第5章

虚拟化与云计算

云计算的含义是云服务商给企业客户提供平台服务，使得企业客户在信息化建设中无须自己考虑诸如机房、网络、服务器之类的基础设施，也无须雇佣业务信息系统运维人员，从而减少信息系统建设和运维的开销。企业客户将自己的业务信息系统迁移至云上运行，交由云服务商运维，不仅能显著降低成本，而且能得到更好的服务质量。对于云服务商来说，通过深化资源共享来提高资源的利用率，通过规模效应和集约效应来降低成本，使得云服务价廉物美。

业务信息系统是应用程序及其数据的总称。常见的业务信息系统有两个应用程序：一个是 Web 应用程序，也叫 Web 服务器；另一个是数据管理应用程序，也叫数据库服务器。每个应用程序包括两部分：①应用程序自身；②运行环境。应用程序依托其运行环境运行。例如，一个 Windows 32 应用程序，其最基本的运行环境为 x86 型号的计算机和 Windows 32 操作系统。具体到 Web 应用程序，它只有在网络正常且能访问到数据库服务器时，才能正常运行。因此，其运行环境还进一步包括网络以及数据库服务器。

传统方式下，先有运行环境，然后构建应用程序。例如，先有 x86 型号的计算机和 Windows 32 操作系统，然后才构建 Windows 32 应用程序。先有网络和数据库服务器，然后才构建 Web 应用程序。也就是说，应用程序是基于运行环境来设计开发和部署。很多情形下，甚至假定运行环境具有静态不变性。例如，对要访问的数据库服务器，将其 IP 地址当作常量处理。对于已经构建好的应用程序，现要将其迁移至云上运行，这就需要云平台为其提供运行环境。5.1 节将详细讲解应用程序与其运行环境之间的耦合关系。

当要运行一个应用程序时，先要为其构建好运行环境。传统构建方式是：先选定计算机，然后在其上安装诸如操作系统之类的支撑软件，再进行安装和配置。云计算提出了新要求：在资源共享的前提下进行运行环境的构建。具体来说，就是要使一个应用程序能在云上到处运行；要使一台计算机上运行的多个应用程序彼此隔离，不会相互干扰。该目标的达成，不能要求应用程序适应云平台，只能是云平台适应应用程序。其原因是：客户的应用程序通常都已经定型固化，不可能对其进行改动，使其适应云平台。解决问题的方法是虚拟化。

要使一个应用程序能在云上到处运行，面临诸多问题。首先是程序语言的差异性问题。应用程序与语言关联，计算机也与语言关联。当与计算机 A 关联的语言，与应用程序 α 关联的语言不一样时，应用程序 α 就不能在计算机 A 上运行。要使计算机 A 能运行应用程序 α，一种解决方法是在计算机 A 上安装一个虚拟机。典型的例子是：要使计算机 A 能运行 Java 应用程序，就要在计算机 A 上安装一个 Java 虚拟机。5.2 节将讲解应用程序的运行方

式，展示虚拟机的实现方法，使得一台计算机支持多种语言。

仅使计算机能运行应用程序，还远远不够。还有一个更深层次的问题，即能否成功运行的问题。应用程序迁移上云之后，其运行环境中的资源不再由一个应用程序独享，而是由多个应用程序共享。服务也不再处于静态不变状态，这就带来了问题：资源对应用程序不一定可用；服务 id 可能动态变化。这两种情形都会使得应用程序不能成功运行。解决该问题的方法还是虚拟化，这种虚拟化是将运行环境虚拟化。具体来说，就是在应用程序与操作系统之间，嵌入虚拟机。使应用程序不再直接运行在操作系统之上，而是在虚拟机之上。虚拟机为应用程序提供一个不变的虚拟运行环境。5.3 节至 5.4 节将讲解导致应用程序运行不成功的原因，以及解决问题的策略和方法。

将一个物理资源虚拟成多个物理资源来使用，是虚拟化的一个方向。将多个物理资源当成一个物理资源使用则是虚拟化的另一个方向。前一种虚拟化是分解共享，后一种虚拟化则是组合共享。集群计算和云计算都将多台计算机组合成一台计算机使用。这种组合既有不变性，又有变化性。不变性体现在客户使用系统的方式保持不变。变化性体现在系统的服务质量上。在客户看来，集群和云具有全新的特质：存储容量无穷大，吞吐量无穷大，响应及时，时刻可用，性价比高。5.15 节将讲解云服务管理系统。

