COBOL——大型机商业编程技术详解（修订版）

马千里　编著

人 民 邮 电 出 版 社
北　京

图书在版编目（CIP）数据

精通COBOL : 大型机商业编程技术详解 / 马千里编著. -- 2版（修订本）. -- 北京 : 人民邮电出版社, 2011.3
ISBN 978-7-115-24646-2

Ⅰ. ①精… Ⅱ. ①马… Ⅲ. ①COBOL语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2010)第243871号

内 容 提 要

COBOL 是应用于大型机开发的主要程序设计语言。本书由浅入深，循序渐进地介绍了如何使用 COBOL 语言以及与 COBOL 语言紧密相关的扩展技术进行实际开发。全书共 18 章，主要内容包括 COBOL 简介、程序结构、常用语句、基本数据类型、字符串及其操作、基本运算、流程控制、数据的排序与合并、COBOL 中的表、程序的调试与测试、子程序调用、COBOL 中的面向对象技术、处理 VSAM 文件、JCL 扩展、DB2 扩展、CICS 扩展、大型机汇编语言扩展、开发小型银行账户管理信息系统等。

本书适合广大 COBOL 程序开发人员、大型机培训班学员和大专院校学生阅读，尤其适合具有一定 C 语言编程基础的人员进行学习。

精通 COBOL——大型机商业编程技术详解（修订版）

◆ 编　　著　马千里
责任编辑　汪　振

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号
邮编　100061　　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
中国铁道出版社印刷厂印刷

◆ 开本：787×1092　1/16
印张：29.75
字数：718 千字　　2011 年 3 月第 2 版
印数：4 601 – 7 600 册　　2011 年 3 月北京第 1 次印刷

ISBN 978-7-115-24646-2

定价：30.00 元

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154

前　言

COBOL 是用于大型机开发的程序设计语言。COBOL 语言主要的特点是面向高端商业用途，是大型商用应用程序开发的首选编程语言。同时，大型机上的许多其他软件产品，如 DB2 数据库、CICS 中间件等，也都是以 COBOL 作为宿主语言的，需要通过 COBOL 进行调用和交互。使用 COBOL 语言开发的程序广泛应用于银行业、保险业、制造业、航空业等。COBOL 语言所开发的软件具有良好的稳定性、安全性，以及强大的并行处理海量数据的能力，因此备受金融部门的青睐，沿用 40 多年仍未被取代。并且，随着国际外包业的发展，全球大量 COBOL 开发的职位正在越来越多地涌入国内。然而由于以前在国内 COBOL 开发主要只应用于银行，相关从业人员多采用的是内部培养的方式，因此了解 COBOL 的人并不多。当前，COBOL 从业人员供不应求，COBOL 相关职业十分走俏。

目前市面上有关 COBOL 书籍多为外文书籍，中文书籍较少。虽然在网上有少数几个关于 COBOL 方面知识的网站和论坛，但所介绍的内容都比较零散，没有全面系统地对 COBOL 进行讲解。即使是此前极少数的几本关于 COBOL 的中文书籍，也由于时间间隔久远已在市面上看不到。并且，这些书籍里多是单纯地对 COBOL 进行介绍，没有涉及到同 COBOL 紧密相关的内容，如 JCL、DB2、CICS 等。这些内容实际上对于 COBOL 从业人员是必须掌握的。本书不仅从最新的角度对 COBOL 进行了全面讲解，同时也涵盖了以上与 COBOL 紧密相关的内容。本书内容循序渐进，讲解过程详尽，不仅可以作为 COBOL 初学者的入门书籍，也可以作为初入 COBOL 行业人员的参考资料。

本书的特点

1．内容全面，对 COBOL 各方面的知识都做了系统详尽的讲解。

2．结构清晰，全书整体结构上遵循从易到难的顺序，且各章节之间都有较强的连续性。

3．内容新颖，结合当前流行的外包行业要求，从最新的角度对 COBOL 进行了讲解。

4．涉及面广，对与 COBOL 紧密相关的扩展部分进行了介绍，如 JCL 作业控制语言、DB2 数据库等。

5．实用性强，本书在各章节中都有大量程序示例，并在最后一章中讲解了对于实际系

统的综合应用开发。

6．针对性强，本书主要立足于实际应用，同当前就业市场的要求联系紧密。

7．用语规范，对于计算机方面的专业术语应用到位，严格遵循计算机科学的学术要求。

8．实例丰富，对于每一个知识点都有相应的应用实例。

9．实例典型，突出 COBOL 语言的应用特点，同时涉及有部分外包业和金融业方面的内容。

10．顺应市场，借助外包业的发展，当前 COBOL 需求旺盛，COBOL 从业人员供不应求。

11．独创性强，当前市面上 COBOL 的中文书籍较少，广大从业人员一书难求。

本书的内容

第 1 章：本章主要从整体上对 COBOL 程序设计语言进行了简要的介绍，包括 COBOL 语言的背景知识，COBOL 语言的语法格式要求，以及如何实际创建一个完整的 COBOL 程序。

第 2 章：本章主要讲解了 COBOL 程序代码的基本结构，包括标志部、环境部、数据部以及过程部。

第 3 章：本章讲解了 COBOL 语言中的各种常用语句。主要包括 MOVE 语句、PERFORM 语句以及与文件相关的语句等。

第 4 章：本章主要讲解了 COBOL 编程中所涉及到的各种基本数据类型。其中重点在于 Numeric Edited Fields 格式输出类型。

第 5 章：本章讲解了 COBOL 中字符串的概念及用法。主要包括合并、拆分、替换字符串，字符串转换，子字符串的应用，基于字符串的统计计算等。

第 6 章：本章主要讲解了 COBOL 程序中所涉及到的基本运算，包括算术运算、关系运算以及逻辑运算。

第 7 章：本章主要介绍了 COBOL 程序中的 3 大典型流程。分别为顺序结构流程、选择结构流程以及循环结构流程。

第 8 章：本章主要介绍了在 COBOL 中是如何实现对文件中的数据进行排序与合并的。此处所说的排序与合并指的是通过特定语句实现的相应功能，并不是通常所说的算法。

第 9 章：本章讲解了 COBOL 中表的概念及应用。主要包含下标表和索引表的概念及应用、常用的对于表的查找方式、定长表和变长表的概念、嵌套表的概念及应用等。

第 10 章：本章讲解了如何调试和测试 COBOL 程序。主要包括测试的基本类型和步骤、数据合法性检测、错误信息列表等。

第 11 章：本章介绍了 COBOL 中的子程序调用。主要包括子程序的调用方式、静态调用和动态调用以及嵌套子程序等。

第 12 章：本章讲解了在 COBOL 中是如何实现面向对象编程技术的。主要包括类的定义、方法的定义、客户程序的概念及应用、子类的应用、工厂对象的应用等。

第 13 章：本章讲解了 VSAM 文件的概念及相关特性。主要包括 LDS、ESDS、RRDS、KSDS，以及 VRRDS 类型的 VSAM 文件的特征及空间计算等。

第 14 章：本章讲解了与 COBOL 紧密相关的 JCL 的概念及应用。主要包括 JCL 的基本概念，JOB 语句、EXEC 语句、DD 语句这 3 类主要的 JCL 语句，以及 JCL 实用程序和过程等。

第 15 章：本章介绍了 COBOL 程序通常所访问的数据库 DB2。主要包括常用 SQL 语句、嵌入式 SQL、动态 SQL、DB2 中的游标和锁、访问路径和 EXPLAIN 优化工具等。

第 16 章：本章讲解了通常以 COBOL 作为宿主语言的用于联机交互的中间件 CICS。主要包括伪会话程序、CICS 中的程序调用、MAP 的概念及应用、CICS 对于文件的操作，以及 CICS 中队列的概念及应用等。

第 17 章：本章讲解了 COBOL 的主要运行平台——大型机上的汇编语言 ASM。主要包括大型机汇编的相关基本概念，指令类型与机器码，数据的定义、传递、运算和转换，宏指令与 DCB 参数等。

第 18 章：本章介绍了以 COBOL 及其相关技术开发一个实际系统的应用实例。该系统模拟小型银行账户管理信息系统，主要包括主菜单模块、添加账户功能模块、删除账户功能模块、修改账户功能模块以及查询账户功能模块。

适合的读者

- COBOL 初学者
- 大型机培训班学员
- 高校学生
- 编程爱好者
- 外包行业从业人员
- 大型机程序设计开发人员

致谢及反馈

这里要特别感谢我在 COBOL 及主机方面的启蒙老师们，他们分别是华中科技大学 IBM 技术中心的黄晓涛副教授、彭娅婷老师、吴驰老师、叶涛老师、王芬老师。同时，还要十分感谢武钢的资深主机工程师方文先生。参与本书编写的还有陈水峰、慈元龙、关蔼婷、贺宇、胡立实、姜磊、李来春、李争亭、刘吉万、柳玲、罗栋、罗玉霞、秦辉、陈杰、陈冠军、项宇峰、唐敏、唐智晖、王安平、王成喜、王淑敏、谢马远、张丹、张迪妮、钟蜀明、竺东、祝庆林。

读者在学习过程中如遇到任何问题，欢迎通过 E-mail 与我们及时联系，我们将尽力为大家解决这些问题。笔者的 E-mail 是endymion-2002@163.com。由于作者水平有限，书中难免会有疏漏之处，恳请广大读者进行批评指正。

作　者

2011 年 1 月

目　　录

第 1 章 COBOL 简介

COBOL（Common Business Oriented Language）即公用面向商业语言。在系统地学习 COBOL 程序设计语言之前，有必要首先对 COBOL 有一个大致的了解。本章将分别从背景知识、语法格式和实际创建 COBOL 程序 3 个方面进行简要介绍。

1.1 背景知识

在介绍 COBOL 之前，首先需要了解一下大型机的概念。大型机也叫主机、大机或 MainFrame 等，属于一种商用高端服务器。目前大型机主要由 IBM 公司生产，最新型号为 z 系列大型机。COBOL 是用于大型机上应用软件开发的主要的程序设计语言。

大型机上的工作大体上可以分为两类：系统方向和开发方向。系统方向的分工很多，包括使用 Tivoli 监控系统，使用 RACF 进行安全管理，使用 SMS 进行存储管理等。开发方向的市场需求相对系统方向则要大得多。对于开发方向，主要需要掌握以下知识。

- COBOL：最主要的程序设计语言。
- JCL：作业控制语言，调用由 COBOL 编写的程序。
- VSAM：程序中通常用到的文件类型。
- DB2：大型机上最主要的数据库，通常以 COBOL 语言为宿主语言。
- CICS：用于联机交易的中间件，通常也是以 COBOL 语言为宿主语言。

对于以上知识，在本书中都有详细介绍。此外，对于大型机底层的开发，还需要掌握大型机上的汇编语言。对于大型机汇编语言，在本书中也有具体讲解。

COBOL 作为大型机上最主要的程序设计语言，迄今已有 40 多年的历史。然而，经过 40 多年的沧桑，COBOL 不仅没有被淘汰，反而在当今愈加流行起来。关于这一点，主要是由以下 3 个因素决定的。

- COBOL 广泛应用于银行业、信用卡业、保险业、制造业、航空业等。这些领域对于稳定性的要求都是十分高的，因此从很大程度上也保证了 COBOL 的不可取代性。
- COBOL 自身也在不断发展和更新中。例如，COBOL 本身主要属于面向过程的语言。

当面向对象技术出现后，COBOL 现在也扩展为可支持面向对象技术了。

- 受信息服务外包业的带动，全球大量需要用到 COBOL 的职位正在不断涌入国内。COBOL 工程师在世界范围内都属于严重紧缺人才。

同时，COBOL 作为一门经久不衰的程序设计语言，包含着许多程序设计方面的本质思想。学习 COBOL 语言，对于深入理解结构化编程是十分有利的。研究 COBOL 语言，不仅具有一定的学术意义，更具有很强的现实意义，主要体现在以下几点。

- 从事 COBOL 工作相对稳定。一方面，COBOL 从业人员在当前供不应求；另一方面，以 COBOL 为主的大型机职位是十分重视工作经验。因此从事 COBOL 开发，不仅不会在将来被淘汰，反而能随着工作年限不断升值。
- 在美国，COBOL 程序员的薪酬比其他程序员的普遍要高一些。同时，COBOL 程序员的工作量相对其他程序员的工作量也要小一些。
- COBOL 主要应用于高端商务领域。因此，从事 COBOL 工作，有机会接触到这些领域上的许多业务知识。从 COBOL 程序员入手，今后可以向多个方向发展，道路是十分广阔的。

1.2 语法格式

COBOL 程序的语法格式在后面各个章节中都会涉及到，因此首先对其进行讲解。在大型机环境中，COBOL 程序的每行代码通常分为 80 列。对于列与列之间不同的范围，其意义各不相同。列于列之间不同的范围及其意义分别如下。

- 第 1 ~ 6 列：序号区。
- 第 7 列：指示符区。
- 第 8 ~ 11 列：A 区。
- 第 12 ~ 72 列：B 区。
- 第 73 ~ 80 列：说明区。

其中序号区中的内容通常不作要求。指示符区通常为空或者包含字符“*”。当包含“*”时，表示该行为注释行，用于书写对程序的注释信息。同时，也可以在“*”后不添加任何注释信息，以起到对程序代码段分隔的作用，利于阅读。A 区中的内容主要包括 COBOL 中的部、节、语句段的标识符，以及 01 层数据层号等。B 区中则主要包含过程部中的各条语句。说明区用于对程序进行注释。关于以上涉及到的一些概念，将在后面章节中详细讲解。

COBOL 程序中用到的各类数据名称主要由字母、数字以及中划线组成。其中字母通常为大写字母。中划线不可作为数据名称的开头或结尾。数据名称通常最多不应超过 30 个字符，并且不可定义为程序关键字。

COBOL 中的关键字主要为各种语句名、特殊数据名、象征常量名等。例如，以下为几个常见的 COBOL 中的关键字。

- PIC
- FILLER
- OPEN
- MOVE

- PERFORM
- SPACES

最后，简要介绍一下本书中用到的 COBOL 程序流程图各部件的书写规范。如表 1.1 所示，反映了流程图中的各个常用部件及其相关意义。

表 1.1　　COBOL 程序流程图规范

流程图中的部件	该部件所表示的意义
	用于表示程序的开始或结束
	用于表示程序中的某一处理步骤
	用于表示程序中数据的输入/输出过程
	用于表示程序中所调用的处理过程或子程序

1.3　COBOL 学习环境配置

本节将介绍如何实际编写并运行 COBOL 程序。在实际工作环境中，通常使用一款叫做 PCOM 的软件连接到大型机上进行开发。但大型机上的登录账号并不是人人都可获得的。此时，便可以使用模拟的大型机系统（Demo 系统）进行实践学习。

1.3.1　模拟大型机系统——Hercules

Hercules 用于模拟真实的大型机系统，即 Demo 系统，也叫 ADCD zOS 系统。该软件可以在以下网站上获取：http://www.chinamvs.net。下载完成后，将该软件直接解压缩，便可以直接使用，安装后的整个文件夹大小约为 4GB。下面重点讲解该软件是如何进行初始配置的。

（1）双击 HercGui.exe 可执行文件，打开 Hercules 软件，打开后的界面如图 1.1 所示。

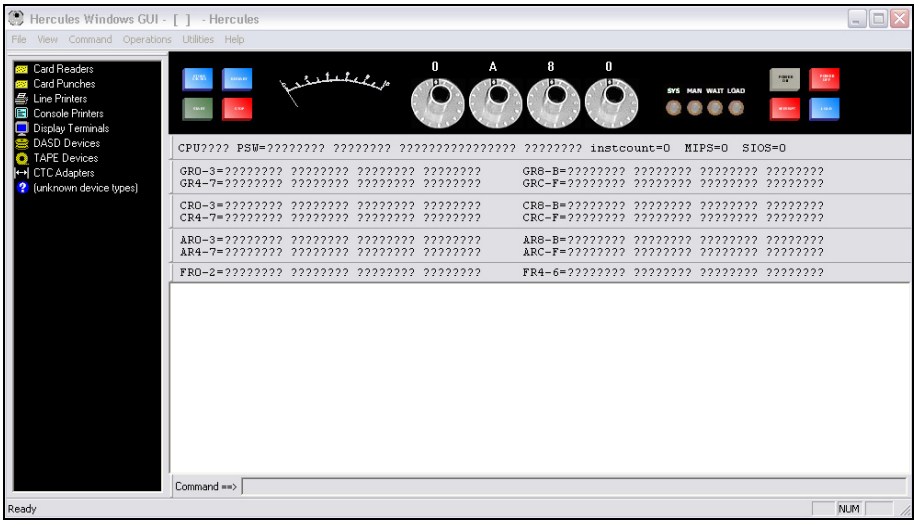

图 1.1　Hercules 软件开启界面

（2）单击“Command”|“Power On”命令，启动 Demo 系统。此时，该软件会弹出一个

用于打开文件的对话框，选择对话框中的 zos.txt 文件会弹出另一个对话框，将该对话框中的参数进行相应配置，如图 1.2 所示。确认配置后，会再次弹出一个对话框，在该对话框中选择“Yes”选项，便可以开启 Demo 系统。

（3）此时应该打开 PCOM 软件与该 Demo 系统建立连接。关于 PCOM 软件的配置，将在下一小节中进行讲解。打开 PCOM 软件后，单击 Hercules 软件界面菜单栏中的“Command”|“IPL”命令，进行系统初始化。IPL 是 Initial Program Loading 的缩写，即初始程序载入的意思。单击“IPL”命令后，在弹出的对话框中进行如图 1.3 所示的参数配置。

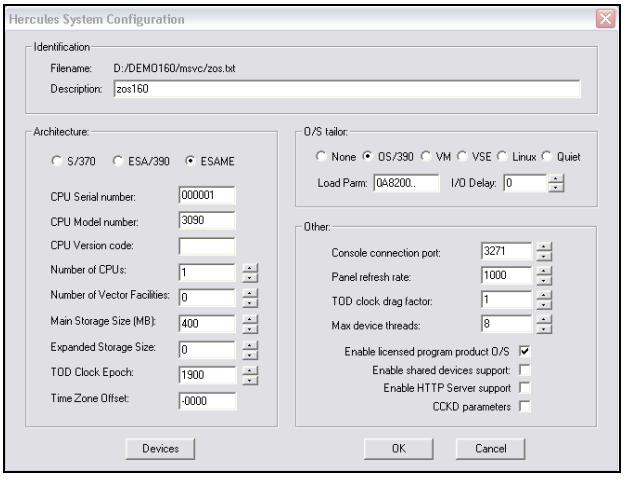

图 1.2　启动 Demo 系统时的相应配置

图 1.3　Hercules 中的 IPL 参数配置

（4）配置完成后，等待 Hercules 进行 IPL 处理。IPL 的处理过程大概会持续几分钟的时间。当整个 IPL 完成后，Hercules 的界面如图 1.4 所示。

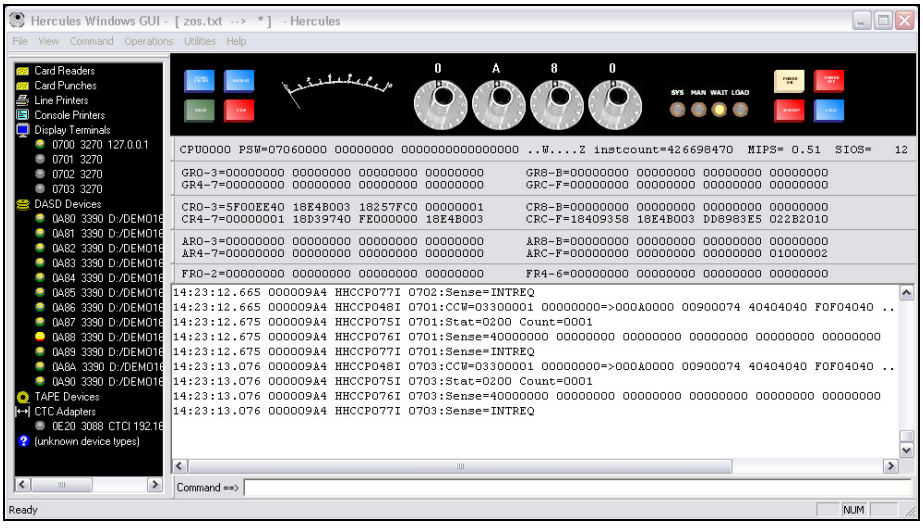

图 1.4　IPL 完成后的 Hercules 界面

此时，关于 Hercules 软件的操作便完成了。下面将重点在 PCOM 软件上进行相应操作。当关闭 Hercules 软件时，需要在界面下方的“Command”任务栏中输入“Exit”命令。

1.3.2 终端连接软件——PCOM

PCOM 软件用于作为终端连接到 Demo 系统中。该软件同样可以通过迅雷、电驴等软件下载，也可以在以下网站上获取：http://webedu.hust.edu.cn 和 http://www.chinamvs.net。

PCOM 软件的安装同普通软件安装类似，按照提示一步步操作就可以了。此外，在 PCOM 软件中，有以下几个快捷键需要特别注意。

- 右 Ctrl 键将对应 Enter 键的功能。因此，在 PCOM 中输入的命令需要通过右 Ctrl 键进行确认。
- F7 键用于对屏幕进行上翻操作，F8 键用于对屏幕进行下翻操作。
- F3 键用于退回到上一个菜单。

下面重点讲解如何配置该软件以及在该软件上的常用操作。

（1）双击*.ws（“*”表示由用户自定义的文件名）可执行文件，打开 PCOM 软件。打开后的界面如图 1.5 所示。

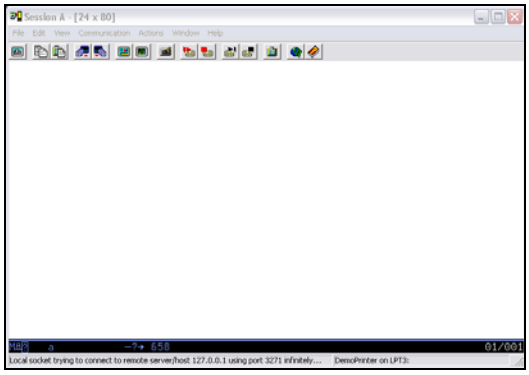

图 1.5　PCOM 主界面

（2）初次使用 PCOM 时，需要进行通信参数的设置。单击“Communication”|“Configuration”命令，打开通信配置界面，如图 1.6 所示。

（3）在以上界面中单击“Link Parameters”按钮，打开连接参数配置界面。在该界面文本框中配置相应 IP 和端口信息，如图 1.7 所示。

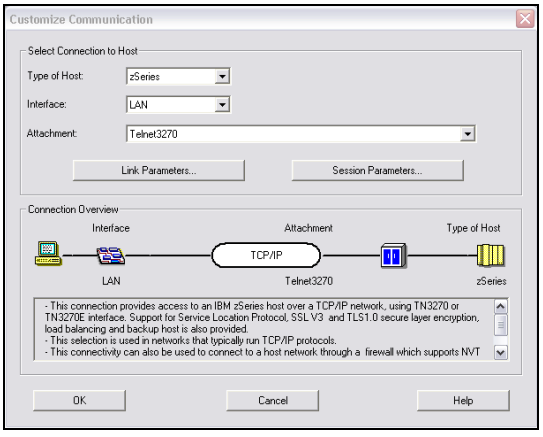

图 1.6　PCOM 通信配置界面

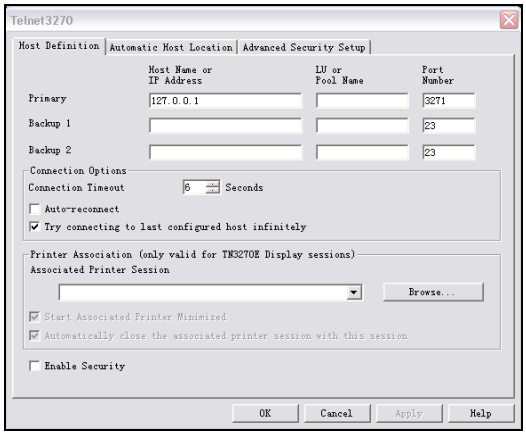

图 1.7　PCOM 连接参数的配置

（4）在 Hercules 软件上进行 IPL 处理，如上一小节所述。IPL 完成后，在 PCOM 界面下方输入“r 00, monoplex”命令以启动控制台，如图 1.8 所示。

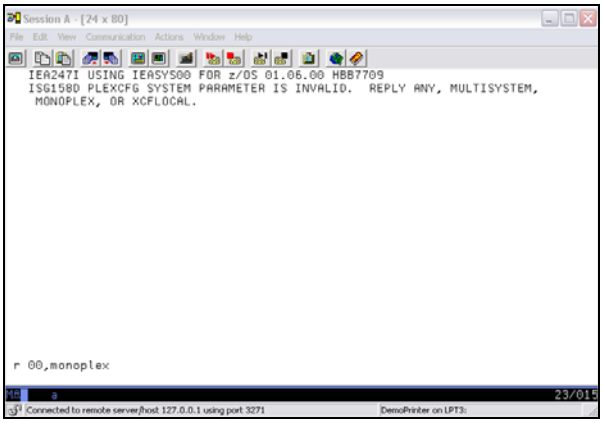

图 1.8　在 PCOM 界面上启动控制台

（5）等待系统生成控制台，直至该界面上不再产生任何信息为止。生成控制台后，启动另一个 PCOM 会话。新会话初始界面将为 Demo 系统的操作主界面，如图 1.9 所示。

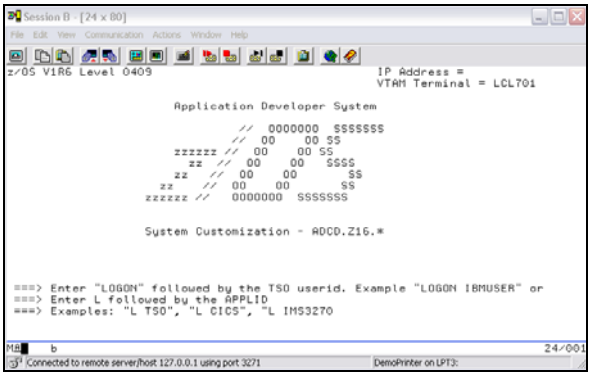

图 1.9　Demo 系统主界面

此时，整个 Demo 系统便配置完成了。进入如图 1.9 所示的 Demo 系统主界面后，便可以在该系统中编译、连接和运行 COBOL 程序了。

1.4　创建第一个 COBOL 程序

下面通过编写并运行 Hello World 程序的示例，说明在该系统上是如何编写并运行 COBOL 程序的。编写及运行 COBOL 程序可由以下几个步骤组成。

（1）在如图 1.9 所示界面下方输入“L TSO”命令，登录到 Demo 系统的 TSO 环境。此时，需要输入登录账号，账号可以为 IBMUSER 或者 ADCD*（*表示从 A 到 Z 的 26 个英文字母）等。下面以 ADCDZ 的账号登录。选择该账号后，将会出现如图 1.10 所示的 TSO 用户登录界面。

（2）在如图 1.10 所示的 TSO 用户登录界面中，需要输入登录密码。其中 ADCDZ 的初始密码为“TEST”。初次登录时，系统会要求将其改为一个由用户设置的新密码，并进行确认。设置

并确认完成新密码后，便可以进入 TSO 环境了。该 Demo 系统下的 TSO 主界面如图 1.11 所示。

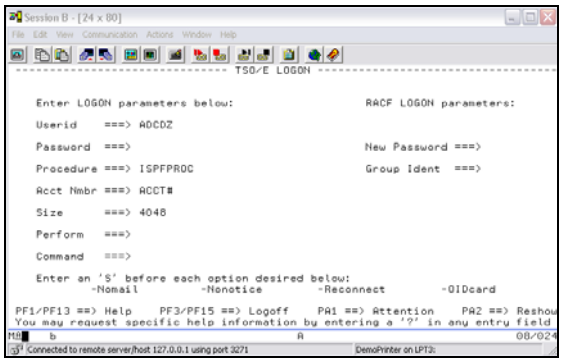

图 1.10　TSO 用户登录界面

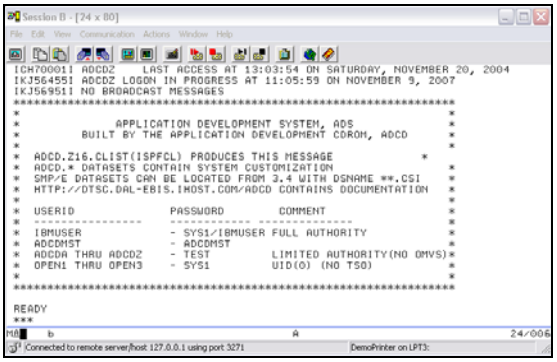

图 1.11　TSO 主界面

（3）在该界面中按下右 Ctrl 键，将会得到一个空白界面。在空白界面中输入“ISPF”命令，进入 ISPF 面板。ISPF 面板如图 1.12 所示。实际上，ISPF 面板相当于一个菜单，绝大多数大型机上的软件产品都是通过 ISPF 面板进入的。

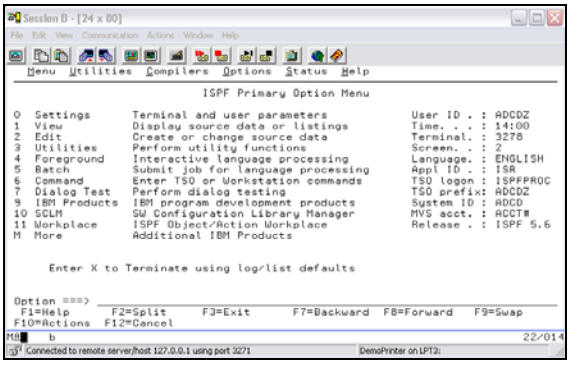

图 1.12　TSO 环境下的 ISPF 面板

（4）在如图 1.12 所示的 ISPF 面板下的“Option”一栏中输入“3.2”选项，以创建相应数据集。在通过“3.2”进入的界面中输入所要创建的数据集名“ADCDZ.COBOL.SOURCE”。其中前缀名“ADCDZ”由系统根据账号自动生成。同时，在该界面的“Opion”一栏中输入命令“A”（A 表示 Allocate，即分配的意思）。整个操作界面如图 1.13 所示。

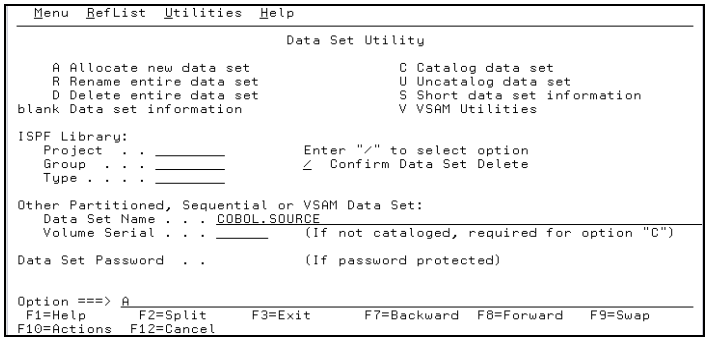

图 1.13　在“3.2”下创建 ADCDZ.COBOL.SOURCE 数据集

（5）执行以上操作后，将会出现一个新的界面。在该界面上需要设置该数据集的相关参数，如图 1.14 所示。

```
 Menu  RefList  Utilities  Help
------------------------------------------------------------------------------
                          Allocate New Data Set
                                                                More:     +
Data Set Name  . . . : ADCDZ.COBOL.SOURCE

Management class . . . ________       (Blank for default management class)
Storage class  . . . . ________       (Blank for default storage class)
 Volume serial . . . . ______         (Blank for system default volume) **
 Device type . . . . . ________       (Generic unit or device address) **
Data class . . . . . . ________       (Blank for default data class)
 Space units . . . . . TRKS           (BLKS, TRKS, CYLS, KB, MB, BYTES
                                       or RECORDS)
 Average record unit   _              (M, K, or U)
 Primary quantity  . . 5              (In above units)
 Secondary quantity    2              (In above units)
 Directory blocks  . . 2              (Zero for sequential data set) *
 Record format . . . . FB
 Record length . . . . 80
 Block size  . . . . . 1600
 Data set name type  : PDS            (LIBRARY, HFS, PDS, or blank)  *
Command ===>
 F1=Help     F2=Split     F3=Exit      F7=Backward  F8=Forward   F9=Swap
F10=Actions  F12=Cancel
```

图 1.14　设置 ADCDZ.COBOL.SOURCE 数据集的相关参数

（6）依照以上方式，再分别创建两个数据集 ADCDZ.COBOL.CNTL 和 ADCDZ.COBOL.LOAD。创建完以上 3 个数据集后，通过 F3 快捷键退回到 ISPF 主面板。在主面板“Option”栏里输入“3.4”选项，以查看并进入创建的数据集。进入“3.4”后的界面如图 1.15 所示。在该界面的相应位置中输入“ADCDZ”，用以查看其作为前缀的编目数据集。

```
                          Data Set List Utility
                                                                More:     +
   blank Display data set list              P Print data set list
       V Display VTOC information          PV Print VTOC information

Enter one or both of the parameters below:
   Dsname Level . . . ADCDZ
   Volume serial  . . ______

Data set list options
   Initial View . . . 1  1. Volume       Enter "/" to select option
                         2. Space        /  Confirm Data Set Delete
                         3. Attrib       /  Confirm Member Delete
                         4. Total        /  Include Additional Qualifiers
                                         /  Display Catalog Name

When the data set list is displayed, enter either:
  "/" on the data set list command field for the command prompt pop-up,
  an ISPF line command, the name of a TSO command, CLIST, or REXX exec, or
Option ===>
```

图 1.15　在“3.4”下选择所要查看的数据集

（7）此时，将会得到如图 1.16 所示的数据集列表。其中前 3 个数据集为之前通过“3.2”所创建的，后 2 个数据集为系统自动生成的。

```
 Menu  Options  View  Utilities  Compilers  Help
------------------------------------------------------------------------------
DSLIST - Data Sets Matching ADCDZ                        Row 1 of 5

Command - Enter "/" to select action              Message          Volume
------------------------------------------------------------------------------
         ADCDZ.COBOL.CNTL                                          Z6SYS1
         ADCDZ.COBOL.LOAD                                          Z6SYS1
         ADCDZ.COBOL.SOURCE                                        Z6SYS1
         ADCDZ.ISPF.ISPPROF                                        Z6SYS1
         ADCDZ.SPFLOG1.LIST                                        Z6SYS1
****************************** End of Data Set list ****************************
```

图 1.16　通过“3.4”得到的数据集列表

实际上，对于以上由用户所创建的 3 个数据集，用途分别如下。

- ADCDZ.COBOL.CNTL：用于存放编译连接以及调用该程序的两个 JCL 文件。
- ADCDZ.COBOL.LOAD：用于存放该程序编译连接后生成的可加载模块（Load Module）。
- ADCDZ.COBOL.SOURCE：用于存放 COBOL 程序源文件。

这 3 个数据集均为分区数据集（PDS），类似于文件夹的概念。而这些数据集中所包含的

成员，则类似于文件的概念。

（8）在以上界面的 ADCDZ.COBOL.SOURCE 右边输入“(SAMPLE)”。同时，在其左边的 Command 列下输入命令“E”，用于创建并编辑该分区数据集下的成员。其中成员名即“SAMPLE”，内容为 COBOL 中的 Hello World 程序源代码。在该成员内编写程序时，可以通过在左边行号处输入命令“I”插入新的一行。编写完成后的源程序如图 1.17 所示。

```
EDIT       ADCDZ.COBOL.SOURCE(SAMPLE) - 01.02            Columns 00001 00072
****** ******************************** Top of Data ********************************
==MSG> -Warning- The UNDO command is not available until you change
==MSG>           your edit profile using the command RECOVERY ON.
=COLS> ----+----1----+----2----+----3----+----4----+----5----+----6----+----7--
000200         IDENTIFICATION DIVISION.
000300         PROGRAM-ID.  SAMPLE.
000400         AUTHOR.    XXX.
000500         ENVIRONMENT DIVISION.
000600         DATA DIVISION.
000700         WORKING-STORAGE SECTION.
000800         01   TEST-DATA   PIC X(20).
000900         PROCEDURE DIVISION.
001000              MOVE 'HELLO WORLD!' TO TEST-DATA.
001100              DISPLAY TEST-DATA.
001200              STOP RUN.
****** ***************************** Bottom of Data ******************************
```

图 1.17　COBOL 中的 Hello World 程序源代码

（9）退回到数据集列表，以类似方式在 ADCDZ.COBOL.CNTL 下创建成员 COMPILE。在该成员内输入以下 JCL 代码，用于编译并连接 Hello World 源程序。编写完成 JCL 代码后，需要在界面下方的 Command 栏中输入“sub”命令，用于向系统提交该 JCL。编译并连接 Hello World 源程序的 JCL 代码如下。

```
//ADCDZ01  JOB (ACCOUNT-NUMBER,ACCOUNTING-INFORMATION,LESSTHEN-143CHR),
// (PROGRAMMERS-NAME),CLASS=A,MSGCLASS=X,MSGLEVEL=(1,1),
//          TIME=(12,10),
//       COND=(12,LE),NOTIFY=&SYSUID
//* ADDRSPC=VIRT/REAL
//* BYTE=(100,(CANCEL/DUMP/WARNING))
//* RESTART=STEPNAME
//* GROUP=????
//* PERFORM=????
//* RD=????
//* SECLABEL=????
//IGYWCL PROC  SYSLBLK=3200,
//           LIBPRFX='CEE',
//           SRCLIB=ADCDZ.COBOL.SOURCE,        /*此处指定源程序所在数据集*/
//           LOADLIB=ADCDZ.COBOL.LOAD,         /*此处指定可加载模块所在数据集*/
//           MEMBER=SAMPLE,                    /*此处存放源程序的成员名*/
//           LOADMO=LOADMOD                    /*此处指定存放可加载模块的成员名*/
//COBOL  EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB  DD  DSNAME=IGY330.SIGYCOMP,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,UNIT=SYSDA,
//           DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//           DCB=(BLKSIZE=&SYSLBLK)
//SYSIN    DD  DSN=&SRCLIB(&MEMBER),DISP=SHR
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
```

```
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//*
//LKED   EXEC PGM=HEWL,COND=(8,LT,COBOL),REGION=1024K
//SYSLIB   DD  DSNAME=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,DISP=(OLD,DELETE)
//        DD  DDNAME=SYSIN
//SYSLMOD  DD  DSNAME=&LOADLIB(&LOADMO),
//           SPACE=(TRK,(10,10,1)),
//           UNIT=SYSDA,DISP=(MOD,PASS)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(TRK,(10,10))
//      PEND
//*
//       EXEC  IGYWCL
//
```

当提交的 JCL 运行成功时，系统会返回一条包含“MAXCC=0”的提示信息。此时，在 ADCDZ.COBOL.LOAD 将会产生一个名为“LOADMOD”的成员。该成员实际上就是 Hello World 程序编译连接后，所生成的可加载模块。

（10）在 ADCDZ.COBOL.CNTL 中再创建一个成员，并将其命名为“RUN”。该成员用于存放调用可加载模块（即执行 Hello World 程序）的 JCL 代码。相应 JCL 代码如图 1.18 所示。编写完成后仍然需要通过“sub”命令将其提交。

```
EDIT       ADCDZ.COBOL.CNTL(RUN) - 01.08                Columns 00001 00072
****** ***************************** Top of Data ******************************
==MSG> -Warning- The UNDO command is not available until you change
==MSG>           your edit profile using the command RECOVERY ON.
000100 //ADCDZ01 JOB CLASS=A,MSGLEVEL=(1,1),MSGCLASS=A,
000200 //         NOTIFY=&SYSUID
000300 //*************************************************
000400 //COBOL01 EXEC PGM=LOADMOD
000500 //*************************************************
000600 //STEPLIB  DD DSN=ADCDZ.COBOL.LOAD,DISP=SHR
000700 //SYSPRINT DD SYSOUT=*
001000 //SYSIN    DD *
001100 //
****** **************************** Bottom of Data ****************************



Command ===> sub                                          Scroll ===> CSR
```

图 1.18　执行 Hello World 程序的 JCL

（11）返回 ISPF 主面板，在“Option”栏里输入“m.5”选项，进入 SDSF 查看程序运行结果。进入 SDSF 后，在其主界面下方的“COMMAND”INPUT 栏中输入命令“ST”，查看相应的作业列表，如图 1.19 所示。

```
HQX7708 -----------------  SDSF PRIMARY OPTION MENU  --------------------------

DA     Active users                      INIT  Initiators
I      Input queue                       PR    Printers
O      Output queue                      PUN   Punches
H      Held output queue                 RDR   Readers
ST     Status of jobs                    LINE  Lines
                                         NODE  Nodes
LOG    System log                        SO    Spool offload
MAS    Members in the MAS
JC     Job classes                       ULOG  User session log
SE     Scheduling environments
RES    WLM resources

Licensed Materials - Property of IBM
COMMAND INPUT ===> ST                                      SCROLL ===> PAGE
```

图 1.19　SDSF 主界面

此时，将可查看到之前由 JCL 所提交的两个作业（Job），如图 1.20 所示。

其中，第一个作业为编译连接 Hello World 程序的作业，第二个为运行该程序的作业。

（12）在如图 1.19 所示的第二个作业左边的 NP 列下输入命令“？”，以查看该作业的详

细信息，如图 1.21 所示。

```
  Display  Filter  View  Print  Options  Help
--------------------------------------------------------------------------------
SDSF STATUS DISPLAY ALL CLASSES                      LINE 523-524 (524)
NP   JOBNAME  JobID    Owner    Prty Queue      C  Pos  SAff  ASys Status
     ADCDZ01  JOB02018 ADCDZ       1 PRINT      A   501
     ADCDZ01  JOB02019 ADCDZ       1 PRINT      A   502
```

图 1.20　之前由 JCL 提交的作业信息列表

```
  Display  Filter  View  Print  Options  Help
--------------------------------------------------------------------------------
SDSF JOB DATA SET DISPLAY - JOB ADCDZ01  (JOB02019)    LINE 1-4 (4)
NP   DDNAME   StepName ProcStep DSID Owner    C Dest              Rec-Cnt Page
     JESMSGLG JES2                2 ADCDZ    A LOCAL                16
     JESJCL   JES2                3 ADCDZ    A LOCAL                 9
     JESYSMSG JES2                4 ADCDZ    A LOCAL                16
     SYSOUT   COBOL01           103 ADCDZ    A LOCAL                 1
```

图 1.21　运行 Hello World 程序的作业详细信息列表

以上列表最下一行的 SYSOUT 即包含 Hello World 程序运行后的输出结果。在该行的 NP 列下输入命令“S”，便可以查看到如图 1.22 所示的程序运行结果了。

```
  Display  Filter  View  Print  Options  Help
--------------------------------------------------------------------------------
 SDSF OUTPUT DISPLAY ADCDZ01  JOB02019  DSID   103 LINE 0       COLUMNS 02- 81
 COMMAND INPUT ===>                                          SCROLL ===> PAGE
********************************** TOP OF DATA ***********************************
HELLO WORLD!
******************************** BOTTOM OF DATA **********************************
```

图 1.22　Hello World 程序运行结果

全部操作完成之后，连续按 F3 键退回到 ISPF 主面板。在 ISPF 主面板中再次按 F3 键，将出现退出操作界面。在该界面指定位置输入“2”后，再输入“Log off”命令，以退出 PCOM 中的 TSO 环境。

当然，在此之后编写并运行 COBOL 程序的步骤不必这么麻烦。因为很多系统环境中的初始化参数不必再次进行设置，同时此处编写的 JCL 也可以反复进行利用。

1.5　本章回顾

本章主要从整体上对 COBOL 程序设计语言进行了大致的介绍。本章内容分别包括 COBOL 语言的背景知识、COBOL 语言的语法格式要求以及如何实际创建一个完整的 COBOL 程序。重点在于如何实际创建完整的 COBOL 程序。

学习本章内容，需要大致了解 COBOL 语言的特点及应用领域，了解 COBOL 语言的基本语法格式要求，熟悉 PCOM 和 Hercules 的基本用法，掌握如何通过 PCOM 在 Hercules 大型机模拟系统上编译、连接以及运行 COBOL 程序。

第 2 章 程序结构

COBOL 程序在编写时需要遵循严格的程序结构规则。每个完整的 COBOL 程序代码都是由 4 个部组成的。这 4 个部依次为标志部、环境部、数据部以及过程部。部与部之间的先后顺序不可更改。下面依次对 COBOL 程序结构里的这 4 个部进行讲解。

2.1 标志部

标志部是 COBOL 程序中的第一个部，由“IDENTIFICATION DIVISION”关键字标识，主要用来描述与程序本身相关的信息。任何一个 COBOL 程序中都必须包含有标志部。标志部中主要包含以下字段。

- PROGRAM-ID。
- AUTHOR。
- INSTALLATION。
- DATE-WRITTEN。
- DATE-COMPILED。
- SECURITY。

其中，第一个字段 PROGRAM-ID 用于指明程序名，是必须具备的字段。其他字段则都是可选的。但通常情况下，除以上第一个字段以外，还会包含有第二个字段。第二个字段用于指明开发该程序的程序员名。因此通常情况下，COBOL 程序在标志部中的格式如下。

```
IDENTIFICATION  DIVISION.
PROGRAM-ID.  SAMPLE-PGM.
AUTHOR.  XXX.
……
```

以上代码指定了该程序的程序名为“SAMPLE-PGM”，开发该程序的程序员为“×××”。需要注意的是，程序名通常应该选择一个比较有意义的名字，最好能反映出该程序的大体功能。在规范的开发部门中，程序名通常是由公司所指定的。

对于标志部中的程序员名字段，虽然不是必须要求具备的，但通常都会包含在内。注明程序员名，对于该程序的后期维护是很重要的。本书中该字段通常都指定为“×××”。在实际编写自己的程序时，可以署上自己的名字或代号。关于标志部中其他几个字段简要介绍如下。

- INSTALLATION：通常指设计该程序的公司或部门。
- DATE-WRITTEN：指明程序编写或修改的日期。
- DATE-COMPILED：指明程序编译的日期。
- SECURITY：通常用于列出具有访问该程序权限的用户。

2.2 环境部

环境部紧接在标志部之后，主要用于指定该程序同外部系统环境之间的各种对应关系。这种对应关系主要体现在程序的逻辑部分和环境的物理部分的对应。环境部包含有两个节(也可称作“区”)，其中一个为配置节，另一个为输入/输出节。以下分别进行讲解。

2.2.1 配置节

在讲解配置节之前，首先需要了解一下该节所在的环境部特征。环境部通过“ENVIRONMENT DIVISION”标识。同标志部不一样，如果该部没有任何内容，是可以省略不写的。不过通常按规范至少应该加上该部的标识字段。环境部在程序中的格式如下。

```
IDENTIFICATION  DIVISION.
PROGRAM-ID.  SAMPLE-PGM.
AUTHOR.  XXX.
*
ENVIRONMENT  DIVISION.
……
```

环境部中的配置节由“CONFIGURATION SECTION”标识。该节主要用于指定程序同外部环境之间的一些配置信息。该节也是可以省略不写的。对于该节所包含的字段简要介绍如下。

- SOURCE-COMPUTER：指示编译程序的计算机。
- OBJECT-COMPUTER：描述运行程序的计算机。
- SPECIAL-NAMES：指定货币符号选择小数点、提供开关名和定义字母表。

2.2.2 输入/输出节

在 COBOL 程序中，数据主要保存在文件中。数据的输入和输出通常是以文件为对象进行的。因此，输入/输出节主要用于指定程序中所用到文件同外部环境之间的对应关系。也就是说，输入/输出节主要是将程序中的逻辑文件同环境中的物理文件相对应起来。输入/输出节由“INPUT-OUTPUT SECTION”标识，它包含有两个字段，分别简要介绍如下。

- FILE-CONTROL：用于指定文件的对应关系，是该节中主要用到的字段。
- I-O-CONTROL：用于定义程序返回点，不同文件共享的内存区，以及多文件卷中文件

的位置。

下面重点讲解输入/输出节中的 FILE-CONTROL 字段。该字段在 COBOL 程序中基本的使用格式如下所示。

```
IDENTIFICATION  DIVISION.
PROGRAM-ID.  SAMPLE-PGM.
AUTHOR.  XXX.
*
ENVIRONMENT  DIVISION.
INPUT-OUTPUT  SECTION.
FILE-CONTROL.
    SELECT  CUSTOMER-FILE
         ASSIGN  TO  S-SYSIN.
    SELECT  PRINT-FILE
         ASSIGN  TO  S-SYSOUT.
......
```

在以上代码中，通过 SELECT 语句指定了两条文件对应关系。其中第一条将 CUSTOMER-FILE 文件同 S-SYSIN 文件相对应。第二条将 PRINT-FILE 文件同 S-SYSOUT 文件相对应。以上 4 个文件所处位置分别如下。

- 在 COBOL 程序中的逻辑文件：CUSTOMER-FILE、PRINT-FILE。
- 在外部环境中的物理文件：S-SYSIN、S-SYSOUT。

需要注意的是，对于 S-SYSIN 和 S-SYSOUT，必须同相应 JCL 中指定的文件名一致。而对于 CUSTOMER-FILE 和 PRINT-FILE，通常需要在接下来的数据部中对其进行相关定义。此处实际上主要是在程序里的逻辑名称和环境中的相应物理设备之间建立起了一定的联系。

2.3 数据部

数据部紧接在环境部之后，由“DATA DIVISION”标识。COBOL 程序中所用到的各项数据都统一在数据部中定义，因此数据部十分重要。数据部中常用的有 3 个节，分别为文件节、工作存储节以及连接节。此处主要讲解文件节和工作存储节。连接节将在后面的子程序调用和 CICS 伪会话程序中具体讲解到。下面对数据部的几个节作详细介绍。

2.3.1 文件节

文件节由“FILE-SECTION”所标识，用于对程序中用到的文件里的数据进行定义。需要注意的是，此处所说的文件指的是程序中的逻辑文件。其文件名必须和环境部输入/输出节中由 SELECT 语句指定的文件名一致。

在文件节中，主要是通过 FD 语句实现对文件的定义。在 FD 语句下，通过 PIC 语句依次定义文件中每条记录所包含的数据项。任何在程序中所使用到的文件都需要在此处定义。以下代码反映了文件节在程序中的大致用法。

```
IDENTIFICATION  DIVISION.
PROGRAM-ID.  SAMPLE-PGM.
AUTHOR.  XXX.
*
ENVIRONMENT  DIVISION.
```

```
INPUT-OUTPUT  SECTION.
FILE-CONTROL.
    SELECT  CUSTOMER-FILE
         ASSIGN  TO  S-SYSIN.
    SELECT  PRINT-FILE
         ASSIGN  TO  S-SYSOUT.
*
DATA  DIVISION.
FILE  SECTION.
FD  CUSTOMER-FILE
     RECORDING MODE IS F.
01  CUST-RECORD.
     05  ACCOUNT-NO    PIC   9(10).
     05  CLIENT-NAME   PIC   X(20).
     05  PRI-FINANCE.
         10  F-SALARY   PIC   9(5).
         10  F-STOCK    PIC   9(7).
         10  F-FUND     PIC   9(7).
         10  F-FOREX    PIC   9(7).
FD  PRINT-FILE
    RECORDING MODE IS F
    LABEL RECORDS ARE OMITTED
    RECORD CONTAINS 132 CHARACTERS
    DATA RECORD IS PRINT-LINE.
01  PRINT-LINE         PIC   X(132).
......
```

以上代码在环境部中指定了两个文件，因此在数据部中也相应的有两条 FD 语句用于对其定义。其中 FD 是“File Descriptor”的缩写，即文件描述符的意思。关于在数据部文件节中的 FD 语句，有以下两点需要注意。

- FD 语句主要用来描述与文件记录相关的物理信息，如记录实际的格式、长度等。
- FD 语句通过句点表示结束。

此外，可以看到，以上代码中的 FD 语句实际上是通过各种从句描述文件记录信息的。关于这些从句分别介绍如下。

- RECORDING MODE IS F：该从句用于指明文件记录的格式。其中 F 表示“Fixed-length records”，即定长记录。通常情况下所用到的文件记录大多都为定长的。当不为定长时，也可通过其他字符进行指明。
- LABEL RECORDS ARE OMITTED：表明忽略文件记录的标号。当缺省时，OMITTED 处为 STANDARD。其中手工指定为 OMITTED 时，将对应于用作输出打印的文件。
- RECORD CONTAINS 132 CHARACTERS：表明每条文件记录包含 132 个字符。标准情况下输出文件中的记录是包含 132 个字符的。同时，此处所指定的字符个数需要与后面由 PIC 语句指定的总字符个数一致。
- DATA RECORD IS PRINT-LINE：指明该文件所包含的记录名。该从句是在 COBOL-85 版本中新创建的。

实际上，关于 FD 语句，完整的语法格式如下。

```
FD  file-name
    BLOCK  CONTAINS  n  RECORDS
    RECORD  CONTAINS  n  CHARACTERS
```

```
    LABEL  RECORD  IS  OMITTED/STANDARD
      (or: LABEL  RECORDS  ARE  OMITTED/STANDARD)
    RECORDING  MODE  IS  F
    DATA  RECORD  IS  record-name.
```

对于以上格式，主要需要补充的是关于 BLOCK CONTAINS n RECORDS 这条从句。该从句用于指明在一个数据块中包含有多少条数据记录。通常情况下，此处的数值 n 为 0，表示在程序执行过程中动态地对数据记录组块。

对于所定义的数据，可以看到，实际上主要分为两大类型。其中一类为数据组，另一类则为单元数据项。数据组是由多个单元数据项或数据分组所组成的。FD 语句下所有的数据组和数据项组成了该文件的一条记录。

实际上，在文件节中各条 FD 语句下所定义的数据组和数据项是对相应文件记录逻辑上的一个划分。在以上示例代码，对于 CUSTOMER-FILE 文件记录中各项数据的定义如下。

```
 ......
DATA  DIVISION.
FILE  SECTION.
FD   CUSTOMER-FILE
     RECORDING MODE IS F.
01   CUST-RECORD.
     05  ACCOUNT-NO    PIC   9(10).
     05  CLIENT-NAME   PIC   X(20).
     05  PRI-FINANCE.
         10  F-SALARY   PIC   9(5).
         10  F-STOCK    PIC   9(7).
         10  F-FUND     PIC   9(7).
         10  F-FOREX    PIC   9(7).
......
```

首先关注一下数据定义中的层。在 COBOL 中，01 层属于最高层，需要在程序中顶格写。各层层号必须按顺序书写，但不一定连续。在 COBOL 中，最多支持 49 层，因此 49 层为最低层。除 01 层外，还存在几个比较特殊的层，分别介绍如下。

- 66 层：为特殊描述符项目保留，在实际中用得较少。
- 77 层：用于某个组的数据项，如程序中的临时变量。
- 88 层：主要用于条件判断中，在后面相应章节有详细讲解。

接下来针对以上代码分析其中所定义的数据组和单元数据项。在该段代码中，所定义的数据组有以下两项。

- CUST-RECORD。
- PRI-FINANCE（该数据组实际上为数据组 CUST-RECORD 下的一个分组）。

所定义的单元数据项及其所在的数据组则分别如下。

- ACCOUNT-NO，所在数据组为 CUST-RECORD。
- CLIENT-NAME，所在数据组为 CUST-RECORD。
- F-SALARY，所在数据组为 PRI-FINANCE。
- F-STOCK，所在数据组为 PRI-FINANCE。
- F-FUND，所在数据组为 PRI-FINANCE。
- F-FOREX，所在数据组为 PRI-FINANCE。

区分数据组和单元数据项最直接的方式是看该数据后是否有 PIC 语句。其中没有 PIC 语句的为数据组，有 PIC 语句的则为单元数据项。

PIC 实际上是 PICTURE 的缩写。PIC 语句用来定义数据的类型及长度。关于 COBOL 中所包含的各种数据类型将在后面章节中详细讲解。此处只需注意其中“×”和“9”的区别。“×”表示任意字符，“9”则表示只能为数字。二者后面括号中的数值则表示相应数据的长度。数据长度也可直接通过连续书写“×”或“9”表示。例如，以下分别为两对等效的表达方式。

```
01  TEST-DATA     PIC X(3).
01  TEST-DATA     PIC XXX.
……
01  TEST-NUM      PIC 9(3).
01  TEST-DATA     PIC 999.
```

2.3.2 工作存储节

工作存储节由“WORKING-STORAGE SECTION”所标识。工作存储节主要用于定义本程序中所需用到的各种数据。其中数据的定义方式同在文件节中的类似，不过此处定义的数据与文件是没有直接关系的。

按照规定，工作存储节应该写在文件节之后。如果程序中没有用到外部文件，文件节是可以缺省的。但工作存储节则通常都是会存在的。因为程序中一般总会要用到本地数据的。以下代码反映了工作存储节的通常用法。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  SAMPLE-PGM.
 AUTHOR.  XXX.
*
 ENVIRONMENT  DIVISION.
 ……
*
 DATA  DIVISION.
 FILE  SECTION.
……
 WORKING-STORAGE  SECTION.
 01  HEADING-LINE.
     05   FILLER     PIC  X(12)
                     VALUE   SPACES.
     05   FILLER     PIC  X(20)
                     VALUE   'CLIENT  NAME  LIST'.
     05   FILLER     PIC  X(100)
                     VALUE   SPACES.
01  DETAIL-LINE.
    05   FILLER     PIC  X(10)
                    VALUE   SPACES.
    05   PRT-NAME   PIC  X(20).
    05   FILLER     PIC   X(102)
                    VALUE   SPACES.
 01  EOF-FLAG       PIC   X  VALUE  'N'.
……
```

在以上代码中，对于各数据组及单元数据项分别介绍如下。

- HEADING-LINE：该数据组主要用于打印标题，在生成报表的程序中经常会用到。

- DETAIL-LINE：该数据组可用于提取输入文件记录中的 CLIENT-NAME 信息，并复制到输出文件记录中。
- EOF-FLAG：该单元数据项用于指示文件是否处理结束。其中 EOF 是 End Of File 的缩写，即文件结束的意思，该字符在 COBOL 中经常会遇到。

需要注意的是，以上所说的主要是指这些数据的意义，与数据的名称无关。实际上，此处的名称通常是可以任意指定的。例如，可用 ERROR-CODE 表示 ERR-CODE，或用 MSG01 表示 MSG 等。但是，所指定的名称应该具备一定的象征意义，能够使人明白该数据的主要用途。

与其他用户自定义的数据名称不同，“FILLER”属于系统的一个关键字。FILLER 所描述的数据具有特殊的意义，不能通过其他用户自定义的数据名取代。使用 FILLER 描述的数据，主要是需要占用记录位置，但在程序又不会用到的。

例如，对于在以上代码中定义的 HEADING-LINE，可将其复制到 PRINT-LINE 中以输出。由于在 HEADING-LINE 中使用 FILLER 定义了相应填充位置，因此输出后的结果如下。

```
        CLIENT  NAME  LIST
```

如果没有使用 FILLER 定义填充位置，则 HEADING-LINE 中将仅包含实际数据的输出信息。此时，对于 HEADING-LINE 的定义如下。

```
01  HEADING-LINE.   PIC  X(20)
                    VALUE   'CLIENT  NAME  LIST'.
```

根据以上定义，HEADING-LINE 输出后的结果如下。

```
CLIENT  NAME  LIST
```

此外，FILLER 关键字是可以缺省的。当在定义数据时不指定任何数据名称，则系统将默认按照 FILLER 的定义方式进行处理。以下代码反映了缺省 FILLER 时的用法。

```
01   WEEK-VALUES.
     05   FILLER  PIC X(10)    VALUE  'MONDAY'.
     05           PIC X(10)    VALUE  'TUESDAY'.
     05           PIC X(10)    VALUE  'WEDNESDAY'.
     05           PIC X(10)    VALUE  'THURSDAY'.
     05           PIC X(10)    VALUE  'FRIDAY'.
     05           PIC X(10)    VALUE  'SATURDAY'.
     05           PIC X(10)    VALUE  'SUNDAY'.
01   WEEK-TABLE-ONE   REDEFINES   WEEK-VALUES.
     05   DAYS     PIC X(10)     OCCURS 7 TIMES.
```

需要注意的是，在程序中是不能像引用其他数据一样对 FILLER 描述的数据进行引用的。例如，以下语句便是错误的。

```
MOVE  'ABC'  TO  FILLER.
ADD  2  TO  3  GIVING  FILLER.
STRING  FILLER  INPUT-DATA
    DELIMITED  BY  SIZE  INTO  RESULT-DATA.
……
```

对于工作存储节中的数据，也是需要使用 PIC 语句定义其类型和长度的。此外，在使用 PIC 语句对数据定义的同时，也可以使用 VALUE 从句对其赋初值。并且，对于 FILLER 描述

的数据项，通常是必须使用 VALUE 从语对其赋初值。例如，以下代码便将字符“ABC”赋予了 TEST-CHAR，将数字 123 赋予了 TEST-NUM，将空格赋予了 FILLER 所描述的数据项。

```
01  TEST-CHAR    PIC X(3)   VALUE  'ABC'.
01  TEST-NUM     PIC 9(3)   VALUE  123.
01  FILLER       PIC X(5)   VALUE  SPACES.
```

2.4 过程部

过程部紧接着数据部之后，是 COBOL 程序中的最后一个部。过程部由“PROCEDURE DIVISION”标识。COBOL 程序的所有逻辑处理部分都是在过程部中编写的。过程部是程序的主体部分，后面各章节所讲解的内容主要都是在过程部中实现的。

过程部中的程序代码是由 COBOL 语句所组成的。同时，可以将实现某一特定功能的多条语句定义为一个语句段。语句段在 COBOL 中类似于函数的概念，可以直接通过语句段名对其进行调用。语句段名需要在程序中顶格写。

此外，关于过程部中的语句，有一点需要特别注意，就是语句之后的句点“.”。通常情况下，语句之后有无句点并无太大影响，但对于以下两种情况需要特别注意。

- 语句段名之后必须有句点。
- 句点能够起到界点同一条件下处理范围的作用。这一点同后面要讲到的 END-IF 或者 END-PERFORM 等标识符功能类似，并可用句点将其替换。

另外需要注意的是，COBOL 中的语句是可以换行书写的。但换行书写时，语句中所包含的单个选项或者变量名在每行内需要书写完整，不能将其拆分成两行。

最后，当程序逻辑代码完全结束后，通常使用“STOP RUN”语句表明程序的结束。在子程序中，也可使用“GOBACK”语句或“EXIT PROGRAM”语句表示程序的结束。关于这 3 者的区别，将在子程序调用一章中详细讲解。

下面综合前面所讲解的 COBOL 中的各个部，给出一段完整的 COBOL 示例程序。该程序包含了各个部的内容，代码如下。

```
 IDENTIFICATION  DIVISION.              /*以下为标志部内容*/
 PROGRAM-ID.  SAMPLE-PGM.
 AUTHOR.  XXX.
*
 ENVIRONMENT  DIVISION.                 /*以下为环境部内容*/
 INPUT-OUTPUT  SECTION.
 FILE-CONTROL.
    SELECT  CUSTOMER-FILE
         ASSIGN  TO  S-SYSIN.
    SELECT  PRINT-FILE
         ASSIGN  TO  S-SYSOUT.
*
 DATA  DIVISION.                        /*以下为数据部内容*/
 FILE  SECTION.
 FD  CUSTOMER-FILE
     RECORDING MODE IS F.
 01  CUST-RECORD.
     05  ACCOUNT-NO     PIC   9(10).
     05  CLIENT-NAME    PIC   X(20).
```

```
       05  PRI-FINANCE.
           10  F-SALARY    PIC   9(5).
           10  F-STOCK     PIC   9(7).
           10  F-FUND      PIC   9(7).
           10  F-FOREX     PIC   9(7).
    FD  PRINT-FILE
        RECORDING MODE IS F
        LABEL RECORDS ARE OMITTED
        RECORD CONTAINS 132 CHARACTERS
        DATA RECORD IS PRINT-LINE.
    01   PRINT-LINE         PIC   X(132).
    WORKING-STORAGE  SECTION.
    01  HEADING-LINE.
      05   FILLER    PIC  X(10)
                     VALUE   SPACES.
      05   FILLER    PIC  X(20)
                     VALUE   'CLIENT  NAME  LIST'.
      05   FILLER    PIC  X(102)
                     VALUE   SPACES.
    01  DETAIL-LINE.
      05   FILLER    PIC  X(12)
                     VALUE   SPACES.
      05   PRT-NAME  PIC  X(20).
      05   FILLER    PIC   X(100)
                     VALUE   SPACES.
   01  EOF-FLAG      PIC   X  VALUE  'N'.
   *
    PROCEDURE  DIVISION.              /*以下为过程部内容*/
    000-PREPARE-CUSTOMER-REPORT.
         OPEN   INPUT    CUSTOMER-FILE
                OUTPUT   PRINT-FILE.
    100-WRITE-HEADING-LINE.
         MOVE  HEADING-LINE  TO  PRINT-LINE.
         WRITE  PRINT-LINE.
    200-PROCESS-RECORDS.
         PERFORM  UNTIL  EOF-FLAG = 'Y'
           READ  CUSTOMER-FILE
              AT  END  MOVE  'Y'  TO  EOF-FLAG
           END-READ
           MOVE  CLIENT-NAME  TO  PRT-NAME
           MOVE  DETAIL-LINE  TO  PRINT-LINE
           WRITE  PRINT-LINE
         END-PERFORM.
         CLOSE   CUSTOMER-FILE
                 PRINT-FILE.
          STOP  RUN.
```

以上程序执行后，将在输出文件中产生如下形式的报表信息。

```
          CLIENT  NAME  LIST
            LING  CHEN
            YU  ZHOU
            LI  GUO  QIANG
            ZHOU  XIN
            CAI  JUN  PU
            ......
```

2.5 本章回顾

本章主要讲解了 COBOL 程序代码的基本结构。COBOL 程序代码的基本结构包括 4 个部，分别为标志部、环境部、数据部和过程部。程序结构是 COBOL 最基本的概念。了解这种程序结构，也是学习 COBOL 最首要的任务。

本章依照各部在 COBOL 程序中的先后顺序，首先讲解了 COBOL 中的标志部。标志部由“IDENTIFICATION DIVISION”标识，用于描述和程序本身相关的各种信息。该部通常主要包括程序名和程序员名。

本章接下来讲解了 COBOL 中的环境部。环境部由“ENVIRONMENT DIVISION”标识，用于指定程序同外部系统环境之间的各种对应关系。环境部中包含有两个节，分别为配置节和输入/输出节。学习这部分内容，需要了解以上这两个节各自描述的信息；重点掌握输入/输出节中对文件的指定方式。

本章之后讲解了 COBOL 中的数据部。数据部由“DATA DIVISION”标识，用于定义程序中用到的各项数据。数据部中主要包含 3 个节，分别为文件节、工作存储节以及连接节。此处只讲解了文件节和工作存储节，连接节将在后面相关章节中详细讲到。学习这部分内容，需要掌握如何通过 FD 语句定义文件信息，掌握数据的层次划分，掌握如何使用 PIC 语句定义数据，理解 FILLER 关键字的用法及意义，掌握如何使用 VALUE 从语对数据赋初值。

本章最后讲解了 COBOL 中的过程部。过程部由“PROCEDURE DIVISION”标识，用于编写程序的逻辑处理部分。过程部是整个 COBOL 程序的主体部分，后面各章节中讲解的内容主要都是用于过程部之中的。学习这部分内容，需要理解语句和语句段各自的意义，理解句点的用法及意义。最后还需要在此基础上，掌握完整 COBOL 程序的基本结构。

第 3 章 常用语句

COBOL 程序的源代码主要是由各条具有一定功能的 COBOL 语句所组成的。在全体 COBOL 语句中，有几条属于常用的 COBOL 语句。这几条常用语句将贯穿于 COBOL 的各部分具体应用之中，本章首先对其进行讲解。

3.1 MOVE 语句

MOVE 语句是整个 COBOL 程序中最常用到的语句。通过 MOVE 语句的复制操作，可以实现变量赋值、参数传递等多项类似的功能。同时，MOVE 语句既可以对单个数据项进行复制，也可以对一组数据进行复制。下面分别进行讲解。

3.1.1 复制单个数据项

复制单个数据项是 MOVE 语句最基本的操作。当复制单个数据项时，直接将源数据项名称放在前面，将目标数据项名称放在后面。两者之间通过“TO”进行连接。例如，假设某程序中所定义的各项数据如下。

```
01  IN-STATUS      PIC X(5).
01  IN-GRP.
   05  IN-DATA1    PIC X(4).
   05  IN-DATA2    PIC 9(3).
01  OUT-STATUS     PIC X(5).
01  OUT-GRP.
    05  OUT-DATA1  PIC X(4).
    05  OUT-DATA2  PIC 9(3).
```

将以上数据作为示例数据，下面为通过 MOVE 语句对其中部分数据进行赋值的代码。

```
MOVE  'READY'  TO  IN-STATUS.
MOVE  'TEST'  TO  IN-DATA1.
MOVE  0  TO  IN-DATA2.
MOVE  SPACES  TO  OUT-STATUS
```

```
                    OUT-DATA1.
MOVE  ZERO  TO  OUT-DATA2.
```

以下代码通过 MOVE 语句实现了各变量之间的数据复制操作。

```
MOVE  IN-STATUS  TO  OUT-STATUS.
MOVE  IN-DATA1  TO  OUT-DATA1.
MOVE  IN-DATA2  TO  OUT-DATA2.
```

此外需要注意的是，当进行复制操作的两个数据项名称相同时，需要指定各数据项所在的数据组名。数据组名即各数据项所在的 01 级数据名。例如，假设在某程序中定义有以下数据。

```
01  SRC-GRP.
05  TEST-DATA1   PIC XX.
05  TEST-DATA2   PIC 9.
01  DES-GRP.
    05  TEST-DATA1  PIC XX.
    05  TEST-DATA2  PIC 9.
```

此处需要将 SRC-GRP 所包含的的两个数据项中的内容分别复制到 DES-GRP 下的两数据项中。注意到这两组数据项的名称分别对应相同，因此完成此项操作需要通过如下代码进行。

```
MOVE  TEST-DATA1  IN  SRC-GRP
      TO  TEST-DATA1  IN  DES-GRP.
MOVE  TEST-DATA2  IN  SRC-GRP
      TO  TEST-DATA2  IN  DES-GRP.
```

3.1.2 复制不同类型和长度的单个数据项

当复制单个数据项时，源数据项和目标数据项的类型和长度也可以不同。其中关于数据项的类型，在通常情况下，有以下两条规律。

- 任何类型的数据都可复制到字符型数据变量中。
- 数值型数据可复制到任何类型的数据变量中。

此处只需关注 MOVE 语句的用法，关于数据类型，将在下一章中详细讲解。以下代码反映了通过 MOVE 语句复制不同类型数据项的情况。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  DIFFER-TYPE.
 AUTHER.  XXX.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE   SECTION.
 01  CHAR-DATA     PIC  X(5).
 01  INT-DATA      PIC  9(5).
 01  EDIT-DATA     PIC  $$$999.
*
 PROCEDURE   DIVISION.
      ......
      MOVE  INT-DATA   TO  CHAR-DATA.
      MOVE  EDIT-DATA  TO  CHAR-DATA.
      MOVE  INT-DATA   TO  EDIT-DATA.
      STOP  RUN.
```

关于数据项的长度，对于不同的数据类型而言情况是不同的。当所复制的对象为字符型

数据时，将按照从左至右的顺序依次对逐个字符进行复制。此时，当源数据项和目标数据项的长度不一致时，将分以下两种情况进行处理。

- 如果目标数据项的长度大于源数据项内容的长度，则在目标数据项的右边以空格填充。
- 如果目标数据项的长度小于源数据项内容的长度，则对右边超出的部分进行截断。

以下代码反映了通过 MOVE 语句复制不同长度字符型数据项的情况。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  DIFFER-LEN1.
 AUTHER.  XXX.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  SOURCE-CHAR   PIC  X(5).
 01  SHORT-CHAR    PIC  X(3).
 01  LONG-CHAR     PIC  X(7).
*
 PROCEDURE   DIVISION.
      ......
      MOVE  'ABCDE'   TO  SOURCE-CHAR.
      MOVE  SOURCE-CHAR  TO  SHORT-CHAR.
      MOVE  SOURCE-CHAR  TO  LONG-DATA.
        STOP  RUN.
```

以上代码执行后，其中的各项数据内容如下。

```
SOURCE-CHAR: 'ABCDE'
SHORT-CHAR: 'ABC'
LONG-CHAR: 'ABCDE'                          /*此处下划线代表空格*/
```

当所复制的对象为数值型数据时，将根据其实际所代表的数值进行复制。此时，当源数据项和目标数据项的长度不一致时，将分以下几种情况进行处理。

- 如果目标数据项的长度大于源数据项内容的长度，则在目标数据项的高位以 0 填充。
- 如果目标数据项的长度小于源数据项内容的长度，则对高位超出的部分进行截断。
- 如果数值含有小数部分，则需要将其作为浮点数进行相应处理。

以下代码反映了通过 MOVE 语句复制不同长度数值型数据项的情况。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  DIFFER-LEN2.
 AUTHER.  XXX.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  SOURCE-NUM    PIC   9(5).
 01  SHORT-NUM     PIC   9(3).
 01  LONG-NUM      PIC   9(7).
 01  FLOAT-NUM     PIC   9(3)V99.
 01  FLOAT-NUM01   PIC   9(5)V9(3).
 01  FLOAT-NUM02   PIC   99V9.
*
 PROCEDURE   DIVISION.
```

```
      ……
      MOVE  12345   TO  SOURCE-NUM.
      MOVE  SOURCE-NUM  TO  SHORT-NUM.
      MOVE  SOURCE-NUM  TO  LONG-NUM.
      MOVE  SOURCE-NUM  TO  FLOAT-NUM.
      MOVE  123.45   TO  FLOAT-NUM01.
      MOVE  123.45   TO  FLOAT-NUM02.
        STOP  RUN.
```

以上代码执行后，其中的各项数据内容如下。

```
SOURCE-NUM: 12345
SHORT-NUM: 345
LONG-NUM: 0012345
FLOAT-NUM: 345^00                    /*此处“^”符号表示虚拟小数点位置*/
FLOAT-NUM01: 00123^450
FLOAT-NUM02: 23.4
```

3.1.3 复制一组数据

通过 MOVE 语句也可对一组数据进行复制。当对一组数据进行复制时，MOVE 语句中的两个变量分别为数据组的名称，即 01 层数据项的名称。这两个 01 层数据项下所有同名的组内数据项将实现复制。对于一组数据的复制，主要有以下几点需要注意。

- 所复制的组内数据项必须同名，不同名的数据项将不被复制。
- 如果数据项在分组中，则分组名称也必须相同。
- 数据项在组内的顺序可以任意。
- 以 FILLER 命名的数据项将不被复制。

例如，以下为在 COBOL 数据部中定义的两组数据。为针对所讨论的问题，此处只给出数据层级和名称，而暂无需考虑通过 PIC 语句定义的数据类型和长度。这两组数据分别如下。

```
01  IN-RECORD.                      /*以下为定义的第一组数据*/
    05  COURSE-INFO.
        10  COR-NO.
        10  COR-NAME.
        10  COR-FEE.
   05   COURSE-LIST.
        10  C-COBOL.
        10  C-JCL.
        10  C-DB2.
        10  C-CICS.
……
01  DETAIL-LINE.                    /*以下为定义的第二组数据*/
   05   COURSE-LIST.
        10  C-COBOL.
        10  C-JCL.
        10  C-IMS.
   05   C-CICS.
   05   COR-INFO.
        10  COR-NO.
        10  COR-NAME.
        10  COR-FEE.
```

对于以上定义的两组数据，可以使用如下 MOVE 语句进行复制。

```
    MOVE  IN-RECORD  TO  DETAIL-LINE.
```

需要注意的是，根据前面讲解的组数据复制的基本原则，不是组内所有的数据都会被复制。在 IN-RECORD 组中，有以下几个数据项将不被复制。

COURSE-INFO 下的 3 个数据项都不会被复制。因为 IN-RECORD 组中的 COURSE-INFO 与 DETAIL-LINE 组中的 COR-INFO 名字并不相同。当上层分组名称不同时，即使分组内数据项名称相同，该数据项也不被复制。

C-DB2 将不被复制。因为 DETAIL-LINE 组中的相应位置为 C-IMS，同 C-DB2 名称并不相同，因此不能进行复制。C-CICS 将不被复制。因为 C-CICS 在 IN-RECORD 组中属于分组 COURSE-LIST 下的数据项。而在 DETAIL-LINE 组中，该数据项为一个独立的数据项，不属于任何分组。

3.2 PERFORM 语句

PERFORM 语句用于执行在 COBOL 中所编写的相应处理过程。这些处理过程通常在程序最后进行编写，且每一处理过程都有一个唯一的处理过程名。处理过程相当于 COBOL 中的函数，而 PERFORM 语句则用于调用这些函数。

例如，假设以下程序在过程部中存在两段相同的使用 MOVE 语句进行复制操作的代码。在程序中并未使用 PERFORM 语句，相应代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  NO-PERFORM.
 AUTHER.  XXX.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  IN-AREA.
     05   IN-DATA1   PIC X.
     05   IN-DATA2   PIC XX.
 01  R-NUM          PIC 9.
 01  SAVE-AREA.
     05   SAVE-REC1   PIC X.
     05   SAVE-REC2   PIC XX.
 01  S-NUM         PIC 9.
*
 PROCEDURE   DIVISION.
      ......
      MOVE  IN-DATA1  TO  SAVE-REC1.
      MOVE  IN-DATA2  TO  SAVE-REC2.
      MOVE  R-NUM  TO  S-NUM.
      ......
      MOVE  IN-DATA1  TO  SAVE-REC1.
      MOVE  IN-DATA2  TO  SAVE-REC2.
      MOVE  R-NUM  TO  S-NUM.
      ......
        STOP  RUN.
```

如果在该程序中将这部分复制代码编写为一个处理过程，放在程序的最后。这样，在程

序中便可以通过 PERFORM 语句多次调用该处理过程以简化编码，并使程序结构清晰。应用 PERFORM 语句的相应代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  WITH-PERFORM.
 AUTHER.  XXX.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  IN-AREA.
     05   IN-DATA1   PIC X.
     05   IN-DATA2   PIC XX.
 01  R-NUM          PIC 9.
 01  SAVE-AREA.
     05   SAVE-REC1   PIC X.
     05   SAVE-REC2   PIC XX.
 01  S-NUM          PIC 9.
*
        PROCEDURE   DIVISION.
        ......
        PERFORM  100-COPY-PROCESS.
        ......
        PERFORM  100-COPY-PROCESS.
        ......
         STOP  RUN.
 100-COPY-PROCESS.
       MOVE  IN-DATA1  TO  SAVE-REC1.
       MOVE  IN-DATA2  TO  SAVE-REC2.
       MOVE  R-NUM  TO  S-NUM.
```

