
- 上述的**类**自然也就是一种对象，因此，
- 也就可以有类的**构造过程**。

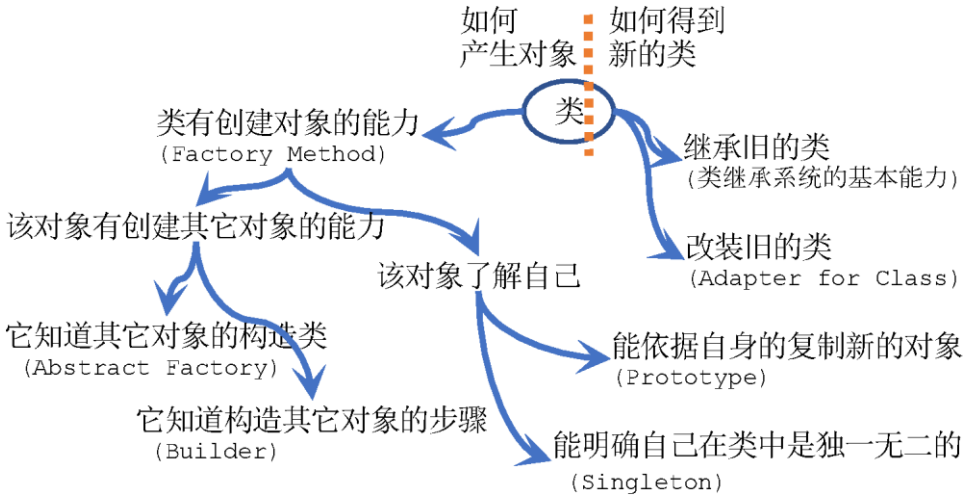
这种构造过程，就被称为**元类**，它是在从对象系统——这一数据定义——中分别出**类**之后，所必需的一种解释。而本章中有关**类**的一切解释也可以同样作用于**元类**，例如元类也可以有上述的三种记录

[①] C 模型是 Delphi 的基本模型，并且由 C 至 B，是 Delphi 在抽象概念上从 Object 到 Component 进化的基本动力——亦即是说，“组件与组件库”是 Delphi 基于 RTTI 的成功实现。在作者的开源项目 Qomo 中，使用了 A 模型来实现“基于原型继承的类继承”，因为 JavaScript 的原型继承本身就实现了“构造过程”这一特性。

对象间的继承关系的方式。

（在上一小节所述之外的）其他六种 GoF 模式讨论了对上述“构造过程”的实现[①] [②]，但并没有强调继承关系通过何种方式来维护——事实上 GoF 隐含地基于并依赖“类继承”模型。这些实现方式的关系如图 30 所示。

图 30 GoF 隐含地基于并依赖“类继承”模型



10.7 数据是一种抽象，所以我们可以泛化从这种抽象中得到的结论

在上述对对象系统的讨论中，我们首先考察的是剥离掉继承性之后的对象——亦即一组性质，或称之为一个复合的数据（或结构体）。对于这样的数据，GoF 模式从“数据间关系”的角度上，为每个子系

[①] GoF 对元类的叙述是没有的，但这不妨碍用 GoF 模式在语言去中实现元类继承。

[②] Adapter for Class、Interpreter 和 Template Method 的“范围”为“类”，表明它们可以是类上的行为，而并非是指它们“产生类”。

统（一组数据）定义了两类可能的产出，其一为结构型，即一组有关系的**数据**；其二为行为型，即一组有关系的**逻辑**。上述所谓**产出**的方式，既可以是指执行该子系统（而得到运算结果），也可以是指对该子系统重新结构（从而最终达到某种外观表现）。因此，GoF 模式本质上是说明：

> $s(f) + s(d)$

即**“结构（逻辑+数据）”**这样的基本模型作用于**数据间关系**之后的产出。

这样看来，尽管我们讨论 GoF 模式时是面向（或基于）对象系统的，但其中的绝大多数模式与“对象”并没有必然关系。同样，即使我们将这些模式的应用泛化到纯粹的“数据的获取、展示与调度”这一层面，也仍然只是它——作为思维方式——的一种应用。

我们可以（也是可选地）将这一思维方式提升到系统层面，那么我们会发现，GoF 模式也可以是一种系统组织方式。这种情况下，系统中的工件并非是数据，而是各种应用与应用间的消息。其中，应用可以理解为 $s(f)$，应用间的消息则可以理解为 $s(d)$。

以此为起点，我们事实上是在将 GoF 思想延伸到系统设计的各个领域。例如著名的系统设计模式 MVC 以及常用的插件框架，就可以视作几种 GoF 模式应用于某种、某类数据以及某个领域的结果[①]。

[①] 参考《程序员修炼之道——从小工到专家》之“29 它只是视图”中的“超越 GUI”一节。

篇四：应用开发基础

> 程序与算数之间的关系，早就有人描述过了。尼古拉斯•沃斯（Niklaus Wirth）说“算法+数据结构=程序”，并把这一论断用作书名，终成名句。其实这与《算数书》这一书名是同一个意思。古人与今人在类似事物上的、最接近本质的理解，其实是一样的。
>
> 从形式上说，特定计算机系统相关的“程序”是略有不同的——它们是为一个可计算系统编制特定序列的编码。这些编码的本意是可以控制机器的指令，只是为了让程序员和计算系统可以存在一致的理解，进而便于将程序员思维映射为计算系统的行为，我们才让这些编码变成了可供人们阅读的代码文本。

然而无论从算数的必要性，还是从计算机系统的必要性来看，我们都无法解释某些代码文本出现的原因，例如Module。因为类似这样的一些抽象概念，既非可参与计算的单元，也非计算机系统可以理解的指令。

那么它们为什么会出现？对这一问题的反思揭示了应用开发思想的出处。

所谓应用开发，并不首要关注对现实系统的抽象或基于该抽象的可计算性的讨论，而是面向“一个应用”整体的系统化思考。因而其思想的核心，便在于如何更加有效地解构和再组织这一系统。最终，我们将这一系统化的产出称为软件，或更进一步地与它的市场行为联系起来，称之为软件产品。

第 11 章 应用开发的背景与成因

11.1 问题的根源：非功能需求与非当前需求

任何一个所谓的“应用”，首先必然是一个或一组“程序”。这意味着它总是能用此前讨论的技术来完成“程序的功能”，例如，我们总是可以将一个现实的问题抽象为对象系统，并面向该对象系统来实现程序逻辑。这一过程在我们此前的讨论中已经一再复述。

但是一旦我们开始讨论应用本身的问题，则必然涉及它的两种内在驱动力量：

- 所面向的是泛计算领域中的某个或某一类用户；
- 所处理的是某一个独立领域中的单一问题或特定范围的问题集[①]。

由于这里的“用户”是指非专业的计算机操作者，因此他们使用与维护程序的方法决定了两类与“现实的问题抽象”相距甚远的需求：

- 非功能性需求；
- 非当前需求。

这些，就是应用开发所面临的全部问题的背景与焦点。

11.2 语言的内建机制剥离了面向计算机的功能需求

对于一个软件产品，我们需要将功能性需求与非功能性需求分开，但这并不是一件容易的事情。程序员总是按他们的习惯来理解眼前

[①] 这是 DSL（领域特定语言，Domain Specific Languages）的出处。

的事物，例如一个用户身份卡（User Id Card）首先是一个内存块，而不是一个抽象数据类型（ADT，Abstract Data Type）或真实的塑料卡片。那么基于这样的认识，什么才是对这个 `User_IdCard` 的功能性需求，什么又是其非功能性需求呢？

问题在于：`User_IdCard` 具有多个层面的数据信息，并且对于不同的程序员来说，对其不同层面的理解都是正确、可实施、可计算的。缘于此，在不同层面上对其功能性需求的定义也就不同。例如对于偏向操作系统的开发人员，`User_IdCard` 只有一个信息是有用的，即[①]：

```
SizeOf(TUser_IdCard)
```

这个长度值决定了程序如何分配和管理该数据的存储。因此这个开发人员会将他所理解的、以该长度值为核心的操作作为一个相对独立的部分区别出来，这些操作可能是如下一些函数或方法的运用[②]：

```
FreeMem  GetMem  ReallocMem
```

例如，一种可能的情况是：

```
// 身份卡的批量分配
//  - Delphi/Pascal Syntax
function BatchAllocCards(count: Integer): PCards;
begin
  GetMem(Result, count * SizeOf(TUser_IdCard));
end;
```

这个函数与 `User_IdCard` 这种数据是有关的，但又与在最终界面上操作该应用软件的用户（例如户籍管理员张三）毫无关系。

---

① 这里基于 Delphi 语法惯例，用前缀字符“T”来表示数据类型，而 `SizeOf()` 函数用于取数据类型所需存储的大小。

② 该例援引的是 Delphi 中的一些内存管理函数，在不同的语言或平台中可能存在差异。

接下来我们还会面临对这个数据的第二类理解：如果这个数据是一系列性质的集合（例如结构体或对象），则每一个性质将对应于现实系统的数据。简单地说，例如：

```
User_IdCard.Age
```

其性质 `Age` 代表了现实系统中某个用户的年龄。与此相类似，`User_IdCard` 的每个性质都有其确定意义，并有相应的行为。这些行为与 `User_IdCard` 有关，但仍然是与张三无关。例如：

```
User_IdCard.setAge()  User_IdCard.getAge()
```

到现在为止，随着计算机应用软件开发技术的发展，我们已经将上述两类对 `User_IdCard` 的理解放在一个语言的基础部分去实现了。大多数情况下，（高级的）计算编程语言通过抽象数据类型及其管理技术（例如对象、读写器、数据验证器[①]以及垃圾回收机制等）来将这些问题隔离在应用程序开发者的视野之外。

忽略了类似上述的问题之后，我们才触到了应用开发的边缘。[②]

11.3 寻找第一个有意义的功能，是探索用户需求的起点

对于“张三”这个户籍管理员来说，第一个真正有意义的功能是什么呢？答案是：**查看身份档案**。

应用开发必须站在用户视角来看问题。张三的日常工作之一是查看身份档案，这也就是他的功能需求。这项需求可能需要分成三个实

[①] 这里的验证器指的是类似 JAVA 中的通过注解来进行数据验证的技术。

[②] “如何屏蔽底层细节”也是应用开发技术的一部分，但对大多数应用开发语言/环境来说，这些都是内建机制。

现过程：

- 过程一：列举身份档案
- 过程二：调出身份档案
- 过程三：显示身份档案

我们需要明确，张三所需功能其实是第三项，即显示身份档案。这可以实现为在用户操作环境中的图形化界面，或是可以进行交互操作的文字框等。而过程一，其实是一个附带的需求，因为从**用户操作流程**上来讲，张三可能[①]需要先看到一个列表，然后才能选择到某个 `User_IdCard`。

过程二则是一个从计算机角度来看待问题的结果。也就是说，对于计算机的数据系统来说，“显示身份档案”是“调出身份档案”之后的一个后续行为。从计算机的角度来看问题，对于张三有意义的功能是第二项；但从张三的角度来说，他只关心第三项功能。

在实际开发中，我们讨论的将是类似如下的代码[②]：

```
// - Delphi/Pascal Syntax


(**
 * 列举身份档案
 * 功能：列举一批档案，返回档案列表
 *   - ListCard()可能是一个数据库或内存中的数据结构相关的函数
 *   - TIdCards 可能是一个与界面无关的数据结构，因此在 ListCard 之后通常有相应的界面
```

① 这只是“可能”，因为张三也可以有许多不同的方式来接触到这个 `User_IdCard`，并最终显示它（亦即是过程三）。这涉及到种种不同的人机交互技术的现实应用，但这些交互行为之任一，都并非是实现过程三所必然的选项。

② 对于现实的户籍管理系统来说，这可能算不得“好的”实现过程，因为它混杂了过程式与面向对象编程风格，并且对于多线程/多界面操作来说存在隐患。

```
过程
 *)
function ListCard(fromId: Integer): TIdCards;
begin
  // …
end;

(**
 * 调出身份档案
 * 功能：使用既有的数据查询/访问接口，从基础数据层获得指定的身份档案
 *)
function LoadCard(Id: Integer): TUser_IdCard;
begin
  // …
end;

(**
 * 显示身份档案
 * 功能：将身份档案显示在输出设备上，可能是手持设备，或 GUI，或控制台。返回用户行为。
 *  - “显示输出（Show）”是一个抽象的功能，对于特定环境下的实现并没有说明。
 *  - “用户行为(Action)”也是抽象定义，但在不同的交互环境中可能是相同的（或仅有实现的不同）。
 *)
var
   CurrentCard : TOperatingCard;

function ShowCard(Id: Integer): TUserAction;
begin
  CurrentCard.User_IdCard := LoadCard(Id);
  // …

  TModalResult(Result) := TShowCardForm.Create(MainForm).ShowModal;
end;
```

在这个示例中，我们严格地将“程序员视角下的过程”与“用户操作视角下的过程”区隔开来：只有当`TShowCardForm.Create()`并`ShowModal()`

发生时[①]，整个程序才是用户交互相关的。这一示例典型地将业务逻辑与用户交互逻辑分离开来，使得观察“什么是用户需求”成为显而易见的事情。

对应于过程一和过程二所述的需求，就其功能的实现来看，其实包括了在三个层次（机器数据层、基础数据层和应用数据层）上对数据的所有操作。其中：

- `TIdCards` 等类型的设计，以及 Card 作为可操作数据的内存分配与管理等，这些是与计算环境相关，针对**机器数据层**的设计；
- Cards 基于有序编号（fromId）、Cards 是否存储在数据库或本地结构化文件中，以及针对上述存储的存取过程等，这些与系统相关的、内在的设定构成了**基础数据层**；
- `CurrentCard` 作为单例的存在，`TUserAction` 以及它与 `TModalResult` 之间的关系等，这些是**应用数据层**的设定——基本上来说，将应用数据层整体去掉之后，基础数据层仍然足以支撑其他的用户需求与业务。

注意一些高级语言在应用开发中刻意地屏蔽了这里提出的三个数据层次中的多数细节 。例如在 Java 中可以直接使用对象来表达 IdCard，并用内置的 ObjectList 来实现 IdCards 列表，因此开发人员并不需要接触到机器数据层和基础数据层，例如不必了解对象结构，以及它们在 32 位或 64 位机器上的不同实现。类似的，在多层设计中，也倾向用 O/R Mapping 技术来屏蔽后端数据库的细节，因为它们同样处于基础数据层或机器数据层。这一切的主要目标，就是让“面向应用开发的程序员”将重心放在应用数据层的设计之上，并基于那些相对确定的、稳定的，以及更远离用户的数据层次以及

[①] 这里使用了 Delphi 的一些技巧：（1）`TShowCardForm()`在关闭时是可以自动释放的；（2）窗体可以向调用过程返回一个值以表明用户所做的界面操作；（3）使用 `ShowModal()`方式打开的窗体能阻止该应用中其他窗体的操作，即这种情况下的用户界面是独占的。

解决方案来完成开发过程。

然而这些对于一个具体的操作人员（例如张三）来说，没有什么意义。最后，对于功能三，如果不考虑显示的具体效果的话，它只是在应用数据层上一个“界面交互”的实作而已。

但正是在这一点上，我们发现“用户需求”完整地影响了过程三的设计。

11.4 界面交互是功能性需求，但开发技术并非它的主要构成

界面交互是否算作功能性需求？

在程序员看来，**显示身份档案**这个功能实现为一个“信息列表”就足够了，因为一个列表（list）足以显示全部信息[①]。但是该档案也可能被显示在触摸屏、移动终端或者桌面电脑等设备中，因而对于实际的应用环境来说，“如何显示身份档案”是该应用的一个实际功能。这样的功能有可能是用户没有提出但使用环境有此要求的。更可能的情况下，“显示成何种形式”以及“如何操作它”已经包含在用户提交的功能需求当中。更进一步地：

- 用户可能要求同时显示指定个数的档案（列表）；
- 用户可能要求显示档案信息的部分或全部（卡片或详情）；
- 用户可能要求显示档案某一信息的局部或改换某种显示效果（显示头像）；
- 用户可能要求显示档案能被持续更新（实时刷新）；
- 用户可能要求某种或某类角色可以修改某些信息（操作员与主管）；

[①] 想象一下把 xml 文件直接展示在文本编辑器或 WEB 浏览器中。

- 用户可能要求使用某种装备来输入某些信息（OCR 接口）。
- ……

在**程序设计**的阶段，我们可以将“信息列表”作为输出验证的手段，例如使用自动测试工具来验证接口（类似前面提到的 `ListCard()`与 `LoadCard()`函数）中数据返回的正确性。但是对**应用开发**来说，根据用户的实际需求而组织一个合理的展示与交互界面，是我们必须完成的功能。在某些情况下，应用程序并不要求有太独特的界面与交互，例如标准的 Windows 桌面程序就可以直接在 Visual C++中用 MFC 标准组件完成。因此基于操作系统或特定环境下的界面与交互，“就程序员的感觉来说”，更像是一种实现的必需，很难被视为是用户功能性需求的一部分。

即便如此，对于一个真正意义上的应用软件产品来说，我们还是必须将界面与交互作为功能性需求。这种产品特性事实上构成了用户对功能的认识，例如界面是用户操作产品功能的一个不可或缺的部分，因此是功能性的。与之相对，安装过程与操作该产品的功能没有直接关系[①]，因此是非功能性的。

考察这一分歧，我们还必须留意到界面与交互带来的细分领域问题：当它作为操作系统（以及其开发标准）的一部分提供时，它与程序员所在的计算机环境是同一领域的；而当它是构成某个**独立领域的用户认识**的一部分时（例如工业控制界面，或户籍管理系统特有的身份证扫描界面），它就是该领域下的一些领域知识。这一观点用

[①] 张三也许并不知道“户籍管理系统”如何安装到他工作的机器上，但这并不妨碍他使用这个系统。

在桌面应用或后台应用开发[①]，以及其他种种软件开发活动中也是适合的。

我需要进一步强调的是，程序员在界面开发中对界面绘制方式（或界面控件）的专业知识，在于“如何实现它”，而不是“如何让它表现得更符合用户需要”，后者是界面交互设计的职责[②]。举例来说，作为程序员，如何在窗体中正确地层叠多张 PNG 图是你所必须掌握的技术，而其中某一张图是否漂亮美观或符合客户的审美情趣，就不是你必备的程序开发技能[③]。

11.5 产品需求通常是非功能性的，但又是最上层的应用表现

那么什么是非功能性需求呢？我将这一类需求理解为：即使这些需求完全不存在，也并不影响应用的主体功能的交付，那么这类需求就称为非功能性需求。例如安装、手册、升级维护工具等。[④]在我们的应用开发中，有大量的工作与技能都是与这些非功能性需求相关的。然而它们并非是不必要的，相反，掌握这些技能是成长为应用开发工程师所必需的。只是这些内容的思维方式与学习方法，与我

① 可以在 PC 机、移动设备以及其他的各种环境中找到“桌面”与“后台”的不同划分。一种不太严格的划分方式是：后台程序总是以控制台或类似方式与操作者交互，而桌面程序则采用种种“友好的”交互方式。

② 这往往被归入产品的设计特性。Marty Cagan（网景副总裁、eBay 产品管理及设计高级副总裁）甚至将设计与功能区分开来，将设计特性作为产品属性而非具体的功能属性。参见《程序员》2011 年之“Marty Cagan 谈产品系列”。

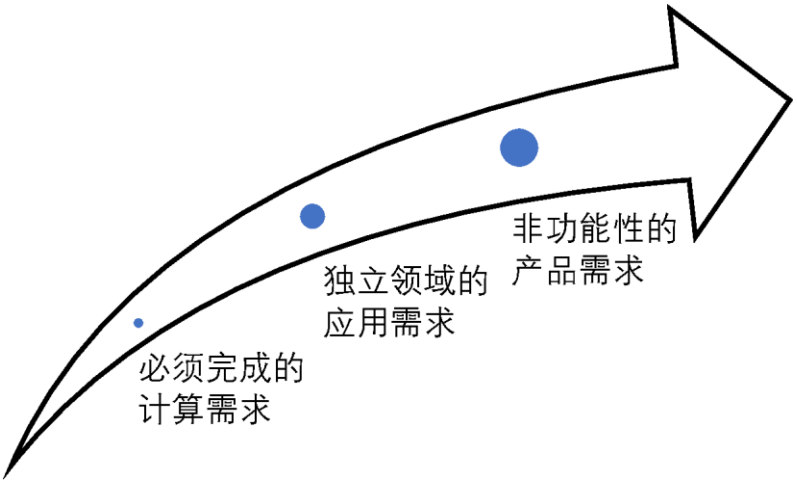
③ 你当然可以具有这样的技能，一专多能或者无所不能并非坏事，但在这里我们只讨论属于程序员的那个部分。

④ 有一类软件叫“绿色软件”，它有一个特定的名称叫“Portable Software”，通常是某个应用产品的定制版。比较绿色软件与相应产品的发布包，其中被删减掉的部分基本上就可以称为非功能性需求。

们此前所讨论的“（严格意义上的）程序”完全不同。

综合上述讨论，应用开发中存在三个层次上的需求，如图 31 所示。

图 31 应用开发中三个层次上的需求



其中，必须完成的**计算需求**是通过系统分析，在机器数据层、基础数据层和应用数据层上所得出的功能性需求。独立领域的**应用需求**，是通过业务分析，在用户的业务领域中所得出的功能性需求。而第三个方面，**产品需求**则通常是非功能性的[①]。最后需要强调，有一部分非功能性需求可能同时也是非当前需求，例如版本管理[②]。

① 这些狭义的、刻意与其他类型的需求区别开来的产品需求当然也表现为具体的功能，这里的“非功能性”是针对目标用户而言。例如产品互联网产品中存在如何运营、营利的问题，这类运营需求往往不是目标用户所关心的，而是互联网企业对于该产品的需求。

② 在这一视角下，“版本”这个概念是相当混杂的，它包括功能性需求、非功能性的需求，以及产品和产品线的需求。在后面我们会再次谈到：在“应用开发”这一领域中所言的版本，仅是这个概念中的一部分。

第 12 章 应用开发技术

12.1 应用开发语言同时存有两个发展方向：软件复用和工程化方法

对于应用开发中的功能性需求（计算需求与应用需求）来说，一切空间因素所致的复杂性，都可以通过组织形式来解决；一切时间因素所致的复杂性，都可以通过抽象模型来解决。[①]

第一类情况更倾向于工程师思维，在工程师看来，1+1 永远都等于 2，因此系统总是可以通过部件的持续增加来得到最终的结果。第二类情况则更倾向于管理者思维，在管理者看来，是否需要 2 就是一个问题。因此他们倾向于先完成 1，再讨论 2 的问题。于是他们只要求以某种形式或方法证明：系统具有可以演化至 2 的“可能性”。

因此在应用开发语言中也同时有着两个发展方向：一个是从“模块/单元”这一角度出发的软件复用，另一个是从“项目/工程”这一角度出发的工程组织。

12.2 模块划分永远不存在最优方案

化整为零是一种简单却不易行的软件开发策略。其关键在于，对于任何一个目标集来说，拆分的可能方案都趋向于无穷。因此我们为论证某一个拆分方案的正确性而付出的代价，往往要高于实施目标集本身。换言之，要讨论如何拆分是“最”好的，有时甚至不如不

[①] 这里的空间与时间因素，是基于在《我的架构思想：基本模型、理论与原则》中对目标属性的分类方式来讨论的：所谓空间需求，是指它们可以通过组成部件的增减来解决；所谓时间需求，是指它们可以划分为多个时间阶段来实施。