5.1　应用程序与其运行环境

应用程序是用程序语言写出的代码。程序语言分为低级程序语言和高级程序语言。机器语言就是低级程序语言。每种型号的计算机都有自己的机器语言。高级程序语言与计算机型号无关，具有通用性。高级程序语言可分为两类：①编译运行类语言，如 C/C++ 语言；②解释运行类语言，也叫脚本语言，例如 Shell 脚本语言和 JavaScript 脚本语言等。对于用编译运行类语言编写的应用程序，就其运行而言，先要选定其运行环境，也就是确定要用哪一种型号的计算机运行它。被选定的型号计算机被称作目标计算机。然后用相应的编译器将其编译成目标代码。如果想要应用程序在多种型号的计算机上都能运行，就要为每种型号的计算机都编译一次，生成对应的目标代码。只有目标代码才能在目标计算机上运行。

说到应用程序，就离不开程序语言。说到程序语言，就离不开运行应用程序的计算机。这里的计算机不一定指硬件。例如，用脚本语言写出的应用程序，就是用解释器运行。解释器就不是硬件，而是软件。Java 字节码用 Java 虚拟机运行。Java 虚拟机也不是硬件，而是软件。当运行应用程序的计算机不是硬件而是软件时，通常就将其称作虚拟机。于是，用 Shell 脚本语言写的应用程序，运行它的解释器便可称作 Shell 虚拟机。虚拟机本身是一个应用程序。虚拟机用来运行应用程序。这表明可用应用程序运行应用程序。这种迭代性正是软件的一个显著特质。

应用程序的运行离不开运行环境。现通过举例说明运行环境的含义。BIOS 是一个应用程序，只要计算机一通电，就被 CPU 运行。其功能是检测计算机硬件，然后将操作系统从磁盘加载至内存，再让 CPU 跳转去执行操作系统的首条指令。BIOS 的运行只需计算机裸机，不依赖于其他任何软件。因此，BIOS 的运行环境就为计算机裸机。除 BIOS 外，很多嵌入式应用程序的运行环境也是计算机裸机。它们和 BIOS 一样，被烧制到 ROM 内存中，只要计算机一通电，就被 CPU 运行。操作系统是软件，也可称作一个应用程序。操作系统

依赖于 BIOS 将其从磁盘加载至内存，并让 CPU 跳转去执行其首条指令。因此，操作系统的运行环境包括 BIOS。

运行环境是指应用程序运行所依赖的前提条件。常见的应用程序，其运行环境是操作系统。应用程序靠操作系统将其从磁盘加载到内存，然后让 CPU 跳转去执行应用程序的首条指令。另外，应用程序要调用操作系统的 API，从这点看，应用程序也依赖于操作系统。用脚本语言编写的应用程序由解释器运行，因此它的运行环境为解释器。解释器也是一个应用程序，它的运行环境通常为操作系统。

运行环境也可视作程序正常运行所需的资源以及应满足的条件。资源既有硬件资源，也有软件资源，有时还包括数据。例如，数据库管理系统这一应用程序，对分配给它使用的内存大小就有要求，不能低于某一阈值。当一个 Web 应用程序要访问数据库服务器时，那么数据库服务器就成了它的运行环境。当 Web 应用程序指定要访问 MySQL 数据库时，那么它的运行环境中，数据库服务器必须是 MySQL 服务器才行。如果 Web 应用程序把 MySQL 数据库的 IP 地址作为常量写在源代码中，那么它对运行环境的要求就更苛刻了：MySQL 服务器的 IP 地址必须为它指定的值。