第二段程序与第一段程序实现的功能相同。但第二段程序通过运用 PEROFRM 语句，显然要比第一段程序结构要清晰，代码编写量也较少。在实际开发中，最好将可能会反复用到的较长代码段都编写为处理过程，以养成良好的编程习惯。

此外，在 PERFORM 语句中还可通过 THRU 选项同时对多个处理过程进行执行。不过通常情况下，为规范编码，此方式下最多只对两个处理过程同时执行。并且第二个处理过程只含有一条 EXIT 语句。相应代码如下。

```
......
 PERFORM  100-TEST-PROC
   THRU  100-END-TEST-PROC.
......
 100-TEST-PROC.
   do something.
 100-END-TEST-PROC.
  EXIT.
```

最后，根据结合 PERFORM 语句中不同的选项及格式，还可以发展为其他几种相关语句。这些语句包括 PERFORM UNTIL 语句、线上 PERFORM 语句以及 PERFORM VARYING 语句。其中前两种语句将在流程控制一章的循环结构中讲解。最后一种语句将在 COBOL 中的表一章中讲解。

3.3 ACCEPT 和 DISPLAY 语句

ACCEPT 语句和 DISPLAY 语句也是在 COBOL 中常用的两条语句。尤其是当对程序进行调试时，这两条语句的灵活应用将在很大程度上方便调试的过程。关于程序的调试，将在后面章节中详细讲解。此处只重点对这两条语句本身进行讲解。

3.3.1 使用 ACCEPT 语句接收数据

ACCEPT 语句主要用于接收数据。该语句既可以接受由用户从终端输入的数据，也可以接收由系统产生的相应数据。所接收的数据将存放在 COBOL 数据部中所定义的相应变量之中。例如，以下程序中定义了一组变量，并通过 ACCEPT 语句接受用户输入的数据，存入相应变量中。相关代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  ACCEPT-ONE.
 AUTHER.  XXX.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  INPUT-AREA.
     05   IN-FIELD1   PIC X.
     05   IN-FIELD2   PIC XX.
     05   IN-FIELD3   PIC 9.
 ......
*
 PROCEDURE   DIVISION.
       ......
       ACCEPT  IN-FIELD1.
       ......
       ACCEPT  IN-FIELD2.
       ......
       ACCEPT  IN-FIELD3.
        STOP  RUN.
```

当用户从终端（键盘）上输入数据时，这些数据将直接通过 ACCEPT 语句保存到相应变量中。由于实际输入数据通常是以文件形式存放的，因此这种方式通常只用于输入少量的控制或调试信息。

使用 ACCEPT 语句接收系统数据通常主要为接受系统相应的日期或时间数据。这些数据主要包括以下几种。

- DATE：对应按月份计算的日期，数据格式为 yymmdd。其中 yy 代表年份，mm 代表月份，dd 代表该月中具体的哪一天。
- DAY：对应按一年中绝对天数计算的日期，数据格式为 yyddd。其中 yy 代表年份，ddd 代表该年中的绝对天数。例如，对于平年而言，12 月 31 日在此便通过 365 表示。
- DAY-OF-WEEK：对应一周之内的星期数。接收数据的格式应定义为 PIC 9，数值范围从 1 ~ 7。

- TIME：对应具体的时间，格式为 hhmmsshh。其中前两个 hh 代表小时数，mm 代表分钟数，ss 代表秒数，而最后两个 hh 表示 hundredths from midnigth。

例如，下面这段程序便首先定义了一组变量，用于保存按月份计算的日期数据。其后再通过 ACCEPT 语句接收相应数据并将数据保存到该组变量中，代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  ACCEPT-TWO.
 AUTHER.  XXX.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  TEST-DATE.
     05   TEST-YEAR    PIC 99.
     05   TEST-MONTH   PIC 99.
     05   TEST-DAY     PIC 99.
 ......
*
 PROCEDURE   DIVISION.
      ......
      ACCEPT  TEST-DATE  FROM  DATE.
      ......
      STOP  RUN.
```

3.3.2 使用 DISPLAY 语句输出数据

DISPLAY 语句用于对数据实现输出操作。在第一章中所举例的 Hello World 程序主要运用的就是 DISPLAY 语句。在该程序中，DISPLAY 语句用于直接输出“Hello World”字符数据。同时，DISPLAY 语句还可对变量进行输出，此时将输出变量中所包含的数据。例如，以下程序将分别输出程序中所设置的变量以及从用户处所接收的变量。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  DISPLAY-ONE.
 AUTHER.  XXX.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  TEST-AREA.
     05   TEST-DATA1   PIC X(5).
     05   TEST-DATA2   PIC X(5).
*
 PROCEDURE   DIVISION.
      MOVE  'SYS'  TO  TEST-DATA1.
      ACCEPT   TEST-DATA2.
      DISPLAY  TEST-DATA1.
      DISPLAY  TEST-DATA2.
        STOP  RUN.
```

如果用户所输入的数据为“USER”，则以上代码执行后，将有如下输出结果。

```
SYS
USER
```

使用 DISPLAY 语句还可同时对多项数据进行输出。这些数据既可为由引号引用的字符（直接数），也可为包含有数据的变量。各数据之间可以用空格或逗号隔开。例如，以下程序将通过 DISPLAY 语句分别对多个数据同时进行输出。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  DISPLAY-TWO.
 AUTHER.  XXX.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  TEST-AREA.
     05   T-LANG       PIC X(10).
     05   T-PLATFORM   PIC X(10).
*
 PROCEDURE   DIVISION.
      MOVE  'COBOL'  TO  T-LANG.
      MOVE  'MAINFRAME'  TO  T-PLATFORM.
      DISPLAY  'THIS  IS  ABOUT :' ,T-LANG.
      DISPLAY  T-LANG  'IS  RUNNED  ON'  T-PLATFORM.
        STOP  RUN.
```

以上代码执行后，将有如下输出结果。

```
THIS  IS  ABOUT  COBOL
COBOL  IS  RUNNED  ON  MAINFRAME
```

3.4 REDEFINES 语句

REDEFINES 语句用于对同一块内存区域进行重定义。也就是说，通过 REDEFINES 语句，可以定义多个指向同一内存区域的变量。例如，以下程序便通过 REDEFINES 语句定义了 3 个指向同一内存区域的变量，代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  RE-DEF.
 AUTHER.  XXX.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  TEST-AREA.
     05   TEST-A        PIC X(4).
     05   TEST-B  REDEFINES  TEST-A.
     05   TEST-C  REDEFINES  TEST-A.
*
 PROCEDURE   DIVISION.
      MOVE  'TEST'  TO  TEST-A.
      DISPLAY  TEST-A.
      DISPLAY  TEST-B.
      DISPLAY  TEST-C.
        STOP  RUN.
```

以上代码执行后，将有如下输出结果。

```
TEST
TEST
TEST
```

需要注意的是，REDEFINES 语句不能用于对 66 层、88 层数据进行重定义。同时，该语句也不能对文件节中的 01 层数据重定义。不过，对于工作存储节中的 01 层数据则是可以重定义的。在后面章节中，将会介绍如何通过该语句初始化 COBOL 中的表。

3.5 文件相关语句

COBOL 程序中实际用到的数据通常主要来源于文件。此处所说的文件主要是指常规文件。关于另一类常用的具有特殊组织结构的 VSAM 文件，将在后面章节中详细讲解。和文件相关的语句，最基本的是用作打开、关闭文件，以及对于文件的读写。下面分别进行讲解。

3.5.1 OPEN 和 CLOSE 语句

OPEN 语句用来实现对文件的打开操作。在对文件进行任何处理之前，必须通过 OPEN 语句打开。同时，OPEN 语句还可指定该文件在打开后是用于输入操作还是输出操作。例如，以下程序通过该语句实现了文件的打开操作。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  FILE-OPEN.
 AUTHER.  XXX.
*
 ENVIRONMENT  DIVISION.
 INPUT-OUTPUT  SECTION.
 FILE-CONTROL.
   SELECT  CUSTOMER-FILE
       ASSIGN  TO  S-SYSIN.
   SELECT  PRINT-FILE
       ASSIGN  TO  S-SYSOUT.
*
 DATA  DIVISION.
 FD  CUSTOMER-FILE.
 ......
 FD  PRINT-FILE.
 ......
*
 PROCEDURE   DIVISION.
    OPEN  INPUT    CUSTOMER-FILE
          OUTPUT   PRINT-FILE.
    ......
      STOP  RUN.
```

此外，OPEN 语句也可用于同时打开多个用于同一操作的文件。各文件之间可通过空格隔开。例如，以下 OPEN 语句将同时打开两个用于输入操作的文件。

```
OPEN  INPUT  IN-FILE1   IN-FILE2.
```

最后，对于使用 OPEN 语句打开文件，主要有以下几点需要注意。

- 文件名需要同环境部输入/输出节中由 SELECT 语句所指定的文件名一致。
- OPEN 语句可以出现在程序任何位置，但通常写在程序开头或第一条读写文件的语句之前。
- 如果在读写文件之前没有打开文件，程序将会非正常终止（ABEND）。

与 OPEN 语句相对应，CLOSE 语句用于关闭已打开的文件。CLOSE 语句同样也可出现在程序的任何位置。不过该语句通常写于程序结尾或最后一条读写文件的语句之后。例如，以下代码反映了在程序中对文件操作前后的大致结构。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  FILE-PERIOD.
 AUTHER.  XXX.
*
 ENVIRONMENT  DIVISION.
 INPUT-OUTPUT  SECTION.
 FILE-CONTROL.
   ......
*
 DATA  DIVISION.
 ......
*
 PROCEDURE   DIVISION.
      OPEN  INPUT    CUSTOMER-FILE
            OUTPUT    PRINT-FILE.
      ......
      CLOSE   CUSTOMER-FILE  PRINT-FILE.
        STOP  RUN.
```

3.5.2 READ 语句

READ 语句用于从文件中读取数据。使用 READ 语句读文件时，实际上是将文件中的数据读到由 FD 语句所定义的变量之中。读取文件结束后，便可以通过这些变量访问到文件中的数据信息了。以下代码反映了 READ 语句的常见用法。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  FILE-READ.
 AUTHER.  XXX.
*
 ENVIRONMENT  DIVISION.
 INPUT-OUTPUT  SECTION.
 FILE-CONTROL.
  SELECT  CUSTOMER-FILE
       ASSIGN  TO  S-SYSIN.
   ......
*
 DATA  DIVISION.
 FILE  SECTION.
 FD  CUSTOMER-FILE
     RECORDING MODE IS F.
 01  CUSTOMER-RECORD.
    05  CUST-NO      PIC 9(5).
    05  CUST-NAME    PIC X(10).
 WORKING-STORAGE  SECTION.
```

```
 01  EOF-FLAG          PIC X   VALUE  'N'.
 ......
*
 PROCEDURE   DIVISION.
      OPEN  INPUT    CUSTOMER-FILE.
      READ  CUSTOMER-FILE
            AT  END MOVE 'Y' TO  EOF-FLAG
      END-READ.
      PERFORM  100-READ-NEXT
           UNTIL  EOF-FLAG = 'Y'.
      CLOSE   CUSTOMER-FILE.
        STOP  RUN.
 100-READ-NEXT.
      process the file record
      READ  CUSTOMER-FILE
            AT  END MOVE 'Y' TO  EOF-FLAG
      END-READ.
```

以上 READ 语句中包含有 AT END 选项。该选项表示的意思是当读到文件末尾时将采取何种操作。此处所做的操作是将 EOF-FLAG 变量置为“Y”。

同时，由于文件中通常包含有多条逻辑记录，但使用 READ 语句每次只能读取一条。因此，通常是将 READ 语句置于一个循环结构体中，以顺次读取到文件中的每一条记录。如图 3.1 所示，为读取文件的处理过程。

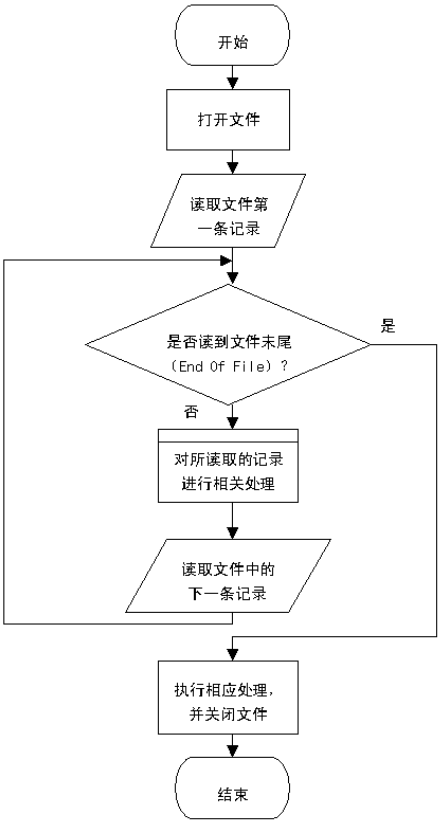

图 3.1　读取文件在程序中通常的处理过程

此外，通过结合不同的选项，READ 语句还有一些其他用法的格式。例如，以下语句将把文件数据读取到指定区域中（并非由 FD 语句定义的变量）。并且在读取结束和非结束时都采取相应操作。语句如下。

```
READ  CUSTOMER-FILE  INTO   OTHER-AREA
  AT  END  MOVE  'Y'  TO  EOF-FLAG
  NOT  AT  END  PERFORM  100-PROCESS-RECORD
END-READ.
```

3.5.3 WRITE 语句

WRITE 语句与 READ 语句相对应，主要用于对文件进行写入。同时，WRITE 语句还可将相应设备，如打印机等作为文件进行写入。此处实际上是通过 WRITE 语句对数据记录进行输出。以下为几种 WRITE 语句的常见使用方式。

```
WRITE  CUSTOMER-RECORD.                  /*以系统默认格式写入/输出记录*/

WRITE  PRINT-LINE                        /*在第 1 行数据之后写入/输出记录*/
     AFTER  ADVANCING 1 LINE.

WRITE  PRINT-LINE                        /*在下一页写入/输出 PAGE-HEADER 里的内容*/
FROM  PAGE-HEADER
AFTER  ADVANCING  PAGE.
```

需要注意的是，READ 语句后面为文件名，而 WRITE 语句后面则为记录名（变量名）。也就是说读取文件时需要指定文件名，而写入文件则需要指定所写入的记录名。

最后，以下这段程序将顺次读取客户文件 CUSTOMER-FILE 中的所有数据记录。并将客户编号大于 100 的相关记录进行输出打印，具体程序如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  FILE-PROCESS.
 AUTHER.  XXX.
*
 ENVIRONMENT  DIVISION.
 INPUT-OUTPUT  SECTION.
 FILE-CONTROL.
  SELECT  CUSTOMER-FILE
       ASSIGN  TO  S-SYSIN.
  SELECT  PRINT-FILE
       ASSIGN  TO  S-SYSOUT.
*
 DATA  DIVISION.
 FILE  SECTION.
 FD  CUSTOMER-FILE
    RECORDING MODE IS F.
 01  CUSTOMER-RECORD.
      05  CUST-NO       PIC 9(5).
      05  CUST-NAME     PIC X(10).
 FD  PRINT-FILE.
      RECORDING MODE IS F
      LABEL RECORDS ARE OMITTED
      RECORD CONTAINS 132 CHARACTERS
      DATA RECORD IS PRINT-LINE.
```

```
 01  PRINT-LINE          PIC X(132).
 WORKING-STORAGE  SECTION.
 01  EOF-FLAG            PIC X   VALUE  'N'.
 01  HEADING-LINE.
     05  FILLER          PIC  X(5)
                         VALUE  SPACES.
     05  FILLER          PIC  X(15)
                         VALUE  'CUSTOMER  NO'.
     05  FILLER          PIC X(7)
                         VALUE  SPACES.
     05  FILLER          PIC X(15)
                         VALUE  'CUSTOMER  NAME'.
     05  FILLER          PIC X(90).
 01  OUTPUT-LINE.
     05  FILLER          PIC  X(10)
                         VALUE  SPACES.
     05  PRT-NO          PIC  X(5).
     05  FILLER          PIC X(15)
                         VALUE  SPACES.
     05  PRT-NAME        PIC X(10).
     05  FILLER          PIC X(92).
*
 PROCEDURE   DIVISION.
      OPEN  INPUT    CUSTOMER-FILE
            OUTPUT  PRINT-FILE.
      READ  CUSTOMER-FILE
            AT  END MOVE 'Y' TO  EOF-FLAG
      END-READ.
      PERFORM  100-WRITE-HEADING.
      PERFORM  200-PROCESS-RECORDS
           UNTIL  EOF-FLAG = 'Y'.
      CLOSE   CUSTOMER-FILE
              PRINT-FILE.
        STOP  RUN.
 100-WRITE-HEADING.
      MOVE  HEADING-LINE  TO  PRINT-LINE.
      WRITE  PRINT-LINE.
 200-PROCESS-RECORDS.
      IF  CUST-NO > 100
          MOVE  CUST-NO  TO  PRT-NO.
          MOVE  CUST-NAME  TO  PRT-NAME.
          WRITE  PRINT-LINE  FROM  OUTPUT-LINE
      END-IF.
      READ  CUSTOMER-FILE
            AT  END MOVE 'Y' TO  EOF-FLAG
      END-READ.
```

3.6 本章回顾

本章主要讲解了 COBOL 语言中的各种常用语句。这些语句在今后各章节中都会经常用到，因此需要熟练掌握。

首先，本章讲解了 MOVE 语句和 PERFORM 语句。其中 MOVE 语句主要用于数据的

复制操作，PERFORM 语句主要用于执行 COBOL 中的处理过程。此处所说的处理过程相当于在 COBOL 中编写的函数。这两条语句十分重要，将贯穿于 COBOL 语言的始终，需要牢固掌握。

然后，分别介绍了 ACCEPT 语句、DISPLAY 语句以及 REDEFINES 语句。其中 ACCEPT 语句用于接收数据，DISPLAY 语句用于输出数据。这两条语句是一个相对的概念。REDEFINES 语句则用于对同一内存区域进行重定义。学习这部分内容，需要熟练掌握这 3 种基本语句的格式及功能。

最后，讲解了与文件相关的基本语句。主要包括用于打开文件的 OPEN 语句、用于关闭文件的 CLOSE 语句、用于读取文件的 READ 语句以及用于写入文件的 WRITE 语句。学习这部分内容，关键需要掌握在 COBOL 中关于文件处理的大致流程。

第 4 章 基本数据类型

本章主要讲解编写 COBOL 程序时，所要涉及到的各种数据类型。本章所包含的数据类型有字符类型、整型数类型、浮点数类型。此外，还包括 COBOL 所特有的 Signed Numbers 符号类型和 Numeric Edited Fields 格式输出类型。本章重点需要掌握 Numeric Edited Fields 格式输出类型。

4.1 基本数据类别

在学习数据类型之前，有必要对 COBOL 中的基本数据类别进行一个大致的了解。COBOL 中的数据类别主要可以分为变量、常量、文字和结构。下面分别对这 4 个类别进行讲解。

4.1.1 变量

COBOL 中所谓的变量就是值可以在程序中被改变的数据。变量通常包含两项内容，即变量名称和变量长度。变量在数据部进行定义，在过程部的具体程序中进行处理，处理中直接引用变量名称。变量的定义格式通常如下。

```
01   variable          PIC X ( n ).
       ↑                       ↑
     变量的名称              变量的长度
```

下面给出一个具体例子来进行说明。例如，某家公司要改变其公司的名称，这时可以定义两个变量。其中一个变量保存公司名称，另一个变量保存该公司新名称的讨论结果。当新名称确定后，就需要将讨论结果中的内容复制到公司名称变量中。代码如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID     COMP-NAME-PROG.
 AUTHER         XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
```

```
WORKING-STORAGE  SECTION.
01    COMPANY-NAME         PIC  X(20).        /*定义公司名称变量*/
01    DISCUSSED-NAME       PIC  X(20).        /*定义讨论结果变量*/
*
PROCEDURE   DIVISION.
     ……                                      /*此处省略讨论过程的代码*/
DISPLAY-NEW-NAME.
     MOVE  DISCUSSED-NAME  TO  COMPANY-NAME.
     DISPLAY  COMPANY-NAME.
     STOP  RUN.
```

以上代码定义的两个变量都为字符型变量。根据前面学习的知识我们知道，字符型变量中既可以为字母也可为数字。也可以定义数字型变量。定义方式如下。

```
01   numeric-var    PIC 9(n).
```

4.1.2 常量

常量通常可以分为普通常量和象征常量两种。不同于变量，常量的内容不可以被更改。即常量在程序中只能唯一对应一项内容。普通常量的该项内容是在定义的同时通过 VALUE 语句给出的。象征常量的该项内容直接通过常量名得出。

1．普通常量

例如，COBOL 程序常用来打印报表，此时报表的名称就应该被定义为一个常量。设某一公司财务报表，在输出打印时首先应包含表头，即该报表的名称。此外，还应包含该报表由谁完成，即提交该报表的职员姓名。因此，这里需要定义两个常量，分别为报表名和职员姓名。其定义方式如下。

```
DATA DIVISION.
……
01   REPORT-HEAD           PIC X (20)  VALUE  'FINANCE REPORT'.
……
01   REPORTER-NAME         PIC X (20)  VALUE  'ROBERT'.
```

以上这段代码完成了对报表名称和报表完成者姓名这两个常量的定义。通过 VALUE 语句在定义的同时对其赋值。其中报表名称常量被赋值为“FINANCE REPORT”，表示该报表为公司财务报表。报表完成者姓名常量被赋值为“ROBERT”，表示该报表是由 ROBERT 完成的。以上这两个常量通常只用于打印输出，在程序中不可被更改。

以上定义的是两个字符型常量，同样也可以定义数字型常量。此种情况下常常涉及到对一个固定数值的反复运算，如对于税率或者存款利息率的计算等。在数学运算中，常常将圆周率定义为一个常量。下面的例子说明了这一点。

假设圆的半径为 3，圆周率为 3.14。现要求用 COBOL 编写程序，计算该圆的周长和面积。这里将圆周率设为一个常量 RATE。完成以上功能的代码如下。

```
IDENTIFICATION   DIVISION.
PROGRAM-ID       COMPUTE-CIRCLE.
AUTHER           XXX.
*
ENVIRONMENT   DIVISION.
*
```

```
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01      RADIUS                  PIC  9        VALUE  3.
 01      RATE                    PIC  9V99     VALUE  3.14.       /*此处定义圆周率常量*/
 01      LENGTH                  PIC  99V99.
 01      AREA                    PIC  99V99.
 01      EDITED-LENGTH           PIC  99.99.
 01      EDITED-AREA             PIC  99.99.
*
 PROCEDURE   DIVISION.
*
 COMPUTE-STATEMENT.
    COMPUTE  LENGTH = RADIUS * RATE *2.
      COMPUTE  AREA = RATE * RADIUS * RADIUS.
*
 RESULT-STAEMENT.
      MOVE   LENGTH TO EDITED-LENGTH.
      MOVE   AREA  TO EDITED-AREA.
      DISPLAY   "LENGTH = ", EDITED-LENGTH.
      DISPLAY   "AREA = ", EDITED-AREA.
STOP  RUN.
```

该代码执行后，应该得到以下结果。

```
LENGTH = 18.84
AREA = 28.26
```

上例中将圆周率定义为了一个常量，该常量名称为 RATE。将圆周率（或者其他常用固定数值）定义为常量主要有以下两点好处。

- 在程序中不用每次都写出具体数值，而可以使用常量名称取代之。特别是对于数值较长的情况下，使用一个长度较短的常量名取代是很方便的。
- 当该固定数值根据实际情况需要更改时，可以直接在定义常量的 VALUE 语句后更改。如以上若需将圆周率精确为 3.1415926，则直接将下面第 1 条语句改为第 2 条语句就可以了。

```
01     RATE                   PIC  9V99   VALUE  3.14.          /*第 1 条语句*/
01     RATE                   PIC  9V99   VALUE  3.1415926.     /*第 2 条语句*/
```

若此处不使用常量定义圆周率，则需要在程序中每次出现该圆周率数值处进行修改，相当麻烦。此外，程序中还涉及到了变量 EDITED-LENGTH 和 EDITED-AREA。这两个变量属于 Numeric Edited Fields 格式输出类型，将在本章后面小节中进行详细讲解。

2．象征常量

COBOL 中还存在一类特殊的常量，称为象征常量。象征常量是通过 COBOL 中的关键字表示的。使用象征常量和使用普通常量方式一样。只是象征常量不用定义，可以直接使用。COBOL 中象征常量名称及其对应的象征内容如表 4.1 所示。

表 4.1　　象征常量及其对应关系

象 征 常 量	象征常量的内容
ZERO（ZEROS，ZEROES）	数字 0 或者字符“0”
SPACE（SPACES）	空格
HIGH-VALUE（HIGH-VALUES）	将对应的二进制码全部置为 1

续表

象 征 常 量	象征常量的内容
LOW-VALUE（LOW-VALUES）	将对应的二进制码全部置为 0
QUOTE (QUOTES)	引号
ALL	相应字符连接而成的字符串

以上表格里括号中的名称表示其复数用法。对于以上常用的象征常量，以下分别举例进行说明。首先，假设有一数据（即一个变量）的定义如下。

```
01   TEST-DATA      PIC X(10).
```

对应以上定义的数据，下面这条语句使用了象征常量 SPACES。该语句的功能是将 TEST-DATA 全部置为空格。由于 TEST-DATA 长度为 10 个字符，因此该语句执行后，TEST-DATA 将含有 10 个连续的空格。代码如下。

```
MOVE  SPACES TO TEST-DATA.
```

下面这条语句使用了象征变量 ALL。该语句执行后，TEST-DATA 中的内容应该为“ABCABCABCA”。代码如下。

```
MOVE  ALL  'ABC'  TO TEST-DATA.
```

下面这条语句使用了象征变量 LOW-VALUES。该语句将 TEST-DATA 中当前内容所对应的二进制码全部置为 0。这条语句常用于在处理 TEST-DATA 数据前，将里面的内容清空。语句如下。

```
MOVE  LOW-VALUES TO TEST-DATA.
```

4.1.3 直接数

直接数相当于一组字符串，并且该字符串的内容就是其本身。直接数在程序中直接给出，而不需要在数据部中进行定义。直接数可以分为数字型直接数和字符型直接数两大类。其中数字型直接数在程序中直接给出具体数字。字符型直接数则是在程序中通过引号表示的。

1．数字型直接数

数字型直接数只能包含数字，正负号和小数点，并且最多只能包含 18 个数字。当数字前面没有出现符号时，系统默认为正号。此外，小数点不能为数字型直接数的最后一个字符，否则将会和语句结束符相混淆。数字型直接数的值为其代数值。以下是几个合法的数字型直接数。

```
5
1024
0
+33
-70
80.23
-30.27
```

以下为非法的数字型直接数。

```
89A                          //数字型直接数不能包含字母“A”
82.                          //小数点不能为最后一个字符
123456789987654321000        //数字型直接数不能超过 18 个数字
'123'                        //这里通过引号扩起来，因此为字符型直接数，而不是数字型直接数
```

数字型直接数通常用于算术运算，有时也可以作为编号信息，如账号等。下面几条语句说明了程序中哪些为数字型直接数，以及数字型直接数的用法。

```
 MOVE 2  TO  NUM1.                              /*此处 2 为数字型直接数*/
 MOVE 3  TO  NUM2.                              /*此处 3 为数字型直接数*/
 ADD  5  TO  NUM1.                              /*此处 5 为数字型直接数*/
 COMPUTE NUM3 = ( NUM1 + NUM2 ) * 100 / 2.      /*此处 100 和 2 均为数字型直接数*/
 IF ACCOUNT-CODE  ( NUM3 ) = 013745             /*此处 013745 为数字型直接数*/
       THEN  DISPLAY ' ACCOUNT CODE FOUND! '
 END-IF.
```

2．字符型直接数

字符型直接数是通过引号表示的。引号既可以为单引号，也可以为双引号。字符型直接数最多包含 160 个字符。字符型直接数既可以包含数字，也可以包含字母。此外，字符型直接数还可以包含其他一些字符，如空格、标点符号等。几乎所有的字符都可以包含在字符型直接数中。不过，当需要在里面包含引号字符时，该引号必须重复出现两次。以下为几个合法的字符型直接数。

```
'ABC'
'Hello World! '
"DOUBLE QUOTE"
'I  love  "COBOL"  ! '
```

字符型直接数通常用在条件判断中对某一变量的比较，以及输出信息等。下面几行代码说明了程序中哪些为字符型直接数，以及字符型直接数的用法。

```
 IF ACCOUNT-NAME  (NUM3) = 'TOM'     /*此处“TOM”为字符型直接数*/
   DISPLAY  ' FOUNDED! '             /*此处“FOUNDED!”为字符型直接数*/
```

最后需要特别注意的是，以下两个数据是不等价的。

```
3.14          //此为数字型直接数，表示一个具体数值。
'3.14'        //此为字符型直接数，表示一个字符串。
```

4.1.4　结构体

一组相关数据可以构成一个结构体。严格的说，结构体实际上也包含两种类型，一种为单元结构体，另一种为组结构体。单元结构体是指只包含一个数据项的结构体。组结构体指包含有多个数据项的结构体。

例如，下面这段代码是对银行账户进行操作的。假设这里的账户号前 4 位为统一编号，后 10 位为对应每位储户的特定编号。这里首先读入账户文件信息，然后再分别将账户姓名和账户的特定编号保存到程序所定义的变量中。代码如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID       ACCOUNT-PROG.
 AUTHER           XXXX.
*
```

```
ENVIRONMENT   DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT  ACCOUNT-FILE
       ASSIGN  TO  ACC-SYSIN.
   SELECT  PRINT-FILE
       ASSIGN  TO  ACC-SYSOUT.
*
DATA  DIVISION.
FILE  SECTION.
FD   ACCOUNT-FILE                          /*此处定义输入文件*/
     RECORDING  MODE  IS  F
     RECORD  CONTAINS  40  CHARACTERS.
01   ACCOUNT-RECORD.
     05   ACCOUNT-NAME.                   /*此处定义账户姓名结构体*/
          10   FIRST-NAME    PIC X (10).
          10   FILLER        PIC X.
          10   LAST-NAME     PIC X(14).
     05  ACCOUNT-NUMBER.                  /*此处定义账户号结构体*/
          10   GENENAL-NUM   PIC 9(5).
          10   SPECIAL-NUM    PIC 9(10).
FD   PRINT-FILE                            /*此处定义输出文件*/
     RECORDING  MODE  IS  F
     LABEL   RECORDS  ARE  OMITTED
     RECORD   CONTAINS   132   CHARACTERS
     DATA    RECORD  IS  PRINT-LINE.
01   PRINT-LINE    PIC  X(132).
WORKING-STORAGE  SECTION.
01   OUTPUT-LINE.                         /*此处定义输出的格式*/
     05  FILLER   PIC X(10)
                  VALUE SPACES.
     05  PRINT-NAME   PIC X(25).
     05  FILLER   PIC X(5)
                  VALUE SPACES.
     05  PRINT-NUM    PIC 9(10).
     05  FILLER   PIC X(82)
                  VALUE SPACES.
01    EOF-FLAG    PIC X(1)
                   VALUE 'N'.              /*定义文件结束标志，并置初值'N'*/
*
PROCEDURE DIVISION.
000-PREPARE-REPORT.
     OPEN   INPUT   ACCOUNT-FILE
            OUTPUT  PRINT-FILE.
     READ   ACCOUNT-FILE
            AT  END  MOVE 'Y' TO EOF-FLAG
     END-READ.                             /*依次读取账户文件的每条记录,直到读完全部记录*/
     PERFORM 100-PROCESS-RECORDS
      UNTIL  EOF-FLAG = 'Y'.
     CLOSE   ACCOUNT-FILE
             PRINT-FILE.
       STOP RUN.
100-PROCESS-RECORDS.
     MOVE ACCOUNT-NAME  TO  PRINT-NAME.    /*将组结构体 ACCOUNT-NAME 进行复制操作*/
     MOVE SPECIAL-NUM  TO  PRINT-NUM.      /*将单元结构体 SPECIAL-NUM 进行复制操作*/
```

```
        WRITE  PRINT-LINE
          FROM  OUTPUT-LINE.
```

这里由于涉及到对文件的操作，因此代码较多。该段代码重点在于对组结构体 ACCOUNT-NAME 和单元结构体 SPECIAL-NUM 的操作。具体实现的功能流程如下。

- 顺次读取账户文件 ACCOUNT-FILE 的每条记录，直到读完全部记录。
- 每读取一条记录，将组结构体 ACCOUNT-NAME 复制到输出变量 PRINT-NAME 中。由于组结构体 ACCOUNT-NAME 的定义如下。

```
    05   ACCOUNT-NAME.
           10   FIRST-NAME     PIC X (10).
           10   FILLER         PIC X.
           10   LAST-NAME      PIC X(14).
```

而输出变量 PRINT-NAME 的定义如下。

```
05  PRINT-NAME   PIC X(25).
```

因此，这里实际上相当于将文件中保存的账户姓名 FIRST-NAME 和 LAST-NAME 进行了合并。合并后的完整姓名被存储到了 PRINT-NAME 变量中。

- 每读取一条记录的同时，还将单元结构体 SPECIAL-NUM 中的内容复制到了输出变量 PRINT-NUM 中。
- 将输出信息通过 WRITE 语句写入输出文件中进行输出。

通过该段代码，需要掌握的关于结构体的概念主要有以下两点。

（1）组结构体包含了单元结构体，单元结构体包含在组结构体中。

例如，对于组结构体 ACCOUNT-NAME 而言，包含了以下 3 个单元结构体。

```
           10   FIRST-NAME     PIC X (10).
           10   FILLER         PIC X.
           10   LAST-NAME      PIC X(14).
```

单元结构体 SPECIAL-NUM 的定义代码如下。

```
05  ACCOUNT-NUMBER.
           10   GENENAL-NUM   PIC 9(5).
           10   SPECIAL-NUM    PIC 9(10).
```

由以上代码，可知其包含在组结构体 ACCOUNT-NUMBER 中。该组结构体同时还包含另一个单元结构体 GENENAL-NUM。

（2）对组结构体进行操作时，实际上是对其包含的所有单元结构体进行的统一操作。即对组结构体的操作是将其所包含的各单元结构体作为一个整体来操作的。

例如，对于以上代码中的语句。

```
MOVE ACCOUNT-NAME  TO  PRINT-NAME.
```

实际上是将 ACCOUNT-NAME 所包含的 3 个单元结构体：FIRST-NAME、FILLER 和 LAST-NAME 中数值都复制到了变量 PRINT-NAME 中。

同样，若假设存在以下这个变量。

```
 01    TEST-DATA     PIC X (40).
```

下面这条语句将整个账户信息（包括账户姓名和账户号）一起复制到该 TEST-DATA 变量中。语句如下。

```
      MOVE  ACCOUNT-RECORD  TO  TEST-DATA.
```

4.2 字符类型

字符类型是 COBOL 中最常用的数据类型。在定义字符类型的数据时，使用 PIC X 语句进行定义。X 表示该数据位置为一任意字符。以下代码表明了字符类型数据的用法。

```
      IDENTIFICATION   DIVISION.
      PROGRAM-ID       PICX-PROG.
      AUTHER           XXX.
     *
      ENVIRONMENT   DIVISION.
     *
      DATA  DIVISION.
      WORKING-STORAGE  SECTION.
      01    CHAR-1      PIC  X .
      01    CHAR-2      PIC  XXX.
      01    CHAR-3      PIC  X(8).
     *
      PROCEDURE   DIVISION.
           MOVE  '! '  TO  CHAR-1.
           MOVE  'P2P'  TO  CHAR-2.
           MOVE  'ABCDABCD'  TO  CHAR-3.
            STOP  RUN.
```

该段代码执行后，程序中所定义的 3 个字符类型的变量及其所保存的结果分别如下。

```
CHAR-1:  !
CHAR-2:  P2P
CHAR-3:  ABCDABCD
```

由此可见，字符类型数据几乎可以包含任何字符组成的数据。并且，当该数据由多个字符组成时，有两种方式可以对其进行定义。其中一种是使用连续出现的 X 进行定义，X 出现的次数和字符的个数一致。另一种是在 X 后面加上一个括号，括号中填写的数值即该数据所含字符的个数。

实际上，由多个连续字符组成的数据也可称作字符串数据。字符串数据在 COBOL 中属于一个比较特殊的数据类型。关于字符串数据类型的特点，以及基于字符串的操作将在字符串一章中进行单独详细讲解。

4.3 整型数类型

上节所讲到的字符类型数据是不能用于进行算术运算的。COBOL 中能进行算术运算的数据通常有两类。一类为整型数，另一类为浮点数。当然，由整型数和浮点数各自所对应的 Signed Numbers 符号类型数据也是可以进行算术运算的。但 Signed Numbers 符号类型数据实际上就是在整型数和浮点数前面加上了正负号而已。因此，整型数和浮点数才是用于进行算

术运算的最基本的两类数据。

本节中只讲解整型数数据类型。整型数是通过 PIC 9 语句进行定义的。这里的“9”同定义字符类型数据所用到的“X”是类似的。以下语句说明了整型数的定义和用法。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID       PIC9-PROG.
 AUTHER           XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    INT-1      PIC  9 .
 01    INT-2      PIC  999.
 01    INT-3      PIC  9(3).
*
 PROCEDURE   DIVISION.
      MOVE  5  TO  INT-1.
      MOVE  100  TO  INT-2.
      COMPUTE  INT-3 = INT-1 * INT-2.
      DISPLAY  'INT-1:' ,INT-1.
      DISPLAY  'INT-2:',INT-2.
      DISPLAY  'INT-3:',INT-3.
       STOP  RUN.
```

以上代码执行后，输出结果如下。

```
INT-1: 5
INT-2: 100
INT-3: 500
```

通过以上代码可以看出，当整型数由多个位数组成时，依然有两种定义方式。一种为定义连续出现的 9，9 出现的次数和该整型数位数相同。另一种为在 9 后面加上一个括号，括号里填写的数值大小为该整型数的位数个数。并且，整形数可以用于算术运算，这一点是最重要的。

实际上，定义整型数类型时所用到的定义符号“9”和定义字符类型用到的“X”是可以同时出现的。当“9”和“X”同时出现时，该数据实际上为一字符类型的数据。但是，对于出现“9”的位置，只能存放数字。以下代码说明了这一点。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID       PIC9X-PROG.
 AUTHER           XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    MIX       PIC  XXX9(3)XXX .    /*该定义语句中“9”和“X”同时出现*/
*
 PROCEDURE   DIVISION.
      MOVE  'ABC123ABC'  TO  MIX.     /*此条语句正确*/
      MOVE  '123ABC123'   TO  MIX.    /*此条语句错误。根据定义，中间 3 个位置只能存放数字*/
       STOP  RUN.
```

4.4 浮点数类型

浮点数即通常所说的小数。浮点数也是可以用于算术运算的。在定义浮点数时，通过定义符号 V 表示小数点的位置。关于浮点数的定义和用法可以通过以下代码说明。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID       PICV1-PROG.
 AUTHER           XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    FLOAT-1      PIC  99V99 .
 01    FLOAT-2      PIC  9V9(7).
 01    FLOAT-3      PIC  9V99.
*
 PROCEDURE   DIVISION.
      MOVE  10.05  TO  FLOAT-1.
      MOVE  3.1415926  TO  FLOAT-2.
      COMPUTE  FLOAT-3 = ( 5 + 6 ) / 4
       STOP  RUN.
```

该段代码执行后，程序中所定义的 3 个浮点数中的内容分别如下。

```
FLOAT-1: 10.05
FLOAT-2: 3.1415926
FLOAT-3: 2.75
```

此外还需特别注意的是，定义浮点数的小数点所使用的符号“V”并不占用实际存储空间。也就是说，程序只是记录了小数点的位置，但并不单独开辟一个字节空间用来存放小数点这个符号。因此，对浮点数直接进行输出时，是看不到小数点的。下面这段代码说明了这一问题。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID       PICV2-PROG.
 AUTHER          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    FLOAT      PIC  99V99 .
*
 PROCEDURE   DIVISION.
      MOVE  10.05  TO  FLOAT.
      DISPLAY FLOAT.
       STOP  RUN.
```

该段代码运行后，屏幕上将输出“1005”，而不是所期望的“10.05”。因为，定义小数点所使用的符号“V”只用来表示小数点的位置，而不能用来存放小数点符号。此处的“V”只是相当于一个虚拟的小数点。至于如何输出小数点，本章将在后面的“Numeric Edited Fields 格式输出类型”一节中进行详细讲解。

4.5 Signed Numbers 符号类型

Signed Numbers 符号类型是针对正负数而言的。其中既可以包含正数，也可以包含负数。该数据类型的定义方式如下。

```
01  S-NUM    PIC  S99V99   VALUE  -12.74.
```

根据前面所学知识，可以发现该数据类型的定义方式和数字型变量的定义方式是有些相似的。以下为普通数字型变量的定义语句。

```
01  NUM    PIC  99V99   VALUE  12.74.
```

对比以上两条语句可以发现，Signed Numbers 符号类型是建立在数字型变量的定义基础之上的。定义 Signed Numbers 符号类型时，就是在通常的 PIC 9x 语句中的 9 前面加上一个“S”。这里的“S”即代表“Signed”，表示该数据是有符号的。

4.5.1 Signed Numbers 符号类型的作用

Signed Numbers 符号类型主要用于保存有符号的正负数。若某一数字型变量没有被定义为 Signed Numbers 符号类型，则该变量只能保存正数。任何赋值给该变量的数，即使本身为负数，最终也将转换为正数存入其中。

以下两段代码将 Signed Numbers 符号类型和普通数字型变量进行了对比。通过对比，显示了 Signed Numbers 符号类型的作用。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID       SNUM-PROG.
 AUTHER           XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    S-NUM                PIC  S99.       /*定义 Signed Numbers 符号类型*/
*
 PROCEDURE   DIVISION.
      MOVE  -10  TO  S-NUM.
      ADD    10  TO  S-NUM.
      DISPLAY  'S-NUM: ' , S-NUM.
       STOP  RUN.
```

该段代码执行后，屏幕上将显示以下信息。

```
S-NUM: 0
```

如果不将 S-COST 定义为一个 Signed Numbers 符号类型，而是定义为一个普通的数字型变量。那么，将得到以下不同的结果。代码如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID       NUM-PROG.
 AUTHER           XXX.
*
```

```
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    NUM                PIC   99.        /*定义为普通数字型变量*/
*
 PROCEDURE   DIVISION.
      MOVE  -10  TO  NUM.
      ADD    10   TO  NUM.
      DISPLAY  'NUM:' , NUM.
       STOP  RUN.
```

该段代码执行后，屏幕上将显示以下信息。

```
NUM: 20
```

4.5.2 Signed Numbers 符号类型的输出

此处需要说明的是，定义 Signed Numbers 符号类型时，PIC 语句后的“S”是不占用存储空间的。“S”只用来表示此处可以有“+”，“–”号。这点和之前学过的表示小数点的“V”是比较类似。二者都是不占用实际存储空间的。

因此，Signed Numbers 符号类型虽然保存了正负号，但并不能对正负号进行输出。程序相当于只是将该符号记录了下来，但并不显示出来。下面这段代码说明了这个概念。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID       NUM-PROG.
 AUTHER           XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    NUM                PIC   S99.        /*定义 Signed Numbers 符号类型*/
*
 PROCEDURE   DIVISION.
      MOVE  10  TO  NUM.
      DISPLAY  '+10 DISPLAYED BY:',NUM.
      MOVE  -10  TO  NUM.
      DISPLAY  '-10 DISPLAYED BY:', NUM.
       STOP  RUN.
```

该段代码执行后，将输出以下信息。

```
+10  DISPLAYED BY 10
-10  DISLAYED BY 10
```

那么，如何使程序能够输出正负号呢？关于这一点，将在接下来的一节“Numeric Edited Fields 格式输出类型”中进行详细的讲解。

4.6 Numeric Edited Fields 格式输出类型

Numeric Edited Fields 格式输出类型主要用于进行特定的格式输出。该类型是 COBOL 实际开发中最常用到的数据类型。因此本节内容十分重要。

4.6.1 货币格式

Numeric Edited Fields 格式输出类型中的货币格式常用于生成报表、工资单、账单等。货币格式通常是以 Signed Numbers 符号类型为原型进行的输出格式转换。货币格式的数据需要在具体数字前加上货币符号“$”。

1. 单一“$”号的货币格式

生成货币格式的基本方式是首先定义一个带有“$”符号的 Numeric Edited Fields 格式输出类型。之后，再将一个存有具体数值的数字型变量 MOVE 到该中。有时币值会出现负值，这时便需要对前面讲到的 Signed Numbers 符号类型进行 MOVE 操作了。关于负值的情况，将在下一节中的算术符号格式中进行讲解。这里只讨论正值的情况。

下面这段代码说明了货币格式的常见用法。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID       EDIT-NUM.
 AUTHER           XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    PRICE                PIC   99.
 01    EDITED-PRICE          PIC  $99.        /* 此处定义货币格式*/
*
 PROCEDURE   DIVISION.
      MOVE  48  TO  PRICE.
      MOVE  PRICE  TO  EDITED-PRICE.
      DISPLAY  'SOURCE  PRICE:' , PRICE.
      DISPLAY  'REAL  PRICE:' ,  EDITED-PRICE.
       STOP  RUN.
```

该程序运行后，在屏幕上输出的结果如下所示。

```
SOURCE  PRICE: 48
REAL  PRICE: $48
```

通过以上代码可以看出，货币格式实际上是由两部分组成的。一部分为前面的货币符号“$”，另一部分为该货币的实际币值大小，是一个具体数字。

2. 多个“$”号的货币格式

货币格式还可在前面定义多个连续的“$”货币符号。此种情况下，当货币格式的总长度超过实际长度时，前面多余的“$”将被空格取代。同时长度范围内的“$”被实际数值取代。当长度不足时，保留第一个“$”符号，同时实际数值被截取，以满足长度匹配。以下代码说明了这种用法。

```
......
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    SOURCE-NUM          PIC    9999.
```

```
 01    EDITED-NUM1          PIC  $$$$99.
 01    EDITED-NUM2          PIC   $$99.
*
PROCEDURE   DIVISION.
      MOVE  1234  TO  SOURCE-NUM.
      MOVE  SOURCE-NUM  TO  EDITED-NUM1.
      MOVE  SOURCE-NUM  TO  EDITED-NUM2.
      DISPLAY  'SOURCE-NUM:', SOURCE-NUM.
      DISPLAY  'EDITED-NUM1:' , EDITED-NUM1.
      DISPLAY  'EDITED-NUM2:' ,  EDITED-NUM2.
       STOP  RUN.
```

此处省略号省略了标志部和环境部的代码。为节省篇幅，以后也经常会省略一些重复的代码，只保留针对所讨论问题的部分关键代码。后面将不再对此另加说明。

该程序执行后，屏幕上应该显示如下。

```
SOURCE-NUM: 1234
EDITED-NUM1:  $1234
EDITED-NUM2: $234
```

以上代码中，货币的币值数据保存在 SOURCE-NUM 变量中，被定义为 PIC 9999，数据位数为 4。并且，该数据通过 MOVE 语句被初始化为数值 1234。当将该数值数据转变为货币型数据时，由于在前面还要加上“$”，因此实际长度 5。

货币型数据 EDITED-NUM1 被定义为 PIC $$$$99，数据位数为 6。该货币型数据的总长度超过了实际长度 5，因此前面多余的“$”将被空格所取代。所以 EDITED-NUM1 中的数据应该为“$1234”，而不是“$$1234”。

与之相应，货币型数据 EDITED-NUM2 被定义为 PIC $$99，数据位数为 4。该货币型数据的总长度小于实际长度 5，因此实际数值要被截取。故 EDITED-NUM1 中的数据应该为“$234”，而不是“$1234”。实际上，也可以通过如下简便方式来判断前缀为多个连续“$”号的货币型数据。

- 将实际数值从右至左依次填入 Numeric Edited Fields 对应的定义字符中。该定义字符为 9 或者$。
- 在最终生成的数据中，至少保留并且仅保留一个“$”号。多余的$号用空格取代，$号不足时截取数值数据。

4.6.2 算术符号格式

算术符号格式主要是针对前面学习的 Signed Numbers 符号类型而言的。算术符号格式的原始数据通常为 Signed Numbers 数据。并且，算术符号格式的具体输出格式也是根据其原始数据的正负号而确定的。算术符号格式可分为两种类型的定义方式。其中一种通过 CR 和 DB 进行定义，另一种通过正负号进行定义。下面分别进行讲解。

1．通过 CR 和 DB 定义算术符号格式

通过 CR 和 DB 定义算术符号格式数据通常应用在银行软件开发项目上。CR 表示 Credit，即信贷的意思；DB 表示 Debit，即借贷的意思。由于此处涉及到财务方面的内容，因此 CR 和 DB 通常是配合货币符号“$”一起使用。CR 和 DB 定义在数据的末尾。并且，CR 和 DB

都只在原始数值为负值的时候才显示输出。关于 CR 和 DB 的应用示例如下。

```
......
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    CREDIT-DATA-POS           PIC   S9999.
01    CREDIT-DATA-NEG           PIC   S9999.
01    DEBIT-DATA-POS            PIC   S9999.
01    DEBIT-DATA-NAG            PIC   S9999.
01    EDITED-CREDIT         PIC   $9999CR.
01    EDITED-DEBIT          PIC   $9999DB.
*
PROCEDURE   DIVISION.
INITIAL-PROCESS.
     MOVE  1024  TO   CREDIT-DATA-POS.
     MOVE  -1024  TO   CREDIT-DATA-NAG.
     MOVE  3270  TO   DEBIT-DATA-POS.
     MOVE  -3270  TO   DEBIT-DATA-NEG.
SHOW-POSITIVE.
     MOVE  CREDIT-DATA-POS  TO  EDITED-CREDIT.
     MOVE  DEBIT-DATA-POS    TO  EDITED-DEBIT.
     DISPLAY   'POS', CREDIT-DATA-POS, 'CONVERTED TO', EDITED-CREDIT.
     DISPLAY   'POS', DEBIT-DATA-POS, 'CONVERTED TO', EDITED-DEBIT.
SHOW-NEGATIVE.
     MOVE  CREDIT-DATA-NEG  TO  EDITED-CREDIT.
     MOVE  DEBIT-DATA-NAG    TO  EDITED-DEBIT.
     DISPLAY   'NEG', CREDIT-DATA-NEG, 'CONVERTED TO', EDITED-CREDIT.
     DISPLAY   'NEG', DEBIT-DATA-NEG, 'CONVERTED TO', EDITED-DEBIT.
      STOP  RUN.
```

以上代码执行后，程序的输出结果如下。

```
POS 1024 CONVERTED TO $1024
POS 3270 CONVERTED TO $3270_ _
NEG 1024 CONVERTED TO $1024CR
NEG 3270 CONVERTED TO $3270DB
```

此处的下划线“_”表示一个空格。通过以上代码可以归纳出使用 CR 和 DB 定义数据时输出格式的特点。

- 当原始数据为正数时，CR 和 DB 被空格取代。
- 当原始数据为负数时，CR 和 DB 方显示出来。

2．通过“+”和“-”定义算术符号格式

通过前面的学习，可以发现程序中并不能任意输出负数。虽然，通过 Signed Numbers 符号类型保存和操作负数。但是，Signed Numbers 符号类型在定义时的“S”符号只是用于保存正负号，并没有实际占用存储空间。因此，即使某一 Signed Numbers 数据内容为负数，但将其直接输出时仍看不到前面的负号。

对此，可以通过 Numeric Edited Fields 输出格式数据类型来实现输出带有符号的正负数。具体做法是使用“+”和“−”定义算术符号格式的 Numeric Edited Fields 输出格式数据。其中“+”和“−”在定义中既可以出现在数据的前面，也可以出现在数据的末尾。关于通过“+”和“−”定义算术符号格式的应用示例如下。

```
……
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    DATA-POS                PIC   S99.
 01    DATA-NEG                PIC   S99.
 01    EDITED-PLUS-HEAD        PIC   +99.
 01    EDITED-PLUS-TAIL        PIC   99+.
 01    EDITED-MINUS-HEAD       PIC   -99.
 01    EDITED-MINUS-TAIL       PIC    99-.
*
 PROCEDURE   DIVISION.
     MOVE  50  TO  DATA-POS.
     MOVE  -50  TO  DATA-NEG.
SHOW-POS.
     MOVE  DATA-POS TO EDITED-PLUS-HEAD.
     MOVE  DATA-POS TO EDITED-PLUS-TAIL.
     MOVE DATA-POS TO EDITED-MINUS-HEAD.
     MOVE DATA-POS TO EDITED-MINUS-TAIL.
     DISPLAY   'POS', DATA-POS, 'CONVERTED TO', EDITED-PLUS-HEAD, 'UNDER PLUS HEAD'.
     DISPLAY   'POS', DATA-POS, 'CONVERTED TO', EDITED-PLUS-TAIL, 'UNDER PLUS TAIL'.
     DISPLAY   'POS', DATA-POS, 'CONVERTED TO ', EDITED-MINUS-HEAD, 'UNDER MINUS HEAD'.
     DISPLAY   'POS', DATA-POS, 'CONVERTED TO ', EDITED-MINUS-TAIL, 'UNDER MINUS TAIL'.
 SHOW-NEG.
     MOVE  DATA-NEG TO EDITED-PLUS-HEAD.
     MOVE  DATA-NEG TO EDITED-PLUS-TAIL.
     MOVE DATA-NEG TO EDITED-MINUS-HEAD.
     MOVE DATA-NEG TO EDITED-MINUS-TAIL.
     DISPLAY   'NEG', DATA-NEG, 'CONVERTED TO', EDITED-PLUS-HEAD,'UNDER PLUS HEAD'.
     DISPLAY   'NEG', DATA-NEG, 'CONVERTED TO', EDITED-PLUS-TAIL,'UNDER PLUS TAIL'.
     DISPLAY   'NEG', DATA-NEG, 'CONVERTED TO', EDITED-MINUS-HEAD,'UNDER MINUS HEAD'.
     DISPLAY   'NEG', DATA-NEG, 'CONVERTED TO', EDITED-MINUS-TAIL,'UNDER MINUS TAIL'.
       STOP  RUN.
```

该段代码执行后，输出结果如下。

```
POS 50  CONVERTED TO  +50  UNDER PLUS HEAD
POS 50  CONVERTED TO  50+  UNDER PLUS TAIL
POS 50  CONVERTED TO  _50  UNDER MINUS HEAD
POS 50  CONVERTED TO  50   UNDER MINUS TAIL
NEG 50  CONVERTED TO  -50  UNDER PLUS HEAD
NEG 50  CONVERTED TO  50-  UNDER PLUS TAIL
NEG 50  CONVERTED TO  -50  UNDER MINUS HEAD
NEG 50  CONVERTED TO  50-  UNDER MINUS TAIL
```

同样，以上结果信息中的下划线“_”代表一个空格。由此可见，当使用“+”定义时，不论原始数据为正数还是负数，都将显示其符号。而若使用“–”定义时，只在原始数据为负数时显示符号。因此，可将对于 Signed Numbers 型数据正负号的输出格式简单归纳为以下两条。

- 当原始数据为正数时，使用“+”定义将输出正号，使用“–”定义输出空格。
- 当原始数据为负数时，使用“+”定义将输出负号，使用“–”定义同样输出负号。

此外，“+”和“–”同之前定义货币格式数据时使用的“$”一样，也可重复连续出现。例如，对于以下定义的这组变量。

```
 05   TEST-DATA      PIC S999  VALUE 390.
 05   EDITED-PLUS    PIC  +++9.
 05   EDITED-MINUS   PIC  ---9.
```

此时，将 TEST-DATA 的内容 MOVE 到其下面的两个变量 EDITED-PLUS 和 EDITED-MINUS 之中。该两变量的内容将分别如下所示。

- EDITED-PLUS : +390
- EDITED-MINUS : 390

4.6.3 算术数格式

这里所说的算术数包括两种类型的数，一种为小数，另一种为位数较多的数。格式输出中的算术数格式主要用于输出小数以及通过逗号分隔后的数位较多的数。

通过前面的学习可以知道，小数点在通常数据定义中是用符号“V”表示的。这里的“V”同 Signed Numbers 符号类型中的“S”是类似的，即都不占实际存储单元。若要将虚拟的小数点进行实际的输出，就需要用到格式输出中的算术数格式了。

对于位数较多的数，通常使用逗号对其进行分隔以方便观察，每 3 位为一个分隔单位。若要输出分隔号，即“,”号，同样要使用格式输出中的算术数格式。关于以上两种类型算术数综合应用的代码如下。

```
......
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    DECIMAL-NUM1         PIC   99V99.
01    LONG-NUM1            PIC   99999.
01    MIX-NUM1             PIC   99999V99.
01    EDITED-DECIMAL1      PIC   99.99.
01    EDITED-LONG1         PIC   99,999.
01    EDITED-MIX1          PIC  99,999.99.
*
PROCEDURE   DIVISION.
     MOVE   12.34  TO  DECIMAL-NUM1.
     MOVE   12345  TO  LONG-NUM1.
     MOVE   12345.67  TO  MIX-NUM1.
     MOVE   DECIMAL-NUM1 TO EDITED-DECIMAL1.
     MOVE   LONG-NUM1 TO EDITED-LONG1.
     MOVE   MIX-NUM1  TO EDITED-MIX1.
     DISPLAY  DECIMAL-NUM1 , 'CONVERTED TO', EDITED-DECIMAL1.
     DISPLAY  LONG-NUM1 , 'CONVERTED TO', EDITED-LONG1.
     DISPLAY  MIX-NUM1 , 'CONVERTED TO', EDITED-MIX1.
      STOP  RUN.
```

该段代码执行后，将输出以下信息。

```
1234 CONVERTED TO 12.34
12345 CONVERTED TO 12,345
1234567 CONVERTED TO 12,345.67
```

可以看出，无论是小数点“.”还是分隔号“,”，其具体位置都是直接在定义中给出的。此外还需注意的是，小数点总是被输出的，而分隔号只在超过 3 位数字时才输出。例如，下面的代码就说明了此类情况。

```
......
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    DECIMAL-DATA2        PIC   99V99.
```

```
 01    LONG-NUM2           PIC   99999.
 01    MIX-NUM2            PIC   99999V99.
 01    EDITED-DECIMAL2     PIC    $$.99.
 01    EDITED-LONG2        PIC  $$$,$$$.
 01    EDITED-MIX2         PIC $$$,$$.99.
*
 PROCEDURE   DIVISION.
      MOVE   0  TO  DECIMAL-NUM2.
      MOVE   10000  TO  LONG-NUM2.
      MOVE   100.00  TO  MIX-NUM2.
      MOVE   DECIMAL-NUM2  TO  EDITED-DECIMAL2.
      MOVE   LONG-NUM2  TO  EDITED-LONG2.
      MOVE   MIX-NUM2    TO  EDITED-MIX2.
      DISPLAY  DECIMAL-NUM2 , 'CONVERTED TO' , EDITED-DECIMAL2.
      DISPLAY  LONG-NUM2 , 'CONVERTED TO' , EDITED-LONG2.
      DISPLAY  MIX-NUM , 'CONVERTED TO' , EDITED-MIX2.
       STOP  RUN.
```

该段代码执行后，将输出以下信息。

```
0  CONVERTED TO  $.00                    //此处输出小数点
0  CONVERTED TO  $10,000                 //此处输出分隔号
0  CONVERTED TO  $100.00                 //此处输出小数点,但不输出分隔号
```

需要注意的是，该段代码使用了货币格式和算术数格式的综合应用。关于各种 Numeric Edited Fields 格式输出类型的综合应用将在本节最后进行详细讲解。

4.6.4 日期格式

日期格式用于对年月日表示的数据进行格式输出。日期格式通常使用反斜杠“/”将年、月、日 3 个数据进行分隔。日期格式的使用方式如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    DATE                PIC   9(8).
 01    EDITED-DATE         PIC   9999/99/99.
*
 PROCEDURE   DIVISION.
     MOVE  20080101 TO DATE.
     MOVE  DATE TO EDITED-DATE.
     DISPLAY 'TODAY IS : 'EDITED-DATE.
       STOP  RUN.
```

该段代码执行后，将输出以下信息。

```
TODAY IS 2008/01/01
```

定义日期格式时使用的“/”号和前面所学的“$”号、CR 和 DB，“+”和“−”都是不同的。“/”并不能被数值数据所填充，也不能根据原始数据的正负性选择输出还是不输出。“/”相当于一个插入符号，直接插入到原始数据中进行输出。

4.6.5 其他格式

有些 Numeric Edited Fields 格式输出类型不好进行严格的划分。比如输出格式中使用星号，或者使用空格对前缀 0 进行压缩等。在此将这些类型统归结为其他格式类型。以下将

根据其他格式类型定义中使用的不同符号分别进行讲解。

1．通过“*”和“Z”定义的其他格式

使用“*”定义其他格式的输出格式时，“*”通常是连续出现的。以下代码说明了使用“*”进行定义时输出格式的转换方式。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    CODE1               PIC   9999V99.
 01    EDITED-CODE1          PIC   ****.99.
*
 PROCEDURE   DIVISION.
       MOVE  0.05 TO CODE1.
       MOVE  CODE1 TO EDITED-CODE1.
       DISPLAY  EDITED-CODE1.
       MOVE  5000   TO CODE1.
       MOVE  CODE1 TO EDITED-CODE1.
       DISPLAY  EDITED-CODE1.
        STOP  RUN.
```

以上代码执行后，输出结果如下。

```
****.05
5000.00
```

由此可见，“*”在此起到的作用实际上是抑制数据前面无效的 0，并将其转换为“*”。当“*”对应的位置为有效数字或者小数点等情况时，“*”将被该字符填充。

使用“Z”进行定义同使用“*”类似。二者所不同的只是“Z”是将数据前面无效的 0 用空格进行的替换。关于“Z”对输出格式的具体转换方式可通过以下代码说明。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    CODE2               PIC   9999V99.
 01    EDITED-CODE2          PIC   ZZZZ.99.
*
 PROCEDURE   DIVISION.
       MOVE  0.05 TO CODE2.
       MOVE  CODE2 TO EDITED-CODE2.
       DISPLAY  EDITED-CODE2.
       MOVE  5000 TO CODE2.
       MOVE  CODE2 TO EDITED-CODE2.
       DISPLAY  EDITED-CODE2.
        STOP  RUN.
```

以上代码执行后，输出结果如下。其中下划线“_”表示空格。

```
_ _ _ _.05
5000.00
```

2．通过“0”和“B”定义的其他格式

“0”和“B”的使用方式同定义日期格式中使用的反斜杠“/”是类似的。即直接在原始数据相应位置进行插入输出，而不涉及到字符转换的问题。其中“0”表示插入字符 0；“B”

代表“Blank”，插入空格。使用“0”和“B”定义其他格式对于输出格式的转换方式如下。

```
......
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    DATE                  PIC   9(8).
01    EDITED-DATE1          PIC   9999099099.
01    EDITED-DATE2          PIC   9999B99B99.
*
PROCEDURE   DIVISION.
      MOVE  20081111 TO DATE.
      MOVE  DATE TO EDITED-DATE1.
      MOVE  DATE TO EDITED-DATE2.
      DISPLAY  EDITED-CODE1.
      DISPLAY  EDITED-CODE1.
       STOP  RUN.
```

该段代码执行后，输出结果如下。其中下划线“_”表示空格。

```
2008011011
2008_11_11
```

4.6.6 各种格式的综合应用

以上所讲的各种格式是可以并行存在的。例如一种输出格式可同时为货币格式和算术符号格式等。实际上，通常各种格式都是混合在一起使用的。以上只是为了讲解清晰，根据所使用的不同定义符号进行了一个大致的分类。

在对各种格式进行综合应用前，首先对已学过的各种格式作一个简单的回顾。前面学过的各种格式所依赖的特殊定义符号及其作用，如表 4.2 所示。

表 4.2　　Numeric Edited Fields 格式输出类型回顾

子类型名称	使用的特殊定义符号	定义符号的作用
货币格式	$	在数据前方插入货币符号$。当$连续出现时，保留且仅保留一个$符号
算术符号格式	CR	在数据末尾出现。当数据为正数时插入两个空格，为负数时插入 CR
	DB	在数据末尾出现。当数据为正数时插入两个空格，为负数时插入 DB
	+	可在数据的两头分别出现。当数据为正数时插入+，为负数时插入
	-	可在数据的两头分别出现。当数据为正数时插入空格，为负数时插入
算术数格式	.	插入小数点“.”
	,	插入数据位数分割号“,”。并且仅当数据位数超过 3 时有效
日期格式	/	插入年，月，日分割号“/”
其他格式	*	抑制数据前面的无效 0，将其转换为“*”
	Z	抑制数据前面的无效 0，将其转换为空格
	0	插入 0
	B	插入空格

进行了简单的回顾之后，可以试着对以上各种格式综合应用了。由于以上各种格式共同组合成了 Numeric Edited Fields 格式输出类型。因此，这里实际上就是对整个 Numeric Edited

Fields 格式输出类型的应用。该应用可以通过下面这一段完整的程序体现出来。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID       EDITED-SUMMARY.
 AUTHER           XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    SOURCE-INPUT-DATA.                /*定义公司名称变量*/
       05   INPUT-1    PIC  9(5)V99    VALUE  123.45.
       05   INPUT-2    PIC  9(7)       VALUE  8004.
       05   INPUT-3    PIC  S9(4)V99   VALUE  -9801.23.
       05   INPUT-4    PIC  S9(4)V99   VALUE  45.67.
       05   INPUT-5    PIC  S9(4)V99   VALUE  890.12.
       05   INPUT-6    PIC  9(6)       VALUE  31508.
       05   INPUT-7    PIC  S9(3)V99   VALUE  -3.45.
       05   INPUT-8    PIC  S9(3)V99   VALUE  678.00.
       05   INPUT-9    PIC  9(6)V99    VALUE  901.23.
       05   INPUT-10   PIC  X(6)       VALUE  'ABCDEF'.
 02     EDITED-OUTPUT-DATA.
       05   OUTPUT-1  PIC  $$$,$$9.99.
       05   OUTPUT-2  PIC  $ZZ,ZZZ,99.
       05   OUTPUT-3  PIC  ++,++9.99.
       05   OUTPUT-4  PIC  ++,++9.99.
       05   OUTPUT-5  PIC  --,--9.99.
       05   OUTPUT-6  PIC  99/99/99.
       05   OUTPUT-7  PIC  $$$9.99CR.
       05   OUTPUT-8  PIC  $$$9.99DB.
       05   OUTPUT-9  PIC  $***,***.99.
       05   OUTPUT-10 PIC   XXBBXX/XX.
*
 PROCEDURE   DIVISION.
      MOVE SOURCE-INPUT-DATA TO EDITED-OUTPUT-DATA.
      STOP  RUN.
```

该段代码执行后，程序中所定义的所有数据及其显示内容，如表 4.3 所示。

表 4.3　　各项数据及其显示内容

数　据　名	数 据 内 容	数　据　名	数 据 内 容
INPUT-1	0012345	OUTPUT-1	$123.45
INPUT-2	0008004	OUTPUT-2	$_8,004.00
INPUT-3	980123	OUTPUT-3	-9,801.23
INPUT-4	004567	OUTPUT-4	+45.67
INPUT-5	089012	OUTPUT-5	890.12
INPUT-6	031508	OUTPUT-6	03/15/08
INPUT-7	00345	OUTPUT-7	$3.45CR
INPUT-8	67800	OUTPUT-8	$678.00_ _
INPUT-9	00090123	OUTPUT-9	$****901.23
INPUT-10	ABCDEF	OUTPUT-10	AB_ _CD/EF

以上代码实现了对 Numeric Edited Fields 格式输出类型的综合应用。通过该段代码，应

该能够掌握 COBOL 程序中常用数据输出格式的定义及其转换方式。

最后需要补充的一点是，Numeric Edited Fields 格式输出类型只能用于定义输出特定的格式。该类型数据不能用于算术运算。例如，下面这段代码是错误的。

```
......
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    TEST-DATA                    PIC   999V99.
01    EDITED-TEST-DATA1            PIC    $$$$.99.
01    EDITED-TEST-DATA2            PIC    ++++.99.
*
PROCEDURE   DIVISION.
    MOVE  123.45  TO  TEST-DATA.
    MOVE  TEST-DATA TO EDITED-TEST-DATA1.
    MOVE  TEST-DATA TO EDITED-TEST-DATA2.
    ADD EDITED-TEST-DATA1 TO EDITED-TEST-DATA2.   ←此处错误，不能将这两个数据用于算术运算
    COMPUTE  EDITED-TEST-DATA1 = ( 100 + 200 ) /2.     ←此处错误，不能将该数据用于算术运算
     STOP  RUN.
```

4.7 本章回顾

本章主要讲解了 COBOL 编程中所涉及到的各种基本数据类型。其中重点在于 Numeric Edited Fields 格式输出类型。

在讲解基本数据类型之前，本章首先对 COBOL 中的基本数据类别进行了相应的介绍。了解基本数据类别对于学习具体的数据类型是有必要的。COBOL 中的基本数据类别大致上可以分为变量、常量、直接数和结构体。分类的标准是根据数据不同的表达方式及具体应用而划定的。

本章讲解的数据类型分别为字符类型、整型数类型、浮点数类型、Signed Numbers 符号类型、Numeric Edited Fields 格式输出类型。其中，字符类型包含了由任意字符组成的数据，通过符号 X 进行定义。字符类型数据通常用于比较和输出，不能用于算术运算。

整型数类型通过符号 9 进行定义，表示具体的数值。浮点数类型在整型数类型的基础上通过符号 V 定义虚拟的小数点。Signed Numbers 符号类型在整型数和浮点数的基础上通过符号 S 定义虚拟的正负号。以上 3 种类型都是可以用于算术运算的。

Numeric Edited Fields 格式输出类型主要用于定义各种不同形式的输出格式。用于该类型的定义符号较多，在该节中的最后一个小节里进行了归纳和总结。

通过本章的学习，要求对 COBOL 中数据的不同类别有一个清晰的概念，对 COBOL 中所使用的各种数据类型有一个全面的了解，能够正确定义各种不同的数据类型，同时重点掌握各种数据输出格式的定义和转换方式。

第 5 章 字符串及其操作

本章主要对 COBOL 程序中涉及到的字符串进行讲解。首先将讲解字符串的基本概念。之后，重点讲解和字符串有关的基本操作，其中包括字符串的合并、拆分、替换以及转换。最后，讲解子字符串的概念，如何得到字符串大小的最值，以及如何计算字符串的长度。

5.1 字符串的基本概念

COBOL 中的字符串实际上就是由一组连续字符所形成的数据。关于这一点，在前面学习的基本数据类型中的字符类型数据时曾提到过。字符串的定义方式如下。

```
 01  STR-ONE    PIC  XXXXX.
 01  STR-TWO    PIC  X(10).
```

以上两种定义方式都是正确的。第一种方式是通过 5 个连续出现的定义符号 X 进行定义的。该语句定义了一个拥有 5 个字符长度的字符串。第二种定义方式是通过在定义符号 X 后面加上一个括号，括号中所填数值为该字符串的长度。这条语句定义了一个拥有 10 个字符长度的字符串。

在完成字符串的定义后，通常还应该对所定义的字符串赋上一个初值。既可以通过直接数对字符串赋值，也可以通过 VALUE 语句对字符串赋值。并且，当某一字符串中已含有数据时，也可以通过 MOVE 语句对另一字符串赋值。以下代码表明了这 3 种赋值方式。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    STR-1          PIC   X(5)  VALUE 'ABCED'.
 01    STR-2          PIC   X(5).
 01    STR-3          PIC   X(5).
*
 PROCEDURE   DIVISION.
      MOVE  'ABCDE'  TO  STR-2.
      MOVE  STR-1  TO   STR-3.
```

```
        DISPLAY  'STR-1:', STR-1.
        DISPLAY  'STR-2:' , STR-2.
        DISPLAY  'STR-3:' , STR-3.
        DISPLAY  'COMPLETED !'.
         STOP  RUN.
```

该段代码执行后，将会有以下输出结果。

```
STR-1: ABCDE
STR-2: ABCDE
STR-3: ABCDE
COMPLETED !
```

对于以上代码，一共定义了 3 个字符串数据类型，分别为 STR-1、STR-2 以及 STR-3。其中字符串 STR-1 是通过 VALUE 语句在定义的同时对其赋的初值；字符串 STR-2 是通过将直接数“ABCDE”对其赋的值；字符串 STR-3 是通过 MOVE 语句将已存有数据的另一字符串 STR-1 中的内容复制到其下而完成的赋值。还需注意的是，程序中出现的两个直接数“ABCDE”和“COMPLETED !”实际上也属于字符串，只是该字符串属于直接数类别，没有进行定义，因而没有对应的名称。

最后还需注意的一点是，COBOL 中的字符串和通常使用 C 语言编写的程序中的字符串是有区别的。在 C 语言中，字符串在机内存储时系统要在末尾添加一个空字符'\0'作为结束标志。而 COBOL 中是不需对字符串添加任何结束标志的。COBOL 中字符串的存储长度即显现出来的实际长度。

5.2 使用 STRING 语句合并字符串

前面一节简单介绍了 COBOL 中字符串的基本概念。下面，将重点介绍对字符串的实际操作。字符串 3 最基本的操作分别为合并、拆分以及转换。本节主要介绍如何对字符串进行合并操作。

5.2.1 STRING 语句的基本用法

当需要将两个以上的字符串合并为一个字符串时，通常使用 STRING 语句完成。下面这段代码表明了使用 STRING 语句合并字符串的基本操作。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    FIRST-NAME          PIC   X(10).
 01    LAST-NAME           PIC   X(10).
 01    FULL-NAME           PIC    X(20).
*
 PROCEDURE   DIVISION.
      MOVE  'ADAM'  TO  FIRST-NAME.
      MOVE  'SMITH'  TO  LAST-NAME.
      STRING
        FIRST-NAME   DELIMITED  BY  SPACE
        ' '  DELIMITED BY SIZE
        LAST-NAME   DELIMITED   BY  SPACE
```

```
          INTO  FULL-NAME.
       DISPLAY  'FIRST NAME:',FIRST-NAME.
       DISPLAY  'LAST NAME:',LAST-NAME.
       DISPLAY  'FULL NAME:',FULL-NAME.
         STOP  RUN.
```

该段代码执行后，将有以下输出结果。

```
FIRST NAME: ADAM
LAST NAME: SMITH
FULL NAME: ADAM SMITH
```

以上程序实现的功能实际上是将分开存储放置的某人姓与名字符串合并为一个完整的姓名字符串。FIRST-NAME 字符串用于存放该人的姓，LAST-NAME 字符串用于存放该人的名。FULL-NAME 字符串用于存放该人完整的姓名，由以上两字符串合并而成。该程序实现字符串合并操作的重点语句如下。

```
STRING
  FIRST-NAME   DELIMITED  BY  SPACE
  ' '  DELIMITED BY SIZE
  LAST-NAME   DELIMITED   BY  SPACE
  INTO  FULL-NAME.
```

以上即为 STRING 语句的基本格式。其中每项用于合并的字符串后面都要加上 DELIMITED BY 子句。DELIMITED BY 子句后面有两个选项，分别为 SPACE 和 SIZE。这两个选项的作用分别如下。

- SPACE: 找到前面用于合并的字符串中第一次出现空格的地方。将该空格以前的部分进行合并，空格以后的内容包括该空格在内不参与合并操作。
- SIZE: 将前面对应的用于合并的字符串中的全部内容进行合并。

对应前面的代码，可以看到参与合并的字符串共有 3 项。这 3 项字符串依次为 FIRST-NAME，由一个空格所形成的直接数字符串，以及 LAST-NAME。其中 FIRST-NAME 和 LAST-NAME 中的实际内容分别如下。

```
FIRST-NAME: ADAM_ _ _ _ _ _
LAST-NAME: SMITH_ _ _ _ _
```

此处使用下划线表示空格。由于这两项字符串数据的定义语句如下，因此存储长度都为 10 个字符的长度。

```
01    FIRST-NAME          PIC   X(10).
01    LAST-NAME           PIC   X(10).
```

由于对这两个字符串的赋值语句如下，赋值的长度小于定义的长度。因此，需要在这两个字符串数据后加上相应的空格以填充其定义的长度。

```
    MOVE  'ADAM'  TO  FIRST-NAME.
      MOVE  'SMITH'  TO  LAST-NAME.
```

在进行合并时，我们并不希望将这些多余的空格也合并到新的字符串中。因此，这里使用了 DELIMITED BY 子句中的 SPACE 选项将多余的空格进行了截取。截取之后，两者实际参与合并的内容如下。

```
FIRST-NAME: ADAM
LAST-NAME: SMITH
```

如果仅仅是对以上这两项内容进行合并时，最终输出结果会将二者连接在一起，不利于区分。即若使用下面这条 STRING 语句进行合并时，输出结果形式将不太理想。

```
STRING
  FIRST-NAME   DELIMITED  BY  SPACE
  LAST-NAME   DELIMITED   BY  SPACE
  INTO  FULL-NAME.
```

将原程序中的 STRING 语句替换为这条 STRING 语句后，输出结果如下所示。

```
FIRST NAME: ADAM
LAST NAME: SMITH
FULL NAME: ADAMSMITH
```

可以看到，这时输出的完整姓名 FULL NAME，将组成姓和名的字母连接到了一起，无法区分开来。因此，这时还需要在合并后的字符串中插入一个空格，以区分哪几个字母是姓，哪几个字母是名。插入空格时就不能再用 DELIMITED BY SAPCE 子句了，否则将插入不进任何东西。插入空格以及插入带空格的字符串时，必须使用 DELIMITED BY SIZE 子句，将整个进行插入。原程序中插入空格的语句如下。

```
' '  DELIMITED BY SIZE
```

这样，原程序依次将 FIRST NAME、一个空格（作为姓和名的分隔符）、LAST NAME 进行了合并。并且，对于 FIRST NAME 和 LAST NAME 这两个字符串，仅将其有效部分进行合并，最终合并后的 3 个字符串便组成了另一个新的字符串。该字符串显示完整的姓名信息，并且在姓和名之间留有一个空格以进行区分。

5.2.2 STRING 语句的综合应用

上一小节简单介绍了 STRING 语句的基本用法。根据其基本用法，该语句在实际开发中常常用于将文件中各条分开放置的记录组成一条完整的输出信息。以下结合一个具体实例，说明 STRING 语句在实际开发中的综合应用。假设某一公司员工各项财务收入信息文件 INCOME-SYSIN 包含有如下信息。

- 员工的工号
- 员工姓名
- 员工住址
- 工资
- 奖金
- 出差报销费用
- 本次工资及奖金发放的具体时间
- 本次出差报销的具体时间

该文件包含有多条记录，分别对应每一位员工的相关信息。为方便说明问题，这里假设其中第一条记录的具体信息如下。