拆分。

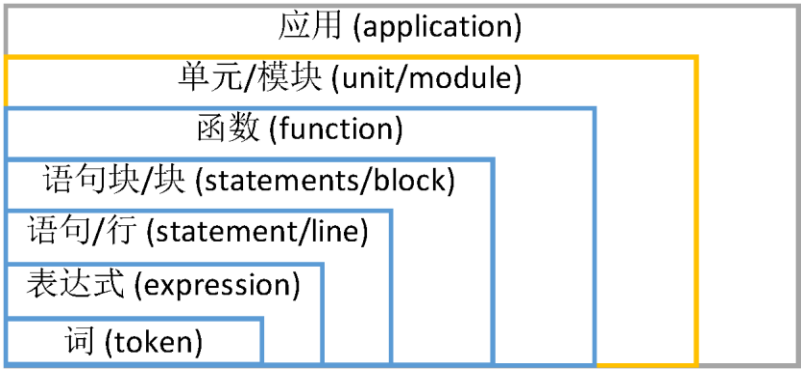
所以事实上对于任何一个项目中的类、单元、子程序或子项目的抽取，我们在现在以及将来实施的都绝不是“最优解”，而是在所处阶段中相对“较可行”的一个方案而已。程序员在这一过程中往往是一个被动的实施者，而非这个“解”正确性如何的评估者。

总有一些规则是“较好的”或“较容易判断与实施的”，但如上所述，它们可能不是“最好的”与“最正确的”。了解这些规则，是渐进提高编程素质的方法之一，但并非某个具体项目的唯一判断标准。

12.3 模块化的精髓不在于外在形式的分离，而在于内在逻辑的延续

图 32 展示了在稍早一些的应用开发语言中，从“代码的粒度”出发的抽象概念。

图 32 从“代码的粒度”出发的抽象概念[①②]



其中，单元或模块用于组织一系列函数，而一个应用[③]则是由单元或模块构成。在这样的体系中，“化整为零”的问题会变得相对简单，即如何有规则或有逻辑地将一堆函数组织成单元。这里的“规则”与“逻辑”阐述了组织法则的两个方向。

其一，我们可以设定一个简单的分类依据，使得位于同一个单元中的函数表现出一定的相似性。例如开发一个图形库，我们可以将与图形设备相关的功能放在 device 库中，将绘制功能放在 graph 库中，将渲染功能放在 render 库中，将与图形库无关但又与计算机基础环境相关的功能放在 base 库中，如此等等。最后，我们将一些杂乱无章的功能放在 misc 库中。请注意，这一切的分类依据是“功能的归属与使用者”。类似地，我们也可以依据数据的位置来建立分类依

① 语句与行的不同，通常也被称为逻辑行与物理行概念上的不同。此外，有些语言是强制要求以物理行来表达“语句”这一概念的，即一行语句必须书写于一行代码中。

② 单元与模块除了称谓的不同，很多时候其抽象概念也并不完全相同或者互相覆盖，例如一个单元可以是（或不是）一个模块。我们这里只取在某些语言中、将模块特指为“一系列函数”的这一概念。

③ 早期的应用开发语言也直接将应用称为“程序”（program）。

据。例如同样是这个图形库，我们可以将基础数据运算放在 bits 库中，并基于此建立关于图形运算的类型抽象库 types。接下来我们定义在不同设备上适用的数据结构，比如在存储设备中的种种文件格式 fileTypes、在内存中复制和运算的 dibs（设备无关位图，Device-Independent Bitmap）以及在某种具体显示设备中适用的 cudaTypes（CUDA，Compute Unified Device Architecture）等。这样依据数据（所处的）位置以及需要进行的计算进行分类，也便于将数据及其副本放在不同的环境下开发。

这一类的方案或试图交付一个可以被使用甚至被共用的功能集，或通过抽取不同层次（例如面向不同设备或不同场景）的代码，使之可以“或多或少”应用于不同的环境。与这个组织原则密不可分的一个问题是：如何使一个“单元/模块”向外公布它所具有的功能集。这形成了著名的“开放细节”与“公开功能但隐藏细节”之争[①]，如今后者已成为应用接口设计思想的主流，前者则部分地影响并推进了开放源代码这一思想。

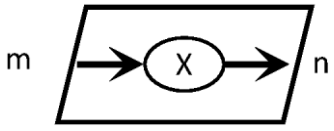
但总的来说，这个组织法则只解决了一个应用中能被静态规则化的部分。无论如何，它无法满足“让程序运行起来”之后可能带来的种种变化。

其二，我们可以使得一个单元或多个单元中的函数存有某种逻辑关系。著名的“自顶向下程序设计”的思想，就处于这一组织法则所代表的方向上。例如：

[①] 参见《人月神话》中“关于信息隐藏，Parnas 是正确的，我是错误的”小节，以及 David Parnas 关于信息隐蔽理论的著名论文：《论将系统分解为模块的准则》、《设计易于扩展和收缩的软件》和《复杂系统的模块化架构》。

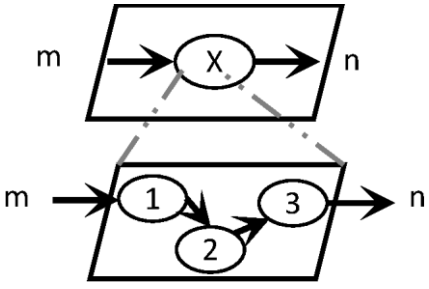
设有一个逻辑（X），功能是将 m 变换为 n，如图 33 所示；

图 33 基本功能：将 m 变换为 n



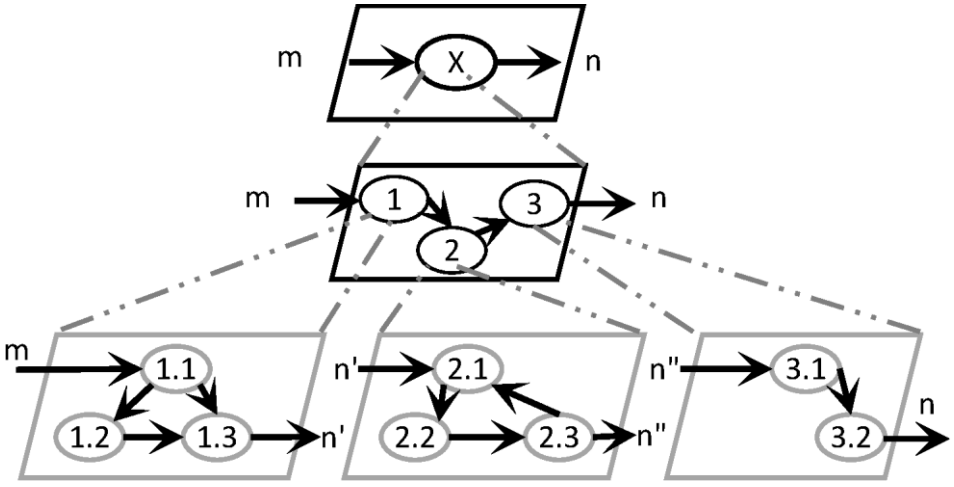
由于 X 的规模巨大，我们将它分成三个逻辑步骤（顺序逻辑 1→2→3）来实现，如图 34 所示。

图 34 分解：三个步骤



虽然向下一层的分解并不限定各步骤之间的关系，但我们注意到此前讨论过的一个事实，即所有逻辑都可以被理解为顺序逻辑中的一个步骤。也就是说，步骤 2 依赖于步骤 1，步骤 3 依赖于步骤 2。随后我们继续向下一层分解，如图 35 所示。

图 35 持续分解：更多的步骤、逻辑或子系统



在图 35 中：

- 步骤 1 的子步骤 1.1 分成了 1.2 和 1.3 两个分支，最终以 1.3 为出口，传出数据 n'；
- 步骤 2 则被分解为三个子步骤的循环，并总是以步骤 2.3 为出口，传出数据 n″；
- 步骤 3 分解为两个顺序的子步骤，得到转换结果 n。

由此无论是针对顶层 X 这个逻辑，还是针对第二、三层各分解的逻辑，整体的逻辑关系都没有变化。所有的逻辑关系在函数与函数之间，以及在“一堆函数”与“另一堆函数”之间都可以被简单地抽象为“顺序依赖”。即使将这些单元/模块之间的关系映射到最终子系统的划分之上，这种逻辑关系也不会变化。例如从子系统划分上看：

- 子系统 1 被设计为“预处理器”（preProcessor）；
- 子系统 2 被设计为“分析器”（analyzer）或“过滤器”（filter）；
- 子系统 3 被设计为（下一阶段的）“数据供应器”（dataProvider）。

这三个子系统以及它们的组织关系就可以构成某种数据处理系统的、整体的、面向运行期的逻辑架构。

进一步地说，通常单元/模块之间的逻辑关系只是简单的依赖关系。这一关系足以支撑由结构化程序设计带来的计算需求，包括支持数据流转与逻辑执行。[①]

12.4 “没有坏味道”的诀窍：如何更好地组织代码

将数据或逻辑具有类似性质的代码放在一起，或者将逻辑之间存有关系的代码放在一起，这两种思路与面向对象用封装性来解决的问题是相类似的。一个对象，其**属性**是一系列（具有同类抽象性质的）相关数据，其**方法**是一系列与上述**属性**相关或相互间存有依赖的逻辑；对象的封装性决定了对象对外或对某个范围公布的接口。因此，一个对象或这个对象的类，其实有着与“单元”（unit）相同的抽象意义。

所以当面向对象出现之后，“一个单元中应该放多少个类”成了一个问题：如果一个单元可以放多个类，那么它与“库（library）[②]是用来容纳多个类的组织单元”这一抽象概念又重叠了。因此在早期面向对象语言的设计中，对这个问题的解释是含糊不清的，例如 Pascal/Delphi 允许在一个单元中放任何多个类，这导致“单元的组

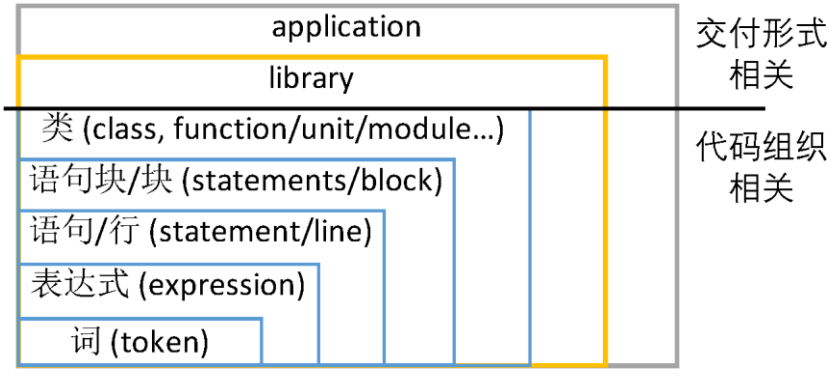
[①] DFD（DataFlow Diagram，数据流图）通常用于解释在上述执行过程中 m-n 之间的转换关系不变（即数据单一入口与单一出口）。但在本例中，它亦用于解释自顶向下过程中的逻辑关系不变，整体保持着顺序执行关系。这一过程是抽象概念——从程序语言中逻辑的**结构化**，到应用系统中组织的**模块化**——的延伸。

[②] 这里指的是对象库（object library）或类库（class library），而普遍意义上的“库”是下一小节讨论的重点。

织原则”变得更加简单而含混：如果类之间相关或相似，就放在同一个单元吧。而晚一些的面向对象语言则较好地解释了这个问题：一个类即是一个单元/模块，或干脆进一步地取消了单元/模块概念。在具体表现上，例如 JAVA 或 C#就推荐在一个文件中存放一个（可以公开的）类，这个文件——或包含许多函数与数据的单元——将作为一个独立的组织单位存在。

图 36 说明了在面向对象的设计观念中，“类”其实是用来替代“function/unit/ module…”等组织方式。不过在某些多范式语言中，例如 pascal 或 javascript，通常也允许这两类组织方式同时存在。但从本质上来说，这只是代码组织方式决定了一个“代码集”（source code package）在形式上有所不同，其内部的逻辑、数据以及更为底层的算法观念其实是大同小异的。

图 36“类”的价值与局限：对传统组织方式的一种替代



随着系统规模的扩大，应用产品对“引入第三方代码”的需求也越来越明显。而“类”作为一个组织单位，其实是将逻辑和与其相关的数据、相关的抽象目标集中在一起发布，因此面向对象技术提供了相当高的可复用性。这一点正好迎合了上述需求（当然，换个角度也可以说，是需求推动了面向对象复用技术），因而如何在类的

基础上进行更大规模的代码组织，成了一个重要的问题。

名字空间（命名空间）的出现与对象复用的思想有着密不可分的关系，但究其本质而言，名字空间下是否包含一个“类簇（class cluster）”却并不要紧。因为名字空间本身只是用于隔离不同的软件厂商、产品和子项目之间的代码，以及这些代码对外交付的接口。这种隔离需求原本是来自于交付物的名字冲突（例如 A 公司与 B 公司的代码库中都存在 TDynamicArray 类），而不是缘于这些交付物的类型或结构抽象冲突。所以无论面向对象是否出现，在“函数/单元/函数库”这样的组织单位持续进化之后，必然会由于跨公司、跨领域的复用而出现与“名字空间”相类似的代码组织方式。

常见的名字空间的命名规范为：

```
CompanyName.TechnologyName
```

例如：

```
Microsoft.Office
```

名字空间可以由更复杂的分类规则构成。例如：

```
Microsoft.Office.Tools.Word.Controls
```

通常其具体规则是由不同的公司/产品/产品线来决定的。例如：

```
com.companyName.puc.biz.deduct.data.types
---------------
             公司
                 ---
            产品/系统
                      -----------
        系统层次/工程/项目组织结构
                                ----
                           子系统/模块
                                      ------
```

系统内定义或其他定义

如同所有的代码组织形式一样，名字空间通常也与作用域相关。由此带来的效果，也就是它解决的需求是：A 公司与 B 公司代码库中的 TDynamicArray 类之所以存在“不同”，是因为它们所处的名字空间不同。这一点与用“单元内、单元外”来隔离标识符系统，以及用函数、语句甚至表达式的“作用域”来隔离标识符系统的性质是完全相同的。它们只是组织规模上的差异，而其抽象概念以及目的是一致的，只是自然地随着规模扩张而延伸罢了。

12.5 交付形式相关的组织方式

库，通常是一种代码或其对应产品构件的交付形式。这意味着库的表现形式是多样的，例如可以是源代码的一种组织形式，也可以是由源代码编译成的二进制结果；它既可能是一些单元或模块的、有或没有规则的包装或集合，也可能是将某些“类”有序集中在一起的一个泛指。一些常见的形式包括：

- 在编译语言中，库可能用来组织代码，或翻译阶段中的中间代码。例如汇编语言中的库（.lib），就用于管理一些目标文件（.obj）的集合。
- 在目标系统中，库可能用来指应用发布的一个组成部分。例如 Windows 中的动态链接库（.DLL）。
- 在面向对象系统中，（类）库通常用来指一些类的集合，但并不表明这些类是以源码形式或是二进制形式提供。例如开源项目中的类库可能是源代码包，而.NET 的类库则是一些可以注册到系统中的、二进制的程序集（Assembly）。
- 在应用环境例如操作系统中，库通常是指可在不同应用产品之间复用的运

行期代码。例如 COM 库，或前面提到的动态链接库（.DLL）[①]。

综合上述以及更多的应用环境，“库”通常都不是指代码本身的组织，而是指它们的交付，包括最终交付之前的某种阶段。

“库”究竟以何种形式交付，以及包含何种内容交付，都取决于最终应用产品对未来交付形式的需求。换而言之，既然库是某个产品（在特定运行环境中）的构件，那么它必然满足该环境的要求和该产品的限制。例如：

- 如果产品以平台依赖的二进制共享文件发布，那么库可能以编译阶段的中间文件，或者执行阶段的依赖模块的形式提供，例如 Pascal 的*.tpu 和*.bpl，或者 delphi 的*.bpk，C/C++的*.lib，以及.NET 的程序集、COM 组件等。
- 如果产品设计为支持插件的，那么库将以动态链接库（DLL）的形式交付，例如 Windows 操作系统中的屏保（*.scr）、控制面板应用（*.cpl）、管理模块（*.msc）等，或者 Photoshop 中的特效插件（*.8bf），这些其实都是修改了文件扩展名的动态链接库。
- 如果产品设计为动态资源的，那么库将以资源包的形式交付，例如Android 开发中的*-res.apk 文件，通常就是作为某个主应用程序的资源包来提供的，这样便于提供同一个应用程序的不同国家/地区的版本。
- 如果产品设计为支持动态脚本的，那么库将以源代码包或脚本库的形式交付，例如 Mozilla Firefox 的*.xpi 插件，本身就是一个.zip 文件包，其内容则是一些 javascript 脚本及其依赖的资源。

“库”作为交付物以及最终产品的一个组成部件来提供，也意味着

[①] 动态链接库也有动态加载和静态加载两种形式，前者是用户代码可以使用 loadLibrary()等函数将该库装载到应用环境中，后者是指在编译期由编译器决定的、在应用运行的初始化阶段由操作系统负责装载的库。在 Windows 环境中，一个库是用于静态加载或动态加载，通常是由应用决定的——这也意味着它的发布与依赖也由应用来决定。

它是具有版本信息的[①]。这种版本信息与交付产品的版本相关，例如不能直接将用于 Mozilla Firefox 3 的插件直接应用于 Mozilla Firefox 6，而这一限制是作为版本信息（install.rdf 文件）包含在插件中一起交付的。

然而作为一个完整的交付物，应用产品可能是一个不带有任何“库”的独立程序（例如 Windows 环境中的.exe 文件），也可能带有更为丰富的产品信息，例如包括：

- 产品依赖的操作系统或主程序版本，例如Firefox插件中的install.rdf文件；
- 自身基于的运行库，例如VC的Runtime库vcredist_x86.msi；
- 与此前的发布版本的冲突或依赖，例如配置文件检查或修复；
- 当前产品版本发布时附带的完整库、文件或资源，例如文件清单；
- 当上述依赖缺失时，可能的获取渠道，例如ActiveX组件的codebase；
- 产品文档或相关信息，例如 readme.txt、帮助文件或在线帮助的网址链接。

通常应用程序开发的集成环境（或某些推荐性的套件、开发工具组合）会提供一系列的方式来生成上述的产品内容[②]，最后将这些文件打包并提供某种统一、便捷的安装方式。这些最终可以由“用户”在其机器环境下自主使用的交付物，在商业意义上称为“产品”，

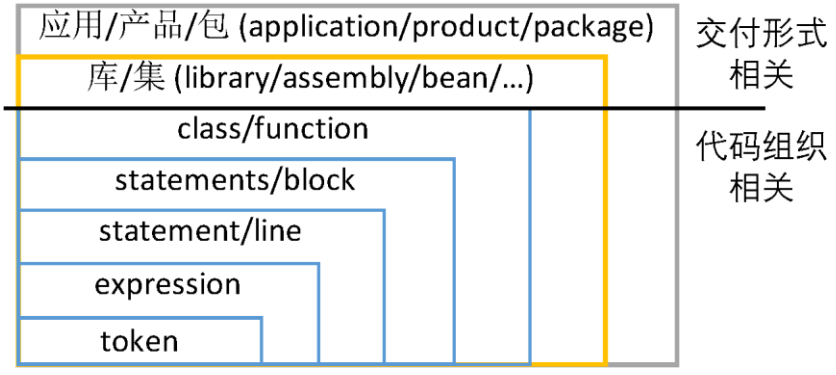
[①] 版本也是导致“DLL 地狱”（DLL Hell）问题的根源之一。所谓“DLL 地狱”，是指在同一执行环境的不同软件产品中，由于使用了同名但版本不同的 DLL 而导致的冲突。这一问题并非 Windows DLL 或 Linux Library 所特有。不同的库，在解决这一问题的策略上不尽相同，但大体上都是以“声明版本依赖关系”为基本思路。

[②] 这些内容可能被称为“库”，也可能被称为“构件”，还可能被称为“依赖项”，如此等等。

而在程序世界中通常就称为“包”[①]。

综上所述，我们将库（library）与包（package）作为产品交付形式相关的两个组织方法如图 37 所示。

图 37 （产品的）交付形式相关的组织方式



[①] 例如对象库或类库的交付物称为“类包”（class package），可执行应用及其完整交付称为“安装包”（install package），而 linux/debian 环境下一个产品或产品系列的发布称为“debian package”。

第 13 章 开发视角下的工程问题

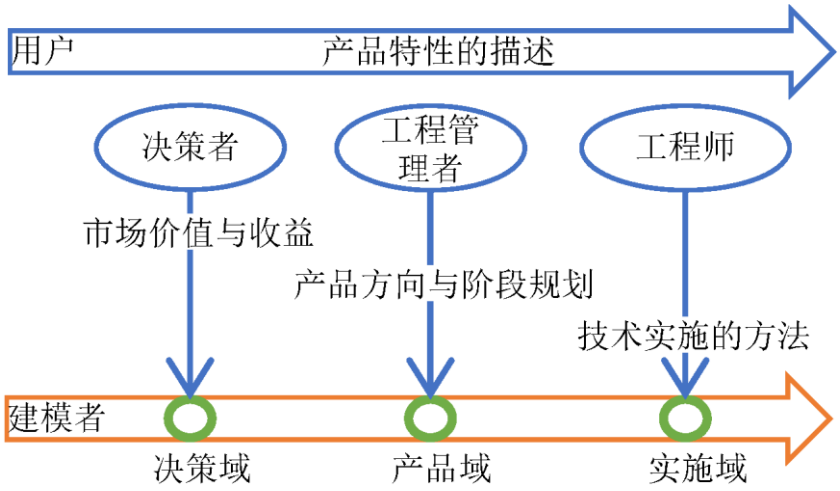
13.1 模型是一种沟通工具，这是它内在的“语言”本质特征

没有软件开发人员会用代码去“写一个模型”，反而是在系统的分析建模完成之后，用代码去描述上述“建模的结果”（即模型）。换言之，编程语言所描述的模型，其实只是建模结果的一个方面，这就如同书面语言、口头语言等都只是语言的一个方面。我们若是仅局限在一个方面去讨论模型本身，便又回到了盲人摸象的困局。

正因为模型描述的是现实目标的一个或多个侧象——这也如同编程语言中的数据并不能描述现实目标的整体一样，所以不同的人从不同的角度观察和理解现实目标时所得到的模型也就不一样。最终在建模过程中所讨论的仅是对系统的一个相对“清晰一些”的理解，所以在建模过程中也往往会有“识别（出某个模型）”这样的说法。

我们不可能也不必要描述系统的全像，这是必然的。因此到底要描述哪些方面，就成了一个实际问题。而这个问题的解，要追溯到问题产生之处。也就是说：谁需要建模？谁做建模？以及作为中介的模型，表达了谁与谁之间的沟通需求？总的来说，一个“建模者”所面临的需求主要来自这几个方面，如图 38 所示。

图 38“建模者”所面临的主要需求



建模者并非一个单一职能的角色，因为整个系统将涉及多个不同领域、不同对话对象的建模。例如，在产品域看来，建模的目的是要保证产品是否实施或者实施过程是否可控，因而需要在这些问题上提供一个“可讨论的对象”；而在实施域看来，同样需要提供一个“可讨论对象”，以保证在用户需求和产品提供之间的一致。

类似这种“可讨论对象”有一个重要的前提，即建模者必须能够提供一种在讨论双方或多方之间理解一致的东西。关于这一点，在敏捷工程与传统工程中存在极大的分歧：传统工程认为应该通过“建模”来得到一个多方共同认识的抽象对象——即模型，并围绕这些模型来推动从**决策**到**实施**的全程；而敏捷工程则认为对于多方来说，最好的、能够无碍理解的东西是产品原型而非抽象模型，因此应该将工程中的多方全部纳入一个基于**产品原型**的、迭代实施的推进过程中，由具体的过程环节来决定沟通的细节。