应用程序对其运行环境的约束和限制越多，它的通适性就越差。以上述 Web 应用程序为例，如果它固化了 MySQL 服务器的 IP 地址，那么当 MySQL 服务器的 IP 地址无法设置成它指定的值时，Web 应用程序就无法正常运行。应用程序是基于运行环境来设计开发和部署的，因此在确定运行环境时，要考虑运行环境的可构建性和易构建性。还是以上述 Web 应用程序为例，如果在设计开发时将 MySQL 服务器的 IP 地址设置成一个可配置参数，那么其运行环境的可构建性就明显增强。如果进一步使用 ODBC/JDBC 驱动来连接数据库服务器，并使用 SQL 与数据库服务器交互，那么要求数据库服务器为 MySQL 服务器这一限制也就取消了。如果 Web 应用程序使用 Java 语言开发，那么其运行环境就与操作系统无关了。

云计算中，要求客户的应用程序能在云上到处运行。要达成此目标，可从两方面努力：①应用程序在设计开发时，就其运行环境而言应尽量减少约束和限制；②提升云服务商对运行环境的构建能力，以满足应用程序的运行需求。这两方面的思路都是虚拟化。在应用程序的设计开发中，应从物理概念中归纳和抽象出逻辑概念，然后基于逻辑概念编写代码，避免在代码中出现物理概念。把逻辑概念到物理概念的映射留待运行时处理。例如，对数据库服务器的 IP 地址，应在程序中用变量表达，不要用常量表达。变量的值留待运行时再从配置文件中读取。在运行环境的构建中，则将物理资源抽象成虚拟资源，以此满足应用程序对资源的需求。

云计算中，要求应用程序能到处运行。给定应用程序 α 和计算机 β，运行涉及 2 个层面的问题：①计算机 β 能不能运行应用程序 α；②应用程序 α 在计算机 β 上能不能成功运行。能不能运行的问题要从程序语言说起。如果应用程序 α 是用程序语言 A 编写的，而计算机 β 又支持程序语言 A，那么计算机 β 就能运行应用程序 α。如果计算机 β 不支持程序语言 A，而是支持程序语言 B，那么计算机 β 就不能直接运行应用程序 α。要使计算机 β 能运行应用程序 α，有两条途径：①对应用程序 α 进行编译，使其适应计算机 β；②在计算机 β 上安装一个解释执行器，使得计算机 β 支持程序语言 A。5.2 节将讲解这两种策略的具体实现。

5.2　应用程序的编译运行与解释运行

程序语言有高级程序语言与低级程序语言之分。无论是高级程序语言，还是低级程序语言，它们都多种多样。高级程序语言由于具有灵活、简洁、通俗、通用特点，因此被用来编写应用程序。应用程序的特点是：开发者与用户彼此相互独立。一个应用程序只有一个开发者，但有很多用户。用户不同，其计算机对语言的支持能力也不同。一台计算机裸机仅支持其自己的机器语言。如果还安装了 Java 虚拟机，那么这台计算机还支持 Java 语言。如果还安装了 C/C++ 编译器，那么它就进一步支持 C/C++ 语言。一台计算机对自身机器语言之外的其他语言提供支持时，要付出代价。代价来自两方面：①支持软件本身要占用计算机资源；②执行语言之间的翻译也要占用计算机资源。

语言多样性问题在日常生活与工作当中也存在。很多重要的文献都是外文。假设用语言 A 写的文献要员工 α 处理，如果员工 α 不懂语言 A，仅懂语言 S，那么就需找一个既懂语言 A 也懂语言 S 的人将文献翻译成用语言 S 写的文献，然后将翻译后的文献交给员工 α。这种处理在计算机上就叫编译运行。准确地说，叫编译后运行。另一种方法是给员工 α 配置一台翻译机 A，让他借助翻译机 A 搞懂文献，然后再处理文献。这种处理在计算机上就叫解释运行。这类似于在计算机上安装一个 Java 虚拟机软件，以便能运行 Java 程序。如果还要叫员工 α 处理用其他语言表达的文献，还得给其再配置另外的翻译机。员工 α 背负的翻译机越多，负担就越重；每处理一次任务，都要做一遍翻译，工作效率低。

传统的编译，其含义是指：将用高级程序语言表达的程序（也叫源程序）翻译成用某种机器语言表达的程序（也叫目标程序）。目标程序可以在对应型号的计算机上直接运行。程序通过编译后再运行，该方式称为编译运行方式。解释执行方式则指：对用某种语言编写的应用程序，使用一个针对该语言的解释器运行。解释器是一个应用程序。编译运行和解释运行这两种方式各有优点和缺点。接下来将从高级程序语言的特性分析入手，讲解编译运行和解释运行的实现方法，然后分析和归纳它们各自的特性，探讨如何取长补短。