```
10742
WANG_WEI _ _ _ _ _ _ _
WU_HAN_SHI_JIANG_HAN_QU_YOU_YI_LU
3500
```

```
1000
802.75
08/01/31
08/01/25
```

现要求只输出员工及其出差报销信息，并且出差报销费用只取其整数部分。此外，每条输出记录要有相应编号，编号从 00001 开始顺次往下排。最后每条输出信息要求统一从第 5 列开始输出。则实现该功能完整的程序代码如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID       INCOME-PROG.
 AUTHER           XXX.
*
 ENVIRONMENT   DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT  INCOME-FILE
      ASSIGN  TO  INCOME-SYSIN.
*
 DATA  DIVISION.
 FILE SECTION.
 FD  INCOME-FILE
     RECORDING MODE IS F.
 01  RECORD.
     05   EMPLOEE-INFO.
          10   EMP-NUMBER    PIC 9(5).
          10   EMP-NAME      PIC X(15).
          10   EMP-ADDRESS   PIC X(35).
     05   INCOME-INFO.
          10   SALARY        PIC 9(4).
          10   BONUS         PIC 9(4).
          10   REIMBURSE     PIC 999.99.
          10   SAL-DATE      PIC X(8).
          10   REIM-DATE     PIC X(8).
 WORKING-STORAGE  SECTION.
 77   PRT-LINE    PIC  X(100).
 77   LINE-POS    PIC  S9(4).
 77   LINE-NO     PIC  9(5).
 77   DEC-POINT   PIC X VALUE '.'.
 77   EOF-FLAG    PIC X VALUE 'N'.
*
 PROCEDURE   DIVISION.
    OPEN  INPUT  INCOME-FILE.
      READ   INCOME-FILE
          AT  END  MOVE 'Y' TO EOF-FLAG
      END-READ.
    PERFORM 100-PROCESS-RECORDS
     VARYING LINE-NO FROM 1 BY 1
     UNTIL  EOF-FLAG = 'Y'.
    CLOSE   TEST-FILE.
     STOP  RUN.
 100-PROCESS-RECORDS.
     MOVE 5 TO LINE-POS.
     STRING
      LINE-NO SAPCE EMPLOEE-INFO  SPACE
```

```
           DELIMITED BY SIZE
       REIMBURSE
           DELIMITED BY DEC-POINT
       SPACE REIMDATE
           DELIMITED BY SIZE
       INTO RPT-LINE
       WITH  POINTER LINE-POS.
       DISPLAY  PRT-LINE.
       READ   INCOME-FILE
         AT  END  MOVE 'Y' TO EOF-FLAG
       END-READ.
```

该段程序执行后，将有如下输出信息（下划线表示空格）。

```
第 5 列
  ↓
  00001 WANG WEI               802 08/01/25
  00002_……
  ……
```

其中省略号省略了其他记录对应的输出信息。该段程序实际上完成了以下 3 个任务。

- 选择相关的字符串信息进行合并。
- 指定每条信息在输出字符串中所在的具体位置。
- 对报销费用字符串进行截取。

对应于 STRING 的基本用法，该程序中所扩展的两条 STRING 子句如下。

- DELIMITED BY DEC-POINT：这是对 DELIMITED BY 子句的延伸。在上一小节 STRING 的基本用法里只讲了在 BY 的后面接上 SIZE 或 SPACE 选项。而此处 BY 后面接的是一个变量名称，该变量的内容为一个小数点。因此这里是将 REIMBURSE 变量中小数点以前的部分进行截取并用于合并。实际上，BY 后可接任何一个变量，并且将该变量中保存的字符作为原字符串的截取符。截取符的用法同 SPACE 选项类似。
- WITH POINTER：该语句指定合并生成字符串的起始位置，类似于一个指针。原程序中其后为 LINE-POS 变量，且该变量保存有数值 5。因此合并后字符串的起始位置在新字符串 PRT-LINE 的第 5 位。

5.3 使用 UNSTRING 语句拆分字符串

同使用 STRING 语句合并字符串相对应，通常使用 UNSTRING 语句拆分字符串。本节先介绍 UNSTRING 语句的基本用法。在此基础上，讲解 UNSTRING 语句的综合应用。

5.3.1 UNSTRING 语句的基本用法

当需要将字符串进行拆分时，通常就要用到 UNSTRING 语句。UNSTRING 语句相当于是 STRING 语句的逆运算。

下面，仍然结合前面利用 STRING 语句合并姓名的例子介绍 UNSTRING 语句的基本用法。这次假设此人的姓名“Adam Smith”已保存在一个变量之中。此程序所需完成的功能是将该条完整的姓名拆分为姓和名两个字符串，并分别存放到不同的变量之中。程序代码如下。

```
......
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    FULL-NAME      PIC   X(20)
                           VALUE  'ADAM SMITH'.
01    LAST-NAME      PIC   X(10).
01    FIRST-NAME     PIC   X(10).
*
PROCEDURE   DIVISION.
    UNSTRING  FULL-NAME
              DELIMITED BY ' '
              INTO   FIRST-NAME
                     LAST-NAME.
    DISPLAY 'FULL NAME:', FULL-NAME.
    DISPLAY 'FIRST NAME:', FIRST-NAME.
    DISPLAY 'LAST NAME:', LAST-NAME.
     STOP RUN.
```

以上代码运行后，有如下输出信息。

```
FULL NAME: ADAM SMITH
FIRST NAME: ADAM
LAST NAME: SMITH
```

由此可见，存放有完整姓名信息的FULL NAME字符串变量被拆分为了两个新的字符串。其中一个字符串名称为FIRST NAME，保存FULL NAME中的前一部分内容“ADAM”。另一个字符串名称为LAST NAME，保存FULL NAME中的后一部分内容“SMITH”。FULL NAME中的这两部分是通过空格分隔开来的。因此，在拆分该字符串时使用了以下这条子句用以划分拆分的内容。

```
 DELIMITED BY ' '
```

同样，此处的空格字符“' '”也可以被其他字符所替代。例如，下面这段程序就使用日期分隔符‘/’替代了上面代码中的空格“' '”。该程序实现的功能是将一个完整的日期分成年、月、日3个部分，依次存放在不同变量中。代码如下。

```
......
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    DATE           PIC   X(10)
01    YEAR           PIC   X(4).
01    MONTH          PIC   X(2).
01    DAY            PIC   X(2).
*
PROCEDURE   DIVISION.
    MOVE '2008/01/15'  TO DATE.
    UNSTRING  DATE
              DELIMITED BY '/ '
              INTO   YEAR
                     MONTH
                     DAY.
    DISPLAY 'DATE:', DATE.
    DISPLAY 'YEAR:', YEAR.
    DISPLAY 'MONTH:', MONTH.
```

```
        DISPLAY 'DAY:', DAY.
         TOP RUN.
```

以上代码运行后，有如下输出信息。

```
DATE: 2008/01/15
YEAR: 2008
MONTH: 01
DAY: 15
```

5.3.2 UNSTRING 语句的综合应用

同 STRING 语句一样，UNSTRING 语句在实际应用中还有一些其他需要注意的地方。下面结合一个实例进行讲解，以加深对 UNSTRING 语句的理解，同时提高实际应用能力。

假设一家公司对其运营的项目进行管理，需要将某一项目不同类别的信息分开保存到不同的变量中。其中完整的项目信息从文件中的记录获得。保存项目不同类别信息的变量在数据部的工作存储节中定义。完整的代码如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID       DIVIDE-PROG.
 AUTHER           XXX.
*
 ENVIRONMENT   DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
 SELECT  PROJECT-FILE
          ASSIGN  TO  P-SYSIN.
*
 DATA  DIVISION.
 FILE SECTION.
 FD  PROJECT-FILE
    RECORDING MODE IS F.
 01   PRO-RCD.
     05   LINE-NO       PIC  9(5).
     05   FILLER        PIX  X.
     05   PRO-CODE      PIC  X(10).
     05   FILLER        PIC  X.
     05   PRO-NAME      PIC  X(21).
     05   FILLER             PIC  X.
     05   LEADER-NO     PIC  9(5).
     05   FILLER             PIC  X.
     05   BUDGET        PIC  9(7).99.
 WORKING-STORAGE  SECTION.
 01   DISPLAY-INFO.
     05   P-NAME             PIC  X(21).
     05   FILLER             PIC  X  VALUE  SPACE.
     05   P-UNIT             PIC  XXX.
     05   FILLER             PIC  X  VALUE  SPACE.
     05   P-BUDGET           PIC  9(7).
 01    OTHER-INFO.
     05   PJT-NO             PIC 9(6).
     05   PJT-LEADER-NO      PIC 9(5).
 77    LEN-1        PIC 99.
 77    LEN-2        PIC 99.
```

```
 77   DLMBY       PIC X.
 77   STDLM-1     PIC X.
 77   STDLM-2     PIC X.
 77   STR-POS     PIC 9.
 77   FIELD-NUM   PIC 9.
 77   EOF-FLAG    PIC X  VALUE 'N'.
*
 PROCEDURE   DIVISION.
      OPEN  INPUT   PROJECT-FILE.
      READ  PROJECT-FILE
            AT  END  MOVE 'Y' TO  EOF-FLAG
      END-READ.
      MOVE  '.'  TO  DLMBY.
      MOVE  7  TO  POS-START.
      PERFORM  100-PROCESS-RECORDS
            UNTIL   EOF-FLAG = 'Y'.
      CLOSE  PROJECT-FILE.
       STOP RUN.
 100-PROCESS-RECORDS.
      UNSTRING  PRO-RCD.
        DELIMITED  BY  ALL  SPACES OR '/ ' OR DLMBY
        INTO    PJT-NO  DELIMITER  IN  STDLM-1
                P-UNIT
                P-NAME  COUNT  IN  LEN-1
                PJT-LEADER-NO  DELIMITER IN STDLM-2  COUNT IN  LEN-2
                P-BUDGET
        WITH  POINTER  STR-POS
        TALLYING  IN  FIELD-NUM
        ON  OVERFLOW  GO  UNSTRING-FINISHED.
 UNSTRING-FINISHED.
      DISPLAY  DISPLAY-INFO.
      READ  PROJECT-FILE
         AT  END  MOVE 'Y' TO EOF-FLAG
      END-READ.
```

以上程序的运行步骤如下。

（1）打开文件 PROJECT-FILE，读入文件第 1 条记录。如果遇到文件结束（即该文件为空），将文件结束标志 EOF-FLAG 置为‘Y’。其中 F-FLAG 默认值为‘N’。对应代码如下。

```
OPEN  INPUT   PROJECT-FILE.
    READ  PROJECT-FILE
          AT  END  MOVE 'Y' TO  EOF-FLAG
    END-READ.
```

（2）将拆分文件时自定义的分隔符小数点‘.’存入变量 DLMBY 中。将拆分字符的起始位置 7 存入变量 STR-POS 中。循环执行 100-PROCESS-RECORDS 处理过程，直至读取文件结束。当读取文件结束时，关闭文件，程序结束。对应代码如下。

```
MOVE  '.'  TO  DLMBY.
MOVE  7  TO  STR-POS.
PERFORM  100-PROCESS-RECORDS
          UNTIL   EOF-FLAG = 'Y'.
CLOSE  PROJECT-FILE.
STOP RUN.
```

（3）执行 100-PROCESS-RECORDS 处理过程，对文件中整条记录的字符串进行拆分。拆分时，从该字符串的第 7 个位置开始拆分。7 由变量 STR-POS 指定。拆分完成后，将拆分的条目数量保存到变量 FIELD-NUM 中。当字符串末尾还有未被拆分数据（该程序省略掉项目运算金额的小数部分，因此都不会被完全拆分）时，转到 UNSTRING-FINISHED 处理过程。对应代码如下。

```
UNSTRING  PRO-RCD.
      DELIMITED  BY  ALL  SPACES OR '/' OR DLMBY
      INTO    PJT-NO  DELIMITER  IN  STDLM.
              P-UNIT
              P-NAME  COUNT  IN  LEN
              PJT-LEADER-NO
              P-BUDGET  DELIMITER IN STDLM-2  COUNT IN  LEN-2
      WITH  POINTER  STR-POS
      TALLYING  IN  FIELD-NUM
      ON  OVERFLOW  GO  UNSTRING-FINISHED.
```

该段代码中重点有以下几条 UNSTRING 语句的子句需要讲解。

- DELIMITER IN 子句：该子句将此处拆分的分隔符保存到变量中。原程序第一条 DELIMITER 子句将此处字符串 ‘307428/ITD’ 中的分隔符 ‘/’ 保存到了变量 STDLM 中。第二条项目预算金额中的小数点 ‘.’ 保存到了变量 STDLM-2 中。
- COUNT IN 子句：该子句记录该拆分部分的字符串长度。原程序中第一条 COUNT IN 子句记录拆分后的项目名称 P-NAME 的长度，并保存到变量 LEN-1 中。第二条将输出项目运算金额的长度保存到变量 LEN-2 中。
- WITH POINTER 子句：该子句后面所跟的变量相当于拆分字符串的指针。原程序对应变量为 STR-POS。该变量中的数字 7 表明从第 7 个字符处开始拆分。拆分完成后，该变量应指向项目预算金额小数点后的位置为 45。
- TALLYING IN 子句：保存拆分的条目数量，即拆分后生成的字符串数量。原程序将字符串拆分成了 5 个短的字符串，因此将 5 存入变量 FIELD-NUM 中。
- ON OVERFLOW 子句：表明原字符未拆分完全时需采取的行动。原程序由于项目预算金额最后两位小数未参与拆分，因此执行该子句。执行后跳转到 UNSTRING-FINISHED 处理过程处。

（4）执行 UNSTRING-FINISHED 处理过程。显示输出信息，并且继续读文件的下一条记录。如果文件结束，将结束标志 EOF-FLAG 置为 ‘Y’。对应语句如下。

```
UNSTRING-FINISHED.
    DISPLAY  DISPLAY-INFO.
    READ  PROJECT-FILE
       AT  END  MOVE 'Y' TO EOF-FLAG
    END-READ.
```

现假设该文件中第一条记录信息如下。

```
00001_307428/ITD_ONLINE-PAYMENT-SYSTEM_B7831_1085080.50
```

该记录中各自对应的不同类别信息及在字符串中的位置如下。

```
LINE-NO    : 00001                          //输入行号,在字符串中的位置为 1~5
```

```
FILLER        : _                          //间隔符,在字符串中的位置为 6
PRO-CODE      : 307428/ITD                 //项目代码,在字符串中的位置为 7
FILLER        :                            //间隔符,在字符串中的位置为 8
PRO-NAME      : ONLINE-PAYMENT-SYSTEM      //项目名称,在字符串中的位置为 9~29
FILLER        :_                           //分隔符,在字符串中的位置为 30
LEADER-NO     : B7831                      //项目组长编号,在字符串中的位置为 31~35
FILLER        :_                           //分隔符,在字符串中的位置为 36
BUDGET        : 1085080.50                 //项目预算金额,在字符串中的位置为 37~46
```

对应这条记录，输出结果如下。

```
ONLINE-PAYMENT-SYSTEM_ ITD_1085080
```

该输出结果所对应的变量内容如下。

```
P-NAME       :  ONLINE-PAYMENT-SYSTEM
FILLER       :
P-UNIT       :  ITD
FILLER       :  _
P-BUDGET     :  1085080
```

程序中定义的其他相关变量中的值分别如下。

```
PJT-NO        :  307428
PJT-LEADER-NO :  B7831
LEN-1         :  21
LEN-2         :  7
STDLM-1       :   /
STDLM-2       :   .
STR-POS       :  45
FIELD-NUM     :   5
```

5.4 利用 INSPECT 语句替换字符串

INSPECT 语句主要用于对字符串中的指定部分进行替换。这里所说的字符串替换和字符串转换是不同的。字符串替换是指将原字符串中的部分字符替换成其他字符。字符串转换则是保持原字符串中的字符不变，只对其格式进行转换。本节主要讲解字符串替换。

5.4.1 对全体字符进行替换

使用 INSPECT 语句可以对字符串中出现的全部相同字符进行整体替换。该项功能是通过在 INSPECT 语句中加上“ALL”选项实现的。下面这段代码将原字符串中所有的字符“A”替换为“B”，并且将所有的 0 替换为中划线。代码如下。

```
......
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    TEST-STRING      PIC   X(10).
*
PROCEDURE   DIVISION.
MOVE 'ADCD0A0A8B' TO TEST-STRING.
DISPLAY  'BEFORE INSPECT:', TEST-STRING.
INSPECT  SOURCE-STRING
        REPLACING  ALL  'A'  BY 'B'.
```

```
INSPECT  SOURCE-STRING
        REPLACING  ALL  0  BY'-'.
DISPLAY  'AFTER INSPECT:', TEST-STRING.
 STOP RUN.
```

该段代码执行后，将有如下输出信息。

```
BEFORE INSPECT: ADCD0A0A8B
AFTER INSPECT:  BCDC-B-B8B
```

由此可见，原字符串 TEST-STRING 中的所有字符 B 被替换为了字符 A。并且，原字符串中的所有数字 0 也被替换成了中划线‘-’。这种替换方式是对字符串中满足条件的全体字符进行的替换。并且替换后生成的新字符串覆盖了以前的字符串。

5.4.2 对前缀字符进行替换

这种替换方式是对字符串前面连续出现的一段字符进行替换，即对前缀字符进行的替换。该项功能是通过在 INSPECT 语句中加上“LEADING”选项实现的。

下面代码对一段表示数字的字符串进行了替换。替换方式为将原字符串中的前缀 0 替换成前缀星号‘*’，以进行适当的输出。代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    TEST-STRING      PIC   X(10).
*
 PROCEDURE   DIVISION.
 MOVE '0000567.90'  TO  TEST-STRING.
 DISPLAY  'BEFORE INSPECT:', TEST-STRING.
    INSPECT   TEST-STRING
       REPLACING  LEADING  0  BY  '*'.
    DISPLAY  'AFTER INSPECT:', TEST-STRING.
STOP RUN.
```

该段代码执行后，将有如下输出信息。

```
BEFORE INSPECT: 0000567.90
AFTER INSPECT:  ****567.90
```

通过以上代码可以发现，使用 LEADING 选项只是对前缀字符进行了替换。这里所说的前缀字符，是从字符串开头处连续出现的一串字符。因此，即使原字符串中在最后还有一个字符 0，也并不对其进行替换。

此外，INSPECT 语句还可统计并保存前缀字符出现的次数。该项功能是通过结合 TALLYING… FOR LEADING …实现的。其中前面一个省略号表示用以保存前缀字符出现次数的变量名。后一个省略号表示该前缀字符。以下代码统计并输出了字符串中前缀字符出现的次数。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    TEST-STRING      PIC   X(10).
 01    COUNT            PIC   9   VALUE ZERO.
```

```
*
 PROCEDURE   DIVISION.
     MOVE '0000567.90'  TO  TEST-STRING.
     DISPLAY  'SOURCE STRING:', TEST-STRING.
     INSPECT   TEST-STRING
       TALLYING  COUNT  FOR  LEADING '0'.
     DISPLAY  'HOW MANY LEADING 0:', COUNT.
      STOP RUN.
```

该段代码执行后，将有如下输出信息。

```
SOURCE STRING: 0000567.90
HOW MANY LEADING 0: 4
```

5.4.3 对首字符进行替换

此处只对字符串中的第一个字符进行替换。该项功能是通过在 INSPECT 语句中加上“FIRST”选项实现的。下面代码结合上一小节中的例子，将前缀为星号的字符串首字符替换为了美元符号‘$’。该段代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    TEST-STRING      PIC   X(10).
*
 PROCEDURE   DIVISION.
     MOVE '0000567.90'  TO  TEST-STRING.
     DISPLAY  'SOURCE  DATA:', TEST-STRING.
     INSPECT   TEST-STRING
       REPLACING  LEADING  0  BY '*'.
     DISPLAY  'CONVERTED  DATA-1:', TEST-STRING.
     INSPECT   TEST-STRING
       REPLACING  FIRST '*' BY '$'.
     DISPLAY  'CONVERTED  DATA-2:', TEST-STRING.
      STOP RUN.
```

该段代码执行后，将有如下输出信息。

```
SOURCE DATA: 0000567.90
CONVERTED  DATA-1: ****567.90
CONVERTED  DATA-2: $***567.90
```

以上程序首先将前缀字符进行转换，之后在此基础上再对首字符进行转换。通过两次转换，最终将原始数据 0000567.90 转换成了标准输出格式$***567.90。

5.4.4 字符串替换的综合应用

除以上讲解的 3 种字符串常用替换方式外，在实际应用中还有一些其他小技巧。下面这个例子说明了字符串替换的综合应用。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    TEST-STRING      PIC   X(12).
*
```

```
PROCEDURE  DIVISION.
    MOVE 'B2.C-AB3H/DE'  TO  INPUT-STRING.
    DISPLAY  'SOURCE  DATA:', TEST-STRING.
    INSPECT   TEST-STRING
       REPLACING  FIRST  'B'  BY  'C'  AFTER  INITIAL  'A'.
    DISPLAY  'CONVERTED-1: ', TEST-STRING.
    INSPECT   TEST-STRING
       REPLACING  CHARACTERS  BY  '0'  BEFORE  INITIAL  '.'.
    DISPLAY  'CONVERTED-2 :', TEST-STRING.
    INSPECT   TEST-STRING
        CONVERTING
          'ABCDEFGHI'   TO
          '123456789'
           AFTER  INITIAL '-'
           BEFORE INITIAL '/'.
    DISPLAY  'CONVERTED-3 :', TEST-STRING.
     STOP RUN.
```

该段代码执行后，将有如下输出信息。

```
SOURCE DATA: B2.C-AB3H/DE
CONVERTED-1: B2.C-AC3H/DE
CONVERTED-1: 00.C-AC3H/DE
CONVERTED-1: 00.C-1338/DE
```

以上程序实际上共对原字符串进行了 3 次替换操作。这 3 次替换操作分别如下。

- 第 1 次替换操作对应的代码如下。

```
INSPECT   TEST-STRING
    REPLACING  FIRST  'B'  BY  'C'  AFTER  INITIAL  'A'.
```

此次操作将原字符串中的第一个字符‘A’后出现的首个字符‘B’替换成了字符‘C’。本次替换前和替换后的字符串分别如下。

```
B2.C-AB3H/DE          //替换前的字符串
B2.C-AC3H/DE          //替换后的字符串
```

- 第 2 次替换操作对应的代码如下。

```
INSPECT   TEST-STRING
    REPLACING  CHARACTERS  BY  '0'  BEFORE  INITIAL  '.'.
```

此次操作将原字符串中小数点前面的所有字符替换为字符‘0’。其中此处的原字符为第一次替换后所生成的新字符串。本次替换前和替换后的字符串分别如下。

```
B2.C-AC3H/DE          //替换前的字符串
00.C-AC3H/DE          //替换后的字符串
```

- 第 3 次替换操作对应的代码如下。

```
INSPECT   TEST-STRING
     CONVERTING
       'ABCDEFGHI'   TO
       '123456789'
        AFTER  INITIAL '-'
        BEFORE INITIAL '/'.
```

此次操作将原字符串里中划线和反斜杠之间的所有字母字符替换成了相应的数字字符。其

中此处的原字符为第二次替换后所生成的新字符串。本次替换前和替换后的字符串分别如下。

```
00.C-AC3H/DE          //替换前的字符串
00.C-1338/DE          //替换后的字符串
```

该段代码反映了字符串替换的综合应用。同前面讲过的 3 种常用的字符串替换方式相比，此处共有以下 3 点扩展内容。

- BEFORE INITIAL 和 AFTER INITIAL：这两个选项用以指定字符串所替换的范围。BEFORE INITIAL 后面的字符对应原字符串中用以替换部分的首字符。AFTER INITIAL 后面的字符对应原字符串中用以替换部分的末尾字符。
- CHARACTERS：泛指全体字符，而非特定的某一个字符。此种表示方式通常结合 BEFORE INITIAL 或 AFTER INITIAL 使用。
- CONVERTING：用于同时指定多个字符的替换内容。该选项相当于多个 REPLACING BY 语句的综合。

5.5 字符串转换

本节将讲解字符串的转换。字符串转换是以原字符串的字符为基础进行的格式转换，并不对原字符进行替换。本节主要介绍两种常用的转换方式。其中一种是对于由字母组成的字符串中字母大小写的转换方式，另一种是将由数字组成的字符串转换为具体数值。

5.5.1 字符串中字母大小写的转换

对于由字母组成的字符串，通常需要对该字符串里面的字母进行大小写转换。字符串中字母大小写的转换主要是通过 FUNCTIONG 语句实现的。其关键之处是在该语句后加上 LOWER-CASE 或者 UPPER-CASE 选项。下面这段代码使用 LOWER-CASE 选项，将字符串中的字母全部转换为小写格式，代码如下。

```
……
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    SOURCE-STR        PIC   X(5).
01    NEW-STR            PIC   X(5).
*
PROCEDURE   DIVISION.
     MOVE  'AB-CDE'  TO  SOURCE-STR.
     DISPLAY  SOURCE-STR.
     DISPLAY  FUNCTION  LOWER-CASE(SOURCE-STR).
     MOVE  'Ab-Cde'  TO   SOURCE-STR.
     DISPLAY  SOURCE-STR.
     DISPLAY  FUNCTION  LOWER-CASE(SOURCE-STR).
     MOVE  FUNCTION  LOWER-CASE(SOURCE-STR) TO NEW-STR.
     DISPLAY  NEW-STR.
     DISPLAY  SOURCE-STR.
      STOP RUN.
```

该段代码执行后，将有以下输出信息。

```
AB-CDE
ab-cde
```

```
Ab-Cde
ab-cde
ab-cde
Ab-Cde
```

通过该段代码，说明了如何使用 LOWER-CASE 将字符串中的字母转换为小写形式。同时对应原程序的输出结果，关于 LOWER-CASE 的使用还有以下几点需要注意。

- LOWER-CASE 只将字符串中的大写字母转换为对应的小写形式。对于字符串中原本为小写的字母字符，以及其他字符不做处理。
- LOWER-CASE 只是临时对其进行转换，转换后的结果只在该条语句有效。即 LOWER-CASE 并不将转换结果保存到原字符串中。若要保存转换后的字符串，通常使用 MOVE 语句将其保存到一个新的字符串变量中。

与 LOWER-CASE 对应，UPPER-CASE 是将字符串中的字母转换为对应的大写形式。下面这段代码综合应用了 LOWER-CASE 和 UPPER-CASE 进行字符串转换。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    SOURCE-STR        PIC   X(10).
 01    NEW-STR            PIC   X(5).
*
 PROCEDURE   DIVISION.
      MOVE  'String  Convert!'  TO  SOURCE-STR.
      DISPLAY  SOURCE-STR.
      DISPLAY  FUNCTION  LOWER-CASE (SOURCE-STR).
      DISPLAY  SOURCE-STR.
      DISPLAY  FUNCTION  UPPER-CASE (SOURCE-STR).
      MOVE  FUNCTION  LOWER-CASE (SOURCE-STR) TO NEW-STR.
      DISPLAY  NEW-STR.
      DISPLAY  SOURCE-STR.
      STOP RUN.
```

该段代码执行后，将有以下输出信息。

```
String Convert!
string convert!
String Convert!
STRING CONVERT!
string convert!
String Convert!
```

5.5.2 将字符串转换为具体数值

将字符串转换为具体数值将是字符串转换中用处很大的一种转换方式。将字符串转换为具体数值后，该数值同其他普通字符一样，便可以参与算术运算了。同时，当需要从键盘输入相应数值时，也更加方便一些。

将字符串转换为具体数值同前一小节将的字符串大小写转换类似，都是使用 FUNCTIONG 语句。不同的是，此处是通过在 FUNCTION 语句后面加上 NUMVAL 或者 NUMVAL-C 实现的。下面先介绍 NUMVAL 的用法，代码如下。

```
 ......
 DATA  DIVISION.
```

```
 WORKING-STORAGE  SECTION.
 01    SOURCE-STR-1         PIC   X(10).
 01    SOURCE-STR-2         PIC   X(10).
 01    RESULT               PIC   S99V99.
 01    RESULT-DPL           PIC   $$$.99.
*
 PROCEDURE   DIVISION.
     MOVE  '-10.50'  TO  SOURCE-STR-1.
     MOVE  '12.80'  TO  SOURCE-STR-2.
     COMPUTE  RESULT =
                FUNCTION NUMVAL (SOURCE-STR-1)
              +  FUNCTION NUMVAL (SOURCE-STR-2).
     MOVE  RESULT TO RESULT-DPL.
     DISPLAY 'RESULT:',RESULT-DPL.
      STOP RUN.
```

该段代码执行后，将有如下输出信息。

```
RESULT:  $2.30
```

对于该段程序，有以下几点需要注意。

- SOURCE-STR-1 变量中所对应的内容‘-10.50’是一个字符串，而非具体数值。该字符串若不转换为具体数值，是不能参与算术运算的。
- SOURCE-STR-2 变量中所对应的内容‘12.80’同样也是一个字符串，而非具体数值。
- FUNCTION NUMVAL 语句将以上两个变量中的字符串转换为具体数值。并将这两个具体数值相加后将结果存入 RESULT 变量中。RESULT 变量被定义为一个 Signed Numbers 符号类型数据，保存的是具体数值。
- 程序最后将 RESULT 变量中的具体数值 MOVE 到 RESULT-DPL 变量中以用于输出。RESULT-DPL 变量被定义为一个 Numeric Edited Fields 格式输出类型。因此，这里相当于又将具体数值转换回成了字符串。

对应于 NUMVAL 选项，下面代码使用 NUMVAL-C 进行字符串转换。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    SOURCE-STR-1         PIC   X(10).
 01    SOURCE-STR-2         PIC   X(10).
 01    RESULT               PIC   S9(5)V99.
 01    RESULT-DPL           PIC   $$$,$$$.99.
*
 PROCEDURE   DIVISION.
     MOVE  '12,000.50'  TO  SOURCE-STR-1.
     MOVE  '$100.50DB'  TO  SOURCE-STR-2.
     COMPUTE  RESULT =
                FUNCTION NUMVAL-C (SOURCE-STR-1)
              +  FUNCTION NUMVAL-C (SOURCE-STR-2).
     MOVE  RESULT TO RESULT-DPL.
     DISPLAY 'RESULT:',RESULT-DPL.
      STOP RUN.
```

该段代码执行后，将有如下输出信息。

```
RESULT:  $11,900.00
```

实际上，NUMVAL-C 相当于是 NUMVAL 的扩展。当原始字符串出现以下情况时，必须使用 NUMVAL-C 选项。

- 原始字符串中有货币流通符号$。
- 原始字符串中有分隔符逗号。

此外需要注意的是，无论 NUMVAL-C 还是 NUMVAL-C 都允许原字符串中出现正负号。并且，转换后的实际数值遵循原字符串中所指定的正负性。另外，根据数值数据的规定，转换后数值数据的长度不能超过 18 个数字。

以上说明了将字符串转换为实际数值后可参与算术运算的功能。下面再简单介绍一下当需要从键盘输入具体数值时，如何使输入方式更加方便。假设输入数值所被保存的变量为 INPUT-DATA，该变量定义如下。

```
01  INPUT-DATA      PIC  S999V99.
```

令该变量中的数值通过键盘输入的语句如下。

```
ACCEPT  INPUT-DATA  FROM CONSOLE
```

此时，用户需要在键盘上输入的数据必须严格按照 INPUT-DATA 定义的格式来进行。例如，当用户需要输入 2.50 和–100.00 这两个数值时，输入方式必须分别如下。

```
 +002.50
 -100.00
```

当使用将字符串转换为具体数值的方式时，代码可以做如下更改。

```
01  INPUT-STR        PIC  X(10).
01  INPUT-DATA       PIC  S999V99.
......
   ACCEPT  INPUT-STR  FROM  CONSOLE.
   COMPUTE  INPUT-DATA = FUNCTION NUMVAL (INPUT-STR).
```

对应如上代码，当用户需要输入数值 2.50 和–100.00 时，只用按照如下方式输入即可。

```
  2.50
 -100
```

这里实际上是先定义一个 INPUT-STR 变量，并将输入的数据作为字符串保存到该变量中。然后，通过 NUMVAL 将其转换为具体数值。并且，该数值的组织方式与 INPUT-DATA 定义方式一致。最后，将其存入到 INPUT-DATA 变量中。

总之，将字符串转换为具体数值的最大功能是可以将字符串中的数字参与算术运算。其次，用户不必管理字符串转换成具体数值后的格式。系统将自动根据保存数值的变量所定义的格式进行相应的转换，方便用户操作。

5.6 子字符串的概念及应用

子字符串相当于原字符串的一个子集，也就是原字符串中的一部分内容。定义子字符串时，通常需要指定两个参数。其中第一个参数为该子字符串首字符在原字符串中的位置，第二个参数为该子字符串的长度。

以下代码从组成日期的一段字符串中提取年、月、日 3 个子字符串并进行输出。该例子

在前面的使用 UNSTRING 语句拆分字符串中曾讲到过。但不同的是此处是通过提取子字符串方式实现的，而不是字符串拆分。代码如下。

```
......
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    DATE           PIC   X(10).
01    YEAR           PIC   X(4).
01    MONTH          PIC   XX.
01    DAY            PIC   XX.
*
PROCEDURE   DIVISION.
    MOVE  '2008/08/15'  TO  DATE.
    MOVE   DATE  (1:4) TO YEAR.
    MOVE   DATE  (6:2) TO MONTH.
    MOVE   DATE  (9:2) TO DAY.
    DISPLAY  'DATE:', DATE.
    DISPLAY  'YEAR:', YEAR.
    DISPLAY  'MONTH:', MONTH.
    DISPLAY  'DAY:', DAY.
     STOP RUN.
```

以上代码的运行结果如下。

```
DATE: 2008/08/15
YEAR:2008
MONTH:08
DAY:15
```

本例中原字符串为‘2008/08/15’，并且该字符串保存在变量 DATE 中。基于此原字符串，程序中共用到了 3 个相应的子字符串，分别介绍如下。

- DATE （1:4）子字符串：该子字符串括号中的第一个参数 1 指定了这条子字符串在原字符串中的起始位置。第二个参数 4 指定了该子字符串的长度。根据以上定义，这条子字符串为：‘2008’。该子字符串最后通过 MOVE 语句保存到了变量 YEAR 中。
- DATE （6:2）子字符串：该子字符串括号中的第一个参数 6 指定了这条子字符串在原字符串中的起始位置。第二个参数 2 指定了该子字符串的长度。根据以上定义，这条子字符串为：‘08’。该子字符串最后通过 MOVE 语句保存到了变量 MONTH 中。
- DATE （9:2）子字符串：该子字符串括号中的第一个参数 9 指定了这条子字符串在原字符串中的起始位置。第二个参数 2 指定了该子字符串的长度。根据以上定义，这条子字符串为：‘15’。该子字符串最后通过 MOVE 语句保存到了变量 DAY 中。

此外，对于子字符串表示方式里括号中的第二个参数，也可以省略不写。由于第二个参数表示的是子字符串的长度，因此没有指定子字符串的具体长度。当不对子字符串的长度进行指定时，该子字符串将为从起始位置直到原字符串末尾的一串字符。以下代码说明了这一点。

```
......
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    SOURCE-STR-1     PIC   X(10).
01    SOURCE-STR-2     PIC   X(14).
01    SUB-STR-1        PIC   X(4).
```

```
 01    SUB-STR-2        PIC   X(8).
*
 PROCEDURE   DIVISION.
     MOVE  '123456ABCD'  TO  SOURCE-STR-1.
     MOVE  '123456ABCDEFGH'  TO  SOURCE-STR-2.
     MOVE   SOURCE-STR-1  (7:) TO SUB-STR-1.
     MOVE   SOURCE-STR-2  (7:) TO SUB-STR-2.
     DISPLAY  'SOURCE STR-1:', SOURCE-STR-1.
     DISPLAY  'SUB STR-1:', SUB-STR-1.
     DISPLAY  'SOURCE STR-2:', SOURCE-STR-2.
     DISPLAY  'SUB STR-2:', SUB-STR-2.
      STOP RUN.
```

以上代码的运行结果如下。

```
SOURCE STR-1: 123456ABCD
SUN STR-1: ABCD
SOURCE STR-2: 123456ABCDEFGH
SUN STR-2: ABCDEFGH
```

本例中子字符串是通过 SOURCE-STR (7:)指定的。此处括号中默认了第二个参数，即没有指定该子字符串的具体长度。因此，子字符串是从原字符串中的第 7 个字符开始，直到原字符串末尾的一串字符。对于不同的原字符串，通过此种方式表达形成的子字符串将有着不同的长度。

最后，以上两个用于指定子字符串起始位置和长度的参数也可以通过变量表示。下面这段代码结合前面学习过的 INSPECT 语句统计出前缀字符的长度，并保存到变量 I 中。然后再通过变量 I 分别指定子字符串的两个参数形成两个新的字符串。最终通过字符串合并将前缀字符变成了后缀字符。代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01   SOURCE-STR    PIC  X(10).
 01   SUB-STR-1     PIC  X(10).
 01   SUB-STR-2     PIC  X(10).
 01   I              PIC  9.
 01   NEW-STR       PIC  X(10).
*
 PROCEDURE   DIVISION.
     MOVE  '0000AB1234'  TO  SOURCE-STR.
     MOVE  ZERO  TO I.
     INSPECT  SOURCE-STR
         TALLYING  I  FOR  LEADING '0'.
     MOVE  SOURCE ( I+1:) TO SUB-STR-1.
     MOVE  SOURCE ( 1:I)  TO SUB-STR-2.
     STRING
         SUB-STR-1  DELIMITED  BY SPACE
         SUB-STR-2  DELIMITED  BY SPACE
         INTO  NEW-STR.
      DISPLAY 'SOURCE:', SOURCE-STR.
      DISPLAY 'NEW:    ',NEW-STR.
      STOP RUN.
```

程序执行后，将有以下输出结果。

```
SOURCE: 0000AB1234
NEW:    AB12340000
```

5.7 通过 MAX 和 MIN 得到最大和最小字符串

字符的大小是根据其在机器内部的编号顺序决定的。在通常的微机中，采用的是 ASCII 编码。而在大型机中，采用的是 EBCDIC 编码。通常的字符在这两种编码中的顺序大小是相似的。例如，字母字符通常都是从 A 到 Z 排序的，字母字符由小到大的顺序也是从 A 到 Z。当将字符串进行比较时，是从第一个字符开始比较起，依次比较每个字符的大小。当存在一个字符串中的字符大于另一个字符中对应的字符时，则该字符串大于另一个字符串。

比较字符串的大小通常使用 FUNCTION 语句，并结合 MAX 和 MIN 选项实现的。下面代码通过 MAX 和 MIN 对两个字符串进行了比较。

```
……
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    STR-1    PIC   X(5).
01    STR-2    PIC   X(5).
01    MAX-STR   PIC   X(5).
01    MIN-STR   PIC   X(5).
*
PROCEDURE   DIVISION.
      MOVE  'JASON'  TO  STR-1.
      MOVE  'SIMON'  TO  STR-2.
      MOVE   FUNCTION  MAX ( STR-1  STR-2)  TO MAX-STR.
      MOVE   FUNCTION  MIN ( STR-1  STR-2)   TO MIN-STR.
      DISPLAY  'MAX STR:', MAX-STR.
      DISPLAY  'MIN STR:', MIN-STR.
       STOP RUN.
```

该段代码的运行结果如下。

```
MAX STR: JASON
MIN STR: SIMON
```

以上是对两个字符串进行的比较，其中一个字符串为‘JASON’，另一个为‘SIMON’。当将这两个字符串进行比较时，遵循的顺序是从左至右依次对组成字符串的单个字符进行比较。对于这两个字符串而言，即最先将‘JASON’中的‘J’与‘SIMON’中的‘S’进行比较。由于按照字母表顺序，‘J’是排在‘S’前面的，因此‘J’大于‘S’。到此，比较就结束了。最后得出字符串‘JASON’大于字符串‘SIMON’的。

同样，利用 MAX 和 MIN 还可以对多个字符串进行比较，并求得其中的最大和最小的字符串。下面代码对 3 个字符串进行了比较，并求取其中的最大值和最小值。

```
……
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    STR-1    PIC   X(10).
01    STR-2    PIC   X(10).
01    STR-3    PIC   X(10).
01    MAX-STR  PIC   X(10).
```

```
 01    MIN-STR   PIC    X(10).
*
 PROCEDURE   DIVISION.
       MOVE  'ROBIN'    TO  STR-1.
       MOVE  'ROBERT'  TO  STR-2.
       MOVE  'ROB'  TO  STR-3.
       MOVE   FUNCTION  MAX ( STR-1  STR-2   STR-3)  TO MAX-STR.
       MOVE   FUNCTION  MIN ( STR-1   STR-2   STR-3)  TO MIN-STR.
       DISPLAY  'MAX STR:', MAX-STR.
       DISPLAY  'MIN STR:', MIN-STR.
        STOP RUN.
```

该段代码的运行结果如下所示。

```
MAX STR: ROBIN
MIN STR: ROB
```

需要注意的是，此程序中当对字符串‘ROBIN’和‘ROBERT’比较时，一共比较了 4 个字符。由于这两个字符串的前 3 个字符都为‘ROB’，因此需要继续往后面比较才能得出大小。当比较到两个字符串的第 4 个字符时，由于‘I’是大于‘E’的，因此‘ROBIN’较大。此外，根据字符编码顺序，字母字符是大于空格字符的。因此，‘ROBERT’是大于‘ROB’的。最后，得出以上 3 个字符串中‘ROBIN’最大，‘ROB’最小。

需要补充的一点是，通过 MAX 和 MIN 还可统计出数值数据中的最值。以下代码对 3 个数值数据进行比较，并输出其中的最大值和最小值。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    NUM-1     PIC    9(3).
 01    NUM-2     PIC    9(3).
 01    NUM-3     PIC    9(3).
 01    MAX-NUM       PIC    9(3).
 01    MIN-NUM       PIC    9(3).
*
 PROCEDURE   DIVISION.
       MOVE  123    TO  NUM-1.
       MOVE    4  TO    NUM-2.
       MOVE  567  TO  NUM-3.
       MOVE   FUNCTION  MAX ( NUM-1  NUM-2   NUM-3)  TO MAX-NUM.
       MOVE   FUNCTION  MIN ( NUM-1   NUM-2   NUM-3)  TO MIN-NUM.
       DISPLAY  'MAX NUM:', MAX-NUM.
       DISPLAY  'MIN NUM:', MIN-NUM.
        STOP RUN.
```

该段代码的运行结果如下。

```
MAX NUM: 567
MIN NUM: 4
```

5.8 求取字符串的长度

通常，可以通过 FUNCTION LENGTH 或者 LENGTH OF 得到字符串的长度。这两种方式得到的结果是相同的。但 FUNCTION LENGTH 语句仅当相应的算术表达式允许的情况下方能使用。而 LENGTH OF 则要灵活得多。LENGTH OF 不仅可以用在一些内部功能的整型

参数变量中，也可以用在 CALL 语句的参数之中。

此外，以上两种方式不仅可以得到字符串的长度，也可以得到数值数据的位数长度。实际上，这里是将数值数据作为一个字符串来看待的，只计算其占据的存储空间大小。

下面这段代码分别使用了 FUNCTION LENGTH 语句和 LENGTH OF 语句得到字符串及数值数据的长度。可以看出，两种方式得到的最终结果是一致的。代码如下。

```
……
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    SOURCE-STR    PIC   X(10).
01    SOURCE-NUM    PIC   9(10).
01    LEN-1         PIC   99.
01    LEN-2         PIC   99.
*
PROCEDURE   DIVISION.
      MOVE   'ABC'    TO  SOURCE-STR.
      MOVE    123     TO  SOURCE-NUM.
      COMPUTE  LEN-1  =  FUNCTION  LENGTH (SOURCE-STR).
      COMPUTE  LEN-2  =  FUNCTION  LENGTH (SOURCE-NUM).
      DISPLAY  'STR BY FUNCTION LENGTH:', LEN-1.
      DISPLAY  'STR BY LENGTH OF:', LENGTH OF SOURCE-STR.
      DISPLAY  'NUM BY FUNCTION LENGTH:', LEN-2.
      DISPLAY  'NUM BY LENGTH OF:', LENGTH OF SOURCE-NUM.
       STOP RUN.
```

该段代码的运行结果如下。

```
STR BY FUNCTION LENGTH: 10
STR BY LENGTH OF:10
NUM BY FUNCTION LENGTH:3
NUM BY LENGTH OF:3
```

需要注意的是，此处是通过 MOVE 语句将字符串‘ABC’复制到变量 SOURCE-STR 的。SOURCE-STR 变量最初被定义为 10 个字符长度。因此，复制后该变量中前 3 个字符为‘ABC’，后面 7 个字符为空格。该变量仍然为 10 个字符长度。

对于变量 SOURCE-NUM 而言，将数据 123 复制到其中后，该数值数据只占用最后 3 个存储单元。因此，SOURCE-NUM 的长度为 3。

下面例子和子字符串一节中的最后一个例子类似，也是将前缀字符转换为后缀字符。不过此处通过计算原字符串的长度，指定了 SUB-STR-1 子字符串的第二个参数数值。代码如下。

```
……
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01   SOURCE-STR    PIC  X(10).
01   SUB-STR-1     PIC  X(10).
01   SUB-STR-2     PIC  X(10).
01   I              PIC  9.
01   NEW-STR        PIC  X(10).
*
PROCEDURE   DIVISION.
     MOVE  '0000AB1234'  TO  SOURCE-STR.
     MOVE  ZERO  TO I.
```

```
        INSPECT  SOURCE-STR
           TALLYING  I  FOR  LEADING '0'.
        MOVE  SOURCE ( I+1: LENGTH OF SOURCE-STR - I )
           TO SUB-STR-1.           /*此处通过原字符串长度指定了子字符串第二个参数数值*/
        MOVE  SOURCE ( 1:I)  TO SUB-STR-2.
        STRING
           SUB-STR-1  DELIMITED  BY SPACE
           SUB-STR-2  DELIMITED  BY SPACE
           INTO  NEW-STR.
        DISPLAY 'SOURCE:', SOURCE-STR.
        DISPLAY 'NEW:    ',NEW-STR.
          STOP RUN.
```

程序执行后，将有以下输出结果。

```
SOURCE: 0000AB1234
NEW:    AB12340000
```

字符串长度的计算有时还会涉及到和求取字符串最值大小的综合应用。下面这段代码直接求得了两个字符串中较大字符串的长度。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    STR-1    PIC   X(10).
 01    STR-2    PIC   X(5).
 01    LEN      PIC   99.
*
 PROCEDURE   DIVISION.
       MOVE  'A'    TO  STR-1.
       MOVE  'B'    TO  STR-2.
       COMPUTE  LEN = FUNCTION  LENGTH ( FUNCTION MAX (STR-1 STR-2) ).
       DISPLAY  'MAX ITEM'S LENGTH:', LEN.
        STOP RUN.
```

该段代码的运行结果如下所示。

```
MAX ITEM'S LENGTH: 5
```

5.9 本章回顾

本章主要讲解了 COBOL 中字符串的概念及用法。其中，对字符串最基本的操作分别为合并字符串、拆分字符串和替换字符串。此外在实际开发中还经常会用到转换字符串，子字符串的应用，最大、最小字符串的提取以及计算字符串的长度。

学习本章需要熟练掌握如何使用 STRING 语句对字符串进行合并，如何使用 UNSTRING 语句对字符串进行拆分，如何使用 INSPECT 语句对字符串进行替换。以上 3 条针对字符串操作的基本语句是本章的重点，一定要牢固掌握。同时，子字符串的概念及应用也是十分重要的，需要熟练掌握。

此外，还应了解如何转换字符串中字母的大小写，如何将字符串转换为实际数值，如何从多个字符串中提取其中最大和最小的字符串，如何得到字符串的长度。这些在实际开发中也是经常会用到的，对该知识点的灵活应用将显著提高开发效率。

第 6 章 基本运算

本章主要介绍 COBOL 开发中常用的一些基本运算。此处所说的基本运算包括 3 大类型的运算：算术运算、关系运算和逻辑运算。本章将依次讲解这 3 种运算。学习本章，关键要能够将所讲解的各种运算在实际中进行灵活的运用。

6.1 算术运算

当使用 COBOL 处理各种数值型数据，如计算理财收益、统计财务收支等，必然需要用到算术运算。此处所说的算术运算包括了加减乘除四则运算、乘方运算以及基于以上运算的复合算术运算。对算术运算的结果进行处理将会贯穿于各种算术运算的始终。

6.1.1 四舍五入运算 ROUNDED

在讲解具体的算术运算之前，有必要先了解一下对运算结果进行处理的方式。其中使用 ROUNDED 选项对运算结果进行四舍五入是最常用的处理方式。这种处理方式在后面所讲解的各种具体的算术运算的实例中都是经常会用到的。

通常，使用 ROUNDED 选项对运算结果进行四舍五入是对运算结果中第 1 位小数进行的舍入操作。也就是说，使用 ROUNDED 选项得到的运算结果通常应该是整数。下面例子说明了 ROUNDED 选项的实际用法。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    NUM1            PIC   99V9.
 01    NUM2            PIC   99V9.
 01    REAL-RESULT     PIC   99V9.
 01    REAL-DISPLAY    PIC   99.9.
 01    RESULT          PIC    99.
*
 PROCEDURE   DIVISION.
```

```
       MOVE   12.3   TO  NUM1.
       MOVE   45.6   TO  NUM2.
       ADD  NUM1  TO  NUM2
            GIVING  REAL-RESULT.
       MOVE  REAL-RESULT  TO  REAL-DISPLAY.
       DISPLAY  'REAL  RESULT: ', REAL-DISPLAY.
       ADD   NUM1  TO  NUM2
           GINVING  RESULT.
       DISPLAY  'RESULT WITHOUT ROUNDED:', RESULT.
       ADD   NUM1  TO  NUM2
           GINVING  RESULT  ROUNDED.       /*此处使用 ROUNDED 选项对运算结果进行舍入*/
       DISPLAY  'RESULT WITH ROUNDED:', RESULT.
        STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
REAL  RESULT: 57.9
RESULT WITHOUT ROUNDED:57
RESULT WITH ROUNDED: 58
```

由此可见，使用 ROUNDED 选项后，对运算结果第 1 位小数进行了四舍五入操作。以上运算中，两数据相加的原始结果为 57.9。当在运算中没有加上 ROUNDED 选项时，结果数据将被截断。根据保存结果的变量 RESULT 的定义，截取原始数据小数点前两位数字，因此结果显示为 57。当在运算中加上 ROUNDED 选项后，将对 57.9 进行四舍五入处理。由于 9 是大于 5 的，因此最终结果为 58。

6.1.2 运算结果溢出报错 ON SIZE ERROR

当结果数据长度超过保存结果数据的变量所定义的长度时，称为运算结果溢出。当运算结果溢出时，可以使用 ON SIZE ERROR 选项进行报错，并执行相应处理过程。

下面这段代码进行相加运算后，得到的结果数据长度超出了保存结果的变量所定义的长度，因此运算结果溢出了。当运算结果溢出时，在屏幕上会输出相应提示信息。代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    NUM1            PIC   99.
 01    NUM2            PIC   99.
 01    REAL-RESULT     PIC   999.
 01    RESULT          PIC    99.
 01    OVER-FLOW-FLAG  PIC   X  VALUE 'N'.
*
 PROCEDURE   DIVISION.
       MOVE   56   TO  NUM1.
       MOVE   78   TO  NUM2.
       ADD  NUM1  TO  NUM2
            GIVING  REAL-RESULT.
       DISPLAY  'REAL  RESULT: ', REAL-RESULT.
       ADD   NUM1  TO  NUM2
            GINVING  RESULT
              ON SIZE ERROR        /*此处使用 ON SIZE ERROR 选项，当结果溢出时进行以下处理*/
              DISPLAY  'OVER FLOW!'
              PERFORM  OVER-FLOW-ROUTINE.
```

```
        IF  OVER-FLOW-FLAG = 'N'
             DISPLAY 'CORRECT RESULT:', RESULT
        END-IF.
        STOP RUN.
 OVER-FLOW-ROUTINE.
        MOVE 'Y' TO OVER-FLOW-FLAG.
        DISPLAY 'TRUNCATED  RESULT:',RESULT.
```

以上代码运行后，将有如下输出结果。

```
REAL  RESULT: 134
OVER FLOW!
TRUNCATED  RESULT: 34
```

本例中结果数据为 134，占 3 个字符长度，而保存结果的变量 RESULT 被定义为只占两个字符长度。因此，结果数据将溢出，RESULT 只能保存实际结果的后两为数字，即 34。此外，当不使用 ON SIZE ERROR 选项时，结果数据仍然将溢出。只是此时不会报错，也不能进行相应处理。

总之，ON SIZE ERROR 选项以及前面所讲的 ROUNDED 选项，都是用于对运算结果进行处理。对运算结果的处理将涵盖在下面所要讲解的各种算术运算的实际应用之中。以上讲解的这两种对运算结果的处理方式区别如下。

- ROUNDED 选项对结果进行四舍五入处理，针对的是小数点右边的小数部分。
- ON SIZE ERROR 选项判断结果是否溢出，并在溢出时进行相应处理。针对的是小数点左边的整数部分。

6.1.3 算术加运算 ADD

前面两小节讲解了对运算结果进行处理的两种常用方式。从本小节开始，将依此讲解 COBOL 中常用的算术运算。其中包括常用的加减乘除四则运算以及乘方运算。最后，将讲解基于以上各种运算的复合算术运算。

本小节主要讲解算术加运算。在 COBOL 中进行算术加运算，通常有两种方式。一种是直接通过运算符号“+”进行算术加运算。另一种则是通过 ADD 语句完成。关于第一种方式，将在复合算术运算一节中介绍。本节只介绍如何通过 ADD 语句实现算术加运算。

下面这段代码将 NUM1 中的数值加到 NUM2 中，并将结果保存在 NUM2 中。运算结果将覆盖 NUM2 中原来的数值，而 NUM1 中数值不变。代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    NUM1            PIC    9.
 01    NUM2            PIC    9.
*
 PROCEDURE   DIVISION.
       MOVE    2    TO  NUM1.
       MOVE    3    TO  NUM2.
       DISPLAY 'NUM1 BEFORE ADD: ',NUM1.
       DISPLAY 'NUM2 BEFORE ADD: ',NUM2.
       ADD  NUM1  TO  NUM2
            ON SEZE ERROR
```

```
            DISPLAY  'OVER FLOW!'
        END-ADD.
        DISPLAY 'NUM1 AFTER ADD:',NUM1.
        DISPLAY 'NUM2 AFTER ADD:',NUM2.
         STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
NUM1 BEFORE ADD:2
NUM2 BEFORE ADD:3
NUM1 AFTER ADD:2
NUM2 AFTER ADD:5
```

上面 NUM1 中的数据为 2，NUM2 中的数据为 3。两数相加后结果为 5，并且结果 5 保存在 NUM2 中，覆盖了原来的数据 3。而 NUM1 中的数据 2 在运算后是不变的。

以上进行的是两个数的算术加运算，实际上，这种方式也可以用于多个数的算术加运算。下面代码实现了 3 个数的相加运算。

```
......
DATA  DIVISION.
WORKING-STORAGE  SECTION.
 01    NUM1            PIC    9  VALUE 2.
 01    NUM2            PIC    9  VALUE 3.
 01    NUM3            PIC    9  VALUE 4.
*
PROCEDURE   DIVISION.
DISPLAY 'NUM1: ',NUM1.
DISPLAY 'NUM2: ',NUM2.
DISPLAY 'NUM3: ',NUM3.
ADD  NUM1  NUM2  TO  NUM3.
         ON SEZE ERROR
         DISPLAY  'OVER FLOW!'
END-ADD.
DISPLAY 'TOTAL: ',NUM3.
     DISPLAY 'NUM3 AFTER ADD: ',NUM3.
       STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
NUM1: 2
NUM2: 3
NUM3: 4
TOTAL: 9
NUM3 AFTER ADD: 9
```

上面讲解的实际上是 ADD 语句的其中一种使用方式。ADD 语句还有另一种使用方式，就是配合 GIVING 子句来使用。这种使用方式中，运算结果并不会覆盖用来相加的某一数值，而是另外保存在一个新的变量中。此外通常也可省略掉 ADD 语句的结束标志 END-ADD，而直接使用句点表示该语句的结束。下面代码使用了 ADD…GIVING 语句，并省略掉了 END-ADD，代码如下。

```
......
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    NUM1            PIC    9  VALUE 2.
```

```
 01    NUM2            PIC   9  VALUE 3.
 01    NUM3            PIC   9  VALUE 4.
 01    TOTAL            PIC   9.
*
 PROCEDURE   DIVISION.
       ADD  NUM1  NUM2  TO  NUM3.
            GIVING  TOTAL .
       DISPLAY 'NUM1: ',NUM1.
       DISPLAY 'NUM2: ',NUM2.
       DISPLAY 'NUM3: ',NUM3.
       DISPLAY 'TOTAL: ',TOTAL .
       DISPLAY 'NUM3 AFTER ADD:',NUM3.
        STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
NUM1: 2
NUM2: 3
NUM3: 4
TOTAL: 9
NUM3 AFTER ADD: 4
```

6.1.4 算术减运算 SUBTRACT

当需要进行算术减运算时，通常可以直接使用运算符号“-”或者使用 SUBTRACT 语句完成。本小节只介绍 SUBTRACT 语句。

同 ADD 语句类似，SUBTRACT 语句也有两种使用方式。一种是将结果覆盖到最后一个运算数中，另一种是将结果保存到一个新的变量中。下面先介绍第一种方式。以下这段代码将 NUM3 中的数据同时减去 NUM1 和 NUM2 中的数据。该段代码使用的是 SUBTRACT 语句的第一种方式，即将结果保存到 NUM3 中，并覆盖其原始数据。代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    NUM1            PIC   9  VALUE 2.
 01    NUM2            PIC   9  VALUE 3.
 01    NUM3            PIC   9  VALUE 9.
 01    TOTAL            PIC   9.
*
 PROCEDURE   DIVISION.
       SUBTRACT  NUM1  NUM2  FROM  NUM3.
       END-SUBTRACT.
       DISPLAY 'NUM1:',NUM1.
       DISPLAY 'NUM2:',NUM2.
       DISPLAY 'NUM3:',NUM3.
       DISPLAY 'RESULT:',NUM3.
       DISPLAY 'NUM3 AFTER SUBTRACT:',NUM3.
        STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
NUM1: 2
NUM2: 3
NUM3: 9
```

```
RESULT: 4
NUM3 AFTER SUBTRACT: 4
```

下面这段代码同样进行算术减运算。被减数为 NUM3，减数为 NUM2 和 NUM1。这里使用的是 SUBTRACT 的第二种方式，将运算结果保存到新变量 RESULT 中。同时，这里使用了浮点数的运算，因此使用 ROUNDED 选项代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    NUM1            PIC   9V9   VALUE 2.1.
 01    NUM2            PIC   9V9   VALUE 3.2.
 01    NUM3            PIC   9V9   VALUE 9.8.
 01    RESULT          PIC   9.
 01    NUM3-PRT        PIC   9.9.
*
 PROCEDURE   DIVISION.
       MOVE NUM3 TO NUM3-PRT.
       DISPLAY 'NUM3 BEFORE SUBTRACT:',NUM3-PRT.
       SUBTRACT  NUM1  NUM2  FROM  NUM3
          GIVING RESULT
          ROUNDED.
       DISPLAY 'RESULT: ',RESULT.
       MOVE NUM3 TO NUM3-PRT.
       DISPLAY 'NUM3 AFTER SUBTRACT:',NUM3-PRT.
        STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
NUM3 BEFORE SUBTRACT: 9.8
RESULT: 5
NUM3 AFTER SUBTRACT: 9.8
```

6.1.5 算术乘运算 MULTIPLY

MULTIPLY 语句和运算符号“*”都可以用于实现算术乘运算，本小节主要讲解 MULTIPLY 语句。该语句的使用方式同前面讲过的 ADD 语句和 SUBTRACT 语句基本上类似。下面先介绍该语句的第一种使用方式，即将运算结果覆盖到最后一个运算数上。实现该方式的示例代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    NUM1            PIC   9.
 01    NUM2            PIC   9.
*
 PROCEDURE   DIVISION.
       MOVE   2   TO  NUM1.
       MOVE   3   TO  NUM2.
       DISPLAY 'NUM1 BEFORE MULTIPLY:',NUM1.
       DISPLAY 'NUM2 BEFORE MULTIPLY:',NUM2.
       MULTIPLY  NUM1  BY  NUM2
            ON SEZE ERROR
            DISPLAY  'OVER FLOW!'
```

```
        END-MULTIPLY.
        DISPLAY 'NUM1 AFTER MULTIPLY:',NUM1.
        DISPLAY 'NUM2 AFTER MULTIPLY:',NUM2.
        STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
NUM1 BEFORE MULTIPLY:2
NUM2 BEFORE MULTIPLY:3
NUM1 AFTER MULTIPLY:2
NUM2 AFTER MULTIPLY:6
```

此处需要特别注意的是，最终的运算结果是保存在第 2 个操作数中，而不是第 1 个。在实际开发中，初学者往往容易根据该语句的字面意思，而误认为结果是保存在第 1 个操作数中。这一点一定要注意避免。

当使用 MULTIPLY 语句的第二种方式时，运算结果保存的位置就要清楚多了。其保存位置即 GIVING 子句后指明的变量。下面代码说明了这一点。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    NUM1            PIC   9.
 01    NUM2            PIC   9.
 01    RESULT           PIC   9.
*
 PROCEDURE   DIVISION.
       MOVE   2   TO  NUM1.
       MOVE   3   TO  NUM2.
       MULTIPLY  NUM1  BY   NUM2  TO
          GIVING  RESULT.
       DISPLAY 'NUM1:',NUM1.
       DISPLAY 'NUM2:',NUM2.
       DISPLAY 'RESULT:',RESULT.
       DISPLAY 'NUM2 AFTER MULTIPLY:',NUM2.
        STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
NUM1: 2
NUM2: 3
REUSLT: 6
NUM2 AFTER MULTIPLY: 3
```

6.1.6 算术除运算 DIVIDE

本小节将讲解基本四则运算中的最后一条运算，即算术除运算。算术除运算同样可以分别由直接运算符号“ / ”或者 DIVIDE 语句完成。关于使用直接运算符号进行算术运算的方式，将统一放在下一小节复合算术运算中讲解。本小节只讲解如何通过 DIVIDE 语句进行算术除运算。

同前面所讲解的 3 种用于基本算术运算的语句一样，DIVIDE 语句也有两种使用方式。其中分别为包含 GIVING 子句的使用方式和不包含 GIVING 子句的使用方式。下面这段代码用到了 DIVIDE 语句的第一种使用方式，代码如下。

```
……
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    NUM1            PIC    99.
01    NUM2            PIC    9.
*
PROCEDURE   DIVISION.
      MOVE    8    TO  NUM1.
      MOVE   20    TO  NUM2.
      DISPLAY 'NUM1 BEFORE DIVIDE:',NUM1.
      DISPLAY 'NUM2 BEFORE DIVIDE:',NUM2.
      DIVIDE   NUM1  INTO  NUM2
      END-DIVIDE.
      DISPLAY 'NUM1 AFTER DIVIDE:',NUM1.
      DISPLAY 'NUM2 AFTER DIVIDE:',NUM2.
      STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
NUM1 BEFORE DIVIDE: 8
NUM2 BEFORE DIVIDE: 20
NUM1 AFTER MULTIPLY:8
NUM2 AFTER MULTIPLY:2
```

通过运行结果可以看到，使用这种方式的 DIVIDE 语句得到的运算结果仍然保存在第 2 个操作数中。第 1 个操作数内容仍然是不变的。实际上所有用于四则运算的语句中，当不含 GIVING 子句时，其运算结果通常都保存在最后一个操作数中。当包含 GIVING 子句时，运算结果由 GIVING 子句后所指定的变量保存。

注意到，以上例子中用 20 除以 8，将得到一个小数，其数值为 2.5。此处仅仅保留了实际结果中的整数部分，而将小数部分直接截断了。由于除法操作常常会出现结果为小数的情况，因此实际应用中常常包含前面所讲的 ROUNDED 选项。当结合 ROUNDED 选项进行该条算术除运算时，将得到不同的结果。代码如下。

```
……
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    NUM1            PIC    99.
01    NUM2            PIC    9.
*
PROCEDURE   DIVISION.
      MOVE    8    TO  NUM1.
      MOVE   20    TO  NUM2.
      DISPLAY 'NUM1 BEFORE DIVIDE:',NUM1.
      DISPLAY 'NUM2 BEFORE DIVIDE:',NUM2.
      DIVIDE   NUM1  INTO  NUM2
         ROUNDED
      END-DIVIDE.
      DISPLAY 'NUM1 AFTER DIVIDE:',NUM1.
      DISPLAY 'NUM2 AFTER DIVIDE:',NUM2.
       STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
NUM1 BEFORE DIVIDE: 8
NUM2 BEFORE DIVIDE: 20
```

```
NUM1 AFTER MULTIPLY:8
NUM2 AFTER MULTIPLY:3
```

当要求运算结果必须为整数时，对于除不尽的情况，将使用余数来表示。但以上这种 DIVIDE 的使用方式是得不到算术除运算所产生的余数。此时可以通过 GIVING 和 REMAINDER 分别得到其商和余数。示例代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    NUM1            PIC   99.
 01    NUM2            PIC   9.
 01    QUOT            PIC   9.
 01    REMAIND          PIC   9.
*
 PROCEDURE   DIVISION.
       MOVE    3   TO  NUM1.
       MOVE   10   TO  NUM2.
       DIVIDE   NUM2  BY  NUM1  GIVING  QUOT
          REMAINDER  REMAIND
       END-DIVIDE.
       DISPLAY 'THE QUOTIENT IS:',QUOT.
       DISPLAY 'THE REMAINDER IS:',REMAIND.
        STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
THE QUOTIEN IS: 3
THE REMAINDER IS: 1
```

该段代码将 10 除以 3，得到的商保存在变量 QUOT 中，得到的余数保存在变量 REMAIND 中。此外，注意到这里使用的是 DIVIDE…BY…的表达形式。因此，此处是将该 DIVIDE 语句中的第 1 个操作数作为被除数，第 2 个作为除数。而前面例子中所使用的 DIVIDE…INTO…的表达形式则不同，该形式下是将 DIVIDE 语句中的第 1 个操作数作为除数，而第 2 个作为被除数。

最后还需注意的是，在 DIVIDE 语句中，ROUNDED 和 REMAINDER 选项是不可同时出现的。否则若通过 ROUNDED 将结果的小数部分进行舍入操作了，又如何能通过 REMAINDER 得到余数呢？这在逻辑上不通的。

6.1.7 乘方运算 COMPUTE

COMPUTE 语句的使用方式和前面所讲的 4 种语句是不同的。在 COMPUTE 语句中，主要是通过算术表达式进行算术运算的。算术表达式中除可以包含前面所讲的基本四则运算的运算符外，还可以包含乘方的运算符。本小节将主要讲解乘方运算。

在 COBOL 中，乘方运算符是由两个连续的乘法运算符组成的，即“**”。其中乘方符号左边的数为进行乘方运算的底数，乘方符号右边的数为该底数的幂。下面这段代码计算 5 的 3 次方，并将结果保存在变量 RESULT 中。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    NUM1            PIC   9.
```

```
 01    NUM2            PIC   9.
 01    RESULT          PIC   999.
*
 PROCEDURE   DIVISION.
       MOVE   5   TO  NUM1.
       MOVE   3   TO  NUM2.
       DISPLAY 'NUM1:',NUM1.
       DISPLAY 'NUM2:',NUM2.
       COMPUTE   RESULT = NUM1  **  NUM2
           ON SEZE ERROR
           DISPLAY  'OVER FLOW!'
       END-COMPUTE.
       DISPLAY 'RESULT:',RESULT.
       STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
NUM1: 5
NUM2: 3
RESULT: 125
```

6.1.8 复合算术运算 COMPUTE

通过上一小节中的例子实际上已可以看到，COMPUTE 语句是直接通过算术表达式进行算术运算的。COMPUTE 语句中通常要包含一个等号。其中等号左边的变量用于保存运算结果，等号右边则为任意的算术表达式。在 COMPUTE 语句中，通常允许包含下面几种参量。

- 常用运算符号
- 数字直接数
- 保存数值数据的变量
- 括号

此外，在算术表达式中需要用空格将各操作数和运算符隔开。但是，在括号及其所包含的内容之间通常是不应该有空格的。关于这一点说明如下。

```
COMPUTE  A  =  (B  +  C  +  D  +   E)  /  4
                ↑                  ↑
```

操作数和运算符间都用空格隔开，括号与所包含的内容之间不应有空格。

实际上，通过 COMPUTE 语句进行的复合算术运算就是由前面所讲的各种基本运算组成的。这里所说的复合算术运算既可以包含 1 条基本运算，也可以包含多条基本运算。下面这段代码通过 COMPUTE 语句进行了各种复合算术运算。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    NUM1            PIC   9.
 01    NUM2            PIC   9.
 01    NUM3            PIC   9.
 01    RESULT          PIC   99.
*
 PROCEDURE   DIVISION.
       MOVE   2   TO  NUM1.
       MOVE   3   TO  NUM2.
```

```
        MOVE    5    TO  NUM3.
        COMPUTE   RESULT = NUM2  +  NUM3
        END-COMPUTE.
        DISPLAY 'RESULT-1:',RESULT.
        COMPUTE   RESULT = (NUM3  -  NUM1)  *  NUM2
        END-COMPUTE.
        DISPLAY 'RESULT-2:',RESULT.
        COMPUTE   RESULT = NUM1  **  NUM2  +  NUM3  *  2
        END-COMPUTE.
        DISPLAY 'RESULT-3:',RESULT.
        COMPUTE   RESULT = (1  +  2)  *  3  -  4
        END-COMPUTE.
        DISPLAY 'RESULT-4:',RESULT.
        COMPUTE   RESULT = NUM1 + ( 3  +  2  -  NUM3)  *  NUM2  /  5 - 2
        END-COMPUTE.
        DISPLAY 'RESULT-5:',RESULT.
         STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
RESULT1: 8
RESULT2: 9
RESULT3: 18
RESULT4: 5
RESULT5: 0
```

通过以上代码，应该可以对如何使用 COMPUTE 语句有一个整体的认识。在此基础上，下面关键需要讲解复合算术运算中各条运算的运算顺序。关于运算顺序，有下面两条基本原则。

- 括号内的先运算，括号外的后运算。
- 当运算符优先级相同时，从左到右顺次进行运算。

其中各运算符的优先级从高到低依此如下。

- 乘方运算符：**。
- 乘除运算符：* 和 /。
- 加减运算符：+ 和 −。

当进行复合算术运算时，首先根据第 1 条基本原则，先算括号内的，再算括号外的。其后，再根据各运算符优先级的高低，先算优先级高的运算，后算优先级低的运算。对应实际运算，也就是先算乘方，再算乘除，最后算加减。最后，当优先级相同时，如乘和除运算、连续的乘运算等，则依照第 2 条基本原则，从左到右顺次运算。

通过下面两段代码对比可以看出，在复合算术运算中，若运算顺序不一致，结果往往也不一致。其中第一段代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    NUM1            PIC    9.
 01    NUM2            PIC    9.
 01    RESULT           PIC    99.
*
 PROCEDURE   DIVISION.
       MOVE    6    TO  NUM1.
       MOVE    3    TO  NUM2.
```

```
       COMPUTE   RESULT = NUM1  +  NUM2  /  3
       END-COMPUTE.
       DISPLAY 'RESULT:',RESULT.
        STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
RESULT: 7
```

该段代码中，根据运算符优先级高低，先进行除法运算。此处即先将 NUM2 中的 3 除以直接数 3，得到 1。然后，再进行加法运算，将 NUM1 中的 6 和除法运算得到的 1 相加，得到的最终结果为 7。

第二段代码将使用和第一段代码相同的操作数和运算符，只是运算顺序不同。此时，将会得到不同的运算结果。该段代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    NUM1            PIC    9.
 01    NUM2            PIC    9.
 01    RESULT          PIC    99.
*
 PROCEDURE   DIVISION.
       MOVE    6    TO  NUM1.
       MOVE    3    TO  NUM2.
       COMPUTE   RESULT = (NUM1  +  NUM2)  /  3
       END-COMPUTE.
       DISPLAY 'RESULT:',RESULT.
        STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
RESULT: 3
```

该段代码中，根据先算括号里的运算，再算括号外的运算的原则，将首先进行加法运算。此时将 NUM1 中的 6 和 NUM2 中的 3 相加，得到 9。然后再算括号外的运算，即除法运算。此时将前面相加的结果 9 除以直接数 3，得到最终结果为 3。这和前面得到的结果 7 是不同的。

最后需补充的一点是，COMPUTE 语句的结束标识符 END-COMPUTE 是在 COBOL-85 版本中推出的。在 COBOL 的前一个版本 COBOL-74 中是没有这一标识符的，该版本中直接使用句点表示该语句的结束。当然，由于 COBOL 是向上兼容的，因此在 COBOL-85 版本中，同样也可以使用句点表示该语句的结束。

6.1.9 算术统计运算 COMPUTE

此处主要介绍的统计算术运算有 3 种，分别为计算总和，计算中位数和计算平均数。统计算术运算只能通过 COMPUTE 语句实现。并且，这也是 COBOL-85 版本所推出的新功能，只能在 COBOL-85 版本中使用。3 种统计算术运算的基本格式分别如下。

- 计算总和：COMPUTE sum-name = FUCTION SUM (name1　name2　name3 …)。
- 计算中位数：COMPUTE middle-name = FUCTION MEDIAN (name1　name2　name3 …)。

- 计算平均数：COMPUTE average-name = FUCTION MEAN (name1　name2　name3 …)。

下面分别结合 3 个实例，具体说明这 3 种统计算术运算在实际中是如何应用的。

1．计算总和

下面这段代码分别使用复合算术运算的方式和统计算术运算的方式计算一串数字的总和。可以看到，这两种方式得到的结果是一致的，但后者是将求和功能进行了集成。代码如下。

```
……
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    NUM1            PIC    9.
01    NUM2            PIC    9.
01    NUM3            PIC    9.
01    TOTAL-1        PIC    99.
01    TOTAL-2        PIC    99.
*
PROCEDURE   DIVISION.
      MOVE    5    TO  NUM1.
      MOVE    6    TO  NUM2.
      MOVE    7    TO  NUM3.
      COMPUTE   RESULT-1  =  (NUM1  +  NUM2  +  NUM3)
      END-COMPUTE.
      COMPUTE   RESULT-2  =  FUNCTION  SUM  (NUM1  NUM2  NUM3).
      DISPLAY 'TOTAL BY COMPLEX ARITHMETIC:'  TOTAL-1.
      DISPLAY 'TOTAL BY FUNCTION ARITHMETIC:'  TOTAL-2.
       STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
TOTAL BY COMPLEX ARITHMETIC: 18
TOTAL BY FUNCTION ARITHMETIC: 18
```

2．计算中位数

所谓中位数，就是一列数值数据中数值的大小正好位于中间的数。下面这段代码通过统计算术运算求取出了 5 个数值数据中其中位数的大小。

```
……
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    NUM1            PIC    9   VALUE     4.
01    NUM2            PIC    9   VALUE     7.
01    NUM3            PIC   99   VALUE    12.
01    NUM4            PIC    9   VALUE     8.
01    NUM5            PIC  999   VALUE  402.
01    MIDDLE-NUM      PIC   99   VALUE    53.
*
PROCEDURE   DIVISION.
      COMPUTE   MIDDLE-NUM  =  FUNCTION  MEDIAN
                             (NUM1  NUM2  NUM3  NUM4  NUM5).
      DISPLAY 'MIDDLE NUM:'  MIDDLE-NUM.
       STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
MIDDLE NUM: 8
```

3．计算平均数

下面这段代码分别使用复合算术运算的方式和统计算术运算的方式计算一串数字的平均数。这两种方式得到的结果是一致的，但后者将功能进行了集成，更为简便。代码如下。

```
......
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    NUM1            PIC    9.
01    NUM2            PIC    9.
01    NUM3            PIC    9.
01    AVG-1           PIC   99.
01    AVG-2           PIC   99.
*
PROCEDURE   DIVISION.
      MOVE    2    TO  NUM1.
      MOVE    3    TO  NUM2.
      MOVE    7    TO  NUM3.
      COMPUTE   AVG-1  =  (NUM1  +  NUM2  +  NUM3)  /  3
      END-COMPUTE.
      COMPUTE   AVG-2  =  FUNCTION  MEAN  (NUM1  NUM2  NUM3).
      DISPLAY 'AVG BY COMPLEX ARITHMETIC:'  AVG-1.
      DISPLAY 'AVG BY FUNCTION ARITHMETIC:'  AVG-2.
       STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
AVG BY COMPLEX ARITHMETIC: 4
AVG BY FUNCTION ARITHMETIC: 4
```

最后需注意的一点是，以上 3 种统计算术运算常用于对 COBOL 中的表进行操作。当对表进行操作时，在此基础上还要加上 ALL 选项。详细情况将在 COBOL 中的表一章中进行讲解。

6.2 关系运算

关系运算主要是对两个操作数间大小的关系进行比较。关系运算的运算结果只有两种，要么为真，要么为假。因此，关系运算通常用于流程控制中的条件判断里。关系运算可以直接使用关系运算符进行表示，也可以使用相应关系运算语句及其简写进行表示。关系运算的表示方式，及其对应的关系主要有以下几种。

- =：运算语句为 EQUAL TO，简写为 EQ，对应等于关系。
- >：运算语句为 GREATER THAN，简写为 GT，对应大于关系。
- <：运算语句为 LESS THAN，简写为 LT，对应小于关系。
- >=：运算语句为 GREATER THAN OR EQUAL TO，简写为 GE，对应大于或等于关系。
- <=：运算语句为 LESS THAN OR EQUAL TO，简写为 LE，对应小于或等于关系。

可以看到，以上 5 种关系实际上是建立在 3 种关系的基础之上的。这 3 种关系分别为等于、大于、小于。其他 2 种关系是对以上 3 种关系的扩展。

下面这段代码依次进行了各种关系运算。通过该段示例代码，可以掌握关系运算在实际

中通常的使用方式。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    NUM1            PIC    9.
 01    NUM2            PIC    9.
 01    NUM3            PIC    9.
 01    CHR1            PIC    X  VALUE  'A'.
 01    CHR2            PIC    X  VALUE  'B'.
*
 PROCEDURE   DIVISION.
       MOVE    2    TO  NUM1.
       MOVE    3    TO  NUM2.
       MOVE    5    TO  NUM3.
       IF  NUM1 = NUM2
          DISPLAY '1-TRUE'
       ELSE
          DISPLAY '1-FALSE'
       END-IF.
       IF  NUM1 + NUM2 > NUM3
           DISPLAY '2-TRUE'
       ELSE
           DISPLAY '2-FALSE'
       END-IF.
       IF  CHR1  <  CHR2
           DISPLAY '3-TRUE'
       ELSE
        DISPLAY '3-FALSE'
       END-IF.
       IF  NUM1 >= NUM3 - NUM2
           DISPLAY '4-TRUE'
      ELSE
       DISPLAY '4-FALSE'
      END-IF.
      IF  NUM1 * NUM2  <=  NUM3 - NUM1
         DISPLAY '5-TRUE'
      ELSE
       DISPLAY '5-FALSE'
      END-IF.
       STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
1-FALSE
2-FALSE
3-TRUE
4-TRUE
5-FALSE
```

通过该段代码应该注意到关系运算包含有以下 3 个特点。

- 关系运算中的等号“=”表示判断两个操作数是否相等。这点和使用 COMPUTE 语句进行的算术运算是不同的。在算术运算中，等号为赋值操作；而在关系运算中，等号为判断比较操作。这一点一定要牢记。
- 关系运算中运算符左右两边的两个操作数可以为算术运算表达式。如上面代码中以下几

条关系运算的操作数就是包含了算术运算表达式的。