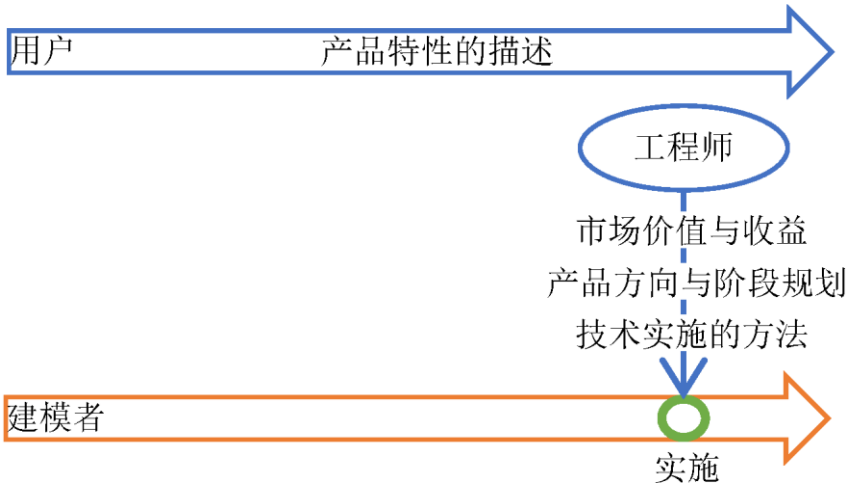
13.2 原型是轻量级的试错，它并没有减少问题的总量，但改变了达到解的方式

但问题是：为什么不能将“原型”也理解为“模型”？为什么非得将画在图纸上、用几何线条描绘的东西才视为模型，并将这些东西的抽象与绘制过程才称为建模呢？建模的目的是得到一个“可讨论对象”，而（多数情况下）原型就是——在敏捷工程所设定的场景中——可以被多方认可并予以讨论的对象。一旦我们承认这一点，那么再讨论“是做原型还是画图纸”的问题就没什么价值了。

从实施推进的角度来看，模型事实上允许我们将系统拆分成多个阶段，并尽早地预期了系统的每个阶段所依赖的（前一个阶段的、可能的）事实基础，因此模型具有可描述、可分析、可预期等性质[①]。而敏捷工程本质上是把决策域与产品域中的需求拉到了实施域中，就地决策与设计（产品），并将这一过程开放给用户，如图 39 所示。

[①] 正因为如此，在“第 12 章 应用开发技术”一开始就讲到：它适于解决时间因素所致的复杂性。

图 39 敏捷工程并不能消灭上述需求



问题的总量并没有减少，但是这里的“可讨论对象”（即原型），变成了纯粹用于工程师与用户之间沟通的桥梁。这符合一些应用开发工程的现实场景，例如用户通常更关心产品特性是否能够得到满足，他们多数时候并不关注市场、产品或实施阶段等问题。因此敏捷（以及类似基于原型、快速迭代的）工程模型有着很强的实用性。

然而原型不能解决一切问题。在一定程度上来说，原型是轻量级的试错，而抽象模型则可以通过严密的论证与分析过程来得到决策依据。因此，原型与模型的适用领域也有所不同。原型适合于在参与者之间建立简单的、直观的、允许快速纠错的讨论对象，更适用于快速推进以及短周期、逼近式的产品开发。而模型则适合于抽象出系统中方向性、支撑性、不易频繁变化的关键环节，在此基础上进行论证，以尽可能减少出错或预期风险，甚至更进一步地提供中长期的系统演进规划（例如产品版本、产品线等）。

13.3 集成化工具需要有配套的生产过程和管理

应用开发的“集成环境”（IDE，例如 Eclipse IDE）不是给一个人用的，开发工具公司的“套件”（Suite，例如 VSTS，Visual Studio Team Suite）是为多种角色提供的。但这两点事实，大多数时候为工程人员所忽视。集成环境或套件都是基于工业生产的理念来提供的，这一理念认为：工业生产整体依赖于分工明确的产品过程。因此，开发人员需要代码文本的编辑环境，测试人员需要自动化测试与脚本驱动的工具，构建人员需要包、构件以及面向资源的管理界面，如此等等。

将这一切组织在一起的 Suite 或 IDE，就如同工业生产线上的硬件环境。正如这个比喻所暗示的：在购买这个生产线的同时，也就意味着你需要这个生产线的过程模型与管理体系。大多数时候，我们的应用软件产品开发并不是工具用得不好，而是生产组织与管理得不好。而这其中，“模型”的指导意义尤其被忽略。

MSF（Microsoft Solution Framework）描述了“VSTS 生产线”背后的工程模型，其核心并非来自于技术实施，而是来自对管理 “项目/产品过程”所需要进行的权衡（图 40）[①]，这与“项目平衡三角（项目管理三要素）”所阐述的问题是一致的（图 41）：

[①] 引自“MSF Process Model v. 3.1”,Microsoft Solutions Framework White Paper。

图 40 项目要素之间的平衡三角

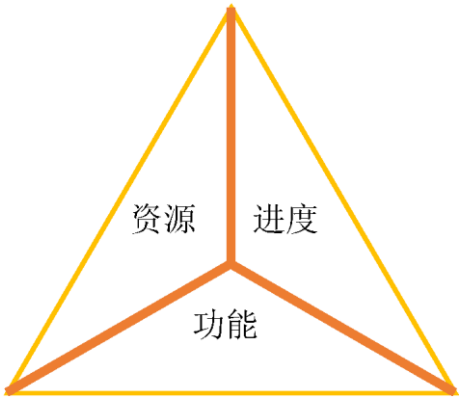
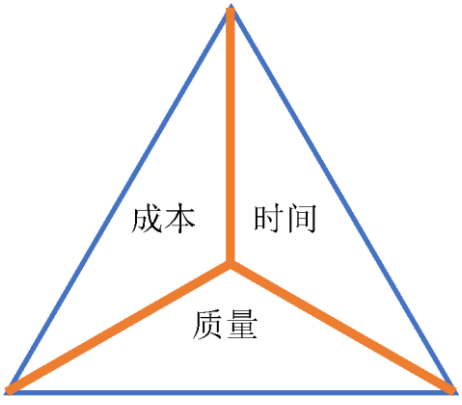
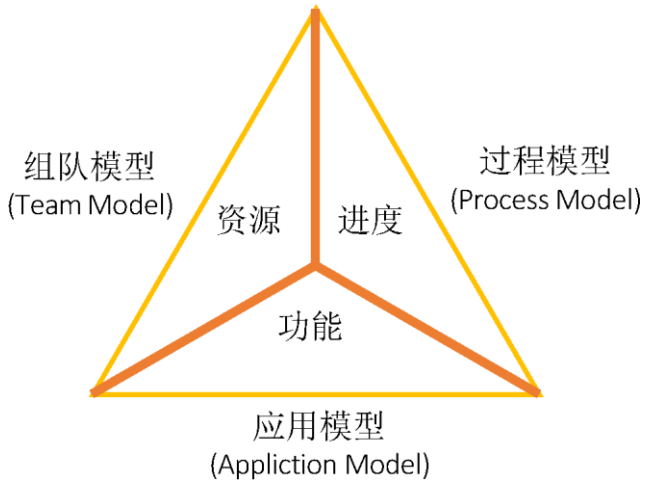


图 41 管理平衡三角



于是进一步地，MSF 提出了三种模型来解决对应的问题，如图 42 所示。

图 42 MSF 的三种模型与项目要素之间的关系



其中组队模型讨论项目创建时的团队、资源、沟通模式，过程模型讨论开发过程的控制与管理方法，应用模型则讨论产品定义、产品实现以及产品特性的管理。当这些模型映射到 Suite 或 IDE 中时，相关的要素就变成了类似如表 9 所示的关系[①]。

表 9 模型要素在理论框架与集成环境中的关系

理论框架（例如MSF）	集成环境（例如VSTS）
组队模型	Project集成、SharePoint集成等
应用模型	开发环境、测试环境、项目与项目组等
过程模型	需求、分析、开发、测试、BUG管理等

这一类工具在“一致性”上的要求与“统一建模”、“统一过程”

① 这种对应关系并非是边界分明的，事实上 IDE 中的同一个功能可能应对的是不同模型中的问题或问题集。

等思想同出一源。更确切地说，它们是“方法 + 过程 + 工具”这一传统工程模型的具体实践，即方法论决定了过程模型，工具是对方法论与过程模型的实现及其具体实施手段。

但问题在于：是要懂得使用工具，还是要懂得方法与过程？

13.4 敏捷工程实践者其实代表了工具精良派的产业工人的声音

集成环境或套件试图通过统一模型来建立整个团队对话、沟通、工作的一致化的界面，但这一过程其实并非依赖或只依赖特定的工具。

敏捷工程敌视那些强行植入**开发工具**的决策需求、管理需求以及产品需求[①]，进而否定“管理者、决策者以及产品相关角色”等在这一环境中的价值，并从“形式上”将这些角色推出团队，同时把产品定义与产品品质这些职责直接交给开发人员与用户。但是从根本上来说，它未能否定“过程模型”的价值。因此，大多数敏捷团队在弱化了管理与产品相关职责之后，仍无法摆脱软件工程过程（例如瀑布模型，以及基于瀑布模型的迭代）的约束[②]，只是他们有空间、有能力去选择更轻量、适用的工具来应付那些决策、管理与产品的需求。

除开这些因素，“敏捷”事实上是在探索用户与工程师进行有效沟

---

① 导致这一局面的部分的、且并非关键的原因在于：一方面，开发工具中使用 Project、ProjectGroup 等关键字来应对项目规模的扩大，而这一问题延伸到工程模型后所得到的抽象概念却是 Product、Version 以及 ProductLine 等；另一方面，工程工具与开发工具的提供商试图将这些合而为一，但形式上的统一未能弥补概念与领域上差异。

② 这进一步导致了团队无法摆脱既有的公司组织与管理结构，于是实施敏捷变得在组织层面上既说不通也行不通。

通的最简方式。这种方式并不是说不要“模型”，敏捷工程师（也包括“不敏捷”的工程师）其实也试图为用户所描述的业务数据与业务过程进行建模[①]。一部分建模工作是这些工程师“乐于”去做的，通常它们是面向业务的“数据与过程”的。因为这跟编程世界中的“数据结构与算法”有着近乎天然的映射关系[②]，只是他们将这些工作换了一门“手艺”来做罢了。例如在 UML 中，逻辑视图与实现视图构建的就是这两类模型。而他们“不喜欢”做另一些建模工作（例如完整的业务建模与产品建模），只是因为这些内容与开发工作没法关联起来，因而在他们的工作中是不需要的。

将 VSTS 等单纯视为“开发工具”是一个根本性的错误。既然这是生产线，那么正确的做法应是明确工程师在生产线上的角色，并将适当的工作交付给他们。但现实是，源于那些错误的认识，一些团队过度地要求工程师在生产线中的职责，反而激化了他们对相应职责的抵制。而敏捷工程与敏捷开发方法，不过是在这样的抵制运动中所诞生的“看起来有点革命性的”产物。

13.5 业务模型与产品模型对实施的价值有限

我们回到问题本身：决策域和产品域为什么需要（它们特有的）模型？

决策者需要的是业务模型。所谓“业务”，其核心包括它的产品构成与收益形式。作为面向商业运作的决策者来说，成本的开销与控制方式以及收益的获得与分配方式是整个业务的关键。**工程管理者**

[①] 例如，正如此前所讨论的，敏捷中的原型事实上也是敏捷工程师与用户之间沟通的模型。

[②] 你可以想象成开发过程中的“定义数据结构和写函数声明”，以及形式化的流程图与部分逻辑代码。

关注的则是一个产品或一系列产品的生存周期。这表现为项目过程，以及与项目过程对应的产品模型。业务建模与产品建模的区别在于：前者并不描述产品的当前特性以及延展特性，而后者正好关注这些内容。

需要特别提及的是，工程管理者通常关注一个产品/项目过程的实施进展、阶段规划、品质保障、成本控制等内容，而对产品之于产品线的中长期特性关注得很少。但事实上工程管理者需要产品模型的真正原因是，他需要一个蓝图，以及这个蓝图在长期演化上“可能性”。对于一个具体的工程管理者（例如某个项目经理）来说，产品的下一个版本或下一个系列可能根本与他无关。不过即使在这种情况下，他也需要在某些决策中用到上述的“可能性”，将其作为趋势判断的依据。

这些问题与城市建筑是类似的。业务模型相当于城市规划：东边要建行政区，西边建成科技园区，市中心以旅游和购物中心为主，而南边则进一步做旧城改造规划等[①]。产品模型，则相当于某些在建住宅小区的“XX 工程几期规划图”。这个图用在工程现场，每个施工者就知道他们在建造一个什么样的东西；用在售楼广告中，购买者就知道他们交钱买单的楼盘特点；用在小区周边规划中，大家就知道哪个地方该开个超市或为小区幼儿园留一个绿色通道；如此等等。

但是我们会发现——事实上我们做如上讨论与比拟的关键价值也在于此——尽管决策域与产品域确实需要基于模型来开展工作，但对于工程现场的施工者来说，这两类模型的意义并不大。例如你不能指望施工者因为楼房有商用与住宅之别就感到开心或沮丧，也不会

[①] 在地图上描绘类似这样的一个规划布局，然后试着将一个人工作生活的行经路线画出来，标上每天、每周、每月重复路线的次数作为权值，于是你就可以论证“为什么我们所在城市的交通这样拥堵”了。

因为大楼模型做得比别的工程现场精美，就能让施工进程更快更好[①]。

因此，在大多数时候，业务模型与产品模型事实上是无助于实施与评估实施的细节的。这也意味着，在软件开发/工程中，这些模型所采用的语言（文字、图、幻灯片或实物样品）只需要与它们主要的沟通环境相匹配即可，不必非要得到用户、工程师或者某个具体团队的认可[②]。

[①] 这里特别地忽略了一些细节，例如某种类型的楼（或软件开发产品）更难建造，或某个地区的楼（或业务方向）更受领导重视等。这些的确影响到具体团队、人员的选择，但它们并非这里讨论的“模型作为沟通工具的价值”。

[②] 一个设计或架构模型“要让所有人都理解”是相当荒谬的，而统一建模语言——请注意，我这里并不是指字面上的 UML，而是指“统一（某些东西）”这种思想——便是这类荒谬想法的具体实践。

第 14 章 应用程序设计语言的复杂性

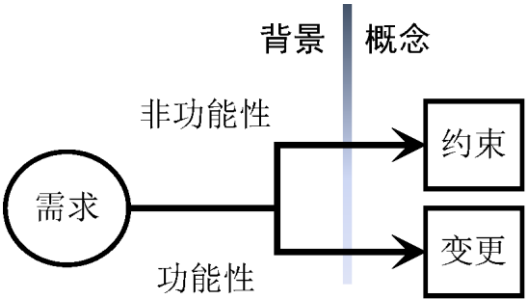
14.1 面向问题根源的两种求解思路：规则化与系统整合

回到最开始的问题。无论如何，工程师最终仍然要通过一个产品实现过程来满足用户的如下需求：

- 非功能性需求；
- 非当前需求。

这些是应用系统复杂性的主要构成。通常地，工程师会将它们映射为系统中的两个主要概念，即“约束”与“变更”，如图 43 所示。

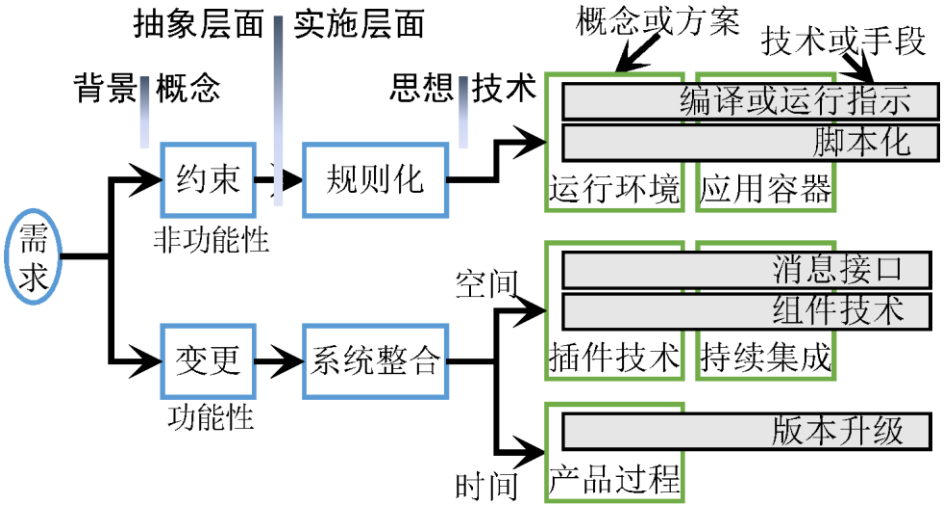
图 43 面对两种需求背景，所提出的主要（实施）概念



至于实施这些概念的具体技术与方法，对于工程师来说也并不陌生。例如非功能性中的跨平台问题，在早期软件开发的实践中，就是通过伪编译指令，即**编译期的约束**来指定目标程序的运行平台；又例如命令行参数，即用**运行期的约束**来指定目标程序的特定配置。就目前来说，这些实现可以归纳为图 44 所示的一些思想与技术[①]。

[①] 对于功能性需求来说，部分（尤其是时间与空间的区划）可参见“第 12 章 应用开发技术”。

图 44 实施层面的思想、技术与具体方法



接下来我们就来讨论这些稍为具体一些的“技术”，包括概念或方案，以及技术或手段。它们是相关的实施思想在具体技术层面的、两个不同维度上的实现。总而言之，所谓的**概念或方案**，总是对我们对于**技术或手段**在具体实施过程中所得经验的总结、归纳以及抽象层面上的提升。

14.2 对应用与应用容器进行标准化，是类似集装箱的一体化解决方案

应用程序并不是直接运行在机器环境中的，它通常依赖操作系统，甚至是依赖操作系统的具体版本。严格来说，所谓“操作系统或其具体版本”所提供的，就是一个运行环境。基本的操作系统主要封装了对硬件系统的访问，这包括：

- 存储（内存、硬盘以及移动存储管理）；
- 标准 IO 设备（显示器、键盘、鼠标、打印机等）；

- CPU 及其分配（进程与线程管理等）；
- 网络、媒体、通信等。

除此之外，操作系统提供文件以及其他操作系统资源的访问。[①]总的来说，操作系统为上述内容提供的 API（应用程序接口，Application Programming Interface）可以理解为运行环境的界面。但多数应用程序不单单依赖这个环境，例如使用 Visual C++编译的应用程序还依赖 Microsoft 发布的 VC++运行库[②]。

应用程序语言中的许多编译指令、伪指令、条件编译指令，甚至配置信息文件等，都可以用来做环境设定以及依赖分析，简单的如汇编语言中著名的伪指令：

```
.386
```

而复杂的则类似于 Linux 系统中的 Make 文件。

Make 文件也有很多种类，而且并不仅仅出现在 Linux 系统或某些面向 Linux 系统的语言中，事实上几乎所有稍具规模的应用程序语言与开发环境都有自己的 Make 程序和 Make 文本。为了提供更为复杂的 Make 功能，许多 Make 文本被设计为“另一种语言”，也带有逻辑、条件、变量等语言特性。但这些特性并不是相应“应用程序开发语言”的组成部分，而仅仅是用来对“这个程序的产生、分发等过程中的一些需求”加以规则化、程序化，因此我们通常也称之为 Make 脚本。

---

① 并不是所有的操作系统都提供文件访问支持，这取决于该系统对于“操作系统(的)资源”的定义。例如 Windows 系统中可以将注册表理解为资源，而 Linux 系统中就没有注册表。

② 对于 Microsoft 产品来说，操作系统中通常会带有该运行库的一个或多个版本，但某些情况下，仍然需要应用程序自己来管理这个库的安装与维护。

通过对规则的抽取、细化以及提供脚本化的支持，“越来越复杂地 Make”成了一个发展方向，几乎所有在“运行之前”能预料到的需求都可以通过下列技术来解决：

- 再编译一次；
- 分发不同的编译版本；
- 提供独特的安装过程；
- 特定的运行期环境依赖声明；
- 调整程序的启动参数。

然而这种方案有三个致命的弱点[①]：其一，我们不可能预料到问题的全集；其二，我们为可能出现的问题所付出的代价，甚至要多于编写应用程序本身；其三，由于这一方案（从逻辑上来讲）容许无度的需求，因而可能因次要特性过于丰富而导致最终产品失控。与这样的**运行环境**相对照的是，**应用容器**通过“将应用限定在一个明确设定的环境中”有效地规避了上述问题[②]——正因为环境特性由容器来决定，所以应用程序的开发与使用都不会超出容器提供的特性集的范围。

应用容器通常明确地设定了应用的类型，例如 Web 应用容器或企业应用容器等。容器会根据应用的需求来搭载相应的可选件，并通过统一、一致的接口提供给应用程序使用。这些选件与接口被标准化，变成应用容器整体方案的一部分。更进一步地，在这些范围确定、接口标准的方案周边，部署方案、优化方案、硬件配置方案、商业

[①] 这些弱点的表现之一是：Linux 下繁杂的产品生态（例如不同的编译、发布与维护版本），以及非功能性需求的实现过于复杂（例如安装）。

[②] 运行环境与应用容器两种方案，在解决问题时所处的层面是不同的。前者是从系统的视角出发来限制性地提供应用的背景，后者是从领域的视角出发来提供对应用范围的假设。

应用方案等渐渐地得以形成。

对应用与应用容器进行标准化，勾画了一个类似于“集装箱”[①]的王国，EJB（Enterprise JavaBeans）是这个方案的成功范例之一。EJB 的特性以及通用应用容器的规范，其本质上仍然是在“规则化”这一方向上，它们通过：

- 编译或运行指示，例如注解（Annotation）、验证器（Validator）、配置文件；
- 脚本化，例如事务脚本（Transaction Scripts）

等技术来实现“将非功能性需求限制在特定范围”。

14.3 模块化思维在产品交付组织形式上的延伸：插件机制

对于一个应用程序来说，“插件”往往意味着它是一个动态链接库、静态链接库，或一个独立发布的脚本包。如前所述，库或包，只是一堆代码在组织上的措辞而已。因此插件的本质在于用组织形式来解决空间因素所致的复杂性[②]。

“插件”这一方案意味着插件的宿主（HOST）与插件本身（Plugins、Addons 或 Extensions 等）是分别发布的。这事实上与应用容器有一定的相似性，只是两者应付的问题集存有稍许差异而已：

- 应用容器主要提供可配置的、可重新实现的环境，并通过标准化来提供跨业务领域的应用支持。尽管应用容器也有应用范围的限制，但通常该范围的边界更大。

[①] 参见《集装箱改变世界》，Marc Levinson 著。

[②] 这仍然是“第 12 章 应用开发技术”所述观念的具体实践。

- 插件的宿主通常在产品、用户和功能这些方面基本确定，在技术方案上也通常是预先可确定的，因此“宿主+插件”更适宜作为一个具体应用产品的实现手段。

与应用容器类似，宿主也有跨平台的问题，例如 Firefox 浏览器。这个问题有两个解决方案：其一，如果产品的用户有能力在不同平台上编译并重新发布宿主[①]，或者宿主是针对特定平台开发而无需重新发布的，那么插件通常实现为与环境相关的二进制模块；其二，如果宿主有明确的跨平台需求，而其编译过程复杂或不宜公开编译方案与代码，或者用户没有能力或没有必要去区别、编译不同平台中的插件版本，那么插件通常实现为文本形式的脚本或预先编译为跨平台的中间代码/代码包（前者例如 Firefox 浏览器中扩展名为.xpi 的插件，后者例如 Java 的.class 文件）。

插件技术是基于应用开发技术的“模块/单元”理念而来的。不同的是，宿主通常可以脱离部分或全部插件独立运行，而应用程序通常不能独立于模块/单元来完成编译。能否从“模块/单元”中去除强约束的依赖性，是识别能否代之以插件技术的可行方法。

如前所述，在面向对象的设计观念中，“类”替代了“function/unit/module…”等组织方式。因此在面向对象的设计中，“HOST 类 + 插件基类”是这一解决方案的基本设计。更进一步地，由于名字空间用于应付更大规模的组织，并且通常也具有“作用域”限制的能力，所以使用“HOST 框架 + （插件的）名字空间”也是一种可行的实现。这在本质上与前一种设计没有不同，只是应

[①] 例如 Apache 采用“WWW 服务器 + 扩展库（动态链接库形式）”的形式来交付，一方面是考虑到二进制模块的执行效率，另一方面也因为编译发布是部署人员应有的技能。通常来说，Linux 用户会更多地面临这一类情况。