5.2.1　高级程序语言的特性

用高级程序语言编写的应用程序具有良好的可读性、可维护性、通用性、可重用性。高级程序语言的可读性强，体现在用名称标识变量。与其相对，低级程序语言使用地址标识变量。除此之外，高级程序语言还提供了丰富的特征表达方式。例如，分支语句的表达不仅有 if，还有 switch、while、for 等。当一个变量有多种取值，而且取值具有离散性时，可用 switch 表达。当循环迭代次数要到运行时才确定时，则用 while 表达。于是用高级语言编写的程序具有良好的可读性，让人一看就一目了然。

用高级程序语言编程，还可任用结合律、交换律、优先级等法则，使代码的表达尽量与习俗相符，与事物的实际情形一致。例如，高级程序语言中的语句行“y = a * x^2 + b * x + c;”，让人一看就知道它是求抛物线上 y 坐标值的计算表达式。如果用低级语言表达这个计算表达式，就会有 5 行代码，依次为：①t1 = x * x；②t2 = a * t1；③t1 = b * x；④t2 = t2 + t1；⑤y = t2 + c。这 5 行代码的可读性显然很差。由此可知，用高级程序语言写出的代码行在字面上表达的是一个完整概念，可读性良好。与之相对，用低级程序语

言写出的代码行，其特点是每行都短小，但行数很多，可读性差。

用高级程序语言编程，可通过抽象化处理，使程序具有良好的通用性和广适性。典型的例子是将所有的输入和输出设备抽象成文件。于是，输入时统一都是对文件调用 read 函数，输出时统一对文件调用 write 函数。调用时传入的文件描述符参数不同，应用程序的表现行为就不同。例如，当调用 write 函数输出数据时，如果传入的文件描述符参数为显示器文件的 id，那么数据就会输出到显示屏上；如果传入的文件描述符参数为普通文件的 id，那么数据就会输出到文件中；如果传入的文件描述符参数为打印机文件的 id，那么数据就被打印出来。用高级程序语言编写的应用程序具有通用性，只需一次编程。想要其在某种型号的计算机上运行，用编译器将它编译成该机型支持的目标代码即可。

5.2.2 源程序的构成特性

编译是指源程序语言到目标程序语言之间的翻译，具体来说，是将用源语言编写的程序翻译成用目标语言编写的程序。对于编译器来说，它的输入是一个包含源代码的文本文件，它的输出是一个包含目标代码的可执行文件。源文件是一个字符流，也称作字符序列。对输入中的字符序列进行切分，便可得到一个词序列。例如，代码 5.1 所示源代码中第 3 行“area = 0;”，便可依次切分出“area”，“=”，“0” 和 “;”这 4 个词。其中的第 1 个词 area，是由 ‘a’，‘r’，‘e’和‘a’这 4 个字符连接而成。该行代码是一个赋值语句，由 4 个词串接而成。对输入中的词序列进行切分，便可得到一个语句序列。代码 5.1 中整个内容为一个函数实现语句，它包含了一个由 5 个语句组成的语句序列。该语句序列中的第 5 个语句为 for 语句。该 for 语句又包含了一个由 2 个语句组成的语句序列。

代码 5.1　C 语言源代码示例

```
(1)  float area (float xStart, float xEnd)  {
(2)    float area, stepLen, x;
(3)    area= 0;
(4)    stepLen = (xEnd - xStart) / 100;
(5)    x = xStart;
(6)    for (int i = 0; i < 100; i++)  {
(7)      area = area + fun(x) * stepLen;
(8)      x = x + stepLen;
(9)    }
(10)   return area;
(11) }
```

由此可知，从构成关系看，程序由语句构成，语句由词构成，词由字符构成。为了翻译，在语句与词之间通常还引入了一些中间结构体概念。例如，运算表达式就是介于语句与词的一个中间结构体概念。代码 5.1 中第 4 行出现的“(xEnd − xStart) / 100”就是一个运算表达式。中间结构体概念通常综合考虑高级程序语言特性和目标语言特性，为实现翻译而定义。于是语句的构成元素除了词之外，还有中间结构体、语句、语句序列。语句中含有中间结构体的例子如代码 5.1 中第 7 行的赋值语句，等号右边的“area + fun(x) * stepLen”是一个运算表达式。语句中含有语句序列的例子如代码 5.1 中第 6 行的 for 语句，它包含了一个由 2 个语句构成的语句序列。由此可知，语句在构成上不再是线性结构，而是树状结