```
NUM1 + NUM2 > NUM3
NUM1 >= NUM3 - NUM2
NUM1 * NUM2  <=  NUM3 - NUM1
```

- 关系运算除可以对数值数据进行比较外，也可以对字母及字符串进行比较。字母及字符串的大小是通过字符编码顺序决定的。普通微机的字符编码为 ASCII 码，MainFrame 上的通常为 EBCDIC 编码。上面代码中的以下这条关系运算就是将两个包含字母的变量进行比较。

```
CHR1  <  CHR2
```

但需要注意的是，关系运算只能将同类型的数据进行比较，而不能将字母与数值进行比较。例如，下面这条关系运算就是错误的。

```
CHR1  <  NUM2
```

此外，关系运算通常还可以在其前面加上 NOT，表示对原关系的取反。加上 NOT 后的运算符所表示的关系如下。

- NOT =：表示不等于。
- NOT >：表示不大于。
- NOT <：表示不小于。

由于在>=前加上 NOT 以及在<=前加上 NOT 实际上分别就是<和>所表示的关系。因此，为方便起见，通常不在<=和>=前加上 NOT。实际上，NOT 就是逻辑非运算的运算符。关于逻辑运算，将在下面一节中就进行详细讲解。

6.3 逻辑运算

逻辑运算是建立在多个关系运算的基础之上的，也常用在流程控制中的条件判断里。逻辑运算的结果同样也只有两种，要么为真，要么为假。逻辑运算分为 3 种类型，分别为逻辑与、逻辑或和逻辑非。本节将依次对这 3 种类型的逻辑运算进行讲解，并在此基础上重点讲解复合逻辑运算。最后，关于逻辑运算表达式的常用省略方式也是值得注意的。

6.3.1 逻辑与运算

在 COBOL 中，逻辑与运算是通过逻辑运算符“AND”表示的。下面这段代码使用了逻辑与运算，并且当结果为真时输出“TRUE”，当结果为假时输出“FALSE”。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    DATA1           PIC    99.
 01    DATA2           PIC    XX.
*
 PROCEDURE   DIVISION.
       MOVE    10    TO  DATA1.
       MOVE    'AA'   TO  DATA2.
      IF DATA1 = 10  AND  DATA2 = 'AA'
```

```
          DISPLAY 'TRUE'
       ELSE
          DISPLAY 'FALSE'
       END-IF.
        STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
TRUE
```

实际上，逻辑与运算是通过比较多个用于条件判断的关系运算的结果而得出的最终运算结果。并且，当且仅当所有关系运算的结果都为真时，逻辑与运算的结果才为真。

下面这段代码里的逻辑与运算中有 3 个关系运算，并且其中有一个关系运算的结果为假。通过和上一段代码的比较，可以看出逻辑与运算的特点。代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    DATA1            PIC    99.
 01    DATA2            PIC    XX.
 01    DATA3            PIC     9.
*
 PROCEDURE   DIVISION.
       MOVE    10    TO  DATA1.
       MOVE    'AA'   TO  DATA2.
       MOVE     8    TO  DATA3.
       IF DATA1 = 8  AND  DATA2 = 'AA'  AND DATA3 < 10
          DISPLAY 'TRUE'
       ELSE
          DISPLAY 'FALSE'
       END-IF.
        STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
FALSE
```

该段代码的逻辑与运算中，第一个关系运算“DATA1=8”的结果就为假。由于逻辑与运算中只要其中有一个关系运算的结果为假，整个运算结果就为假。因此，此时不必再去考虑后面两个关系运算（虽然后两个结果都为真），而直接得出运算结果为假。

6.3.2 逻辑或运算

逻辑或运算的运算符为“OR”。和逻辑与运算不同的是，逻辑或运算只要其中有一个关系运算的结果为真，最终结果就为真。下面这段代码说明了逻辑或运算的特点。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    DATA1            PIC    99.
 01    DATA2            PIC    XX.
 01    DATA3            PIC     9.
*
 PROCEDURE   DIVISION.
```

```
       MOVE   10   TO  DATA1.
       MOVE   'AA'  TO  DATA2.
       MOVE    8   TO  DATA3.
       IF DATA1 = 8  OR  DATA2 = 'AA'  OR  DATA3 < 10
          DISPLAY 'TRUE'
       ELSE
          DISPLAY 'FALSE'
       END-IF.
       STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
TRUE
```

该段代码的逻辑或运算中，第一个关系运算“DATA1=8”为假，此时继续进行后面的关系运算。第二个关系运算“DATA2='AA'”为真。由于逻辑或运算中只要有一个关系运算结果为真，最终结果就为真。因此，此时不必再去理会第 3 个关系运算，而直接得出其最终结果为真。

6.3.3 逻辑非运算

逻辑非运算的运算符为“NOT”。逻辑非运算和前面两种逻辑运算有所不同，该运算中并不包含多个关系运算语句。逻辑非运算实际上就是对逻辑结果求反，将逻辑真变为假，逻辑假变为真。下面这段代码说明了逻辑非运算的特点。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    DATA1           PIC    99.
*
 PROCEDURE   DIVISION.
       MOVE   10   TO  DATA1.
      IF DATA1 = 8
         DISPLAY 'BEFORE  "NOT"  IS  TRUE'
       ELSE
         DISPLAY 'BEFORE  "NOT"  IS  FALSE'
       END-IF.
       IF DATA1 NOT = 8
         DISPLAY 'AFTER  "NOT"  IS  TRUE'
       ELSE
         DISPLAY 'AFTER  "NOT"  IS  FALSE'
       END-IF.
        STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
BEFORE 'NOT' IS TRUE
AFTER 'NOT' IS FALSE
```

6.3.4 复合逻辑运算

复合逻辑运算是逻辑运算一节中的重点内容。注意到前面所讲的复合算术运算是由多条基本算术运算所组成的。与之类似，复合逻辑运算则主要是由多条基本逻辑运算所组成的。学习复合逻辑运算，关键要掌握其包含的各条基本逻辑运算，以及其他一些常用运算的处理顺序。通常，复合逻辑运算中包括基本逻辑运算在内，各项运算的处理顺序由先到后依次如下。

- 算术运算
- 关系运算
- 逻辑非运算
- 逻辑与运算（如果同时多个联系出现，则从左至右依次执行）
- 逻辑或运算（如果同时多个联系出现，则从左至右依次执行）

下面首先看一段示例代码，该代码中使用了复合逻辑运算。在给出其输出结果之后，再回头分析其复合逻辑运算是如何一步步执行的。代码如下。

```
......
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    DATA1           PIC    XX  VALUE 'AA'.
01    DATA2           PIC    XX  VALUE 'BB'.
01    DATA3           PIC    XX  VALUE 'CC'.
01    NUM1            PIC    99  VALUE 10.
01    NUM2            PIC    99  VALUE 20.
01    NUM3            PIC    99  VALUE 30.
*
PROCEDURE   DIVISION.
     IF  DATA1 NOT = DATA2  AND  NUM1 > NUM2  AND  (NUM1 + NUM2) = NUM3
        OR  DATA2  <=  DATA3
        DISPLAY 'THE RESULT IS TRUE'
     ELSE
        DISPLAY 'THE RESULT IS FALSE'
     END-IF.
      STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
THE RESULT IS TRUE
```

该段代码中所重点涉及到的复合逻辑运算实际上可提取为以下这条运算表达式。

```
DATA1 NOT = DATA2  AND  NUM1 > NUM2  AND  (NUM1 + NUM2) = NUM3   OR  DATA2  <=  DATA3
```

为分析方便，首先将各变量名替换为其所包含的实际内容，表达式如下。

```
'AA' NOT = 'BB'  AND  10 > 20  AND  (10 + 20) = 30   OR  'BB'  <=  'CC'
```

下面，根据复合逻辑运算中各运算的先后顺序，具体分析该条复合逻辑运算表达是如何执行的。

（1）根据前面所讲的运算顺序，首先进行算术运算。该表达式中的算术运算只有一条，即算术加。执行完算术运算后表达式如下。

```
'AA' NOT = 'BB'  AND  10 > 20  AND  30 = 30   OR  'BB'  <=  'CC'
```

（2）算术运算执行后，将执行关系运算。该表达式中的关系运算有 3 条，分别为大于，等于，以及小于或等于。关系运算的结果要么为真，要么为假。此处不妨以符号“T”代表真，符号“F”代表假。执行关系运算后表达式如下。

```
'AA' NOT = 'BB'  AND  F  AND  T   OR  T
```

此处需要注意的是，关于'BB' <= 'CC'这条关系运算实际上是对字符串大小的比较。

根据 mainframe 的 EBCDIC 编码机制，字符 B 是小于字符 C 的。此处只要字符串 BB 小于或等于字符串 CC 便为真。因此，该条关系运算结果为真。

（3）接下来再进行逻辑非运算。实际上，该表达式中的逻辑非运算是和关系运算等于结合在一起的。由于字符串 AA 和 BB 显然是不相同的，因此运算结果为真。此时，表达式将如下。

```
T  AND  F  AND  T   OR  T
```

（4）其后进行逻辑与运算。以上表达式中有两条连续的逻辑与运算，因此此处将从左至右依次执行。根据逻辑与运算的运算特点，执行完第一条运算后的表达式如下。

```
F      AND  T   OR  T
```

执行完第二条逻辑与运算后，表达式如下。

```
F       OR  T
```

（5）最后，执行逻辑或运算。以上表达式中仅包含一条逻辑或运算。根据逻辑或运算的运算特点，最终的到的整个复合逻辑运算的结果如下。

```
          T
```

此外，也可以在复合逻辑运算表达式中加上括号。根据通常运算中先算括号内的，再算括号外的顺序，可以改变和控制复合逻辑运算的运算顺序。通过使用括号，往往能使运算顺序更加清晰。下面使用括号注明了上例中的复合逻辑运算表达式。此处使用的括号没有改变该表达式原本的运算顺序，因此得到的结果也是一致的。使用括号注明后的表达式如下。

```
(((DATA1 NOT = DATA2)  AND  (NUM1 > NUM2))  AND  ((NUM1 + NUM2) = NUM3))   OR  (DATA2  <=  DATA3)
```

这样，对应每层括号都为一独立的运算，其中只包含一个运算符。显然，注明后该运算表达式的结构将更加清晰。但需注意的是，若不清楚其原本的执行顺序，乱用括号往往会改变原本的顺序，并得到错误的结果。下面这则包含括号的表达式将得到和原表达式相反的结果。

```
(DATA1 NOT = DATA2)  AND  (NUM1 > NUM2)  AND  (((NUM1 + NUM2) = NUM3)   OR  (DATA2  <=  DATA3))
```

最后需要说明的一点是，复合逻辑运算通常是用于流程控制中的条件判断里的。在实际中，过于复杂的复合逻辑运算往往会造成结构比较混乱，难于理解和判断。此时，通常使用多层嵌套的 IF 语句来进行替代。关于 IF 语句，将在流程控制一章中进行更详细的讲解。

6.3.5 逻辑运算表达式中常用的省略方式

逻辑运算表达式中，若前一个运算的操作数和后一个相同，通常可省略后一个运算中的左操作数。此处所说的左操作数，即出现在运算符左边的操作数。此时，前一个运算的左操作数将暗含在后一个运算中。下面这段代码通过比较，说明了在逻辑运算表达式中具体是如何进行省略的。

```
......
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01    TEST-NUM         PIC     99.
*
```

```
 PROCEDURE   DIVISION.
       MOVE  18  TO  TEST-NUM.
       IF  TEST-NUM > 8  AND  TEST-NUM < 10
          DISPLAY  'TRUE'
       ELSE
          DISPLAY  'FALSE'
       END-IF.
       IF  TEST-NUM > 8  AND  < 10              /*此处省略了第二个运算中的左操作数*/
          DISPLAY  'WITH OMITTED IDENTIFIER: TRUE'
       ELSE
          DISPLAY  'WITH OMITTED IDENTIFIER: FALSE'
       END-IF.
        STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
TRUE
WITH OMITTED NUM: TRUE
```

该段代码中，逻辑与运算两边的两个关系运算中左操作数都为 TEST-NUM，因此可以进行省略。省略后，该运算的性质及其得到的运算结果是不变的。

同时，当后一个运算中不仅左操作数和前一个的相同，而且运算符也相同时，连运算符也可省略。下面这段代码揭示了这一点。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    TEST-NUM        PIC     99.
*
 PROCEDURE   DIVISION.
      MOVE  18  TO  TEST-NUM.
      IF  TEST-NUM = 8  OR  TEST-NUM = 10
         DISPLAY  'TRUE'
      ELSE
         DISPLAY  'FALSE'
      END-IF.
      IF  TEST-NUM = 8  OR  10              /*此处省略了第二个运算中的左操作数和运算符*/
         DISPLAY  'WITH OMITTED IDENTIFIER AND OPERATOR: TRUE'
      ELSE
         DISPLAY  'WITH OMITTED IDENTIFIER AND OPERATOR: FALSE'
      END-IF.
       STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
FALSE
WITH OMITTED IDENTIFIER AND OPERATOR: FALSE
```

最后需要注意的一点是，当进行省略后，前一个运算的逻辑非运算是不能暗含在后一个运算中的。例如，下面为一条含逻辑非运算的表达式，其中省略了前一个运算的左操作数。

```
TEST-NUM NOT = 8 OR = 10
```

对于以上省略后的表达式，初学者往往容易想当然地认为其对应的原始表达式如下。

```
TEST-NUM NOT = 8 OR TEST-NUM NOT = 10      /*错误的*/
```

实际上，此处的逻辑非运算并不能暗含在第二个运算式中。因此，其真正对应的原始表达式应该为如下这条。

```
TEST-NUM NOT = 8 OR TEST-NUM = 10
```

最后需补充的一点是，虽然在编码中可以对逻辑运算表达式进行部分省略，但并不提倡初学者进行。在初学 COBOL 时，最好还是将完整的表达式写出来，以利于观察和学习，并且不易出错。当达到一定熟练程度后，再进行省略为好。

6.4 本章回顾

本章主要讲解了 COBOL 程序中所涉及到的基本运算。此处所说的基本运算包括 3 大类运算，分别为算术运算、关系运算以及逻辑运算。

在算术运算中，本章首先介绍了对运算结果进行处理的两种常用方式。其中一种为对结果中的小数进行四舍五入操作。另一种为当结果数据溢出时进行相应的处理。

其后，依次介绍了算术运算中常用的加减乘除四则运算。其中每种运算又包含两种形式。一种为将运算结果保存在一个操作数，另一种为将运算结果保存在一个新的变量中。

在介绍完基本的四则运算后，最后又分别介绍了如何使用 COMPUTE 语句进行一些算术运算。其中依次为使用 COMPUTE 语句进行乘方运算、复合算术运算和统计算术运算。在统计算术运算中，主要包含 3 种类型的运算，分别为计算总和计算中位数和计算平均数。关于这 3 种类型的统计算术运算，还将在后面要学习的 COBOL 中的表中所涉及到。

在讲解完算术运算后，本章接下来讲解了关系运算。关系运算中共分为 5 种关系比较，分别为等于、大于、小于、大于等于和小于等于。关系运算的操作数中常常包含着前面所讲的算术运算。同时，关系运算也是后面学习逻辑运算的基础。

本章最后讲解了逻辑运算。逻辑运算是建立在多个关系运算的基础之上的。逻辑运算主要包含 3 种，分别为逻辑与运算、逻辑或运算和逻辑非运算。介绍完以上 3 种基本的逻辑运算后，本章重点讲解了复合逻辑运算及其内部运算顺序。同时，在最后还简单介绍了一下逻辑运算表达式在实际编程中书写上的常用省略方式。

学习本章，需要在牢固掌握各种运算的基础上，能够对其进行灵活的应用。由于 COBOL 程序主要是用于对大量数据进行的处理，因此本章所讲内容在实际开发中将会经常用到。

第 7 章 流程控制

本章主要介绍 COBOL 程序中常用的流程控制。程序中最基本的流程结构有 3 种，分别为顺序结构、选择结构和循环结构。本章将依次对这 3 种基本流程进行详细介绍。学习本章不仅要学习 COBOL 语言是如何进行以上 3 种流程控制的，而且要学习程序流程设计的思想。流程是任何算法的基础，算法是任何程序的灵魂。

7.1 顺序结构流程控制

顺序结构是 3 种流程结构中最基本的一种。顺序结构实际上就是顺序执行程序中的每条语句。虽然通常按照人的常规思维，一般程序本该是如此执行的。但根据严格的程序流程定义，作为一种单独的流程结构，还是有必要进行说明的。

下面这一段完整的程序根据 4 个季度的销售额得出全年的总销售额，以及每个季度的平均销售额。其中每个季度的销售额通过用户从键盘输入，并且输出结果需要包含标题。该程序使用的便是顺序结构，程序代码如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID       SEQUENTIAL-PROG.
 AUTHER           XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    SALES-1          PIC  9(6)V9.
 01    SALES-2          PIC  9(6)V9.
 01    SALES-3          PIC  9(6)V9.
 01    SALES-4          PIC  9(6)V9.
 01    ANNUAL-TOTAL     PIC  9(7).
 01    ANNUAL-AVERAGE   PIC  9(6).
 01    TITLE            PIC  X(50).
*
 PROCEDURE   DIVISION.
```

```
*
 DISPLAY-TITLE.
       MOVE '   ANNUAL SALES REPORT '  TO TITLE.
       DISPLAY  TITLE.
*
 GET-QUARTER-SALES.
       DISPLAY   'THE FIRST QUARTER SALES :'.
       ACCEPT   SALES-1.
       DISPLAY   'THE SECOND QUARTER SALES :'.
       ACCEPT   SALES-2.
       DISPLAY   'THE THIRD QUARTER SALES :'.
       ACCEPT   SALES-3.
       DISPLAY   'THE FOURTH QUARTER SALES :'.
       ACCEPT   SALES-4.
*
 COMPUTER-ANNUAL-SALE.
       COMPUTE  ANNUAL-TOTAL  =  (SALES-1 + SALES-2 + SALES-3 + SALES-4)
                                 ROUNDED.
       COMPUTE  ANNUAL-AVERAGE = (SALES-1 + SALES-2 + SALES-3 + SALES-4)  /  4
                                 ROUNDED.
*
 DISPLAY-RESULT.
       DISPLAY 'ANNUAL  TOTAL: '  ANNUAL-TOTAL.
       DISPLAY 'ANNUAL  AVERAGE: '  ANNUAL-AVERAGE.
       STOP  RUN.
```

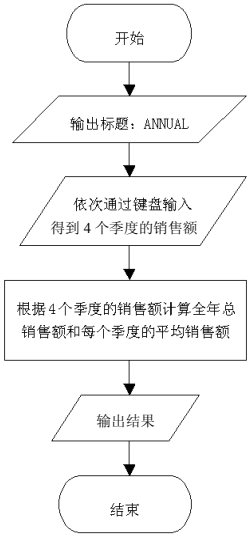

图 7.1　顺序结构示例程序流程图

该程序执行后，应该有如下输出结果。

```
            ANNUAL SALES REPORT
THE FIRST QUARTER SALES :
104784.5
THE SECOND QUARTER SALES :
124784.3
THE THIRD QUARTER SALES :
100987.6
```

```
THE FOURTH QUARTER SALES :
135476.8
ANNUAL  TOTAL:  466033
ANNUAL AVERAGE:  116508
```

该段示例程序中各个季度的销售额是通过用户从键盘上直接输入的。以上结果显示中的季度销售额为任意选取的 4 个数。

通过该段程序，关键要理解顺序结构在实际中的应用。下面给出该程序的流程图，以便更好的理解程序流程中的顺序结构。该程序流程图如图 7.1 所示。

7.2 选择结构流程控制

选择结构相对顺序结构要复杂一些，本节将分成几个小节来详细讲解该结构流程。在 COBOL 中，控制选择结构的语句主要为 IF 语句和 EVALUATE 语句。同时，ZERO 和 88 层条件名的使用也常用来简化选择结构的编码。本节将对以上内容分别进行讲解。

7.2.1 选择结构的基本流程

选择结构是根据选择条件判断的结果而选择不同的语句进行执行。条件判断只有两种结果，要么为真，要么为假。因此，选择结构中，对于某一条主干语句只有两条分支语句。并且，这两条分之语句中的其中一条也可为空语句。同时，当某一分支语句作为一个新的主干语句时，又可以对应两条新的分支语句。此时便形成了嵌套选择结构。下面分别对这 3 种情况介绍之。

1. 基本选择结构

下面直接给出基本选择结构的流程图。通过该流程图，可以更好的理解选择结构的基本流程。流程图如图 7.2 所示。

2. 一个分支语句为空语句的选择结构

下面直接给出一个分支语句为空语句的选择结构流程图。通过该流程图，可以更好地理解在其中一个分支语句为空语句的情况下，选择结构的流程，如图 7.3 所示。

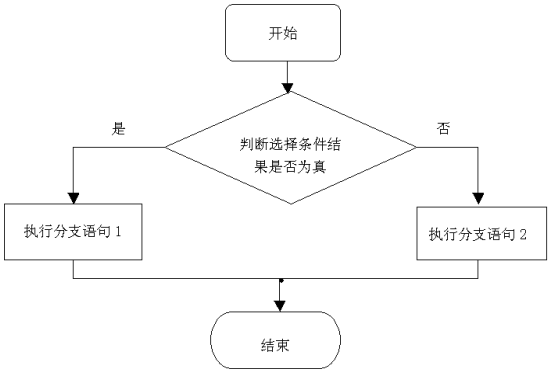

图 7.2 基本选择结构的流程图

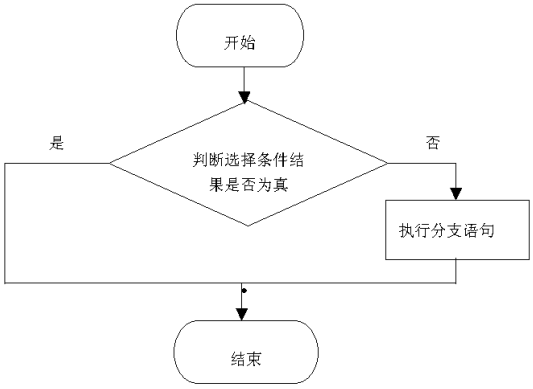

图 7.3 一个分支语句为空语句情况下选择结构的流程图

3．嵌套的选择结构

下面仍然直接给出一个嵌套选择结构的流程图。该嵌套选择结构含有两层嵌套。通过两层嵌套，也可以类推到多层嵌套。通过该嵌套选择结构的流程图，可以更好地理解嵌套选择结构的流程。该流程图如图 7.4 所示。

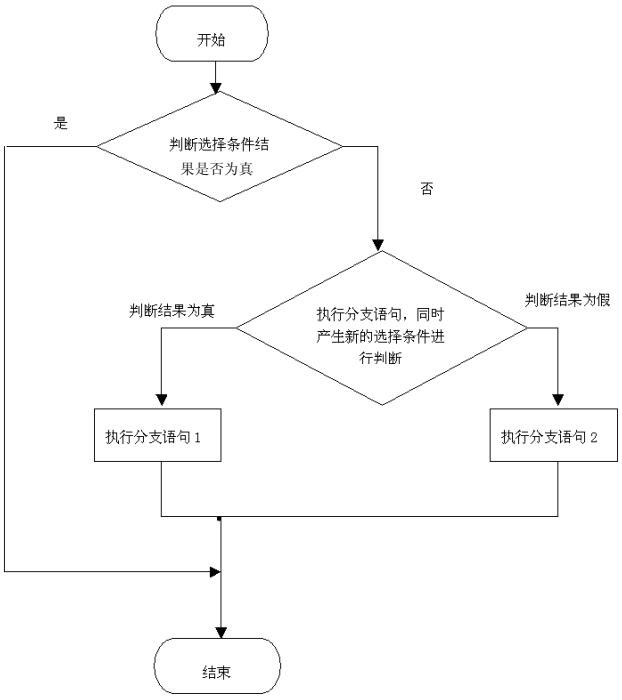

图 7.4　嵌套选择结构的流程图

7.2.2　条件判断表达式

选择结构流程中具体选择哪一条分支执行是根据其中的条件判断得到的。因此，条件判断对于选择结构流程而言至关重要。实际上，条件判断不仅用于选择结构，同样也用于后面要讲的循环结构中。循环结构中是通过条件判断得出其具体循环次数的。此处主要讲解选择结构中的条件判断表达式。

在程序流程图中，条件判断的内容按规定是写在菱形框内的，以区别于程序中其他通常的执行内容。此外需要注意的是，条件判断只在选择结构和循环结构中出现。在顺序结构中，由于程序的所有语句都是顺序执行的，因此没有条件判断。

条件判断是由条件判断表达式所具体形成的。条件判断表达式主要是由前面所讲的关系运算和逻辑运算组成的。例如，下面这几条表达式就是几种常见的选择结构中的条件判断表达式。

```
IF  NUM > 10…
IF  STR = '123'…
IF  CHAR >= 'A'…
```

```
IF  STR  IS  LESS THAN  '456'…
IF  NUM1  IS  NOT = NUM2…
IF  CHAR  IS  GREATER THAN OR EQUAL TO 'ABC'…
……
```

以上各条件判断表达式后面的省略号表示省略完成该条件判断后，所执行的相关操作。可以看到，在选择结构中的条件判断表达式都是紧跟在 IF 后面的。并且，这些条件判断表达式实际上就是前面所讲的关系运算表达式和逻辑运算表达式。

对于以上表达式，为更清楚地指明条件，在表达式中加上了 IS 选项。实际上，加或者不加 IS 项在通常情况下是等价的。并且为方便起见，通常是不加 IS 项的。例如，下面这两条语句就是等价的，并且通常采用第 2 条语句。

```
IF  STR  IS  LESS THAN OR EQUAL TO '456'…
IF  STR  <= '456'…
```

需要再次强调的一点是，在条件判断表达式中，只能将同类型的数据进行比较。此处所说的同类型数据通常包含两类，一类是数值数据，另一类是字符型数据。即只能将数值数据和数值数据比较，字符型数据和字符型数据比较。将数值数据和字符型数据比较是不允许的，这在逻辑上也是不通的。

例如，下面这几条条件判断表达式就是合法的。

```
IF  3 > 4…
IF  NUM1 <= NUM2…
IF  'AA' > 'BB'…
IF  CHAR1 NOT = CHAR2…
IF  STR1 > STR2…
IF   CHAR < STR…
IF  '123' = '456'…
```

以下几条条件判断表达式是非法的。

```
IF  3 > 'A'…
IF  NUM <= CHAR…
IF  STR > NUM…
IF  CHAR NOT = 2…
IF  123 = '123'…
```

总之，在流程控制中，主要通过条件判断表达式来控制程序中各语句的执行流程的。本小节内容既是对前面所学的关系运算和逻辑运算的一个回顾，也是为更复杂的流程控制奠定一个基础。

7.2.3 使用 IF 语句控制选择结构流程

在 COBOL 中对选择结构流程进行具体编码，通常是通过 IF 语句和 EVALUATE 语句实现的。本节将介绍如何使用 IF 语句来控制选择结构的流程。

下面假设需要比较两个数的大小，并且根据比较结果输出相应信息。实现这一功能的程序将对应于最基本的选择结构。最基本的选择结构也就是对主干语句进行条件判断后有两条分支语句以供执行。该程序对应的流程图如图 7.5 所示。

此处需要补充的一点是，在画选择结构的流程图时，通常是将条件判断结果为真的分支放在左边。这样做的原因是根据人从左到右的思维习惯，通常会先去看左边的分支语句。而

在程序中，根据从上到下的顺序，条件判断结果为真的语句是在前面的。因此，将条件判断结果为真的分支语句放在选择结构流程图的左边分支更利于观察。

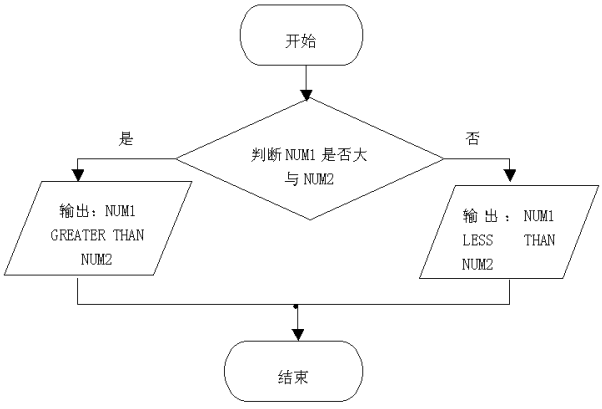

图 7.5　比较两数大小流程图

对应上面的流程图，下面这段代码使用 IF 语句实现了程序所要求的功能。代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    NUM1            PIC    9  VALUE 1.
 01    NUM2            PIC    9  VALUE 2..
*
 PROCEDURE   DIVISION.
       IF  NUM1 >  NUM2
          DISPLAY  ' NUM1 GREATER THAN NUM2'
       ELSE
          DISPLAY  ' NUM1 LESS THAN NUM2'
       END-IF.
        STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
NUM1 LESS THAN NUM2
```

下面再假设有一信息安全管理系统。当登录该系统时，需要判断管理员名称是否为“TOM”，以及密码是否为“1234”。若符合条件则输出相应信息，并将该系统启动指示标志置为“ON”；否则不执行任何操作。实现该功能的程序流程对应于一个分支语句为空语句的选择结构流程。该流程图如图 7.6 所示。

对应上面的流程图，下面这段代码使用 IF 语句实现了程序所要求的功能。代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    ADMIN           PIC   XXX.
 01    ADMIN-SOURCE    PIC   XXX  VALUE 'TOM'.
 01    PASSWORD        PIC   XXXX.
 01    PW-SOURCE       PIC   XXXX  VALUE '1234'.
 01    SYS-STATE-FLAG  PIC   X(5).
*
```

```
 PROCEDURE   DIVISION.
       DISPLAY  'ENTER ADMIN NAME: '.
       ACCEPT  ADMIN.
       DISPLAY  'ENTER PASSWORD: '.
        ACCEPT  PASSWORD.
       IF  ADMIN = ADMIN-SOURCE AND PASSWORD = PW-SOURCE
         DISPLAY 'SYSTEM IS POWERED ON'
         MOVE 'ON' TO SYS-STATE-FLAG
      END-IF.
       STOP RUN.
```

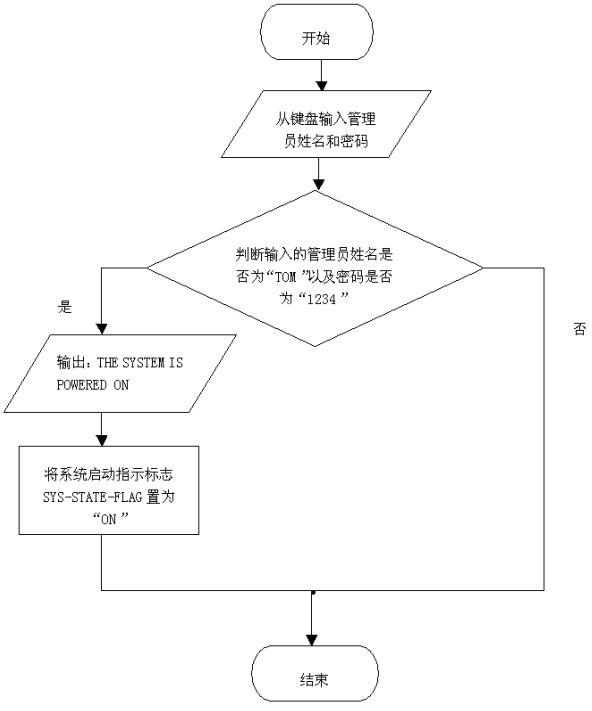

图 7.6　信息安全系统登录流程图

以上代码运行后，将有如下输出结果。

```
ENTER ADMIN NAME:
TOM
ENTER PASSWORD:
1234
SYSTEM IS POWERED ON
```

其中管理员姓名“TOM”和密码“1234”是由用户从键盘上输入的。若输入其他姓名或密码，将不会执行任何操作。例如，当输入错误的密码后，将会有如下输出结果。

```
ENTER ADMIN NAME:
TOM
ENTER PASSWORD:
4321
```

通过对比可以发现，对于基本的选择结构流程，使用 IF 语句进行控制的格式如下。

```
IF condition
```

```
  do something
ELSE
  do something else
END-IF.
```

对于一边分支语句为空语句的选择结构流程，使用 IF 语句进行控制的格式如下。

```
IF condition
  do something
END-IF.
```

二者的区别在于前一个 IF 语句中是有 ELSE 项的，而后一个 IF 语句中是没有 ELSE 项的。ELSE 就是用来指定当不满足 IF 后所列条件时（即条件判断表达式结果为假），程序将如何执行。

此外，在 IF 语句的条件判断表达式后还可以加上 THEN 选项。加上 THEN 选项一般只是根据各人的编码习惯而来，同不加 THEN 选项效果通常是等价的。下面是加上 THEN 选项后的表示形式。

```
IF condition THEN
  do something
ELSE
  do something else
END-IF.
```

7.2.4 使用嵌套 IF 语句控制选择结构流程

当一条 IF 语句中执行的分支语句中也含有 IF 语句时，就形成了嵌套的 IF 语句。嵌套 IF 语句主要是为了控制嵌套的选择结构流程的。

如下为一个嵌套 IF 语句的结构。

```
IF  condition1
|   IF  condition2
|   |   statements2
|   |   IF  condition3
|   |   |   statements3
|   |   END-IF
|   ELSE
|   |   statements4
|   END-IF
ELSE
|   statements5
END-IF.
```

通过该结构可以看到，该嵌套 IF 语句共嵌套了 3 层。其中最里面一层为一个不含 ELSE 项的 IF 语句。外面两层都为含有 ELSE 项的标准 IF 语句。此处使用竖线将同一层的 IF 语句连接了起来，以便于观察其结构。

嵌套 IF 语句中存在多个 IF，ELSE 以及 END-IF 项。相比单纯的 IF 语句，要复杂得多。对于嵌套 IF 语句，一定要特别注意以下两点。

- 在嵌套 IF 语句中，END-IF 将和离其最近同时又没有其他 END-IF 配对的 IF 进行配对。
- 在嵌套 IF 语句中，ELSE 将和离其最近同时又没有其他 ELSE 配对的 IF 进行配对。

在实际编程中，有人常常为图方便，省略掉 END-IF 项不写。虽然此时程序也可正常执

行，但在嵌套 IF 语句中省略掉 END-IF 项往往会改变程序原本的结构。此时，虽然不会有语法错误，但会出现逻辑错误。

例如，当省略掉以上示例嵌套 IF 语句中的 END-IF 时，其结构将变成如下所示。

```
IF  condition1
    IF  condition2
        statements2
        IF  condition3
            statements3
    ELSE
        statements4
ELSE
    statements5.
```

该结构的改变实际上是根据以下这点原则所来的。

“在嵌套 IF 语句中，ELSE 将和离其最近同时又没有其他 ELSE 配对的 IF 进行配对。”

因此，在编程中书写 IF 语句时，特别是嵌套 IF 语句，最好应该在每条 IF 后面加上对应的 END-IF 语句。在初学编程时，一定要养成良好的编程习惯。

下面举一个实例用以说明嵌套 IF 语句的用法。设有一堆物品，数量为 10。另外有两个背包，较大的背包最多可以装 8 个物品，较小的最多可以装 3 个物品。现要求如下。

- 首先用较大的背包来装所有的物品。若全部装下，则输出 “COMPLETE”；否则，将该较大的背包装满，程序继续执行。
- 将剩下的物品用较小的背包来装。若全部装下，则输出 “COMPLETE”；否则，说明物品用这两个背包装不完，输出 “FAIL”。

根据以上要求，程序需要采用嵌套选择结构流程来完成，所对应的流程图如图 7.7 所示。

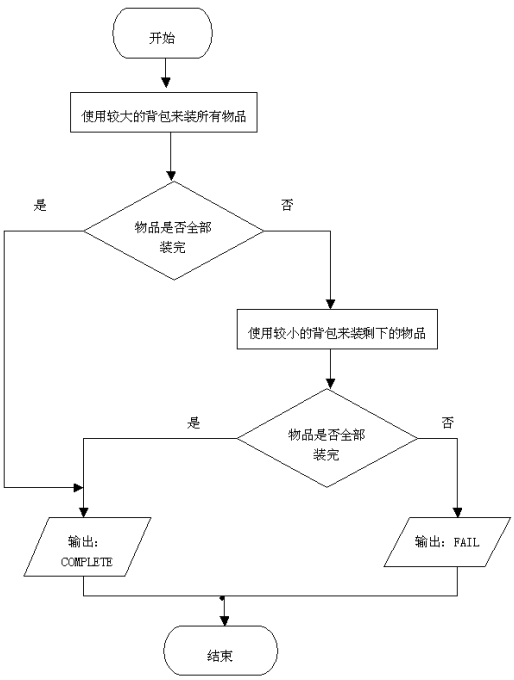

图 7.7　采用大小两个背包来装物品的流程图

对应该流程图，下面采用嵌套 IF 语句来具体实现其功能。代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01    PACK-BIG        PIC   9 VALUE 8.
 01    PACK-SMALL      PIC   9 VALUE 3.
 01    ITEM-TOTAL      PIC   99 VALUE 10.
 01    ITEM-REST       PIC   9.
*
 PROCEDURE   DIVISION.
       IF  ITEM-TOTAL  <=  PACK-BIG
           DISPLAY 'COMPLETE'
       ELSE
           SUBTRACT PACK-BIG FROM ITEM-TOTAL GIVING ITEM-REST
           IF  ITEM-REST  <=  PACK-SMALL
                DISPLAY 'COMPLETE'
           ELSE
                DISPLAY 'FAIL'
           END-IF
       END-IF.
        STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
COMPLETE
```

由于程序中较大背包容量为 8，第一次将其装满后，物品还剩 2 个，继续用较小背包来装。由于较小背包容量为 3，大于所剩物品数 2，因此可以装满。结果输出为“COMPLETE”。

此外需要说明的是，该段代码所实现的并不是所谓的“背包问题”。在“背包问题”中，背包不止两个，因此通常要用到递归和回溯的算法。但是该代码所含的算法思想和“背包问题”是类似的，即最优化算法思想。简单地说，就是该程序执行的思路是先用较大的背包来装物品，而并不是最先用较小的背包来装。

最后需要补充的一点是，在 COBOL-74 版本中是没有 END-IF 结束标志的。COBOL-74 里采用 NEXT SENTENCE 选项用以控制嵌套 IF 语句的结构。例如，对应于前面示例的嵌套 IF 语句结构，采用 NEXT SENTENCE 选项方式代码将如下。

```
IF  condition1
   IF  condition2
      statements2
      IF  condition3
         statements3
      ELSE
         NEXT SENTENCE
   ELSE
      statements4
ELSE
   statements5.
```

可以看出，NEXT SENTENCE 实际上起到了跳出里层嵌套的结构，而到达外层嵌套的作用。然而。在新的 COBOL-85 版本中，通常不使用 NEXT SENTENCE 来控制嵌套 IF 语句的结构。因为在 COBOL-85 中，NEXT SENTENCE 所实现的功能实际上已经被 END-IF 所取代了。

7.2.5 使用 EVALUATE 语句控制多分支选择结构流程

前面所讲的通常的选择结构只有两条分支选择。多分支选择结构实际上也是建立在通常的选择结构基础之上的。但多分支选择结构在选择条件后有多条分支以供选择。EVALUATE 语句正是用于控制多分支选择结构流程的。同时，EVALUATE 语句也是在 COBOL-85 版本中所新推出的功能语句。该语句实际上是建立在 IF 语句的基础之上的。在 COBOL-74 中，可以用 IF 语句控制的只有两条分支的选择结构替代多分支选择结构。

此处需要注意到，前面所说的只有两条分支语句的选择结构是对于选择结构的本质而言的。多分支选择结构是建立在该选择结构的本质的基础之上发展而来的，相当于一个特殊的选择结构。例如，以下是一段采用嵌套 IF 语句控制的只有两条分支的选择结构。

```
IF  condition1
   statements1
ELSE
   IF  condition2
      statements2
   ELSE
       IF condition3
         statements3
       ELSE
         Statements4
       END-IF
   END-IF
   END-IF.
```

以该两条分支的选择结构为基础，可将其改写成为较特殊的多分支选择结构。通过 EVALUATE 语句改写的多分支选择结构代码如下。

```
EVALUATE  TRUE
  WHEN  condition1
    statements1
  WHEN  condition2
    statements2
  WHEN  condition3
    statements3
  WHEN  OTHER
    statements4
 END-EVALUATE.
```

下面结合一个实例，具体说明 EVALUATE 语句在实际编程中是如何应用的。该实例所要实现的功能是由用户在键盘上任意输入一个数字，并根据输入情况产生相应输出。其中对于输入数据，要求必须为从 1～4 的整数，否则将输出错误提示信息。如果采用多分支选择结构，则实现以上功能的流程图如图 7.8 所示。

对应以上图例，使用 EVALUATE 语句实现该功能的代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE   SECTION.
 01    INPUT-DATA       PIC   9.
*
```

```
PROCEDURE   DIVISION.
     DISPLAY 'ENTER A CHOICE TO SELECT A FUNCTION: '.
     ACCEPT INPUT-DATA.
     EVALUATE  TRUE
        WHEN   INPUT-DATA = 1
             DISPLAY 'FUNCTION 1'
        WHEN   INPUT-DATA = 2
             DISPLAY 'FUNCTION 2'
        WHEN   INPUT-DATA = 3
             DISPLAY 'FUNCTION 3'
        WHEN   INPUT-DATA = 4
             DISPLAY 'FUNCTION 4'
        WHEN   OTHER
             DISPLAY 'WRONG'
     END-EVALUATE.
      STOP RUN.
```

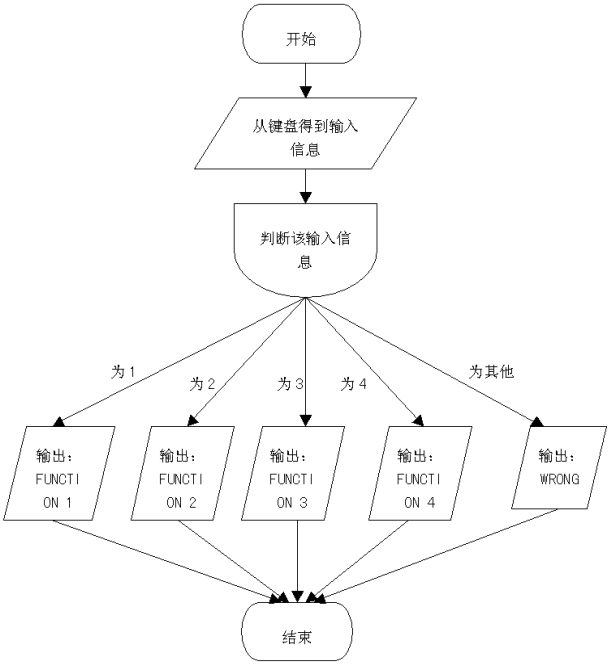

图 7.8　对输入数据进行相关处理过程的流程图

以上代码运行后，若用户输入数字 3，将有如下输出结果。

```
ENTER A CHOICE TO SELECT A FUNCTION:
3
FUNCTION 3
```

若用户输入数字 5，将有如下输出结果。

```
ENTER A CHOICE TO SELECT A FUNCTION:
5
WRONG
```

因此，最终的输出结果是根据具体输入数据的不同而不同的。同时，在实际中为方便起见，也可将以上代码中的 EVALUATE 语句写为如下形式。

```
    EVALUATE  INPUT-DATA
          WHEN   1
                 DISPLAY 'FUNCTION 1'
          WHEN    2
                DISPLAY 'FUNCTION 2'
          WHEN    3
                DISPLAY 'FUNCTION 3'
          WHEN    4
                DISPLAY 'FUNCTION 4'
          WHEN   OTHER
                DISPLAY 'WRONG'
      END-EVALUATE.
```

此处 EVALUATE 语句中的条件判断结果似乎为多项，分别为 1、2、3、4 以及其他。这似乎与前面所讲的选择结构中条件判断只有真假两种结果相矛盾。实际上，从程序本质的角度来看，EVALUATE 语句实际上是有多项条件判断所组成的。因此，对于每项条件判断而言，仍然只有真假两个结果。

例如，以上代码中的 EVALUATE 语句实际上共有 4 项条件判断，分别如下。

```
INPUT-DATA = 1
INPUT-DATA = 2
INPUT-DATA = 3
INPUT-DATA = 4
```

实际执行中，前 3 个条件判断分别对应一个分支语句为空语句的选择结构。当满足条件，即条件判断结果为真时，输出相应信息。当不满足条件，即条件判断结果为假时，则不做任何操作。最后一个条件判断对应一个基本的选择结构。当其条件判断结果为真时，输出“FUNCTION 4”。当条件判断结果为假时，输出“WRONG”。

总之，EVALUATE 语句实际上是基于 IF 语句所发展而来的。任何 EVLAUATE 语句所写的代码都可通过相应的 IF 语句进行改写。当然，能够使用 EVALUATE 语句编码的时候，尽量还是用该语句编码。这样结构上会更加直观，编码上也更加简便。

7.2.6 使用 ZERO 简化选择结构编码

在前面的基本数据类型一章中曾提到，ZERO 属于一个象征常量。该象征常量所象征的内容为数字 0 或者字符“0”。因此，当在选择结构中的条件判断里要使用数字 0 或者字符“0”时，用 ZERO 替代将更为方便一些。使用 ZERO 后，将不必再去理会此处应该是数字 0 还是字符“0”。COBOL 系统将自动根据实际情况进行相应的转换。

例如，下面这段代码判断数值变量 NUM，及字符变量 CHAR 是否分别为数字 0 和字符“0”。如果是则执行相应输出，否则不进行任何操作。代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE   SECTION.
 01    NUM       PIC   9   VALUE   0.
 01    CHAR      PIC   X   VALUE  '0'.
*
 PROCEDURE   DIVISION.
       IF  NUM = 0 AND CHAR = '0'
          DISPLAY  'THEY ARE BOTH ZERO'
```

```
       END-IF.
        STOP RUN.
```

以上代码运行后，将有如下输出结果。

```
THEY ARE BOTH ZERO
```

但是在实际开发中，特别是在对大段程序的编码中，常常会将数字 0 和字符“0”弄混淆。此时，若统一用 ZERO 替代两者，将方便得多。下面这段代码就使用 ZERO 替代了上段代码选择结构中条件判断表达式里的 0 和“0”。此时，将不必考虑到底哪个地方该写 0，哪个地方该写“0”。不必再担心程序会因 0 和“0”的不同而出错。改写后的代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE   SECTION.
 01    NUM       PIC   9   VALUE   0.
 01    CHAR      PIC   X   VALUE  '0'.
*
 PROCEDURE   DIVISION.
       IF  NUM = ZERO AND CHAR = ZERO
          DISPLAY  'THEY ARE BOTH ZERO'
       END-IF.
        STOP RUN.
```

7.2.7 使用 88 层条件名简化选择结构编码

88 层条件名主要是为了方便对多重条件进行编码的。88 层条件名在数据部进行定义，层号命名为 88。下面是一段 88 层条件名的定义方式。

```
......
DATA  DIVISION.
WORKING-STORAGE   SECTION.
01    CONDITION       PIC  9.
      88   COND1     VALUE  1.
      88  COND2   VALUE  2.
      88  COND3   VALUE  3.
      88  COND4   VALUE  4.
      88  COND5   VALUE  5.
      88  COND6   VALUE  6.
      88  COND7   VALUE  7.
```

对于以上 88 层条件名的定义方式，有以下 3 点需要注意。

- 88 层条件名是按通常方式定义的变量的下面进行定义的。如上面示例中的 88 层条件名都是在变量 CONDITION 的下面进行定义的。
- 88 层条件名在定义时是不使用 PIC 语句的。
- 88 层条件名在定义时需要使用 VALUE 语句。特别需要注意的是，此处 VALUE 语句后的值并不是对该条件名赋的初值。VALUE 后的值仅表示当在上面对应的变量内容为该值时，该值所对应的条件名为真。

下面给出一个具体实例用以说明 88 层条件名具体是如何使用的。该实例中由用户从键盘上输入一个数字。程序将根据该数字输出相应的星期数。例如用户输入数字 1，则输出星期一的名称，依此类推。该段代码使用了 88 层条件名方式，代码如下。

```
......
DATA  DIVISION.
WORKING-STORAGE   SECTION.
01    DAY-NUM       PIC   9.
      88    MONDAY      VALUE  1.
      88    TUESDAY     VALUE  2.
      88    WEDNESDAY   VALUE  3.
      88    THURSDAY    VALUE  4.
      88    FRIDAY      VALUE  5.
      88    SATURDAY    VALUE  6.
      88    SUNDAY      VALUE  7.
*
 PROCEDURE   DIVISION.
      DISPLAY  'PLS  INPUT  A  DAY  CODE:'.
      ACCEPT  DAY-NUM.
      IF   MONDAY
           DISPLAY  'MONDAY'.
      IF   TUESDAY
           DISPLAY  'TUESDAY'.
      IF   WEDNESDAY
           DISPLAY  'WEDNESDAY'.
      IF   THURSDAY
           DISPLAY  'THURSDAY'.
      IF   FRIDAY
           DISPLAY  'FRIDAY'.
      IF   SATURDAY
           DISPLAY  'SATURDAY'.
      IF   SUNDAY
           DISPLAY  'SUNDAY'.
          STOP RUN.
```

以上代码运行后，若用户输入数字 3，则程序将有如下输出结果。

```
WEDNESDAY
```

当然，对于以上代码也可以使用 EVALUATE 语句完成同样的功能。但如果程序只要求对应周六和周日产生相应输出，则使用 EVALUATE 语句就不太适合了。并且，以上只是为了方便说明而做的示例，在实际中选择结构往往并不在一处出现，而是分散的。对于分散出现的选择结构，则只能使用 IF 语句了。例如，实际中的情况往往是这样的。

```
......
PROCEDURE   DIVISION.
      DISPLAY  'PLS  INPUT  A  DAY  CODE:'.
      ACCEPT  DAY-NUM.
      ......
      IF   SATURDAY
           do something
      ......
      IF   SUNDAY
           do something else
      ......
           STOP RUN.
```

以上代码中的省略号表示省略了其他操作语句。这种情况下，通常还是使用 IF 语句比较好。此时，合理地使用 88 层条件名必将简化编码。下面是对应上面示例中的两条 IF 语句的

条件判断部分。其中第二条使用了 88 层条件名技术，而第一条则没有。对比如下。

```
IF  DAY-NUM = 5 …
IF  FRIDAY…
```

以上两条语句是等价的。但显然，第二条语句要比第一条更加易于阅读，更加直观。这对程序后期的调试和维护工作是十分重要的。

此外，在 88 层条件名中，不仅可以定义一个具体条件数值，也可以定义一个条件范围。如下就定义了所有的工作日和周末的条件名。

```
  88   WEEK-DAY     VALUES  1
                    THRU    5.
  88   WEEK-END     VALUES  6
                    THRU    7.
```

对于工作日而言，下面这两行条件判断是等价的。

```
IF  DAY-NUM  > 0  AND  <  6 …
IF  WEEK-DAY…
```

对于休息日而言，下面这两行条件判断是等价的。

```
IF  DAY-NUM  =  6  OR  7…
IF  WEEK-END…
```

需要注意的是，此时对于条件数值将可能会出现二义性情况。具体而言，就是对于一个条件数值，可能会有两个条件名与之对应。例如，下面为一个直接使用条件数值的条件判断。

```
IF  DAY-NUM  =  2 …
```

对于上面这一条件判断，将有以下两条使用了 88 层条件名的判断与之对应。

```
IF  TUESDAY …
IF  WEEK-DAY …
```

当然，通常是使用 88 层条件名表示条件数值，而不是使用条件数值表示 88 层条件名。因此，此处对程序影响不大。总之，使用 88 层条件名的好处主要有以下这两点。

- 使程序更易于阅读，更加直观。
- 当需要改变判断条件时，只用在数据部进行统一修改，而不用去过程部修改每条相关语句。

7.2.8 选择结构的综合应用

下面以一个图书管理系统模型为实例，说明选择结构在实际中是如何综合应用的。通过该实例讲解，加深对选择结构流程的理解和掌握。该模型的功能如下。

- 首先要求输入系统管理员 ID。若 ID 正确，则继续要求输入密码；若 ID 错误，则输出“INVALID ID”，退出程序。此处不妨令管理员 ID 为“ABCD”。
- 令密码为“1234”。若密码输入正确，则进入系统；若密码输入错误，则输出“WRONG PASSWORD”，退出程序。
- 输入相应功能编号进行相关操作。此处可实现的功能分别为增加图书、删除图书、修改图书、查询图书，以及退出系统。其对应的功能编号依此为：1、2、3、4、0。

- 令该图书馆目前有 4 类图书，分别为 TP、F、G、A。其中原来的馆藏图书每类为 10 本。当增加图书时，要求输入该书的分类号，并将对应类别图书的总量加 1。
- 其余功能尽量简化，每一功能对应 1 条输出信息。

对于以上所要求的功能，可以分成 3 个功能模块来实现。其中第 1 个模块对应登录系统的功能，第 2 个模快对应操作选择的功能，第 3 个模块对应增加图书功能。登录系统功能模块所对应的流程图如图 7.9 所示。

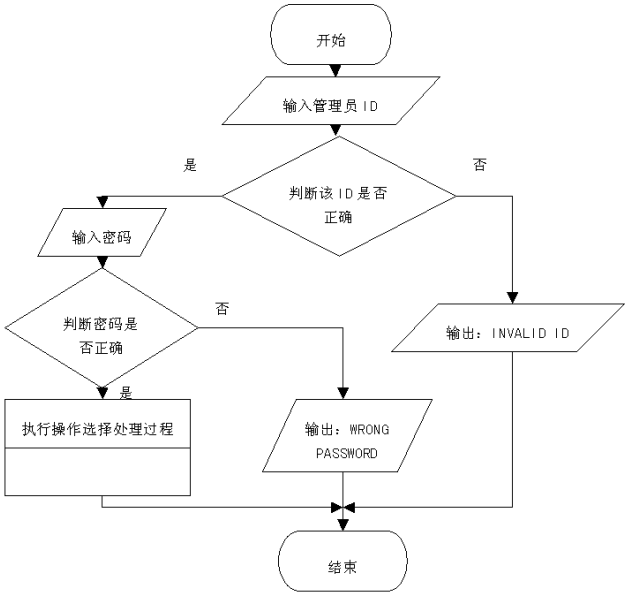

图 7.9　登录系统功能流程图

选择操作功能模块的流程图如图 7.10 所示。

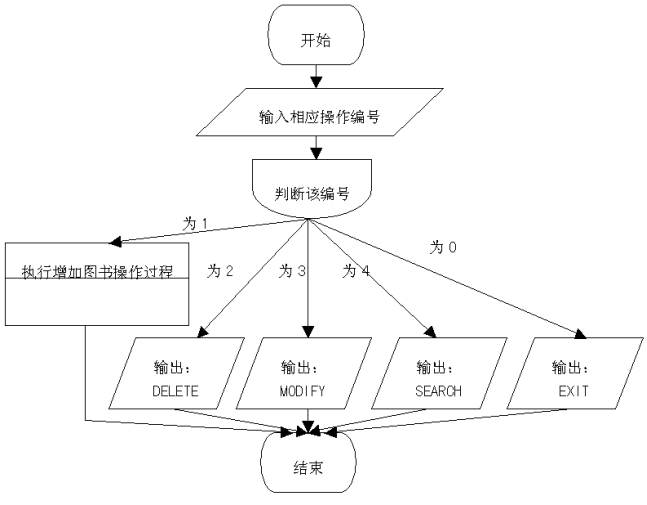

图 7.10　选择操作功能流程图

增加图书功能模块的流程图如图 7.11 所示。

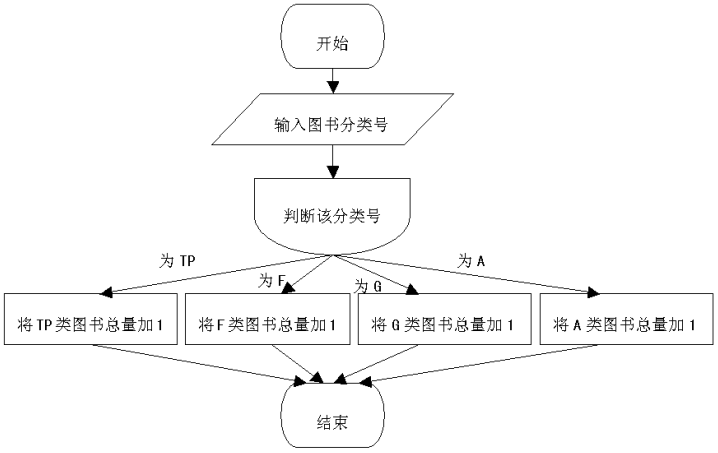

图 7.11　增加图书功能流程图

实现该模型功能的代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE   SECTION.
 01  ADMIN-ID      PIC X(4).
 01  ID-SOURCE     PIC X(4)  VALUE  'ABCD'.
 01  PASS-WORD     PIC X(4).
 01  PW-SOURCE     PIC X(4)  VALUE  '1234'.
 01  FUNCTION-MENU PIC  9.
     88    ADD         VALUE  1.
     88    DELETE      VALUE  2.
     88    MODIFY      VALUE  3.
     88    SEARCH      VALUE  4.
 01  BOOK-TYPE    PIC XX.
 01  BOOK-TYPE-NUM.
     05   TP-NUM   PIC 99 VALUE 10.
     05   F-NUM    PIC 99 VALUE 10.
     05   G-NUM    PIC 99 VALUE 10.
     05   A-NUM    PIC 99 VALUE 10.
*
 PROCEDURE   DIVISION.
      ACCEPT  ADMIN-ID.
      IF ADMIN-ID = ID-SOURCE
          ACCEPT PASS-WORD
            IF PASS-WORD = PW-SOURCE
              PERFORM  100-SELECT-FUN.
            ELSE  DISPLAY ' WRONG PASSWORD'
            END-IF
       ELSE DISPLAY 'INVALID ID'
       END-IF.
        STOP RUN.
 100-SELECT-FUN.
      ACCEPT FUNCTION-MENU.
      IF   ADD  PERFORM 200-ADD-BOOK.
      IF   DELETE   DISPLAY 'DELETE'.
      IF   MODIFY   DISPLAY 'MODIFY'.
```

```
        IF   SEARCH  DISPLAY 'SEARCH'.
        IF   FUNCTION-MENU = ZERO
             DISPLAY  'EXIT'.
    200-ADD-BOOK.
        ACCEPT BOOK-TYPE.
        EVALUATE  TRUE
          WHEN  BOOK-TYPE = 'TP'  ADD 1 TO TP-NUM
          WHEN  BOOK-TYPE = 'F'   ADD 1 TO F-NUM
          WHEN  BOOK-TYPE = 'G'   ADD 1 TO G-NUM
          WHEN  BOOK-TYPE = 'A'   ADD 1 TO A-NUM
        END-EVALUATE.
```

以上代码综合应用了 IF 语句，嵌套 IF 语句，EVALUATE 语句。同时，该段代码还用到了 ZERO 和 88 层条件名简化方法。通过以上示例代码，可以对前面所学的选择结构流程控制做一个整体的回顾。

7.3 循环结构流程控制

本节将介绍流程控制中的最后一个流程结构，即循环结构。在 COBOL 中，循环结构主要是由 PERFORM UNTIL 语句控制的。当循环结构中处理的内容不多时，有时也可以使用线上 PERFORM 语句控制。下面分别对循环结构的基本流程、控制语句以及综合应用进行讲解。

7.3.1 循环结构的基本流程

循环结构是 3 种流程结构中最复杂的一种，不过它的功能也是最强大的。很多典型的算法，如快速排序算法、冒泡排序算法、二分查找算法等，都是基于循环结构的。此外一些需要用到递归和回溯的算法，如二叉数的遍历、背包问题的求解等也是离不开循环结构的。因此，学好循环结构对于程序设计而言是至关重要的。下面首先给出一个典型的循环结构流程图，该流程图如图 7.12 所示。

上面流程图中，初始化处理包括对循环条件进行初始化。初始化完毕，将判断循环条件是否满足。当循环条件满足时，进行循环主体程序处理以及基于循环条件的增量处理。此处所谓的基于循环条件的增量处理主要是为了不断改变循环条件的值，使循环最终能正常退出。当循环条件不满足时，将退出循环，并进行循环结束后的相关处理，最终程序结束。

此处需要注意的是，上面的循环结构中在初始化完毕后，首先进行的是循环条件判断。为规范流程编码，编码时更不易出错，通常是将该条件判断放在循环过程的最前面。下面这一流程图将循环条件判断放在循环过程的最下面，通常是不提倡的。该流程图如图 7.13 所示。

下面通过一个具体的小例子，以便更好地说明循环结构的基本流程在实际中通常是如何应用的。该例子要求计算从 1～100 的所有自然数的总和。此处只要求画出流程图，不要求编写代码实现。则实现此功能对应的流程，如图 7.14 所示。

求连续自然数的总和是使用循环结构进行处理的最典型的问题之一。实际上，对上图略作改变，可以实现其他许多类似功能。例如。

- 当将初始化处理改变时，可以计算从任意小于 100 的自然数开始到 100 的总和。例如可以计算从 3 ~ 100 的总和、从 28 ~ 100 的总和等。
- 当将循环条件改变时，可以计算从 1 开始直到任何数为止的所有自然数总和。例如可以

计算从 1～50 的总和、从 1～1000 的总和等。

- 当将增量处理改变时，可以计算从 1～100 任意间隔的自然数总和。例如可以计算全体奇数的总和、全体偶数的总和、全体为 3 的倍数的自然数的总和等。

当然，也可同时改变以上 3 个条件，此时可以实现以上并行的功能。

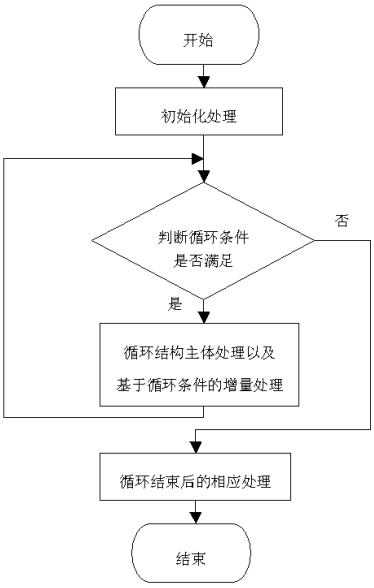

图 7.12　典型循环结构流程图

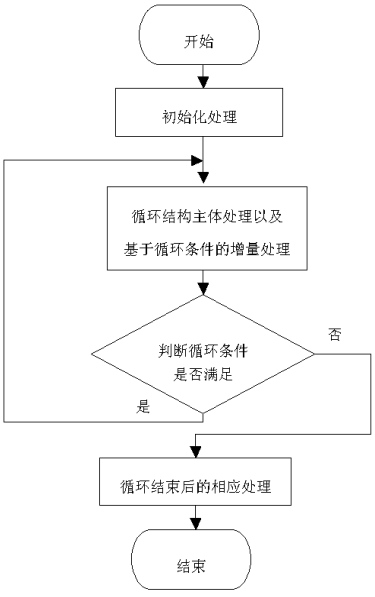

图 7.13　不被提倡的循环结构流程图

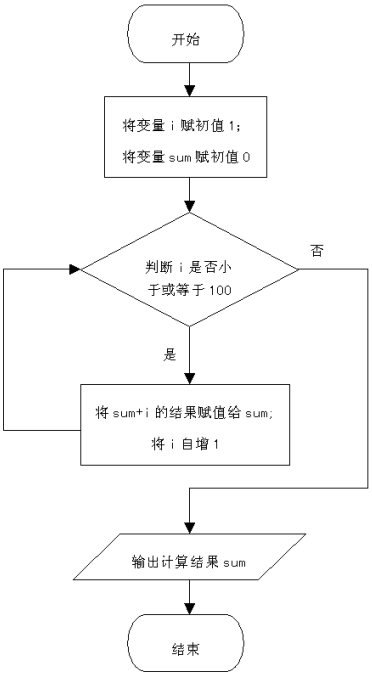

图 7.14　计算 1～100 所有自然数总和的流程图

7.3.2 使用 PERFORM UNTIL 语句控制循环结构流程

PERFORM UNTIL 语句实际上是在 PERFORM 语句后加上 UNTIL 选项形成的。有的书上也将它称为 PERFORM 语句的一个格式。UNTIL 选项作为循环结构的条件判断，起到控制循环次数的作用。因此，通过 PERFORM UNTIL 语句能够控制循环结构的流程。并且，在实际中通常也是用该语句编写循环结构的。对应前面求 1~100 全体自然数总和的流程图，下面可以借助 PERFORM UNTIL 语句实现其功能。

```
......
DATA  DIVISION.
WORKING-STORAGE   SECTION.
01    I           PIC   9(3).
01    SUM        PIC   9(4).
*
PROCEDURE   DIVISION.
     MOVE  1  TO  I.
     MOVE  0  TO  SUM.
     PERFORM   100-ADD-TOTAL
        UNTIL   I > 100.
     DISPLAY 'THE SUM IS: '  SUM.
      STOP RUN.
*
100-ADD-TOTAL.
     ADD  I  TO  SUM.
     ADD  1  TO  I.
```

以上代码运行后，输出结果如下所示。

```
THE SUM IS: 5050
```

由此，便得到了从 1～100 全体自然数总和的正确结果，即 5050。可以看到，通过 PERFORM UNTIL 语句将能方便地实现循环结构流程控制，进而实现相应的功能。

下面，不妨再结合一个例子，加深对使用 PERFORM UNTIL 语句控制循环结构流程的掌握。该例要求计算从 1～9 的自然数中，同时以该数字作为底数和幂的所有乘方运算结果的总和。写成代数式，即要求计算下面这一式子的结果。

$$1^1 + 2^2 + 3^3 + 4^4 + 5^5 + 6^6 + 7^7 + 8^8 + 9^9$$

显然，为实现该功能，必须采用循环结构流程。首先，做出该循环结构的流程图，如图 7.15 所示。

根据以上流程图，完成该功能的代码如下。

```
......
DATA  DIVISION.
WORKING-STORAGE   SECTION.
01    I           PIC   99.
01    J           PIC   99.
01    II          PIC   99.
01    S           PIC  999.
*
PROCEDURE   DIVISION.
```

```
       MOVE  1  TO  I.
       MOVE  0  TO  S.
       PERFORM   100-ADD-DATA
          UNTIL   I > 9.
       DISPLAY 'THE  RESULT  IS: '  S.
          STOP RUN.
*
 100-ADD-DATA.
       MOVE 1  TO  J.
       MOVE 1  TO  II.
       PERFORM   200-MULTIPLY-DATA
          UNTIL   J > I.
       COMPUTE S = S + II
       ADD 1 TO I.
*
 200-MULTIPLY-DATA.
       COMPUTE II = I * II.
       ADD 1 TO J.
```

以上代码运行后，将有如下输出结果。

```
THE  RESULT  IS: 285
```

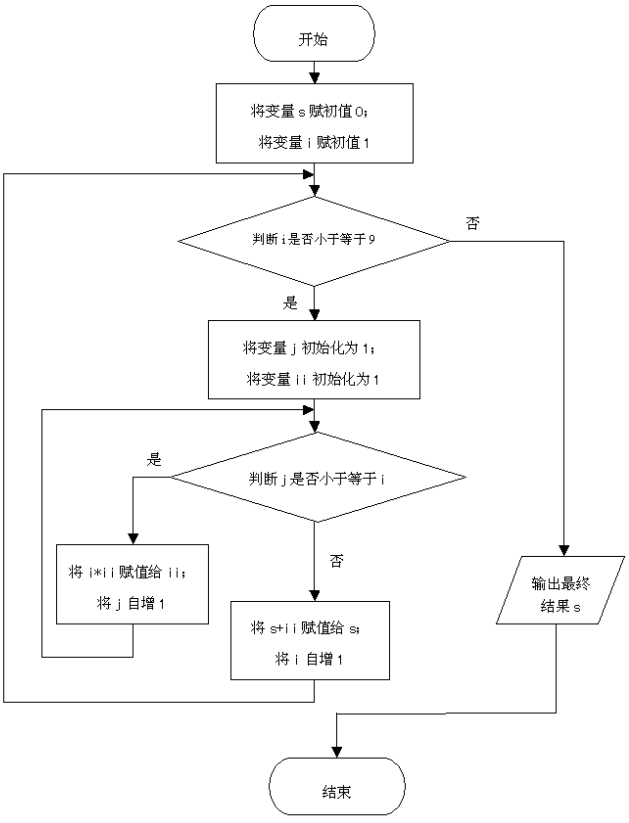

图 7.15　计算连续自然数乘方的总和

该段代码中有两个 PERFORM UNTIL 语句。并且，其中一个 PERFORM UNTIL 语句包含在另一个 PERFORM UNTIL 语句的处理过程之中。实际上，此处便形成了两层相嵌套的循环

结构。根据这两层循环结构对应的条件判断，可以看出这两层循环结构分别是由不同的条件变量控制的。其中外层循环结构是由变量 i 所控制的，而内层循环结构则是由变量 j 所控制的。

总之，在 COBOL 中编写循环结构流程最常使用的便是 PERFORM UNTIL 语句。结合前面的实例，可以知道在循环结构流程中使用 PERFORM UNTIL 语句的基本格式如下。

```
init statement.
PERFORM  process statement
         UNTIL   condition.
         finish statement.
```

对应以上基本格式，此处对使用 PERFORM UNTIL 语句控制循环结构流程的方式总结如下。

- 在使用 PERFORM UNTIL 语句之前，通过 init statement 对循环条件进行初始化处理。
- 通过PERFORM后面的process statement进行循环结构主体处理以及循环条件增量处理。
- 通过 UNTIL 后面的 condition 循环条件控制循环次数。
- PERFORM UNTIL 语句运行结束后，通过 finish statement 进行循环结束后的处理。

最后需要补充的一点是，通过循环条件增量处理，最终必须能使循环条件不满足而退出循环。否则，循环将无限次进行下去，形成死循环。此时将会造成系统错误。

7.3.3 使用线上 PERFORM 语句控制循环结构流程

线上 PERFORM 语句和 PERFORM UNTIL 语句类似，也是主要用于控制循环结构流程的。并且，二者都是 PERFORM 语句的延伸，实际上都属于 PERFORM 语句。

线上 PERFORM 语句与 PERFORM UNTIL 语句最大的区别是该语句将所有的处理都放在语句内了。前面所讲到的 PERFORM UNTIL 语句中，循环结构主体处理以及循环条件增量处理都是通过在 PERFORM 后面指定的。这种方式的 PERFORM 语句也叫线外 PERFORM 语句。而线上 PERFORM 语句，则将循环结构主体处理和循环条件增量处理都放在该 PERFORM 语句内部了。

下面两则代码分别使用线外 PERFORM 语句和线上 PERFORM 语句计算从 10~50 所有奇数的总和。通过比较，可以更好地了解二者的差别。

其中使用线外 PERFORM 语句的代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE   SECTION.
 01    I        PIC   9(3).
 01    SUM      PIC   9(4).
*
 PROCEDURE   DIVISION.
      MOVE  10  TO  I.
      MOVE  0  TO  SUM.
      PERFORM   100-ADD-TOTAL
         UNTIL   I > 50.
      DISPLAY 'THE SUM IS: '  SUM.
       STOP RUN.
*
 100-ADD-TOTAL.
      ADD  I  TO  SUM.
      ADD  2  TO  I.
```

使用线上 PERFORM 语句的代码如下。

```
 ......
 DATA  DIVISION.
 WORKING-STORAGE   SECTION.
 01    I        PIC   9(3).
 01    SUM      PIC   9(4).
*
 PROCEDURE   DIVISION.
      MOVE  10  TO  I.
      MOVE  0  TO  SUM.
      PERFORM
         UNTIL   I > 50
            ADD  I  TO  SUM
            ADD  2  TO  I
      END-PERFORM.
      DISPLAY 'THE SUM IS:'  SUM.
       STOP RUN.
```

二者运行后，结果输出都将如下。

```
THE SUM IS: 625
```

对比以上两段代码，会发现线上 PERFORM 语句和线外 PERFORM 语句的区别主要有以下几点。

- 线上 PERFORM 语句在 PERFORM 标志符后直接是 UNTIL 选项；而线外 PERFORM 语句在 PERFORM 标志符后是处理过程名。
- 线上 PERFORM 语句通过 END-PERFORM 结束标志表示语句结束；而线外 PERFORM 语句通过句点表示语句结束。
- 线上 PERFORM 语句在 UNTIL 选项所列的循环条件之后，与语句结束之前有几条其他处理语句；而线外 PERFORM 语句在 UNTIL 选项所列循环条件后，直接结束语句。

由此可见，虽然二者都用于循环结构流程控制，但各自的表达形式是不同的。此外，关于线上 PERFORM 语句，还有以下几点需要注意。

- 线上 PERFORM 语句主要用在其循环结构主体处理部分较简单的情况下，该语句所包含的循环结构主体处理语句，以及循环条件增量处理语句通常总共不应超过 6 条。
- END-PERFORM 结束标志仅用于线上 PERFORM 语句中，而不用于线外 PERFORM 语句。
- 线上 PERFORM 语句中若出现句点，该句点同样可以结束线上 PERFORM 语句。因此，提倡使用 END-PERFORM 来结束线上 PERFORM 语句，否则容易出错。
- 线上 PERFORM 语句和 PERFORM UNTIL 语句是两个并行的概念。实际上，本小节介绍的线上 PERFORM 语句也同时都为 PERFORM UNTIL 语句。
- 线上 PERFORM 语句属于较特殊的 PERFORM 语句。若不特别强调，通常所说的 PERFORM 语句都是指线外 PERFORM 语句。
- 线上 PERFORM 语句是在 COBOL-85 版本中所新推出的。

最后，需要再次强调的是，线上 PERFORM 语句仅用于循环结构主体处理较简单的情况。此时将处理语句全部置于该 PERFORM 语句内，而不另写处理过程，将简化编码和结构。实际上，这也正是线上 PERFORM 语句最主要的用途。当开发大型程序，循环主体处理较复杂

时，还是应该使用通常的 PERFORM 语句。

7.3.4 循环结构的综合应用

同选择结构流程控制一节类似，在本节最后，仍然结合一个实例来说明循环结构的综合应用。首先，来看下面一个由星号组成的三角形。

```
                *                第 1 层
              *   *              第 2 层
            *   *   *            第 3 层
          *   *   *   *          第 4 层
        *   *   *   *   *        第 5 层
            ......               ......
......                           第 n 层
```

根据该结构，三角形的层数与其所包含的星号数对应关系如下。

- 若三角形只有 1 层，包含的星号数为 1 个。
- 若三角形有 2 层，包含的星号数为 1+2=3 个。
- 若三角形有 3 层，包含的星号数为 1+2+3=6 个。
- ……
- 若三角形有 n 层，包含的星号数为 1+2+3+……+（n−1）+n 个。

本实例以该三角形结构为基础，要求分别输出层数从 1 直到 10 的每一个三角形面积的大小。其中面积为其所包含的星号个数。实现该功能的流程如图 7.16 所示。

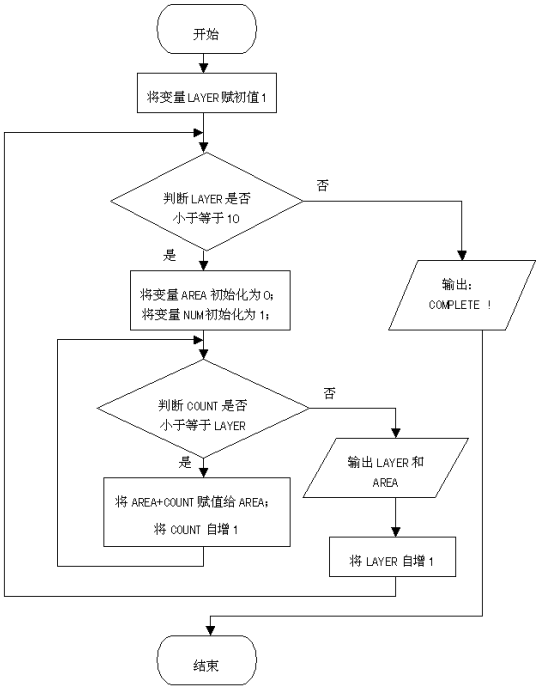

图 7.16　计算每层三角形面积流程图

根据以上流程图，完成该功能的代码如下。

```
......
DATA  DIVISION.
WORKING-STORAGE   SECTION.
01    LAYER          PIC   99.
01    AREA           PIC   99.
01    COUNT          PIC   99.
*
PROCEDURE   DIVISION.
     MOVE  1  TO  LAYER.
     PERFORM   100-OUTER-LOOP
        UNTIL   LAYER > 10.
     DISPLAY 'COMPLETE ! ' .
         STOP RUN.
*
100-OUTER-LOOP.
     MOVE 0  TO  AREA.
     MOVE 1  TO  COUNT.
     PERFORM
        UNTIL   COUNT > LAYER
     COMPUTE AREA = AREA + COUNT
     ADD 1 TO COUNT
     END-PERFORM.
     DISPLAY 'LAYER:'  LAYER     'AREA:' AREA.
     ADD 1 TO LAYER.
```

以上代码运行后，将有如下输出结果。