用的规模更大，而且作用域范围控制更为灵活，类似安全性、账户等级限制等特性也往往更加容易实现。

但是基于面向对象的插件技术的问题在于：通常宿主与插件都基于相同的“面向对象语言”，因此具体的对象实现技术、二进制编译技术、对象封装技术等限制了插件的通用性。例如很难用 Delphi 开发一个“类”，并将它作为一个插件用在 Visual C++开发的宿主上。

这个问题的核心在于如何实现“组件复用”。这里的“组件”（Component）与此前的“模块”的区别在于：组件通常用于表达面向对象中的**可复用对象/类**，而模块通常用于表达结构化编程中的一个**可复用件**[①]。这两者所面临的问题是完全一致的，其解决方案也有着延伸关系，例如 Windows 中的 COM 复用与 DLL 复用其实是一脉相承的，而.NET 中的 Assembely 复用与 COM 复用也是类似的关系。

这通常涉及三种不同类型的复用（事实上划分它们的依据也并非唯一）：

- 二进制复用，例如 COM；
- 公共指令集复用，例如 JAVA 类；
- 文本复用，例如脚本。

无论技术方案的提供商如何渲染这些技术，其最终的结果是一样的：开发人员终于可以用 Delphi 写一个 COM 组件，并让它运行在一个 Visual C++写的类工厂中；或者用任意的 Java 编译器编译一个.class，并分发给其他编译、运行环境下的宿主。当这一切成为

[①] 并不一定是源代码中的一个单元（unit），而是指在指定的应用开发环境中，由语言或平台决定的“可复用单位”。

现实[①]之后，组件技术的核心便集中在了宿主框架的设计、插件的加载机制，以及隔离宿主与插件之间或多个插件之间的安全性这几个方面[②]。

插件提供了通过剪裁功能来重新定义产品的不同版本（例如标准版、专业版）的能力。与此同时，正因为它在概念上继承自“模块/单元”，所以也是一个“化整为零”的典型方案，这使得在大型项目中实施持续集成具有了可能性。这其中的一个有趣的事实是：如果集成需要所有“模块/单元”的参与，那么集成的失败率就会在系统规模达到一定程度后出现级数性的增长；只有被集成的产品可以通过类似插件、可复用件的机制，将单一部件的规模控制在一定范围内，持续集成——而不致阻碍整个 Team 的推进——才成为可能。

14.4 应用程序设计语言：缺乏真正的“产品版本”观念的语言是不成熟的

通过应用开发来“交付一个版本”，只是产品过程中的一个环节。只有当“产品”本身就是源代码包的时候，这个产品过程才变得跟开发人员息息相关，例如开发人员将 GitHub 或 ClearCase 视为版本管理工具，并将其上的某一个分支或基线作为一个“版本”。

然而从产品过程的全程来看，一个“版本”包含的内容更为丰富，上述“（开发人员所理解的）版本管理”只是产品过程的需求在开发环节的一个投射而已。总的来说，产品过程是一个工程问题，而非一个开发环节的技术与工具问题。它的部分问题集，被开发商置

[①] 我的意思是组件复用成为应用开发环境的一个内置技术，例如 Java beans。

[②] 尽管并不明显，但事实上这种隔离通常与消息机制有关。无论是从“消息”这一概念的历史还是从它在应用系统中的使用来说，它都是“体系性”相当明显的一种技术。因此本书将在“编六：系统的基本组织方法与原理”中去讨论它。

入了集成开发工具，并交付给开发人员使用。这个“部分问题集”实际上包括需求的变化以及与此相关的、变化的实现过程，而这是目前对于这一问题的“几乎全部”理解。

然而事实上这并不完整。例如我们的集成开发工具以 Project 或 ProjectGroup 为关键词来管理一个项目，但在产品过程中却是一个 Product，或一个 ProductLine。这一抽象概念上的差别带来了极大的思维空间，即开发人员是否应当基于 Product/ProductLine 来组织开发活动并进行所谓的“版本管理”？换言之，在 IDE 中是否应该出现比 Project/ProjectGroup 更高层次的组织行为，以及相应的、代码中的关键字？

事实上，加入了 CompanyName、ProductName 的名字空间就已经有了类似的性质。然而这一切，与在 IDE 中对产品过程加以映射、组织、管理与维护还有相当大的距离。

14.5 语言的进化方向——从“Hello World!”中可见的事实

现在，大多数程序员都可以写出一个具有典范意义的“Hello World”程序[①]：

```
class HelloWorld {
       public static void main (String args[]) {
              for (;;) {
                     System.out.print("Hello World");
              }
       }
}
```

同时也会真正地、从个人意识中忘掉这样一个程序的含义以及需求

[①] 引自：http://www2.latech.edu/~acm/helloworld/java.html

不过是[①]：

```
print("Hello World");
```

如果仅以这段程序而言，用户的需求仅用上述代码即可实现。而 Java 或其他一些应用程序开发语言则在这样的代码中加入了更多的概念，诸如：

- 类与对象等，例如：`class HelloWorld`
- 名字空间、导入导出等，例如：`System.out` 以及隐式地导入 `System`
- 方法、属性、事件，以及调用约定等，例如：`print()`
- 类方法、类静态成员等，例如：`static`
- 引用、值与无值，以及基本类型系统等，例如：`void`
- 可见性、作用域等，例如：`public`
- 字符串值、字符串类等，例如：`"Hello World"`与 `String args[]`
- 应用入口与运行环境约束等，例如：`void main()`

除了这些概念[②]，在具体的开发环境中还会有容器、包、配置脚本、服务、模型、验证器、指示字、伪指令、分发、部署、版本容器、基线等概念，以应付不同角色的需求。

当一门语言从“实现程序功能”变成要“实现产品需求”时，其内部的语言设计思想也渐渐地变得不遵守“算法 + 数据结构 = 程序”这一经典法则。回顾“篇二：语言及其面临的系统”，我们可以将这一切的变化以及可预期的、语言进化的方向都归结为：

> 通过在程序组织上的结构化来解决规模问题。

[①] 从语言实现上来说，其语义是 `print "Hello World"`，而 `()` 和 `;` 等符号是具体语言的语法。

[②] 其大部分与“面向对象”这一语言范式有关，部分则出于应用环境、语法习惯等在语言的具体设计上的选择。

篇五：系统的基础部件

> 解决“不确定”问题，需要首先将其背景置入到“确定”与“不确定”得以出现的本质原因中去。
>
> 正是数据不确定带来了对观察者的限制，进而这种限制带来了所谓数据连续或非连续这样的特性。但反过来说，如果数据是确定的，我们将不必限制观察者，也不必讨论系统是并行的还是串行的问题。
>
> 在一个足够小的生存周期中，我们可以做到数据确定；在可做到数据确定的前提下，我们将数据的生存周期扩展到足够大，则可以做到数据连续。在现实系统中，前者影响的是实时性，后者影响的是并发性。也就是说，高实时与高并发是最难兼得的系统特性，因为高实时意味着数据的生存周期小，也就意味着并发中面临数据失效（即连续性的背景——生存周期，达不到足够大）的可能性大。
>
> 大而化之，从结构的层面来讲，其中是可以有很多种解释的。

第 15 章 分布

15.1 系统应付规模问题只有两个总法则

聪明的曹冲称出了大象的重量。此前我们仅仅将这一思想归纳为“通过某种系统将大象的重量映射为石头的重量”，但这还远远不够。因为我们只触碰到了这个问题的一个解——映射，而“为什么需要映射”才真正是问题的本身。

曹冲的高明之处在于，他认识到“不能称象”的本质原因是：大象不能被分割。“不能分割”才是灾难之源。因此，如果**系统**如我们此前所讨论的，是“通过在程序组织上的结构化来解决规模问题”的一种策略，那么程序所解决的问题集“能否分割”以及“如何正确地分割”，就是所有**系统问题**的核心所在。

对此，曹冲称象的故事提出了一种可能的解：如果“被运算对象”是不可分割的，那么我们可以将它映射为可分割的对象。但即使这个解总是存在的，我们也只是看到了问题的一半。因为在曹冲称象的故事中，我们忽略了一个非常重要的事物：秤。

秤，实质上就是一个数据处理系统：

- 其一，它具备一个数据处理系统的两个基本要素：处理信息与反馈信息，例如称一块肉，并反馈结果：二斤六两。
- 其二，它存有一个基本限制：能够处理的信息边界，例如只能称重 100 斤。

也正是因为秤的数据处理能力有限，我们才称不出大象的重量。所

以整个“称象问题”既可以看做是“象太大”[①]，也可以看做是“秤太小”。

我们的计算机理论是基于这样一个事实，即计算系统的本质就是“算数”。而“被运算对象”的分割只是解决了其中“**数**的问题”。因此当我们将逻辑上的计算系统映射为一个实际实现——例如“称象”或者“计算机”时，我们也可以尝试去解决“**算**的问题”。

换言之，**系统**应付规模问题的总法则只有两个：

> 运算能力的分布，以及运算对象的分布。

15.2 分布的两个基本特性：可拆分与可处理

但什么是“分布”呢？

分布并不等于分割。“分割”是指一个问题集（无论是运算能力还是运算对象）能否被切分，例如前面一再提及的大象就是不可被分割的。而“分布”，指的是分割的结果能否被各个独立地加以处理。因此，我们说大象映射为石头之后具有了“（数据的）可分布性”，并不仅仅是因为在形式上进行了分割，还因为这些石头能被逐一称重。所以说：

> “可拆分”与拆分的结果“可处理”，这两个特性在“分布”中缺一不可。

与一把秤相类似，一个函数实质上也是一个数据处理系统：

[①] 可见“象太大”才是最原始、顶层的根本问题，所谓“大象不能分割”事实上是“化整为零”这个求解方案所带来的第二层问题；而“将大象重量映射为石头重量”，是进一步的求解方案。“映射/如何映射”作为第三层的问题，是远离原始问题的以及其本质的。

- 其一，很明显，它能“处理数据”[①]并反馈一个返回值；
- 其二，函数能处理的数据也有类型、边界以及运算总量的限制。

就我们通常使用的、单处理器的个人计算机而言，“所有软件构成的全集”所提供的功能总和可以被理解为“一个函数”（设为函数 F）。因为从计算机通电运行开始，系统开始了唯一一个程序入口与处理过程：

- 步骤一：进行系统自检、BIOS 预设等常规的、硬件系统自有的处理程序；
- 步骤二：尝试按照约定次序加载移动存储、外部存储等设备中的处理程序[②]；
- 步骤三：将系统的控制权转移给步骤二中找到的处理程序（的入口）。

请注意，在上述这个过程以及其后的全过程中，处理器（CPU）的处理其实是在单一的时间序列中进行的。而我们的操作系统之所以能同时运行多个程序（例如 Windows 的资源管理器与记事本），以及在后台与前台运行不同的服务与应用程序等，是操作系统：

- 将“所有软件所提供的功能总和”，即是我们上面假定的函数 F，分成了多个函数 F_0，…，F_n，计为 F'；
- 将 F' 理解为进程的入口，并假定 F' 可以再分成多个函数 F'_0，…，F'_n，计为 F''；
- 将 F'' 理解为线程的入口，并假定 F'' 可以再分成……

可见我们的操作系统（或某个硬件环境下的软件系统）无非是在对需要运算的总量进行拆分，并尝试将这些拆分结果分布在“不同的

[①] 这其实并不那么明显。其一，**处理**（process）是函数的基本抽象含义，如同它在数学中的含义是求解；其二，**数据**（data）是处理的具体对象，这是函数入口参数的基本抽象含义，如同数学中的公式是函数，而代入公式的那些数才是求解的问题（即数据）。

[②] 引导光盘、引导软盘，以及硬盘活动分区的引导扇区等。

逻辑单元”[①]中进行处理。

这一计算模型在单处理器时代被称为“分时处理”，即通过任务调度，将单一处理器的处理能力分配给不同的函数。这些函数在宏观层面上是同时运行的进程、线程等执行体，即**并行**[②]；而在微观层面上是分时运行的、单处理器的时间序列下的一个时间片，即**串行**。

回顾分时处理模型，其核心仍然基于“顺序机器”这一基本假设。与此相应地，其基本计算逻辑也是基于结构化程序设计观念，即可以将分支逻辑与循环逻辑统一为顺序逻辑的一个部分。进一步地，也可以将函数理解为一个子函数序列的连续运算。再进一步地，可以说：

> 如果子函数可以分布，则整个系统是可以分布的。

15.3 当依赖在时间维度下不可分解

在“进程－线程”模型中，如果将进程想象为一个函数（例如`main()`），那么操作系统将认为“各个进程所对应的函数`main()`”之间是可以分布的，也就是**可拆分**，并且每个拆分单元都是**可处理**的。因此为了达到这一效果，每个进程配置的“资源”也都是一样的，例如各自拥有显示器、硬盘、内存（地址空间）、键盘等虚拟设备。大多数情况下，在操作系统层面屏蔽了这个事实：上述的设备是硬

[①] 这里仅基于 Windows 系统的“进程－线程”模型进行讨论。事实上这些逻辑单元在不同系统中有着多种类型、多种层次与关系的抽象，例如作业（Job）、任务（Task）、进程（Process）、线程（Thread）、协和（Coroutine）、流（Flow）、事务（Work）、会话（Session）等。

[②] 注意这样的实现其实是基于“进程-线程”模型的并发（Concurrency），这些执行体只是在宏观上看来是并行的而已，并非真正意义上的、与串行（Sequential）相对的并行计算（Parallel, Parallel computing）。

件唯一的，每个独立进程只是持有了这个设备的一个“（可操作的）映像”而已。

无论如何，这些构成了我们的操作系统“能分时处理”的事实。随后工业界便一股脑地将同样的问题与同样的解决方案套用到“多处理器（多核）”计算机中去，认为在这样的计算系统中，无非是将“能分时处理的”那些函数放在了不同的处理器中而已。

然而这一切的基础并不牢靠：子函数真的是可以分布的吗？

对于一个函数而言，可拆分总是必然的。这是基于顺序执行的一个简单推理：若一个函数总是由顺序、分支与循环逻辑构成，且分支与循环总是可以被视为顺序逻辑的一个步骤，则函数必然可以拆分成多个顺序逻辑的步骤。

但拆分的结果（设为函数 A 与函数 B）是否都是可处理的呢？既然函数 A 与函数 B 是拆分自同一时序下的两个逻辑，这涉及两个关键问题：

- 其一，若函数的逻辑本身不依赖该时序，则可以处理；
- 其二，若函数 B 所处理的数据，在时序上不依赖函数 A 的处理结果，则函数 B 可以处理，反之亦然。

这两个问题的反例可以分别被称为：

- 逻辑依赖时序，例如函数 B 用于计算函数 A 的执行时长，则函数 B 必须在函数 A 逻辑结束之后执行；
- 数据依赖时序，例如函数 B 用于计算函数 A 的结果的倍数，则函数 B 必须使用函数 A 的结果数据。

它们准确地说就是“（逻辑或数据的）时序依赖”，即在时间维度下不可分解。这预示着我们的“函数”总存在无法拆分的可能。换

言之，必然存在无法通过“分布（或组织的结构化）”来解决的规模/复杂性系统问题。

15.4 要么是数据的全集，要么是它的映像

所幸在不那么学术的环境中，我们的应用系统只需要解决问题的部分，而非全部。[①]所以我们面临的往往是第三类问题，即对于函数 A 与函数 B 来说：

- 若函数在拆分时已经持有了所需处理数据的全集，则总是可以处理。例如：函数 B 计算 `x*3`，而函数 A 计算 `x*x` 的值。

所以就现在讨论的问题（的子集）来说，函数是否可以再被拆分其实受限于它所处理的数据是否可以分布。这也存在两类问题：其一，函数 A 与函数 B 所持有的**能否是 *x* 的不同映像**，即 *x* 可否存在各自独立的多个数据映像；其二，函数 A 与函数 B 是否能够持有**所需处理的数据全集**。

这在逻辑上是有解的。曹冲称象的故事提出了一种可能的解：如果“被运算对象”是不可分割的，那么我们可以将它映射为可分割的对象。所以在逻辑上，函数 A 与函数 B 总是可以持有

- 所需处理的数据全集，或
- 其各个部分的映射（的部分或全部），或
- 其整体的单一映像。

而我们最终要解决的，只是存放这些数据、映射或映像的方法，即存储问题。

[①] 这句话的意思是说：我们只好让那些不能拆分的逻辑运行在同一个处理单元中。

15.5 在结构化的思维框架下，函数拆分的可能求解

如前所述，函数的拆分与数据的拆分有一定的关系。总的来看，若函数 A 与函数 B 之间没有时序依赖，则函数 A 与函数 B 能否拆分取决于它们所处理的数据是否能拆分或复制（映像）。

根据函数本身的结构化性质，当某个函数拆分成函数 A 与函数 B 时，必然是三种逻辑结构所映射的关系。进一步地，它们对数据拆分的需要也各有不同。

其一，顺序结构意味着函数 A 与函数 B 可以使用数据的映射（的部分或全部）。例如下面的代码：

```
1  // JavaScript Syntax
2
3  // 示例 1
4  function foo() {
5    var a = 100, b = '...', c = 'hello';
6    a += 1;
7    b = c + b;
8    return [a, b];
9  }
```

`foo()`函数持有了 a、b、c 三个数据的全集，并且我们假设——事实上我们是特意这样构造的——函数的两个子步骤（代码 6 、 7 行）之间没有时序依赖。那么我们可以将 a、b、c 映射为两个数据，并在各自的子函数中使用它们：

```
// 示例 2，将 foo_1()与 foo_2()分布在不同的子系统中运算，结果 foo()的值将与上例一致
function foo_1() {
  var data = { a: 100 };
  return data.a + 1;
}
function foo_2() {
  var data = { b: '...', c: 'hello' };
```

```
  return data.c + data.b
}

function foo() {
  return [foo_1(), foo_2()]
}
```

在示例 1 中使用“`var`”声明的数据总量被拆分成示例 2 中的两个对象，由于示例 1 中的两个步骤之间是顺序关系，因此它们可以分别使用示例 2 中的两个 `data`，即“数据总量的映射”的部分[①]。

其二，分支结构（以及多重分支）类似于顺序结构，函数 A 与函数 B 可以使用数据的映射（的部分或全部），但是函数 A 与函数 B 相对于条件判断逻辑都存在“（逻辑的）时序依赖”。例如：

```
// 示例 3
function foo(x) {
  var a = 100, b = '...', c = 'hello';
  if (x) {
    return a += 1;
  }
  else {
    return b = c + b;
  }
}
```

在这个 `foo()`示例中，函数的两个分支之间是没有时序依赖的，但是它们都必须在 x 这个逻辑之后执行。由于 x 相对于两个分支不存在——或可以不存在——数据依赖关系，因此两个分支也可以持有各自的 data。例如：

```
// 示例 4
```

[①] 如果使用一个自动分布程序来处理函数 `foo()`，我们显然只需要进行完整的语法扫描，以确定 `foo_1` 与 `foo_2` 各自使用的那一部分数据即可。

```
function foo_1() {
  var data = { a : 100 };
  return data.a + 1;
}
function foo_2() {
  var data = { b: '...', c: 'hello' };
  return data.c + data.b
}

function foo(x) {
  return x ? foo_1() : foo_2()
}
```

请注意一个有趣的事实：示例 4 所使用的 `foo_1()`和 `foo_2()`，与示例 2 中是完全一致的。这意味着这两个逻辑以及相关的数据，与其外在的其他逻辑无关。这体现了它们的可分布性，即可拆分与可处理。

在示例 4 中，对于 `foo()`函数来讲，`foo_1()`与 `foo_2()`相对于 x 这个逻辑都存在“逻辑上的”时序依赖，但它们之间以及它们之于 x 的数据，都不存在依赖。

其三，循环结构意味着函数 A 与函数 B 使用数据全集，或其整体的单一映像。例如：

```
// 示例 5
function foo() {
  var a = 100, b = '...', c = 'hello';
  for (var i=0; i<100; i++) {
    a += 1;
    b = c + b;
  }
  return [a, b];
}
```

首先，一种错误的理解在于将 5、6 两行代码视作不存在依赖的两个

子过程，进而做这样的处理：

```
// 示例 6 - 不正确的逻辑
function foo_1() {
  // 对于 a+=1 循环 100 次, 返回 a
}

function foo_2() {
  // 对于 b = c + b 循环 100 次, 返回 b
}

function foo() {
  return [foo_1(), foo_2()];
}
```

尽管在这样的逻辑中，`foo_1()`与 `foo_2()`是可以持有数据的部分或部分映像的。但这与我们在这里讨论“循环逻辑”的初衷是相背离的。

我们事实上是在讨论将“一个具有循环逻辑性质的函数”拆分为多个子函数的情况。我们的目标是找到与“循环逻辑”这一性质相关的数据处理方案，而非将循环逻辑映射为多个却无视该逻辑之于数据的关系。在上述方案中，循环之于数据的性质是没有丝毫变化的。

将“循环逻辑本身”拆分开来，其基本含义是循环项次的展开。也就是说，我们能够将 100 次循环变成两个 50 次，或者 100 个 1 次。我们讨论的是这 50 次或 1 次中的数据之于“循环项次的展开”的逻辑间的关系。然而关于这一问题的答案是简单的：每一个循环项次，都必然面临数据的全集，或其全集的映像。因为 5、6 两行代码在时间上——可以理解为在一个时间区段中——是关联的，而“循环项次的展开”只是将时间区段趋向无限小的分隔，而并没有将上述这一关联关系解构。

以函数式语言的处理为例，我们可以将上述逻辑变成一个基于函数

参数界面的递归，例如：

```
// 示例 7 - 使用递归的方案
var a = 100, b = '...', c = 'hello';
function foo_x(i) {
  a += 1;
  b = c + b;
  return (--i <= 0) ? [a, b] : foo_x(i);
}

function foo() {
  return foo_x(100);
}
```

我们应该注意到，仅以“循环逻辑的展开”而言，函数 `foo_x()`的任意一个实例都只依赖调用界面上的 *i* 值。而这个 *i* 值是一个循环过程中的中间值，或是一个传入的确值，都是与这个函数无关的。因此，任意递归函数的单一实例，对于“循环逻辑”都是透明的。

然而再观察上述的示例 7，我们发现函数 `foo_x()`的任意一个实例，无论它仅是一个单次递归，或是分布到其他计算环境中的一个迭代区段，它都必将面临整个数据全集：

```
var a = 100, b = '...', c = 'hello';
```

一种较好的、较可行的方案是将这个数据全集也放在函数的参数界面上[①]。例如：

```
// 示例 8 - 使用递归的方案，并将数据关联在函数参数界面上
function foo_x(i, data) {
  data.a += 1;
  data.b = data.c + data.b;
```

① 对于在分布环境下的“函数的参数界面”，在 Erlang 中可以理解为消息，而在另外一些基于数据库、数据中心或数据结点的解决方案中，可以理解为持锁的数据项或结点。

```
  return (--i <= 0) ? [data.a, data.b] : foo_x(i, data);
}

function foo() {
  return foo_x(100, {a: 100, b: '...', c: 'hello'});
}
```

这样带来的结果是：`foo_x()`的执行可以被分布，但其“所有分布（的各个服务之间）”存在着逻辑之于数据全集的关联。在现实中，这一分布带来了逻辑向计算系统迁移的可能性，即一个大的循环过程可以分布在多个计算系统中完成，因而仍然是非常重要的大型系统下的分布解决方案。

但是整个循环逻辑与其占用的时间区段的总量并没有变化。

15.6 分布成本与处理成本也是难于平衡的

对于任意一个大型复杂系统来说，如果它能被拆分的话，我们拆分的模式无非上述三种。而且这种从逻辑的视角进行的拆分活动，最终也会表现为数据的拆分，例如子系统与子系统各自的数据库。更进一步地，当我们不再需要考虑这些子系统之间的数据关系时，整个系统将是可持续分布、并行计算的。