构。程序由语句构成，自然也是树状结构。

5.2.3　编译过程与方法

在高级程序语言中，词是程序的最小构成元素。对于词，每门高级程序语言都定义了自己的构成法则。例如，在 C 语言中，变量只能由下画线、字母、数字 3 种字符构成，而且首字符不能是数字，这就是有关变量的构成规则。词的构成法则也叫词法。编译器依据词法对输入的字符序列进行分析，将其切分成词序列，这就叫词法分析。词法分析是编译器要做的第一项工作。

编译器要做的第二项工作是依据语法对词序列进行切分，构建出源程序的语法分析树。例如，代码 5.1 中第 7 行是一个赋值语句。这个赋值语句的语法分析树如图 5.1 所示。这棵语法分析树的树根是符号 S，S 表示语句(Statement)。由 S 的子结点可知，这是一个赋值语句。赋值语句由 4 个元素构成，分别是被赋值变量 V(variable)、等于号(=)、运算表达式 E(Expression)，以及分号(;)。再来看被赋值变量 V，由它的子结点可知，它由一个变量(用 id 表示)构成，这个变量为 area。再来看运算表达式 E，由它的子结点可知，它是一个加法运算表达式的运算结果。加法运算表达式由 2 个操作数做相加运算。这两个操作数也是其他运算表达式的运算结果，因此也是 E。

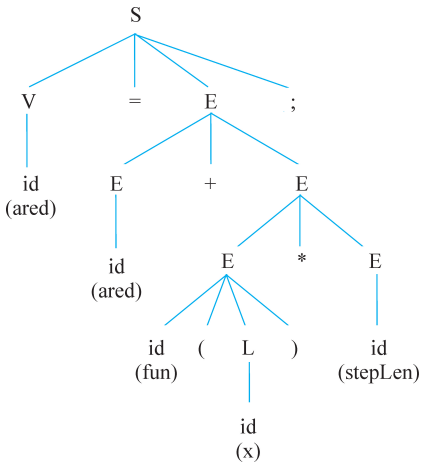

图 5.1　赋值语句“area = area + fun(x) * stepLen;”的语法分析树

一个变量能算作一个运算表达式，它的值就是运算表达式的值。因此，第 1 个操作数 E 的子结点为一个变量，这个变量为 area。第 2 个操作数由它的子结点可知，它是一个乘法运算表达式的运算结果。乘法运算表达式由 2 个操作数做相乘运算。其中第 1 个操作数当然也是一个运算表达式 E，由它的子结点可知，它是一个函数调用的结果。函数调用由 4 个元素构成，分别是函数名 id、左括号、参数列表 L，以及右括号。在这里，函数名为 fun。从参数列表 L 的子结点看，它由一个变量构成，这个变量为 x。乘法运算的第 2 个操作数也是一个运算表达式 E。由它的子结点可知，它由一个变量构成。这个变量为 stepLen。

观察这棵语法分析树，可知：①非叶子结点都是诸如语句(S)、运算表达式(E)等之类的结构体概念，这些概念都有自己的构成语法；②叶子结点则都是词。将树的所有叶子结点从左到右串起来，就是输入的词序列，即“area = area + fun(x) * stepLen;”。另外，由语

法分析树可知，尽管输入的词序列中，加法运算写在前面，乘法运算写在后面，但乘法运算执行在加法运算的前面，实现了计算顺序的调转。原因是做加法运算的 2 个操作数中有一个是乘法运算的结果。运算的优先级通过语法表达。

从这个例子可知，语法分析的基本策略就是先定义结构体概念，例如程序(P)、语句(S)、运算表达式(E)以及变量列表(V)之类，然后看每个概念分别又有多少种构成情形，即多少个类别。再归纳出每个类别的构成语法。例如，语句就有变量定义语句、赋值语句、for 语句等类别。表达式有加法运算表达式、乘法运算表达式等类别。对于变量定义语句，它由数据类型、变量列表、分号 3 个元素构成。通过归纳和抽象得出程序的语法构成规则，再用其对程序的词序列进行匹配，构建出程序的语法分析树，这就是语法分析的策略。