```
LAYER: 1     AREA: 1
LAYER: 2     AREA: 3
LAYER: 3     AREA: 6
LAYER: 4     AREA: 10
LAYER: 5     AREA: 15
LAYER: 6     AREA: 21
LAYER: 7     AREA: 28
LAYER: 8     AREA: 36
LAYER: 9     AREA: 45
LAYER: 10    AREA: 55
COMPLETE !
```

以上结果列举出了不同层数的三角形分别所对应的面积。其中此处共有 10 个三角形，其层数依次从 1 层到 10 层。该代码通过循环结构流程，完成了实例的要求。

该代码中的循环结构实际上主要是由 PERFORM UNTIL 语句和线上 PERFORM 语句所完成的。同时，以上循环结构仍然是一个嵌套的循环结构。由于嵌套的循环结构是基于循环结构所产生的，其嵌套过程同选择结构中的嵌套类似。本节不将嵌套循环结构单独作为一个小节来讲解，而是放在循环结构中一并讲解。

大部分算法都是建立在循环结构流程的基础之上的，学好循环结构流程对于今后的实际开发是十分重要的。

7.4 本章回顾

本章主要介绍了 COBOL 程序中的 3 大典型流程。分别为顺序结构流程、选择结构流程以及循环结构流程。

- 顺序结构流程是指程序顺序执行其中的每条语句。该流程不需要用到特别的流程控制语句。作为 3 种流程中最基本的流程，可以说几乎任何程序都用到了该流程。
- 选择结构流程是指含有分支结构的流程。在选择结构流程中，根据条件判断的结果选择相应的语句执行，并不执行所有的语句。选择结构流程通常可以通过 IF 语句、嵌套 IF 语句以及 EVALUATE 语句控制。此外，通过使用 ZERO 以及 88 层条件名可以简化选择结构的编码。
- 循环结构流程是 3 种流程中最复杂的流程。在程序中，最常见的是使用 PERFORM UNTIL 语句控制循环结构流程。同时，当循环主体处理较简单时，也可使用线上 PERFORM 语句。该语句和 PERFORM UNTIL 语句的概念是并行的。

实际上，任何程序设计语言都离不开以上这 3 种流程。程序设计的核心在于算法，算法是独立于语言而存在的，而流程则是算法的根源。因此，本章所讲解的流程控制将贯穿于程序设计的始终。

第 8 章 数据的排序与合并

本章主要介绍 COBOL 中数据的排序和合并功能。在数据的排序一节中将首先介绍如何定义排序中间文件以及指定用于排序的输入输出文件。之后，重点讲解如何使用 SORT 语句进行排序。同时，在介绍完使用 SORT 语句排序后，还将介绍如何编写排序的输入输出过程。最后，介绍使用包含有输入输出过程的 SORT 语句进行排序。

对于数据的合并，本章将首先介绍如何指定用于排序的输入输出文件。之后，介绍如何编写合并的输出过程。最后，重点介绍如何使用 MERGE 语句进行合并。

8.1 排序与合并概述

首先需要说明的一点是，本章讲解的排序与合并是 COBOL 中所自带的功能。二者在此处并非指通常所说的排序与合并的算法。同时，这里所说的排序与合并都是针对文件中的数据而言的。因此，有的书上也将其称做文件的排序与合并。

8.1.1 排序的基本概念

所谓数据的排序，就是将文件中的数据按一定的顺序进行重新排列。COBOL 中的文件数据实际上是由记录所组成的。因此，对数据的排序实际上通常就是对记录的排序。一条记录中往往含有多个数据项，将记录排序时，必须以一个或多个数据项为基准。此处的数据项，也就是排序中所要用到的关键字。

例如，下面为一记录员工信息的文件。文件中每条记录对应一个员工信息。其中每个员工信息包括该员工的工号、姓名、住址、邮编。该文件中的数据如下。

```
1235    ZHU BO              WU HAN        430074
1024    WU GUO JUN          BEI JING      100085
1346    ZHANG XIANG         SHEN ZHEN     518057
1005    XIE FEI             WU HAN        430023
1253    XIE FEI             WU HAN        430022
```

此时，如果以员工工号作为关键字，并按照升序进行排序，则排序后的数据如下。

```
1005    XIE FEI                 WU HAN        430023
1024    WU GUO JUN              BEI JING      100085
1235    ZHU BO                  WU HAN        430074
1253    XIE FEI                 WU HAN        430022
1346    ZHANG XIANG             SHEN ZHEN     518057
```

如果按照员工工号的降序排列，则排序后的数据如下。

```
1346    ZHANG XIANG             SHEN ZHEN     518057
1253    XIE FEI                 WU HAN        430022
1235    ZHU BO                  WU HAN        430074
1024    WU GUO JUN              BEI JING      100085
1005    XIE FEI                 WU HAN        430023
```

此外，还可以按照员工的姓名排序。由于存在重名现象，因此可以将工号作为第二关键字。若将员工姓名作为第一关键字按升序排列，将工号作为第二关键字按降序排列，排序结果将如下。

```
1024    WU GUO JUN              BEI JING      100085
1253    XIE FEI                 WU HAN        430022
1005    XIE FEI                 WU HAN        430023
1346    ZHANG XIANG             SHEN ZHEN     518057
1235    ZHU BO                  WU HAN        430074
```

由此可见，排序的结果主要是由两方面因素所决定的。一方面因素是排序的方式。排序方式分为升序和降序两种。排序方式的不同将导致完全相反的排序结果。另一方面因素是排序关键字的选取。关键字选取的不同通常也将直接导致排序结果的不同。

以上谈到了排序的基本概念。对于在 COBOL 中的实际开发，通常情况下，使用排序功能时主要需要用到以下 3 种文件。

- 排序输入文件。该文件即排序操作的对象，其中包含用来进行排序的原始无序数据。
- 排序中间文件。该文件是一个临时工作文件，仅用于排序操作。该文件的功能是从排序输入文件得到原始数据，并将排序后的结果数据输出到排序输出文件中。该文件在此处相当于一个临时缓冲区。
- 排序输出文件。该文件保存排序后的结果，其中包含排序完成后的有序数据。

关于以上 3 种文件的使用方式以及排序功能的具体实现，将在后面的小节中进行详细讲解。此处只需要了解 COBOL 中排序的基本概念。

8.1.2 合并的基本概念

合并通常是指将两个及两个以上的文件中的数据合并到一个新的文件中。并且，用于合并的文件通常应该是已经排序过的文件。合并后的文件中的数据也是有序数据。实际上，合并也是基于相关数据的顺序而进行的操作。

例如，下面为一个用于合并的记录大型机从业人员信息的文件。令文件中每条记录包括人员编号、姓名、公司、入职时间这几个数据项。假设该文件中的具体数据如下。

```
001000   WANG WEI      IBM CDL       2003/06/01
001003   YU ZHU        EDS           2007/07/05
001008   ZHENG JI      CSTS          2005/09/08
001010   LI QIANG      IBM CSDL      2007/06/08
001025   JIANG TAO     IBM ISSC      2006/11/20
```

另一个用于合并的文件中的数据如下。

```
001004   WANG WEI      IBM TSS       2002/06/20
001007   YU ZHU        FDI           2004/07/12
001012   ZHENG JI      ISTC          2007/09/15
```

将以上两个文件按关键字为人员编号进行升序合并后，将生成一个新的合并后的文件。该合并后的文件中的数据如下。

```
001000   WANG WEI      IBM CDL       2003/06/01
001003   YU ZHU        EDS           2007/07/05
001004   WANG WEI      IBM TSS       2002/06/20
001007   YU ZHU        FDI           2004/07/12
001008   ZHENG JI      CSTS          2005/09/08
001010   LI QIANG      IBM CSDL      2007/06/08
001012   ZHENG JI      ISTC          2007/09/15
001025   JIANG TAO     IBM ISSC      2006/11/20
```

同样，类似于排序，合并在实际应用中也是需要 3 种文件的。这 3 种文件分别为合并输入文件，合并中间文件，以及合并输出文件。以上 3 种文件的概念和排序中 3 种文件的概念基本类似，只是此处是用于合并功能的。

最后需要补充的一点是，无论是排序还是合并，其结果并不是一定要保存在一个文件中的。排序或合并的结果既可以保存在多个文件中，又可以不保存在任何文件中，直接提供给程序进行处理。

8.2 数据的排序

前面已经讲到，数据的排序实际上是对文件中的数据进行排序。并且，用于排序的文件有 3 种，分别为排序中间文件，排序输入文件，以及排序输出文件。本节将首先讲解如何实际应用以上 3 种文件，然后重点讲解在程序中是如何实际进行排序的，在讲解完如何进行排序后，还将介绍通常是如何编写排序输入输出过程的，最后还将讲解包含有输入输出过程的数据排序操作。

8.2.1 使用 SD 语句定义排序中间文件

排序中间文件是一个临时工作文件，因此和普通的文件是不同的。普通文件是通过在数据部的文件节中通过 FD 语句定义的。而排序中间文件则是通过 SD 语句进行定义的。其定义位置则仍然是在数据部的文件节中。

此处，令在前面排序的基本概念一节中的员工信息文件为排序输入文件。该文件内容如下。

```
1235     ZHU BO               WU HAN        430074
1024     WU GUO JUN           BEI JING      100085
1346     ZHANG XIANG          SHEN ZHEN     518057
1005     XIE FEI              WU HAN        430023
1253     XIE FEI              WU HAN        430022
```

假设对应以上内容需要定义的排序中间文件的文件名为“TEST-SORT-FILE”。同任何文件一样，在使用 SD 语句定义该文件之前，首先需要在环境部进行指定。指定方式如下。

```
……
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
    SELECT  TEST-SORT-FILE
                ASSIGN  TO  S-SORT.
……
```

以上代码中的“S-SORT”为系统中所指定的一个文件名。而“TEST-SORT-FILE”则是本 COBOL 程序编写中所用到的文件名。两文件名对应同一个文件。此外，为体现程序结构的完整性，该段代码中分别使用两个省略号表示程序中的其他部分。其中前一个省略号表示标志部中的内容，后一个省略号表示环境部其他内容及后面数据部中的内容。

在环境部指定该文件后，便可使用 SD 语句在数据部的文件节中进行相应定义了。使用 SD 语句定义该文件的方式如下。

```
……
 DATA DIVISION.
 FILE SECTION.
 SD  TEST-SORT-FILE.
     RECORD  CONTAINS 80 CHARACTERS
     DATA   RECORD  IS  SORT-RECORD.
 01   SORT-RECORD.
       05  EMP-CODE    PIC  9(4).
       05  FILLER       PIC  X(4)
                         VALUE SPACES.
       05  EMP-NAME    PIC  X(17).
       05  FILLER       PIC  XXX
                         VALUE SPACES.
       05  EMP-ADDR    PIC  X(12).
       05  FILLER       PIC  X(4)
                         VALUE SPACES.
       05  ADDR-CODE   PIC  9(6).
       05  FILLER       PIC  X(30)
                         VALUE SPACES.
……
```

同样，为体现程序结构的完整性，该段代码中也使用省略号表示其上下文内容。其中前一个省略号表示环境部中的内容，后一个省略号表示数据部其他内容及过程部中的内容。

以上便完成了排序中间文件的定义。关于所定义的排序中间文件，还有以下两点需要注意。

- 排序中间文件在 SD 语句定义之后必须包含有一条记录。例如，上面例子中的记录就为“SORT-RECORD”。
- 排序中间文件仅用于排序操作。也就是说，对于该文件而言，不能如通常文件一样对其进行输入输出操作。

8.2.2 使用 USING 短语指定排序输入文件

排序输入文件实际上就是通常的文件，只是此处用于对该文件的数据进行排序而已。即排序输入文件为排序功能提供了原始数据。仍然使用前面员工信息文件的例子，该文件作为排序输入文件，在程序数据部中的定义应该如下。

```
……
DATA DIVISION.
FILE SECTION.
FD  TEST-INPUT-FILE.
01   INPUT-RECORD.
      05  EMP-CODE     PIC  9(4).
      05  FILLER       PIC  X(4)
                          VALUE SPACES.
      05  EMP-NAME     PIC  X(17).
      05  FILLER       PIC  XXX
                          VALUE SPACES.
      05  EMP-ADDR     PIC  X(12).
      05  FILLER       PIC  X(4)
                          VALUE SPACES.
      05  ADDR-CODE    PIC  9(6).
      05  FILLER       PIC  X(30)
                          VALUE SPACES.
……
```

对于以上定义的排序输入文件，此处重点需要说明其在排序操作中是如何应用的。实际上，该文件是通过在用于排序功能的 SORT 语句里的 USING 短语所指定的。指定位置在程序过程部的具体编码之中。相关代码如下。

```
……
PROCEDURE DIVISION.
     ……
     SORT  TEST-SORT-FILE
        ON   ASCENDING  KEY  EMP-CODE  OF SORT-RECORD
        USING   TEST-INPUT-FILE /*此处指定排序输入文件*/
        GIVING   TEST-OUTPUT-FILE.
     ……
```

关于 USING 短语，在排序功能中的内部处理过程如下。

（1）打开与其后文件名相对应的排序输入文件。

（2）读取该排序输入文件的每一条记录。

（3）将读取的每一条记录释放到 SORT 排序之中去。

（4）关闭排序输入文件。

8.2.3 使用 GIVING 短语指定排序输出文件

排序输出文件所定义的内部组织形式也必须和前两种文件一致，否则将无法将记录内容正确输出。因此，对应于以上定义的排序中间文件和排序输入文件，此处的排序输出文件应定义如下。

```
……
DATA DIVISION.
FILE SECTION.
FD  TEST-OUTPUT-FILE.
01   OUTPUT-RECORD.
      05  EMP-CODE     PIC  9(4).
      05  FILLER       PIC  X(4)
                          VALUE SPACES.
      05  EMP-NAME     PIC  X(17).
      05  FILLER       PIC  XXX
```

```
                          VALUE SPACES.
       05  EMP-ADDR     PIC  X(12).
       05  FILLER       PIC  X(4)
                          VALUE SPACES.
       05  ADDR-CODE    PIC  9(6).
       05  FILLER       PIC  X(30)
                          VALUE SPACES.
......
```

在实际排序中，以上所定义的排序输出文件是通过 SORT 语句里的 GIVING 短语指定的。对照前面使用 USING 短语指定的排序输出文件，该文件在排序中的应用代码如下。

```
......
PROCEDURE DIVISION.
     ......
     SORT  TEST-SORT-FILE
        ON   ASCENDING  KEY  EMP-CODE  OF SORT-RECORD
        USING   TEST-INPUT-FILE
        GIVING   TEST-OUTPUT-FILE.       /*此处指定排序输出文件*/
     ......
```

同样，GIVING 短语在排序功能中的内部处理过程如下。

（1）打开与其后文件名相对应的排序输出文件。

（2）将排序中间文件里的每条记录顺次写入到该排序输出文件之中。

（3）关闭排序输出文件。

8.2.4 使用 SORT 语句进行排序

在 COBOL 中对于数据的排序功能主要是通过 SORT 语句实现的。使用 SORT 语句进行数据排序时，其排序的内部算法对于程序员是透明的。即此时程序员只用指定排序的输入输出文件以及排序中间文件，并指定排序的关键字和排序方式。具体如何实现排序，则由系统自动完成。

SORT 语句的基本格式如下。

```
SORT  sort-file-name
      ON   ASCENDING  KEY  sort-filed
      USING    input-file-name
      GIVING   output-file-name.
```

该语句实际上在上两小节中也出现过。此处关键需要注意该语句是如何指定排序关键字及排序方式的。排序关键字及排序方式的指定方法如下。

- 通过“ON ASCENDING KEY”选项指定此处的排序方式为升序。若要指定排序方式为降序，则通过“ON DESCENDING KEY”指定。
- 通过“sort-filed”指定排序关键字。其中该关键字为排序中间文件里所定义的一个数据项。当用到多关键字排序时，此处相应有多个关键字名称。

下面给出完成一个文件数据排序的完整程序，以便更好地说明排序功能是如何具体实现的。该程序代码如下。

```
IDENTIFICATION  DIVISION.
PROGRAM-ID.  SORT-TEST-PROG.
AUTHER.      XXX.
```

```
*
 ENVIRONMENT  DIVISION.
 INPUT-OUTPUT  SECTION.
 FILE-CONTROL.
   SELECT  TEST-INPUT-FILE
           ASSIGN TO S-SYSIN.
   SELECT  TEST-OUTPUT-FILE
           ASSIGN TO S-SYSOUT.
   SELECT  TEST-SORT-FILE
           ASSIGN TO S-SORT.
*
 DATA  DIVISION.
 FILE SECTION.
 SD  TEST-SORT-FILE.
     RECORD  CONTAINS 80 CHARACTERS
     DATA   RECORD  IS  SORT-RECORD.
 01   SORT-RECORD.
       05  EMP-CODE     PIC  9(4).
       05  FILLER       PIC  X(4)   VALUE SPACES.
       05  EMP-NAME     PIC  X(17).
       05  FILLER       PIC  XXX   VALUE SPACES.
       05  EMP-ADDR     PIC  X(12).
       05  FILLER       PIC  X(4)   VALUE SPACES.
       05  ADDR-CODE    PIC  9(6).
       05  FILLER       PIC  X(30)  VALUE SPACES.
*
 FD   TEST-INPUT-FILE.
 01   INPUT-RECORD.
       05  EMP-CODE     PIC  9(4).
       05  FILLER       PIC  X(4)   VALUE SPACES.
       05  EMP-NAME     PIC  X(17).
       05  FILLER       PIC  XXX   VALUE SPACES.
       05  EMP-ADDR     PIC  X(12).
       05  FILLER       PIC  X(4)   VALUE SPACES.
       05  ADDR-CODE    PIC  9(6).
       05  FILLER       PIC  X(30)  VALUE SPACES.
*
 FD   TEST-OUTPUT-FILE.
 01   OUTPUT-RECORD.
       05  EMP-CODE     PIC  9(4).
       05  FILLER       PIC  X(4)   VALUE SPACES.
       05  EMP-NAME     PIC  X(17).
       05  FILLER       PIC  XXX   VALUE SPACES.
       05  EMP-ADDR     PIC  X(12).
       05  FILLER       PIC  X(4)   VALUE SPACES.
       05  ADDR-CODE    PIC  9(6).
       05  FILLER       PIC  X(30)  VALUE SPACES.
*
 PROCEDURE   DIVISION.
     SORT  TEST-SORT-FILE
           ON   ASCENDING  KEY  EMP-NAME  OF SORT-RECORD
                DESCENDING KEY  EMP-CODE  OF SORT-RECORD
           USING   TEST-INPUT-FILE
           GIVING   TEST-OUTPUT-FILE.
     STOP  RUN.
```

该程序完整地实现了数据排序的功能。程序中所定义的排序输入文件为 TEST-INPUT-FILE，排序输出文件为 TEST-OUTPUT-FILE。排序中间文件为通过 SD 语句定义的 TEST-SORT-FILE。

假设排序输入文件 TEST-INPUT-FILE 中的数据内容如下。

```
1235   ZHU BO              WU HAN        430074
1024   WU GUO JUN          BEI JING      100085
1346   ZHANG XIANG         SHEN ZHEN     518057
1005   XIE FEI             WU HAN        430023
1253   XIE FEI             WU HAN        430022
```

该程序执行后，排序输出文件 TEST-OUTPUT-FILE 中将有如下数据。

```
1024   WU GUO JUN          BEI JING      100085
1253   XIE FEI             WU HAN        430022
1005   XIE FEI             WU HAN        430023
1346   ZHANG XIANG         SHEN ZHEN     518057
1235   ZHU BO              WU HAN        430074
```

实际上，该程序用于排序的核心代码就是过程部中的 SORT 语句，语句如下。

```
SORT  TEST-SORT-FILE
     ON   ASCENDING  KEY  EMP-NAME  OF SORT-RECORD
          DESCENDING KEY  EMP-CODE  OF SORT-RECORD
     USING   TEST-INPUT-FILE
     GIVING   TEST-OUTPUT-FILE.
```

该语句对应的功能分别如下。

- 指定排序输入文件为 TEST-INPUT-FILE。
- 指定排序输出文件为 TEST-OUTPUT-FILE。
- 指定排序中间文件为 TEST-SORT-FILE。
- 指定排序中的第一关键字为 SORT-RECORD 记录中的 EMP-NAME 数据项。
- 指定第一关键字的排序方式为升序。
- 指定排序中的第二关键字为 SORT-RECORD 记录中的 EMP-CODE 数据项。
- 指定第二关键字的排序方式为降序。

根据其现实意义，该程序的处理过程如下。

（1）指定员工信息文件 TEST-INPUT-FILE 中的数据为原始数据。

（2）以原始数据记录中的员工姓名作为第一关键字，并按照升序排序。

（3）以原始数据记录中的员工工号作为第二关键字，并按照降序排序。

（4）将排序后的信息输出保存到 TEST-OUT-FILE 中。

该程序中的排序输入文件里的内容和 8.1.1 小节排序基本概念中所用到的文件一致。并且，本程序中指定的排序关键字及其排序方式也是和该小节中的第 3 个排序例子是等同的。因此，本程序最终得到的排序结果也和该小节中的第 3 个排序例子所得到的结果是一致的。

以上结合具体实例详细说明了在 COBOL 中是如何进行文件数据的排序的。可以看到，在实际排序中对于文件的定义以及 SORT 语句的使用是最重要的。

此外还需补充的一点是，当排序中包含多个关键字，并且每一关键字的排序方式相同时，也可简化编码。此时不必写明每一关键字所对应的排序方式，只指明第一个关键字的排序方

式便可。例如以下代码将员工姓名、员工住址、员工工号分别作为第一、二、三关键字。并且，所有关键字的排序方式都为升序。该段代码如下。

```
 SORT  TEST-SORT-FILE
         ON   ASCENDING  KEY  EMP-NAME  OF SORT-RECORD
                              EMP-ADDR  OF SORT-RECORD
                              EMP-CODE  OF SORT-RECORD
         USING   TEST-INPUT-FILE
         GIVING   TEST-OUTPUT-FILE.
```

最后，关于数据的排序，还有以下几点需要注意。

- 在 OS/390 和 VM 平台上使用 COBOL 进行排序时，必须具有 IBM 授予的 DFSORT 或相关权限。
- 数据排序中用到的排序输入文件，排序输出文件以及排序中间文件所定义的组织形式必须一致。
- 在完成排序之后，可以使用包含有序数据的排序输出文件进行打印报表或者数据合并等操作。此时，必须重新打开排序输出文件，并读取其中的数据以进行处理。
- 除使用 COBOL 中的 SORT 语句对文件数据进行排序外，也可以使用 JCL 中的实用程序进行。关于 JCL 将在本书的 COBOL 扩展篇中详细讲解。

8.2.5 编写排序中的输入处理过程

排序中的输入处理过程是用于在排序之前对其输入的原始数据进行处理的。通常来说，输入处理过程主要有以下 3 点用途。

- 在正式排序之前，对从排序输入文件中读入的原始数据进行筛选或者处理。例如，若读入的一组数据为包含正负数在内的整型数据，可以令其只对其中的正数进行排序。或者，对所有数据进行乘方运算后再将其进行排序。
- 将程序中其他地方读入的数据作为排序的输入数据。
- 将工作存储节中的数据作为排序的输入数据。

在实际程序中，输入处理过程是通过 SORT 语句中的短语 INPUT PROCEDURE 表示的。例如，下面这段代码就使用输入处理过程将工作存储节中的数据作为排序的输入数据。代码如下。

```
 DATA  DIVISION.
 FILE  SECTION.
 SD  SORT-PROC-FILE.
 01   SORT-PROC-REC.
     05   S-KEY     PIC X(5).
     05   S-DATA    PIC X(75).
 FD   OUT-FILE.
 01   OUT-REC.
     05   OUT-KEY   PIC X(5).
     05   OUT-DATA  PIC X(75).
*
 WORKING-STORAGE SECTION.
 01   WS-REC.
     05   WS-KEY    PIC X(5).
     05   WS-DATA   PIC X(75).
```

```
*
 PROCEDURE  DIVISION.
      SORT  SORT-PROC-FILE
         ON  ASCENDING  KEY  S-KEY
         INPUT  PROCEDURE  SORT-INPUT-PROC     /*此处指明输入处理过程*/
         GIVING  OUT-FILE.
       STOP RUN.
*
 SORT-INPUT-PROC.
      PERFORM   UNTIL  WS-KEY = '00000'
        ACCEPT  WS-REC
        RELEASE  SORT-PROC-REC   FROM  WS-REC
      END-PERFORM.
```

以上代码通过将工作存储节中的数据作为排序的输入数据进行排序。关于此处的输入处理过程，有以下两点需要注意。

- 由于此处已指定了工作存储节中的数据为排序的输入数据，因此不需再另行指定排序输入文件。
- 在此处的输入处理过程中，关键需要使用 RELEASE FROM 语句将输入数据释放到排序过程中。

8.2.6 编写排序中的输出处理过程

与输入处理过程对应，排序中的输出处理过程主要是用于对排序后的结果数据进行处理的。对结果数据的处理通常有以下 3 种方式。

- 选择部分结果数据写入到排序输出文件中。例如，对一组字符串数据进行排序后，可以只将其中首字母为 A 的数据写入到排序输出文件中。
- 对排序后的结果数据进行处理后再将其写入到排序输出文件中。例如，对于上面字符串数据的排序，可将结果数据中所有字母 A 转换为 B 后再进行写入。
- 将排序后的结果数据输出到其他位置。例如，可以将其输出到显示屏上。

在实际程序中，输出处理过程是通过 SORT 语句中的短语 OUTPUT PROCEDURE 表示的。例如，下面这段代码就使用输出处理过程将排序后的结果数据在显示屏上进行输出。代码如下。

```
 DATA  DIVISION.
 FILE  SECTION.
 SD  SORT-PROC-FILE.
 01   SORT-PROC-REC.
     05   S-KEY     PIC X(5).
     05   S-DATA    PIC X(75).
 FD   IN-FILE.
 01   IN-REC.
     05   IN-KEY   PIC X(5).
     05   IN-DATA  PIC X(75).
*
 WORKING-STORAGE SECTION.
 01   WS-REC.
     05   WS-KEY    PIC X(5).
     05   WS-DATA   PIC X(75).
 01   END-OF-RECORDS   PIC X  VALUE  'N'.
```

```
*
 PROCEDURE  DIVISION.
      SORT  SORT-PROC-FILE
          ON  ASCENDING  KEY  S-KEY
          USING IN-FILE
          OUTPUT  PROCEDURE  IS   SORT-OUTPUT-PROC.      /*此处指明输出处理过程*/
       STOP RUN.
*
 SORT-OUTPUT-PROC.
     RETURN  SORT-PROC-FILE
       AT END
         SET  END-OF-RECORDS  TO  TRUE.
     PERFORM  UNTIL  END-OF-RECORDS
           DISPLAY  'SORTED-REC :'  SORT-PROC-REC
           RETURN  SORT-PROC-FILE
            AT  END
              SET  END-OF-RECORDS  TO  TRUE
      END-PERFORM.
```

关于排序中的输出处理过程，还有以下几点需要注意。

- 在排序输出处理过程中，至少需要包含一条 RETURN 语句。RETURN 语句使每一条排序后的记录能够被输出处理过程所使用。RETURN 语句作用于排序中间文件，就好比 READ 语句作用于输入文件。
- 可以使用 RETURN INTO 语句替代 RETURN 语句。当使用 RETURN INTO 语句时，记录将返回到工作存储节或一块用于输出的区域。
- 在 RETURN 语句中，可以使用 AT END 或者 END-RETURN 短语指明返回结束后的操作。其中 AT END 短语表示在所有记录 RETURN 完毕后执行其后的操作。END-RETURN 短语则相当于一个强制终止符，此处直接结束 RETURN，并执行其后的操作。

8.2.7 包含有输入输出处理过程的 SORT 语句排序

前面分别介绍了排序中的输入输出处理是如何编写的。此处将把这两种处理过程综合应用到使用 SORT 语句进行的排序之中。通过综合应用，加深对以上两种处理过程的掌握，同时巩固并扩充应用数据排序的能力。

下面结合一个具体实例进行讲解。仍然以此前的员工信息文件为例，此处在原由的基础上，新增一个员工状态的数据项。假设新的员工信息文件内容如下。

```
1235   ZHU BO              WU HAN        430074     A
1024   WU GUO JUN          BEI JING      100085     A
1346   ZHANG XIANG         SHEN ZHEN     518057     N
1005   XIE FEI             WU HAN        430023     A
1253   XIE FEI             WU HAN        430022     A
1028   ZHENG GANG          BEI JING      100085     N
1204   ZOU LI QIANG        SHANG HAI     201620     A
1536   LING CHEN           BEI JING      100080     N
1791   TIAN BO             SHANG HAI     201821     N
```

本例要求仅对员工状态信息为“A”的员工记录数据进行排序。同时，在排序完成后，要求将所有的员工姓名转换为用小写字母表示，最后再写入排序输出文件中。排序要求以员工姓名作为第一关键字，员工工号作为第二关键字，全部用升序排列。实现以上功能的完整

程序代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  ADVANCE-SORT-PROG.
 AUTHER.      XXX.
*
 ENVIRONMENT  DIVISION.
 INPUT-OUTPUT  SECTION.
 FILE-CONTROL.
   SELECT  TEST-INPUT-FILE
           ASSIGN TO S-SYSIN.
   SELECT  TEST-OUTPUT-FILE
           ASSIGN TO S-SYSOUT.
   SELECT  TEST-SORT-FILE
           ASSIGN TO S-SORT.
*
 DATA  DIVISION.
 FILE SECTION.
 SD  SORT-WORK-FILE.
     RECORD  CONTAINS 80 CHARACTERS
     DATA   RECORD  IS  SORT-RECORD.
 01   SORT-RECORD.
       05  EMP-CODE     PIC  9(4).
       05  FILLER       PIC  X(4)     VALUE SPACES.
       05  EMP-NAME     PIC  X(17).
       05  FILLER       PIC  XXX      VALUE SPACES.
       05  EMP-ADDR     PIC  X(12).
       05  FILLER       PIC  X(4)     VALUE SPACES.
       05  ADDR-CODE    PIC  9(6).
       05  FILLER       PIC  X(4)     VALUE SPACES.
       05  EMP-STATE    PIC  X.
       05  FILLER       PIC   X(25)   VALUE SPACES.
*
 FD   IN-FILE.
 01   INPUT-RECORD.
       05  EMP-CODE     PIC  9(4).
       05  FILLER       PIC  X(4)     VALUE SPACES.
       05  EMP-NAME     PIC  X(17).
       05  FILLER       PIC  XXX      VALUE SPACES.
       05  EMP-ADDR     PIC  X(12).
       05  FILLER       PIC  X(4)     VALUE SPACES.
       05  ADDR-CODE    PIC  9(6).
       05  FILLER       PIC  X(4)     VALUE SPACES.
       05  EMP-STATE    PIC  X.
       05  FILLER       PIC  X(25)    VALUE SPACES.
*
 FD   OUT-FILE.
 01   OUTPUT-RECORD.
       05  EMP-CODE     PIC  9(4).
       05  FILLER       PIC  X(4)     VALUE SPACES.
       05  EMP-NAME     PIC  X(17).
       05  FILLER       PIC  XXX      VALUE SPACES.
       05  EMP-ADDR     PIC  X(12).
       05  FILLER       PIC  X(4)     VALUE SPACES.
       05  ADDR-CODE    PIC  9(6).
```

```
       05  FILLER       PIC  X(4)     VALUE SPACES.
       05  EMP-STATE   PIC  X.
       05  FILLER       PIC  X(25)    VALUE SPACES.
*
 PROCEDURE   DIVISION.
       SORT  SORT-WORK-FILE
          ON   ASCENDING  KEY  EMP-NAME  OF SORT-RECORD
                               EMP-CODE  OF SORT-RECORD
          INPUT  PROCEDURE  IS  IN-PROC-01  THRU  IN-PROC-EXIT
          INPUT  PROCEDURE  IS  OUT-PROC-01  THRU  OUT-PROC-EXIT
       STOP  RUN.
*
 IN-PROC-01.
       OPEN  INPUT  IN-FILE.
 IN-PROC-02.
       READ  IN-FILE
          AT END   GO TO IN-PROC-EXIT
       END-READ.
       IF  EMP-STATE OF INPUT-RECORD  NOT = 'A'
          GO TO IN-PROC-02
        END-IF.
        RELEASE  SORT-RECORD  FROM  INPUT-RECORD.
        GO TO  IN-PROC-02.
 IN-PROC-EXIT.
        CLOSE  IN-FILE.
*
 OUT-PROC-01.
       OPEN  OUTPUT  OUT-FILE.
 OUT-PROC-02.
       RETURN  SORT-WORK-FILE
         AT END
            SET  END-OF-RECORDS  TO TRUE.
       PERFORM  UNTIL  END-OF-RECORDS
         WRITE   OUTPUT-RECORD  FROM  SORT-RECORD
         MOVE  FUNCTION  LOWER-CASE(EMP-NAME OF SORT-RECORD)
             TO  EMP-NAME OF OUTPUT-RECORD
         RETURN  SORT-WORK-FILE
           AT END
            SET  END-OF-RECORDS  TO TRUE
       END-PERFORM.
 OUT-PROC-EXIT.
       CLOSE  OUT-FILE.
```

以上程序执行后，在其指定的排序输出文件 OUT-FILE 中将有如下数据记录。

```
1024   wu guo jun          BEI JING      100085    A
1005   xie fei             WU HAN        430023    A
1253   xie fei             WU HAN        430022    A
1235   zhu bo              WU HAN        430074    A
1204   zou li qiang        SHANG HAI     201620    A
```

该段程序综合应用了排序中的输入处理过程和输出处理过程。其中输入处理过程主要用于从输入的员工信息数据中筛选出人员状态为“A”的数据进行排序。输出处理过程将排序后的员工信息中的员工姓名数据项转换为小写字母表示。此处既可以作为对输入输出处理过程知识的一个回顾与扩充，也可以作为对数据排序技能的一个巩固和提高。

8.3 数据的合并

前面讲到了如何进行数据的排序。实际上，数据的合并和排序有很多相同之处，例如文件的定义、输出处理过程的编写等。并且，实际中数据的合并通常是在数据排序的基础之上完成的。因此，本节在讲解完数据排序的基础上，将较简略地讲解数据的合并。在学习数据的合并时，可对照前面所讲的数据的排序，以便更好地理解。

8.3.1 指定合并输入输出文件

合并同排序一样，通常也是有 3 种相关文件的。这 3 种相关文件分别为合并输入文件，合并输出文件，以及合并中间文件。其中合并中间文件的定义及使用同排序中间文件的几乎完全一致。因此，此处只讲解如何指定合并输入文件与合并输出文件。

1. 指定合并输入文件

合并输入文件也是通过 USING 短语进行指定的。但需要注意的是，对于合并而言，其输入文件至少应指定两个。并且，这些文件中的数据应该已经是排序过的，为有序数据。下面是合并输入文件的指定方式。

```
  MERGE  merge-file-name
     ON   ASCENDING  KEY  key-name
     USING    in-file-one  in-file-two  …      /*此处指定合并输入文件*/
     GIVING   out-file.
```

此处还需注意的是，合并输入文件是不能被指定为排序中间文件或者合并中间文件的。实际上，用于数据排序的排序输入文件同样也是有此要求的。只是在数据合并中，这一点往往更容易被忽略掉。

总之，关于数据合并中的合并输入文件，主要有以下几点需要注意。

- 合并输入文件通过 MERGE 语句中的 USING 短语进行指定。
- 合并输入文件应该指定两个及其以上的文件，不能只指定一个文件作为合并输入文件。
- 合并输入文件中的数据应该为有序数据。
- 不能指定通过 SD 语句定义的排序中间文件，或者合并中间文件为合并输入文件。

2. 指定合并输出文件

合并输出文件是通过 MERGE 语句中的 GIVING 短语所指定的。合并输出文件既可以只指定一个，也可以同时指定多个。该文件的指定方式如下。

```
MERGE  merge-file-name
     ON   ASCENDING  KEY  key-name
     USING    in-file-one  in-file-two  …
     GIVING   out-file.    /*此处指定合并输出文件*/
```

关于合并输出文件，需要注意的是当对该文件进行写入时，输入文件并不会被关闭。因此，不能将同一文件同时指定为合并输入文件与合并输出文件。例如，下面这段代码所指定的合并输出文件 EMP-FILE-ONE，同时也是合并输入文件中的一个。因此，这种指定方式是

错误的。该代码如下。

```
MERGE  TEST-MERGE-FILE
       ON   ASCENDING  KEY  EMP-CODE
       USING    EMP-FILE-ONE  EMP-FILE-TWO
       GIVING    EMP-FILE-ONE.     /*错误，不能将同一文件同时指定为合并输入和输出文件*/
```

作为补充说明，该方式在使用 SORT 语句进行的数据排序中则是允许的。因为在排序中对输出文件进行写入时，输入文件是关闭的。例如，以下代码是正确的。

```
SORT  TEST-SORT-FILE
      ON   ASCENDING  KEY  EMP-CODE
      USING    EMP-FILE
      GIVING    EMP-FILE.
```

8.3.2 编写合并中的输出处理过程

首先需要注意的一点是，在数据的合并中是没有输入处理过程的。因此，此处只讲解如何编写合并中的输出处理过程。

合并中输出处理过程的编写同排序中输出处理过程的编写基本类似。在数据的合并中，该输出处理过程是通过 MERGE 语句里的 OUTPUT PROCEDURE 所指定的。下面为一段包含有输出处理过程的数据合并操作代码。

```
 MERGE  TEST-MERGE-FILE
      ON   ASCENDING  KEY  EMP-CODE
      USING    EMP-FILE-ONE  EMP-FILE-TWO
      OUTPUT  PROCEDURE IS    TEST-MERGE-PROC. /*此处指定输出处理过程*/
……
TEST-MERGE-PROC.
 RETURN TEST-MERGE-FILE
   AT END SET END-OF-RECORDS TO TRUE
   ……
```

需要补充说明的一点是，该处理过程也可通过 THRU 短语指定多个处理步骤。实际上，这在之前的包含有输入输出处理过程的排序综合应用例子中已使用过。此方式输出处理过程指定代码如下。

```
 MERGE  TEST-MERGE-FILE
      ON   ASCENDING  KEY  EMP-CODE
      USING    EMP-FILE-ONE  EMP-FILE-TWO
      OUTPUT  PROCEDURE IS    MERGE-PROC-01  THRU  MERGE-PROC-EXIT.
……
MERGE-PROC-01.
  do something
   ……
MERGE-PROC-02.
   do something else
……
MERGE-PROC-EXIT.
   do something to end the procedure
```

最后需要强调的一点，是在数据合并中的输出处理过程里，也是通常需要用到 RETURN 语句的。RETURN 语句的语法格式与其在 SORT 语句中的语法格式相同。该语句在此处主要

用于按顺序将读入的每条记录放入合并中间文件中用于合并。

8.3.3 使用 MERGE 语句进行合并

使用 MERGE 语句对数据进行合并同使用 SORT 语句对数据进行排序基本类似。下面结合在合并基本概念这一小节中所举的例子进行讲解。

记录大型机从业人员信息的其中一个文件 IN-FILE-ONE 中的数据如下。

```
001000   WANG WEI      IBM CDL       2003/06/01
001003   YU ZHU        EDS           2007/07/05
001008   ZHENG JI      CSTS          2005/09/08
001010   LI QIANG      IBM CSDL      2007/06/08
001025   JIANG TAO     IBM ISSC      2006/11/20
```

另一个记录大型机从业人员信息的文件 IN-FILE-TWO 中的数据如下。

```
001004   WANG WEI      IBM TSS       2002/06/20
001007   YU ZHU        FDI           2004/07/12
001012   ZHENG JI      ISTC          2007/09/15
```

下面这个程序将以上两个文件中的数据进行合并。其中合并关键字取为人员编号数据项，合并顺序为升序。合并后所生成的文件名为 OUT-FILE。该程序完整代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  MERGE-TEST-PROG.
 AUTHER.      XXX.
*
 ENVIRONMENT  DIVISION.
 INPUT-OUTPUT  SECTION.
 FILE-CONTROL.
    SELECT  IN-FILE-ONE  ASSIGN TO S-SYSIN-1.
    SELECT  IN-FILE-TWO  ASSIGN TO S-SYSIN-2.
    SELECT  OUT-FILE   ASSIGN TO S-SYSOUT.
    SELECT  TEST-MERGE-FILE  ASSIGN TO S-MERGE.
*
 DATA  DIVISION.
 FILE SECTION.
 SD  TEST-MERGE-FILE.
     RECORD  CONTAINS 80 CHARACTERS
     DATA   RECORD  IS  SORT-REC.
 01   SORT-REC.
       05  MEM-CODE     PIC  9(6).
       05  FILLER       PIC  X(4)     VALUE SPACES.
       05  MEM-NAME     PIC  X(17).
       05  FILLER       PIC  XXX      VALUE SPACES.
       05  MEM-COMP     PIC  X(17).
       05  FILLER       PIC  XXX      VALUE SPACES.
       05  ON-BOARD     PIC  X(10).
       05  FILLER       PIC  X(20)    VALUE SPACES.
 FD   IN-FILE-ONE.
 01   IN-REC-ONE.
       05  MEM-CODE     PIC  9(6).
       05  FILLER       PIC  X(4)     VALUE SPACES.
       05  MEM-NAME     PIC  X(17).
       05  FILLER       PIC  XXX      VALUE SPACES.
```

```
       05  MEM-COMP     PIC  X(17).
       05  FILLER       PIC  XXX      VALUE SPACES.
       05  ON-BOARD     PIC  X(10).
       05  FILLER       PIC  X(20)    VALUE SPACES.
 FD   IN-FILE-TWO.
 01   IN-REC-TWO.
       05  MEM-CODE     PIC  9(6).
       05  FILLER       PIC  X(4)     VALUE SPACES.
       05  MEM-NAME     PIC  X(17).
       05  FILLER       PIC  XXX      VALUE SPACES.
       05  MEM-COMP     PIC  X(17).
       05  FILLER       PIC  XXX      VALUE SPACES.
       05  ON-BOARD     PIC  X(10).
       05  FILLER       PIC  X(20)    VALUE SPACES.
 FD   OUT-FILE.
 01    OUT-REC.
       05  MEM-CODE     PIC  9(6).
       05  FILLER       PIC  X(4)     VALUE SPACES.
       05  MEM-NAME     PIC  X(17).
       05  FILLER       PIC  XXX      VALUE SPACES.
       05  MEM-COMP     PIC  X(17).
       05  FILLER       PIC  XXX      VALUE SPACES.
       05  ON-BOARD     PIC  X(10).
       05  FILLER       PIC  X(20)    VALUE SPACES.
*
PROCEDURE   DIVISION.
      MERGE  TEST-MERGE-FILE
          ON   ASCENDING  KEY  MEM-CODE  OF SORT-REC
          USING   IN-FILE-ONE   IN-FILE-TWO
          GIVING   OUT-FILE.
      STOP  RUN.
```

该程序执行后，输出文件 OUT-FILE 中的数据应该如下。

```
001000   WANG WEI      IBM CDL       2003/06/01
001003   YU ZHU        EDS           2007/07/05
001004   WANG WEI      IBM TSS       2002/06/20
001007   YU ZHU        FDI           2004/07/12
001008   ZHENG JI      CSTS          2005/09/08
001010   LI QIANG      IBM CSDL      2007/06/08
001012   ZHENG JI      ISTC          2007/09/15
001025   JIANG TAO     IBM ISSC      2006/11/20
```

由此可见，数据的合并同数据的排序基本类似。对于合并同排序的区别与联系，可以简单地归纳为如下几条。

- 数据的合并使用 MERGE 语句；数据的排序使用 SORT 语句。
- 合并输入文件至少有两个；排序输入文件通常则为一个。
- 合并输入文件中的数据必须为有序数据；排序输入文件中的数据通常为无序数据。
- 合并中只能有输出处理过程；排序中既可有输出处理过程，也可有输入处理过程。
- 合并与排序通常都具有合并/排序输入输出文件以及合并/排序中间文件。以上 3 种文件在合并与排序中的定义方式是相同的。并且，每种文件之间的组织结构也应该是相同的。

8.4 本章回顾

本章主要介绍了在 COBOL 中是如何实现对文件中的数据进行排序与合并的。此处所说的排序与合并指的是通过特定语句实现的相应功能，而并不是通常所说的算法。

数据的排序主要是通过 SORT 语句实现的。关于数据的排序，需要掌握如何使用 SD 语句定义排序中间文件，掌握如何通过 SORT 语句中的 USING 短语和 GIVING 短语指定排序输入输出文件，能够使用完整的 SORT 语句进行基本的数据排序。同时，还应该掌握如何编写排序中所用到的输入输出处理过程，以及如何通过 SORT 语句并综合应用排序中的输入输出过程完成相对较复杂的数据排序。

数据的合并功能在很大程度上和数据的排序是类似的。关于数据的合并，同样需要能够指定合并输入输出文件，能够编写合并中的输出处理过程，能够使用 MERGE 语句完成基本的合并功能。并且在最后能够通过比较排序与合并的异同，更深入地理解二者的概念及应用。

第 9 章 COBOL 中的表

表是一组存储在连续空间上的相似数据的集合。表可以用于减少程序员书写的代码量，方便对大量相似数据的处理，提高程序的运行效率。此外，表还有一个最重要的功能，就是对数据的查找。本章将介绍表的概念和用途，并重点介绍几类典型结构的表。同时，还将介绍使用表进行数据查找的一些常用方式和语句。

9.1 表的简介

表是 COBOL 语言所特有的一个概念。表最基本的用途是方便程序员的开发工作，提高程序执行效率，方便数据的查找。本节将主要介绍表的基本概念和用途。在最后还会简单介绍一下几类典型结构表的大致概念，为下面的学习打好基础。

9.1.1 为什么要使用表

COBOL 语言主要应用于大型高端领域的开发，如国内的金融银行业。因此，需要用到对于大量数据的处理。在实际应用中，这些大量的数据中往往很多都是有联系性和相似性的。将这些有联系性和相似性的数据作为一个整体来进行处理，主要有以下两大好处。

- 减少代码编写量，极大地方便程序员的开发工作。
- 优化数据操作，提高程序执行效率。

将有着相似属性的数据作为一个整体来进行处理的方式，实际上正是引入表这样一个机制的初衷。在实际应用中，基于以上概念所产生的表最主要的用途是对数据的查找。

9.1.2 表的基本概念

表就是一组存储在连续空间上的相似数据的集合。所有这些数据被指定为一个名称，即表的一个条目名。以下是一个表的典型结构。

```
01  EXAMPLE-TABLE-ONE.                    /*表的名称*/
   05  TABLE-COL  OCCURS 5 TIMES.         /*表的一列，共重复 5 列*/
      10  TABLE-ITEM-1     PIC X(1).      /*表的第一个条目*/
```

```
        ......                                  /*省略表的其他条目*/
        10  TABLE-ITEM-N     PIC X(3).          /*表的最后一个条目*/
```

由以上代码可看出，表这种数据结构主要是通过 OCCURS 语句创建的。关于 OCCURES 语句，将在后面进行更详细的讲解。表的一列包含了表的所有条目。当表中仅有一个条目时，则不需要定义表的列，而直接由条目取代。在该情况下表的定义如下。

```
01  EXAMPLE-TABLE-TWO.                         /*表的名称*/
    05  TABLE-ITEM     PIC X(5).              /*表的条目*/
                       OCCURES 3 TIMES.       /*该条目重复 3 次*/
```

9.1.3 表的基本用途

表的基本用途是减少代码编写量，优化数据操作，便于数据查找。对于如何利用表进行数据查找，在本章后面几节中将详细讲到。对于如何通过表减少代码编写量和优化数据操作，下面将结合一个简单实例进行具体讲解。

在银行系统的应用软件中，一个类似于零存整取的系统是十分常见的。为突出本节讨论的问题，可以对此进行一个抽象模型的提取。具体做法就以最单纯的零存整取系统为基本模型，不考虑利率和税率等。并假设存取时间为一年，同时只涉及到每个月的存款额这样一个数据项。

为更好地体现出使用表机制的优势，可以将其分为不使用表和使用表这两种情况分别进行讨论。通过对比，更直观地理解表的用途。下面对这两种情况作详细介绍。

1．不使用表的情况

在不使用表的情况下，通常将得到下面这样一组数据定义。

```
01  ANNUAL-DEPOSIT-DATA.                       /*年度存款数据，是一个 01 级数据。*/
    05  JAN-DEPOSIT     PIC 9(5).             /*一月份存款数据，是一个 05 级数据。*/
    05  FEB-DEPOSIT     PIC 9(5).             /*二月份存款数据，是一个 05 级数据。*/
    05  MAR-DEPOSIT     PIC 9(5).             /*三月份存款数据，是一个 05 级数据。*/
......                                         /*省略四月份到十一份的存款数据*/
    05  DEC-DEPOSIT     PIC 9(5)              /*十二月份存款数据，是一个 05 级数据。*/
```

然而，这样一种定义方式在实际应用中是存在着很多不足之处的，分别列举如下。

- 在 COBOL 中的数据部中编写以上这段定义的完整代码就是一项费时费力，并且相当枯燥的工作。
- 以上只是假设存取时间为一年，并且只包含月存款额这样一个数据，实际中往往并不是这样的。例如某人的存取时间为 10 年，那么依照以上这种定义方式，就不仅仅是 12 条 PIC 语句了，而是 120 条 PIC 语句。同时，在实际操作中每个月所涉及的数据也不可能只包含存款额而已，至少还要包括一些利息额、税款等。这样，即使存取时间仅为一年，在定义中也不只有 12 条 PIC 语句了。
- 在 COBOL 程序的过程部里对通过以上方式定义的数据进行处理也是件十分麻烦的事情。最常见的，比如用之前讲到的 EVALUATE 语句对该组数据进行处理，这时就需要对应地在其后添加 12 个 WHEN 字句来完成操作了。

2．使用表的情况

对于不使用表机制所造成的种种弊端，于是引入了表的机制。仔细分析表的定义，可以

看到若要形成一个表，必须具备以下两个基本条件。

- 数据存储在连续空间上。
- 所有数据具有相同的数据类型（即相似数据）。

每个月的存款额是依据月份顺序来安排的，在内存中显然应该将其存储在一片连续的存储空间上。这样，该数据结构就实现了第一个条件。

对于每个月所存入的金额这样一个数据类型，月和月之间必然具有相似的属性。结合前面不使用表机制所定义的月存款额，可以看到这些数据都有着相同的数据类型——PIC 9(5)。于是，该数据结构又实现了第二个条件。

实现建立表的两个基本条件后，便可以重新将以上数据定义为一个表，代码如下。

```
01  ANNUAL-DEPOSIT-TABLE.
    05  DEPOSITS      PIC 9(5)
                       OCCURES 12 TIMES.
```

代码说明如下。

- ANNUAL-DEPOSIT-TABLE：年度存款数据，是一个 01 级数据，同时也是该表的名称。
- DEPOSITS：每个月存款数据，是一个 05 级数据，同时也是该表的一个条目。
- OCCURES 12 TIMES：重复出现 12 次。
- PIC 9（5）：每个月的存款数据类型，为 5 个单位长度的数值数据类型。

这种方式的定义，和之前没有使用表机制的定义在本质上是等同的。然而，使用该方式完成定义所需编写的代码量大大减少了，也方便了对所定义数据的具体操作。以上诸多不使用表机制所造成的弊端在这里都得到了很好的解决。由此可以看到，引入了表的机制后，将大大方便程序员的开发工作，也方便了数据处理，提高了程序运行效率。

9.1.4 几类典型结构的表

由表的基本概念发展开来，在 COBOL 中有几类典型结构的表。这几类典型结构的表在大体上可分为下标表和索引表两大类。另外，根据表的重复次数定义又有定长表和变长表。此外，表还允许嵌套，因此还有嵌套表。这几类表均符合表的基本定义，都是一组连续存储的相似数据的集合。但每种类型的表在此基础上又各有特色。以下几种类型的表两两之间各自存在排它性，即一个表不可能同时为此两种类型的表。

- 下标表和索引表。
- 定长表和变长表。

其余各表之间都是并行的。例如一个表可同时为嵌套表和索引表，或同时为定长表、下标表和嵌套表等。

对于下标表，本章将主要介绍 OCCURS 语句、PERFORM VARYING 语句和 FUNCTION 语句。并且，该部分内容还将涉及到下标的基本概念和表的初始化。最后，还将介绍 COBOL 中 3 种基于表的典型数据查找方式，即直接查找方式、顺序查找方式、二分查找方式。

对于索引表的操作，将主要介绍基于该表操作的 SET 语句，SEARCH 语句，SEARCH ALL 语句。另外还将比较索引表和下标表的异同，并分析索引表的内部存储结构。

对于定长表和变长表，本章将在其后一并进行讲解。重点在于通过 DEPENDING ON 语句区分以上两种表。

对于嵌套表，主要是对以上各表的一个嵌套处理，因此将在最后讲解。

9.2 下标表

下标表也叫 Subscripted 表，通过下标组织和访问表中定义的数据条目。此种类型的表是最常见的表，其他各种类型的表的结构可以说都是由该表的结构派生而来的。本书中，若不特别指定，所说的表均为下标表。

9.2.1 如何定义下标表

在 COBOL 中，下标相当于各数据在表中的编号。以下标管理数据的表也就是下标表。下标表实际上就是通常所说的最基本的表。因此，对于下标表的定义方式，同上节中表的定义方式类似。这里继续延用上节的例子，则下标表的结构定义如下。

```
01  ANNUAL-DEPOSIT-TABLE.
    05  DEPOSITS      PIC 9(5)
                      OCCURES 12 TIMES.
```

此外，由于作为下标表，因此还需再定义一个下标。下标名称任意指定，但通常是在对应的表的条目后加上 SUB，以利区分。定义语句如下。

```
05  DEPOSITS-SUB    PIC 99    USAGE IS COMP.
```

将下标定义语句加入到表的基本结构定义语句中就可定义一个下标表了。对于上节零存整取系统的例子而言，完整的下标表的定义语句如下。

```
01  ANNUAL-DEPOSIT-TABLE.
    05  DEPOSITS      PIC 9(5)
                      OCCURES 12 TIMES.
  05  DEPOSITS-SUB    PIC 99    USAGE IS COMP.
```

这里，ANNUAL-DEPOSIT-TABLE 是该下标表的名称；DEPOSITS 是该表的一个条目名，共包括 12 个相似数据；DEPOSITS-SUB 是该表的下标。

9.2.2 下标的作用

前面已经讲到，在表中，相似数据被指定为同一个名称，作为表的一个条目。那么，如何访问这些拥有同一个名称的不同数据？这里就要用到下标了。下标相当于表中各数据的编号，以此指定和访问表中定义的各项数据。

仍然通过上例进行讲解。对应上面定义的下标表，原始数据结构即在表的基本用途一节中“不使用表的情况”下的定义。代码如下。

```
01  ANNUAL-DEPOSIT-DATA.
    05  JAN-DEPOSIT     PIC 9(5).
    05  FEB-DEPOSIT     PIC 9(5).
    05  MAR-DEPOSIT     PIC 9(5).
    ......
    05  DEC-DEPOSIT     PIC 9(5)
```

当使用此前定义的下标表来访问上面定义的不同月份的存款额时，就需要用到下标了。访问不同月份存款额所对应的含有不同下标的条目依此如下。

- 访问 JAN-DEPOSIT 对应于 DEPOSITS （1）
- 访问 FEB-DEPOSIT 对应于 DEPOSITS （2）
- 访问 MAR-DEPOSIT 对应于 DEPOSITS （3）

……

- 访问 DEC-DEPOSIT 对应于 DEPOSITS （12）

在本例中的下标表里已定义了下标，即 DEPOSITS-SUB。因此，通常是用 DEPOSITS-SUB 变量保存 1 到 12 这 12 个数字，以访问所对应的不同月份的数据。

使用下标 DEPOSITS-SUB 访问数据，即将表的数据条目和下标相组合。组合后形成的表示方式如下。

```
DEPOSITS (DEPOSITS-SUB)
```

该表示方式可以看作是一项独立的数据，如某一个月的存款额等。DEPOSITS-SUB 根据定义为不超过两位数字的整数。当 DEPOSITS-SUB 为 5 时，以上数据就为 5 月份的存款额；当 DEPOSITS-SUB 为 10 时，以上数据就为 10 月份的存款额，依此类推。

这样，通过在数据条目后使用下标，就可以访问表中任意一项具体的数据了。此外，下标作为一个数值类型的变量，还可以指定相关数据项或进行条件判断。

使用下标指定相关数据项，通常是通过“+”，“−”号实现的。比如，若要比较 3 月份和 5 月份存款额的大小，可以通过以下代码进行。

```
MOVE 3 TO DEPOSITS-SUB
IF  DEPOSITS (DEPOSITS-SUB) >= DEPOSITS (DEPOSITS-SUB + 2)
    IF  DEPOSITS (DEPOSITS-SUB) > DEPOSITS (DEPOSITS-SUB + 2)
        MOVE 'MARCH' TO MAX-MONTH
    ELSE
        MOVE 'EQUAL' TO MAX-MONTH
    END-IF
ELSE
    MOVE 'MAY' TO MAX-MONTH
```

该段代码中，首先将“3”MOVE 到下标 DEPOSITS-SUB 中。这样，表示 3 月份存款额数据的代码如下。

```
DEPOSITS (DEPOSITS-SUB)
```

对于 5 月份的存款额，同样也可以通过这种方式进行表示，但这样就略嫌麻烦。若此前在下标中已保存有数据，则可以通过“+”，“−”号来访问相关数据。本例中由于下标 DEPOSITS-SUB 已经保存了数字“3”，因此可以直接表示 5 月份的数据如下。

```
DEPOSITS (DEPOSITS-SUB + 2)
```

这时，DEPOSITS-SUB 仍然保存数字“3”。但括号中的下标通过运算已为数字“5”了，表示的是 5 月份的存款数据。当然，也可以先得到 5 月份的数据，再以 5 月份的数据为基准得到 3 月份的数据。实现方式见以下代码，效果和上面的是等同的。

```
MOVE 5 TO DEPOSITS-SUB
IF  DEPOSITS (DEPOSITS-SUB - 2) >= DEPOSITS (DEPOSITS-SUB)
     IF  DEPOSITS (DEPOSITS-SUB - 2) > DEPOSITS (DEPOSITS-SUB)
        MOVE 'MARCH' TO MAX-MONTH
     ELSE
        MOVE 'EQUAL' TO MAX-MONTH
```

```
        END-IF
    ELSE
        MOVE 'MAY' TO MAX-MONTH
```

对于使用下标进行条件判断，同使用其他变量进行一样。关于条件判断，可参阅本书的流程控制一章，此处不再赘述。

9.2.3 下标的格式要求

关于下标的格式要求，主要有以下几点注意的地方。

（1）下标必须为整型数据。

（2）下标既可以为具体数字，也可以为在数据部定义的一个变量。

通过具体数字定义的下标包含在以下示例语句中。

```
MOVE  DEPOSITS (10) TO  PL-DEPOSIT.
```

通过变量定义的下标包含在以下示例语句中。

```
SUBTRACT  TAX-RATE  FROM  DEPOSITS (DEPOSITS-SUB).
```

（3）下标可以通过输入数据得到。但在实际开发中，通常仍然需要在数据部的工作存储节 WORKING-STORAGE 中定义。

（4）在定义下标时，最好在后面加上 USAGE IS COMP 代码。USAGE IS COMP 代码并非必须要求添加，只是加上后可以提高该数据访问的效率。定义方式如下。

```
 05  DEPOSITS-SUB    PIC 99    USAGE IS COMP.
```

此处也可简写为“COMP”，方式如下。

```
05  DEPOSITS-SUB    PIC 99    COMP.
```

（5）每个后面跟有 OCCURS 语句的数据条目都应有相应的下标。如此，定义的一组数据方能通过下标访问到。

（6）在定义拥有下标结构的数据时，括号和前面的数据条目名称之间至少应有一个空格。否则，系统会将其认为是一个含有括号的非法数据。该注意事项如下所示。

```
DEPOSITS   (DEPOSITS-SUB)
          ↑
```

此处至少应有一个以上的空格（含一个）

最后，关于第（4）点和第（6）点需要再次强调一下，这两点内容是十分重要的。在初学 COBOL 进行开发时，也是最容易被忽视的。下面特将此二点要求提取出来，列举如下。

- 在定义下标时，提倡在后面加上 USAGE IS COMP 代码。该代码不添加亦不会报错，但添加后可以提高该数据访问的效率。
- 在定义拥有下标结构的数据时，括号和前面的数据条目名称之间至少应有一个空格。

9.3 定义表语句 OCCURS

从前面所述已经可以看到，定义一个表关键是通过 OCCURS 语句进行的。OCCURS 语句定义了表中条目的重复次数。拥有可重复的数据条目结构是表区别于其他数据结构的本质

特征。因此，可以直观地根据是否具有 OCCURS 语句，来判断该数据结构是否为表。

9.3.1 OCCURS 语句的使用方法

OCCURS 语句的语法格式如下。

```
OCCURS n TIMES.
```

其中，“n”为一个整型数据类型，指明了重复的次数。“TIMES”是该语句固定格式的一部分内容，不可任意指定。OCCURS 语句用于在数据部的工作存储节定义表，并且 OCCURS 不能出现在 01 级数据类型后。例如，以下代码就错误地使用了 OCCURS 语句。

```
01   TEST-TABLE-ONE            OCCURS 5 TIMES.
     05    TEST-ITEM-ONE    PIC  X (3).
     05    TEST-ITEM-TWO    PIC  X (5).
```

这里，TEST-TABLE 是一个 01 级数据，其后是不允许跟 OCCURS 语句的。通常，应该如下使用 OCCURS 语句进行表的定义。

```
01   TEST-TABLE-ONE
     05    TEST-ITEM-ONE    PIC  X (3).    OCCURS 5 TIMES.
     05    TEST-ITEM-TWO    PIC  X (5).    OCCURS 5 TIMES.
```

以上代码中，TEST-ITEM-ONE 数据条目和 TEST-ITEM-TWO 数据条目重复次数相同。因此，为简便起见，通常情况下可使用表的列来统一定义。当使用表的列来定义时，OCCURS 语句应出现在表的列中，代码如下。

```
01   TEST-TABLE-ONE
     05    TEST-ROW         OCCURS 5 TIMES.        /*OCCURS 语句在表的列后*/
         10    TEST-ITEM-ONE    PIC  X (3).
         10    TEST-ITEM-TWO    PIC  X (5).
```

这样，该段代码和上段代码实现的效果是等价的，但此处只出现了一条 OCCURS 语句。当表中数据条目较多时，可以很显著地简化代码的书写。

9.3.2 使用 OCCURS 语句得到的表空间结构

这里需要说明的是，使用 OCCURS 语句只是避免了重复书写相类似的定义代码。OCCURS 语句并没有压缩定义数据的存储空间。数据的存储空间大小和不使用 OCCURS 语句时，定义数据的存储空间大小是相同的。

通过下面的图解可以更直观地说明这一点。首先，使用 OCCURS 语句定义一个普通结构的表，代码如下。

```
01   TEST-TABLE-TWO
     05    TEST-ROW         OCCURS 3 TIMES.
         10    TEST-ITEM-ONE    PIC  X (1).
         10    TEST-ITEM-TWO    PIC  X (2).
```

对应该表数据的存储结构如图 9.1 所示。

由此可见，使用表定义的数据存储空间大小和不使用表的大小是一样的。使用表定义的数据存储空间大小的计算公式如下。

```
( item 1  + item 2 + item 3 + ……  +  item n ) * OCCURS 重复的次数 =  空间大小
```

这里，item 1 指数据条目 1 的存储空间大小。对应于上例，如下代码中的后面 PIC(1)中的“1”。

```
10    TEST-ITEM-ONE    PIC  X (1).
```

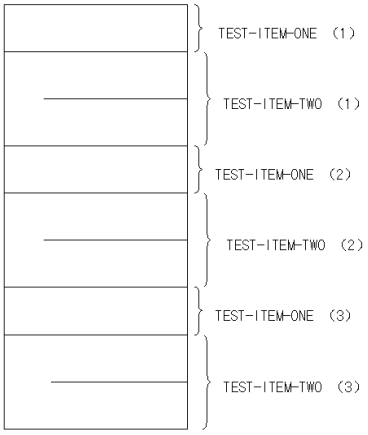

图 9.1　TEST-TABLE-TWO 表的存储结构

Item 2 指数据条目 2 的存储空间大小。对应上例，代码中的后面 PIC(2)中的“2”。

```
 10    TEST-ITEM-TWO    PIC  X (2).
```

从 item 3 往后依此类推，直到最后一个数据项 item n 为止。上例中只有两个数据条目，因此 item n 即 item 2。最后，根据以下语句得到 OCCURS 的重复次数为“3”。

```
05    TEST-ROW         OCCURS 3 TIMES.
```

这样，对比上面计算表空间大小的公式，已经知道了 item 1 直到 item n 的大小，同时也知道了 OCCURS 重复的次数。因此，可根据该公式计算表 TEST-TABLE-TWO 的存储空间大小如下。

```
(1 + 2) * 3 = 9
```

对于初学者而言，常常容易忘记在最后乘上 OCCURS 重复的次数。如果没有乘上 OCCURS 重复的次数，将得到错误的空间大小如下。

```
1 + 2 =3              ← 错误的计算方式
```

以上这种计算方式，只是简单地将各数据条目的大小当作数据项的大小相加。这样做，是错误地理解了表的结构意义，忽视了 OCCURS 语句的存在，失去了表的意义。

9.4　浏览表语句 PERFORM VARYING

上一小节中讲到的 OCCURS 语句是在表的定义中需要使用到的。本节将主要讲解在表的操作中经常用到的一个语句，即 PERFORM VARYING 语句。

9.4.1　PERFORM VARYING 语句的使用方法

PERFORM VARYING 语句的基本格式如下所示。

```
PERFORM …
VARYING …FROM …BY…
UNTIL  …
```

其中，“…”部分是相应的变量，根据具体的不同情况而有所不同。

- statement 变量：进行处理过程的名称。
- item-sub 变量：通常为表中数据对应的下标。
- x1 变量：通常指从表中的哪一个下标开始处理。
- x2 变量：通常指每次处理后对于下标的增量。
- x3 变量：通常指下标经过增量处理后达到的上限值。

使用 PERFORM VARYING 语句处理过程所对应的流程图如图 9.2 所示。

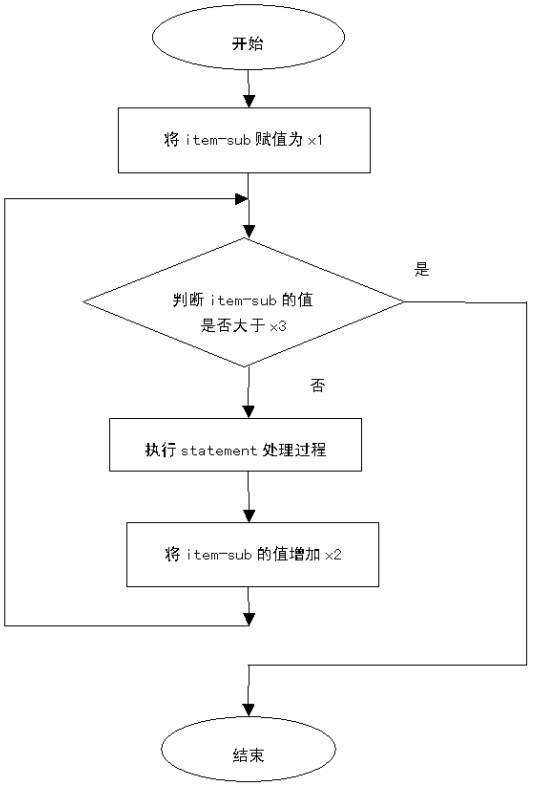

图 9.2　PERFORM VARYING 语句处理过程流程图

通过此流程图可以看到，PERFORM VARYING 语句的处理过程实际上是通过以下步骤完成的。

（1）将表的下标变量 item-sub 赋值为 x1。

（2）判断 item-sub 的值是否大于 x3。若不大于，则顺次执行步骤 3，否则跳转到步骤 6。

（3）执行 statement 处理过程。该处理过程通常都会涉及到对含有下标数据类型的操作。

（4）将 item-sub 的值增加 x2，同时保存在 item-sub 里。

（5）跳转到步骤 2 继续执行。

（6）结束处理。

9.4.2 如何使用 PERFORM VARYING 语句处理表中数据

上面谈到了 PERFORM VARYING 语句的基本处理流程。那么，如何使用 PERFORM VARYING 语句处理表中数据呢？下面仍然结合具体实例进行讲解。

以员工工资管理系统为例，首先进行简化模型提取。提取模型只包括一个员工一年 12 个月的工资，即只含有 12 个数据项。将这 12 个数据定义为下标表结构如下。

```
01  ANNUAL-SALARY-TABLE.
   05   SALARIES          PIC 9(5)
                          OCCURS 12 TIMES.
05   SALARY-SUB            PIC 99    USAGE IS COMP.
```

这时，若要计算全年总工资，通常的做法只能是将每月工资一条条地相加。代码如下。

```
COMPUTE  ANNUAL-TOTAL =
          SALARIES  (1) + SALARIES  (2) + SALARIES  (3) +
          SALARIES  (4) + SALARIES  (5) + SALARIES  (6) +
          SALARIES  (7) + SALARIES  (8) + SALARIES  (9) +
          SALARIES  (10) + SALARIES  (11) + SALARIES  (12) .
```

显然，这种方法非常浪费时间。既然表中定义的数据都是相似数据，并且都通过下标有着对应的数据表示方式。那么，通过对下标的相关处理，必然可以简化操作。利用 PERFORM VARYING 语句进行对表的操作，正是基于这个原理。

使用 PERFORM VARYING 语句计算全年总工资的代码如下。

```
PERFORM   200-ADD-TO-TOTAL
           VARYING   SALARY-SUB  FROM 1 BY 1
           UNTIL     SALARY-SUB > 12
......
200-ADD-TO-TOTAL.
           ADD SALARIES  (SALARY-SUB)
               TO  ANNUAL-TOTAL.
```

该段代码对应于 PERFORM VARYING 原型语句结构各名称如下。

- 200-ADD-TO-TOTAL 对应于原型结构中的 statement。
- SALARY-SUB 对应于原型结构中的 item-sub。
- FROM 1 BY 1 前一个“1”对应于原型结构中的 x1。
- FROM 1 BY 1 后一个“1”对应于原型结构中的 x2。
- SALARY-SUB > 12 中的“12”对应于原型结构中的 x3。

其中 200-ADD-TO-TOTAL 处理过程的代码如下。

```
200-ADD-TO-TOTAL.
           ADD SALARIES  (SALARY-SUB)
               TO  ANNUAL-TOTAL.
```

该段代码实现的功能是将 SALARIES (SALARY-SUB)的值添加到 ANNUAL-TOTAL 数据中。由于 SALARIES (SALARY-SUB)数据的下标——SALARY-SUB 的值是从“ 1 ”到“12”的，并且，每步处理过程后，SALARY-SUB 的值都相应地增加“1”。因此，以上利用 PERFORM VARYING 语句进行处理的代码实现了计算全年总工资的功能。

通过上例可以看到，使用 PERFORM VARYING 语句大大方便了处理过程。实际上，利

用 PERFORM VARYING 语句处理表中数据，主要是利用了表中下标这一特征属性的。PERFORM VARYING 语句主要用到了循环结构。基于数据的相关性，以及下标的连续性，通过循环结构进行数据处理，正是该语句的本质特征。

9.4.3 PERFORM VARYING 语句的一些灵活应用

在实际应用中，PERFORM VARYING 语句不仅仅用于计算表中所有数据的总和。下面，仍然结合工资管理系统的模型，通过几个例子讲解 PERFORM VARYING 语句的灵活应用。

（1）计算单号月份的工资总和。这里实际上是要计算 1、3、5、7、9、11 这 6 个月的工资总和。可以看出，相邻两个月之间都相隔 1 个月，这是有规律的。实际上，具体到代码实现，也就是表中的下标每次增量要求为 2。代码如下。

```
PERFORM   200-ADD-TO-TOTAL
          VARYING  SALARY-SUB    FROM 1 BY 2
          UNTIL     SALARY-SUB > 11
```

（2）计算第四季度的工资总和。这里是要计算 10 月份、11 月份、12 月份这 3 个月的工资总和。因此，就不能再从 1 月份开始计算起了。代码如下。

```
PERFORM   200-ADD-TO-TOTAL
          VARYING  SALARY-SUB    FROM 10 BY 1
          UNTIL     SALARY-SUB > 12
```

（3）计算前半年的工资总和。这里是要计算 1 月份～6 月份的工资总和。因此，最后就不能以 12 月份作为结束了。实际上，例 1 中最后也是以 11 月份作为结束的。本例实现代码如下。

```
PERFORM   200-ADD-TO-TOTAL
          VARYING  SALARY-SUB    FROM 1 BY 1
          UNTIL     SALARY-SUB > 6
```

9.4.4 PERFORM VARYING 语句和 PERFORM 语句的比较

在本小节最后，来比较一下 PERFORM VARYING 语句和之前学习过的 PERFORM 语句。通过比较，加深对于 PERFORM VARYING 语句的理解。同时，也可以此作为对于 PERFORM 语句的一个简单回顾。

在上面用到的的工资管理系统模型中，使用 PERFORM VARYING 语句计算全年总工资的代码如下。

```
PERFORM   200-ADD-TO-TOTAL
          VARYING   SALARY-SUB  FROM 1 BY 1
          UNTIL      SALARY-SUB > 12
......
200-ADD-TO-TOTAL.
           ADD SALARIES  (SALARY-SUB)
               TO  ANNUAL-TOTAL.
```

该段代码实现的功能，实际上同样也可以使用 PERFORM 语句完成，代码如下。

```
MOVE  1  TO   SALARY-SUB.
PERFORM  200-ADD-TO-TOTAL
```

```
          UNTIL     SALARY-SUB > 12
……
200-ADD-TO-TOTAL.
          ADD  SALARIES  (SALARY-SUB)
               TO  ANNUAL-TOTAL.
          ADD 1 TO SALARY-SUB.
```

由此可见，PERFORM VARYING 语句和 PERFORM 语句都具有循环结构。但 PERFORM VARYING 语句在循环结构的基础上，还增加了一个步进的功能。该功能用于对表进行操作是十分方便的。

9.5 表的初始化

在实际对表进行处理前，通常还要对其进行初始化。对表进行初试化可以分为硬性编码方式和输入文件载入方式两种类型。此外，还有一些初始化表的灵活技巧。以下分别进行介绍。

9.5.1 使用硬性编码方式初始化表

使用硬性编码方式初始化表，主要是通过 VALUE 语句实现的。VALUE 跟在每个具体数据项之后（非数据条目）。VALUE 语句指定的值也就是该数据的初始化值。对表中每个数据项初始化后，也就实现了对整个表的初始化。

例如，对于保存一周 7 天的数据的表，通过硬性编码方式初始化代码如下。

```
01   WEEK-VALUES.
     05   FILLER   PIC X(10)    VALUE  'MONDAY'.
     05          PIC X(10)    VALUE  'TUESDAY'.
     05          PIC X(10)    VALUE  'WEDNESDAY'.
     05          PIC X(10)    VALUE  'THURSDAY'.
     05          PIC X(10)    VALUE  'FRIDAY'.
     05          PIC X(10)    VALUE  'SATURDAY'.
     05          PIC X(10)    VALUE  'SUNDAY'.
01   WEEK-TABLE-ONE   REDEFINES   WEEK-VALUES.
     05   DAYS    PIC X(10)    OCCURS 7 TIMES.
```

可以看到，使用硬性编码方式初试化表，实际上可依此通过以下 3 步组成。

（1）使用 VALUE 语句给表中需要用到的各数据赋初值，并写入存储空间。

（2）使用 REDEFINES 语句为以上数据存储空间指定另一个名称，即表名。

（3）使用 OCCURS 语句建立表。

这里需要特别注意的是，REDEFINES 语句仅仅是为以上存储空间另指定一个名称而已。该语句并没有再另外单独开辟一块存储空间。该语句作用于存储空间的方式如图 9.3 所示。

9.5.2 使用输入文件载入方式初始化表

使用输入文件载入方式初始化表，首先需要有一个输入文件。该输入文件保存表中各数据的初始化值。将该输入文件在过程部读入后，使用前面讲到的 PERFORM VARYING 语句完成对表的初始化。对于硬性编码初始表所用到的例子，可以通过输入文件载入方式来初始化。首先，需要有一个输入文件，假设该文件的文件名为 INPUT-WEEK-FILE。该文件的内

容如表 9.1 所示。

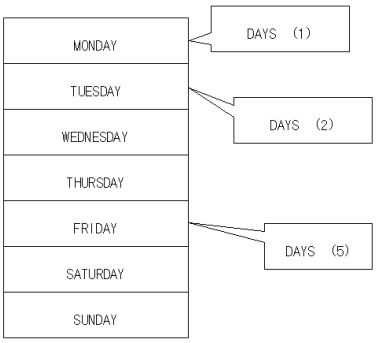

图 9.3　使用 REDEFINES 语句后的存储空间

表 9.1　　INPUT-WEEK-FILE 文件内容

MONDAY	FRIDAY
TUESDAY	SATURDAY
WEDNESDAY	SUNDAY
THURSDAY	

每行对应该文件的一个记录。可以看到，该文件的记录对应周一至周日这七天的数据，正好为硬性编码中赋给表中各数据的初始化值。在使用输入文件载入方式初始化表前，首先要对表进行定义。这点和硬性编码方式是不同的。在硬性编码方式里，是直接通过 REDEFINES 对表进行定义的。定义代码如下。

```
01   WEEK-TABLE-TWO.
     05   DAYS     PIC X(10)
                   OCCURS  7  TIMES.
    05    DAY-SUB   PIC 9     USAGE IS COMP.
```

对输入文件也要进行定义。定义输入文件的方式主要用到前面所讲 FD 语句。通过使用 FD 语句，在数据部的文件节里对输入文件进行定义。代码如下。

```
DATA DIVISION.
FILE SECTION.
FD  INPUT-WEEK-FILE
RECORDING MODE IS F.
01  INPUT-DATA     PIC X(10).
```

以上代码中，INPUT-WEEK-FILE 为该输入文件的文件名。INPUT-DATA 为该文件中每条记录的名称。每条记录的模式为 F，也就是定长模式。

完成了表和输入文件的定义后，可以通过以下代码完成对表的初始化。

```
100-MAIN.
   READ   INPUT-WEEK-FILE.
   PERFORM   200-PROCESS-RECORD
             VARYING DAY-SUB   FROM 1 BY 1
             UNTIL    DAY-SUB > 7.
......
200-PROCESS-RECORD.
```

```
    MOVE  IN-DATA   TO   DAYS  (DAY-SUB).
    READ  INPUT.WEEK.DATA.
```

这里所使用的初始化方式便是输入文件载入方式。该方式的特点是将初始化数据保存在文件中。对表进行初始化时，先要将文件载入到程序中。载入文件数据后，再通过 PERFORM VARYING 语句依次对表中各项数据进行赋值，从而完成对表的初始化。

总之，对于数据量小，并且数据不是经常用到的情况下，数据多为临时数据。因此通常采用硬性编码方式初始化表；对于数据量大、数据需要经常用到的情况下，数据应该保存在文件上。因此通常采用输入文件载入方式初始化表。

9.5.3 对表初始化的一些灵活技巧

掌握对表初始化的一些灵活技巧，对于实际编码将带来一定的方便之处。下面分别介绍两种常用的灵活技巧。

（1）直接在 01 级数据项后对整张表进行初始化，方式如下。

```
01  SAMPLE-TABLE-ONE           VALUE 'ABCDE'.
    05   SAMPLE-ITEMS-ONE        PIC  X
                                 OCCURS 5 TIMES.
```

该方式和使用下面初始化代码实现的效果是一样的。