虽然从我们目前的分析来看，这一切是成立的，但是现实中的系统往往并不这样简单。其中的问题之一，在于“数据并不总是可以拆分”。例如一个简单的统计日志功能，其原始的需求如下：分析某个日志文件，统计日志的总行数。

缘于这个日志文件过大，我们对日志文件进行了数据拆分：每 100M 数据分成一个文件。这样，我们在整个逻辑上就可以理解为：对于文件 log1.txt 使用逻辑 A_1，对于文件 log2.txt 使用逻辑 A_2，如此等

等；对于整个逻辑，我们将对 A_1，…，A_n返回值求和[①]。

但是 log1.txt，log2.txt，…，logN.txt 这些文件之间却存在着关系。由于数据按 100M 分段，而换行并不总是发生在分段的边界之上，所以 log1.txt 末尾可能有“半行”属于 log2.txt 的第一行……类似这样的关系，使得我们在拆分 log.txt 这个数据总量时存在一些实际限制。

这些数据方面的限制，通常是通过逻辑来弥补的。我们总是试图让“数据拆分”这一活动更简单，或更规则。其简单，意味着分布它的成本很低，例如“数据按 100M 分段”的分布成本仅仅是物理存储上的限制；其规则，意味着处理它的成本很低，例如“数据总是在换行边界上分段”，那么处理它的程序将非常容易写，并且判断逻辑会少很多，执行效率也就高很多。

但如同算法时间与空间复杂度难于平衡一样，分布成本低通常处理成本就高，反之亦然。也就是说，对于既已存在的数据集来说[②]，简单而又规则地拆分是两难之事。

[①] 在这个问题中，A_1，…，A_n是否是相同的逻辑呢？这并非一个关键问题，因为我们现在仅在讨论数据的分布。

[②] 这也提出了数据规划的必要性问题：若我们现在有机会规划这些数据，那么必然能降低将来处理它们的成本。

第 16 章 依赖

16.1 在逻辑/数据时序依赖之间转换的基本方法

我们提到过“（逻辑或数据的）时序依赖”，是在时间维度下不可分解的。这意味着某些逻辑不可分布，而另一些情况下，某些数据也不可分布。对于后者，我们讨论了一种可能的、与处理逻辑相关的解：数据不可分布时通常会使逻辑趋向于复杂，例如需“判断 100M 数据的边界上是否存在换行”等。但这样的思路让我们陷入了困局：对于逻辑的时序依赖，那么又是否有解呢？

“挖坑、栽树”是一个（典型的、纯粹的、逻辑的）时序依赖活动。首先，这个系统中涉及的“坑”与“树”是不同的数据，相互间没有必然的依赖关系。其次，我们的的确确无法在时序上把它的逻辑倒置过来：无论是一个人来做，还是一群人来做这件事情，总得等挖完坑后才能栽树，这时多个团队或者 CPU 的多核并没有起到作用。

但注意一个细节 ：只有当我们将“挖坑、栽树”过程理解为“在**位置 A** 上挖坑，并在**位置 A** 上种树”时，“位置 A”才是这两个步骤的数据依赖。一旦去掉这个数据依赖，“挖坑”与“栽树”这两个行为本质上其实是没有依赖的，例如在“在**位置 A** 上挖坑，并在**位置 B** 上种树”就变得可行了。

这带来一个有趣的结论：我们（也许）[①]总是可以通过添加一层数据抽象，来将“逻辑的”时序依赖，变成“数据的”时序依赖。例如我们此前提到的：

[①] 很遗憾我无法证实这一点，但就目前的实践来说，这总是成立的。

- 函数 B 用于计算函数 A 的执行时长，则函数 B 必须在函数 A 逻辑结束之后执行，

就可以理解为：

- 设有时间戳 X，函数 B 修改该时间戳，而函数 A 统计该时间戳的修改。

两个逻辑作用于同一个数据（无论它们分别拿该数据来做何种处理），这是一个明显的特性。当这种特性存在时，我们称该数据是多个逻辑的**数据依赖**[①]。仅当我们能够通过对数据依赖的分析（而非对动态的逻辑执行结果的分析）就能确立这种依赖时，我们才可以自动化地分布与协调这些分布。

换言之，分布方式的确立以及在多个分布逻辑之间的协调等问题，都可以转化为对**数据依赖**的处理。再综合我们此前讨论的“数据的不可分布能够通过复杂的逻辑来解决”，最终可以得出这样一个结论：逻辑的不可分布并非无解，它最终可以被聚焦于“用于处理**数据依赖**问题的逻辑”的复杂性。

16.2 数据(x)的全集 = 数据(x')+ 操作(x")+ 状态(Sx)

一个简单的数据依赖的模型可以是这样的[②]：

```
x = 100;
function foo_1() {
  x++;
```

[①] 我们已经提到多个与“依赖”相关的概念。若我们只是单独使用“依赖”，则表明它是自然语言中的、表明多个事或物之间存有依存关系的含义。当我们使用“时序依赖”时，它是包括**逻辑的**与**数据的**时序依赖两种情况。当我们使用“数据依赖”时，我们是用这个概念来统一了上述两种时序依赖。

[②] 这个示例基于 Peter Van Roy 和 Seif Hardi 在《计算机编程的概念、技术和模型》（MIT Press，2004）一书中对“可观测的非确定性”（Observable nondeterminism）的讨论，原示例使用 Oz 语言。

```
}
function foo_2() {
  x--;
}
```

在这个模型中，若 `foo_1()` 与 `foo_2()` 是分布的，我们最终将不知道 x 如何分布，因此它们都必须使用数据 x 的全集。

如前所述，我们承认这种状况必然存在，无论它是发生于 `foo()` 函数的循环展开，还是用于解决 `foo_1`，…，`foo_N` 之间逻辑的时序依赖问题。我们的问题仅仅在于：如何让 `foo_1`，…，`foo_N` 都**持有这个数据 *x***。

我们必须再次确认“x 的全集”的含义。若“x 的全集”是一个有限的数据集合[①] [②]，例如某个内存块或者某个对象，那么“持有这个数据 x''意味着对这个 x 的全集的任意操作。这样的全集应当包括三个信息：数据、数据的操作以及数据的状态。

其一，x 包括数据 x'。显然，x 应当包括 x'，它必然是可计算数据的一部分。尽管 x' 在概念上能够指代可计算数据的全体，但在应用中仅是其部分，这就如同**变量**在概念上指代任何数据，但在具体的代码中只表达确定数据。

其二，x 包括上述数据集合 x'的所有已知操作 x''。对于某个确定的

---

① 若“x 的全集”是一个无限的、增量的数据集合，那么我们这里的规则就未必适用了。对于这种情况，通常有两种处理方案：其一，若该增量具有唯一的起始或标识，则可以以之作为**数据**全集的映射，并进一步用它来映射其后的状态；其二，若该增量是持续发生的，则它应当被抽象为一个与时间相关的函数，因而任意逻辑在任意时刻都只可能持有该函数运行结果的一个片断——换言之，它是不可能被持有全集的。

② 在对“x 的全集”这一概念的实现上，若语言中存在变量，则**变量**本身即可被理解为上述的“具有唯一的起始或标识”的无限增量数据，这是它适用于上述讨论的根本原因。若语言中不存在变量，则每一个数据——在它被唯一持有时——就是全集。其三，**闭包**这一概念的提出，在于无论数据在时间线上映像为何，在任意时刻闭包总代表着数据全集。

时间（所谓“已知”，便是与时间相关的），x'具有确定的操作 x''。这基于两条理由：首先，使得 x'在某一时间的操作具有不确定性——也就是说，既能够应用某个操作，又不可应用该操作——的逻辑是不存在的[①]；其次，若存在在将来可能发生于 x'的其他操作，那么这些操作与 x'也将构成其他的数据集合。

其三，x 包括指示它自身的状态 S_x。S_x 是 x 之于时间的信息，这意味着 x 在一个持续过程中可以被分别处理，这个信息既可以看做是 x 之于时间的依赖，也可以看做是 x 之于“某时的处理者”的依赖。

综上，x 必须是$\{x', x'', S_x\}$的全集。

16.3 状态：含义与可操作性完全明确的数据（值）

在 x 这个集合中的 S_x 只是数据，而并不应当包括 S_x 的操作。这是考虑到 S_x 本身不能再有任何的依赖——包括时间，因此它必须是一个含义与可操作性都非常明确的状态值。若使 S_x 与其操作都存在于同一个集合中，则它们必须只能是一个可分布系统的“数据全集 x_2”一部分，并且该系统的数据集 x_2 中也就存在相对应的 S_{x2}。

作出这一点限制才可以使 x 作为一个独立概念加以考察，并且可以将“数据”与“数据的状态”在概念上有效地区分开来。由于任何状态

- 本身也是数据
- 也能被作为数据 x_2'复合到一个“数据全集 x_2”中去
- 也能存有其确定的操作 x_2''

[①] 此前我们已经讨论过“计算的不确定性是对机器计算是否有价值的终极拷问”这一问题。它既是我们能通过“计算机语言”进行机器计算的基础，也是使得我们能够正确地将这样的计算应用于一个“数据全集”的基础。

因此我们才可以把状态作为一个数据，在程序中传递它、处理它，而不是仅视其为一个惟只绑定在“数据全集 x”中的状态。

作为“状态”本身，S_x 在现在或将来，或是在“数据全集”的任意一个持有者手中，都必须以及必然是**含义与可操作性**[①]完全明确的。例如，它仅仅是一个“0、1”状态，它的操作仅有唯一的“逻辑非”操作。当然，状态可能更为复杂，但只要有与之相应的、明确的、完整的逻辑，并且这些逻辑可以独立于这一状态，可以由不同的持有者实现，我们都可以将这样的状态称为 S_x。

综上，数据依赖在概念上只表明多个逻辑作用于同一个数据，它最终将被表达为面向$\{x', x'', S_x\}$的操作，其中$\{x', x''\}$表明被依赖的数据与其可确定的行为，而$\{S_x\}$表明一个有明确**含义与可操作性**的状态。

但我们并没有限制“多个逻辑”之间的时序关系。也就是说，数据依赖只用“状态”来表明：多个逻辑与数据的全集 x 都存在关系，但并不表明是一个并行系统下的关系，或是一个串行系统下的关系。例如，“（对同一数据的）多读单写”显然是数据依赖的，但在多读时多个逻辑之间是并行的，而在单写时它们却是串行的。

在我们的定义中，所谓“并发多任务”系统，是面向数据依赖的一个实现。例如，上述“多读单写”系统是可以依赖针对于某个状态的一组逻辑来实现的。更确切地说，所谓“并发模型”，其实就是[②]：

- 对上述状态定义一个操作集，并
- 在多个逻辑中实现该操作集。

---

① 我们稍后会再来讨论这一限制的实际含义。

② 并发的性能将取决于该操作集的效率，而并行的实质在于去除这个操作集。

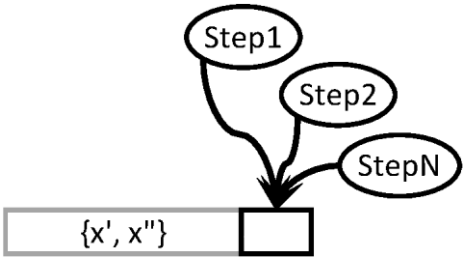
第 17 章 消息

17.1 函数的数据含义（返回值）只能表明 x'与 Sx 之一

我们为什么需要状态？

事实上我们是把两种时序依赖都最终归结于“数据依赖”，进而将这种依赖关系委托于这个“（数据全集中的）状态”。并且，我们要求状态本身只是纯粹的数据，而将状态的相关逻辑视作公共规约。我们其实是在讨论如图 45 所示的模型：

图 45 将状态从纯数据中剥离，并讨论与逻辑步骤(Step)间的关系



这一模型的本质仍是基于“逻辑作用于数据”这一思想。也就是说，它是基于命令式语言设计范式：我们将状态$\{S_x\}$视为$\{x',\ x''\}$的操作句柄，并通过某种规则在 Step1, ... , StepN 之间传递该句柄的执有权（类似于一种令牌）。

既然 S_x 的抽象含义要求“含义与可操作性”明确，同时我们也知道，函数既包含一个数值含义的传出，也包含可操作性明确的逻辑，可见函数在抽象概念上是满足“状态”的两个要素的。那么，S_x 本身为什么不能是一个函数呢？

问题仅在于，“状态”作为数据含义时是与$\{x',\ x''\}$相关的，而

“函数”作为数据含义时是指它的传出。也就是说：

```
function foo() {
  return 5;
}
```

可以在数据上表明：

```
foo = 5
```

但这也导致我们无法通过 `return` 来表明 `foo()`这个函数的（数据含义的）状态[①]。

解决这一问题的方法，就是消息。例如：

```
function foo() {
  send(bar, '...')
  return 5;
}
```

这个例子中，`foo()`向 `bar()`发送了一个消息’…’，这个消息用于指明 `foo` 当前所持有的数据的某个状态。由于 `foo()`可以持有多个数据——多个数据全集，或者由多个子集构成的集合——所以它也可以发送一个复合信息的消息，或用某个单一消息来指示所有数据的状态。

17.2 在函数式泛型下，函数=x'+x"，而消息用于表明 Sx

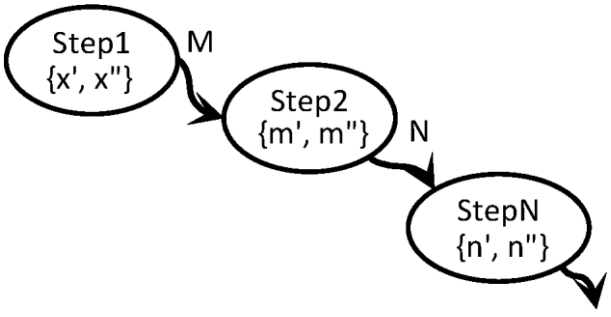
我们看到，消息本身跟状态是同义的——具有明确的“含义与可操作性”，只是在函数（以及函数式语言设计范式）中，接收者可以视为发出者一定持有了{*x'*, *x"*}，并通过消息 M 来**通告**了上述数据

[①] `return` 既已用于表明 foo 的数据含义，则必然无法用于表示“含义与可操作性确定的、foo()的整体含义”。即它不能既表示 foo()的部分含义，同时又表示 foo()的整体含义。这种“既是又非”有违于基本逻辑的矛盾律。

的状态[①]。

消息发布与处理，成为多个函数之间的一种关系，如图 46 所示。

图 46 消息的发布与处理：在整个逻辑步骤(Step)链条上的数据隔离



在这一关系中：

- Step1 与 Step2 之间的时序关系与数据依赖关系是不明确的：所有类似的关系与依赖变成了它们之于消息 M 的处理规则，亦即是更复杂的逻辑[②]；
- 数据之间的关系被彻底屏蔽了：Step2 并不依赖 Step1 的任何数据，包括它的入口数据或出口数据[③]。

表面上看，消息类似于调用，例如我们可以将 Step1 与 Step2 之间的关系看做：

```
function Step1() {
  Step2()
  ...
}
```

① 我们用到了“通告”这个词，意味着消息的发布可以是单个的，也可以是多个的。前一种情况可以实现令牌式的同步，后一种则可以实现多读单写式的同步。

② 这些逻辑基于 M 的数据含义，即$\{m', m''\}$，它被 Step2 作为私有的数据信息加以处理。

③ 同样地，当 Step2 发出消息 N 时，N 与 M，或 N 与$\{m', m''\}$并没有必然关系。

一方面，这种“类似调用关系”的形式的确意味着 Step2 在时序上依赖Step1；但另一方面，在我们讨论的消息中，实际并没有“调用”这层关系。也就是说，Step1 向 Step2 发出消息 M 之后就可以执行自身的逻辑了，并不需要等待 Step2 的执行，也不依赖 Step2 的结果数据[①]。

消息并没有类似回调的性质，尽管消息可以用于实现这一性质。Step1 发出消息 M 这一行为，并不表明 Step2 必然执行某种由 Step1 决定的逻辑（例如回调函数），也并不表明消息 M 是某个函数的数据依赖。

消息也没有类型与结构的约束。一个消息以何种复杂程度的方式来记录消息体，是 Step1 与 Step2 之间或者 Step2 与 StepN 之间的约定。除了约定的双方或多方对规则的强约束外，消息本身并不存在语言层面或计算机系统层面的约束。因此，基于消息的模型很容易适应复杂的计算环境。

17.3 消息是剥离了所有数据性质的状态

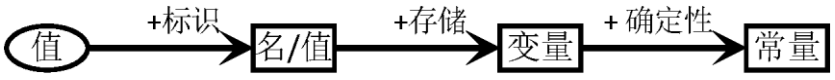
状态与消息其实是一个统而一之的东西，它们不过是两个程序设计范式对同一事物的不同叙述。这样我们又回到了最初讨论问题的方式，即“两个侧像”。

回顾此前的讨论，我们曾经设定数据的全部性质中有一个不可或缺的子集为**标识、值和确定性**。其中值是数据的本义，标识是人与计

[①] 这意味着，“发出消息”对于 Step1 来说是一个“有确定结果的确定行为”。这一确定结果是“没结果”，因为在 Step1 中不存在对这一结果任何有意义的依赖。而“函数调用”正好是存有对某种结果的依赖的。

算机进行沟通时所需的抽象定义[①]，确定性则是数据与使用这些数据的计算系统的最终极的问题[②]。基本上，我们可以将数据在计算机语言中抽象描述为图 47[③]：

图 47 数据在计算机语言中的种种抽象与概念递进关系



但是我们上面讨论的**状态**，是否需要全部三个性质呢？

这一设问的关键在于：我们将状态 S_x 作为数据全集 x 的一个分量，就意味着 S_x 的三种性质与 x 的其他分量 x'、x''必须是协同分布的[④]。因此设问若成立，就意味着任意一个数据在其分布过程中都要保证这三种特性的协同分布。例如数据 A，其分布为 A_1，...，A_n，在同一时刻 A_1，...，A_n都必须是同时确定的，或同时不确定的。又例如——以其字面上的含义来看——我们必然面临类似的需求：在一个分布系统中的多个子系统中，要求同一个状态使用完全相同的标识。

对于状态，Peter Van Roy 是将它作为多种范式中对数据的不同理解

① 这是程序设计语言的需求，即标识符（token）、名称（name）或词（word/keyword）的本义。

② 在之前的种种讨论（事实上也包括接下来的讨论）中，凡涉及对“数据性质中的确定性”的追问，必然会对我们在计算模型、语言范式等方面提出新的挑战。

③ 在从变量到常量的概念变迁中，某些语言并不引入“常量”这一概念，而是通过强调“变量不可写”来强制数据必须具有确定性。在类似的语言范式（例如数据流范式）的基础观念中，通常还涉及变量仅作为标识符而未有确定值时的抽象。例如在 Oz 与 MapReduce 中，如果一个变量尚未绑定值，则它将阻止运算它的函数与任务（job 或 process），因为这时的运算也是不确定的。直到该变量存在一个确定值，阻止才得以解除。

④ 这里的协同分布是指：未必一致，但至少是通过某种规则来约束。

来看待的[①]：

> 状态是记忆信息的一种能力，更精确地说，是及时存贮值序列的能力。它的表现能力极大地受到其所在范式的影响。我们将其表现能力划分出四个等级，它们的不同在于状态是未命名的或命名的、确定的或非确定的，以及串行的或并发的……非确定性对真实世界中的交互很重要（例如在"客户端/服务器"编程模型中），而命名状态则对模块化有着相当重要的意义。

在这一观念中，状态是一个可变的数据：表达**某个**存储位置上、**某时**的信息。换言之，数据的不确定性——包括其静态的确定性——表现为状态。这使得**状态**在概念上很类似于**变量**，不过这也并没有什么不妥，毕竟在大多数语言中，我们的确是用变量来实现状态以及相关的机制。基于此，Peter Van Roy 非常深入地剖析了状态的数据性质。

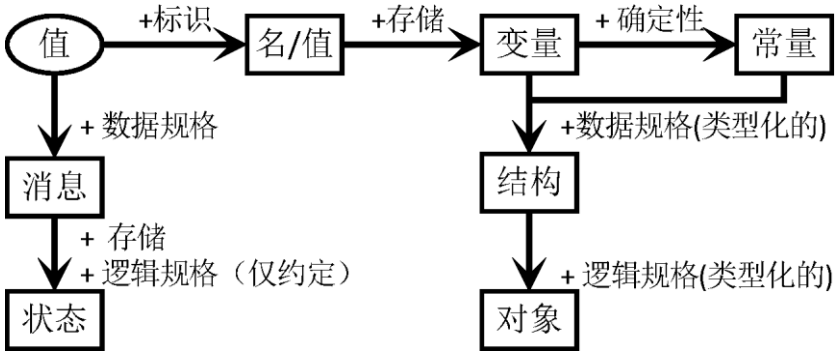
把 Peter Van Roy 的思考逆向探讨一番，我们可以得出这样一个结论：当不考虑一个存储位置上的命名特性时，它就既非变量或常量，也非某个确定的运算对象（例如"对象"等高级的抽象概念），而只是一种更加泛义的"状态"；同时存储这个状态本身的事物，由于没有位置、时序等概念，所以借用我们之前使用过的名词，可以称之为"**存储位置**（cells）"。

一旦我们不考虑这个存储位置本身的物理限制，而只考虑其中数据本身的、值的含义时，我们也就可以脱离系统（例如某个系统的多个子系统）地称之为"消息"。消息既是系统之间的**约定**，也可以视为是游离于系统之间的、借着这些约定来约束系统的**数据**。

[①] 引自 Peter Van Roy 对"主要编程范式"一图的说明。参见：
www.info.ucl.ac.be/~pvr/paradigms.html

综上所述，状态与消息的抽象概念的区别在于：前者往往带有位置之于存储，或数据之于逻辑的含义；后者则将位置与逻辑含义都抽取掉，只认为消息是一个约定所需的数据部分。图 48 表达了二者的关系[①]。

> 图 48 约定规格：值性质下（状态与消息）与变量性质下（结构与对象）的抽象概念对比



其中，对于消息：

- 其数据规格部分是应用中的所需，而非概念上的必需；
- 约定所需的逻辑部分是由其他系统根据约定实现的；
- 约定（在概念上）是可遵守、可更新，以及可违约的。

既然其约定是一个非数据的规约，是系统间在逻辑实现上的约束，并且事实上这一约束也可能不被遵守，所以可以说：

> 消息就是消息，消息没有任何特定含义的性质。

[①] 该图也从侧面表达了变量与状态在数据性质上的相似性，以及在结构化程序设计和面向对象程序设计中“可以以结构、对象等类型化数据来实现**状态**”这一事实。

第 18 章 系统

18.1 天生支持完美分布的系统：煮鸡蛋

煮鸡蛋，大概是最简单的美食了。这个世界上已知的最完美而又同时是最简单的煮鸡蛋方法就是：把鸡蛋和水放在锅里，煮。

有两个极简单但不强制的限定条件：其一，水要漫过鸡蛋；其二，烧火的时间要够把鸡蛋煮熟。所谓“不强制”，是指在大多数情况下：水不漫过鸡蛋也是可以的，只要别煮干；而鸡蛋煮得不是很熟，或者多煮了一些时间也没关系。

而且，煮一锅鸡蛋与煮一个鸡蛋的方法和限制条件居然是完全一样的！这意味着，我们煮鸡蛋的算法以及作为数据的鸡蛋，乃至煮鸡蛋所需的整个系统，天生就是支持完美分布的！

但问题是，你并不能确保做好这份美食。

18.2 把鸡蛋煮熟是不可能完成的任务

当一口锅架在那里的时候，你没有办法保证：

- 没有人或其他动物突然闯入，打翻它；
- 没有地震、飓风等自然灾害发生，破坏它；
- 没有柴尽水干的极限情况；
- 时间不会停止；
- 空气不会凝固；
- 战争不会爆发；
- ……