程序的构成法则除上述词法规则和语法规则外，还有语义规则。语义规则的含义现在通过举例说明。在 C 语言中，指针变量不能执行乘除运算，也不能和整数之外的其他类型数据做加减运算，这就是一条语义规则。break 语句只能用在循环语句中，或者 switch 语句中，这是另一条语义规则。函数的使用要和函数的定义一致，这也是一条语义规则。

语义规则和语法规则是 2 个完全不同的概念。语法表达构成，而语义表达约束。语法描述的是某个结构体的构成，包括构成元素以及元素之间的顺序。例如，赋值语句这一结构体由变量列表 V、等于号(=)、运算表达式 E，以及分号(;)这 4 个元素串接而成。其中等于号(=)和分号(;)是词，即基本元素。而变量列表 V 和运算表达式 E 都是中间结构体，它们各有自己的构成语法。与其相对，函数的使用和函数的定义相一致，则是语义规则。其含义是：语法分析过程中，一看到函数调用这一结构体出现，就要检查前面的函数定义记录，首先看该函数是否已定义。如果定义了，则进一步检查函数调用中的参数个数、顺序、类型是否与函数定义一致。如果未定义或者不一致，就认为该程序有语义错误，不符合语义规则。

编译器要做的第 3 项工作，就是确保输入的源程序没有违背语言中定义的所有语义规则。编译器做语义检查时，一旦发现输入的源程序违背了语义规则，就会报错，停止编译，要求程序员对源程序进行改正。当然，在第一项工作中发现有词法错误，或者在第二项工作中发现有语法错误时，也会报错，也要求程序员对源程序进行改正。在具体实现中，语义分析通常并不是一个独立的环节，而是在语法分析的过程中附带完成。

对输入的源程序，编译器构建出其语法分析树后，接下来的第 4 项工作是生成中间代码。按理来说，有了程序的语法分析树，就可直接生成目标机器代码。但是通常并不这么做，而是生成中间代码，然后再把中间代码翻译成目标机器代码。也就是说，由语法分析树到目标机器代码是一个大台阶，一步攀越不太好。在这个大台阶的中间再设一个中间代码台阶，将一个大台阶变成 2 个小台阶，这样做具有更好的可操作性。

为了阐释设置中间代码环节的好处，现举一个类似的例子。世界上有很多语言，例如汉语、俄语、德语、日语等。在计算机界也是如此，有很多种高级程序语言，也有很多种机器语言。人类语言之间的翻译通常不是直接将源语言翻译成目标语言，而是将源语言翻译成英语，再将英语翻译成目标语言。采用这种模式，好处是：任何一个国家的翻译工作者，只掌握好本国语言与英语之间的翻译即可。任选两国语言，假定为 A 和 B，做其翻译只需在 A 国找一个翻译，在 B 国找一个翻译，通过 2 步翻译完成。这样做的可操作度、可靠度、质量、效率、成本，远比找一个既懂 A 国语言又懂 B 国语言的翻译做要好得多。程序语言的翻译也是如此。

设置了中间语言，能使程序的编译和运行更加灵活。用中间语言表达的程序代码，既可用解释器解释运行，也可进一步编译成目标代码。另一个好处是：用任何一门高级语言编写的程序，都只需经过两步编译，便能得到任何一种计算机型号的目标代码，使得高级程序语言与机器语言两者之间既彼此相互独立，又能对接组合。

源代码到中间代码的翻译，是在语法分析的过程中完成的。语法分析对词法分析所得的词序列从头到尾逐一扫描，拿其与语法规则匹配，以此发现结构体的完整实例。结构体是语法中定义的概念，例如加法运算表达式、函数调用、变量定义语句、赋值语句、if 语句、函数定义语句等都是结构体。语法分析中，一旦某种结构体的完整实例出现，便基于该结构体的含义得出其中间代码。