```
01  SAMPLE-DATA-ONE.
    05    FILLER   PIC  X  VALUE  'A'.
    05             PIC  X  VALUE  'B'.
    05             PIC  X  VALUE  'C'.
    05             PIC  X  VALUE  'D'.
    05             PIC  X  VALUE  'E'.
01   SAMPLE-TABLE-ONE    REDEFINES  SAMPLE-DATA-ONE.
    05    SAMPLE-ITEMS-ONE    PIC X  OCCURS 5 TIMES.
```

（2）直接在数据项后对其进行初始化，方式如下。

```
01  SAMPLE-TABLE-TWO.
    05    SAMPLE-ENTRY-TWO    OCCURS 2 TIMES.
          10    SAMPLE-ITEM-NUM    PIC  99  VALUE  00.
          10    SAMPLE-ITEM-DATA   PIC  XX  VALUE  'AA'.
```

该方式和使用下面初始化代码实现的效果是一样的。

```
01  SAMPLE-DATA-TWO.
    05    FILLER   PIC 99   VALUE  00.
    05             PIC XX   VALUE  'AA'.
    05             PIC 99   VALUE  00.
    05             PIC XX   VALUE  'AA'.
01  SAMPLE-TABLE-TWO    REDEFINES  SAMPLE-DATA-TWO.
    05    SAMPLE-ENTRY-TWO    OCCURS 2 TIMES.
           10    SAMPLE-ITEM-NUM   PIC 99.
           10    SAMPLE-ITEM-DATA  PIC XX.
```

9.6 直接查找方式

对表进行初始化后，就可以对表中数据进行查找操作了。通常的查找方式共有 3 种，即直接查找方式、顺序查找方式和二分查找方式。本小节介绍直接查找方式。直接查找方式是

COBOL 中基于表的 3 种典型查找方式中最简单的一种。基于直接查找方式的下标表最本质的特征就是表中数据的编号和其所存储的位置是相同的。

9.6.1 如何定义用于直接查找的表

在使用表进行数据查找时，首先应明确数据编号这个概念。数据编号是由人为指定的，用于代表相应数据的一种编号。编号通常为数字形式，也可看作一项数据，并不等同于下标。

用于直接查找的表中数据的编号和存储位置是相同的，因此在定义该表中不必指定数据编号。数据的编号直接由其所在位置确定。因此，定义用于直接查找的表时往往只用包含一个条目，即数据的条目。例如，对于一个大学里全体人员的管理系统，可以简单地将各人员的数据定义如下。

```
01  UNI-ROLE-TABLE.
    05  ROLE         PIC X(10)                  /*大学里各人员对应的角色*/
                      OCCURES 8 TIMES.           /*共有 8 种不同的角色*/
```

在实际应用中，还需要将该表进行初始化。这里可以通过上节所讲到的任意一种初始化表的方法进行初始化。即硬性编码方式或输入载入方式。假设没有对应的输入加载数据文件，则可以通过硬性编码方式初始化表，代码如下。

```
01  UNI-ROLE-VALUES.
    05  FILLER      PIC X(10)    VALUE  'FRESHMAN'.
    05               PIC X(10)    VALUE  'SOPHOMORE'.
    05               PIC X(10)    VALUE  'JUNIOR'.
    05               PIC X(10)    VALUE  'SENIOR'.
    05               PIC X(10)    VALUE  'MASTER'.
    05               PIC X(10)    VALUE  'DOCTOR'.
    05               PIC X(10)    VALUE  'FACULTY'.
    05               PIC X(10)    VALUE  'STAFF'.
01  UNI-ROLE-TABLE  REDEFINES  UNI-ROLE-VALUES.
    05   ROLE      PIC X(10)     OCCURS 8 TIMES.
```

初始化后的表结构如表 9.2 所示。

表 9.2 大学全体人员表

Freshman	Master
Sophomore	Doctor
Junior	Faculty
Senior	Staff

以上表格所包含的内容，从上到下依次为：大一学生、大二学生、大三学生、大四学生、研究生、博士生、全体教员和全体职员。其中每项数据所在表中的编号和其存储位置都是一一对应的，并且编号从数字 1 开始。例如，大一学生对应的编号为 1，大二学生对应的编号为 2，等等。

9.6.2 如何进行直接查找

对于上例所定义的表，可以通过默认的编号对数据进行直接查找。例如，当需要访问研究生这一项数据时，可依照以下步骤进行。

- 首先，找到表中数据的存储位置，得到该数据对应的编号。对于本例而言，研究生对应于“Master”数据项，在表中存储位置为 5。因此，该数据的编号即为 5。
- 其次，将编号移动到数据条目的下标中，得到 ROLE(5)。
- 最后，通过 ROLE（5）直接访问该表中的研究生数据。

总之，对于用于直接查找方式的表，知道了表中数据的存储位置，就可以得到该数据的编号。反之，若知道一个数据在表中的编号，就知道了该数据的存储位置。

9.6.3 对查找数据的处理

在 COBOL 的实际开发中，访问哪项数据往往并不是由人为指定的。实际需要访问表中的哪项数据，一般情况下是通过第 5 章所讲到流程控制中的条件判断指定的。对于本例而言，在实际开发中常常还需要指定以下两个变量。

- ROLE-CODE：用于保存在条件判断过程中数据的编号，此处同其下标一致。
- DEAL-ROLE：用于保存完成条件判断后得到的需要处理数据的编号。

这样，在完成条件判断后，最终对数据进行处理的 COBOL 代码如下。

```
IF  ROLE-CODE > 0 AND ROLE-CODE < 8
   MOVE ROLE  (ROLE-CODE) TO DEAL-ROLE
ELSE
   MOVE 'Y' TO ERROR-FLAG
END-IF
```

通过该段代码可以看到，这里主要是用 ROLE-CODE 这一变量取代了人为直接指定的数字。对于 ROLE-CODE 变量究竟保存哪一个数字，是通过之前的条件判断得到的。同时，ROLE-CODE 保存的数字即表中需要处理数据的编号。由于该表中数据的编号是从 1～8，因此，在最后往往还应加上一条数据合法性检测语句。该检测语句为以下这条。

```
IF  ROLE-CODE > 0 AND ROLE-CODE < 9
```

这样，如果经过条件判断后得到的最终处理数据的编号在表的范围中，将会对其进行处理。处理方式为将该编号从 ROLE-CODE 变量中移动到 DEAL-CODE 变量中。对应的代码为以下这条。

```
MOVE ROLE (ROLE-CODE) TO DEAL-ROLE
```

当该编号不在表的范围中，将报错。处理方式为将错误标志 ERROR-FLAG 置为“Y”。对应的代码为以下这条。

```
 MOVE 'Y' TO ERROR-FLAG
```

综上所述，对查找数据进行处理主要在于以下两点。

- 使用变量表示下标，而非指定具体数值，使其能够通过程序中的条件判断指定查找哪一项数据。
- 对通过条件判断后得到的数据编号进行合法性检测。

9.6.4 直接查找方式的适用范围

基于直接查找方式的表的特性就是表中数据的编号和存储位置是一一对应的。基于该项特性，直接查找方式有下面两点优势。一方面，基于直接查找方式的表不用另行存储数据编号，节省程序空间；另一方面，直接查找方式是通过编号对数据进行直接查找的，提高了查找效率。

在实际的数据处理中有时还会遇到这样一种情况。即如果使用直接查找方式，需要定义大量的数据，而真正用到的却只占很小一部分。

例如，对于一个网上电子商务的用户名，假设长度为 5 位，并且只允许为英文字母。如果通过直接查找方式进行处理，所需定义的数据共为 26 的 5 次方，即 1200 万个之多。而实际注册或者容许注册的用户，可能仅为 100 万个左右。因此，有大量的数据虽然定义了，但并未被使用。此时，虽然也可以通过直接查找方式完成处理，但将浪费大量的存储空间，不被提倡。

基于直接查找方式以上的优缺点，该方式的适用范围需要满足以下两点。

- 数据编号同数据存放位置一致。
- 数据使用频率平均，不致浪费大量存储空间。

其中第一条为必要条件，是必须要满足的。第二条为非必要条件，但若不满足，则会造成很大的弊端。对于不满足以上两点的情况，可以通过顺序查找方式或二分查找方式进行处理。关于这两类查找方式，将在下面的小节里依次介绍。

9.7 顺序查找方式

当数据编号同数据存放位置不一致时，就不能使用直接查找方式了。这时通常可以使用顺序查找方式或二分查找方式。本小节主要介绍顺序查找方式。

顺序查找方式实际上是最通用的查找方式，因为它所需满足的条件最少。可以说，任何表都可以通过顺序查找方式查找数据。

9.7.1 如何进行顺序查找

下面以一个银行账户管理系统为例子，具体分析在通常情况下是如何进行顺序查找的。对于银行账户管理系统而言，最主要的数据是以下两条。

- 账号
- 账号所对应的用户名

下面仍然针对顺序查找提取一个简化的模型。该模型只包含以上两类数据，并且假设用户数为 10 人。对此，可以定义银行账户管理表如下。

```
01  BANK-ACCOUNT-TABLE.
05  ACCOUNT    OCCURS  10  TIMES.
    10   ACCOUNT-NUM     PIC 9(9).       /*定义账号。并假设其由 9 个数字组成*/
    10   ACCOUNT-NAME    PIC X(10).      /*定义用户名。并假设其由不超过 10 个的字符组成*/
```

将该表通过任意一种方式初始化后，假设表中数据如表 9.3 所示。

表 9.3 银行账户表

ACCOUNT-NUM	ACCOUNT-NAME	ACCOUNT-NUM	ACCOUNT-NAME
021845984	TOM	032874512	ROBIN
547874956	HARRY	064840574	SIMON
201874698	PITTER	120873954	FRANK
004847624	SMITH	657842216	SANDY
478206045	JONES	689-574-402	SUE

以上表格中的数据都是任意指定的。另外，这里需要再强调一下，数据编号和下标并不是一个概念。对于本例而言，数据编号就是账号 ACCOUNT-NUM。该数据编号用以代表相应数据，即用户名 ACCOUNT-NAME。数据通常是有具体意义的，如 ACCOUNT-NAME 数据表示人名。而编号则是没有具体意义的，只是为了更方便地表示数据。

通过上表可以看出，此处数据编号和数据存放的位置并不是一致的。为更好地说明这一点，银行账户表中的数据编号，数据以及数据存放位置的对应关系如表 9.4 所示。

表 9.4　　数据编号，数据和数据存放位置的对应关系

数 据 编 号	数　　据	数据存放位置
021845984	TOM	1
547874956	HARRY	2
201874698	PITTER	3
004847624	SMITH	4
478206045	JONES	5
032874512	ROBIN	6
064840574	SIMON	7
120873954	FRANK	8
657842216	SANDY	9
689574402	SUE	10

对于该表而言，假设需要查找数据编号（即账号）为“064840574”的数据（即用户名）。则使用顺序查找方式的代码如下。

```
……
05   ACCOUNT-SUB     PIC 99  USAGE IS COMP.      /*定义下标*/
05   FOUND-FLAG      PIC X.                      /*定义找到标志*/
05   KEY-NUM         PIC 9(9).                   /*该变量用于存放指定的数据编号*/
05   FOUND-NAME      PIC X(10).                  /*该变量用于存放查找到的数据*/
……
MOVE 'N' TO FOUND-FLAG.                          /*将找到标志设为默认值'N'*/
MOVE '064840574' TO KEY-NUM.                     /*将指定的数据编号 MOVE 到 KEY-NUM 变量中*/
PERFORM  100-FIND-NAME
         VARYING ACCOUNT-SUB FROM 1 BY 1
         UNTIL   ACCOUNT-SUB > 10
         OR
         FOUND-FLAG = 'Y'.
```

以上是通过 PERFORM VARYING 语句从表的第一条数据开始，一条条顺次查找的。因此，这也是该查找方式被称为顺序查找方式的原因。

对于上面的 100-FIND-NAME 具体每一步的查找过程，代码如下。

```
100-FIND-NAME.
     IF  ACCOUNT-NUM  (ACCOUNT-SUB) = KEY-NUM
         MOVE 'Y' TO FOUND-FLAG
         MOVE ACCOUNT-NAME  (ACCOUNT-SUB) TO FOUND-NAME
     END-IF.
```

通过顺序查找方式查找后，最终找到的数据应该为“SIMON”。整个查找过程共进行了 7 次查找，查找过的数据如表 9.5 所示。

表 9.5　　查找过的数据

ACCOUNT-NUM	ACCOUNT-NAME	ACCOUNT-NUM	ACCOUNT-NAME
021845984	TOM	478206045	JONES
547874956	HARRY	032874512	ROBIN
201874698	PITTER	064840574	SIMON
004847624	SMITH		

上面最后一条记录中的数据“Simon”即为所要查找的数据。

此外，整个查找结束后，下标 ACCOUNT-SUB 也应该为 7。数据的存放位置实际上是和下标对应的，而不是数据的编号。数据编号和数据都要通过下标引用。使用下标引用数据编号的方式如下。

```
ACCOUNT-NUM  (ACCOUNT-SUB)
```

使用下标引用数据的方式如下。

```
ACCOUNT-NAME  (ACCOUNT-SUB)
```

9.7.2　使用顺序查找方式的注意事项

从以上顺序查找方式可以看出，该方式实际上是从表的第一条记录开始查找起。一条一条地顺次查找，直到找到所要求的数据项为止。因此，该查找方式平均查找次数的计算公式如下。

```
顺序查找方式平均查找次数 = 表中所有数据数量 / 2
```

为提高顺序查找方式的效率，通常是将常用数据尽量放在靠近表头的位置。因为顺序查找方式是从表头开始查找的，因此越放在靠近表头的位置，越容易被找到。这样，查找效率从整体上会得到提高，平均查找次数在实际中将小于上面公式的运算结果。

此外，在数据的查找过程中通常还存在一个“2/8 原理”。也就是说 80%用到的数据通常只占总量的 20%；而剩下 80%的数据使用率还不到 20%。因此，根据“2/8 原理”动态分配表中数据的存放位置，将显著提高顺序查找的效率。

9.8　二分查找方式

无论是直接查找方式，还是顺序查找方式，其中涉及到的算法都比较简单。而二分查找方式，主要建立在二分查找算法（也有书称作折半查找）的基础上。因此，二分查找方式是 3 种查找方式中最难的一种，故而放在最后进行讲解。二分查找概括地说主要有两大特点。

- 二分查找对应的表中数据排列方式必须有序。
- 二分查找从表的中间开始查找。

以下分别就可用于二分查找的表的特征，如何进行二分查找以及二分查找的好处进行详细讲解。其中，如何进行二分查找涵盖了二分查找的算法，是本小节的重点。

9.8.1　可用于二分查找的表的特征

不同于顺序查找方式，二分查找方式对表中数据的位置有着严格的要求。表中数据的排

列必须有序，可以为升序，也可以为降序。

下面，以学生成绩管理系统为例，提取简化模型，定义表的结构如下。

```
01  STUDENT-REPORT-TABLE.
    05   REPORT   OCCURS  8  TIMES.                    /*设共有 8 门课的成绩*/
           10   SUBJECT-CODE         PIC 9            /*课程编号*/
           10   SUBJECT-NAME         PIC X(10)        /*课程名称*/
           10   MARK                 PIC 99           /*分数*/
```

对应于以上定义的表，显然 SUBJECT-CODE 应作为数据编号。SUBJECT-NAME 和 MARK 都是该数据编号对应的一组相关数据。

为了能够进行二分查找，表中数据排列必须有序。这里所说的有序，实际上指的应该是表中数据编号要么从大到小，要么从小到大。因为单纯的数据，如考试科目名称 SUBJECT-NAME，是无所谓大小的，也就无法进行排序。

定义完成之后，仍然需要对表初始化。此时，便不能同顺序查找方式一样，对表中数据任意进行初始化了。为实现二分查找，数据编号 SUBJECT-CODE 必须有序。因此，初始化后的表其内部数据通常如表 9.6 所示。

表 9.6　　学生成绩表 1

SUBJECT-CODE	SUBJECT-NAME	MARK
1	Politics	88
2	Math	93
3	Chinese	86
4	English	78
5	Physics	73
6	Chemistry	75
7	Biology	82
8	Geography	85

这里，课程编号是 SUBJECT-CODE 从 1~8 是有序的。由于本表中课程编号即为数据编号，数据编号的有序即表明数据的有序。因此，该表可以用于二分查找。

可以看到，以上初始化后的表，表中数据为升序排列。由于对于二分查找方式而言，只要表中数据有序排列就可以了，而不论升序还是降序。因此，以下将初始化数据进行降序排列的方式也是正确的。表内数据如表 9.7 所示。

表 9.7　　学生成绩表 2

SUBJECT-CODE	SUBJECT-NAME	MARK
8	Politics	88
7	Math	93
6	Chinese	86
5	English	78
4	Physics	73
3	Chemistry	75
2	Biology	82
1	Geography	85

此外，以上两张表将数据编号都设为了一组连续数字。在没有特别要求的情况下，通常为方便起见，数据编号最好指定为连续数字。当然，根据用于二分查找方式表的要求说明，数据只要有序就行了，而不一定非要连续。下面这组数据按照习惯虽不常用，但仍然是正确的。相应数据如表 9.8 所示。

表 9.8　　学生成绩表 3

SUBJECT-CODE	SUBJECT-NAME	MARK
2	Politics	88
5	Math	93
8	Chinese	86
10	English	78
12	Physics	73
15	Chemistry	75
21	Biology	82
34	Geography	85

9.8.2　如何进行二分查找方式

二分查找方式最大的特点是从表的中间开始查找，这样能大幅提高查找效率。二分查找方式是一种基于算法的高效查找方式。以下结合上面学生成绩管理的例子，看看二分查找是如何进行的。

为便于分析，以表 9.6 中的数据作为例进行讲解。对于表 9.6 而言，若要查找课程编号为 7 所对应的课程名称及该学生的分数，通常由以下几步进行。

（1）在查找之前，要知道整张表的大小。即对表要有一个通篇的浏览。整张表的数据内容如下。

SUBJECT-CODE	SUBJECT-NAME	MARK
1	Politics	88
2	Math	93
3	Chinese	86
4	English	78
5	Physics	73
6	Chemistry	75
7	Biology	82
8	Geography	85

（2）从上表的中间位置开始查找。即从数据编号为 4 的位置查找。比较 4 和所要查找的编号 7 的大小，显然 7 比 4 大。由于表中数据编号是顺序编排的，并且为升序，因此编号为 4 及其以上的编号都比 7 要小。因此，可以略去编号为 4 及其以上的半张表不看，只对下半张表进行查找。剩下需要关注的半张表如下。

5	Physics	73
6	Chemistry	75
7	Biology	82
8	Geography	85

（3）将剩下的这半张表当作一张新的表，仍然按照上面的方式处理。这样，该表从 6 以上的半张表又将被略去，剩下的半张表如下。

7	Biology	82
8	Geography	85

（4）再将上面的表一分为二。发现上半张表仅剩一条记录，具体如下所示。

7	Biology	82

而这正好就是我们要找的数据。通过该条记录可以知道，课程编号为 7 所对应的课程名称为 Biology，该生该门课程的考试分数为 82 分。

以上具体分析了使用二分查找法是如何查找表中数据的。通过上例可以看到，二分查找实际上就是每次将一张表从中间一分为二。通过将所需查找的数据编号和表正中间的数据编号进行比较，略去不可能存在该数据的半张表。这样，每次查找后，数据都将减少一半，大大提高了查找效率。

需要提到的一点是，对于整张表具有偶数条数据项而言，二分查找所选取的中间为总数据量除以 2。比如，上面那张表最初共有 8 条记录。每次进行查找时所比较的数据编号分别如下。

- 第一次查找将 8 除以 2，得到 4。比较该表中第 4 条记录，也即编号为 4 的记录。
- 第二次查找时只剩 4 条记录。将 4 除以 2，得到 2。比较该表的第 2 条记录，即编号为 6 的记录。
- 第三次查找时只剩 2 条记录。将 2 除以 1，得到 1。比较该表的第 1 条记录，即编号为 7 的记录。由于正好要求查找编号为 7 的相关数据，成功找到，查找结束。

但对于整张表具有奇数条数据项的表而言，将总数据量除以 2 将得到一个小数。比如若整张表有 7 条记录，将 7 除以 2 得到 3.5，但没有第 3.5 条记录这么一说。此时，既可以对第 3 条记录进行比较，也可以对第 4 条记录进行比较，效果是类似的。具体和那条比较比较，取决于内部的算法。

上面仅从直观上分析了二分查找是如何进行的，并没有说明在机器内部是如何实现的。下面，给出二分查找的伪代码，以了解其算法思想。

以下代码表示在可用于二分查找的表 BST 中查找数据编号为 key 的数据。若找到，函数返回该数据在表中的位置，否则返回 0。代码如下。

```
int Search.Binary (BSTable BST, KeyNum key )
{
low=1;
high=BST.length;
while(low <= high )
{
 mid = (low + high) / 2 ;
 if EQ (key, BST.elem[mid].key)
              return mid ;
else if   LT(key , ST.elem[mid].key)
              high = mid - 1 ;
else     low = mid + 1 ;
}
return 0 ;
}
```

此段代码为伪代码，只用于表示算法过程，并不能实现具体的功能。若要使用 COBOL 编写该段代码，实现具体的功能，则难度比较大。由于本书是 COBOL 的入门教材，因此不去深究这个问题。有兴趣的读者朋友可以自己尝试编写。

最后，虽然本书并不要求使用 COBOL 编写实现二分查找功能的代码，但该算法思想是要掌握的。并且，在后面要讲到的用 SEARCH ALL 语句查找索引表，正是使用的二分查找方式。

9.8.3 二分查找方式的好处

二分查找方式最大的好处，就是查找效率高。特别是在表中数据量很大的时候，这点优势更加明显。通常，随着表中数据量越大，二分查找方式相对于顺序查找方式的比较优势就越大。下面，假设一张表的数据共有 100 条，具体看看二分查找方式的优势是如何体现出来的。

如果采用顺序查找方式，最坏的情况下，是查找第 100 条数据。因此，若采用顺序查找方式，最大查找次数为 100。若不考虑极端情况，只计算该查找方式下的平均查找次数，对应公式如下。

```
顺序查找方式平均查找次数 = 表中所有数据数量 / 2
```

根据公式，可知其平均查找次数为 100/2=50 次。若采用二分查找方式，在最坏的情况下，查找次数也远远小于顺序查找方式的平均查找次数。具体查找过程如下。

- 第 1 次查找，将 100 一分为二，比较第 50 条数据，并将拥有 50 条数据的半张表作为新表。
- 第 2 次查找，将 50 一分为二，比较新表中的第 25 条数据，将拥有 25 条数据的半张表作为新表。
- 第 3 次查找，将 25 一分为二，比较新表中的第 13 条数据，将含 12 条数据的半张表作为新表。
- 第 4 次查找，将 12 一分为二，比较新表中的第 6 条数据，将含 6 条数据的半张表作为新表。
- 第 5 次查找，将 6 一分为二，比较第 3 条数据，将含 3 条数据的半张表作为新表。
- 第 6 次查找，将 3 一分为二，比较新表中的第 2 条数据，将含 1 条数据的半张表作为新表。
- 第 7 次查找，比较新表中的唯一 1 条数据，若满足查找条件，查找成功，否则查找失败。完成整张表的查找。

因此，可以看到，即使在最坏情况下，二分查找总共查找次数也只有 7 次。相比顺序查找最坏情况下的 100 次和平均情况下的 50 次，查找次数大大减少了，效率相应大大提高。

9.9 3 种查找方式的比较和总结

上面 3 小节依次介绍了直接查找方式、顺序查找方式和二分查找方式。这里对以上 3 种查找方式作一个简单的回顾和总结。

9.9.1 对表的要求

直接查找方式要求表中数据的编号和数据存储位置必须一致。用于直接查找方式的表，可不指定数据编号项。数据编号隐含在数据在表中的存储位置。

顺序查找方式对表中数据没有特别的要求。由于该方式下表中数据编号和数据存储位置往往并不一致，因此，表中按常规必须包括数据编号。

二分查找方式要求表中数据排列必须有序。实际上，就是指表中的数据编号必须单调递增或单调递减，即要么为升序，要么为降序。

9.9.2 具体查找过程

以上 3 种查找方式实际是根据各自不同的查找过程进行区分的。下面对这 3 种查找方式的查找过程分别予以讲解。

1．直接查找方式

直接查找方式就是通过数据编号直接对表中数据进行查找。具体做法就是将数据编号的值 MOVE 到下标中，再通过下标引用所要查找的数据。此外，由于下标是直接给出的，因此最后还要对其进行合法性检测。实现代码如下。

```
MOVE key TO sub.
IF  sub > 0 AND sub < total                          /*对下标进行合法性检测*/
   MOVE data  (sub)  TO  found-data
ELSE
   MOVE 'Y' TO ERROR-FLAG
END-IF
```

其中 key 指要查找的数据编号，sub 指下标，data 是数据条目名称，total 指表中数据总量。found-data 变量保存查找到的数据。

2．顺序查找方式

顺序查找方式就是从表的第一条数据开始，顺次往下进行查找，直到找到所需查找数据为止。若直到表中最后一条数据仍没有符号要求的，则查找失败。顺序查找方式通常要结合 PERFORM VARYING 语句完成。实现代码如下。

```
PERFORM  find-process
        VARYING  sub FROM 1 BY 1
        UNTIL    sub > total
        OR
        FOUND-FLAG = 'Y'.
……
find-process.
        IF  data-num  (sub) = key
            MOVE 'Y' TO FOUND-FLAG
            MOVE data  (sub) TO found-data
        END-IF.
```

其中，find-process 为每一步查找过程的名称。data-num 为表中数据编号条目名称，并通过下标引用具体的数据编号。引用形式为 data-num (sub)。其余名称对应的变量意义和上面直接查找的代码中对应的是一样。

3．二分查找方式

二分查找方式就是每次从表的中间数据开始查找起。每查找一次，将忽略掉一半数据，

只在剩下的一半中继续查找。二分查找方式实际上是用到了递归的算法。

关于二分查找方式的具体实现过程，本书不要求用 COBOL 语言来编写相应代码。但二分查找的算法思想是必须要掌握的。并且，在后面要讲到的用 SEARCH ALL 语句查找索引表，就是依据二分查找的算法进行的。

9.9.3 查找效率

对于直接查找方式而言，知道了数据编号也就知道了数据存储位置。因此，查找次数都仅为 1 次。但实际上这种方式下并没进行实际的查找算法，因此不能从算法的角度分析其查找效率。

对于顺序查找方式，这里分别从最好情况、最坏情况和平均情况分析其查找效率。假设表中总数据量为 *n*。

- 最好情况下，表中第一条数据即为所要查找的数据，查找次数为 1。
- 最坏情况下，表中最后一条数据才为所要查找数据，查找次数为 *n*。
- 平均情况下，根据以下公式，查找次数为 *n*/2。

```
顺序查找方式平均查找次数 = 表中所有数据数量 / 2
```

对于二分查找方式，这里仅从最好情况和最坏情况下分析其查找效率。对于二分查找平均情况下的查找效率，这里暂且不作讨论。仍然假设表中总数据量为 *n*。

- 最好情况下，表中中间数据即为所要查找的数据，查找次数为 1。
- 最坏情况下，表中第一条或最后一条数据才为所要查找的数据。则对应的查找次数为对 *n* 开方，取整后再加 1。

因此可以看出，从算法的角度看，二分查找方式是最高效的。并且，表中数据量 *n* 越大，使用二分查找方式相对于顺序查找方式而言效率越高。

9.9.4 查找方式小结

以下综合 3 种查找方式对表的要求，各自的查找过程以及查找效率进行总结。关于这 3 种查找方式的总结如表 9.9 所示。

表 9.9　　3 种查找方式的总结

	对表的要求	查 找 过 程	查 找 效 率
直接查找方式	表中数据编号同数据存放的位置要求一致。可省略定义数据编号	将数据编号 MOVE 到下标中，通过下标引用数据	每次查找次数均为 1 次 （不能从算法角度分析其查找效率。）
顺序查找方式	无特别要求	从表中第一条数据开始查找，顺次查找整张表，直到找到数据或查找到表中最后一条数据	最好情况下查找 1 次 最坏情况下查找 *n* 次 （*n* 为表中数据总量，下同。） 平均情况下查找 *n*/2 次
二分查找方式	表中数据必须有序排列。可以为升序，也可以为降序	从表中间的数据开始查找，每次查找后将查找数据量缩小一半，通过递归算法实现	最好情况下查找 1 次 最坏情况下查找次数为对 *n* 开平方，取整，再加 1

9.10 对表中数据的统计计算

以上谈到了如何对表中数据进行查找。对表中数据进行查找是使用表操作最重要的用途。此外，有时还需对表中数据进行统计计算。统计计算中最常见的是计算表中数据总和，求取平均数和求取中位数。下面分别予以讲解。

9.10.1 计算数据总和

这里不妨以二分查找方式中的学生成绩表为原始数据分别进行讲解。该原始数据如表9.10所示。

表 9.10 **学生成绩表 1**

SUBJECT-CODE	SUBJECT-NAME	MARK
1	Politics	88
2	Math	93
3	Chinese	86
4	English	78
5	Physics	73
6	Chemistry	75
7	Biology	82
8	Geography	85

其中对于表中各数据的定义如下。

```
01  STUDENT-REPORT-TABLE.
   05   REPORT   OCCURS  8  TIMES.
            10   SUBJECT-CODE        PIC 9。
            10   SUBJECT-NAME        PIC X(10).
            10   MARK                PIC 99.
```

对于上表，若要求取这位学生各门功课的总成绩，就涉及到了对表中数据的一个统计计算。通常，可以有两种方法实现这一功能。

第一种是不使用 FUNCTION 子句，直接进行计算。代码如下。

```
COMPUTE  TOTAL-MARK  =
        MARK  (1) + MARK  (2)  + MARK  (3)  +
        MARK  (4) + MARK  (5)  + MARK  (6)  +
        MARK  (7) + MARK  (8).
```

对于这种方法，显然是非常费时费力的。并且，上表仅有 8 条数据，尚可以通过这种方法实现。但若对于有 100 条或更多数据的表而言，若要计算其数据总和，则要写 100 多个数据和加号。这显然是不现实的。

第二种方法就是使用 FUNCTION 子句来计算。代码如下。

```
COMPUTER  TOTAL-MARK =
        FUNCTION SUM (MARK (ALL)).
```

这里使用的是 FUNCTION 子句中的求和语句 COMPUTER FUNCTION SUM。SUM 即代表总和的意思。同时，需要在数据条目名称 MARK 后加上“（ALL）”，表示是对表中全体数

据进行的计算。这种方法求得的结果和上面直接计算的结果是一样的，但显然要简便得多。

9.10.2 计算平均数

仍然是针对上面的学生成绩表，这次要求取该生各门功课的平均成绩。依然分两种情况进行讨论，即不使用 FUNCTION 子句的情况和使用该语句的情况。若不使用 FUNCTION 子句，仍然可以直接进行计算，代码如下。

```
COMPUTE  AVERAGE-MARK  =
       ( MARK  (1) + MARK  (2)  + MARK  (3)  +
       MARK  (4) + MARK  (5)  + MARK  (6)  +
       MARK  (7) + MARK  (8) ) / 8.
```

这种方法同计算数据总和时用到的直接计算方法是一样费时费力的。并且，当数据量较大时，同样在实际操作中无法实现。

若使用 FUNCTION 子句，则代码如下。

```
COMPUTER AVERAGE-MARK =
       FUNCTION MEAN (MARK (ALL)).
```

这里使用的是 FUNCTION 子句中的求平均数语句 COMPUTER FUNCTION MEAN。MEAN 在这里代表平均数的意思。同时，仍然需要在数据条目名称 MARK 后加上“（ALL）”，以表示是对表中全体数据进行计算。同样，这种方法求得的结果和上面使用直接计算得到的结果仍然是一样的，但显然要简便得多。

9.10.3 计算中位数

所谓中位数，就是一列数据从小到大排列正中间的数据。对于一列数据中的中位数而言，比它大的数和比它小的数应该是一样多的。

这里，若要求取该生所有考试成绩的中位数，仍然可以分两种方法进行。一种是不使用 FUNCTION 子句，另一种是使用 FUNCTION 子句。对于不使用 FUNCTION 子句的方法而言，涉及到较复杂的算法设计，此处不加讨论。

下面，只看使用 FUNCTION 子句如何完成操作。若使用 FUCNTION 子句，代码如下。

```
COMPUTER MID-MARK =
       FUNCTION MEDIAN (MARK (ALL)).
```

这里使用的是 FUNCTION 子句中的求中位数语句 COMPUTER FUNCTION MEDIAN。MEDIAN 在这里代表中位数的意思。对于该语句而言，仍需在数据条目名称 MARK 后加上“（ALL）”，以表示对全体数据进行计算。

此外，对于前面讲到的二分查找而言，原始数据的编号也可不连续。数据编号不连续的情况，如表 9.11 所示。

表 9.11　　学生成绩表 2

SUBJECT-CODE	SUBJECT-NAME	MARK
2	Politics	88
5	Math	93
8	Chinese	86

续表

SUBJECT-CODE	SUBJECT-NAME	MARK
10	English	78
12	Physics	73
15	Chemistry	75
21	Biology	82
34	Geography	85

SUBJECT-CODE 为数据编号，但只是有序排列，并不连续。根据二分查找的方式，需要从表的中间数据项开始查找。此时若要找到表的中间数据，就不能使用类似连续编号情况下的方式进行。否则，若根据通常连续编号情况下的方式进行，将得到以下错误结果。

```
2+(34-2)/8=6
```

实际上，对于该表，中间数据编号应该为 10 或者 12（根据具体算法细节不同而不同）。这里用求取中位数的方法便可以很方便地得到正确结果，代码如下。

```
COMPUTER MID-CODE =
       FUNCTION MEDIAN (SUBJECT (ALL)).
```

9.10.4 统计计算小结

通过前面的例子可以看到，对表中数据进行统计计算的代码主体结构基本相似。主要都是使用的 COMPUTER 语句中的 FUNCTION 子句完成的。基本结构如下。

```
COMPUTER required-data=
         FUNCTION option (data (ALL)).
```

其中“required-data”表示要计算的统计数据类型，如总和、平均数或是中位数。“option”指相应不同的统计数据的不同选项。“data”指表中数据条目的名称。对表的统计计算最常见的是求以下 3 种数据：总和、平均数、中位数。

计算表中数据总和的代码如下。

```
COMPUTER required-data=
         FUNCTION SUM (data (ALL)).
```

计算表中数据平均数的代码如下。

```
COMPUTER required-data=
         FUNCTION MEAN (data (ALL)).
```

计算表中数据中位数的代码如下。

```
COMPUTER required-data=
         FUNCTION MEDIAN (data (ALL)).
```

以上各段代码和第 4 章中讲到的 COMPUTER FUNCTION 子句的 3 种功能代码是类似的。对于表的统计计算而言，关键在于处理的数据对象为表中数据条目名称其后再加上“（ALL）”参数。形式如下。

```
data (ALL)
```

9.11 索引表

上节中讲到的下标表是通过下标来管理表中数据的。与之对应，通过索引来管理数据的表称作索引表。索引表相对比较特殊。若某一张表为索引表，则必须特别指明。以下分别从各方面对索引表进行详细讲解。

9.11.1 为何要使用索引表

索引表指的是表中数据通过索引索引来管理的表。学习索引表，必须建立在学习下标表的基础之上。对于索引表中所涉及到的和下标表类似的知识，如 OCCURS 语句的用法，二分查找的概念等，本节将不再赘述。

使用索引表的目的除了包含使用表的用途外，主要是用于更高效地对数据进行查找。具体表现在以下两个方面。

- 提供更高效的目标代码，即索引。程序在实际运行中对索引的使用效率要远高于对下标变量的使用效率。
- 提供更快捷的查找方式。在索引表中，可以使用 SEARCH 和 SEARCH ALL 语句对数据进行查找。这两种语句使用的查找方式是二分查找方式，故更加快捷。同时，若要使用 SEARCH 和 SEARCH ALL 语句对数据查找，必须针对索引表进行。

9.11.2 如何定义索引表

定义索引表，除了要指明表的名称、数据条目、重复次数外，还应包含一个索引变量。这里需要特别注意的是，索引变量只用指明，而不必进行定义。下面给出定义一个索引表的基本框架，代码如下。

```
01  sample-indexed-table.
   05   item-list    OCCURS  x1  TIMES
                       INDEXED BY  data-ndx.            /*此处指明索引变量*/
        10   item-num     PIC 9(x2).
        10   item-data     PIC X(x3).
```

这里，sample-indexed-table 为该索引表的名称。item-list 为表的一个列。由于表中涉及到两个相关数据 item-num 和 item-data，因此可以用列统一进行管理。item-num 为数据编号，item-data 为实际数据。x1、x2、x3 为任意自然数。

通过以上代码可以看到，索引表不同于其他表最大之处在于索引表包含有一个索引变量。索引变量在定义索引表时就应该指明，指明方式如下。

```
      INDEXED BY  data-ndx.
```

由此可见，索引表中的索引变量是通过 INDEXED BY 语句进行指明的。并且，该语句通常出现在 OCCURS 语句之后。INDEXED BY 语句后面出现的变量名 data-ndx 即为索引变量。索引变量不用另外进行定义。

9.11.3 索引的特点

理解索引表，最重要是要理解索引的概念。下面结合一个使用索引表定义的银行账户信

息，具体分析索引的特点。定义该索引表的代码如下。

```
01  BANK-ACCOUNT-TABLE.
    05    ACCOUNT-DATA    OCCURS  10  TIMES
                          INDEXED BY  ACC-NDX.
          10   ACCOUNT-NUM     PIC 9(9).
          10   ACCOUNT-NAME   PIC X(10).
```

以上代码中，ACC-NDX 即为该索引表的索引。对于 ACC-NDX 而言，主要有下面几个特点，或者说是需要注意的地方。

1．索引不用被直接定义

通过上面代码可以看到，被显示定义的只有两项数据，即 ACCOUNT-NUM 和 ACCOUNT-NAME。其中 ACCOUNT-NUM 为银行账户的号码，作为表中的数据编号。ACCOUNT-NAME 为该账户的姓名，是实际数据。

ACCOUNT-NUM 被定义为一个含有 9 位数字的整型数据，定义代码如下。

```
10   ACCOUNT-NUM     PIC 9(9).
```

ACCOUNT-NAME 被定义为一个含有 10 个字符的字符型数据，定义代码如下。

```
10   ACCOUNT-NAME   PIC X(10).
```

相比较而言，索引变量 ACC-NDX 则没有如上进行定义。ACC-NDX 既不像 ACCOUNT-NUM 那样被定义为一整型数据，也不像 ACCOUNT-NAME 那样被定义为一字符型数据。

既然没有被定义，那么 ACC-NDX 是否因此不占用内存空间呢？答案是否定的。实际上，ACC-NDX 是直接由 COBOL 系统本身存储在某一内存单元上的。ACC-NDX 是占用内存空间的，但作为程序员不用理会它存储在何处。

2．不能对索引使用 MOVE 赋值操作

索引作为一个特殊的数据，是不用在数据部 DATA DIVISION 中进行定义的。同时，它的特殊之处还表现在不能对其使用 MOVE 语句。

对于本例中的索引变量 ACC-NDX 而言，下面的代码是错误的。

```
MOVE  5  TO  ACC-NDX.                    ←错误的用法，索引不能用 MOVE 语句进行赋值操作
```

这段代码企图将整型数据赋值给索引 变量 ACC-NDX。然而，ACC-NDX 作为一个特殊的索引变量，是不允许对其使用 MOVE 语句进行赋值操作的。

同样，下面这段代码企图将一个变量中保存的数据赋值给 ACC-NDX，也是错误的。原因仍然是不能对索引变量使用 MOVE 语句。代码如下。

```
MOVE  TEST-DATA  TO  ACC-NDX.            ←错误的用法，索引不能用 MOVE 语句进行赋值操作
```

下面这段代码企图将 ACC-NDX 中的数据赋值给另外一个变量，同样也是错误的。代码如下。

```
MOVE   ACC-NDX   TO   PL-DATA.           ←错误的用法，索引不能用 MOVE 语句进行赋值操作
```

3．不能对索引使用算术运算符号语句

对索引变量 ACC-NDX 使用前面章节中讲到的算术运算也是不允许的。例如，下面几行

代码都错误地对 ACC-NDX 进行了算术运算操作。

```
ADD 3 TO ACC-NDX.
SUBTRACT DATA-SUB FROM ACC-NDX.
MULTIPLY ACC-NDX BY 5 GIVING RESULT-DATA.
COMPUTER ACC-NDX = (2+3) * 5.
……
```

那么，如何对索引变量进行赋值操作或算术运算呢？这就需要使用到 SET 语句。关于 SET 语句，将在后面章节进行详细讲解。

9.11.4 索引表的内部存储结构

索引表的内部存储结构相对于下标表而言要相对复杂一些。在下标表中，数据的存储位置即下标的数值。而在索引表，数据的存储位置和索引数值往往并不一致，需要计算得到。

对于索引表而言，其中的索引保存的是数据偏移量，而并不是数据实际存储位置。数据的实际存储位置，通过将索引变量除以每个数据的大小得到。

下面以一个公司一周内的营业额为基本模型，讨论索引表的内部存储结构。首先，对该表进行定义及初始化，代码如下。

```
01   WEEK-SALES-DATA.
     05  FILLER     PIC 9(5)    VALUE  '15874'.
     05             PIC 9(5)    VALUE  '10876'.
     05             PIC 9(5)    VALUE  '14586'.
     05             PIC 9(5)    VALUE  '13587'.
     05             PIC 9(5)    VALUE  '19784'.
     05             PIC 9(5)    VALUE  '10030'.
     05             PIC 9(5)    VALUE  '10088'.
01    WEEK-SALES-TABLE  REDEFINES  WEEK-SALES-DATA.
     05    SALES       PIC  9(5)
                       OCCURS   7  TIMES
                       INDEXED BY  SALE-NDX.
```

为方便讨论，这里没有定义数据编号，只包含一周内每天的营业额 SALES 这一个数据条目。以上代码实际上也是定义一个最小化索引表的代码。

假设这里要引用周五的营业额数据，代码如下。

```
SET  5   TO   SALE-NDX.
MOVE  SALES  (SALE-NDX)   TO  RESULT-DATA.
```

这里，通过使用 SET 语句，将 5 赋值给 SALE-NDX，以引用周五的营业额数据。但是，在内部存储结构里，SALE-NDX 实际存储的并不是数字 5，而是第 5 项数据的偏移量。

关于数据偏移量，也就是该数据相对于索引表中第一条数据的偏移位置量。简单的说，一个数据的偏移量就是该数据和索引表中第一条数据之间有多少个存储单元。以上周营业额索引表中索引的内部存储结构如表 9.12 所示。

表 9.12　　周营业额中的索引内部存储结构

实际对应的时间	营业额数据 SALES	索引变量 SALE-NDX
周一	15874	0
周二	10876	5

续表

实际对应的时间	营业额数据 SALES	索引变量 SALE-NDX
周三	14586	10
周四	13587	15
周五	19784	20
周六	10030	25
周日	10088	30

通过上表可以看到，对应周五数据的索引变量实际上并不是 5，而是 20。这是因为，根据以下定义语句：

```
05    SALES       PIC  9(5)
```

该表中每项数据都占用 5 个单位的存储单元。因此，第 5 条数据相对于第 1 条数据的偏移量为 20。

这里不妨假设一张索引表中每条数据占用 x 个存储单元。则第 n 条数据相应的索引在内部存储中的数值（即该数据的偏移量），可依据以下公式计算得到。

```
第 n 条数据的偏移量=（n-1）* x
```

当然，对于上例而言，虽然通过以下语句将 SALE-NDX 实际的值置为了（5−1）*5=20。

```
SET  5   TO   SALE-NDX.
```

但在程序中，SALE-NDX 的值仍然以数字 5 的形式出现。也就是说，程序员在编码中只需将其看作数值 5 来进行处理（虽然其实际数值为 20）。程序中的数值 5 和实际内存中的偏移量 20 是由 COBOL 系统本身进行转换的。

9.11.5 索引表和下标表的比较

为加深对索引表的理解，同时也对之前学习过的下标表作一个回顾，现将二者进行比较。索引表和下标表既有不同之处，也有相似之处，下面分别进行讲解。

1．索引表和下标表的不同之处

索引表和下标表主要有 4 点区别，以下分别进行讲解。

（1）定义方式不同。依据前面学过的知识，对于下标表，除定义表的基本结构外，还要定义一个下标。定义下标表的代码通常如下。

```
01  sample-subscripted-table.
   05  item-list         OCCURES x1 TIMES.
       10  item-num       PIC 9(x2).
       10  item-data      PIC X(x3).
   05  sub                PIC 9(x4)     USAGE IS COMP.    /*此处定义下标*/
```

对于索引表而言，则需要指明相应的索引。这里需要再次强调一下，索引表中的索引只用指明，不需要定义。索引的存储位置和存储空间大小是由 COBOL 系统本身管理的，程序员不必去理会。定义索引表的代码通常如下。

```
01  sample-indexed-table.
   05    item-list    OCCURS  x1  TIMES
                INDEXED BY  data-ndx.              /*此处是指明索引变量，不是定义*/
```

```
           10   item-num    PIC 9(x2).
           10   item-data    PIC X(x3).
```

（2）下标变量和索引变量类型不同。下标表是通过下标变量管理的，索引表是通过索引变量管理的。此二种变量类型的不同也是下标表和索引表的一个重要区别。

下标变量本身是一个整型数据变量。因此，同其他的整型数据变量一样，可对其使用MOVE 语句操作和运算符语句操作。

例如，以下代码都是正确的。

```
MOVE subscript TO data-field.
MOVE data-field TO subscript.
ADD 3 TO subscript.
COMPUTE subscript = (5+7) / 3.
……
```

因此，可以看出下标变量除用于引用下标表中数据时，还可用于其他操作。也就是说，可以将下标变量当作一个通常的整型数据变量进行任何相应的操作。

前面已经讲到，索引变量是不能直接对其使用 MOVE 语句和运算符语句的。索引变量是一个特殊的数据类型，由 COBOL 系统本身管理，只能用于引用索引表中的数据。除此之外，索引变量不能用于其他任何操作。

（3）适用语句不同。对下标表进行操作，实际上通常是对其中的下标变量进行操作。因此，下标表的适用语句也可看作是下标变量的适用语句。适用语句通常有以下几条。

- MOVE 语句
- PERFORM VARYING 语句
- ADD 语句
- SUBTRACT 语句
- MULTIPLY 语句
- DIVIDE 语句
- COMPUTER 语句

同样，适用于索引表的语句也可认为是适用于索引变量的语句。适用于索引表的语句有一些和适用于下标表是一样的，但更多的则并不相同。适用于索引表的语句有下面这几条。

- PERFORM VARYING 语句
- SET 语句
- SEARCH 语句
- SEARCH ALL 语句

（4）查找效率不同。索引表的查找效率通常要高于下标表的查找效率。这通常可以从两个方面体现出来。下面分别对此进行介绍。

一方面，索引表用来查找的索引变量访问效率要高于下标表中对应的下标。虽然下标变量可以通过在定义后面使用 USAGE IS COMP 来提高访问效率，代码如下。

```
05   sub   PIC 9(x1) USAGE IS COMP.
```

或者简写为以下形式。

```
05   sub   PIC 9(x1) COMP.
```

即使是通过这种方式提高了访问效率，其效率仍然还是没有索引的高。由于索引变量是

直接有 COBOL 系统本身来管理的，因此具有最高的访问效率。

另一方面，在索引表中可以方便地进行二分查找。前面已经讲到，二分查找方式是从算法角度来讲最高效的查找方式。因此，索引表的查找效率通常要高于下标表的查找效率。

2．索引表和下标表的相似之处

索引表和下标表最大的相似之处在于对数据的引用格式。二者都是通过在数据条目名称加上相应的索引名或下标名，完成对具体数据项的引用。对于下标表，引用数据的方式如下。

```
item-data   ( sub ).
```

对于索引表，引用数据的方式如下。

```
item-data  (index).
```

9.12 处理索引语句 SET

前面已经讲到，对于索引表中的索引变量而言，是不能将其视作通常的变量进行处理的。如果要对索引变量进行操作，必须使用 SET 语句。使用 SET 语句对索引变量进行操作，大体上可以分为两大类型的操作。其中一种类型的操作是对其进行赋值操作，另一种是对其进行算术运算操作。

下面结合一个实例分别对二者进行详细讲解。首先，以超市管理系统为基本模型。定义相应的索引表如下。

```
01  MARKET-TABLE.
   05    STORE-ITEM    OCCURS  10  TIMES
                        INDEXED BY  STORE-NDX.
         10   STORE-ITEM-NUM     PIC 9(9).
         10   STORE-ITEM-NAME   PIC X(10).
   05    SALE-ITEM    OCCURS  10  TIMES
                       INDEXED BY  SALE-NDX.
         10   SALE-ITEM-NUM     PIC 9(9).
         10   SALE-ITEM-NAME   PIC X(10).
```

这里有两类数据，分别为超市进货数据和超市售出商品数据。其中，以 STORE 开头的数据对应进货数据，以 SALE 开头的数据对应售出商品数据。此外，以 NUM 结尾的数据表示商品编号，以 NAME 结尾的数据表示商品名称。

该表中的索引变量相应也有两个，分别为 STORE-NDX 和 SALE-NDX。以下将重点对这两个索引变量进行讨论。

9.12.1 使用 SET 语句对索引赋值

前面讲过，不能按照通常的方式，使用 MOVE 语句对索引变量赋值。对索引变量赋值，必须使用 SET 语句。下面结合几个小例子，具体谈谈如何使用 SET 语句进行赋值操作。

（1）求取索引为 3 所对应的进货商品编号和名称。此处可以首先将索引变量 STORE-NDX 置为 3。然后，再通过该索引变量引用所求数据，并将结果存入相应位置。代码如下。

```
SET  STORE-NDX  TO  3.                    /*这里使用 SET 语句对索引变量赋值*/
MOVE  STORE-ITEM-NUM  (STORE-NDX)  TO RESULT-NUM.
MOVE  STORE-ITEM-NAME  (STORE-NDX)  TO RESULT-NAME.
```

（2）求取索引为 5 对应的进货商品编号以及售出商品编号。此处可以采用和例 1 中同样的方式，分别对 STORE-NDX 和 SALE-NDX 赋值。然后，再通过这两个索引变量各自引用其对应的数据，完成操作。代码如下。

```
SET  STORE-NDX  TO  5.
SET  SALE-NDX   TO  5.
MOVE  STORE-ITEM-NUM  (STORE-NDX) TO RESULT-STORE-NUM.
MOVE  SALE-ITEM-NUM  (SALE-NDX) TO RESULT-SALE-NUM.
```

然而，注意到这里对两个索引变量所赋的值是一样的，都为数字 5。因此，还可以通过 SET 语句对这两个变量统一进行赋值。这种方式下的代码如下。

```
SET  STORE-NDX  SALE-NDX  TO  5.
MOVE  STORE-ITEM-NUM  (STORE-NDX) TO RESULT-STORE-NUM.
MOVE  SALE-ITEM-NUM  (SALE-NDX) TO RESULT-SALE-NUM.
```

（3）假设程序中另外存在两个整型数据变量，分别为 INPUT-NUM 和 OUTPUT-NUM。并且，其中 INPUT-NUM 变量中已保存有相应数据。现需引用索引为 INPUT-NUM 的卖出商品名称。其后，再将此时的索引数值输出到 OUTPUT-NUM 变量中，以便观察。

对于本例而言，主要涉及到索引变量和其他通常变量之间的赋值操作。仍然使用 SET 语句完成操作。代码如下。

```
SET  INPUT-NUM  TO  SALE-NDX.
MOVE  SALE-ITEM-NAME  (SALE-NDX)  TO  RESULT-SALE-NAME.
SET  SALE-NDX   TO  OUTPUT-NUM.
```

9.12.2 使用 SET 语句对索引进行算术运算

对索引变量进行算术运算操作，仍然需要使用 SET 语句。以下依然结合上面定义的超市管理系统的索引表，通过几个小例子进行讲解。

（1）假设 STORE-NDX 已存有数据。现要求以该索引变量为基准，其后偏移 2 位索引的进货商品编号。这里并不知道当前 STORE-NDX 中的数据为多少。因此，必须使用算术运算，直接将 STORE-NDX 加上 2，再以此引用指定数据。代码如下。

```
SET  STORE-NDX  UP  BY  2.
MOVE  STORE-ITEM-NUM   ( STORE-NDX)  TO RESULT-STORE-NUM.
```

（2）假设 SALE-NDX 已存有数据。现要求以该索引变量为基准，往前偏移 2 位索引的进货商品名称。注意到，这里是往前偏移，而不是往后偏移。因此，完成本例要求的代码如下。

```
SET   SALE-NDX  DOWN  BY  2.
MOVE  SALE-ITEM-NAME   ( SALE-NDX)  TO  RESULT-SALE-NAME.
```

（3）假设 STORE-NDX 和 SALE-NDX 中均已存有数据。分别求取当前索引后偏移 3 位的进货商品名称和售出商品名称。这里涉及到对两个不同索引变量进行相同的操作，即都增加 3。因此，同样可以使用一条 SET 语句一并完成。代码如下。

```
SET   STORE-NDX  SALE-NDX  UP BY  3.
MOVE  STORE-ITEM-NAME  (STORE-NDX)  TO RESULT-STORE-NAME.
MOVE  SALE-ITEM-NAME  (SALE-NDX)  TO RESULT-SALE-NAME.
```

（4）假设程序中另外存在一个整型数据变量 TEMP-DATA。现要求将 STORE-NDX 加上

该变量中保存的数据，并将 SALE-NDX 减去此数据。本例主要是为了说明，使用 SET 语句进行算术运算时，同样可以包含普通变量。实现本例功能的代码如下。

```
SET   STORE-NDX  UP  BY  TEMP-DATA.
SET   SALE-NDX    DOWN  BY  TEMP-DATA.
```

注意 当使用 SET 语句时，语句中必须至少包含一个索引变量。例如，下面的几条 SET 语句都没有包含索引变量，因此都是错误的。

```
SET  3  TO  TEST-DATA.
SET  TEST-DATA  UP  BY  2.
SET  TEST-DATA-1  DOWN  BY  TEST-DATA-2.
```

9.13 查找索引表语句 SEARCH

SEARCH 语句用于对索引表中数据进行查找。并且，SEARCH 语句只能用于查找索引表，不能用于查找下标表。

9.13.1 SEARCH 语句的格式

在 SEARCH 语句中，主要需要包含所要查找的索引表表名、判断条件以及满足条件后的处理过程。此处所说的判断条件通常包含两个。其中一个是由用户指定的判断条件， 另一个则对应数据查找结束时的情况。SEARCH 语句最基本的格式如下。

```
SEARCH  indexed-table-entry
         AT   END
              do  something
         WHEN    condition
              do  something else
END-SEARCH.
```

关于这段代码，有以下两点需要特别注意。

- 对于表中出现的 do something，实际上为一组操作语句。并且，该组操作语句通常最多只包含 2 ~ 3 条语句。
- 对于表中出现的 condition，为用户指定的判断条件。该判断条件表达式中必须包含以索引引用的表中关键数据项（通常为数据编号）。并且，该关键数据项应出现在判断表达式的左边，即应该作为判断表达式的第一个操作数。以下语句反映了这一要求。

```
WHEN   TEST-KEY  (TEST-NDX)  =  SAME-DATA
```

此外，在实际使用 SEARCH 语句时，还要在之前对索引变量初始化。这样，SEARCH 语句方能知道从哪一条数据开始“SEARCH”。因此，在实际使用中，完整的代码如下。

```
SET   index  TO  x.                /*x 为任意自然数*/
SEARCH  indexed-table-entry
         AT   END
              do  something
         WHEN    condition
              do  something else
END-SEARCH.
```

最后，SEARCH 语句中还可以同时拥有多条 WHEN 子句，即可以进行多重条件判断。例如，以下几个条件。

- 当索引所引用的数据为 1 时，执行 statement 1。
- 当索引所引用的数据为 2 时，执行 statement 2。
- 当索引所引用的数据为 3 时，执行 statement 3。

通过多重条件判断，使用 SEARCH 语句的代码如下。

```
SEARCH  indexed-table-entry
         AT   END
              do  something
         WHEN     data  (index)  =  1
              perform  statemen1
         WHEN     data  (index)  =  2
              perform  statemen2
         WHEN     data  (index)  =  3
              perform  statemen3
END-SEARCH.
```

9.13.2 SEARCH 语句的功能

SEARCH 语句的功能实际上是实现对索引表中数据的顺序查找。SEARCH 语句实现的功能步骤如下。

- SEARCH 语句从指定的索引处开始查找。若 WHEN 子句后的条件没有得到满足，将索引变量相应增加一个数据单位。查找索引表中的下一条数据。同时继续判断 WHEN 子句后所列条件。
- 若 WHEN 子句后所列条件一旦满足，则停止查找。索引变量为当前满足条件的关键数据项所对应的索引值。
- 若直到该索引表中的最后一条数据仍然没有满足 WHEN 子句后所列条件的，将停止查找。同时，AT END 子句下面的语句将被执行。若没有写 AT END 子句，直接执行 SEARCH 语句后面的下一条语句。

以下结合一个具体实例进行讲解。

首先，建立一张索引表，其中包含一组记录有相关人员姓名的数据。这里由于只有一种数据类型，因此人员姓名数据即为该表的关键数据项。定义并初始化该索引表的代码如下。

```
01   MEMBER-DATA.
     05  FILLER     PIC X(10)    VALUE  'ZHANG SAN'.
     05             PIC X(10)    VALUE  'LI  SI'.
     05             PIC X(10)    VALUE  'WANG  WU'.
     05             PIC X(10)    VALUE  'LIU  DA'.
     05             PIC X(10)    VALUE  'XU  CHAO'.
01    MEMBER-TABLE  REDEFINES  MEMBER-DATA.
     05    MEMBERS       PIC  X(10)
                         OCCURS   5  TIMES
                         INDEXED BY  MEMBER-NDX.
```

假设需要查找姓名为“WANG WU”的人员。若找到，将变量 FOUND-FLAG 置为“Y”，否则将其置为“N”。

可以通过前面讲过的 PERFORM VARYING 语句实现该功能。使用 PERFORM VARYING

语句的代码如下。

```
MOVE   'N'   TO  FOUND-FLAG.
PERFORM  100-FOUND-PROCESS
          VARYING  MEMBER-NDX  FROM  1  BY  1
          UNTIL   MEMBER-NDX  >  5  OR  FOUND-FLAG = 'Y'.
......
100-FOUND-PROCESS.
     IF  MEMEBERS  (MEMBER-NDX)  = 'WANG WU'
         MOVE  'Y'  TO  FOUND-FLAG
     END-IF.
```

由于该表为索引表，还可以通过 SEARCH 语句进行查找，代码如下。

```
SET   MEMBER-NDX   TO  1.
SEARCH   MEMBERS
     AT   END
          MOVE  'N'  TO  FOUND-FLAG
    WHEN   MEMBERS  (MEMBER-NDX)  =  'WANG WU'
          MOVE  'Y'  TO  FOUND-FLAG
END-SEARCH.
```

以上两段代码执行后，FOUND-FLAG 中的值都将为‘Y’。

由此可见，使用 SEARCH 语句和使用 PERFORM VARYING 语句进行查找的效果是等同的。因为二者都是基于顺序查找方式进行查找的。但是，对于索引表而言，通常使用的是 SEARCH 语句。因为 SEARCH 语句除可以进行顺序查找外，还有以下两点好处。

- 只用指定起始查找位置，并且可含有多条条件判断语句（WHEN 语句）。因此，该语句在功能结构上较为清晰。
- 相对于 PERFORM VARYING 语句而言，SEARCH 语句的查找是自动执行的。因此，在编码上更为简便。

9.14 查找索引表语句 SEARCH ALL

SEARCH ALL 语句也是索引表所特有的数据查找语句。SEARCH ALL 语句最大的特点在于其使用的是二分查找方式，因此查找效率较高。并且，这种查找方式是该语句所内嵌的，不用进行编码。

9.14.1 SEARCH ALL 语句的格式要求

SEARCH ALL 语句的语法格式同 SEARCH 语句比较类似。在 SEARCH ALL 语句中，仍然主要包含所要查找的索引表表名、判断条件以及满足条件后的处理过程。SEARCH ALL 语句只是在系统内部处理过程中与 SEARCH 语句不同。SEARCH ALL 语句的基本格式如下。

```
SEARCH  ALL  indexed-table-entry
         AT   END
              do  something
         WHEN    condition
              do  something else
END-SEARCH.
```

与 SEARCH 语句类似，同样需要注意以下两点。

- 对于以 do something 表示的一组操作语句，通常该组语句内最多只有 2~3 条语句。
- 对于 condition 判断条件，必须包含以索引引用的具体数据项。并且该数据应该为判断表达式的第一个操作数。

此外，由于 SEARCH ALL 采用的是二分查找方式，因此相应的索引表中的数据必须有序。因此，能够使用 SEARCH ALL 语句的索引表，必须参照如下方式定义。

```
01   sample-indexed-table.
     05   sample-entry-list      OCCURS  x1  TIMES
                                  INDEXED  BY  sample-index.
                                  ASCENDING KEY sample-num.
          10    sample-num    PIC 9(x2).
          10    sample-data    PIC X(x3).
```

以上定义的能使用 SEARCH ALL 语句的索引表，为表示数据的顺序，需要定义数据编号。该段代码定义的数据编号即为 sample-num。同时需要注意的是，这里通过 ASCENDING KEY 表明了该索引表中的数据为升序排列。

上面代码中定义的是数据按升序排列的索引表。同样可以定义数据按降序排列的索引表。定义方式如下。

```
01   sample-indexed-table.
     05   sample-entry-list      OCCURS  x1  TIMES
                                  INDEXED  BY  sample-index.
                                  DESCENDING KEY sample-num.
          10    sample-num    PIC 9(x2).
          10    sample-data    PIC X(x3).
```

总之，SEARCH ALL 语句只能用于查找数据排列有序的索引表。这是由 SEARCH ALL 语句的查找方式——二分查找法的算法特点所决定的。对于此类索引表的定义，通常可分两种情况讨论。

- 对于升序排列的数据，通过 ASCENDING KEY 加上数据编号指明。
- 对于降序排列的数据，通过 DESCENDING KEY 加上数据编号指明。

最后需要注意的一点是，SEARCH ALL 语句中判断条件只能为一条。即在 SEARCH ALL 语句中，WHEN 子句只能出现一条。因此，下面的这段代码是错误的。

```
SEARCH  ALL  indexed-table-entry
         AT   END
              do  something
         WHEN     sample-data  (sample-index)  =  1
              perform  statemen1
         WHEN     sample-data  (sample-index)  =  2
              perform  statemen2
         WHEN     sample-data  (sample-index)  =  3
              perform  statemen3          ←错误的代码，WHEN 在 SEARCH ALL 语句中只能出现一条
END-SEARCH.
```

9.14.2 SEARCH ALL 语句的实际应用

此处以一个图书管理系统对于读者的管理为例，讲解 SEARCH ALL 语句在实际中的应用。对于读者数据而言，主要应包含读者的借书证号和读者的姓名这两项数据。假设表中只

包含 4 名读者，定义并初始化相应的索引表如下。

```
01  READERS-DATA.
    05   FILLER    PIC  9(4)    VALUE  1002.
    05             PIC  X(10)   VALUE 'XIAO MING'.
    05             PIC  9(4)    VALUE  1005.
    05             PIC  X(10)   VALUE 'XIAO HONG'.
    05             PIC  9(4)    VALUE  1023.
    05             PIC  X(10)   VALUE 'XIAO GANG'.
    05             PIC  9(4)    VALUE  1036.
    05             PIC  X(10)   VALUE 'XIAO WANG'.
01  READERS-TABLE   REDEFINES  READERS-DATA.
    05   READERS        OCCURS  4 TIMES
                         INDEXED  BY  READER-NDX
                         ASCENDING  KEY  READER-CODE.
        10   READER-CODE     PIC 9(4).
        10   READER-NAME     PIC X(10).
```

假设这里要查找借书证号为“1023”的读者姓名。若找到，则将该读者姓名存放在 FOUND-NAME 变量中，同时置 FOUND-FLAG 变量为“Y”。否则将 FOUND-NAME 置空，同时将 FOUND-FLAG 置为“N”。使用 SEARCH ALL 语句进行查找的代码如下。

```
SEARCH  ALL  READERS-ENTRY
        AT END
          MOVE  SPACES  TO  FOUND-NAME
          MOVE  'N'   TO  FOUND-FLAG
        WHEN   READER-CODE  (READER-NDX) = 1023
          MOVE   READER-NAME  ( READER-NDX)  TO FOUND-NAME
          MOVE   'Y'  TO  FOUND-FLAG
END-SEARCH.
```

该段代码执行后的结果如下所示。

- FOUND-FLAG 变量中保存为“Y”。
- FOUND-NAME 变量中保存为“XIAO GANG”。

最后需要注意的是，由于 SEARCH ALL 语句是对整张表进行查找的，因此不必指定索引值。

9.15 定长表和变长表

定长表就是表中数据条目的重复次数在定义时，被指定为一个常量，数据项个数不可变。前面讲到的表都属于定长表。变长表是表中数据条目的重复次数在定义时，只指定一个范围，具体数值根据其他变量得到。变长表的数据项个数是可变的。

9.15.1 定长表

定长表简单的说就是表中数据项个数不可变的表。前面例子中用到的表都是定长表。这里对定长表进行讨论，也可以作为对前面所学知识的一个回顾。下面代码定义了一张最基本的定长表。

```
01  FIXED-TABLE-ONE.
    05    ENTRY-ONE    OCCURS  10  TIMES.
          10   NUM-ONE    PIC 9(5).
```

```
        10   DATA-ONE   PIC X(10).
    05    SUB-ONE        PIC 99  COMP.
```

该段代码定义了一张下标表，同时该表也是定长表。根据 OCCURS 语句得到该表共有 10 个数据条目，因此可认为该表长度为 10。该表存储空间大小为 10×（5+10）=150。

同样，也可定义定长的索引表，定义方式如下。

```
01  FIXED-TABLE-TWO.
    05   ENTRY-TWO    OCCURS   10   TIMES
                      INDEXED   BY  TWO-NDX.
         10   NUM-TWO      PIC 9(5).
         10   DATA-TWO     PIC X(10).
```

以上定义的定长索引表的长度仍然为 10，同样是通过 OCCURS 语句后的重复次数得到的。该表既是定长表，也是索引表，这两个概念是并行的。

若要定义可用于二分查找的索引表，则还需使表中数据有序排列。若数据为升序排列，使用 ASCENDING KEY BY 语句完成；降序使用 DESCENDING KEY BY 语句。

升序排列的定长索引表定义方式如下。

```
01  FIXED-TABLE-THREE.
    05   ENTRY-THREE    OCCURS   10   TIMES
                        INDEXED   BY   THREE-NDX
                        ASCENDING  KEY  NUM-THREE.
         10   NUM-THREE      PIC 9(5).
         10   DATA-THREE     PIC X(10).
```

降序排列的定长索引表定义方式如下。

```
01  FIXED-TABLE-FOUR.
    05   ENTRY-FOUR     OCCURS   10   TIMES
                        INDEXED   BY   FOUR-NDX
                        DESCENDING  KEY  NUM-FOUR.
         10   NUM-FOUR      PIC 9(5).
         10   DATA-FOUR     PIC X(10).
```

以上列举了几种类型的定长表，同时也是对前面两节中关于表的定义的一个简单回顾。实际上，定长表最根本的特征就是定义表语句 OCCURS 后的重复次数是一个确定的数字。

9.15.2 如何定义变长表

变长表是本节要讨论的重点内容。当表中数据数目不确定时，使用变长表。变长表中的数据数目根据程序中执行的具体情况而确定。定义变长表时，主要是通过在 OCCURS 语句后加上 DEPENDING ON 选项实现的。另外，此处的 OCCURS 语句后通常不再是一个具体的数字，而是一个范围大小。定义变长表最关键的语句如下。

```
X  OCCURS  x1 TO x2  TIMES  DEPENDING ON  Y.
```

完整的定义方式如下。

```
01   VARIABLE-TABLE-ONE.
     05   FIELD-ONE        OCCURS  1 TO 9  TIMES
                            DEPENDING ON TABLE-LENGTH-ONE.
          10  VARI-NUM-ONE     PIC 9(5).
          10  VARI-DATA-ONE    PIC X(10).
     05   TABLE-LENGTH-ONE        PIC  9.
```

以上定义的变长表，表中数据条目最少为 1 条，最多为 9 条。具体的数据条目数通过整型变量 TABLE-LENGTH 指定。可以看到，定义变长表时关键用到以下这条语句。

此外，关于变长表的定义方式，有以下几点需要注意。

1．变长表的起始长度可以从 0 开始

当变长表的长度为 0 时，该表为空表，空表不包含任何数据。空表只是定义了一个表的结构，表中数据可在程序中根据实际情况添加。该情况下的定义语句如下。

```
01   VARIABLE-TABLE-TWO.
     05   FIELD-TWO        OCCURS  0 TO 9  TIMES
                            DEPENDING ON TABLE-LENGTH-TWO.
          10  VARI-NUM-TWO     PIC 9(5).
          10  VARI-DATA-TWO    PIC X(10).
     05   TABLE-LENGTH-TWO        PIC  9.
```

2．变长表的终止长度必须大于起始长度

对于这一点，即 OCCURS 语句中第 2 个操作数必须大于第 1 个操作数。例如，对于下面这段代码，就是错误的。

```
01   VARIABLE-TABLE-ERR1.
     05   FIELD-ERR1         OCCURS  5 TO 2  TIMES     ←错误。变长表的终止长度必须大于起始长度
                             DEPENDING ON TABLE-LENGTH-ERR1.
          10  VARI-NUM-ERR1         PIC 9(5).
          10  VARI-DATA-ERR1        PIC X(10).
     05   TABLE-LENGTH-ERR1        PIC  9.
```

3．变长表 DEPENDION ON 选项后变量的内容必须合法

这一点也是初学者在开发过程中常常容易忽视的。如果 DEPEDNING ON 选项后的变量非法，程序将非正常结束。例如以下代码就是错误的。

```
01   VARIABLE-TABLE-ERR2.
     05   FIELD-ERR2             OCCURS  1 TO 9  TIMES
                                 DEPENDING ON TABLE-LENGTH-ERR2.
          10  VARI-NUM-ERR2         PIC 9(5).
          10  VARI-DATA-ERR2        PIC X(10).
     05   TABLE-LENGTH-ERR2        PIC X   VALUE 'A'.   ←此处错误
```

以上代码中的 DEPENDING ON 选项后的变量为 TABLE-LENGTH-ERR2。该变量用于指定该变长表的实际长度大小。然而该变量却被定义为字符型变量，内容为字符“A”，因此是错误的。

同样，下面这段代码虽然将控制变长表长度的变量定义为整型变量。但该变量所保存的内容超出了变长表长度的范围，仍然是错误的。代码如下。

```
01   VARIABLE-TABLE-ERR2.
     05   FIELD-ERR3             OCCURS  1 TO 5  TIMES
                                 DEPENDING ON TABLE-LENGTH-ERR3.
          10  VARI-NUM-ERR3         PIC 9(5).
          10  VARI-DATA-ERR3        PIC X(10).
     05   TABLE-LENGTH-ERR2        PIC 9   VALUE  9.
```

4．OCCURS 后也可只出现一个数字

该情况下的定义代码如下。

```
01   VARIABLE-TABLE-TWO.
     05   FIELD-TWO          OCCURS   9  TIMES
                             DEPENDING ON TABLE-LENGTH-TWO.
          10  VARI-NUM-TWO     PIC 9(5).
          10  VARI-DATA-TWO    PIC X(10).
     05   TABLE-LENGTH-TWO        PIC  9  VALUE 5.
```

由于变长表的实际长度是根据 DEPENDING ON 后面的变量指定的。因此，对于以上变长表，实际长度为 5，而不是 9。

9.15.3 变长表中数据的引用范围

引用变长表中的数据，通常要对该数据初始化。变长表的初始化方式同普通表的初始化方式基本一样。关于表的初始化，可参看本章前面所讲解的内容。只是需要说明，对于变长表，初始化后的数据并不总是能被引用的。

例如，对于下面这张初始化过的变长表。

```
01   VARI-TABLE       VALUE 'ABCDE3'.
     05   X           OCCURS 5 TIMES
                      DEPENDING ON  Y
                      PIC  X.
     05   Y            PIC  9.
```

通过在 01 级数据项后对整张表进行初始化后，表中各数据内容如下。

- X（1）：A
- X（2）：B
- X（3）：C
- X（4）：D
- X（5）：E

此外，变量 Y 中包含的是数字 3。

由于变长表的实际长度是根据 DEPENDING ON 后面的变量指定的。对于以上代码，即通过变量 Y 指定。因此，以上变长表的实际长度为 3，而不是 5。因此对于以上变长表，可引用的数据只有以下几个。

- X（1）：A
- X（2）：B
- X（3）：C

X（4）和 X（5）这两项数据实际也是存在的，但不能被引用。若在后面的程序中变量 Y 的值被重置为 5 或者更大，则可对 X（4）和 X（5）这两项数据进行引用。

9.15.4 变长表应用举例

变长表在实际应用中，通常需要对其控制表的长度的变量进行操作。以下结合一个具体的例子，反映变长表在实际中通常是如何应用的。

设有一张旧表，包含了一组课程名称。表的定义及初始化数据如下。

```
01   COURSE-DATA.
     05   FILLER     PIC  X(5)   VALUE  'C'.
     05              PIC  X(5)   VALUE  'C++'.
     05              PIC  X(5)   VALUE  'JAVA'.
     05              PIC  X(5)   VALUE  'COBOL'.
     05              PIC  X(5)   VALUE  'JCL'.
     05              PIC  X(5)   VALUE  'DB2'.
     05              PIC  X(5)   VALUE  'CICS'.
01   OLD-COURSE-TABLE      REDEFINES   COURSE-DATA.
     05   OLD-COURSES   PIC  X(5)    OCCURS 7 TIMES.
```

若目前只知道该旧表中的程序设计语言类课程是从第一条数据项开始连续排列的。并且，最后一门程序设计语言课程为 COBOL。整张表共包含 7 门课程。现要建立一张新表，新表中只包含程序设计语言类课程。

由于这里并不知道新表的实际长度，因此需要将新表定义为变长表。该变长表的长度最小为 1，最大为 7。当长度为 1 时表示旧表中只含有 COBOL 这一门程序设计语言课程。当长度为 7 时，表示旧表中所有课程都为程序设计语言类课程。定义的代码如下。

```
01   NEW-COURSE-TABLE.
     05   NEW-COURSES    PIC  X(5)  OCCURS 0 TO 7 TIMES
                                    DEPENDING ON NEW-LENGTH.
     05   NEW-LENGTH     PIC  9.
```

定义完成后，再通过 PERFORM VARYING 语句将需要保存的数据依次放入新表之中。此外，由于题目中已说明“COBOL”为最后一条记录。因此，当数据读到 COBOL，或已经达到表中数据量最大值时，该语句应该停止。实现以上功能的代码如下。

```
MOVE   'N'   TO  END-FLAG.
PERFORM  100-MOVE-STATE.
        VARYING   NEW-LENGTH   FROM  1  BY   1
        UNTIL     END-FLAG  =  'Y'
        OR        NEW-LENGTH > 7.
......
100-MOVE-STATE.
       MOVE  OLD-COURSES  (NEW-LENGTH)  TO  NEW-COURSES  (NEW-LENGTH).
       IF  NEW-COURSES  (NEW-LENGTH)  =  'COBOL'
          MOVE  'Y'  TO END-FLAG.
       END-IF.
```

以上代码完成了题目的要求。代码执行后，生成的新表中的各数据内容如下。

- NEW-COURSES （1）：C
- NEW-COURSES （2）：C++
- NEW-COURSES （3）：JAVA
- NEW-COURSES （4）：COBOL

此外，NEW-LENGTH 变量包含数字 4，表示该新表的长度。

9.16 嵌套表

嵌套表也叫多维表格。嵌套表中含有多条 OCCURS 语句，各 OCCURS 语句是一个层次关系，而非并列关系。在 COBOL-85 标准中，最多允许 7 层嵌套。

9.16.1 如何定义嵌套表

首先从最基本的 2 层嵌套表开始学习。2 层嵌套表也叫二维表格，其中包含两条层次关系的 OCCURS 语句。以下代码定义了一个 2 层嵌套表。

```
01   NESTED-TABLE-2.
     05   TABLE-ROW   OCCURS  2  TIMES.
          10    TABLE-COLUMN  OCCURS   3  TIMES.
                15   TABLE-ITEM-1        PIC X(1).
                15   TABLE-ITEM-2        PIC X(2).
```

该表中的数据存储结构如图 9.4 所示。

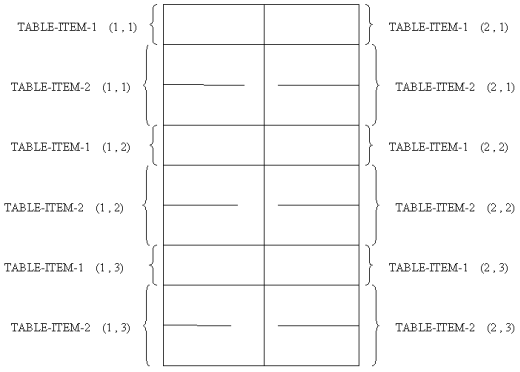

图 9.4　2 层嵌套表的存储结构

根据 2 层嵌套表的定义方式，可定义 3 层嵌套表如下。

```
01   NESTED-TABLE-3.
     05   TABLE-DEPTH   OCCURS  2  TIMES.
          10    TABLE-ROW-3  OCCURS   2  TIMES.
                15   TABLE-COLUMN-3  OCCURS  3  TIMES.
                     20   TABLE-ITEM3-1        PIC X(1).
                     20   TABLE-ITEM3-2        PIC X(2).
```

对应于以上定义的 3 层嵌套表，表中数据存储结构如图 9.5 所示。

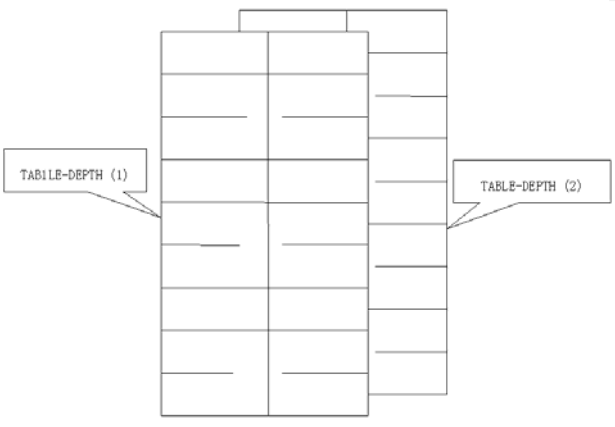

图 9.5　3 层嵌套表的存储结构

依此类推，同样可以定义 4 层嵌套表直到 7 层嵌套表（COBOL-85 中最多只能嵌套 7 层）。表中数据的存储方式依次在上一层嵌套的基础上加上一个“维度”。因此，相应的 3 层嵌套表也叫 3 维表格，4 层嵌套表也叫 4 维表格等。

9.16.2 嵌套下标表

嵌套下标表是将下标表进行嵌套生成的。上一小节中定义的两个嵌套表都是嵌套下标表。假设某一 2 层嵌套下标表的定义如下。