事实上，除开这些宏观因素，你也不能保证这样一些眼下所需的条件：

- 没有不好的鸡蛋；
- 没有不爆的木柴；
- 没有不漏的锅；
- 没有不塌的灶；
- 端锅时不烫到手指；
- 取蛋时不滚到地上；
- ……

更有甚者，你想煮熟鸡蛋，你却根本不知道：

- 如何看出这颗鸡蛋已经熟了；
- 如何保证一锅鸡蛋的每一个都熟了；
- 保证每一个享用美食的人都吃到他们认为“熟了”的鸡蛋；
- ……

所以，你看，我当年打算学厨的时候，就被我父亲断然拒绝了。

18.3 我们事实上只能关注可控领域中的可控因素

Joy 都可以煮熟鸡蛋[①]，但作为架构师的我，却只得出了一个结论：煮熟鸡蛋的可能性似乎连 1%都不到。

为什么？

因为当架构师看待一件事物时，他看到的是它在系统中的全局问题；

[①] 其一，看过前言的人应该知道 Joy 是谁；其二，Joy 总是把鸡蛋煮得很熟，这是一种过度的安全措施。

而一个实施者，却只需要去做局部的那个部分。当我们把“全局”这个问题牵扯进来之后，任何事情都变得有无限种可能性，因为任何一个系统都是更大系统的局部，而且这个命题是无限递归的。

蝴蝶的翅膀，只是一个生物学的问题；蝴蝶翅膀的扇动，只是一个空气动力学的问题；如此等等。如果我们不能像这样地限定一个系统的领域边界，那么架构师所能得出的结论永远就只有一个：我们什么也做不出来。

所以在大型系统（以及任何一个需要架构的系统）中，我们得承认两点事实：其一，我们的系统永远都只是更大系统的局部；其二，我们只能在实施中关注可控领域中的可控因素。

那么，如何应付未知系统以及那些非可控因素呢？

18.4 聚焦领域之于系统的主要需求：维护状态或接受消息

答案是：忽略。

忽略掉一部分问题，是我们在系统实施中所必需的策略[①]。除了能降低当前系统与外部系统的耦合关系、减少系统处理异常情况时的成本之外，更重要的一点是，这样的策略有助于我们快速地将问题聚焦到该系统的核心领域。

除开计算机的两个关键问题——计算总量与数据总量之外，我们将领域之于系统的主要需求变成一个简单的定义：计算系统如何应对

[①] 对于灾害，一个简单而实用的策略是：备灾甚于防灾。例如考虑到海底光缆断裂，那么就让 30%的数据通过陆路光缆；考虑到地震频发，那么就将应用分布到不同地区的机房，并在各机房做全镜像数据。这样的措施也被扩展到类似的系统问题上，例如考虑到战争，就将重要的计算资源部署在中立国；考虑到断电，就要增加备用电源，以及设计机械解决方案。但总的来看，这一类措施都是“忽略问题本身”的。

外部系统的**需要**。而进一步地考察这一需要，就可以发现：在“可计算”方面，任意领域的系统对“当前系统”的需要只有两个，即寻求计算资源，或寻求数据资源。

换言之，任何一个子系统——作为一个更大系统的组成部件——对外部系统只需要承担这样两种责任[①]：

- 维护可对外公示的状态，使得外部系统可以将明确的行为（逻辑）施于自身；
- 接受外部的消息，使得外部可以通过传递数据来影响自身。

对于接下来要讨论的问题来说，“系统由子系统构成”是一个基本设定。更具体地说系统包括**子系统**、**通信**与**验证**三个基础部件[②]，而本书的下一章将基于这些部件，来讨论与架构实施相关的技术方法，以及通讯与验证相关的问题。

[①] 或之一，或全部，取决于系统设计的具体策略。

[②] 将**子系统**、**通信**与**验证**作为系统的三个基础部件，是在《我的架构思想：基本模型、理论与原则》一书中讨论架构组成论的一个基础。

篇六：系统的基本组织方法与原理

> 甲邀请乙：“周六我等你一起去打球”。这个邀请，很有可能会以甲无休止的等待而告终。无论乙是出于何种原因爽约，例如赴约途中被掉下的卫星碎片砸中，甲终将因为发起“一份失败的邀请”而懊恼一周、两周……从此他可能不再对乙抱以信任，进而对所有人都不再信任，于是他的“人际交往系统”受到种种限制，乃至出现失效，最终不得不求助于心理辅导……
>
> 这里我们看到了两出悲剧：一种是处理异常，例如乙的爽约；另一种是框架失效，例如甲的邀请。
>
> 这两种悲剧都会出现在我们的软件系统之中。因此我们的系统不稳定，既来自于客观上不可避免的异常，也来自于主观上框架设计的失败。

第 19 章 行为的组织及其抽象

19.1 领域间的交叉与交互才是系统规模问题的根源

在系统的定义及其规模化这两个方面，一个大型的系统与煮一个鸡蛋的系统并没有本质的区别；当这样的系统在数量指标上增长时，它面对的问题与煮一锅鸡蛋也是相类似的。但我们并不是要试图去讨论“它是不是一个系统”，或者“它是不是一个在数据量或运算量上足够‘大’的系统”，因为总的来说，这两个方面并不是我们在这本书中定义“系统”（system）这个开发规模时的本义。

反观“应用”（application）这个规模，它关注特定的应用领域，却并不强调对该领域之上的整个行业链条的观察。例如开发一个看图软件，我们并不会将数码打印行业的功能给整合进去。又例如设计一个资产管理软件（有人也称为“某某系统”），我们并不会将相应公司的人员管理体系也纳入设计范畴。**特定应用领域**决定了应用的特性及其在领域知识上的复杂程度，并且在一定程度上，也表明其产品构件可以在相同领域中复用。

而“系统”这个规模是包含“应用”的，因为这一规模的定义本身就是由跨领域引申而来。例如在线支付系统，它本身涉及 Web 应用、支付应用、金融类应用、在线交易类应用，同时也涉及服务器端数据安全和挖掘、大规模数据和计算的分布以及容灾等领域。不同的领域有着各自的领域知识，以及相对独立的开发技术、框架与业务模型。

但系统本身的复杂性并不是由这些领域带来的，正如我曾经说过：牌局的复杂性其实不是由“分开牌”这个动作导致的，而是由“交

叉牌”这个动作导致的。我们将一个系统规划为各个领域，这些领域自身的问题仍然只在“应用”这个规模级别之上，而领域之间的交叉、交互才是“系统”这个规模自身的问题。

这也是“系统何以作为一个规模”这样的问题的根源。

19.2 大型系统已经逐渐走入细分领域的时代

当然，数据量或运算量上的规模依然是系统的一个（外在的、表现上的、实际应用中的）问题。就目前的实践而言，我们事实上也把它们独立为计算机系统开发中的领域，例如分布式计算领域以及分布式存储领域[①]。

分布并不等于并发，这一点我们此前已经讨论过。但分布必然意味着要处理逻辑与数据之间的耦合关系：其一，耦合紧密的逻辑不易拆分，也就不易分布；其二，逻辑与数据紧密耦合，也存有类似的问题。现实中的通用应用开发语言（例如C、Java 等）是在“算法 + 数据结构 = 程序”这样的结构化理论上发展的，为了增强语言的通用性，其结构化的特性非常明显。这并不利于数据与逻辑的解耦，例如在一个既存的面向对象应用中，想要将对象的数据成员与方法成员解耦并分布，是一件不可想象的事情。但在这个问题上，并不是说具体的语言有多大的问题，而是这个程序设计范式本身的理念与我们讨论的背景并不适配。

[①] 事实上，业界现在又把这两个学术化的领域统一称为“云”，包括云计算、云存储等具体的解决方案。关于数据或运算的规模也存在许多其他细分的领域和混杂的概念，例如早期的网格（Grid）与现在的大数据（Big Data）。无论如何，这些都只是分布问题在业界的具体表现而已。

在现在的大型通用应用开发语言[①]中，接口的引入是解决上述问题的一个良好实践。但接口带来的是面向行为的设计理念，例如在这样的环境下，我们讨论的问题通常是：某个对象或某个子系统所具有的哪些行为，是可以被抽象为接口的。而实践过程中的这些具体工作，与对象本身的设计以及与面向对象系统的设计是没有多少关系的。也就是说，接口设计与面向对象设计是两回事，只是实践中把两者综合在一起，并试图解决后者的一些实际问题。

这些“实际问题”，具体来说，就是“逻辑如何分布”[②]。

另一方面，现实的软件开发中的“数据”，并不总是和对象一样地通过一个实体（instance）与方法绑定起来。我们面临的这些数据[③]要么是结构化的，例如结构化文本或结构化数据库，要么是非结构化的，例如待索引的文件或者流式数据。这意味着，通用应用开发语言对于处理大数据量几乎毫无帮助。例如在实际工作中，如果这些数据存在于关系型数据库中，我们将使用 SQL 语言来操作；如果它们存在于 NoSQL（非关系型）数据库中，我们就会采用类似于数据流式语言或其他特定的数据处理语言来操作。

所以综合来说，即使仅仅观察“数据+算法”这样的软件开发本质工作，大型系统也被具体分成多个领域了。而软件开发进入领域细分时代的大背景，才是像“云计算”这样的理念得以提出的基础。

---

① 就目前来说，通用应用开发语言主要是指面向对象语言或结构化程序设计语言，例如 Java、C#，以及 C 语言等。

② 因此，在缺乏这样的需求——例如在一个小一些的、跨领域特性并不突出——的系统或应用中，要求实施“面向接口设计”是一件相当多余而又学术化的事情。

③ 另外，我们也可能通过静态数据与增量数据、本地数据与远程数据等分类法来区别不同的数据。我在这里采用“结构化”这一分类法只是一种选择，仅为了说明常见的问题，而非一个强制设定。

换言之，我们一再讨论的分布问题，本身也可以被理解为领域问题。

19.3 服务与结点：“一组接口”在两种视角下的抽象概念

系统由领域组成，领域又由细分领域组成……由此我们最终所讨论的、具体的、作为组成部件的领域被表达为：

- 包含了特定领域知识，
- 提供了基于上述知识的处理能力，
- 反馈了在领域间可理解信息的一组行为。

对于这样一组行为，我们称之为“（该子系统/领域的）接口”。请注意，我们讨论系统中的接口时，它是脱离具体语言的。我们需要这样一层抽象概念的根本原因在于：

(1) 将领域问题转义为一组逻辑的界面；

(2) 将领域间有无关系，转义为“领域对领域（调用者与被调用者）”的接口是否可达；

(3) 屏蔽领域内的数据性质[①]，迫使开发者必须通过特定的设计来解决领域间的数据问题（例如数据类型、数据依赖等）。

在这三种原因（或可称为三类需求及其解决手段）中，第一种和第二种是显性的，它们由接口的抽象特性决定：其一，表达为一组方法；其二，能否使用这些方法，取决于能否获得接口。

第三种则是隐性的，并且也是决定系统稳定性的最主要因素。事实上在这样的背景之下，开发者能够选择的方案也相对有限，主要包

[①] 这涉及数据的全部三种性质：标识、值与确定性，以及由确定性带来的分布、状态等性质。

括远程方法或远程对象、消息/状态、序列化、分布等。

最终，我们将提供一组接口的系统组成构件称为一个“服务”，或一个“结点”。前者是功能视角下的一个概念，后者则基于部署视角[①]。但总的来说，服务/结点都意味“非本地”的支持，这也可以理解为对“跨领域的需求”的支持。

[①] 两者除了抽象视角的不同，在实现上通常也有差异（这里的服务与结点主要分别基于 Java 和 Erlang 中的概念）。

第 20 章 领域间的组织

20.1 得到系统的基本方法是部署，而非开发

系统与子系统之间面临的第一个问题，是系统的整体部署。系统的部署方案几乎决定或限制了大多数有关系统的决策，其中首当其冲的是数据的结构化与预结构化问题。

数据在哪里？数据的型式是什么？数据的量级、频度与粒度如何？这是系统整体部署中避无可避的三个关键问题。我们先讨论 B/S 架构下的数据，通常它们的大多数数据项表现为微数据，其数据粒度小、频度高、可靠性差。

频度高意味着单次处理数据的 CPU 占用要尽可能小，这是维持大的系统处理能力的诀窍。但如果一笔数据的结构不确定，发现数据有效性以及决策计算路径的代价就将变得极其巨大。举例来说，Java 的 Web 应用框架大多提供“将 HTTP Request 转换为一个数据对象”的能力，其优势是在应用层中不需要关心数据来源与有效性。这一方面可以让应用层用类同的方式处理请求，另一方面也可以屏蔽一些框架逻辑，例如通过注解（annotation）来做数据验证。而且在第三个方面，这还可以将服务端与请求端无缝隔离，例如一个应用处理逻辑并不关心请求是来自于 Web 客户端，还是来自于 Open API 接口。

但是“将 HTTP Request 转换为一个数据对象”的代价极其巨大，其本质上是在服务端、在应用服务器的框架层上进行数据的预结构化。如果我们理解这一点，那么我们在 Web 客户端对数据格式进行预处理，并与服务器端将这一规格协商作为协议，其效果将是十分显著

的。例如大多数 B/S 应用会面临登录验证的问题，它们需要处理大抵三类需求：

- 如果用户未登录，则一些客户端请求无效；
- 如果用户未登录，则一些客户端请求被保持，并在再次登录后提供服务；
- 如果用户已登录，则可以为客户端请求提供正常服务。

“登录与否”有许多种特定的验证方法，例如是否存在用户名（UserName），是否存在用户会话（SessionId），是否存在验证数据（CertData）等。但我们这里先关注“验证数据的获取”，这在一个 HTTP Request 中有几种来源[①]：Cookie、Get Fields、Post URL 和 Post Data。其中 Get Fields 与 Post URL 是同类型的东西，因为“Method 为 Get 的表单”中的字段最终会作为 URL 请求中的字段信息提交到服务端，URL 中包括 Path、Action 和 Fields 等。

那么请问，验证信息（以 SessionId 为例）应该在哪儿呢？

20.2 数据的规格化要尽量远离具体的处理逻辑

一个 HTTP Request 通常要经过与服务器端的多次通信才能完成提交，过程大致如下：

- Web 服务器只负责接收数据，并在适当的时候将执行权转换到应用的框架层（Framework）；
- 框架层会解析这些接收到的数据并构建（Build）请求对象，或将某些数据持久化（例如文件上传）；

[①] 在定制的 HTTP 客户端中，也可能将验证信息直接放在 HTTP Head 中，例如超星浏览器等使用 HTTP 协议的软件。但是这在通常的、基于浏览器的 B/S 应用中很难有通用性（如果非要这样做，可通过浏览器插件来实现）。

- 最后，框架层将一个请求对象作为应用可以理解的数据格式，例如 Java 对象，提供给应用逻辑来处理。

在这个过程中，如果数据有效性验证发生在应用逻辑中，将是效率最差的。相较而言，较好的位置是放在框架层“构建（Build）请求对象”这一环节中（A 方案），而更好的位置则是放在服务器接收数据这一环节中（B 方案）。

B 方案通常是通过 Web 服务器插件来实现，它不太方便的地方在于：如果在其中加入大量逻辑，对服务器整体的稳定性将构成影响，并且这些逻辑可能是重复而无意义的。举例来说，如果我们试图以 B 方案在一个插件中实现预结构化 URL 中的请求数据，那么 Web 服务器需要解析 HTTP 请求的 Head 部分，读取 URL 并解码、定位字段、验证……而这些过程既拖慢了服务器响应，而且相同的逻辑还需要在框架层上再做一次。所以虽然 B 方案中看起来是“更好的位置”，但对于我们这里要解决的问题来说，却是相当不妥的。[①]

A 方案通常是应用服务器内部的逻辑。为避免细节，这里只提及一个问题：需要“将 HTTP Request 转换为一个数据对象”完全完成之后，才能进行后续处理吗？这个问题是“数据如何进行结构化，以及在哪里进行结构化”的核心。举例来说，如果客户端具备规格化数据的能力，那么我们可以要求“在一个有效的 HTTP Request 中，字段数据的第一项必须是 SessionId”[②]。将这作为一个约定时，我们看

[①] 在这里提及这种可能性，是因为某些情况下，在这个位置上会验证客户端的来源（例如 IP），或强制使用 Get/Post 请求，或判断是否有文件上传等。这些措施通常通过添加服务器模块（Module/Filter）并通过配置文件来实现，也通常是类似防火墙（Gateway/Firewall）软件所需的功能。

[②] 也可能存在其他的约定，事实上有些人认为将 SessionId 放在 Cookies 中更加有效。但在这里，我们是假设要同时验证“HTTP 请求的行为(Action) + SessionId”两项，因而可以放在 URL 中并通过一次解析来得到完整信息。

看 B/S 框架的各层之间是否能实现这一协议：

- 在浏览器端，可以通过在 URL 后添加字段数据的方法使 Get/Post 请求都满足上一协议。以 Post 请求为例，当请求的 URL 中带有“?SessionId=xxxx”时，它可以被提交到服务端作为 Request Fields 数据。
- 在服务端，当解析 HTTP Request 并尝试 Build 时，可以从 URL 中顺序读取到 Action 信息，以及接下来的 SessionId 信息，这应当是在一次解析过程的前后连续动作中发生的（先是 Action，接下来是 Query Fileds）。因此可以按顺序逻辑来实现“判断第一个 Field 是否是 SessionId，并判断它是否有效”这一行为。
- 在服务端，如果上述判断为“否”，则框架层完全有能力响应浏览器端请求，例如（1）重定向到登录页面，或（2）返回标准错误信息，或（3）保存当前请求作为事务数据，以开始一个“登录+重新提交”的过程。

接下来让我们看看这一系列的复杂过程有何收益？其一，当一个客户端请求发生时，服务器可以在没有完全接收数据的情况下作出响应，这提高了客户端的体验；其二，一个无效的请求在抵达（执行过程将更占用 CPU 的）应用层之前，在仅需最少的执行周期的情况下，就已经获得了正确的判断信息并处理了。

从这一过程中，我们可以得到一个结论：数据的结构化阶段离处理阶段越远，其系统的整体收益也就越高，即数据要“尽可能早地”结构化。但结构化是一个逻辑过程，需要相应的部署环节的支持，例如我们需要考虑 Web 客户端层的整体系统结构。

只有在部署许可的情况下，我们才有讨论“结构化与预结构化”的位置的可能性。例如基于客户端与基于 Web 的即时信息（IM，Instant Messaging）软件，就非常不同，前者对数据的预结构化能力非常有限。通常在系统中，我们寄期望于数据的发出者具备结构

化的能力，并尽量提供在多个子系统[①]之间的标准（Standard）、基础（Base）和基元（Meta）信息格式。如果这一点不能满足，我们也通常不在具体应用逻辑中处理格式，而是将这一过程交由中间层来实现。例如将来源数据读取为内存数据库，或添加数据接收服务器，来将端口或远端数据转换为系统内部支持的格式。

以尽量远离处理逻辑的方式实现数据规格化，即使仅仅是形成了类似“Name/Value”这样粗粒度的数据索引[②]，也会给后续处理逻辑带来巨大的收益。

20.3 关系型数据库的原罪：序列关系 + 键关系

一般性的数据处理的思路，是通过多次的数据筛选将待处理数据层层精化；在得到所需的处理数据之后，将这些处理数据从数据层迁移到应用层；最后，由应用层来处理之。关系型数据的基本原则，是根据“应用需要”来构建库中数据项次之间的关系，并规格化存储（包括索引与键等）。这一原则其实是有利于实现上述的“一般性思路”。例如我们通常通过在数据库中执行 SQL 语句来得到一个数据子集，然后通过应用程序中的数据库客户端得到它，并转换为应用程序识别的数据对象再加以处理。

不过，这个思路有一个致命的问题：如果“应用逻辑所需要的处理

---

① 在系统组织中，不要仅仅将当前正在开发的应用程序视为系统的组成构件。例如某个已经部署上线的“文件服务器”，就可以通过我们在上一小节中提到过的基于部署视角被理解为数据“结点”。这一结点的界面是“操作系统的文件存取接口”，包括 System API 或 SMB（Server Message Block）协议下的共享访问接口等。

② 基于 Linux 系统在文件存储上的优势，“Path/File”也可以被视为“Name/Value”或“Key/Value”数据存储的一种，并且通过文件路径以及文件名（filePath+fileName）也可以实现简单 Hash。事实上，这些面向存储的数据处理，比我们通常在应用程序中、发生在函数调用界面上的数据更为常见。

数据”本身就是一个极大的数据量呢？例如搜索，无论用户输入怎样简略的一个关键字，他的原始意图都得是在整个数据全集中检索之。事实上，搜索的特殊性就在于任何一个应用逻辑都是面向数据全集的。关系型数据库对这一问题的解释是，检索行为应当交给数据库端来实现，进而相类似的、基于数据全集的应用逻辑也都应当交给数据库端来实现。

于是我们看到了基于关系型数据的数据处理得以进化的动力：由于数据从数据层迁移到应用层的成本过高（这包括数据本身的传输成本，以及在应用层实现相关处理逻辑的成本），因此将相关处理作为独立技术在数据层实现是必然之选。在这一层次上的需求已经不是传统的基于数据库管理（DBMS）技术所能应付的，例如联机事务处理（OLTP）试图在一个事务中完成对数据的切片、加锁与运算过程，而这往往导致整个应用层需要花大量时间来“等待”数据层的运算结果。

反思这个运算瓶颈的核心，关键在于关系型数据库对数据规划的理解，它本质上是强调数据项（Rows）间存在序列关系，以及数据列（Cols）间存在的键关系。这两类关系使得大数据处理时，无论基于列的纵向拆分或是基于行的横向拆分都面临“如何处理关系”的问题。例如将一个类似于日志数据的大表（Rows 项次过大）作横向拆分，我们必须在处理层——无论是数据库的事务或是应用层的逻辑代码——中解决对“基于年、月、日或季度、周等时间区段”的数据的归并以及跨区段处理的问题。又例如将一个类似于订单数据的宽表（Cols 项次过大）作纵向拆分，我们就必须通过多表的 Join 来得到最终所需的运算数据，这又涉及 Keys 的维护，从而加大了应用端或数据库端的逻辑复杂程度。在这个问题上，联机分析处理（OLAP）在本质上也只是通过多次的、渐增的数据规划来强化数据

关系，通过需求规划来增加中间数据，从而减少单次处理的时间。

那么是否需要将这两类关系“解锁”？去除掉这两类关系的数据，又是何种数据呢？

20.4 NoSQL 代表了对“数据可变”的理解

对这个问题的思考将数据处理带入了 NoSQL 数据库的领域。其中具有一定代表性的是 Key-Value 型的非关系型数据存储方案，即基于 Keys 的分布来存储数据项，这使得数据项在 Rows 维度上是无关系的。对于类似日志这样“天然存在 Rows 关系”的数据，Key-Value 的解决方案是分段/区存储，这其实是在规划阶段对数据进行了横向拆分，而这事实上又导致了小规模处理此类数据时的不便。

Key-Value 数据很好地从纵向规划的角度将大数据切分成多个数据簇。在这个过程中，Key-Value 是允许存在冗余开销的。对于关系型数据库来说，这类似于索引的存储开销。但正因为一笔数据在多个位置冗余，所以 Key-Value 面临同一数据的多点更新问题，这使得数据的 Update 周期变长。这一策略换取的收益是：通过一个单一入口获取信息变得迅速而简洁，并且可以用较小代价来应付多层 Key-Value 关系，而后者正是关系型数据库的弱项。

如果 SQL 是被视为一门语言（事实上它既是语言，也是一类数据库系统的称谓），那么 NoSQL 就是另一类与之本质上存有不同的语言的统称。SQL 的语言特性就是基于批量数据的提取与后续操作，它在定义上总是基于“一个数据子集”来思考的。与之相对照，NoSQL 却是基于“如果数据子集的获取是一个过程，那么这个过程本身是否可以被分布”来思考。

在 NoSQL 的思维方式中，数据是通过一大批获取过程、在一系列被分布的数据中“收集”而最终得到的一个结果集。在这一过程中，将“数据获取”分成多个逻辑以及多个匹配路径是必需的。NoSQL 的这一设问，正好与我们此前讨论的“是数据的拆分，还是逻辑的拆分”不谋而合。在 NoSQL 环境下，大数据将不再是一个中心，而是一个被提前规划、提取与规格化的“数据广场”。一个很好的比喻是这样的：关系型数据库将数据理解为矩阵化的养鸡场，而 NoSQL 环境下面对的数据环境则如同鸽子广场。要在这样的数据环境中实现运算，“从不同维度来规划、标识与筛选鸽子”将是数据处理者的责任，而不再是数据库厂商的“不传之秘”。