以图 5.1 所示的语法分析树为例，首先出现的结构体完整实例是函数调用 fun(x)，于是根据函数调用的含义得出其中间代码，见代码 5.2 中第 1 行和第 2 行。其中第 1 行的含义是创建一个类型为 float(其类型 id 为 1)的临时变量，用来临时存储函数调用的返回值。该创建操作的返回值为被创建的临时变量的序号。第 2 个出现的结构体完整实例是乘法运算表达式，于是根据乘法运算表达式的含义得出其中间代码，见代码 5.2 中第 3 行。第 3 个出现的结构体完整实例是加法运算表达式，于是根据加法运算表达式的含义得出中间代码，见代码 5.2 中第 4 行。第 4 个出现的结构体完整实例是赋值语句，于是根据赋值语句的含义得出其中间代码，见代码 5.2 中第 5 行和第 6 行。第 6 行的含义是释放序号为 0 的临时变量。

代码 5.2　赋值语句“area ＝ area ＋ fun(x) * stepLen;”的中间代码

```
(1)  NewTempVariable 1;
(2)  tmp:0 = call function:2 local:2
(3)  tmp:0 = floatMultipleWithVarToVar tmp:0 local:1
(4)  tmp:0 = floatAddWithVarToVar local:0 tmp:0
(5)  local:0 = floatVarAssign tmp:0
(6)  freeTempVariable 0
```

代码 5.2 所示中间代码中的“NewTempVariable”，“tmp：”，“＝”，“function：”，“local：”，“floatMultipleWithVarToVar”，“floatAddWithVarToVar”，“floatVarAssign”，“freeTempVariable”都是中间语言中的关键字。其中 floatMultipleWithVarToVar 的含义是两个 float 类型的变量执行乘法运算，而 floatVarAssign 的含义则是把一个 float 类型的变量值赋给另一个 float 类型的变量。在中间语言中，变量不再用名称标识，而是用逻辑地址标识。第 2 行中的“function：2”是函数 fun 的逻辑地址，其含义是：该函数是程序所有函数中序号为 2 的函数。“local：2”是局部变量 x 的逻辑地址，其含义是：在函数的局部变量中，该变量是序号为 2 的变量。“tmp:0”表示序号为 0 的临时变量。

将源代码翻译成中间代码，会将代码和数据进行分离。于是中间代码由 2 部分组成：①代码表；②符号表。代码表中记录中间代码，每行中间代码用行号标识。符号表中记录程序中定义的数据类型、变量、函数、字符串常量。符号表是一个总称，它包含 4 种表：①数据类型表；②函数表；③变量表；④常量表。由于变量又分 5 种：①全局变量；②局部变量；③成员变量；④形参；⑤临时变量，因此变量表又有 5 种：①全局变量表；②局部变量表；③成员变量表；④形参表；⑤临时变量表。在语法分析的过程中，每遇到一个新的数据

类型定义，就将其添加到数据类型表中；每遇到一个新的函数定义，就将其添加到函数表中。以此类推。对于数据类型、变量、函数等，每遇到其使用时，都要到符号表中查找其定义，检查是否符合语义规则，得到其逻辑地址，以备中间代码生成之用。

类型表、函数表、局部变量表中的内容示例分别如表 5.1～表 5.3 所示。类型、函数、变量都用序号（也叫行号）标识。在中间代码中，数据和函数都用逻辑地址标识。例如，逻辑地址“function：2”标识一个函数，其详细信息记录在函数表中序号为 2 的行中。而逻辑地址“local：1”标识一个局部变量，其详细信息记录在所在函数的局部变量表中序号为 1 的行中。类型有基本数据类型和自定义数据类型之分。基本数据类型有 int，float 和 char，假设其类型序号分别为 0，1 和 2，如表 5.1 所示。每个自定义类型（例如 C 语言中的 struct 和 C++ 语言中的 class）都有其成员变量，记录在成员变量表中。另一种类型为指针类型。对于指针类型，有引用类型概念。例如表 5.1 中所示类型例子中，就有 char * 和 Student * 这两种指针类型，其引用类型的 id 分别为 2 和 4。

表 5.1　类型表

序号（index）	类型名称（name）	引用类型的 id（referenceTypeId）	类型宽度（width）
0	int		
1	float		
2	char		
3	char *	2	
4	Student		
5	Student *	4	