```
01   NESTED-TABLE-2.
     05   TABLE-ROW   OCCURS  2  TIMES.
          10    TABLE-COLUMN  OCCURS   3  TIMES.
                15   TABLE-ITEM          PIC X.
```

对于该表而言，若要引用其中的具体数据，方式如下。

```
TABLE-ITEM  ( x1, x2 )
```

或者使用如下方式。

```
TABLE-ITEM  ( x1  x2 )
```

其中括号中的 x1 和 x2 为该数据的两个下标。这两个下标分别对应 TABLE-ROW 和 TABLE-COLUMN。因此，x1 的取值范围为 1 到 2，x2 的取值范围为 1~3。

同样，对于一个 3 层嵌套表，结果定义如下。

```
01   NESTED-TABLE-3.
     05   TABLE-DEPTH   OCCURS  2  TIMES.
          10    TABLE-ROW-3  OCCURS   2  TIMES.
                15   TABLE-COLUMN-3  OCCURS  3  TIMES.
                     20   TABLE-ITEM-3        PIC X.
```

对其中具体数据的引用方式如下。

```
TABLE-ITEM-3  ( x1, x2, x3 ).
```

或者使用如下方式。

```
TABLE-ITEM-3  ( x1  x2  x3 ).
```

以上数据共有 3 个下标，即 x1、x2、x3。这 3 个下标的对应关系及取值范围分别如下。

- x1 对应 TABLE-DEPTH，取值范围：1，2。
- x2 对应 TABLE-ROW-3，取值范围：1，2。
- x2 对应 TABLE-COLUMN-3，取值范围：1，2，3。

由此可见，引用嵌套下标表中的数据时，嵌套有几层，该数据相应的就有几个下标。第 1 个下标对应表中第 1 层嵌套，第 *n* 个下标对应表中第 *n* 层嵌套。

9.16.3 嵌套索引表

嵌套索引表是将索引表嵌套而生成的表。嵌套索引表每层嵌套的都是一张索引表，因此有多少层嵌套就有多少个索引。下面定义一个 3 层嵌套索引表。

```
01  NESTED-INDEX-TABLE.
05   TABLE-DEPTH   OCCURS  2  TIMES
                   INDEXED  BY  NDX-A.
     10    TABLE-ROW   OCCURS   4  TIMES
                       INDEXED  BY  NDX-B.
           15    TABLE-COLUMN   OCCURS  5  TIMES
                                INDEXED  BY  NDX-C
                                PIC  X (10).
```

以上定义的嵌套索引表是 3 层嵌套的，因此也可被称为三维表格。其中的每一维度相当于这里每一层嵌套的数据项。其中每层数据项所占存储空间大小如下。

- 对于第一层嵌套数据项 TABLE-COLUMN 而言，空间大小为 10 个字节。并且该数据项重复出现 5 次。
- 对于第二层嵌套数据项 TABLE-ROW 而言，空间大小为 5 个第一层嵌套数据项的总和。由于第一层嵌套数据项 TABLE-COLUMN 的大小为 5 个字节。因此，TABLE-ROW 所占空间大小通过以下代数式得出。

```
5 字节（TABLE-COLUMN 的大小）* 10 （TABLE-COLUMN 的重复次数）= 50 字节
```

- 对于第三层嵌套数据项 TABLE-DEPTH 而言，同样可得到其空间大小如下。

```
50 字节（TABLE-ROW 的大小）* 4 （TABLE-ROW 的重复次数）= 200 字节
```

计算以上嵌套索引表中各层嵌套数据的空间大小，主要是为了下面计算数据偏移量。结合前面所学知识，索引表中的索引在内存中存储的是所引用数据的偏移量。此偏移量的大小即索引在内存中所存储的真实数值。

下面，首先类似嵌套下标表，对以上嵌套索引表中任意引用一条数据如下。

```
TABLE-COLUMN    （2 , 3, 1）
```

其中括号中各个数字对应的索引如下。

- 2 对应索引变量 NDX-A，即第三层嵌套索引变量。
- 3 对应索引变量 NDX-B。即第二层嵌套索引变量。
- 1 对应索引变量 NDX-C。即第一层嵌套索引变量。

结合前面计算得到的各层嵌套数据的空间大小。

- 第三层嵌套数据项 TABLE-DEPTH 每条大小为 200 字节。
- 第二层嵌套数据项 TABLE-ROW 每条大小为 50 字节。
- 第一层嵌套数据项 TABLE-COLUMN 每条大小为 10 字节。

得到该条数据的实际偏移量如下。

```
TABLE-COLUMN （2 , 3, 1）的偏移量 = （2 * 200 + 3 * 50 +1*10）字节 = 560 字节
```

其中，各个 index 变量在内存中的实际数值大小分别如下所示。

- NDX-A 实际数值为 2×200=400
- NDX-B 实际数值为 3×50=150
- NDX-C 实际数值为 1×10=10

此外，有时还常常对嵌套索引表中的数据以如下方式进行引用。

```
TABLE-COLUMN    （NDX-A+2 , NDX-B+3, NDX-C-1）
```

此时计算该条数据偏移量大小的式子如下。

```
TABLE-COLUMN    (NDX-A+2 , NDX-B+3, NDX-C-1)的偏移量  =
                          NDX-A在内存中的实际数值大小 + 2 * 200
                        + NDX-B在内存中的实际数值大小 + 3 * 50
                        + NDX-B在内存中的实际数值大小 - 1 * 10
```

9.17 本章回顾

本章主要讲解了 COBOL 中表的概念及应用。其中包含两类重要的表：下标表和索引表。之后介绍了定长表和变长表的概念。最后介绍了嵌套表的概念。

本章首先对 COBOL 中的表进行了简介，之后重点讲解了下标表及其相关的内容。学习这部分内容，需要理解表的基本概念，理解下标表的概念及特点，掌握如何使用 OCCURS 语句定义表，掌握如何使用 PERFORM VARYING 语句对表进行操作，掌握如何对表进行初始化，深入理解 3 种针对表的查找方式的算法思想，能够独立编写代码对表中数据进行直接查找和顺序查找，掌握如何使用 PERFORM FUNCTION 语句对表中数据进行统计计算。

本章接下来讲解了索引表及其相关的内容。学习这部分内容，需要理解索引表的概念及其特点，掌握如何使用 SET 语句对索引表中的索引进行相关操作，掌握如何使用 SEARCH 语句和 SEARCH ALL 语句对索引表进行查找。

本章最后对定长表、变长表以及嵌套表进行了介绍。学习这部分内容，需要理解定长表和变长表的概念，掌握如何定义和使用变长表，如何定义和使用嵌套表等。

第 10 章 程序的调试与测试

通过前面各章节的学习，已经能够编写具有一定功能的 COBOL 小程序了。在程序开发的过程中及开发结束后，会涉及到对程序的调试与测试。实际上，在整个软件生存周期中，对于软件的后期维护工作通常占到 80%以上。而软件后期维护工作主要是建立在程序的调试与测试的基础之上的。因此，本章所讲解的内容十分重要。

10.1 调试与测试的基本概念

程序的调试与测试是两个不同的概念。调试有时也叫 Debug，主要是针对程序源代码的查错与排错；而测试通常被称作 Test，主要是用于对软件产品性能的分析。下面分别对二者的基本概念进行具体的介绍。

10.1.1 调试的基本概念

程序的调试属于程序开发的一部分。一个人调试能力的大小通常也反映了其开发能力的大小。程序的调试主要是指在开发过程中及开发完成后对程序源代码的查错与排错。调试程序的目的是为了确定和修复造成程序不良功能的原因，主要是为了保证程序的正确性。在程序的编写之中，任何人都难免会造成这样或那样的错误，因此需要不断地对程序进行调试。

在编码中因处理过程或语句的句点没有标注而造成程序不能正确编译是时有发生的，需要对程序进行调试。此外，有时程序虽然能够被编译运行，却得不到所期望的运行结果。这时也需要对程序进行调试。实际上，前者出现的错误属于语法错误，后者属于逻辑错误。关于程序调试中所需处理的错误类型，以及常用的调试方法，将在后面的章节中详细讲解。

10.1.2 测试的基本概念

程序的测试和程序开发是两个独立的概念。实际上是两种不同的工作性质。然而，二者同时又是紧密相连的，在具体工作中需要经常联系和配合。

测试主要是针对由程序所形成的软件产品的性能而言的。用于进行测试的程序，都应该是通过调试之后，可以运行的程序。测试是在完成初期开发之后进行的，主要用于检测软件产品质量是否符合客户的要求。

此外，测试也应该包括对程序算法设计质量的检测。通常而言，一个好的算法应该考虑达到以下 4 个目标。

- 正确性：此处所说的正确性不仅是指程序能够对于一组特定的输入数据得出正确的结果。而且通常指程序对于精心选择的典型、苛刻而带有刁难性的几组输入数据也能得出正确的结果。
- 可读性：可读性好坏主要是指其逻辑结构是否清晰，代码注释是否到位等。一段可读性好的代码是利于其后期的维护以及基于此的二次开发。
- 健壮性：是指当输入数据非法时，算法也能适当地做出反应或进行处理，而不至于产生莫名其妙的输出结果。
- 效率与低存储量需求：效率是指算法的执行时间，执行时间越短，效率越高。存储量需求指的是算法在执行过程中所需要的最大存储空间。对于实现同样功能的算法，通常存储量需求越低，该算法越好。

例如，下面是一段求两个数之和的程序。其中一个数字由程序所给定，另一个数字通过用户从键盘输入。该段程序完整的代码如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID.      SUM-PROG-ONE.
 AUTHER.          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   NUM1   PIC   99.
 01   NUM2   PIC   99.
 01   SUM    PIC  999.
*
 PROCEDURE   DIVISION.
      MOVE   20   TO  NUM1.
      ACCEPT  NUM2.
      ADD NUM1 TO NUM2
             GIVING SUM.
     DISPLAY  'THE SUM IS:'  SUM.
      STOP  RUN.
```

该段程序在输入正确数据的前提下是可以正常执行的。但是，假若此处用户输入数据非法，程序将会产生莫名其妙的输出结果。例如，如果用户此处输入的不是数字，而是字母，那么程序所产生的输出结果必将不会如预期那样。

也就是说，通过测试，可以发现该程序的健壮性是不够的。程序健壮性不够对于一些实习问题或练习题的影响可能并不大。但对于实际的软件产品质量而言，程序健壮性则是十分关键的一项。

对于前面提到的用户输入非数字字符的情况，可以通过如下方式进行处理，可以增强程序的健壮性。修改后的代码如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID.      SUM-PROG-TWO.
 AUTHER.          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   NUM1   PIC   99.
 01   NUM2   PIC   99.
 01   SUM    PIC  999.
*
 PROCEDURE   DIVISION.
    MOVE   20   TO  NUM1.
     ACCEPT  NUM2.
     IF  NUM2  IS  NOT  NUMERIC
         DISPLAY  'INPUT  DATA  ERROR !'
     ELSE
         ADD NUM1 TO NUM2
                 GIVING SUM
      DISPLAY  'THE SUM IS:'  SUM
      END-IF.
       STOP  RUN.
```

10.2 调试所需处理的错误类型

前面讲到，调试主要是针对程序中的错误而言，是为了进行查出错误和排除错误。通常所说的错误可以分成两大类型，其中一种类型为语法错误，另一种类型为逻辑错误。下面分别对二者进行讲解。

10.2.1 语法错误

在编写程序时，难免会出现一些所谓的“笔误”。此处的“笔误”就是程序调试中所说的语法错误。

语法错误是最常见的一种错误，造成的后果是程序不能正确地被编译和运行。语法错误是相对易于检查和排除的。对于 COBOL 而言，可以通过在 SDSF 里查看所提交程序的编译结果，从而得知程序中的语法错误。实际上，程序的语法错误都是可以通过编译器检测出来的。在 COBOL 程序中，语法错误主要有以下几种情况。

- 漏写或多写句点结束标志。
- 漏写或多写中划线连字符。
- 代码拼写错误。
- 漏写空格。
- 将字符变量当作数值变量进行处理。
- 错误地使用程序关键字。
- 语句格式错误。

第一种情况在前面的例子中已经讲过，连字符和空格的错误和第一种情况下的句点的错

误是类似的。将字符变量错当成数值变量处理的情况，在基本数据类型一章中已反复讲到过；错误地使用程序关键字的情况，通常在编写大型程序时，使变量名中至少包含一个连字符便可解决，因为 COBOL 程序中的关键字都是没有连字符的，这样便不会错将变量名命名为关键字了。

下面主要以拼写错误和语句格式错误为例进行讲解。例如，对于下面这段代码，就有两处代码拼写错误。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID.      SPELL-ERROR-PROG.
 AUTHER.          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   TEST-DATA    PIC  X(12).
*
 PROCEDURE   DIVISION.
     MOVE  'hello world!'  TO  TEST-DATE.  /*此处错误*/
     PRINTF  TEST-DATA./*此处错误*/
      STOP  RUN.
```

对于该段代码中的第一处语法错误，是将变量名 TEST-DATA 错写成了 TEST-DATE。变量名的拼写错误情况是很容易出现的，尤其是在大型程序之中。该段代码中的第二处错误是将语句名 DISPLAY 写成了 PRINTF。这种情况往往是初学者容易犯的，随着开发经验增多，这种情况的错误会相对减少。

下面这段代码使用了 EVALUATE 语句进行多重条件判断。但在该语句的使用格式上出现了错误，代码如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID.      FORMAT-ERROR-PROG.
 AUTHER.          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   TEST-NUM    PIC  9.
*
 PROCEDURE   DIVISION.
     ACCEPT  TEST-NUM.
     EVALUATE /*此处错误*/
       WHEN  TEST-NUM = 1
           DISPLAY  'TEST NUM IS 1'
       WHEN  TEST-NUM = 2
           DISPLAY  'TEST NUM IS 2'
       WHEN  TEST-NUM = 3
           DISPLAY  'TEST NUM IS 3'
      END-EVALUATE.
       STOP  RUN.
```

该段代码使用了 EVALUATE 语句，但使用格式是错误的。根据前面所讲的 EVALUATE

语句的正确使用格式，该段代码修正后如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID.      FORMAT-CORRECT-PROG.
 AUTHER.          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   TEST-NUM    PIC  9.
*
 PROCEDURE   DIVISION.
     ACCEPT  TEST-NUM.
     EVALUATE TRUE
       WHEN  TEST-NUM = 1
          DISPLAY  'TEST NUM IS 1'
       WHEN  TEST-NUM = 2
          DISPLAY  'TEST NUM IS 2'
       WHEN  TEST-NUM = 3
          DISPLAY  'TEST NUM IS 3'
      END-EVALUATE.
       STOP  RUN.
```

此外，通过 SDSF 虽然可以看到 COBOL 程序在编译中的报错信息。但是，程序作为一个整体，是牵一发而动全身的。因此，对于在 SDSF 里看到的报错信息，问题常常并不出现在该处，而是由于此前的错误所造成的。这时，便需要根据调试经验和调试能力进行查错和排错了。

最后需要注意的是，单一的错误在编译中往往会造成多处报错信息，比如非法使用数据名称等。因此，应先修改数据部中的错误，这样就不会在其后的过程部中反复出现同类错误了。

10.2.2 逻辑错误

逻辑错误通常是指程序虽然可以被编译和运行，但运行的结果却并不是所期望的。由于程序此时在编译中并不会报错，因此逻辑错误通常比语法错误更加难以发现和排除。

例如，下面这段代码所要实现的功能是依次判断 FIRST-DOOR 和 SECOND-DOOR 是否为 Y。其中 FIRST-DOOR 相当于第一道门，SECOND-DOOR 相当于第二道门。只有进入第一道门，方能进入第二道门。这两个数据为 Y 的情况下，表示其对应的门可以通过。代码如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID.      LOGICAL-ERROR-PROG.
 AUTHER.          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   FIRST-DOOR      PIC  X.
 01   SECOND-DOOR    PIC  X.
*
 PROCEDURE   DIVISION.
```

```
        MOVE  'Y'  TO  FIRST-DOOR.
        MOVE  'N'  TO  SECOND-DOOR.
        IF  FIRST- = 'Y'
           DISPLAY 'PASSED THE FIRST DOOR'
           IF  DATA-TWO = 'Y'
              DISPLAY 'PASSED THE SECOND DOOR'
        ELSE
           DISPLAY 'COULD NOT PASS'
        END-IF.
         STOP  RUN.
```

该段代码的本意是当 FIRST-DOOR 不为 Y 的情况下，直接输出 COULD NOT PASS。当 FIRST-DOOR 为 Y 的情况下，继续判断 SECOND-DOOR 的内容。根据前面所学的嵌套 IF 语句知识，该代码中的 ELSE 将匹配第二个 IF，于是将得到如下输出结果。

```
COULD NOT PASS
```

根据条件，FIRST-DOOR 为 Y，SECOND-DOOR 为 N，应该能够通过第一道门。然而该程序输出的结果却是 COULD NOT PASS，表示连一道门都不能通过。此处便是出现了逻辑错误。该逻辑错误是由于错误地使用了嵌套 IF 语句的逻辑结构而造成的。为实现题中所指定的功能，正确的代码应该如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID.      LOGICAL-CORRECT-PROG.
 AUTHER.          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   FIRST-DOOR      PIC  X.
 01   SECOND-DOOR    PIC  X.
*
 PROCEDURE   DIVISION.
     MOVE  'Y'  TO  FIRST-DOOR.
     MOVE  'N'  TO  SECOND-DOOR.
     IF  FIRST- = 'Y'
        DISPLAY 'PASSED THE FIRST DOOR'
        IF  DATA-TWO = 'Y'
           DISPLAY 'PASSED THE SECOND DOOR'
        END-IF
     ELSE
        DISPLAY 'COULD NOT PASS'
     END-IF.
      STOP  RUN.
```

该段代码执行后，程序有如下输出结果。

```
PASSED THE FIRST DOOR
```

最后还需注意的是有时在代码中并没有错误，但程序仍然不能正常地编译和运行。错误是发生在系统上面的。例如，读入的文件已被删除，或者程序所在的数据集空间不够等。这时程序便会出现 ABEND（ABnormal End of program），即非正常结束。其中数据集空间不够往往是在实际开发中经常会遇到的情况。

COBOL 程序通常是存放于分区数据集的成员之中的。因此，在反复的提交作业进行调

试后，其分区数据集的空间往往会被占满。此时，再次使用 JCL 提交作业时，系统便会报错，即使所提交的程序完全正确。对这一问题解决的方式是在该分区数据集前，通过行命令对其进行压缩。行命令的使用方式在本书第一章基础知识里有所介绍。

此外需要补充的一点是，程序出现 ABEND 时也会出现相应的错误信息编号。例如，下面就为两则常见的错误编号及其对应的错误内容。

- S322: 表示程序中存在死循环。通常需要通过检查 PERFORM 语句进行排错。
- S722: 表示程序超出了可打印输出的行数。通常需要通过检查循环结构中的打印输出语句，或者检查多余的 DISPLAY 语句进行排错。

10.3 增殖式调试方法

调试程序往往比编写程序花费的时间要更多。通常，使用增殖式调试方法可以提高整个调试和开发的效率。

所谓增殖式调试方法，简单地说就是先开发出整个程序的一部分，调试直至其无误后，再继续开发。并且，这是一个不断反复的过程，即开发一部分，调试，再开发下一部分，再调试……直到开发出整个程序为止。这样做能够保证程序开发的执行进度，使得程序最终能够如期完成。否则，如果等整个程序编写好之后再进行调试，其调试过程将变得非常复杂。并且很有可能发现程序是在最开始便出现了严重的错误，此时后果将不堪设想。

例如，对于任何一款软件产品而言，都可分为两大部分。其一是界面部分，其二是功能部分。使用 COBOL 在大型机上所编写的软件，界面部分通常主要是通过 CICS 实现的。关于 CICS，将在后面的 COBOL 扩展篇中进行详细讲解。关于功能部分，通常 MIS 系统上都必不可少以下 4 大功能。

- 增加数据功能：使系统中的数据能够通过文件或输入进行添加，并存放在指定位置中。
- 删除数据功能：能够对系统中已有数据进行删除，删除后保存原来系统中的数据结构不变。
- 修改数据功能：能够对系统中已有数据内容进行修改，修改过程中不得破坏其他数据内容。
- 查询数据功能：能够对系统中已有数据内容进行查询，查询得到的数据必须正确、完整。

下面以银行账户管理系统模型为例，并且只考虑其功能部分。首先应该编写其功能主菜单的代码，完整的代码如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID.      MIS-TEST-PROG.
 AUTHER.          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   ADD-FUNC       PIC  X(10).
 01   DELETE-FUNC    PIC  X(10).
 01   MODIFY-FUNC    PIC  X(10).
 01   INQUIRE-FUNC   PIC  X(10).
 01   FUNC-CHOICE    PIC  X(10).
*
 PROCEDURE   DIVISION.
```

```
INITIAL-PROCESS.
    MOVE  'ADD'  TO  ADD-FUNC.
    MOVE  'DELETE'  TO  DELETE-FUNC.
    MOVE  'MODIFY'  TO  MODIFY-FUNC.
    MOVE  'INQUIRE'  TO  INQUIRE-FUNC.
SELECT-PROCESS.
    ACCEPT  FUNC-CHOICE.
    EVALUATE  FUNC-CHOICE
      WHEN ADD-FUNC
       DISPLAY 'ADD FUNCTION ACTIVATE'
      WHEN DELETE-FUNC
       DISPLAY 'DELETE FUNCTION ACTIVATE'
    WHEN MODIFY-FUNC
       DISPLAY 'MODIFY FUNCTION ACTIVATE'
    WHEN INQUIRE-FUNC
       DISPLAY 'INQUIRE FUNCTION ACTIVATE'
    END-EVALUATE.
     STOP  RUN.
```

此时对以上程序进行调试。当调试无误后，继续进行编码。通常，是将增加数据等具体功能模块作为一个单独的程序进行开发。待该功能模块调试无误后，再将其连接到功能主菜单中进行调试。由于调用外部程序目前还没讲到，因此作为一个系统模型，以程序中的处理过程取代程序模块。在本次调试完成后，继续编码的结果如下。

```
IDENTIFICATION   DIVISION.
PROGRAM-ID.     MIS-TEST-PROG.
AUTHER.         XXX.
*
ENVIRONMENT   DIVISION.
*
DATA  DIVISION.
WORKING STORAGE SECTION.
01   ADD-FUNC       PIC  X(10).
01   DELETE-FUNC     PIC  X(10).
01   MODIFY-FUNC    PIC  X(10).
01   INQUIRE-FUNC   PIC  X(10).
01   FUNC-CHOICE    PIC  X(10).
*
PROCEDURE   DIVISION.
INITIAL-PROCESS.
    MOVE  'ADD'  TO  ADD-FUNC.
    MOVE  'DELETE'  TO  DELETE-FUNC.
    MOVE  'MODIFY'  TO  MODIFY-FUNC.
    MOVE  'INQUIRE'  TO  INQUIRE-FUNC.
SELECT-PROCESS.
    ACCEPT  FUNC-CHOICE.
    EVALUATE  FUNC-CHOICE
      WHEN ADD-FUNC
       PERFORM 100-ADD-PROCESS
      WHEN DELETE-FUNC
       PERFORM 200-DELETE-PROCESS
     WHEN MODIFY-FUNC
       PERFORM 300-MODIFY-PROCESS
     WHEN INQUIRE-FUNC
       PERFORM 400-INQUIRE-PROCESS
```

```
        END-EVALUATE.
         STOP  RUN.
    100-ADD-PROCESS.
         ……
    200-DELETE-PROCESS.
         ……
    300-MODIFY-PROCESS.
         ……
    400-INQUIRE-PROCESS.
         ……
```

以上就是通过增殖式调试方法完成程序开发及调试的大体步骤。需要注意的是，实际开发中，该系统要比以上代码复杂得多。例如，在界面上要用到 CICS，数据存储上会用到 VSAM 文件，数据管理上要用到 DB2 数据库，模块之间要用到程序调用等。此处只是使用一个模型，主要用于介绍增殖式调试方法的基本思想。增殖式调试方法的基本步骤如下。

（1）编写程序的一部分代码（通常最初为主功能菜单），并进行调试。

（2）编写程序的另一部分代码，并进行调试。

（3）将该段代码连接到上一代码中进行调试。

（4）继续编写下一段代码，重复以上步骤，直至整个程序编写完毕。

实际上，增殖式调试方法是将调试分散到各开发阶段之中，边开发边调试。这样做的好处主要有以下两点。

- 使错误相对独立，不会使一个模块的错误影响到另一个模块的执行。
- 能够尽早发现错误予以排除，不至于到了最后积重难返。

最后需要再次强调的是，增殖式调试方法需要贯彻到程序开发的始终。特别是对于使用 COBOL 开发的大型软件产品，这点尤其重要。

10.4 使用 DISPLAY 语句辅助调试

在程序的实际调试过程中，借助 DISPLAY 语句进行调试是一个不错的选择。在程序中使用 DISPLAY 语句辅助调试，主要是用来输出显示一些中间变量。通过中间变量的结果，可以更方便地清楚错误之所在，从而更好地进行调试。此外，用于调试的 DISPLAY 语句和用于输出的 DISPLAY 语句是不同的。用于调试的 DISPLAY 语句根据情况而临时添加，在调试完成后是应该删除的。

例如，下面这段程序根据数学中的等差数列求和公式计算 1~100 间的自然数总和。其中等差数列的求和公式如下：

```
总和 = （首项 + 末项） *  项数  /  2
```

相应的代码如下。

```
……
DATA  DIVISION.
WORKING STORAGE SECTION.
01   FIRST-ITEM       PIC    9.
01   LAST-ITEM        PIC   99.
01   ITEM-ACCT        PIC   99.
01   ITEM-SUM         PIC 9999.
*
PROCEDURE   DIVISION.
```

```
       MOVE  1  TO  FIRST-ITEM.
       MOVE  100  TO  LAST-ITEM.
       MOVE  10   TO  ITEM-ACCT.
       COMPUTE  ITEM-SUM = ( FIRST-ITEM + LAST-ITEM)
                                * ITEM-ACCT / 2
        DISPLAY  'THE SUM IS:          '  ITEM-SUM.
        DISPLAY  'THE FIRST ITEM IS:    '  FIRST-ITEM.
        DISPLAY  'THE LAST ITEM IS:     '   LAST-ITEM.
        DISPLAY  'THE ITEM ACCOUNT IS:'   ITEM-ACCT.
        STOP  RUN.
```

该段代码运行后的结果如下所示。

```
THE SUM IS:         505
THE FIRST ITEM IS:    1
THE LAST ITEM IS:     100
THE ITEM ACCOUNT IS:10
```

通过 DISPLAY 语句显示的结果，可以看到此处的最终结果错误是由于项数写错了。项数应该为 100，而此处是 10，少写了一个 0。如此，通过 DISPLAY 语句能很方便地知道程序的错误所在。下面将该程序进行了修正。当运行结果正确时，应该将用于辅助调试的 DISPLAY 语句删除掉。最终代码如下。

```
 ......
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   FIRST-ITEM       PIC    9.
 01   LAST-ITEM        PIC   99.
 01   ITEM-ACCT       PIC   99.
 01   ITEM-SUM        PIC  9999.
*
 PROCEDURE   DIVISION.
      MOVE  1  TO  FIRST-ITEM.
      MOVE  100  TO  LAST-ITEM.
      MOVE  100  TO  ITEM-ACCT.
      COMPUTE  ITEM-SUM = ( FIRST-ITEM + LAST-ITEM)
                                * ITEM-ACCT / 2
 DISPLAY  'THE SUM IS:'  ITEM-SUM.
 STOP  RUN.
```

该段代码运行后，将得到如下正确的结果。

```
THE SUM IS:  5050
```

需要说明的是，本章前面所讨论的调试与测试是针对整个程序而言的。对于整个程序，有时也会在标志部和环境部出现错误。因此，前面的示例程序都将标志部和环境部的代码包含在其中。此处只讨论如何使用 DISPLAY 语句辅助调试，只和数据部与过程部有关。因此同前面章节一样，此处将标志部和环境部进行了省略。

10.5 测试基本类型

前面主要介绍了程序调试方面的相关知识，本节开始讲解关于程序测试方面的内容。通常来说，测试大体上可以分为两种基本类型。一种类型称作黑盒测试，也叫黑箱子测试；另

一种类型称作白盒测试，也叫白箱子测试。以下分别予以讲解。

10.5.1 黑盒测试

顾名思义，黑盒测试是将系统看成一个黑盒子进行测试。此处所说的黑盒，是指测试时不用考虑程序内部的逻辑，只从程序外部对其进行测试。从程序外部进行测试，通常就是根据需求规格说明书的要求，来检查程序的功能是否符合它的功能说明。因此，黑盒测试在专业上，也可称作为功能测试或数据驱动测试。

例如，某一程序的功能是根据用户任意输入的一个数字而产生相应的输出。若输入一个负数，则输出该负数的绝对值；若输入一个正数，则判断该数为奇数还是偶数，并产生相应输出信息；若输入的既非正数也非负数，同样也产生相应的提示信息。实现以上功能的代码如下。

```
.....
DATA  DIVISION.
WORKING STORAGE SECTION.
01   TEST-NUM       PIC    S9.
01   TEMP-QUOT     PIC      9.
01   TEST-REMAIND   PIC      9.
*
PROCEDURE   DIVISION.
      ACCEPT   TEST-NUM.
      EVALUATE  TRUE
        WHEN  TEST-NUM < 0
           DISPLAY  TEST-NUM
        WHEN  TEST-NUM > 0
           PERFORM  PLUS-NUM-PROC
        WHEN  OTHER
           DISPLAY  'NOT PLUS NOR MINUS NUMBER'
     END-EVALUATE.
       STOP  RUN.
PLUS-NUM-PROC.
      DIVIDE  TEST-NUM  BY  2  GIVING  TEMP-QUOT
              REMAINDER  TEST-REMAIND.
      IF  REMAINDER = 0
             DISPLAY  ' PLUS AND EVEN NUMBER '
       ELSE
             DISPLAY  ' PLUS AND ODD NUMBER'
END-IF.
```

下面根据功能要求进行黑盒测试。若输入测试数据–9，产生以下输出信息。

```
9
```

若输入测试数据 9，产生以下输出信息。

```
PLUS AND ODD NUMBER
```

若输入测试数据 4，产生以下输出信息。

```
PLUS AND EVEN NUMBER
```

若输入测试数据 0，产生以下输出信息。

```
NOT PLUS NOR MINUS NUMBER
```

通过以上测试，得到了程序所产生的全部输出信息，完成了黑盒测试。通过黑盒测试，可以证实该程序完成了题目所指定的要求。

10.5.2 白盒测试

白盒测试是基于系统内部逻辑结构而进行的测试。进行黑盒测试时，测试人员通常只用看需求规格说明书，而不必看程序源代码。而进行白盒测试时，测试人员必须了解程序内部结构。此时不仅需要看需求规格说明书，还要看概要设计说明书，详细设计说明书以及程序源代码。白盒测试在专业上也叫结构测试或逻辑驱动测试。

由于白盒测试是建立在程序内部结构基础上的，因此根据内部结构覆盖标准不同有多种测试用例。以下分别进行介绍。先来看一个示例程序内部结构流程图，如图 10.1 所示。

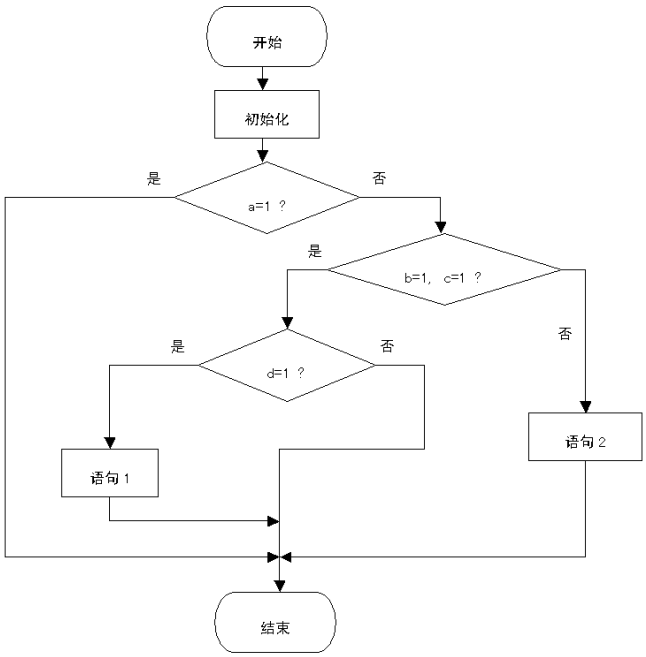

图 10.1 用于白盒测试的示例程序内部结构流程图

根据以上程序内部结构，下面分别讲解各种基于不同覆盖标准的白盒测试用例。

1．语句覆盖

语句覆盖就是选择足够的测试用例，使程序中的每条语句都能至少执行一遍。对于以上示例程序结构，由于其内部只有两条用于功能执行的语句，因此测试数据可以分为以下两组。

```
a=0, b=1, c=1, d=1
a=0, b=0/x, c=x/0, d=x
```

需要说明的是，此处为便于讨论，只假设各种数据仅存在 1 和 0 这两种取值情况。当取值为 x 时，表示既可取 0 也可取 1。此处的等号表示该测试数据的取值，反斜杠表示对应的两种情况下任取一种。例如，上面第二组测试数据实际上为下面两组数据中的任意一组。

```
a=0, b=0, c=x, d=x
a=0, b=x, c=0, d=x
```

语句覆盖实际上是一种最弱的覆盖。其覆盖标准仅能保证每条语句能够顺利执行，但无法发现判定中逻辑运算的错误。

2．判定覆盖

判定覆盖是指选择足够的测试用例，使程序中每个判定的所有可能结果都至少出现一次。对于以上示例程序结构，其中共有 3 处判定，每一个判定结果都有 2 种。因此，测试数据可以分为以下 6 组。

```
a=1, b=x, c=x, d=x
a=0, b=x, c=x, d=x
a=0, b=0/x, c=x/0, d=x
a=0, b=1, c=1, d=x
a=0, b=1, c=1, d=1
a=0, b=1, c=1, d=0
```

判定覆盖比语句覆盖要强，满足判定覆盖的测试用例一定也能够满足语句覆盖。但是，若程序中的判定是由几个条件联合构成的，不一定能发现每个条件的错误。

此外需要说明的是，以上测试数据中第 4 组数据实际上已经涵盖了第 5 组数据和第 6 组数据。这里只是为了体现逻辑上的处理顺序，在实际中是可以去掉重复的测试数据的。

3．条件覆盖

条件覆盖指选择足够的测试用例，使得判定中每个条件的所有可能结果至少出现一次。示例程序中，3 个判定内分别有 a、b、c、d 共 4 个条件。因此，测试数据可以分为以下 5 组。

```
a=1, b=x, c=x, d=x
a=0, b=1, c=0, d=x
a=0, b=0, c=1, d=x
a=0, b=1, c=1, d=0
a=0, b=1, c=1, d=1
```

条件覆盖同样也比语句覆盖要强，满足条件覆盖的测试用例也一定能够满足语句覆盖。但是，条件覆盖不一定能覆盖程序中的全部分支。

4．判定/条件覆盖

判定/条件覆盖相当于判定覆盖和条件覆盖的综合。一方面，判定/条件覆盖要求判定中每个条件的所有可能结果至少出现一次。另一方面，判定/条件覆盖同时也要求判定本身的所有可能结果至少出现一次。对于以上示例程序，此时测试数据可以分为以下几组。

```
a=1, b=x, c=x, d=x
a=0, b=x, c=x, d=x
a=0, b=0, c=1, d=x
a=0, b=1, c=0, d=x
a=0, b=1, c=1, d=x
a=0, b=1, c=1, d=1
a=0, b=1, c=1, d=0
```

判定/条件覆盖综合了判定覆盖和条件覆盖，是一种更强的覆盖。满足判定/条件覆盖的测试用例也一定能够满足条件覆盖、判定覆盖和语句覆盖。

5．条件组合覆盖

条件组合覆盖使每个判定中条件结果的所有可能组合至少出现一次。注意，此处所说的结果组合和单纯的结果是不同的。例如，对于“b=1, c=1 ?”判定中，通过两组数据便可满足其中条件的所有可能结果出现一次。但若要求满足结果的所有组合出现一次，至少需要使用4组数据。对于以上示例程序，测试数据可以分为以下几组。

```
a=1, b=x, c=x, d=x
a=0, b=x, c=x, d=x
a=0, b=0, c=0, d=x
a=0, b=0, c=1, d=x
a=0, b=1, c=0, d=x
a=0, b=1, c=1, d=x
a=0, b=1, c=1, d=0
a=0, b=1, c=1, d=1
```

条件组合覆盖既可满足判定覆盖，也可满足条件覆盖。实际上，条件组合覆盖是除路径覆盖外最强的一种覆盖。

6．路径覆盖

路径覆盖同前几种覆盖不同，并不是基于判断和条件的覆盖。路径覆盖指的是程序中每条可能执行到的路径都至少执行一次。对于以上示例程序，测试数据可以分为以下几组。

```
a=0, b=x, c=x, d=x
a=0, b=1, c=1, d=0
a=0, b=1, c=1, d=1
a=0, b=0/x, c=x/0, d=x
```

路径覆盖也是一种比较强的覆盖。但是路径覆盖并未考虑判定条件结果的组合，因此并不能代替条件覆盖和条件组合覆盖。

10.6 测试基本步骤

测试的基本步骤依次为单元测试、集成测试、确认测试、系统测试。下面以一个银行账户在线交易信息管理系统为例，分别对以上测试步骤进行讲解。

1．单元测试

在实际开发的大型软件系统中，通常是将整个程序划分为几个模块分别进行开发的。模块可以说是软件设计中的最小独立单位。单元测试正是基于程序模块而进行的测试。有的地方甚至直接将单元测试称作模块测试。

作为之前示例中曾提到的银行账户在线交易信息管理系统，可以划分为以下几个模块。

- 用户界面设计模块（在大型机平台上通常通过 CICS 完成）
- 增加新账户功能模块

- 删除账户功能模块
- 修改账户信息功能模块
- 查询账户信息功能模块
- 账户交易支出功能模块
- 账户交易收入功能模块
- 查询各种统计数据功能模块

单元测试的目的，是对以上各个功能模块进行测试，保证各个模块运行正常。单元测试通常以前面讲到的白盒测试类型为主，并辅之以黑盒测试，通过各种测试用例进行详细测试，是最基础、最底层的测试。

2．集成测试

集成测试是建立在单元测试的基础之上的。当程序各个模块通过单元测试正确无误后，便可将各模块拼装起来进行集成测试了。集成测试主要需要考虑两个问题：一个问题是各模块接口之间的数据是否能够正确传输，不被丢失；另一个问题是一个模块的功能在执行时，是否会影响到另一个模块的功能执行。集成测试也分为两种方式，分别如下。

- 一次性集成方式
- 增殖式集成方式

对于示例中的系统，若使用一次性集成方式，即将其中总共 8 个模块全部拼装起来后进行测试。相当于是对整个完整的系统进行测试，测试次数仅为 1 次，但测试复杂度较高。

若使用增殖式集成方式，则是将一个个模块逐步拼装起来进行测试的。例如，可以先将增加新账户功能模块和删除账户功能模块拼装起来进行测试。测试无误后，再将修改账户信息功能模块拼装进来进行测试。依次类推，直至将所有模块拼装并测试完毕。若按以上这种集成方式，测试次数总共需要 7 次，但每次测试的复杂度相对较低。当程序模块较多时，推荐使用增殖式集成方式进行测试。

3．确认测试

确认测试是在集成测试完成之后进行的。主要是根据需求规格说明书进行测试。确认测试的目的是验证软件产品的功能及性能等是否满足用户的需求。确认测试主要是通过黑盒测试进行。对于示例中的系统，主要是将其作为一个软件产品进行测试，并在最后请用户进行验收。

4．系统测试

系统测试是将软件产品放在实际运行环境中进行投入生产应用之前的最后一轮测试。此处所说的实际运行环境，包括实际运行的软硬件平台、接口、外设、支持软件、数据和人员等。对于示例中的系统，即将其放在实际运行环境中，如生产系统中的 S/390 或 z900 大型机上测试。

最后需要说明的是，测试作为一门单独的行业，是具有一个完整的知识体系结构的。此处只是大致介绍了一下最基本的概念和方法。由于开发和测试是紧密相关的，因此了解一下测试的知识对于学习开发是有所必要的。

10.7 数据合法性检测

数据合法性检测严格的说并不属于程序的测试，但也是与本章内容相关的。通常，输入数据很多时候并不是合乎要求的。这时便要进行数据合法性检测。数据合法性检测可以直接排除某些关于输入数据非法的错误。同时，数据合法性检测也是加强程序健壮性的一种重要方式。数据合法性检测可以分为几种不同的情况，下面分别进行介绍。

10.7.1 数字与字母检测

有的输入数据只允许为数字，如果输入字母，会产生相应的错误。此时，便可通过数据合法性检测，将输入为字母的数据排除在外，或进行相应处理。也可以加强程序的健壮性。例如，若要求某一系统中的用户编号必须为数字。下面这段代码便通过限定其用户编号数据必须为数字而实现了数据合法性检测。代码如下。

```
 ……
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   CUSTOMER-CODE    PIC   X(5).
 01   CUSTOMER-NAME    PIC   X(15).
 ……
*
 PROCEDURE   DIVISION.
     ……
     IF  CUSTOMER-CODE   IS   NOT  NUMERIC
        PERFORM   NUMERIC-ERROR-PROC
     END-IF.
……
 NUMERIC-ERROR-PROC.
      ……
STOP  RUN.
```

由此可见，检测一个数据中是否只包含数字，可通过下面两行代码进行。

```
IF   some-data   IS  NUMERIC ……          /*当 some-data 数据中全部为数字时执行相应操作*/
IF   some-data   IS  NOT  NUMERIC ……     /*当 some-data 数据中不全为数字时执行相应操作*/
```

与检测数字类似，也可进行数据是否只包含字母的合法性检测。例如，下面这段代码检测其用户姓名是否全部为从 A~Z 的 26 个字母。代码如下。

```
 ……
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   CUSTOMER-CODE    PIC   X(5).
 01   CUSTOMER-NAME    PIC   X(15).
 ……
*
 PROCEDURE   DIVISION.
     ……
     IF  CUSTOMER-NAME   IS   NOT  ALPHABETIC
        PERFORM   ALPHABETIC-ERROR-PROC
     END-IF.
```

```
      ……
ALPHABETIC-ERROR-PROC.
      ……
       STOP  RUN.
```

实际上，对于字母而言，不仅可检测某一数据是否全为字母，还可检测其是否全为大写或小写字母。关于字母的数据合法性检测主要有以下这几条。

```
 IF some-data IS ALPHABETIC ……           /*当 some-data 数据全部为字母时执行相应操作*/
IF some-data IS NOT ALPHABETIC ……        /*当 some-data 数据不全为字母时执行相应操作*/
IF some-data IS ALPHABETIC-UPPER ……      /*当 some-data 数据全部为大写字母时执行相应操作*/
IF some-data IS NOT ALPHABETIC-UPPER …… /*当 some-data 数据不全为大写字母时执行相应操作*/
IF some-data IS ALPHABETIC-LOWER ……      /*当 some-data 数据全部为小写字母时执行相应操作*/
IF some-data IS NOT ALPHABETIC-LOWER …… /*当 some-data 数据不全为小写字母时执行相应操作*/
```

10.7.2 数据正负检测

数据正负性检测主要是针对数值型数据。有时程序要求所输入的数值数据只能为正数、负数或 0。这时就需要进行数据正负性检测。

例如，对于某一超市收银系统，要求售出商品价格的数据必须为正数。通过对其数据进行正负的合法性检测，可保证达到这一要求。相应代码如下。

```
……
DATA  DIVISION.
WORKING STORAGE SECTION.
01   SALES-DATA    PIC   9(5).
……
*
PROCEDURE   DIVISION.
     ……
     IF  SALES-DATA   IS   NOT  POSITIVE
        PERFORM   SIGN-ERROR-PROC
     END-IF.
     ……
SIGN-ERROR-PROC.
      ……
       STOP  RUN.
```

实际上，关于数据正负的合法性检测，主要有以下几条。

```
 IF some-data IS POSITIVE ……             /*当 some-data 数据为正数时执行相应操作*/
 IF some-data IS NOT POSITIVE ……         /*当 some-data 数据不为正数时执行相应操作*/
 IF some-data IS NEGATIVE ……             /*当 some-data 数据为负数时执行相应操作*/
 IF some-data IS NOT NEGATIVE ……         /*当 some-data 数据不为负数时执行相应操作*/
 IF some-data IS ZERO ……                 /*当 some-data 数据为 0 时执行相应操作*/
 IF some-data IS NOT ZERO ……             /*当 some-data 数据不为 0 时执行相应操作*/
```

10.7.3 数据范围检测

通常，很多数据都是有一定范围限制的。例如，表示月份的数据必须是在 1～12 之间，表示某一天为星期几的数据必须是在 1～7 之间等。

下面这段代码对一个表示月份的数据进行了范围检测。该数据名称为 MONTH-NUM，保存 1～12 这 12 个数值数据，分别对应 1～12 这 12 个月份。代码如下。

```
……
DATA  DIVISION.
WORKING STORAGE SECTION.
01   MONTH-NUM    PIC   99.
……
*
PROCEDURE   DIVISION.
     ……
     IF   MONTH-NUM < 1 OR > 12
        PERFORM   MONTH-ERROR-PROC
     END-IF.
     ……
MONTH-ERROR-PROC.
      ……
       STOP  RUN.
```

此外，对于前面所学的 COBOL 中的表，其下标 Subscript 也是应该有一个范围的。下面这段代码便实现了对其范围的数据合法性检测。

```
……
DATA  DIVISION.
WORKING STORAGE SECTION.
01   TEST-TABLE.
     05  TABLE-ITEMS      PIC X
                          OCCURES  10  TIMES.
     05  TABLE-SUB        PIC 99   USAGE IS COMP.
……
*
PROCEDURE   DIVISION.
     ……
     IF   TABLE-SUB < 1 OR > 10
        PERFORM   SUB-ERROR-PROC
     END-IF.
     ……
SUB-ERROR-PROC.
      ……
       STOP  RUN.
```

该段代码中，所定义的表中应该有 10 个数据条目，对应 10 个 Subscript。并且，Subscript 的范围应该是从 1～10 这 10 个自然数。此处通过对其进行范围检测，排除了 Subscript 越界的潜在错误，增强了程序的健壮性。

10.7.4 数据顺序检测

有时候，输入数据是要求有一定顺序的。例如，对于一个公司的员工管理系统来说，新员工的编号其数值就应该大于老员工的编号。关于该项要求的数据顺序检测代码如下。

```
……
DATA  DIVISION.
WORKING STORAGE SECTION.
01   NEW-EMP-CODE    PIC   9(5).
01   OLD-EMP-CODE    PIC    9(5).
……
*
```

```
PROCEDURE   DIVISION.
    ……
    IF   NEW-EMP-CODE < OLD-EMP-CODE
       PERFORM   SEQUENCE-ERROR-PROC
    END-IF.
    ……
SEQUENCE-ERROR-PROC.
     ……
      STOP  RUN.
```

对于索引文件而言，后一个索引往往应该比前一个索引其数值要大。此时，同样也需要进行关于数据顺序的检测。

10.7.5 数据存在检测

此处所说的数据存在检测，主要是指检测某一变量中是否存在有数据。当变量中不存在数据时，该变量通常所包含的是空格。因此，可以通过检测某一变量是否全部为空格，从而检测该变量中的数据是否存在。例如，下面这一段代码便进行了数据存在检测。

```
……
DATA  DIVISION.
WORKING STORAGE SECTION.
01   TEST-DATA    PIC   X(5).
 ……
*
PROCEDURE   DIVISION.
    ……
    IF   TEST-DATA = SPACES
       PERFORM   EXIST-ERROR-PROC
    END-IF.
    ……
EXIST-ERROR-PROC.
     ……
      STOP  RUN.
```

最后需要说明的是，数据合法性检测实际上并不只有以上 5 种。数据合法性检测主要是为了保证输入数据为合法数据。对于不同要求的输入数据，有着不同方式的数据合法性检测。

10.8 错误信息列表

错误信息列表是将程序中所涉及到的各种错误信息，集中在一起所形成的一个 Subscript 表。错误信息列表在大型程序中经常会用到。此处所说的错误并非编码中的语法错误和逻辑错误。此处的错误通常是指程序在实际使用中产生的错误，如用户输入数据错误、程序处理过程错误等。

例如，为保证程序的健壮性，某一系统需要针对以下错误进行相应的输出提示。

- 非法的功能选择
- 输入数据为空
- 输入数据非法
- 在需要输入数字时，没有输入数字

- 在需要输入字母时，没有输入字母
- 程序在处理过程中出错

根据以上错误类型，可定义错误信息列表如下。

```
01  ERROR-MESSAGE-VALUES.
     05      PIC X(25)   VALUE  'INVALID OPTION SELECTED'
     05      PIC X(25)   VALUE  'INPUT DATA REQUIRED'
     05      PIC X(25)   VALUE  'INVALID INPUT DATA'
     05      PIC X(25)   VALUE  'DATA NOT NUMERIC'
     05      PIC X(25)   VALUE  'DATA NOT ALPHABETIC'
     05      PIC X(25)   VALUE  'PROCESSING ERROR'
 01  ERROR-MESSAGE-TABLE
     REDEFINES  ERROR-MESSAGE-VALUES.
     05      ERROR-MESSAGE   PIC X(25)   OCCURS 6 TIMES.
```

由此可见，错误信息列表实际上先是将各条错误输出信息列在一起。在所有的错误输出信息列完后，再通过 REDEFINES 将其定义为一张表。错误信息列表在程序中的使用方式如下。

```
 ......
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01  ERROR-MESSAGE-VALUES.
     05      PIC X(25)   VALUE  'INVALID OPTION SELECTED'
     05      PIC X(25)   VALUE  'INPUT DATA REQUIRED'
     05      PIC X(25)   VALUE  'INVALID INPUT DATA'
     05      PIC X(25)   VALUE  'DATA NOT NUMERIC'
     05      PIC X(25)   VALUE  'DATA NOT ALPHABETIC'
     05      PIC X(25)   VALUE  'PROCESSING ERROR'
 01  ERROR-MESSAGE-TABLE
     REDEFINES  ERROR-MESSAGE-VALUES.
     05      ERROR-MESSAGE   PIC X(25)   OCCURS 6 TIMES.
 01  TEST-NUM   PIC X.
 01  PRINT-ERROR-MESSAGE   PIC X(25).
......
*
PROCEDURE   DIVISION.
     ......
     ACCEPT  TEST-NUM
     IF  TEST-NUM = SPACES
         MOVE  ERROR-MESSAGE  (2)  TO PRINT-ERROR-MESSAGE
     END-IF.
     ......
     IF  TEST-NUM  IS  NOT  NUMERIC
         MOVE  ERROR-MESSAGE  (4)  TO PRINT-ERROR-MESSAGE
     END-IF.

      ......
       STOP  RUN.
```

由此可见，使用错误信息列表后，在程序中可直接通过引用表中数据的方式得到相关的错误信息。使用错误信息列表主要有以下 3 点好处。

- 明确程序在运行中会产生哪些错误信息。
- 可对相应错误信息的内容进行统一修改。

- 当程序中有多处出现同类错误时，可以保证错误信息的一致性。

10.9 本章回顾

本章首先介绍了程序调试与测试的基本概念及其重要性。调试与测试在平时容易被混淆，实际上二者既有联系，也有区别。

本章接下来针对程序调试进行了相应讲解。学习程序调试，需要理解语法错误和逻辑错误各自的特点，理解增殖式调试方法的思想原理，掌握如何使用 DISPLAY 语句辅助调试。

关于程序测试，本章主要讲解了测试的基本类型和步骤。测试基本类型分为黑盒测试和白盒测试两类。对于白盒测试，需要理解测试用例所对应的不同覆盖标准，其中包括语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、条件组合覆盖以及路径覆盖。测试基本步骤依次为单元测试、集成测试、确认测试、系统测试。学习程序测试，对以上 4 个测试步骤也是需要了解的。

本章在最后介绍了数据合法性检测和错误信息列表。这两方面内容和程序的调试与测试是有一定关联的。并且，这两方面内容在 COBOL 实际开发中是会经常用到的。在 COBOL 实际开发中，要养成对输入数据进行数据合法性检测的习惯。对于大型 COBOL 程序而言，要学会使用错误信息列表对各种错误信息进行统一的管理。

第 11 章 子程序调用

一个大型的软件产品通常是由多个程序所成的。其中一个程序为主程序，该程序实现系统主体框架，并调用其他程序完成系统各个模块的功能。被调用的程序即称作子程序。本章将讲解子程序调用方面的知识，为开发大型软件产品奠定基础。

11.1 子程序调用的作用

在大型软件产品开发中，为提高开发进度，保证产品质量，通常需要将其划分为几个模块分别开发的。不同的模块间，程序通常是相对独立的。而为将其完整的连接成一个整体，需要用到子程序调用。

除此之外，有时在同一个模块内，也是存在子程序调用的。通常来说，子程序调用的用途主要有以下几点，下面分别进行介绍。

11.1.1 提高代码可重用性

对于某一部分功能代码，有时可能会在多个程序中被反复用到。最好的方式是将该功能代码编写为一个独立的子程序，通过子程序调用进行使用。

例如，下面这两段程序都要到一个关于利率计算的功能段。假设利率为 1.5，通过 RATE 变量表示，并假设第一段程序代码如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID     TEST-PROG-ONE.
 AUTHER         XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01  RATE               PIC  S9V9.
 01  ORIGIN-DATA1        PIC  S9(5).
 01  NEW-DATA1          PIC  S9(5).
```

```
……
*
 PROCEDURE   DIVISION.
     ……
     PERFORM  100-COMPUTE-RATE.
     ……
     STOP RUN.
 ……
 100-COMPUTE-RATE.
     MOVE 1.5 TO RATE.
     MULTIPLY  ORIGIN-DATA1  BY  RATE  GIVING  NEW-DATA1.
```

另一段程序代码如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID      TEST-PROG-TWO.
 AUTHER          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01  RATE               PIC  S9V9.
 01  ORIGIN-DATA2       PIC  S9(5).
 01  NEW-DATA2          PIC  S9(5).
  ……
*
 PROCEDURE   DIVISION.
     ……
     PERFORM  100-COMPUTE-RATE.
     ……
     STOP RUN.
 ……
 100-COMPUTE-RATE.
     MOVE 1.5 TO RATE.
     MULTIPLY  ORIGIN-DATA2  BY  RATE  GIVING  NEW-DATA2.
```

通过比较以上两段程序可以发现，这两段程序中利率计算的功能段代码实际上是类似的。虽然以上两段程序都将该功能段代码写为了一个处理过程，便于程序内通过 PERFORM 语句调用。但是，若将这两段代码看作一个整体，仍然对利率计算的功能段代码进行了重复编写，不够简便。因此，考虑到子程序调用技术，可以先将利率计算的功能段代码编写为如下一个小的程序。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID     SUBPROG.
 AUTHER         XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01  RATE               PIC  S9V9.
 LINKAGE SECTION.
 77  ORIGIN-DATA         PIC   S9(5).
```

```
 77  NEW-DATA           PIC   S9(5).
*
 PROCEDURE   DIVISION  USING  ORIGIN-DATA  NEW-DATA.
     MOVE 1.5 TO RATE.
     MULTIPLY  ORIGIN-DATA  BY  RATE
              GIVING  NEW-DATA.
       GOBACK.
```

此时，之前的两段程序便可通过子程序调用的方式重复实现利率计算的功能了。根据以上子程序，可以将之前的第一段程序改写如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID      TEST-PROG-ONE.
 AUTHER          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 77  ORIGIN-DATA1         PIC  S9(5).
 77  NEW-DATA1            PIC  S9(5).
  ......
*
 PROCEDURE   DIVISION.
     ......
     CALL  'SUBPROG'   USING  ORIGIN-DATA1
                              NEW-DATA1.
     ......
     STOP RUN.
```

第二段程序相应的改写如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID     TEST-PROG-TWO.
 AUTHER         XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 77  ORIGIN-DATA2         PIC  S9(5).
 77  NEW-DATA2           PIC  S9(5).
  ......
*
 PROCEDURE   DIVISION.
     ......
     CALL  'SUBPROG'   USING  ORIGIN-DATA2
                              NEW-DATA2.
     ......
     STOP RUN.
```

可以看到，使用子程序调用后，以上两段程序中都不用重复编写利率计算这一功能段代码了。该功能是直接通过调用利率计算的子程序所实现的。基于子程序调用技术实现的正是一次编码，多次使用。并且，当使用同一功能段代码的程序越多，采用子程序调用方式的优势就越明显。

11.1.2 提高部分功能段执行效率

COBOL 可调用的子程序并不一定要求由 COBOL 所编写，可以由其他语言编写，如汇编语言、PL/1 语言等。

对于某些功能，使用其他语言编写往往执行效率会更高。例如，在大型机环境下使用 S/390 汇编语言编写的程序，通常运行效率就比使用 COBOL 编写的要高。但是，汇编语言编写起来是不如 COBOL 方便的。因此通常可以用 COBOL 编写主程序，而将其中经常使用的部分功能段使用汇编语言编写。这样既不会过于增大开发难度，也能从整体上提高程序的运行效率。

例如，下面这段 COBOL 程序便调用了汇编程序。其中所调用的程序名为 ASMPGM，传递的参数依次为 DATA1、DATA2、DATA3。代码如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID       CALL-ASM-PROG.
 AUTHER           XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 77  DATA1         PIC  X.
 77  DATA2         PIC  X.
 77  DATA3         PIC  X.
  ......
*
 PROCEDURE   DIVISION.
      ......
      CALL  'ASMPGM'   USING  DATA1
                              DATA2
                              DATA3.
      ......
      STOP RUN.
```

调用的汇编程序代码如下。

```
ASMPGM  CSECT
      STM  R14,R12,12(R13)
      BALR  R12,R0
      USING  *,R12
      LM  R2,R4,0(R1)
      ......
      BCR  B'1111',14
      DC  A(PARM1)
      DC  A(PARM2)
      DC  X'80',AL3(LASTPARM)
      END  ASMPGM
```

该汇编程序中，R2、R3、R4 这 3 个寄存器将分别与 COBOL 中的 3 个参数相对应。其中 R2 与 DATA1 相对应，R3 与 DATA2 相对应，R4 与 DATA3 相对应。

此处主要是为了说明通过 COBOL 可以调用其他语言编写的子程序，并以此提高程序整体执行效率。关于大型机上的汇编语言将在后面的章节中进行讲解，这里不必深究。

11.1.3 防止数据意外丢失或被更改

COBOL 程序中的数据通常都为全局性数据。因此当程序较大、数据较多时，某些数据在程序运行中就有可能会意外丢失或被更改。如果使用子程序调用，只将某一功能要使用的数据传递给相应的子程序，能较好的防止此类问题。

例如，下面这段程序中便用到了较多的数据，代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID      TEST-PROG.
 AUTHER          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01  TEST-DATA.
     05   TEST-DATA-1   PIC X.
     05   TEST-DATA-2   PIC X.
     05   TEST-DATA-3   PIC X.
     ......
     05   TEST-DATA-N   PIC X.
*
 PROCEDURE   DIVISION.
     process TEST-DATA-1.
     process TEST-DATA-2.
     process TEST-DATA-3.
     ......
     process TEST-DATA-N.
      STOP RUN.
```

此时，在同一程序中同时处理过多的数据，容易造成部分数据意外丢失或被更改。如果将部分涉及到数据处理的功能段独立起来，编写为子程序，便可通过子程序减少此类错误发生。因为此时数据分散到各个子程序之间进行处理，增强了数据间的相对独立性。并且，对于每一个子程序而言，处理的数据相对来说也不太多。使用子程序调用方式后的主程序代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID      TEST-PROG.
 AUTHER          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01  TEST-DATA.
     05   TEST-DATA-1   PIC X.
     05   TEST-DATA-2   PIC X.
     05   TEST-DATA-3   PIC X.
     ......
     05   TEST-DATA-N   PIC X.
*
 PROCEDURE   DIVISION.
```

```
        CALL  'SUBPGM1'   USING  TEST-DATA-1
                                 TEST-DATA-2
                                 TEST-DATA-3.
        CALL  'SUBPGM2'   USING   TEST-DATA-4
                                 TEST-DATA-5.
        ......
        process TEST-DATA-N.
         STOP RUN.
```

11.2 子程序调用的特点

本节仅讨论子程序调用最基本的特点以及一些常见的注意事项。关于子程序调用更详细的特点及应用，将在后面的章节中陆续进行讲解。此处主要从子程序的命名规则、子程序的调用顺序以及子程序的终止方式这 3 个方面进行讲解。

11.2.1 子程序的命名规则

子程序的命名通常和普通程序的命名方式相同。但是需要注意的是，对于子程序而言，不可将其前缀命名为以下几个名字。

```
AFB      AFH      CBC      CEE      EDC
IBM      IFY      IGY      IGZ      ILB
```

以上名字都属于 IBM 相关产品的名字。如果将子程序的前缀命名为以上名字，在主程序中将不能对其进行调用。当在主程序中试图调用该子程序时，系统将会从 IBM 的库，或者编译器例程中寻求解决方案。

例如，以下这种子程序调用便是错误的。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID      EDC-SUB-PROG.
 AUTHER          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
     ......
 LINKAGE SECTION.
     define some parameters
     ......
*
 PROCEDURE   DIVISION  USING  some parameters.
     ......
       GOBACK.
```

主程序代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID      TEST-MAIN-PROG.
 AUTHER          XXX.
*
 ENVIRONMENT   DIVISION.
```

```
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 77   PARM-DATA   PIC X.
      ……
*
 PROCEDURE   DIVISION.
      ……
      CALL 'EDC-SUB-PROG'  USING  PARM-DATA.
      ……
       STOP RUN.
```

以上被调用的子程序名字为“EDC-SUB-PROG”。由于其前缀为“EDC”，属于 IBM 的保留字，因此该子程序不能被调用。

11.2.2 子程序的调用顺序

首先需要说明的是，子程序实际上就是普通的程序。当某一程序被其他程序调用时，该程序才相对于调用的程序被称作子程序。因此，当某一程序作为子程序被调用后，该程序也可再将其他程序作为子程序进行调用。

当多个程序进行调用时，便存在一个调用顺序的问题。关于子程序的调用顺序，通常是提倡进行线性调用的。

例如，以下这种调用顺序对于初学者是提倡的。

- 程序 A 调用程序 B
- 程序 B 调用程序 C
- 程序 C 调用程序 D

而以下这种调用顺序对于初学者则是不被提倡的。

- 程序 A 调用程序 B
- 程序 B 调用程序 C
- 程序 C 调用程序 A

以上这种方式虽然不被提倡，但也是允许存在的。这种调用方式也叫递归调用。指的是程序直接或间接的自己调用自己。当使用递归调用时，需要在该程序的 PROGRAM-ID 段内加上 RECURSIVE 子句。否则，当进行递归调用时，将会出现相应的条件提示符。如果该条件提示符没有被清除，该程序单元将停止运行。

总之，对于初学者而言，是提倡以线性方式调用程序的。递归调用涉及到的算法和参数传递较为复杂，可在今后的学习中逐步掌握。

11.2.3 子程序的终止方式

当某一子程序被调用运行时，该程序通常有以下 3 种发展方向。

- 调用另一个子程序。
- 终止本子程序，并返回上一层主调用程序。
- 终止本子程序，同时退出整个系统。

关于第一条发展方向，实际上在上一小节中已有所提到。此处重点看后两种发展方向。其中对于第 2 种发展方向，主要是通过 GOBACK 或 EXIT PROGRAM 语句实现的。对于第 3

种发展方向，主要是通过 STOP RUN 语句实现的。下面分别就以上 3 种语句进行讲解。

1．GOBACK 语句

GOBACK 语句作为程序的结束标志，既可以在子程序中使用，也可以在主程序中使用。二者的共同之处都是终止程序，但也有各自的区别。

- 对于子程序而言，使用 GOBACK 语句有两方面作用。其中一方面作用是表示该子程序在此处终止。另一方面作用是表示该子程序终止后，将把控制权返回给主程序或是上一层子程序。
- 对于主程序而言，GOBACK 语句通常表示返回到操作系统。此时，包含该程序的作业将结束运行。实际上，在主程序中使用 GOBACK 语句和使用 STOP RUN 语句是等效的。

2．EXIT PROGRAM 语句

当在子程序中使用 EXIT PROGRAM 语句时，表示直接返回到主程序或上一层子程序中。并且，保持此时的运行单元不被终止。当子程序中所有语句执行完毕后，将会产生并执行一条隐含的 EXIT PROGRAM 语句。在主程序中使用 EXIT PROGRAM 语句时，不会产生任何动作。因此，通常不在主程序中使用 EXIT PROGRAM 语句。

3．STOP RUN 语句

在子程序中使用 STOP RUN 语句时，通常将直接返回到操作系统中。并且，此时相应的作业也将结束运行。当在主程序中使用 STOP RUN 语句时，通常也是直接返回到操作系统。并且，STOP RUN 语句将终止整个运行单元。同时，该语句还将把所有动态调用的子程序，以及相应的编译连接程序从运行单元中移除掉。但需要注意的是，该语句并不删除主程序。

11.3 主调用程序

前面讲到，子程序作为一个独立的程序，也可继续调用其他子程序。从调用关系的角度出发，将程序分为主调用程序和被调用程序两种。其中主调用程序为进行子程序调用的程序，既可以为主程序，也可以为子程序。被调用程序则只可以为子程序。本节主要讲解主调用程序。

11.3.1 主调用程序中参数的定义

当主调用程序进行子程序调用时，通常需要向子程序传递参数。主调用程序所传递的参数一般情况下，可以为在该程序内所定义的任何数据。这些数据可以在主调用程序数据部的以下几个节中进行定义。

- 文件节：此时在 FD 语句下面进行定义，数据内容通过读取相关文件得到。
- 工作存储节：在该节内可直接进行定义，数据内容通过在程序的处理过程中得到。
- 连接节：在该节内的定义方式同在工作存储节类似，但数据内容通过其他程序所传递的参数得到。并且，这些数据将作为该主调用程序的参数再次传递给其所调用的程序。

下面来看一个主调用示例程序，重点看里面的参数定义，代码如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID      CALLING-PROG.
 AUTHER          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   PARM-GROUP.
      05   PARM1   PIC  X.
      05   PARM2   PIC  X(7).
      05   PARM3    PIC  99.
 77   PARM-NUMBER   PIC 9(5).
      ……
*
 PROCEDURE   DIVISION.
      ……
       STOP RUN.
```

可以看到，以上主调用程序 CALLING-PROG 中的参数数据都是在工作存储节内定义的。关于主调用程序中的参数，主要有以下几点需要注意。

- 用于传递的参数通常必须定义在 01 层或 77 层。例如，以上程序中的参数 PARM-GROUP 就定义在 01 层，而参数 PARM-NUMBER 定义在 77 层。
- 在主程序中定义的参数与该参数所定义的位置和顺序是有关系的。例如，以上程序中参数 PARM-GROUP 是在参数 PARM-NUMBER 前面进行定义的。这种顺序通常不可更改。
- 参数是在主调用程序中被分配存储空间的，并不在被调用程序中进行分配。例如，以上程序中共分配 10 个字节大小的存储空间给第一个参数。而第二个参数则被分配 5 个字节大小的存储空间。
- 在程序调用的参数传递过程中，缺省情况下所传递的是参数的地址，而不是参数的内容。因此，可以通过被调用程序改变该参数的实际内容。
- COBOL 程序将自动把最后一个参数的高位置为 1，作为参数传递的结束标志。从此，COBOL 程序便可支持变长参数队列的传递。

11.3.2 主调用程序中的调用过程

在主调用程序中进行子程序调用时，主要是通过 CALL 语句进行的。例如，下面这段程序就使用 CALL 语句实现了程序调用，代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID      MAIN-CALLING-PGM.
 AUTHER          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   PM-LIST.
      05   PM-SIGN   PIC  X.
      05   PM-DATA   PIC  X(5).
```

```
 77   PM-NUM    PIC 99.
      ......
*
 PROCEDURE   DIVISION.
      ......
      CALL  'SUB-CALLED-PGM'  USING  PM-LIST
                                     PM-NUM.
      ......
       STOP RUN.
```

以上是在主调用程序中对被调用程序最常见的调用方式。其中主调用程序名为 MAIN-CALLING-PGM，被调用程序名为 SUB-CALLED-PGM。在调用中所传递的参数依次为 PM-LIST 和 PM-NUM。

关于在主调用程序中使用 CALL 语句进行程序调用，有以下几点需要注意。

- CALL 短语是 CALL 语句的主体部分，用以指明被调用的程序。
- USING 短语通常也是 CALL 语句的必备短语，用以指明传递给被调用程序的参数。
- 被调用程序名通常在 CALL 短语后使用单引号括起来。若被调用的程序是 COBOL 程序，则程序名在该程序中通过 PROGRAM-ID 指明。若被调用的程序是 ASM 汇编程序，则程序名通过 CSECT 指明。

此外，在 CALL 语句中也可加上 ON OVERFLOW 短语。该短语通常只应用于动态调用。当程序在运行时，如果被调用程序不能被动态地载入，则 ON OVERFLOW 短语后的语句将被执行。包含有 ON OVERFLOW 短语的 CALL 语句示例如下。

```
CALL  'subprogram name'   USING  parm 1
                                 parm 2
                                 ......
                                 parm n
                ON OVERFLOW  imperative statement.
```

11.4 被调用程序

前面讲到了在调用关系中处于主动地位的程序，即主调用程序。和主调用程序相对应，在调用关系中还存在着被调用程序。被调用程序在调用关系中是处于被动地位的。本节将从 3 个方面来介绍被调用程序。

11.4.1 被调用程序中参数的定义

被调用程序的参数都是在其连接节中定义的。前面讲到，在主调用程序中也可存在连接节，但在主调用程序中的连接节是可选的。当该主调用程序同时也作为被调用程序时才使用连接节。而在被调用程序中，存在于数据部的连接节通常是必须要求存在的。连接节在程序代码中对应为 LINKAGE SECTION。

被调用程序中的连接节主要用于描述传递给该程序的各项参数。需要注意的是，在连接节中所定义的参数数据是不被分配存储空间的。通常，在子程序调用中所传递的参数都是在主调用程序中被分配存储空间的。

例如，下面为一段被调用程序的代码，重点看里面连接节内的参数数据定义。代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID      CALLED-PROG.
 AUTHER          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 ......
 LINKAGE SECTION.
 77   PARM-NUM   PIC 9(5).
 01   PARM-LIST.
      05   PM1    PIC  X(8).
      05   PM2    PIC  XX.
*
 PROCEDURE   DIVISION  USING  PARM-LIST  PARM-NUM.
     ......
      GOBACK.
```

被调用程序和主调用程序都是成对出现的，二者构成一个整体，不可分割。在讨论被调用程序时，必须结合相应的主调用程序进行讲解。例如，以上被调用程序实际上可以对应下面这段主调用程序，该主调用程序代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID      CALLING-PROG.
 AUTHER          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   PARM-GROUP.
      05   PARM1    PIC  X.
      05   PARM2    PIC  X(7).
      05   PARM3    PIC  99.
 77   PARM-NUM     PIC 9(5).
     ......
*
 PROCEDURE   DIVISION.
     ......
     CALL   'CALLED-PROG'   USING   PARM-GROUP
                                    PARM-NUM.
     ......
     STOP RUN.
```

结合以上主调用程序，关于示例中被调用程序的参数定义主要有以下几点需要注意。

（1）被调用程序和主调用程序中对应的参数命名既可相同，也可不同。

例如，以上程序调用中共有两个参数被传递。其中被调用程序中的参数 PARM-NUM 对应主调用程序中的参数 PARM-NUM。此时二者的命名是相同的。

被调用程序中的另一个参数 PARM-LIST 对应主调用程序中的参数 PARM-GROUP。此时二者的命名则是不同的。

（2）被调用程序和主调用程序中对应的参数可定义为不同的数据类型。

例如，以上被调用程序中，参数 PARM-LIST 的最后两个字节被定义为字符类型数据。该字符类型数据通过变量 PM2 引用，定义方式如下。

```
    05   PM2     PIC  XX.
```

该部分所对应的为主调用程序中参数 PARM-GROUP 的最后两个字节内容。在主调用程序中的这部分内容通过变量 PARM3 引用，并且被定义为一个整型数类型数据。定义方式如下。

```
        05   PARM3    PIC  99.
```

因此，以上二者中一个被定义为字符类型数据，另一个被定义为整型数类型数据。虽然数据类型不同，但所占存储空间大小是一致的，都为 2 个字节。这种情况是允许的。

（3）被调用程序和主调用程序中对应的参数可定义为不同的组织形式。

例如，以上被调用程序中，参数 PARM-LIST 是由两个字符型数据所组成的。该参数的组织形式定义如下。

```
01   PARM-LIST.
     05   PM1     PIC  X(8).
     05   PM2     PIC  XX.
```

而在主调用程序中，与之对应的是参数 PARM-GROUP。该参数是由两个字符型数据和一个整型数数据所组成的，其组织形式定义如下。

```
01   PARM-GROUP.
     05   PARM1    PIC  X.
     05   PARM2    PIC  X(7).
     05   PARM3    PIC  99.
```

由以上定义形式看出，这两个对应的参数在各自的程序中组织形式并不相同。但是，这两个参数的总长度都是一样的，都为 10 个字节。这种情况是允许的。此时，被调用程序中的 PM1 参数变量对应于主调用程序中的 PARM1 和 PARM2。而被调用程序中的 PM2 参数变量，则对应于主调用程序中的 PARM3。

此外，不仅可使 PM1 参数变量同时对应两个变量，甚至还可以将 PM1 定义为表的形式。例如，以下这种定义方式也是允许的。

```
01   PARM-LIST.
     05   PM1     PIC  X
                    OCCURS 8 TIMES.
     05   PM2     PIC  XX.
```

总之，被调用程序和主调用程序间对应的参数通常只要求其所占存储空间大小保持一致。至于命名方式、数据类型、组织形式都可以不同。

最后需要注意的是，与主调用程序不同，被调用程序中参数定义的顺序是不做特别要求的。例如，对于以上被调用程序中的参数，以下这两种定义方式通常是等效的。其中一种定义方式如下。

```
77   PARM-NUM   PIC 9(5).
01   PARM-LIST.
     05   PM1     PIC  X(8).
     05   PM2     PIC  XX.
```

另一种定义方式如下。

```
01   PARM-LIST.
     05   PM1     PIC  X(8).
     05   PM2     PIC  XX.
77   PARM-NUM   PIC 9(5).
```

11.4.2 被调用程序中参数的引用

被调用程序是在过程部标志 PROCEDURE DIVISION 后实现参数引用的。引用方式仍然是通过 USING 语句。实际上，这种引用方式在前面的被调用程序代码中已有所涉及。此处不妨以上一小节中的被调用程序为例，重点关注其对于参数的引用。该程序代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID      CALLED-PROG.
 AUTHER          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 ......
 LINKAGE SECTION.
 77   PARM-NUM   PIC 9(5).
 01   PARM-LIST.
      05   PM1     PIC  X(8).
      05   PM2     PIC  XX.
*
 PROCEDURE   DIVISION  USING  PARM-LIST  PARM-NUM.
     ......
      GOBACK.
```

关于以上被调用程序中的参数引用，有以下两点需要注意。

1. 参数引用的顺序必须和主调用程序中相应参数的定义一致

前面曾提到，在被调用程序中，参数只用主调用程序中对应的参数空间大小定义为一致。其他的诸如命名方式、数据类型、组织结构都可以不同。但是，并不可能只根据大小便将两个程序中的参数相对应起来。因为各程序中的参数之间有可能很多大小是相同的。

因此，为将两程序间的参数相对应起来，必须通过被调用程序中参数的引用顺序来实现。对于以上被调用程序，其主程序中的参数是按如下顺序定义的。

```
 01   PARM-GROUP.
      05   PARM1    PIC  X.
      05   PARM2    PIC  X(7).
      05   PARM3    PIC  99.
 77   PARM-NUM     PIC 9(5).
```

其中参数 PARM-GROUP 在前，参数 PARM-NUM 在后。而在此处的被调用程序中，参数 PARM-LIST 是和 PARM-GROUP 对应的，参数 PARM-NUM 和 PARM-NUM 对应。因此，在引用时，应该根据顺序，将 PARM-LIST 放在前面，PARM-NUM 放在后面。如此，便实现了被调用程序和主调用程序间参数的一一对应。

2．引用的参数名为被调用程序中的参数名，但内容为主调用程序中的参数

对于以上被调用程序，所引用的参数名 PARM-LIST 和 PARM-NUM 都是在连接节内所定义的。然而，前面曾提到，参数在被调用程序中是不被分配存储空间的。因此，以上引用的参数名虽然为该被调用程序中所定义的，但实际内容则为主调用程序中的参数。

这种方式实际上使被调用程序被多个主调用程序所调用成为可能。例如，仍然保持以上被调用程序代码不变，可使用另一个主调用程序对其进行调用。不妨假设新的主调用程序代码如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID     NEW-CALLING-PROG.
 AUTHER         XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   NEW-GROUP.
     05   NEW1    PIC  9(9).
     05   NEW3    PIC  X.
 77   NEW-NUM     PIC  X(5).
      ......
*
 PROCEDURE   DIVISION.
      ......
      CALL   'CALLED-PROG'   USING   NEW-LIST
                                     PARM-NUM.
      ......
       STOP RUN.
```

对于以上主调用程序，此时该被调用程序中所引用的参数将有所不同。被调用程序中，参数 PARM-LIST 将对应参数 NEW-GROUP。而参数 PARM-NUM 将对应参数 NEW-NUM。这样，对于不同的主调用程序，将有不同的参数引用。而被调用程序的代码则始终不变，从而实现了被调用程序的一次编写、多次调用。

11.4.3 被调用程序中的入口地址

使用 COBOL 语言编写的程序调用通常只用提供被调用程序的程序名以及所传递的参数。真正在系统内部实现的程序调用，是按以下步骤执行的。

- 保护现场。
- 访问被调用程序的入口地址。
- 进入并执行被调用程序。
- 退出被调用程序。
- 恢复现场。

由此可见，被调用程序的入口地址在程序调用中是十分重要的。该入口地址决定了在何处能够访问到被调用程序。通常在 COBOL 高级语言中，被调用程序的入口地址是通过程序名反映出来的。例如，下面为一段被调用程序代码。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID      CALLED-PGM.
 AUTHER          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
    ……
 LINKAGE SECTION.
 77  CALLED-PARM    PIC  X.
*
 PROCEDURE   DIVISION  USING  CALLED-PARM.
     ……
      GOBACK.
```