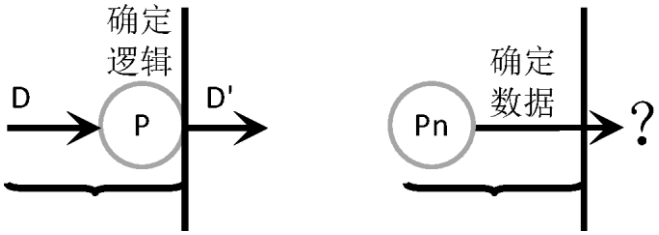
这些（既是负担，也是自由的）责任在 NoSQL 方案下仍然是通过语言来实现的，例如 MapReduce（如果我们把它看成是一种编程范式的话）。在这一模型下，Map 过程用于从一个数据系列中获得数据子集，而 Reduce 过程则将些子集再次规格化为新数据：如果还需要后续处理，则持续进行 Map/Reduce 以得到适合应用层处理的数据。这一设计思想是基于（比结构化编程范式）更为原始的数据流编程范式（Data flow programming paradigm），从本质上来说，这也是函数式的思维模式。这些对程序设计语言范式的考察揭示了大数据处理下的新思维体系与传统体系的差异。将 NoSQL 与 SQL 对比来看：

- NoSQL 代表对“数据可变”的理解，而 SQL 代表对“逻辑可变”的理解；
- NoSQL 认为逻辑（例如 MapReduce 的数据规格化过程）是不变的，应当让数据“流过”逻辑，进而得到数据某些方面的映像；而 SQL 认为数据（例如关系型数据的既定结构）是不可变的，应当让逻辑“作用于”数据，进而得到一个可呈现的观察。

20.5 海量数据运算中公开的秘术：传递逻辑而不是传输数据

数据或逻辑的不变性是这两类大型系统（以及系统开发语言）的核心区别，如图 49 所示的两种“PD 模型”[①]：

图 49 两种“PD 模型”：数据或逻辑的不变性



函数式语言，例如 Erlang 和 MapReduce，主张第一种系统模型（另一种模型在后文中另作讨论），因此适宜于编写逻辑确定的系统。它使数据 D 穿过逻辑，形成 D’，这一过程是确定的；而由 D’与其他的、后续的逻辑构成的部分（子系统或领域），并不影响当前系统的确定性。

但是我们也可能会注意到，在网络上产生数据（例如发表博客或提交回复）时，数据是动态产生的；而当这些数据被收集起来之后，我们展示它们时数据却是静态的。简单地说，数据收集和规格化，与基于数据的应用程序开发看起来并不是一回事。在后者，即对于我们一般意义上的应用程序开发（而非数据处理），我们通常还是会选择第二种模型——让逻辑作用于数据。

而这一模型事实上并不简单。例如，即使我们的数据是确定的，但

[①] PD 模型描述处理（Process）与数据（Data）之间的关系，通常用于对计算范式的描述。

如果它本身是海量的、分布的、复杂结构的，又应当如何处理呢？仍以我们在上一小节讨论过的“搜索”为例，如果一个搜索引擎已经获得了数以亿计的网页，分布在不同物理位置的存储设备中，又如何能让一个关键字搜索快速得到响应呢？

首先，真正发生一个“按 Key 搜索”的行为时，事实上是不会到具体网页中去“逐字节查找”的。关于“Search Keys”与“Page Keys”的关系以及“Page Keys”与存储的部署关系，将涉及非常多的系统策略，在此我们不细讨论。我们这里只关注“如何发生查找逻辑”这一个问题。而这个问题的本质是：如果逻辑所需处理的数据是不可迁移的，那么逻辑可否迁移？例如，如果我们的“按 Key 搜索”行为可以被分拆为多个子处理，那么能否将这些子处理“送入到”数据所在的集群（Cluster）并完成运算呢？

传递逻辑而不是传输数据，是海量数据运算的另一个思路，而这一思路依赖的条件是：逻辑的子过程的分拆是可能的、可控的[①]。在类似 MapReduce 的方案中，Map/Reduce Jobs 的执行就具有类似的特点。也就是说，我们必须关注另一个事实：

> 语言选型与系统架构在“数据与逻辑的可变性”上是可以互为补充的。

20.6 以数据为中心：从会话中抽离状态

接下来我们讨论三种常见的、以数据为中心的服务：会话、登录与镜像。

[①] 我们讨论过，这正是函数式语言的优势——将逻辑作为可迁移对象通常是基于运行机制来保障的、具有逻辑自身的不知觉性，而命令式语言则难于实现这一点。但看起来，我们在这里所面临的又似乎是典型的“数据不变”的系统环境。所以语言范式之于系统模型，是两种语境。

会话这一服务，是将数据理解为公共数据与状态。通过会话来形成事务是一种常见的策略，但其可靠性是受置疑的。例如我们可以将用户在 Web 上的一次交易，看做是一个有着松散的事务关系的过程，从选择商品、下单、确认、支付以及转入到发货环节。这在整体上有着工作流的特点，在部分结点上有着事务性的要求。在这样的背景下，我们要求从选择商品开始建立一个会话，并在其后的行为中基于会话来实现处理逻辑。

将会话作为一个“状态”其实是非常危险的，因为 Web 交易中不可避免地会发生用户刷新页面导致的重复提交。因此，在同一个状态上出现两个等待逻辑的状况就会出现。若这种逻辑又正好是在事务型的结点上，例如支付，那么该怎么办呢？

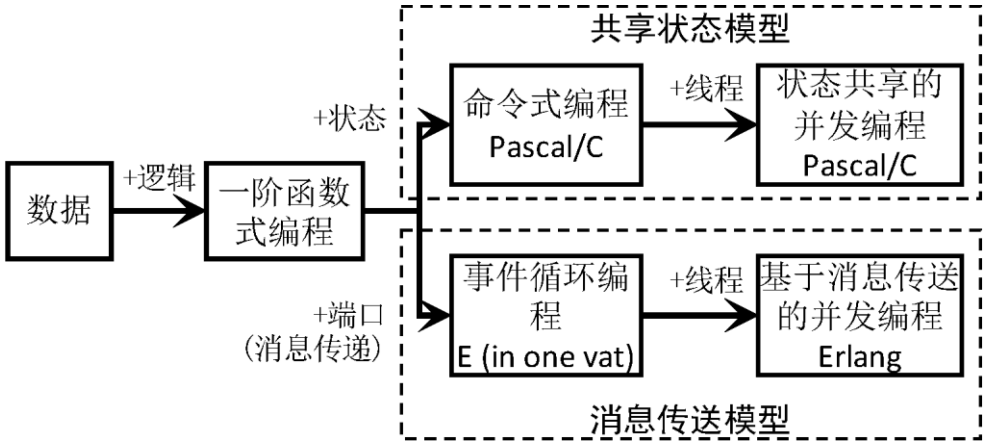
在稍小一些的系统中，由于相关的领域和应用结点不是非常多，因而我们还有能力应付交叉逻辑中的种种锁关系。但如果系统的规模相当大，应用逻辑相当复杂，那我们就必须回到最原初的设定上进行重新思考：会话，是否应当将数据作为状态？

在此前我们讲到过，一个能保证确定性的“数据全集 x”必须是$\{x', x'', S_x\}$整个集合，其中的S_x就是状态。将会话数据作为状态是典型的$\{x', S_x\}$集合，因此它表现为数据确定而逻辑（x''）可变。我们之所以会在逻辑中出现锁关系，便是因为逻辑对 S_x 产生了依赖（而非对 x' 产生依赖）。因而将“状态(S_x)”这一性质从“会话数据”中抽离是我们必然面临的问题。

一个可选的策略是加入消息服务。这基本上借鉴自语言设计中对并发的处理，例如在 Pascal/Delphi 系列中采用的共享状态模型，以及在 Erlang 中采用的消息传送模型。图 50 说明相关语言特性的演

化[①]：

图 50 两种可选可替代的并发模型：共享状态模型与消息传递模型



这说明共享状态模型与消息传送模型是可以相互替代的。具体来说，如果我们试图从会话中抽取掉状态，则应该考虑在多个任务（task）中“有效地传递或限制传递”这一状态(S_x)，而不是将 S_x放到另一个数据结点中去。后面这种策略，以及之前提到的会话服务事实上都是因为 S_x的存在而形成了数据单点。

当我们加入消息服务 M 之后（消息本身的“数据性质”中并不包括 S_x，并且也不包括逻辑 x''），任务 $T_1, \ldots, T_n$之间便只因为“共享了会话数据”而变成相同的 n 个任务。当它们的“修改状态”请求被投送给 M 之后，M 的端口（或其他消息限制策略）将请求的消息信息队列化，进而将状态 S_x中的时序信息拿掉[②]，因而在具体的、在消

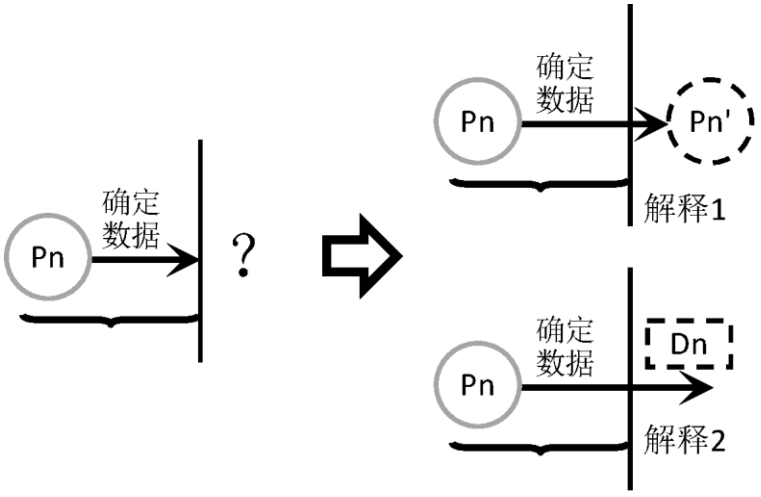
① 该图是对 Peter Van Roy 的“主要编程范式”一图中部分内容的重新表达。参见：http://www.info.ucl.ac.be/~pvr/paradigms.html

② 从概念上来说，数据全集 x 包括指示它自身的状态 S_x；也就是说，S_x 是 x 之于时间的信息。

息服务 M 中的、有关消息响应的逻辑中，就不必再处理时序信息了。

在消息服务 M 中是否使用同一个会话数据作为上下文，是消息模型中的一个可选策略。这可以表达为对“（确定数据下的）PD 模型”的两种不同理解，如图 51 所示。

图 51 对确定数据下的 PD 模型的两种不同理解



在解释 1 中，认为逻辑 Pn 通过确定数据之后，得到的是一个包括数据映像的新逻辑 Pn'。在这种情况下，新逻辑是可以基于数据映像进一步求值的。这一思路可以理解为缓存，即在一个存储上加上 IO 逻辑。当通过 IO 逻辑得到数据时，我们并不关注数据是原始的确定数据，还是这一数据的一个映像，并且事实上我们也不关心映像与确定数据之间的同步问题，这是 IO 逻辑之下的服务层的责任。解释 2 则认为逻辑通过确定数据之后，得到的是可以用作后续计算的数据。因此，这一思路可以理解为函数连续调用，也是函数式语言的一个基本范式。

这两种模型都可以解释“消息服务 M 中是否使用同一个会话数据作

为上下文”[①]：对于解释 1，要求会话数据是通过一个存取界面得到的；对于解释 2，要求会话数据是在消息 M 的处理逻辑中自行持有的（例如可以队列化、加锁，或使用类似闭包的方式得到一个映像）。

20.7 以数据为中心：单点

登录，是第二种以数据为中心的服务，它本质上就是以数据为单一结点的。也就是说，登录在逻辑概念上，就是数据单点的。登录的问题通常包括并发数量巨大、容易形成访问峰值，以及逻辑过长三种情况。

第一种情况通常在大规模的应用中都必然会出现，例如 Gmail 或者 QQ。当规模渐增，且我们必须“先完成登录验证再提供服务”时，登录逻辑的 CPU 消耗总量是不可能消减的。解决这一问题的基本方法是增加“登录服务”的物理服务器的数量，例如在 QQ 客户端上加入一个登录服务器列表，按照客户端的 QQ 号 Hash 到某个服务器去登录。又例如 Web 客户端（或其他没有处理登录位置能力的客户端），则可以考虑通过 DNS 轮循[②]来将请求投送到不同的服务器。

第二种情况常见于区域崩溃的恢复以及类似暴力攻击的时候。例如 2007 年台湾海峡的地震导致海底光缆断裂时，七条连接香港与外地的海缆中只剩下一条能维持有限度服务。在这种情况下，区域部署的系统（在软件与硬件上）都将受到访问峰值的冲击（访问浪涌）。即使不讨论这样极端的情况，在服务器维护进行重启后的短时间内，也通常会形成这样的峰值。而对于大多数在线系统（Online

[①] 这里的“同一个会话数据”即是指数据确定，并强制要求使用该会话数据的逻辑不得有 update 操作。

[②] 通过配置 DNS 服务，使相同的域名解析为不同的 IP，对应于不同的物理服务器。

Service），首当其冲的就是登录服务。其原因便在于此前提到的：登录在逻辑概念上，就是数据单点的。应付浪涌的通常方法是“排队”，即在客户端或服务端添加有效的排队机制，将对登录服务的冲击控制在一定规模之下[①]。对于服务端来说，使用专用的队列服务，而不是依赖登录服务自身提供的任务池机制来处理也是相当重要的措施，专用队列可以负载更高，且对登录服务的处理能力不构成负面影响。对于客户端来说，提示用户“何时将会再次登录”是一个算法问题。事实上，我就遇到过同时打开某网站的十几个页面，而这些页面都卡在“短时间打开过多页面，请等待刷新”的提示页上。当使用浏览器的书签组功能时，这是常见的。这些页面采用的刷新值都是 5 秒钟，而十来个页面请求在 5 秒钟后“同时投送”到服务端的几率是相当高的。当你将这些页面想象成客户端连接，你就知道客户端延迟会导致请求最终像“共振”一样趋向同时抵达服务端。于是在上面的例子中，服务端与客户端都将同时处于 5 秒钟的饥饿等待与 5 秒钟后突如其来的请求爆发当中。

第三个问题比较特殊，通常我们不会主动在登录中加入过多的逻辑，但那些看似“无可拒绝”的产品需求则是例外。其中，“登录+权限”是一个最经常的假设，另一个则是“登录+验证码”。对于前者，事实上可以作为两个结点来进行设计，如果我们也将登录信息与权限信息设计为不同的数据的话[②]。对于后者，其计算的复杂性在于获取和核查验证码导致的外部调用（我们通常不会把验证码系统做在登

[①] 所谓“一定规模”，即是在服务上线前对“响应数/秒”的理论计算，也需在服务期间进行长期跟踪调适。运维与监控是大型系统中相当重要的组成部分。

[②] 只不过它必然带来二次交互，或在登录中发生向权限服务的远程调用。但一旦将验证与权限设计为两种数据，则这两种方案替代的成本极小。我在实践中，就曾经要求在开发中使用一次交互（在登录服务中向权限服务远程调用），而在上线时使用二次交互并分别部署登录与权限服务。

录服务内），而这通常是通过验证码的预生成和本地缓存来解决的。然而还有一种特殊情况，即类似于在做第三方接入时，我们会异地验证用户的有效性。例如在系统中接入新浪微博账号，就需要远程调用其 Open API 并“等待”返回结果。第三方的计算复杂度或调用延迟将大大地拖慢登录服务的响应，进而导致 CPU 耗用很低而连接数很高（处于大量等待）的情况。从本质上来说，这也是因为在登录中加入了其他逻辑而导致的。

综合上述情况，如果系统中必然出现“（数据的或逻辑的）单点”，则应当一方面在单点服务内部入手，消减逻辑复杂度并剥离出更轻巧的服务，以减少单点逻辑的规模、提升服务性能；另一方面从客户端与服务端两面入手，尝试提供更合理的“请求－响应”逻辑，保障服务提供的体验；第三个方面则要努力尝试通过逻辑与数据的分布来消灭单点本身。

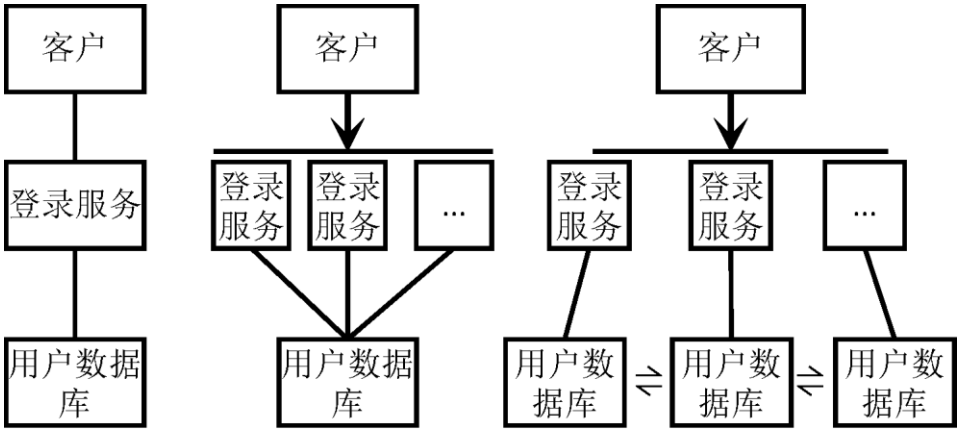
20.8 以数据为中心：数据结点——用数据映像替代数据全集

对于上述的第三个方面，作为登录以及类似这样的服务，是否有可能设计为不是单点？数据镜像机制提供了一种可能，这也是我们要谈到的第三种以数据为中心的服务。其根本出发点在于通过后台的镜像与同步机制，将数据作为不变对象以保障逻辑对“数据的位置”的不知觉性。

图 52 比较了三种登录模型。

图 52 三种登录模型：单数据库方案（左）、增加应用服务器的方

案（中）、增加数据镜像的方案（右）

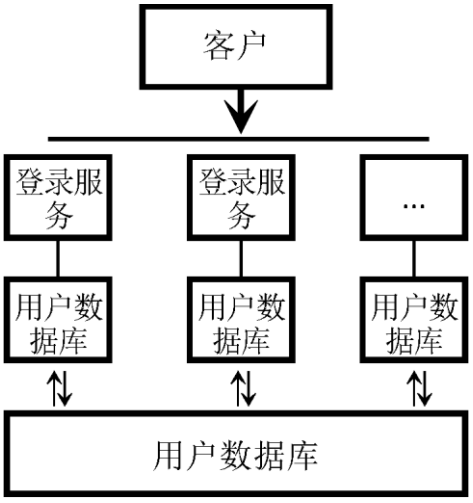


第一种是传统的单数据库方案。第二种是登录服务逻辑过大导致应用服务器资源紧张，同时数据库压力不大时，通过部署应用服务来提升系统整体处理能力的方案。第三种则是在第二种的基础上用以应付数据库压力的典型措施，即传统的数据库镜像方案。

几乎所有的大型数据库系统/产品，以及流行的开源数据库解决方案都提供了数据库后台镜像的能力，通常也提供增量镜像服务。这些系统以及这一典型解决方案事实上工作得很好，但是它也同时带来了大量的存储开销，并且不恰当地为“构建以单一数据库中心为基础”的系统提供了支持。

源于“单一数据库”的数据在总量上的变化，使得我们从传统的“集中式”向“复制式”发展变得越来越不可取。而另一种途径，即从“集中式”向“分割式”过渡的方案开始渐行渐显，也就是所谓的分表分库，如图 53 所示。

图 53 从“集中式”向“分割式”过渡：分表分库的方案



其中，最下层“（总）用户数据库”是一种出于安全性考虑的可选策略。分表分库决定了登录服务与各个子数据库之间存在某种强关系，例如：

- 采用同一的分布策略；
- 使用规则化的方法将分布策略写入到应用端（登录服务）的数据存储逻辑；
- 通过较大的存储过程来将应用端的请求转化为多个表之间的关联查询。

这种强关系带来了更大的系统架构负担，也就是说，我们必须在（包含上述三种方案在内的）多种方案中权衡。但无论如何权衡，都涉及在应用或数据库的逻辑层上的设计，以及由此带来的开发规模不可控。

反思这些变化背后的某些关联，“单一数据库”越来越明显地成为大系统灾难之源。而这是源于早期数据库应用（例如基于数据的三层解决方案）所带来的惯性思维。在我们面临的现实系统中，数据的性质在四个方面几乎在同时发生变化：

（1）数据的碎片化和即时性；

（2）数据向非结构化发展的趋势明显；

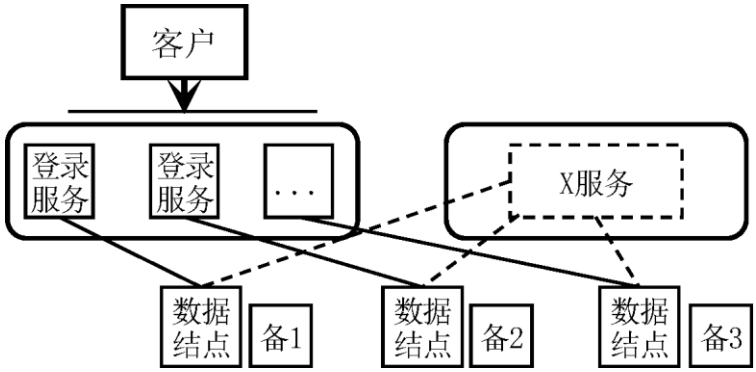
（3）数据项次的量级，即数据笔数随着即时性以及系统处理能力的增强而快速增长；

（4）数据与关联数据的不对等及其总量的规模化，例如一条微博可能附带一个视频文件。

这些数据性质的变化迫使我们在“数据库”之外寻求解决方案。其中，相对重要的原因是，数据库维护中对于“数据结构化”的要求以及实施基于数据库分布方案的代价，两者总的来说与重建一个数据存储系统基本相同。以上述的第四种情况为例，如果我们以前面提到的数据库方案来存储微博，那么附带视频文件的管理就要求对数据库存储与文件存储作统一设计；而当我们采用“镜像数据库”或“镜像文件存储”的方案来解决系统压力时，前端的内容管理与存取接口，乃至于系统整体就几乎要重新设计一番了。

当我们的思考不再局限于“数据库”时，“将数据作为一个系统结点”的思维方式开始走向前台，如图 54 所示。

图 54“数据结点”作为独立系统层次中的可部署对象



在这样的系统模型中，登录服务与其他种种服务并列在应用层，它们与“数据结点”一样作为可部署对象参与系统规划[①]。在这一模型中，系统的分布与并发策略建立在结点之间的 PD-IO 关系之上，而增加“*备*$_1$，…，*备*$_n$”这样的数据镜像，就只是出于对数据结点各自的安全性考量，而非是应付 IO 压力的临时策略了。

20.9 面向数据结点的系统架构

在这样的视角下，我们将越来越趋向于对系统的可组织性与组织方式的观察。

以一个实际的、传统的系统架构为例：对于图 55 所示的整个系统，可以简单地描述为对“逻辑、数据及其交错关系”的求解（图 56）。例如我们将上述模型中的“P-D 关系”理解为某种数据层的提供方案，则可以得到如图 57 所示的模型：