表 5.2　函数表

序号（index）	函数名称（name）	返回值的类型 id（typeIndex）	首行代码的序号（startIndex）	形参的总宽度（formalTotalWidth）	局部变量的总宽度（localTotalWidth）
0	main	0	0		
1	area	1	67		
2	fun	1	198		

表 5.3　局部变量表

函数 id（functionIndex）	序号（index）	变量名称（name）	类型 id（typeIndex）	宽度（width）	偏移量（offset）
1	0	area	1		
1	1	stepLen	1		
1	2	x	1		
1	3	i	0		

对于类型和变量，都有宽度（即在内存中所占字节数）概念。自定义类型的宽度为其成

员变量的宽度之和。变量的宽度为其类型的宽度。知道了变量的宽度，在运行时就知道要为其申请多少内存。类型和变量在中间语言中只有宽度概念，但没有具体值。宽度值要等到目标计算机确定之后才填写。其原因是：诸如 int，float 之类的基本数据类型的宽度随计算机型号不同而不同。例如，float 类型的宽度在 16 位计算机中为 4，在 32 位计算机中为 8，在 64 位计算机中为 16。

一旦目标计算机确定，那么基本数据类型的宽度就已知，于是自定义数据类型的宽度就能算出。变量的宽度为其数据类型的宽度，于是也为已知常量。现假定目标计算机是 32 位计算机，那么 int 类型，float 类型，指针类型的宽度分别为 4，8，4。对于表 5.3 中所示的 4 个局部变量，其宽度分别为 8，8，8，4。程序在运行时，无论是变量还是函数，都用内存地址标识。内存地址又由基地址和偏移量(offset)两部分构成。对于一个函数的局部变量，第 1 个变量的 offset 值为 0。第 1 个变量的 offset 值加上第 1 个变量的宽度就为第 2 个变量的 offset 值，其他以此类推。于是表 5.3 中所示的 4 个局部变量，其 offset 值分别是 0，8，16，24。其他类型的变量也是如此。

将中间代码翻译成目标代码时，要将逻辑地址全部转换成内存地址。每个程序和每个函数，其实都可视作一种数据类型。于是全局变量就是程序这一数据类型的成员变量，局部变量就是函数这一数据类型的成员变量。运行一个程序时，就要创建程序这一数据类型的实例对象；调用一个函数时，就要创建被调函数这一数据类型的实例对象。实例对象的内存地址为其成员变量的基地址。基地址加上成员变量的 offset 值，就为成员变量的内存地址。offset 值也叫相对地址。当目标计算机支持相对寻址时，使用寄存器存储变量的基地址，就能使目标代码中出现的内存地址都为常量。

从上述举例和分析可知，中间代码已经接近机器代码了。中间代码中的每行都表示一项操作(也叫运算)。每行中间代码都很短小，但行数很多。不过，在中间代码中，存储器只有一种类型。而在物理计算机中，存储器有寄存器和内存之分。另外，中间代码的可读性很差，其原因是函数、变量、字符串常量不再用名字表达，而是用逻辑地址表达。

5.2.4　中间代码的优化

从 5.2.3 节编译过程与方法可知，源代码到中间代码的翻译是基于结构体含义(即语法)的一种机械式翻译，没有考虑上下文彼此之间的联系。这种翻译得出的中间代码具有可优化性。下面举例展示这种翻译特性。代码 5.3 是功能为矩阵转置变换的源代码中的一个片段，其中局部变量 a 为一个二维数组，假定其定义为 int a[10][20]。在中间代码中，数据要用其逻辑地址表达。地址空间是线性空间。数组元素是数据，因此在中间代码中也要用其逻辑地址表达。假定数组元素 a[i，j]的逻辑地址用 a[x]表达，其中的 a 表示数组变量 a 的逻辑地址，x 为另一个变量的逻辑地址，其值为数组元素 a [i，j]相对数组 a 的起始地址的偏移量(offset)，单位是字节，于是有 x ＝ (20 * i ＋ j) * WIDTH (int)，其中 WIDTH (int) 表示整数的宽度。

代码 5.3　源程序片段

```
(1)  k = a [i, j];
(2)  a [i, j] = a [j, i];
(3)  a [j, i] = k;
```