[①] 这样的思维框架是将所有的可部署对象作为结点，而并非唯只是数据结点。

图 55 传统的三层或多层系统架构

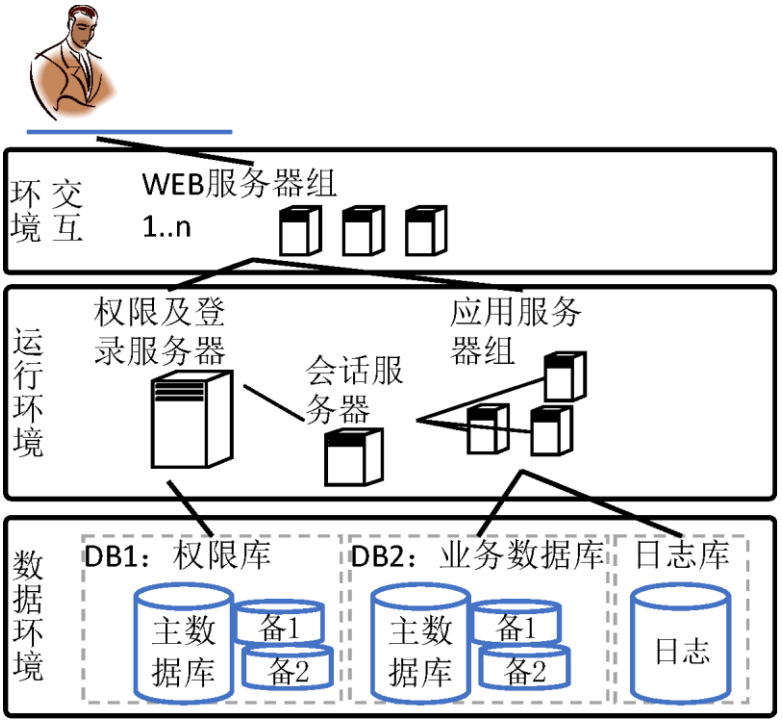


图 56 图 55 中架构的“P-D 关系”模型

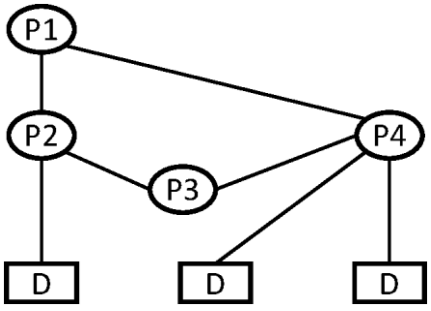
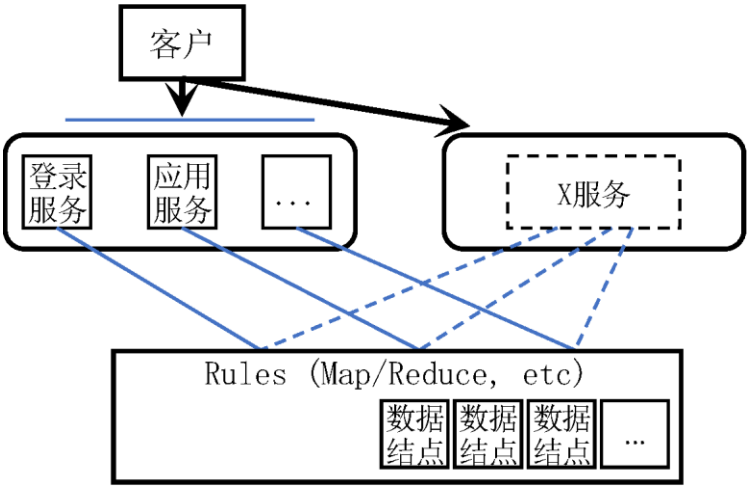
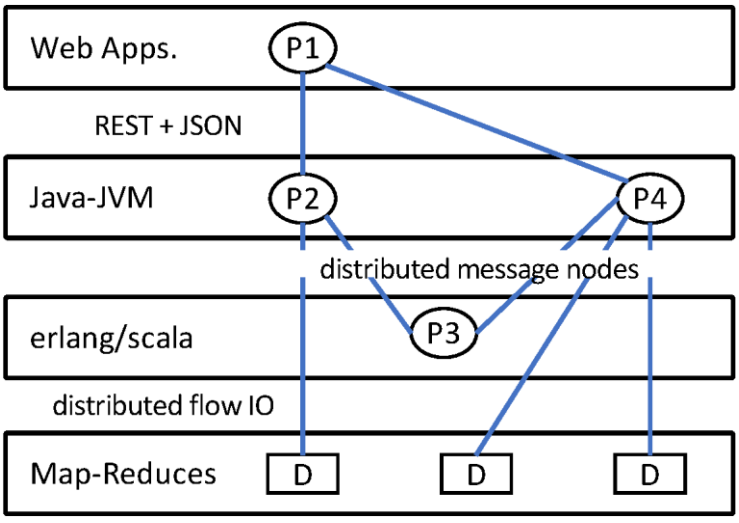


图 57 分布式框架：对界面（对规则的封装）的思考



也就是说，我们将上述模型中的“P-D 关系”理解为某种数据层的提供方案，进而将服务对底层数据的存取变成一组分布逻辑，并将这些分布规则（Rules）通过语言（例如 MapReduce 等）固化在对数据层的存储界面中，并将该界面理解为对规则的封装。如此一来，一种面向底层数据的分布式框架就形成了。类似地，我们也可以规划“P-P 关系”。例如我们可以讨论会话服务是否存有较多的业务逻辑、是否依赖实时响应等限制条件，并据此来选择合适的语言或执行环境。一种可能的情况如图 58 所示。

图 58 分布式框架：领域问题及其方案带来的思考



在这样的模型下，我们讨论的是在不同的层间所采用的领域方案，以及在这些领域交互界面上的可行性、安全性与系统代价。在不同的层间，由于所关注的系统性质不同，因而候选的标准与工具也不同，其数据的格式与存储要求也存有差异，但总体来说，是对 PDIO 四个方面的综合考虑，例如逻辑(P)的复杂度、数据(D)的量级、IO 的频度等。

除开这些在分布、部署、调度等技术细节上的分析与选择，我们要讨论的将是这些层间的规划与层间关系的模型，以及如何通过系统化方法来实现这些层之间，亦即是领域间的协作开发。而这些将部分会是《我的架构思想：基本模型、理论与原则》一书所涉及的内容。

附一：“主要编程范式”及其语言特性关系

> 节选自博客文章“‘主要的编程范式’及其语言特性关系”（2009 年 10 月）。文章对 Peter Van Roy 的“主要编程范式”一图进行了解读，Peter Van Roy 的原文参见：
>
> http://www.info.ucl.ac.be/~pvr/paradigms.html

【一】

Peter 将一个最重要的概念“state”引入进来。而这个 state 也是 Peter 对语言进行分类并考察其变化的主要依据。但 Peter 所使用的 state 概念以及专用名词“cells”都相当地令人困惑。所以在这个图的补充说明中，Peter 对此专门做了解释：“状态是记忆信息的一种能力，更精确地说，是及时存贮值序列的能力。”

你觉得这个概念像什么？对了，的确，非常像是“变量”。事实上，状态在编程核心概念的“数据　逻辑”抽象中，表明的正是“数据的可变性”。也就是说，数据的可变性表现为状态。更进一步地，当数据被命名时，它称为变量或常量；当数据未被命名时，它成为游离的、无名称的、时序含义的存储单元，即“cells”。这也是

Peter 使用“cells”这个专用名词的原因：在本图的讨论中，需要从“变量/常量”这样的概念中，剥离掉“未命名或命名的、确定的或非确定的，以及串行的或并发的”这三个方面的性质。

当不考虑一个存储位置上的命名特性时，它就既非变量或常量，也非某个确定的运算对象（例如“对象”等高级的抽象概念），而只是一种更加泛义的“状态”；同时存储这个状态本身的事物，由于没有位置、时序等概念，所以被称为“cells”。

【二】

《程序设计语言：实践之路》这本书解释过命令式语言的本质特性，即用算法改变数据。如果用两个以上的逻辑（例如两行代码）去影响同一个存储位置（cells），使它的状态改变，并最终在该 cell 产生运算结果，那么它就是一种命令式语言。

再简单一点（但没有上面这样严谨）地说：在程序中不断地重写变量，变量值即是程序的最终结果。所以在本图中，Peter 把这个衍生关系表达为：命令式=纯数据+算法+状态维护。

无论是在串行还是在并发的编程中，命令式编程范式对“状态”的理解都是：共享状态。在串行（例如单线程）的编程中，状态是时序相关的。因为不断地重写“状态（数据/cells/变量）”，所以前一行和后一行所面对都是同一个共享状态的不同的值/副本。在这个过程中，正因为状态与时序相关，所以前一分钟与后一分钟的状态是不确定的。但是在同一时刻，这个状态是确定的。

与上面相类似地，在多线程中，同一时刻，不同线程也将面临这个值/副本。但正是因为多线程（并发）中，线程 A 与线程 B 对于同一个 cell，在同一时刻所得到的状态也是不确定的——我们可以假想

为多核 CPU 在对同一个内存地址读写（于是就出现了我们所谓的“同步”问题，进而也就出现了“锁”的问题）。所以在这个分支中，当加入“线程”概念之后，新的编程范式全都变成了“可观测的非确定性”为“yes”的情况。

【三】

我们显然可以发现，问题出在由于多个线程都在“写 cell”。在命令式的解决方案中，采用的方法是“加锁”；持锁存取的最经济的方法之一是“多读单写”，即保证同时只有一个线程能“写 cell”。

但是这给应用带来了负担。如果一个应用程序有多个线程（分布或不分布在多个 CPU 核上），在它们都要读取同一个 cell 而又有某个线程要写该 cell 时，那么大家就都要被挂起来，直到这个写操作完成。整个应用程序在 CPU 使用（或者说效率）上就大大打了折扣。

如果这只是一个桌面程序（例如记事本），大概没人会说什么。但如果这是个服务器程序（例如 WWW Service），那么整个网络、所有的会话就都处于等待状态了，但同时，服务器的 CPU 占用可能会远远小于 1%！

解决问题的终极方法，就是不解决这个问题。既然写 cell 带来了问题，那么我们就“不写 cell”。我们由前面所有讲述的内容开始倒推，问题根本是由“命令式语言”这个编程范式本身决定的：用算法改变数据。

所以我们回到了原始的问题：如果算法不改变cells（数据/状态/变量）呢？

附二：继承与混合，略谈系统的构建方式

> 节选自博客文章“继承与混合，略谈系统的构建方式”（2010 年 12 月），文章讨论了对“基于对象系统进行系统构建”的认识与实现。

面向对象系统有三种对象的继承方式，即原型、类和元类。这三种方式都可以构建大型对象系统。在后续讨论之前，我们先在名词概念上做一些强调。所谓“对象系统”，是指由“一组对象构成的系统”，这些对象之间存在或不存在某种联系，但通过一些规则组织起来。所谓“面向对象系统”，是指以上述“对象系统”为基础延伸演化的系统，新系统满足前对象系统的组织规则。

所谓“对象系统的三个要素”——继承、封装与多态，即是上述组织规则的要件。孟岩同学从 C/C++出发[①]，从另一个侧面谈论对象系统，所持的观点我相当认可。他指出，“对象范式的基本观念中不包括继承、封装与多态”，这一观点有其确切的背景与思考方法，

[①] 参见孟岩的博客文章“function/bind 的救赎”：
http://blog.csdn.net/myan/article/details/5928531。

值得一谈。

我们在这里要讨论的是“对象系统”，即对象是如何组织起来的问题。在这个问题上，组织规则之一就是“继承”。JavaScript 中基本的继承模型是原型继承，其特点是“新对象实例的特性，复制自一个原型对象实例”。Qomo 以及其他一些项目，通过语言扩展的方式，在 JavaScript 上添加了类继承的模型，其特点是“对象构建自类，类是其父类的一个派生”，这里的“派生”与“特性复制”有潜在的关系，即子类的特性也复制自父类。正是由于“派生”其实是“特性复制”的一种形式，所以事实上 Qomo 中的类继承是通过原型继承来实现的，因为原型继承本质上也就是“特性复制”。

无论是原型继承、类继承还是这里没有进一步讨论的元类继承，继承的最终目的都是构建一个“对象系统”，而不是“系统”。这一个措辞上小小的区别，有着本质上的深刻意义，这也是我提及孟岩的那一篇文章的原因。通常由“继承”入手理解的“对象系统”其实是静态的，以至于我们在面向对象系统开发的最后一步，仍然需要框架来驱动它。例如`TApplication.Run()`，或者类似的`new Application()`等。继承所带来的，主要仍然是指对象系统的组织性，而非其运行过程中的动态特性。

于是我们通过更多类或其他对象系统，来将一个系统的动态特性静态化。例如将对象之间的交互关系抽取出来，变成控制类。我们做这些事情的目的，仅仅是因为我们约定了对象系统的组织规则，要面向这个对象系统开发，也必然满足（或契合）这一组织规则。组织规则限定了我们构建系统的方式——继承、封装与多态，这在一定程度上说是“对象系统构建”的一个方案，并非“系统构建”的方案。而孟岩在文章中所讨论的，正是“系统构建”的问题。所以

孟岩提出两点：

- 程序是由对象组成的；
- 对象之间互相发送消息，协作完成任务。

其中第一条，是对象系统的基本特性，是谓系统成员；第二条，是对象系统如何演进为系统的特性，是谓系统通信。一个系统的约束，既包括其成员（以及成员的组织规则），也包括成员间的通信。

附三：像大师们一样思考——从“UML 何时死掉”谈起

> 节选自博客文章“像大师们一样思考——从‘UML 何时死掉’谈起”（2008 年 10 月），文章是在与“UML 之父”Ivar Jacobson 先生座谈后的反思。

算盘用了几千年，谁问过“算盘为什么能算东西”？算珠、进位、栏，这些东西是不是基本的存储结构？用算盘的“我们”，是不是计算单元？珠算表是不是运算规则？那些珠子表达出来的“0~9”的排列，是不是输入输出的界面？

“我们+算盘”就是一个完整的计算系统。这样的计算系统的完整性，图灵用了一个假想加以说明。图灵不过是用一个假想描述了一个事实，而这个事实“看起来”能被机器实现。于是，我们的计算机时代就开始了。

图灵是否证明过“大笨象吃意大利面条为什么是一个完备的计算机系统”呢？不，最初等的问题，往往最难于证明。往往，他的证明过程，或应用过程，只是触发了一个想象。

对计算机根本问题的思考，许多会追溯到哲学思想层面。IOPD 和 PDIO 的问题、“算法 + 数据结构 = 程序”的问题等，就属于这一类。还有一些会追溯到人类行为学、语言学等层面，例如语言、语法、语义，以及像有没有语用这样的问题。大多数时候，真正推动计算机发展的，不是对具体问题的推理求解，而是对问题本身的抽象。在 Dijkstra 的叙述中，抽象更像是终极武器。按照 Brooks 的观点：

> 数据的表现形式（数据结构，抽象的结果之一）是编程的根本。

而按照 Dijkstra 的引述：

> 引用未解释过的名词阐述公理或定理和作用于未解析过的操作数的（命了名的）运算两者之间有着某种平行的相似性。

无论如何，我们“做一个计算机”，原始的目的不外两个：其一是“让它计算数学”，其二是“让它像人一样思考”。请注意，我的确是说“计算数学（即算数）”。数学是人类的理论学科，“怎么算”以及算的内容等，都是由我们自己设定的。而计算机的能力，只是计算“数学”这个它未知的对象而已。事实上，我们现在讲的“命令式”、“函数式”或“说明式”，只是我们为计算机设定的“最基础的运算方式”。在这个“运算系统”中，“数学”并不是最初设定的。

命令式如何计算，函数式如何计算……诸如此类的问题了解清楚了，我们对这类语言也就了解了。至于什么高阶函数（higher-order function）、克里化（Currying）、延续（Continuation）或发生迭代器（Generator-Iterator）之类，那已经是具体语言的表象，而非“这一类语言”的本质。举例来说，JavaScript 1.5 还没有实现过“生成器对象”（Generator Object），但并没有人否认它是

函数式语言。反过来说，“Generator Object”原本就不是函数式语言的必备要素。

LISP 表达了函数式语言的全部“必备要素”，然而 LISP 七个原子运算也只是针对 LIST 这个结构抽象来说的。对于一个“（顺序的）表”，这七个原子运算是必需的，而对于另一个“（关系的）表”就未必如此了。所以某些原子运算，也不必放在函数式的必备要素中。像 LUA 这样的函数式语言实现方法的出现，也证明了这一点。[①]

那么函数式还剩什么？

要真正理解函数式的秘密，是要一个语言一个语言地学习下去吗？是要一种运算法一种运算法地学习下去吗？我们听完人家说“延续”，于是就开始了解延续，而没有去追问延续为什么出现在函数式里面，或它是不是函数式的必备要素，又或是函数式运算系统的自身的“问题”。我们正是迷失于种种语言和概念的表象，而最终没能像大师一样去思考“计算机不过是大笨象吃意大利面条”这样的抽象层面的问题。

我们要改变的是思想，我们要增强的是能力。大多数人只是增强能力，而不改变思想。这就是我们大多数人不是大师的原因。

[①] LISP 的基础数据结构是索引数组（表，LIST），而 LUA 的基础数据结构是关联数组（表，MAP）。

附四：VCL 已死，RAD 已死

> 节选自博客文章“VCL 已死，RAD 已死”（2008 年 12 月），文章从用户界面开始，最终推进到对象系统界面与组织方式的思考。

当浏览器成为普通用户使用计算设备（包括移动的、桌面的、嵌入的等）的首选时，它便隔离了操作系统与 Web 环境下的 UI。我们没有在任何地方看到一项要求说：一个 Page 必须要有一个跟浏览器 Toolbar 风格相同的工具条，或跟窗体风格相同的菜单。从本质上来说，是浏览器的便捷与普众，催生了 B/S 结构下的应用和服务开发。而这样一来，桌面原生的客户端就不复存在了，C/S 结构的应用渐渐地开始消失，除非在客户端存在较大的运算、逻辑或对计算环境的控制。

重量级的客户端软件越来越少，因为从根底上说，人们不喜欢用复杂的软件。领域的边界，从浏览器编程界面退缩到网络界面。也就是说，浏览器端（Web 客户端、B 端）的开发人员不再要求“能够调用 Win32 API”，而是要求“能够进行网络交互”。而当这一阵线真正推进到面向 socket 的二进制编程时，操作系统就被从这个体系中切割了出去。

Flash Socket 以及 HTML5 中的 HTML Socket 带来了这种趋势，这种

趋势让微软猝不及防。一方面Sliverlight还在为Flash仓促应战，另一方面 IE+JScript 的结构尚未完成六年来最大、最根本的变革（IE8、IE9 或 IE10）。然而即使如此，一个如同操作系统一般庞大的 Web 领域，已然形成。在这个领域中，微软仍在第一战线，且树敌良多。

当我们把 Web 看成一个像操作系统一样的产品平台时，“程序员”便成为产品生成链条中的一环，程序员文化是被重点考虑的对象，但不是全部。包括平面开发人员、设计师、架构师、部署专家、行业分析人员等在内的团队模型必然会建立。小型的 XP 团队仍将存在，但这取决于应付的系统规模，以及在纵向切分上同质性的多少。

横向切分将出现在浏览器端开发的整个过程中，这不但是指整个 UI，还会有 UI 过程中的各个细节，例如框架、数据交互、网络界面等。在这一过程中，纵向切分依然会成为补充。例如将网络界面与数据交互并成一个独立的部分，交由Flash Socket来实现，或交由独立的 comet 兼容层来实现。但更确切地说，横向分层仍会带来更细分领域的繁荣，例如JSON或其他微数据格式，以及其他基于Socket或http/https进行交互的二进制数据格式将成为专门的研究领域。

这其中的原因是，在 B 端带来的领域必然扩大到一个无法通过纵向切分来一次性交付的地步，因而必然在这一端出现更细化的横向分层。从经验来看，当一个领域足够成熟时，就意味着它可以接受横向分层了，正如现在的桌面作为一个领域，可以接受 UC、UCC 以及 NDA 等[①]更为细化的分层一样。

横向切分是领域合作的模式，这也导致横向切分与金字塔式的管理

[①] UC：UI/Client；UCC：UI/Control/Client；NDA：Network/Data/Application。

模型结合时，会存在多领域专家汇聚在金字塔顶端的情况。当这种情况出现时，就需要更高的决策层来应对，这也意味着决策层需要更多的经验和能力。当然，我们仍然会失败，因为即使我们把系统先纵后横地切成网状，我们仍然面临总体规模上的复杂性。同时，管理规模的扩张，也导致我们的成本增加，周期拉长。

所以如果你不是做 3~5 年的规划或者常常被人垢病的“太空项目”，那么你不需要考虑一个问题的全集。你需要关注的是，在某个具体项目中，是否更合适于某些层面的横向分层，并且有意识地培养该层上的开发人员与相关角色。

我认为可以有颠覆性的思想，但从来不指望颠覆性的变革。所以能同时兼容横向分层的后 RAD 时代是漫长的，不过即使是三两年，我想，在 IT 业来说，也算得上是漫长的了。

再接下来，更为迎合这种面向领域组织团队并开发的工具便会出现。但这种工具不再期望整合各个领域的实现技术（注意我不是说“开发技术”），而是提供领域间的交付标准，或者更为直接地提供交付物。更多领域专精的公司受到关注（例如现在的 Macromedia），大厂商开始并购更多的专属领域的公司，以整合他们的业务。更大的平台化产品会出现，远程的、分布的、可迁移的运算理论和解决方案被普及；而与此同时，更细分的领域带来了更多的专属工具和专精人才；项目的整体规模扩张，并由多个团队来实现（像工程的包工队一样）；而单一团队中人员结构更为复杂，但团队规模仍然被保持在 10 人以内，以 15~20 人为上限。类似于测试等技术，将会作为领域而出现，类似于现在建筑业的工程验收与监理也会出现。这些专属领域仍然有它独特的标准、技术与工具，并提供独立的交付物。

对于整个工程来说，RAD 彻底死掉了。也许类似于建筑行业中的沙盘/模型制作公司会出现，并成为产品过程中的一环，但再也没有了在多个横向切分层面上贯通的 RAD 工具。基于原型理论的 RAD 过程，以及迭代的 RUP 会慢慢地退出去。因为如同没有人能够提供一个楼房建筑的迭代过程一样，工程的复杂性已经让人远远不能控制单次迭代的成本。在这种规模级别之下，对层间界面的控制决定了系统的稳定性，以及能否持续开发。因此架构师在全程成为必须，而且架构的职责将更为细分，例如精密到一个数据 IO 界面，可能都需要特定的架构师来确认和论证。同样地，局部的失败或失效将在系统的更早时间内被发现和验证，返工在局部成为常态，但在全程却不复存在——或直接地让整个工程失败掉。

工程越来越依赖于经验，以及由经验带来的技术模型。例如我常常提及的塔楼式建筑，并不是某个工程学家或领域专家头脑风暴的产物，而是在实践中总结的经验。工程的可复制性增强，而复制规模和环境则更趋简单（过于复杂的规则与界面是难于复制的），这一切依赖于横向分层的界面上的简洁性。

模块的复用与重构依然会长期存在，绝不会消亡。更细的复用粒度必然从现在的对象扩展到业务组件复用，但本质上仍然是以无业务逻辑存在的基础对象复用为前提。用户界面（UI）作为组件将会在很长的历史上出现、消亡与重现；更多的层间界面（interface）将以协议标准的形式出现。而在各自独立的层上，基于层间界面的逻辑组件将成批地涌现，“数据”仍然是它们的唯一约束与编程本质。

我们会有对计算模型的新的尝试，但本质仍与如今一致。函数式语言将会走向前台，脚本语言成为领域间的粘合剂（尽管现在在某些领域间已是如此），确定的语言将在确定的层次上大放异彩（或这

也是领域语言的一个代名词）；同时掌握关联的多个层次上的开发工具的专家，以及领域专长的、与程序无关的专家将成为热门。对于整个体系来说，前者主要是实现具体的业务逻辑，而后者则专注于数据定义。不过，可以想见的是，没有多少人会特定用 XML 来作为这些数据定义的载体，专属的数据格式将是层间交互的首选。

最后，从工程实践的角度上来讲，二十年之内，我想不会出现一本名为《软件工程之营造法式》的书。当然，某些噱头制造者或纸张贩售者的吹嘘除外。