以上程序的入口地址直接通过编译后的程序名反映出来。该程序名为 CALLED-PGM。因此，在主调用程序中使用 CALL 语句对其调用的方式如下。

```
   CALL  'CALLED-PGM'  USING  CALLING-PARM.
```

除此之外，还可在被调用程序中通过 ENTRY 语句指定选择性入口地址。主调用程序同样可以通过该选择性入口地址实现程序的调用。当在以上被调用程序中通过 ENTRY 语句指定选择性入口地址时，该程序代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID     CALLED-PGM.
 AUTHER         XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
     ……
 LINKAGE SECTION.
 77  CALLED-PARM    PIC  X.
*
 PROCEDURE   DIVISION  USING  CALLED-PARM.
     ……
       GOBACK.
     ENTRY  'ALTENTRY'   USING  CALLED-PARM.
     ……
      GOBACK.
```

在主调用程序中，便可通过使用 ENTRY 语句定义的选择性入口地址实现程序调用。调用方式如下。

```
   CALL  'ALTENTRY'   USING  CALLING-PARM.
```

11.5 静态调用

在子程序调用过程中，通常有两种不同的调用方式。一种为静态调用，另一种为动态调用。本节将讲解静态调用。

11.5.1 静态调用的基本概念

静态调用发生时通常需要具有两个条件：一个条件是使用 CALL literal，另一个条件是程序被编译时使用 NODYNAM 和 NODLL 选项。静态调用最基本的特征是子程序和主程序一并链接，存放于同一个加载模块（Load Module）中。即子程序在被调用之前，已经存在于内存之中了。

基于这一点，在静态调用中，主要有以下几个方面需要注意。

- 当某一程序被多次静态调用时，每次调用时该程序的状态为其最近一次被调用之后的状态。即在多次通过静态调用同一程序时，该程序每次的状态可能并不相同。这种状态主要表现在该程序的程序状态字及其内部数据上。因此，若要使该程序每次被调用时，内部数据都相同，则需要在每次调用前都对其进行初始化。
- 静态调用所占的存储空间通常相对比较大。主要是因为在调用程序后，不能将其从内存中移除掉以释放存储空间。而对于在后面要讲解的动态调用中，调用结束后是可以通过 CANCEL 语句，将其从内存中移除掉的。
- 静态调用的效率相对比较高。这是因为在静态调用时，子程序已存在于内存之中了，不用反复读写内存，因此提高了效率。

11.5.2 静态调用程序示例

下面通过具体的程序示例，以便更好地说明静态调用的特点及用法。首先，假设在静态调用中，主程序代码如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID     STATIC-MAIN.
 AUTHER         XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 77   TEST-NUM    PIC  99.
*
 PROCEDURE   DIVISION.
     PERFORM  INIT-TEST-NUM.
     CALL   'STATIC-SUB'   USING    TEST-NUM.
     DISPLAY 'TEST-NUM AFTER THE FIRST CALL: '  TEST-NUM.
     PERFORM  INIT-TEST-NUM.
     CALL   'STATIC-ENTRY'   USING  TEST-NUM.
     DISPLAY 'TEST-NUM AFTER THE SECOND CALL: '  TEST-NUM.
       STOP RUN.
 INIT-TEST-NUM.
     MOVE  10  TO  TEST-NUM.
```

令该程序的子程序，即与之所对应的被调用程序 STATIC-SUB 的代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID     STATIC-SUB.
 AUTHER         XXX.
*
```

```
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   PLUS-NUM    PIC  99   VALUE  15.
 LINKAGE SECTION.
 77   MAIN-NUM    PIC  99.
*
 PROCEDURE   DIVISION   USING  MAIN-NUM.
     ADD   MAIN-NUM  TO  PLUS-NUM.
     MOVE  PLUS-NUM  TO  MAIN-NUM.
       GOBACK.
     ENTRY  'STATIC-ENTRY'   USING  MAIN-NUM.
     ADD   MAIN-NUM  TO  PLUS-NUM.
     MOVE  PLUS-NUM  TO  MAIN-NUM.
       GOBACK.
```

以上程序运行后，将有如下输出结果。

```
TEST-NUM AFTER THE FIRST CALL: 25
TEST-NUM AFTER THE SECOND CALL: 35
```

从上述代码中可以看出，在第一次调用前，子程序中的变量 PLUS-NUM 通过 VALUE 语句初始化为 15。将该变量中的 15 和主程序中所传递的参数 TEST-NUM 中的 10 相加后，结果将为 25。并且，此时子程序中的变量 PLUS-NUM 在相加运算结束后已由 15 变成了 25。

当进行第二次调用时，子程序为第一次调用结束后的状态。此时，子程序中的变量 PLUS-NUM 为 25，而并非如第一次调用前的 15。虽然第二次实际上仍然是对于子程序 STATIC-SUB 的调用，但此时该程序中的数据已不同了。第二次调用是将 PLUS-NUM 中的 25 和 TEST-NUM 中的 10 相加，因此最终结果为 35。

若要希望两次调用时子程序的状态都一致，需要在子程序中进行相应的初始化。此处所说的初始化通常是指在该程序中的每一个入口地址后，对工作存储节中的本地数据进行初始化。进行初始化后的子程序如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID      STATIC-SUB.
 AUTHER          XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   PLUS-NUM    PIC  99   VALUE  15.
 LINKAGE SECTION.
 77   MAIN-NUM    PIC  99.
*
 PROCEDURE   DIVISION   USING  MAIN-NUM.
     PERFORM  INIT-PLUS-NUM.
     ADD   MAIN-NUM  TO  PLUS-NUM.
     MOVE  PLUS-NUM  TO  MAIN-NUM.
       GOBACK.
    ENTRY  'STATIC-ENTRY'   USING  MAIN-NUM.
     PERFORM  INIT-PLUS-NUM.
```

```
        ADD   MAIN-NUM  TO  PLUS-NUM.
        MOVE  PLUS-NUM  TO  MAIN-NUM.
          GOBACK.
    INIT-PLUS-NUM.
        MOVE  15  TO  PLUS-NUM.
```

仍然采用前面的主程序对以上子程序进行调用，运行后的结果将如下。

```
TEST-NUM AFTER THE FIRST CALL: 25
TEST-NUM AFTER THE SECOND CALL: 25
```

总之，静态调用的程序每次调用前都为其上一次调用后的状态。这一点是关于静态调用最需注意的地方，一定要牢记。

11.6 动态调用

动态调用是子程序调用中的另一种调用方式，和前面所讲的静态调用是对应的。本节将对动态调用进行讲解。

11.6.1 动态调用的基本概念

动态调用发生的情况有两种。一种情况是使用 CALL literal，并且程序被编译时使用 DYNAM 和 NODLL 选项。另一种情况是使用 CALL identifier，并且程序被编译时使用 NODLL 选项。而且，当使用 CALL identifier 时，即使使用了 NODYNAM 选项，仍然是动态调用。

与静态调用相对应，动态调用最基本的特征是主程序与子程序并不在同一加载模块中。动态调用中的子程序是在被调用时，才被读入内存的。关于程序的动态调用，有以下几点需要注意。

- 当多次进行动态调用时，每次调用的程序都会是其最近一次被调用后的状态。这点同静态调用是类似的。但是，在动态调用中，可以通过 CANCEL 语句从内存中移除掉被调用程序。此时被调用程序仍然是存在的，只是其被分配的内存存储空间被释放掉了。因此，当下一次再调用该程序时，该程序将为最初的状态。
- 动态调用所占的存储空间相对较小。这一点实际上通常是使用动态调用最主要的原因。例如，假设某一主程序根据执行情况，将调用 10~100 个子程序。按照静态调用方式，将会把 100 个子程序全部存放到内存中。此时，如果该程序实际上仅调用了 10 个子程序，则这种方式显然是对内存空间的一种浪费。因此，这里通常应该使用动态调用方式，调用什么程序，就加载什么程序，动态地进行调用。
- 动态调用执行效率相对较慢。这是因为每次进行调用时，都要读写内存，所以增加了调用执行的时间。

11.6.2 动态调用程序示例

下面通过具体的程序示例，以便更好地说明动态调用的特点及用法。假设在动态调用中，某一主程序代码如下。

```
    IDENTIFICATION   DIVISION.
    PROGRAM-ID     DYNAMIC-MAIN.
```

```
 AUTHER         XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   IDENTIF      PIC  X(10).
 77   TEST-NUM    PIC  99.
*
 PROCEDURE   DIVISION.
     PERFORM  INIT-TEST-NUM.
     MOVE   'DYNAMIC-SUB'  TO  IDENTIF.
     CALL   IDENTIF   USING   TEST-NUM.
     DISPLAY 'TEST-NUM AFTER THE FIRST CALL: '  TEST-NUM.
     PERFORM  INIT-TEST-NUM.
     CANCEL  IDENTIF.            /*此处将第一次调用后的子程序从内存中移除*/
     MOVE   'DYNAMIC-ENTRY' TO  IDENTIF.
     CALL    IDENTIF  USING  TEST-NUM.
     DISPLAY 'TEST-NUM AFTER THE SECOND CALL: '  TEST-NUM.
       STOP RUN.
 INIT-TEST-NUM.
     MOVE  10  TO  TEST-NUM.
```

令该主程序中调用的子程序 DYNAMIC-SUB 的代码如下。

```
 IDENTIFICATION   DIVISION.
 PROGRAM-ID     DYNAMIC-SUB.
 AUTHER         XXX.
*
 ENVIRONMENT   DIVISION.
*
 DATA  DIVISION.
 WORKING STORAGE SECTION.
 01   PLUS-NUM    PIC  99   VALUE  15.
 LINKAGE SECTION.
 77   MAIN-NUM    PIC  99.
*
 PROCEDURE   DIVISION   USING  MAIN-NUM.
     ADD   MAIN-NUM  TO  PLUS-NUM.
     MOVE  PLUS-NUM  TO  MAIN-NUM.
       GOBACK.
    ENTRY  'DYNAMIC-ENTRY'   USING  MAIN-NUM.
     ADD   MAIN-NUM  TO  PLUS-NUM.
     MOVE  PLUS-NUM  TO  MAIN-NUM.
       GOBACK.
```

以上程序执行后，将产生如下输出结果。

```
TEST-NUM AFTER THE FIRST CALL: 25
TEST-NUM AFTER THE SECOND CALL: 25
```

可以看到，以上这两段程序实际上和静态调用中的两段示例程序是类似的。不过，此处子程序中虽然没有手工进行初始化，但每次调用时仍为最初始的状态。原因是在主程序中，第一次调用该子程序后使用了 CANCEL 语句将其从内存中移除掉了。这样，在第二次调用时，该子程序将被重新读入内存，其状态和第一次调用时是一样的。

最后，对本节所讲的动态调用与其所对应的静态调用总结如下。

- 在 COBOL 程序中，静态调用通常使用 CALL literal 语句实现。CALL literal 语句实际上就是将所调用的程序名，或选择性入口地址名作为直接数进行调用。静态调用的子程序每次被调用时的状态，为其上一次调用之后的状态。
- 动态调用通常使用 CALL identifier 语句实现。CALL identifier 语句实际上就是将所调用的程序名，或选择性入口地址名 MOVE 到变量中调用。并且，凡是使用 CALL identifier 语句进行的调用都为动态调用。在动态调用中，可以通过 CANCEL identifier 语句，将调用后的子程序从内存中移除掉。这样，便可使得下一次调用时该子程序为最初始的状态。

11.7 嵌套子程序

嵌套子程序是指在一个程序代码段内嵌套有多个子程序。由于嵌套的子程序是写在主程序的代码内部的，因此有的书上也将其称为内部子程序。相对于内部子程序，前面所讲解的都称作外部子程序。本节将主要就嵌套子程序的结构以及调用权限这两方面的内容进行讲解。

11.7.1 嵌套子程序的结构

所谓嵌套，实际上也可以理解为一种包含关系。在 COBOL 中的嵌套子程序，存在两种包含关系。一种为直接包含，另一种为间接包含。直接包含对应于一层嵌套，间接包含对应于多层嵌套。下面是一段嵌套子程序的代码示例。

```
IDENTIFICATION  DIVISION.
PROGRAM-ID  PGM-A.
PROCEDURE DIVISION.
     DISPLAY  'THIS IS PROGRAM A'.
     CALL  'PGM-B'.
     CALL  'PGM-C'.
       STOP RUN.
IDENTIFICATION  DIVISION.
PROGRAM-ID  PGM-B.
PROCEDURE DIVISION.
     DISPLAY  'THIS IS PROGRAM B'.
     EXIT PROGRAM.
END PROGRAM B.
IDENTIFICATION  DIVISION.
PROGRAM-ID  PGM-C.
PROCEDURE DIVISION.
     DISPLAY  'THIS IS PROGRAM C'.
      CALL  'PGM-C1'.
      CALL  'PGM-C2'.
     EXIT  PROGRAM.
IDENTIFICATION  DIVISION.
PROGRAM-ID  PGM-C1.
PROCEDURE DIVISION.
     DISPLAY  'THIS IS PROGRAM C1'.
     EXIT PROGRAM.
END PROGRAM C1.
IDENTIFICATION  DIVISION.
PROGRAM-ID  PGM-C2.
```

```
PROCEDURE DIVISION.
     DISPLAY  'THIS IS PROGRAM C2'.
     EXIT PROGRAM.
END PROGRAM C2.
END PROGRAM C.
END PROGRAM A.
```

通过以上代码注意到，在嵌套子程序中通常是根据 END PROGRAM 语句表示程序的结束。实际上，对于以上嵌套子程序，分别有以下直接和间接包含关系。

- PGM-A 为其中的主程序。该程序直接包含的程序有 PGM-B 和 PGM-C。间接包含的程序有 PGM-C1 和 PGM-C2。
- PGM-C 直接包含的程序为 PGM-C1 和 PGM-C2。该程序没有间接包含的程序。
- PGM-B 以及 PGM-C1 和 PGM-C2 都既没有直接包含的程序，也没有间接包含的程序。

通过以上各程序间的包含（嵌套）关系，可以得出该嵌套子程序的内部结构。实际上，以上嵌套子程序中连同主程序在内共有 5 个程序。并且，里面嵌套层数最多为 2 层。

总之，了解嵌套子程序的结构，关键是要找准其中每一个程序的起始和结束标志。其中程序的起始标志是通过标志部字段 IDENTIFICATION DIVISION 表示的。而程序的结束标志在此处则是通过 END PROGRAM 语句表示。

11.7.2 嵌套子程序的调用权限

在嵌套子程序中，程序通常只能调用其直接包含的子程序。若要调用其他子程序，必须在该子程序的标志部定义的程序名后面加上 COMMON 选项。例如，下面为一段嵌套子程序的大体结构。为便于观察，只显示其程序名及结束标志。该结构如下。

```
PROGRAM-ID.  P.
  PROGRAM-ID.  P1  IS  COMMON.
  END PROGRAM  P1.
  PROGRAM-ID.  P2  IS  COMMON.
  END PROGRAM  P2.
  PROGRAM-ID.  P3.
    PROGRAM-ID.  P31 IS COMMON.
    END PROGRAM P31.
    PROGRAM-ID.  P32.
      PROGRAM-ID.  P321.
      END PROGRAM  P321.
    END PROGRAM  P32.
  END PROGRAM  P3.
END PROGRAM P.
```

以上各程序调用及被调用的权限分别如下。

- 程序 P 可以调用程序 P1、P2、P3，但不能被任何程序所调用。
- 程序 P1 可以调用程序 P2。并且，该程序能被程序 P、P2、P3、P31、P32、P321 所调用。
- 程序 P2 可以调用程序 P1。并且，该程序能被程序 P、P1、P3、P31、P32、P321 所调用。
- 程序 P3 可以调用程序 P1、P2、P31、P32。该程序只能被程序 P 所调用。
- 程序 P31 可以调用程序 P1、P2。并且，该程序能被程序 P3、P32、P321 所调用。
- 程序 P32 可以调用程序 P1、P2、P31、P321。该程序只能被程序 P3 所调用。
- 程序 P321 可以调用程序 P1、P2、P31。该程序只能被程序 P32 所调用。

实际上，使用 COMMON 标注的程序是可以被同一嵌套层的其他程序所调用的。例如，P3 可以调用 P2，因为二者都被嵌套在程序 P 之后的同一层，并且 P2 被标注为 COMMON。而 P2 则不可以反过来调用 P3，因为 P3 并不被 P2 包含在内，并且 P3 也没标注为 COMMON。

11.8 本章回顾

本章在学习完单一 COBOL 程序的基础上，介绍了如何在多个程序之间进行互相调用的概念和方法。COBOL 中程序的调用通常也称为子程序调用。本章首先介绍了子程序调用的基本作用和特点，之后分别从 3 个方面对子程序调用进行了讲解。

- 第一个方面是从调用关系的角度出发，将调用过程中所涉及到的程序分为两类。其中，一类为主调用程序，另一类为被调用程序。关于主调用程序和被调用程序，需要掌握各自程序中参数的定义、传递及引用，掌握使用 CALL 语句在主调用程序中进行程序调用，掌握使用 ENTRY 语句在被调用程序中编写选择性入口地址。
- 第二个方面根据调用方式，将其分为静态调用和动态调用两种。关于静态调用和动态调用，需要理解各自的联系及区别，掌握使用 CALL literal 语句和 CALL identifier 语句区别以上两种调用方式，掌握在动态调用中使用 CANCEL 语句，将调用后的程序从内存中移除。
- 第 3 个方面介绍了嵌套子程序的相关概念。关于嵌套子程序，需要掌握如何划分其内部结构，并明确其中各程序的调用权限。

第 12 章 COBOL 中的面向对象技术

通常而言，使用 COBOL 语言大多数进行的是面向过程的编程。不过在新的 COBOL 标准中，也可实现面向对象编程。面向对象技术是当前一种比较流行的编程技术，它本身就是一门独立的学科。

12.1 面向对象的基本概念

面向对象的概念是同面向过程的概念相对应的。传统的编程方式是面向过程的。面向过程的编程方式通常也叫做结构化编程。面向过程的技术是基于程序的执行步骤，根据程序的具体结构而来的。即面向过程的编程思想是从机器的角度出发来考虑问题的。最常见的基于面向过程的程序设计语言是 C 语言。

与之对应，面向对象的编程思想是从现实世界的角度出发来考虑问题的。当前流行的 Java 语言和 C++语言通常使用的就是面向对象的编程技术。在面向对象编程中通常会涉及到以下几个基本概念，下面分别进行介绍。

12.1.1 对象的概念

前面讲到，面向对象的编程思想是从现实世界的角度出发来考虑问题的。因此，在现实世界中存在的任何一个具体事物都可以作为面向对象技术中的对象。例如，以下就为 3 个现实对象。

- 一个人
- 一辆车
- 一只狗

需要注意的是，现实世界中的对象都有两个性质：对象的状态和对象的行为。例如，对于以上 3 个对象，其各自的状态可以如下。

- 这个人的姓名、年龄、当前心情的好坏等。
- 这辆车的型号、价格、新旧程度等。

- 这只狗的品种、颜色、是否饿了等。

而这 3 个对象各自的行为则可以如下。

- 工作、娱乐、学习 COBOL 等。
- 启动、刹车、加速等。
- 叫、抓东西、摇尾巴等。

以上列举的是现实对象，当将其应用到程序中，称之为软件对象。软件对象同样也是有状态和行为的。软件对象中的状态是通过其中的变量表示的，而软件对象中的行为则是通过其中的方法实现的。方法在面向对象技术中通常是指一个与对象相关联的函数。在 COBOL 中，方法需要单独定义。COBOL 中的方法相当于一系列处理过程。

因此，严格的说，对象是变量和相关方法的软件组合。其中特定的对象被称为实例。此外，将对象的变量置于它的方法的保护之下，这种技术也被称为封装。

12.1.2 类的概念

在现实世界中，从相同类型的对象中提取出一个抽象的概念，便形成了类。例如，对于前面提到的 3 个对象，实际上分别属于以下这 3 个类。

- 人
- 车
- 狗

前面所说的一个人这个对象也就是人这个类的一个实例。人是有一些共性的，在面向对象中称之为共同的状态和行为。这些共同的状态和行为便是在类中进行定义的。

因此，严格地说，类是蓝图或原型，它定义了所有某种类的对象的共有的变量和方法。类变量包含由此类的所有实例共享的信息，所有实例共享此变量。类方法可以直接从类进行调用，而实例方法则必须在一个特定的实例上调用。

12.1.3 继承的概念

关于继承，需要了解上级类和子类的概念。上级类也可称为基类或父类，而子类也可称为派生类。例如，若将前面所说的 3 个类作为上级类，下面分别可以作为与其相对应的子类。

- 男人、女人、老人、好人等。
- 公交车、小轿车、卡车、自行车等。
- 家犬、警犬、猎犬、牧羊犬等。

当子类对象只有一个上级类对象时，称为单继承。例如，对于一个男人这个对象，是从一个人这个上级类对象继承而来，是单继承。

当子类对象有两个或多个上级类对象时，称为多继承。例如，可以将上面的男人类和好人类同时作为上级类，生成一个好男人的子类。对于一个好男人这样一个对象，是分别从其上级类对象一个男人和一个好人继承而来的。这种情况便属于多继承。COBOL 语言是支持多继承的。

从状态和行为的角度来看，继承可以分为取代继承、包含继承、受限继承和异化继承。下面分别进行介绍。

- 取代继承：子类对象完整地继承了所有上级类的所有状态和行为。并且，子类没有修改

原有的状态和行为或者增加新的状态和行为。

- 包含继承：子类对象完整地继承了所有上级类的所有状态和行为。同时，子类还增加了新的状态和行为。
- 受限继承：子类对象部分地继承了上级类的状态和行为，并且没有增加新的状态和行为。
- 异化继承：子类对象继承了上级类的状态和行为，并且对原有的状态和行为进行了修改。

12.1.4 消息的概念

消息是用于软件对象的交互和通信的。例如，当一个对象希望另一个对象执行一个方法时，则该对象需要发送一个消息给另一个对象。例如，对于前面的例子，在以下的过程执行中将传递相应的消息。

- 一个人向另一个人打招呼。
- 一个人将一辆车启动。
- 一个人给一只狗喂食。

通常，消息是由以下 3 个部分组成的。

- 消息被发送到的对象。
- 要执行的方法的名称。
- 需要由此方法传递的任何参数。

12.1.5 多态的概念

多态可以分为编译时多态和运行时多态两种类别。其中编译时多态通常也称作重载。默认情况下，多态一般指的是运行时多态。

多态通常是根据消息而来的。例如，在课堂上老师给学生上课时，老师可以作为一个对象，而各位学生则为另一些对象。当老师讲述了一个知识点，可以认为是老师这个对象向不同的学生对象发送了一条消息。这时，有的学生将其记在了笔记上，有的学生仅仅是暂时记在了脑中，还有的学生可能在打瞌睡……这种现象便是所谓的多态性。

严格地说，多态性是指对于同一消息而言，不同类中的同名方法可能作用不同。对于上面例子而言，可以认为共有 3 种不同的学生类。对于老师这个对象在讲课时所发送的一条消息，这 3 个类都有一个同名的方法用以执行，即听课。其中一类学生听课的作用是记笔记，另一类学生听课的作用是记在脑中，还有一类学生听课的作用是打瞌睡。

12.1.6 接口的概念

接口是无关对象用来进行彼此交互的设备。每个对象都有接口，并且一个对象可以实现多个接口。在接口中的所有方法都是抽象方法。此外，每个类也都是有接口的。在 COBOL 中，类通常都是有两个接口的。一个接口称为工厂接口，另一个接口称为对象接口或者是实例接口。

例如，大型的宠物商店可以通过进销存系统来管理其中的一只狗。此时，进销存系统可以不关心它管理的项目的类，只需每个项目提供某些信息，如价格和条形码等。该系统并不在无关的项目上实施类关系，而是建立一个通信协议。该协议采用的形式便是接口中包含的一组常量和方法定义。

通常，接口主要有以下几点用处。

- 获得无关类之间的相似之处，而不必在无关项目上实施类关系。
- 声明一个或多个类可能要实现的方法。
- 显示对象的编程接口，而不必显示它的类。

12.2 定义 COBOL 中的类

使用面向对象技术最首要的任务是定义程序中的类。在 COBOL 中，定义类同编写完整的 COBOL 程序一样，通常也是需要依次编写 4 个部的代码。下面分别进行介绍。

12.2.1 标志部中的定义

标志部是在定义 COBOL 中的类时要求必须具有的一个部。在标志部中主要用以说明类的名称，以及该类的继承关系。当标志部用来定义类时，该部通常含有以下几个代码段。

- CLASS-ID 代码段。
- AUTHOR 代码段。
- INSTALLATION 代码段。
- DATE-WRITTEN 代码段。
- DATE-COMPILED 代码段。

对于以上这几个代码段，只有 CLASS-ID 代码段是必须具备的。该代码段用于指明类的名称及其继承关系。其余几个代码段是可选的。

例如，以下为一段标志部中类的定义代码。

```
IDENTIFICATION  DIVISION.
CLASS-ID.  HUMANCLS   INHERITS  BASECLS.
……
```

以上代码指定了类的名称和其继承关系。其中类的名称是通过 CLASS-ID 字段指定的，该类的名称为“HUMANCLS”。类的继承是通过 INHERITS 字段指定的。以上所定义的类是从名称为“BASECLS”的类继承而来的。

此外，由于 COBOL 是支持多继承的，因此在标志部中也可指定多个继承的类。指定多个继承类的方式如下。

```
IDENTIFICATION  DIVISION.
CLASS-ID.  HUMANCLS   INHERITS  BASECLS-1
                                BASECLS-2
                                ……
                                BASECLS-N.
……
```

最后需要注意的是，该类所继承的类通常是需要在后面环境部中的 REPOSITORY 代码段中进行指定的。而该类本身在 REPOSITORY 代码段中的指定通常是可选的。

12.2.2 环境部中的定义

环境部也是在定义 COBOL 中的类时必须要具备的一个部。环境部主要用于指明程序中

的类和程序外部环境中的类之间的对应关系。当环境部用来定义类时，该部通常含有配置节中的以下字段。

- REPOSITORY 字段
- SOURCE-COMPUTER 字段
- OBJECT-COMPUTER 字段
- SPECIAL-NAMES 字段

对于以上字段，只有 REPOSITORY 字段是必须具备的。其中，REPOSITORY 字段中用于指定程序中的类和程序外部环境中的类的对应关系。这种指定方式类似于在普通 COBOL 程序环境部文件节 FILE-CONTROL 字段中对于文件的指定。此外，在实际编写代码中，不要忘记注明配置节的标识符“CONFIGURATION SECTION”。

例如，下面为一段环境部中类的定义代码。此处的类的定义是紧接着以上标志部中的定义而来的。该段代码如下。

```
......
ENVIRONMENT  DIVISION.
CONFIGURATION  SECTION.
REPOSITORY.
   CLASS  BASECLS  IS  'BASECLS'
   CLASS  HUMANCLS   IS  'HUMANCLS'.
......
```

以上程序外部环境中的类是用引号包含起来的。程序外部环境中的类的名字既可以和程序中类的名字相同，也可以不同。该段代码中两者的名字都是相同的。此外，有的书上也将此处的“IS”写为“AS”，这两者实际上是等效的。

下面给出普通程序中环境部中的通常格式。通过对比，可以发现环境部在定义类和普通程序中的格式是有类似之处的。该段代码如下。

```
......
ENVIRONMENT  DIVISION.
INPUT-OUTPUT  SECTION.
FILE-CONTROL.
   SELECT  FILE-IN  ASSIGN  TO  S-SYSIN
   SELECT  FILE-OUT  ASSIGN  TO  S-SYSOUT.
......
```

通过对比，可以发现二者在格式上有以下几点类似之处。

- 定义类中的 CONFIGURATION SECTION 对应于普通程序中的 INPUT-OUTPUT SECTION。
- 定义类中的 REPOSITORY 对应于普通程序中的 FILE-CONTROL。
- 定义类中的 CLASS…IS 对应于普通程序中的 SELECT…ASSIGN TO。
- 定义类中的 BASECLS 和 HUMANCLS 分别对应于普通程序中的 FILE-IN 和 FILE-OUT。
- 定义类中的 ‘BASECLS’ 和 ‘HUMANCLS’ 分别对应于普通程序中的 S-SYSIN 和 S-SYSOUT。

12.2.3 数据部中的定义

数据部主要用于定义一些类中用到的变量。这些变量属于类变量，通过该类所产生的对

象将共享这些变量。根据前面所讲，对象中的这些变量是用来指定该对象的软件状态的。

数据部中只有 WORKING-STORAGE SECTION（工作存储节）这一个节。该节是可选的。并且，在类的定义中，数据部本身也是可选的。

例如，接着前面 HUMANCLS 类的定义，该类在数据部中的代码可以如下。

```
......
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01   NAME    PIC  X(10).
01   AGE      PIC  999.
01   MOOD    PIC  X(5).
......
```

以上定义的 3 个类变量在当该类的实例被创建时作为该实例的数据，被静态的分配存储空间。有多少个实例被创建，也就相应的分配多少个存储空间。并且，这些数据直到该实例被删除时才释放其存储空间。

默认情况下，这些实例中的数据对于该实例中的方法而言是全局数据。即该实例通过在其类中所定义的任何方法都可以访问这些数据。但是，实例中的数据同时又具有私有属性。即任何其他的类，包括该类的子类都无法直接对这些数据进行访问。

工作存储节在定义类中的格式要求和在普通程序中的大体类似。如 01 层、05 层数据的写法，PIC 语句的使用等。不过，该节在定义类时也有一些特别的要求，分别如下。

- 不可使用 VALUE 语句对类中的变量进行初始化。不过，在 88 层数据则是可以使用 VALUE 语句的。
- 不可使用外部属性。
- 可以使用全局属性，不过通常没什么实际效果。

12.2.4 类的完整定义

前面依次讲到了类在标志部、环境部、数据部中相应的定义步骤。过程部主要用于定义类中的方法。如前面所讲，这些方法构成了类的软件行为。过程部在类的定义中也是可选的。

关于过程部中方法的定义，情况比较复杂，将在后面单独的一节中进行详细讲解。因此不妨结合前面 3 个小节所讲解的内容，暂不考虑过程部中的定义，实现类的完整定义框架结构。关于前面例子中 HUMANCLS 类的完整定义结构如下。

```
 IDENTIFICATION  DIVISION.
 CLASS-ID.  HUMANCLS   INHERITS  BASECLS.
*
 ENVIRONMENT  DIVISION.
 CONFIGURATION  SECTION.
 REPOSITORY.
    CLASS  BASECLS  IS  'BASECLS'
    CLASS  HUMANCLS   IS  'HUMANCLS'.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01   NAME    PIC  X(10).
 01   AGE      PIC  999.
 01   MOOD    PIC  X(5).
```

```
*
 PROCEDURE  DIVISION.
     ......
 END CLASS  HUMANCLS.
```

通过以上代码可以看到，当定义完一个类时，需要用 END CLASS 加上类名表示定义结束。此外，由于在定义类中数据部和过程部并不是必须的，因此下面的定义也是正确的。

```
 IDENTIFICATION  DIVISION.
 CLASS-ID.  HUMANCLS   INHERITS  BASECLS.
*
 ENVIRONMENT  DIVISION.
 CONFIGURATION  SECTION.
 REPOSITORY.
    CLASS  BASECLS  IS  'BASECLS'
    CLASS  HUMANCLS   IS  'HUMANCLS'.
    ......
 END CLASS  HUMANCLS.
```

通过后面的学习可以知道，类的完整定义结构是类似于前面章节所讲的嵌套子程序的。类本身为最外层的嵌套，对象和方法等嵌套于其中。此处的省略号正是表示省略了类中所嵌套的其他内容。

12.3 COBOL 中的方法

COBOL 中的方法构成了 COBOL 中对象的软件行为。同类一样，在 COBOL 中，方法也是需要单独定义的。方法的定义通常是在类或者对象的过程部中进行的。COBOL 中的方法可以看作是一系列处理过程。

12.3.1 方法的定义

同定义类一样，定义方法也是分别由标志部、环境部、数据部和过程部中的定义所组成的。不过定义方法中只有标志部是必须具备的，其他 3 个部都是可选的。此外，当方法定义完成后，是使用 END METHOD 语句表明结束的。

在标志部中，是通过 METHOD-ID 来指定方法的名称的。其他方法或程序通过此处所指定的名称来调用该方法。

在环境部中，此处仅有 INPUT-OUT SECTION（输入输出节）。该节的用法和普通程序中的类似。这点和类的定义是不同的。

定义方法的数据部相对要复杂一些。当数据部用于定义方法时，该部的内容通常可以由以下这几个节组成。

- FILE SECTION（文件节）：用于指定该方法中所需用到的文件。指定方式同在普通程序中的类似，不过此处仅能用于指定外部文件。
- LOCAL-STORAGE SECTION（本地存储节）：用于定义该方法所用到的本地临时数据。当方法执行完毕返回时，这些数据的存储空间也相应地被释放。
- WORKING-STORAGE SECTION（工作存储节）：用于定义该方法所用到的静态数据。这些数据在每一次调用该方法时，都为最近一次调用后的状态。

- LINGKAGE SECTION（连接节）：用于定义该方法需要用到的一些参数。使用方式和普通程序中的类似。

根据前面的例子，以下 3 个方法分别表示人的工作、娱乐和学习 COBOL 这 3 个行为。其中表示人工作行为的方法定义如下。

```
 IDENTIFICATION  DIVISION.
 METHOD-ID.  WORKING.
*
 PROCEDURE DIVISION.
       DISPLAY  'I AM WORKING'.
 END METHOD  WORKING.
```

表示娱乐行为的方法定义如下。

```
 IDENTIFICATION  DIVISION.
 METHOD-ID.  PLAYING.
*
 PROCEDURE DIVISION.
      DISPLAY  'I AM PLAYING'.
 END METHOD  PLAYING.
```

表示学习 COBOL 的方法定义如下。

```
 IDENTIFICATION  DIVISION.
 METHOD-ID.  LEARNING.
*
 PROCEDURE DIVISION.
     DISPLAY  'I AM LEARNING COBOL'.
 END METHOD  LEARNING.
```

以上只是分别使用了 DISPLAY 语句用以简单地象征各方法的执行过程。COBOL 中的方法实际上可以包含几乎任何 COBOL 中的常用语句。不过，对于以下这两条语句，在方法的定义中是不可以出现的。

- ENTRY 语句
- EXIT PROGRAM 语句

此外还有一些语句，如 ALTER 语句、SEGMENTATION 语句等。由于这些语句在 ANSI COBOL-85 中已经废除掉了，因此通常不必予以考虑。

12.3.2 嵌套在类与对象中的方法

方法既可以嵌套在类中，也可以嵌套在对象中。嵌套在类中的方法称为类的方法。嵌套在对象中的方法成为实例方法。

1. 嵌套在类中的方法

以下将上一小节中定义的 3 个方法嵌套在 MENCLS 类中所形成的完整的类结构。该结构的代码如下。

```
 IDENTIFICATION  DIVISION.
 CLASS-ID.  HUMANCLS   INHERITS  BASECLS.
*
 ENVIRONMENT  DIVISION.
```

```
 CONFIGURATION  SECTION.
 REPOSITORY.
    CLASS  BASECLS  IS  'BASECLS'
     CLASS  HUMANCLS   IS  'HUMANCLS'.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01   NAME    PIC  X(10).
 01   AGE     PIC  999.
 01   MOOD    PIC  X(5).
*
 PROCEDURE  DIVISION.
*
 IDENTIFICATION  DIVISION.
 METHOD-ID.  WORKING.
 PROCEDURE DIVISION.
      DISPLAY  'I AM WORKING'.
 END METHOD  WORKING.
*
 IDENTIFICATION  DIVISION.
 METHOD-ID.  PLAYING.
 PROCEDURE DIVISION.
      DISPLAY  'I AM PLAYING'.
 END METHOD  PLAYING.
*
 IDENTIFICATION  DIVISION.
 METHOD-ID.  LEARNING.
 PROCEDURE DIVISION.
      DISPLAY  'I AM LEARNING COBOL'.
 END METHOD  LEARNING.
 END CLASS  HUMANCLS.
```

可以看到，以上依次将 3 个方法嵌套在 MENCLS 类的过程部中进行了定义。因此形成了一个完整的、具有一定功能的类。

2．嵌套在对象中的方法

关于嵌套在对象中的方法，首先要明确对象的定义。对象通常也是嵌套在类的结构中进行定义的。例如，以下代码在 MENCLS 中定义了一个对象。并且，此处还将 MENCLS 中原有的类变量定义为了该对象的实例变量。代码如下。

```
 IDENTIFICATION  DIVISION.
 CLASS-ID.  HUMANCLS   INHERITS  BASECLS.
*
 ENVIRONMENT  DIVISION.
 CONFIGURATION  SECTION.
 REPOSITORY.
    CLASS  BASECLS  IS  'BASECLS'
    CLASS  HUMANCLS   IS  'HUMANCLS'.
*
 IDENTIFICATION  DIVISION.
 OBJECT.
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
```

```
01   NAME    PIC  X(10).
01   AGE      PIC  999.
01   MOOD    PIC  X(5).
PROCEDURE  DIVISION.
     ......
END OBJECT.
END CLASS  MENCLS.
```

以上便实现了类中对象的定义。嵌套在对象中的方法是直接在对象的过程部中实现的定义。此处相当于是一个二层嵌套。将以上 3 个方法在对象中定义为实例方法的完整代码如下。

```
IDENTIFICATION  DIVISION.
CLASS-ID.  HUMANCLS   INHERITS  BASECLS.
*
ENVIRONMENT  DIVISION.
CONFIGURATION  SECTION.
REPOSITORY.
   CLASS  BASECLS  IS  'BASECLS'
   CLASS  HUMANCLS   IS  'HUMANCLS'.
*
IDENTIFICATION  DIVISION.
OBJECT.
DATA  DIVISION.
WORKING-STORAGE  SECTION.
01   NAME    PIC  X(10).
01   AGE      PIC  999.
01   MOOD    PIC  X(5).
PROCEDURE  DIVISION.
*
IDENTIFICATION  DIVISION.
METHOD-ID.  WORKING.
PROCEDURE DIVISION.
    DISPLAY  'I AM WORKING'.
END METHOD  WORKING.
*
IDENTIFICATION  DIVISION.
METHOD-ID.  PLAYING.
PROCEDURE DIVISION.
    DISPLAY  'I AM PLAYING'.
END METHOD  PLAYING.
*
IDENTIFICATION  DIVISION.
METHOD-ID.  LEARNING.
PROCEDURE DIVISION.
     DISPLAY  'I AM LEARNING COBOL'.
END METHOD  LEARNING.
END OBJECT.
END CLASS  MENCLS.
```

以上代码表示在类 HUMANCLS 中的对象里有 3 个软件状态及 3 个软件行为。其中，3 个软件状态分别使用以下 3 个实例变量来表示。

- NAME
- AGE
- MOOD

该对象中的 3 个软件行为分别是通过其中的 3 个实例方法来表示的。这 3 个实例方法的名称依次如下。

- WORKING
- PLAYING
- LEARNING

12.4 COBOL 中的客户程序

用来对方法进行调用的程序称为客户程序。类只是对变量和方法进行了定义，是一个静态的概念。而客户程序是执行类中定义的方法，是一个动态的概念。

12.4.1 客户程序的定义

客户程序也是依次由标志部、环境部、数据部和过程部所组成的。其中标志部和环境部是必须具备的，而数据部和过程部则是可选的。下面依次对其进行介绍。

1. 标志部中的定义

标志部主要用来定义客户程序的程序名。客户程序在标志部中的定义方式同普通程序的类似。下面为一段客户程序在标志部中的定义代码。

```
IDENTIFICATION  DIVISION.
PROGRAM-ID.  CLIENT.
……
```

2. 环境部中的定义

环境部主要用来指明客户程序中所用到的类和外部环境类之间的关系。在环境部中必须具有以下这两个字段。

- CONFIGURATION　SECTION
- REPOSITORY

接着上面在标志部中定义的客户程序，该程序在环境部中的定义可以如下。

```
……
ENVIRONMENT  DIVISION.
CONFIGURATION  SECTION.
REPOSITORY.
   CLASS  HUMANCLS   IS  'HUMANCLS'.
……
```

3. 数据部中的定义

数据部用来定义客户程序中需要用到的一些数据。这些数据通常用来指向一个类。在数据部中，仅有一个 WORKING-STORAGE SECTION（工作存储节），并且该节是可选的。接着上面在环境部中定义的客户程序，该程序在数据部中的定义如下。

```
……
DATA  DIVISION.
```

```
WORKING-STORAGE  SECTION.
01  M   USAGE  OBJECT  REFERENCE  HUMANCLS.
......
```

以上代码中，M 为该客户程序中所定义的数据。该数据通过“USAGE OBJECT REFERENCE”指向 HUMANCLS 类。

4．过程部中的定义

过程部主要用来创建或释放类的实例，操作实例中的变量，调用实例中的方法。过程部通常使用 STOP RUN 或 EXIT PROGRAM 表示客户程序的结束。此外，在程序结束标志后，还应使用 END PROGRAM 表示整个程序代码的结束。接着上面在数据部中定义的客户程序，该程序在过程部中的定义可以如下。

```
......
PROCEDURE  DIVISION.
   INVOKE  HUMANCLS  'NEW'  RETURNING  M.
   INVOKE  M  'LEARNING'.
   EXIT  PROGRAM.
END PROGRAM  CLIENT.
```

通过以上代码可以看出，在过程部中实际上是通过 INVOKE 语句实现实例的创建和方法的调用的。该段代码创建了一个 HUMANCLS 的实例，并且将该实例通过数据 M 引用。此后，该段代码再通过数据 M 调用了该实例的一个方法“LEARNING”。最后，该代码通过 EXIT PROGRAM 结束程序，并通过 END PROGRAM CLIENT 表示 CLIENT 程序代码段的结束。

12.4.2　通过客户程序调用方法

客户程序主要是为了调用实例中的方法，从而实现实例的软件行为。以前面定义的 HUMANCLS 类为例，下面这段客户程序分别实现了其中的 3 个方法。代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  CLIENT.
*
 ENVIRONMENT  DIVISION.
 CONFIGURATION  SECTION.
 REPOSITORY.
   CLASS  HUMANCLS   IS  'HUMANCLS'.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  M   USAGE  OBJECT  REFERENCE  HUMANCLS.
*
 PROCEDURE  DIVISION.
   INVOKE  HUMANCLS  'NEW'  RETURNING  M.
   INVOKE  M  'WORKING'.
   INVOKE  M  'PLAYING'.
   INVOKE  M  'LEARNING'.
   EXIT  PROGRAM.
 END PROGRAM  CLIENT.
```

以上客户程序执行后，将有如下输出信息。

```
I AM WORKING
I AM PLAYING
I AM LEARNING COBOL
```

该段客户程序首先创建了一个基于 HUMANCLS 类的实例。并且，该程序使用数据部中定义的数据 M 来引用该实例。其后，该程序通过数据 M 实现了 HUMANCLS 类的实例中 3 个方法的调用。关于 HUMANCLS 类的原型，同前面章节中的一致。

12.4.3 包含实例变量的方法调用

上一小节仅是使用客户程序实现了最基本的方法调用。在以上所调用的方法中，只包含了一条输出语句，并未涉及到实例中的变量。本节将在此基础上，讨论如何通过客户程序实现包含有实例变量的方法调用。

例如，在 HUMANCLS 类中有 3 个实例变量，可分别表示姓名、年龄和心情状态。此处要求通过客户程序，分别调用包含有以上 3 个实例变量的方法。其中所要实现的功能是能够对实例状态进行设置，并得到该状态的输出信息。

为实现以上功能，首先需要重写 HUMANCLS 类。不妨将重写后的 HUMANCLS 类命名为 HUMANCLS2。重写后的 HUMANCLS2 类需要包含 6 个方法。设置姓名、年龄和心情状态为其中的 3 个方法。输出姓名、年龄和心情状态信息为另外的 3 种方法。HUMANCLS2 类的代码如下。

```
 IDENTIFICATION  DIVISION.
 CLASS-ID.  HUMANCLS2   INHERITS  BASECLS.
*
 ENVIRONMENT  DIVISION.
 CONFIGURATION  SECTION.
 REPOSITORY.
    CLASS  BASECLS  IS  'BASECLS'
    CLASS  HUMANCLS   IS  'HUMANCLS'.
*
 IDENTIFICATION  DIVISION.
 OBJECT.
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01   NAME    PIC  X(10).
 01   AGE     PIC  999.
 01   MOOD    PIC  X(5).
 PROCEDURE  DIVISION.
*
 IDENTIFICATION  DIVISION.
 METHOD-ID.  SETNAME.
 DATA  DIVISION.
 LINKAGE  SECTION.
 01   PASS-NAME   PIC  X(10).
 PROCEDURE DIVISION  USING  PASS-NAME.
      MOVE  PASS-NAME  TO  NAME.
 END METHOD  SETNAME.
*
 IDENTIFICATION  DIVISION.
 METHOD-ID.  SETAGE.
 DATA  DIVISION.
```

```
 LINKAGE  SECTION.
 01   PASS-AGE   PIC  999.
 PROCEDURE DIVISION  USING  PASS-AGE.
      MOVE  PASS-AGE  TO  AGE.
 END METHOD  SETAGE.
*
 IDENTIFICATION  DIVISION.
 METHOD-ID.  SETMOOD.
 DATA  DIVISION.
 LINKAGE  SECTION.
 01   PASS-MOOD   PIC  X(5).
 PROCEDURE DIVISION  USING  PASS-MOOD.
      MOVE  PASS-MOOD  TO  MOOD.
 END METHOD  SETMOOD.
*
 IDENTIFICATION  DIVISION.
 METHOD-ID.  TELLNAME.
 PROCEDURE DIVISION.
       DISPLAY  'NAME IS: '  NAME.
 END METHOD  TELLNAME.
*
 IDENTIFICATION  DIVISION.
 METHOD-ID.  TELLAGE.
 PROCEDURE DIVISION.
      DISPLAY  'AGE IS: '  AGE.
 END METHOD  TELLAGE.
*
 IDENTIFICATION  DIVISION.
 METHOD-ID.  TELLMOOD.
 PROCEDURE DIVISION.
      DISPLAY  'MOOD IS: '  MOOD.
 END METHOD  TELLMOOD.
*
 END OBJECT.
 END CLASS  MENCLS.
```

基于以上重写的 HUMANCLS 类，实现指定功能的客户程序代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  CLIENT.
*
 ENVIRONMENT  DIVISION.
 CONFIGURATION  SECTION.
 REPOSITORY.
 CLASS  HUMANCLS   IS  'HUMANCLS'.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  M   USAGE  OBJECT  REFERENCE  HUMANCLS.
*
 PROCEDURE  DIVISION.
     INVOKE  HUMANCLS  'NEW'  RETURNING  M.
     INVOKE  M  'SETNAME'  USING  BY  CONTENT  'ROBIN'.
     INVOKE  M  'TELLNAME'.
     INVOKE  M  'SETAGE'  USING  BY  CONTENT  23.
     INVOKE  M  'TELLAGE'.
```

```
        INVOKE  M  'SETMOOD'  USING  BY  CONTENT  'GOOD'.
        INVOKE  M  'TELLMOOD'.
        EXIT  PROGRAM.
    END PROGRAM  CLIENT.
```

该段客户程序执行后，将有以下输出信息。

```
NAME IS: ROBIN
AGE IS: 23
MOOD IS: GOOD
```

最后需要注意的是，以上客户程序在调用设置实例变量的方法时，用到了参数传递。并通过所传递的参数对实例变量进行设置。在调用方法时实现参数传递的语法格式如下。

```
INVOKE  data  'method'  USING  BY  CONTENT  'parameter'.
```

12.5 COBOL 中的子类

实际上，此前使用的 HUMANCLS 类以及 HUMANCLS2 类都属于 BASECLS 的子类。不过，前面仅从类的角度出发，并未体现子类的特征及用途。本节将重点从子类的角度出发，讨论面向对象中基于子类的继承技术。

12.5.1 子类的定义

子类的定义方式和前面章节中所讲到的类的定义方式大体相同。例如，以下为一个类名为 TESTCLS 的子类定义代码。

```
    IDENTIFICATION  DIVISION.
    CLASS-ID.  TESTCLS   INHERITS  HUMANCLS  HUMANCLS2.
   *
    ENVIRONMENT  DIVISION.
    CONFIGURATION  SECTION.
    REPOSITORY.
      CLASS  HUMANCLS   IS  'HUMANCLS'.
      CLASS  HUMANCLS2   IS  'HUMANCLS2'.
   *
    DATA  DIVISION.
    WORKING-STORAGE  SECTION.
    01   TEST-NAME    PIC  X(10).
    01   TEST-AGE     PIC  999.
    01   TEST-MOOD    PIC  X(5).
   *
    PROCEDURE  DIVISION.
   *
    IDENTIFICATION  DIVISION.
    METHOD-ID.  TESTMETHOD.
    PROCEDURE DIVISION.
         DISPLAY  'THIS IS A SUBCLASS FOR TEST'.
         INVOKE  SELF  'LEARNING'.
    END METHOD  TESTMETHOD.
   *
    END CLASS  TESTCLS.
```

对于以上定义的子类，有以下两点需要特别注意。

- 该子类有两个上级类，分别为 HUMANCLS 和 HUMANCLS2。该子类实现了多继承，对应的代码如下。

```
CLASS-ID.  TESTCLS   INHERITS  HUMANCLS  HUMANCLS2.
```

- 该子类中定义了一个方法，方法名为 TESTCLS。在该方法中，调用了其上级类的方法 LEARNING，对应的代码如下。

```
INVOKE  SELF  'LEARNING'.
```

12.5.2 子类的应用

在面向对象技术中，使用子类主要体现了继承的概念和作用。具体而言，在程序中应用子类通常有以下几个目的。

- 增强代码可重用性。这点主要体现在通过子类可以直接调用其上级类中已定义的方法，而不必再次进行定义。
- 使类更加具体、精确。例如，男人这个类作为人这个类的子类，就比人这个类所指的范围更加精确。在软件实现上主要是通过在子类中增加新的变量或方法，从而生成更加具体的实例。这一点实际上对应于前面所讲的包含继承。
- 可以重载从上级类中继承的方法，从而实现类的多样性，并可使由类生成的对象具有多态性。这一点实际上对应于前面所讲的异化继承。

下面结合具体的例子，以便更好地说明子类在实际程序中的应用。首先需要定义一个子类。设该子类的类名为 MANCLS，其上级类为 HUMANCLS。该类中包含一个对象的定义，在对象中可通过 VALUE 语句实现一些变量的初始化。定义代码如下。

```
 IDENTIFICATION  DIVISION.
 CLASS-ID.  MANCLS   INHERITS  HUMANCLS.
*
 ENVIRONMENT  DIVISION.
 CONFIGURATION  SECTION.
 REPOSITORY.
    CLASS  HUMANCLS   IS  'HUMANCLS'.
*
 IDENTIFICATION  DIVISION.
 OBJECT.
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01   WORK-CODE     PIC  9.
      88  COBOL      VALUE 1.
      88  JAVA       VALUE 2.
      88  DELPHI     VALUE 3.
      88  VALID      VALUES  4
                     THRU  8.
 01   WORK-VALUES
      05  FILLER    PIC X(30)   VALUE  'PROGRAMING WITH COBOL'.
      05            PIC X(30)   VALUE  'PROGRAMING WITH JAVA'.
      05            PIC X(30)   VALUE  'PROGRAMING WITH DELPHI'.
      05            PIC X(30)   VALUE  'OTHER WORK'.
      05            PIC X(30)   VALUE  'INVALID WORK CODE'.
 01   WORK-TABLE  REDEFINES  WORK-VALUES.
      05   WORKS  PIC X(30)   OCCURS  5  TIMES.
```

```
 01   WORK-SUB    PIC 9.
*
 PROCEDURE  DIVISION.
*
 IDENTIFICATION  DIVISION.
 METHOD-ID.  WORKMETHOD.
 DATA  DIVISION.
 LINKAGE  SECTION.
 01   IN-CODE   PIC  9.
 PROCEDURE DIVISION  USING  IN-CODE.
     EVALUATE  TRUE
       WHEN   COBOL
          MOVE  1  TO  WORK-SUB
       WHEN   JAVA
          MOVE  2  TO  WORK-SUB
       WHEN   DELPHI
          MOVE  3  TO  WORK-SUB
       WHEN  VALID
          MOVE  4  TO  WORK-SUB
       WHEN   OTHER
          MOVE  5  TO  WORK-SUB
      END-EVALUATE.
      DISPLAY  WORKS  (WORK-SUB).
 END METHOD  WORKMETHOD.
*
 END OBJECT.
 END CLASS  TESTCLS.
```

以上完成了 HUMANCLS 类的子类 MANCLS 的定义。下面通过一段客户程序对该子类进行操作，客户程序的代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  CLIENT.
*
 ENVIRONMENT  DIVISION.
 CONFIGURATION  SECTION.
 REPOSITORY.
    CLASS  MANCLS   IS  'MANCLS'.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  M   USAGE  OBJECT  REFERENCE  MANCLS.
*
 PROCEDURE  DIVISION.
     INVOKE  MANCLS  'NEW'  RETURNING  M.
     INVOKE  M  'WORKING'.
     INVOKE  M  'WORKMETHOD'  USING  1.'.
     EXIT  PROGRAM.
 END PROGRAM  CLIENT.
```

以上客户程序执行后，将有如下输出信息。

```
I AM WORKING
PROGRAMING WITH COBOL
```

对于以上代码需要注意的是，在客户程序中是可以直接调用实例在上级类中定义的方法的。例如，该段代码里 WORKING 为 M 所引用的对象在其上级类 HUMANCLS 中定义的方

法。虽然该方法在对象所在的 MANCLS 子类中没有定义，但仍然是可以调用的。这一点体现了面向对象技术中继承的概念。

12.6 COBOL 中的工厂对象

工厂对象是包含在类中的一个特殊的对象。每个类只能有一个工厂对象。工厂对象主要包括与创建类的对象及其相关任务的数据和方法。在工厂对象中定义的变量和方法可以被该类的所有对象中的方法访问到。

12.6.1 工厂对象的定义

工厂对象是嵌套在类中进行定义的。与对象不同，工厂对象在定义时使用的标志符是 FACTORY，而不是 OBJECT。例如，以下为一段工厂对象的定义代码。

```
 IDENTIFICATION  DIVISION.
 CLASS-ID.  FATRYCLS   INHERITS  TESTCLS.
*
 ENVIRONMENT  DIVISION.
 CONFIGURATION  SECTION.
 REPOSITORY.
    CLASS  TESTCLS   IS  'TESTCLS'.
*
 IDENTIFICATION  DIVISION.
 FACTORY.
 DATA  DIVISION.
 WORKING-STORAGE SECTION.
 01  FATRY-DATA   PIC  X(10).
 PROCEDURE  DIVISION.
*
 IDENTIFICATION  DIVISION.
 METHOD-ID.  FATRYMETHOD.
 DATA  DIVISION.
 LINKAGE  SECTION.
 01  IN-OBJ   USAGE OBJECT REFERENCE SELF.
 PROCEDURE DIVISION  RETURNING  IN-OBJ.
     INVOKE SELF  'NEW'  RETURNING  IN-OBJ.
     INVOKE  IN-OBJ  'some method'.
 END METHOD FATRYMETHOD.
 END FACTORY.
*
IDENTIFICATION  DIVISION.
 OBJECT.
 ......
 END OBJECT.
END CLASS  FATRYCLS.
```

以上代码表示了在类中是如何定义工厂对象的。该工厂对象中定义了一个工厂变量和一个工厂方法，分别为 FATRY-DATA 和 FATRYMETHOD。在 FATRYMETHOD 中，需要注意以下这段代码。

```
 INVOKE SELF  'NEW'  RETURNING  IN-OBJ.
 INVOKE  IN-OBJ  'some method'.
```

该段代码表示通过“NEW”构造出了 FATRYCLS 类的一个对象，并且使用数据 IN-OBJ 对其引用。此后，直接通过数据 IN-OBJ 调用该对象中定义的一些方法。这一过程在工厂对象的方法中是常用的。

12.6.2 工厂对象的应用

下面结合一个具体的例子，以便更好地理解工厂对象在实际中的应用。首先，定义一个名为 HUMANCLS3 的类，该类包含有一个工厂对象。其中该工厂对象中没有定义工厂变量，但定义了一个工厂方法。完整的定义代码如下。

```
 IDENTIFICATION  DIVISION.
 CLASS-ID.  HUMANCLS3   INHERITS  BASECLS.
*
 ENVIRONMENT  DIVISION.
 CONFIGURATION  SECTION.
 REPOSITORY.
    CLASS  BASECLS   IS  'BASECLS'.
*
 IDENTIFICATION  DIVISION.
 FACTORY.
 PROCEDURE  DIVISION.
*
 IDENTIFICATION  DIVISION.
 METHOD-ID.  CREATENM.
 DATA  DIVISION.
 LINKAGE  SECTION.
 01  IN-NAME  PIC  X(10).
 01  IN-OBJ   USAGE OBJECT REFERENCE SELF.
 PROCEDURE DIVISION  USING IN-NAME  RETURNING  IN-OBJ.
     INVOKE SELF  'NEW'  RETURNING  IN-OBJ.
     INVOKE  IN-OBJ  'SETNAME'  USING  IN-NAME.
 END METHOD  CREATENM.
 END FACTORY.
*
 IDENTIFICATION  DIVISION.
 OBJECT.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  NAME  PIC X(10).
 PROCEDURE  DIVISION.
*
 IDENTIFICATION  DIVISION.
 METHOD-ID.  SETNAME.
 DATA  DIVISION.
 LINKAGE  SECTION.
 01  PASS-NAME  PIC  X(10).
 PROCEDURE  DIVISION  USING  PASS-NAME.
     MOVE  PASS-NAME TO NAME.
 END METHOD  SETNAME.
*
 IDENTIFICATION  DIVISION.
 METHOD-ID.  TELLNAME.
 PROCEDURE  DIVISION.
```

```
        DISPLAY  'NAME IS: '  NAME.
    END METHOD  TELLNAME.
   *
    END OBJECT.
    END CLASS  HUMANCLS3.
```

以上完成了包含有工厂对象的类的定义。根据该类定义的情况，下面为一段相应的客户程序，以实现对其中方法的调用。该程序代码如下。

```
    IDENTIFICATION  DIVISION.
    PROGRAM-ID.  CLIENT.
   *
    ENVIRONMENT  DIVISION.
    CONFIGURATION  SECTION.
    REPOSITORY.
      CLASS  HUMANCLS3   IS  'HUMANCLS3'.
   *
    DATA  DIVISION.
    WORKING-STORAGE  SECTION.
    01  A   USAGE  OBJECT  REFERENCE  HUMANCLS3.
    01  B   USAGE  OBJECT  REFERENCE  HUMANCLS3.
    01  C   USAGE  OBJECT  REFERENCE  HUMANCLS3.
   *
    PROCEDURE  DIVISION.
        INVOKE  HUMANCLS3  'CREATENM'
            USING  BY  CONTENT  'ALEX'  RETURNING  A.
        INVOKE  HUMANCLS3  'CREATENM'
            USING  BY  CONTENT  'PETER'  RETURNING  B.
        INVOKE  HUMANCLS3  'CREATENM'
            USING  BY  CONTENT  'TOM'  RETURNING  C.
        INVOKE  A  'TELLNAME'
        INVOKE  B  'TELLNAME'
        INVOKE  C  'TELLNAME'
        EXIT  PROGRAM.
   END PROGRAM  CLIENT.
```

以上客户程序执行后，将有以下输出信息。

```
NAME IS: ALEX
NAME IS: PETER
NAME IS: TOM
```

以上代码所实现的功能顺序分别如下。

- 依次调用 HUMANCLS3 类中工厂对象的方法 “CREATENM” 生成 3 个该类的对象。这 3 个对象分别通过数据 A、B、C 引用。
- 在生成对象的同时，通过参数传递依次设置这些对象中的变量。其中设置变量的方法是在工厂对象的方法中进行调用的。
- 依次调用 3 个对象中的方法，显示每个对象中被设置后的变量。

12.7 异常处理

所谓异常，是指在程序执行期间中断指令的正常流程的事件。异常处理可以代替程序的错误处理过程，从而使程序的处理流程更加简明和清晰。

任何基于面向对象的程序设计语言通常都有其特定的异常处理方式。在 COBOL 中，异常指令是通过 USE 语句表示的。该语句的格式如下。

```
USE  AFTER  EXCEPTION  exception-name.
```

在以上语句格式中，exception-name 表示相关异常的名字。在面向对象的 COBOL 中，通常有以下几种异常的名字。

- EC-OO
- EC-OO-CONFORMANCE
- EC-OO-EXCEPTION
- EC-OO-METHOD
- EC-OO-NULL
- EC-OO-RESOURCE
- EC-OO-UNIVERSAL
- EC-OO-IMP

这些异常的名字实际上是具有一定的意义的。例如，EC-OO-METHOD 表示和方法相关的异常，而 EC-OO-RESOURCE 表示和资源相关的异常，等等。

以上这些名字只是表示具体的异常种类。当对异常进行处理时，还需要使用相关的异常处理函数实现处理过程。以下为一段常见的异常处理语句代码。

```
……
USE  AFTER  EXCEPTION  EC-OO- UNIVERSAL.
……
DISPLAY  FUNCTION  function-name  FUNCTION function-name ……
……
```

以上代码段中的 function-name 表示相关的异常处理函数名。在 COBOL 中，通常有以下这几种异常处理函数名。

- EXCEPTION-STATUS
- EXCEPTION-STATEMENT
- EXCEPTION-LOCATION
- EXCEPTION-OBJECT

在系统内部，异常处理过程是由以下几步完成的。

（1）当程序在运行中产生一个异常时，会创建一个描述异常的对象。该对象称作异常对象。创建异常对象并将其交给系统称作抛出异常。

（2）当抛出异常后，系统会首先在抛出异常的函数内部寻找异常处理过程。如果没有找到，继续在调用该函数的函数内寻找。

（3）在系统中通常会存在一些包含可以处理异常的代码块，这些代码块被称作异常处理器。如果某一异常处理器可以处理的类型和抛出的异常对象的类型相同，则该处理器是合适的。选择合适的异常处理器并将异常传递给它，称作捕获异常。系统最终要么捕获异常，要么直接中断。

12.8 本章回顾

本章主要讲解了在 COBOL 语言中是如何实现面向对象的编程技术。虽然 COBOL 仍然

以面向过程的编程方式为主，但面向对象技术不失为当前发展的一个趋势。

在具体讲解 COBOL 中的面向对象技术之前，本章简单介绍了面向对象的一些基本概念。这些概念包括对象、实例、封装、类、继承、消息、多态、重载、接口等。

介绍完面向对象的基本概念之后，接下来详细讲解了 COBOL 中的类和方法。由于方法通常也是存在于对象中的，因此在方法一章中也包含了类的对象的定义。类和方法构成了 COBOL 中面向对象技术的基本框架。

本章在其后重点介绍了 COBOL 中的客户程序。客户程序用来实现对方法的调用，是 COBOL 中面向对象技术的一个特点。在 COBOL 中实现面向对象的编程，通常都需要用到两种类型的代码。其中一种类型的代码用来表示类，另一种类型的代码则对应于客户程序。

本章最后分别介绍了子类、工厂对象和异常处理的概念及应用。其中子类主要体现了面向对象技术中继承的特点及用途。工厂对象是 COBOL 中的一个特点，类似于其他面向对象语言中的构造函数。异常处理是几乎任何面向对象语言都具备的，此处仅需对此有一个大致的了解。

第 13 章 处理 VSAM 文件

VSAM 即虚拟存储访问方式的意思。使用 VSAM 组织数据，管理数据信息的文件称为 VSAM 文件。VSAM 文件所在的数据集称为 VSAM 数据集。本章将介绍 VSAM 文件的特性，以及如何使用 COBOL 处理 VSAM 文件。

13.1 VSAM 文件的基本概念

VSAM 文件对于数据的存储及管理具有很强的灵活性及高效性，因此常用来组织和存放数据。COBOL 程序中对于文件的访问，很大程度上是对 VSAM 文件的访问。关于 VSAM 文件的基本概念，需要了解以下几点，下面分别进行介绍。

13.1.1 VSAM 文件的分类及作用

学习 VSAM 文件，首先需要了解它的分类。而对于 VSAM 文件的分类，实际上也就是对相应的 VSAM 数据集的分类。VSAM 数据集通常分为以下几类。

- LDS：线性数据集（Linear Data Set）。
- ESDS：进入顺序数据集（Entry Sequenced Data Set）。
- RRDS：相对记录数据集（Relative Record Data Set）。
- KSDS：索引顺序数据集（Key Sequenced Data Set）。
- VRRDS：变长相对记录数据集（Variable-length Relative Record Data Set）。

以上 5 类数据集都属于 VSAM 数据集。关于这一点，是 VSAM 最基本的概念，一定要牢记。在这些数据集中，以 KSDS 为最常用。关于这 5 类 VSAM 数据集各自的概念及特征，将在后面的章节中详细讲解。

VSAM 文件使用特殊的方式实现对数据的组织与管理，作用体现在多方面上。VSAM 文件区别于普通文件的作用主要如下。

- 提供存储设备的独立性。
- 保障数据资料的安全性。

- 对数据具有高度的存取效率。
- 便于数据资料的转换。
- 便于数据资料的复原。
- 可以进行集中管理。
- 可适用于不同类型的作业处理方式。
- 更加容易使用与管理。

13.1.2 VSAM 文件的管理方式

系统对 VSAM 文件的管理主要涉及到两方面的内容。一方面为编目管理，另一方面为记录管理。并且，编目管理为其核心部分。

编目管理主要用于对 VSAM 文件所在的数据集进行编目。并且，所有 VSAM 文件所在的数据集都是必须要被编目的。编目管理主要是通过综合编目设备 ICF（Integrated Catalog Facility）进行的。关于 ICF 在 VSAM 中的作用，主要包含以下两条。

- 定位 VSAM 数据集。
- 记录 VSAM 数据集的结构及状态信息。

系统中所说的编目主要包含两类。一类为主编目（Master Catalog），另一类为用户编目（User Catalog）。主编目是在初始程序加载 IPL（Initial Program Load）时建立的。主编目必须要建立，并且只能建立一个。

主编目中包含有系统信息以及用户编目的信息。一个主编目中可以包含多个用户编目。同时，主编目中也可包含 VSAM 文件的信息。主编目是整个系统的中心所在，因此十分重要。

用户编目用来记录和用户相关的信息，是可选的。用户编目中主要包含用户创建的一些程序及文件，同时也包含有用户创建的 VSAM 文件。用户编目自身的信息包含在主编目中，并且用户编目下不能再有其他的用户编目。用户编目的作用有以下 3 点。

- 缩短找寻编目的时间。
- 增强数据资料的完整性。
- 提供数据资料的可携带性。

系统中以 VSAM 文件为主的组织结构如图 13.1 所示。

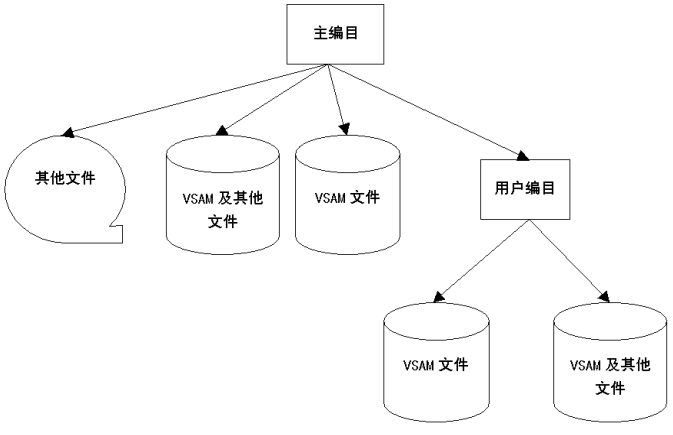

图 13.1　文件组织结构图

VSAM 文件中的记录管理通过组织 VSAM 文件中数据记录的格式，以提供多种访问方式。这些访问方式包括直接访问、顺序访问以及混合访问。同时，这些 VSAM 数据集是存在于直接访问存储设备 DASD（Direct Access Storage Device）上的。因此，通过 VSAM 文件的记录管理，能够顺序或直接存取磁盘中定长或不定长的记录。

最后需要说明的是，COBOL 只是实现对 VSAM 文件的读写操作。至于 VSAM 文件的创建、复制、删除等操作是通过 JCL 中的 IDCAMS 实用程序实现的。关于如何通过 JCL 管理 VSAM 文件，将在 JCL 扩展一章中进行讲解。

13.1.3 VSAM 文件的组织结构

VSAM 文件内部的组织结构主要涉及到 CI 和 CA 的概念。这两个概念十分重要，特别是在计算各类 VSAM 文件的大小中将会反复用到。此外，对于 VSAM 文件中数据记录的组织编号，还需要了解 RBA 的概念。以下分别对这 3 种概念进行讲解。

1. CI 的概念

CI 是 Control Interval 的简称，表示在一次 I/O 操作中数据的转移量。即 VSAM 中数据的每次存取是以 CI 为基本单位的。CI 中不仅记录了 VSAM 文件中的数据，也包含有与数据相关的一些信息。例如，如图 13.2 所示，为一个 CI 的结构。

记录 1 120 字节	记录 2 80 字节	记录 3 150 字节	自由空间（Free Space） 360 字节	R D F 3	R D F 2	R D F 1	C I D F

图 13.2　CI 结构示例

图 13.2 中 CI 的各项内容，分别介绍如下。

- CI 中的记录实际上为逻辑记录，简称为 LR（Logical Record）。VSAM 文件是以 CI 为单位进行数据存取的，而 COBOL 程序是以 LR 为单位进行记录读写。
- CI 中的 Free Space 可在创建 VSAM 文件时进行定义，要求 VSAM 预留一定比率的空白区。该空白区的大小通过 FREESPACE 参数指定。通常只在 KSDS 和 VRRDS 中存在 Free Space。
- CI 中的 RDF（Record Definition Field）用于描述 CI 中的记录。此处需要注意 RDF 与所描述记录的对应方式。上图中相同编号的 RDF 与记录相对应。当 CI 中有更多的记录时，也是通过这种首尾对应的方式进行匹配的。每一个 RDF 的大小为 3 个字节。RDF 的数量则根据 CI 中记录的数量以及记录长度是否相同而定。当记录长度都相同时，只需要 2 个 RDF。
- CI 中最右边的 CIDF（Control Interval Data Field）用于记录 Free Space 的大小及其位置。CIDF 的长度为 4 个字节。

当 COBOL 程序对 VSAM 文件中的数据记录进行访问时，系统会将整个 CI 读入地址空间。通常一个 CI 中会包含多条逻辑记录，而程序只访问其中的一条逻辑记录。这一条逻辑记录将从读入的 CI 中被选择出来，并读入程序的工作区域。

有时候逻辑记录的长度会比较大，并有可能超过一个 CI 的大小。因此需要使用 Spanned Record 技术。Spanned Record 技术将超过一个 CI 大小的逻辑记录进行切分，存放到多个 CI 之中。这一技术通常只适用于 KSDS 和 ESDS。

最后需要注意的是，整个 CI 的大小最少为 512 个字节，最多为 32KB。CI 的大小在创建时通过 CISZ 参数指定，并且必须为 512 的整倍数。

2．CA 的概念

CA 是 Control Area 的简称，由多个 CI 所组成。在同一个 VSAM 数据集中，每一个 CA 都包含有相同数目的 CI。关于 CA，有以下几点需要注意。

- CA 的大小最小为 1 个磁道，最大为一个柱面。
- CA 由 VSAM 基于“unit of allocation”（分配单元）参数进行选择。
- 所有的分配空间必须为 CA 的整倍数。

CA 中所包含的 CI 可以为多种样式。这些 CI 既可存储变长逻辑记录，也可存储定长逻辑记录，还可以不存储任何逻辑记录。如图 13.3 所示，为 CA 通常的结构。

图 13.3　CA 结构示例

3．RBA 的概念

RBA 是 Relative Byte Address 的缩写，即相对字节位置的意思。VSAM 文件中每一个逻辑记录都有一个相应的 RBA。其中第一个记录的 RBA 为 0，第二个记录的 RBA 为相对于第一个记录的偏移量。实际上，每一个记录的 RBA 都为其前面所有数据大小的总和。

例如，对于本小节中的示例 CI，其中所包含的 3 个逻辑记录的 RBA 分别如下。

- 记录 1：RBA 为 0。
- 记录 2：RBA 为 120。
- 记录 3：RBA 为 200。

通过以上 RBA 的定义方式，可以知道 RBA 与 CI 在内存中的物理位置是无关的。当程序访问 VSAM 文件中的数据时，实际上是根据数据记录的 RBA 进行存取的。因此，通过对每一记录进行 RBA 编号，VSAM 文件提供了对于存储设备的独立性。

13.1.4　VSAM 文件的设计步骤

此处所说的 VSAM 文件的设计，是指在系统中进行完整的设计。不是指通过使用 JCL 中的 IDCAMS 实用程序创建 VSAM 文件。在系统中设计 VSAM 文件，需要遵循严格的步骤进行。具体步骤依次如下。

（1）定义主编目是使用 VSAM 文件的第一步。此处可通过 AMS 中的 DEFINE MASTERCATALOG 命令来定义主编目。

（2）定义用户编目。当主编目建立好后，可以根据实际需要定义用户编目。用户编目可以通过 AMS 中的 DEFINE USERCATALOG 命令建立。

（3）定义空间。定义好目录之后，需要在磁盘中预留一块空间以存放 VSAM 文件。可使用 ASM 中的 DEFINE SPACE 命令定义空间。

（4）定义文件（Cluster）。当编目与空间都设置好后，可以定义 VSAM 文件了。此处所说的文件是从结构上的角度而言的，而不是数据上的角度。因此，这里使用 Cluster 而不是 File 来描述文件。定义 Cluster，可以通过 AMS 中的 DEFINE CLUSTER 命令进行。

（5）载入文件（File）。以上所定义的 VSAM 文件是空的，不是需要载入数据到其中。载入数据也称为 Data Set Loading，是 VSAM 文件设计的最后一步。既可以通过 AMS 中的 REPRO 命令进行载入，也可以使用应用程序进行载入。

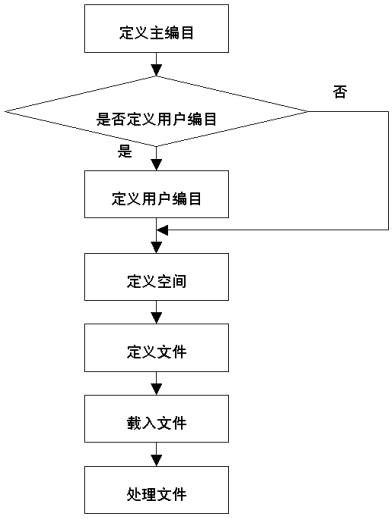

图 13.4　VSAM 文件设计步骤

当 VSAM 文件建立完毕，便可以编写 COBOL 程序或使用 AMS 命令存取其中的数据了。如图 13.4 所示，直观地反映了以上 VSAM 文件的设计步骤。

13.2　VSAM 中的 LDS

LDS 即线性数据集（Linear Data Set）。LDS 是最简单的一类 VSAM 数据集，只包含有数据，不含任何控制信息。LDS 通常仅用于保存和备份数据。

13.2.1　LDS 的结构及特征

LDS 中仅包含有数据部分，这些数据是纯粹的数据，没有形成逻辑记录。因此，在 LDS 的 CI 中是没有 RDF 及 ICDF 控制信息的。这一点是 LDS 区别于其他 VSAM 数据集最大的特点。当然，LDS 同其他 VSAM 数据集一样，也是由多个 CA 所组成，并且每个 CA 又由多个 CI 所组成。如图 13.5 所示，为 LDS 的基本结构。

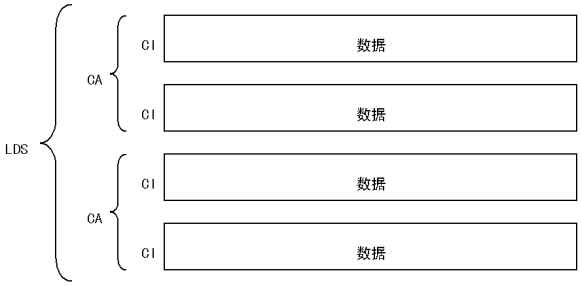

图 13.5　LDS 基本结构图

由于 LDS 中仅含有数据，并且这些数据是线性的。因此，LDS 有着某一些特征。这些特征分别如下。

- LDS 仅由 ICF 进行编目管理，而没有记录管理。
- LDS 中仅含有数据部分，而没有索引部分。
- 在 LDS 中不能定义次索引（Alternate Index）。
- 在 LDS 中不能使用 Spanned Record 技术。
- LDS 中 CI 的大小最小为 4KB，最大为 32KB。
- LDS 中的 CI 不含控制信息 RDF 以及 CIDF。
- COBOL 应用程序必须清楚 LDS 中各段数据所表示的意义。

13.2.2 计算 LDS 的空间大小

计算 VSAM 数据集的空间大小是学习 VSAM 所需要掌握的一项最基本的技能。在创建 VSAM 文件之前，都需要精确计算 VSAM 文件所在的 VSAM 数据集的空间大小。同时，根据 VSAM 数据集的空间大小，也可以清楚在程序中对该 VSAM 文件最大的数据写入量。

计算 LDS 的空间大小，通常按照以下 5 个步骤进行。这 5 个步骤对于计算其他类别的 VSAM 数据集同样也是适用的。步骤如下。

（1）计算每一 CI 中数据记录的数量。对于 LDS 而言，该数量为 1。

（2）计算所需要的 CI 数量。对于 LDS 而言，该数量通常直接给出。

（3）计算每一 CA 中所包含 CI 的数量，计算公式如下。

```
每一 CA 中所含 CI 数量 = 每一磁道上的 CI 数量 * 每一 CA 所包含的磁道数量
```

（4）计算所需要的 CA 数量，计算公式如下。

```
所需 CA 数量 = 所需 CI 数量 / (每一 CA 中所含 CI 数量 * (1- CA 中的自由空间))
```

实际上，由于在 LDS 中并不能指定 Free Space，因此上式中“CA 中的自由空间”这一项通常为 0。以上公式只是为了统一反映 VSAM 数据集空间大小的计算。当通过该公式得到的运算结果包含小数时，需要对小数部分向上取整。

（5）计算所需要的柱面或磁道数量，计算公式如下。

```
所需柱面或磁道数量 = CA 的大小 * 所需 CA 的数量
```

需要注意的是，当 CA 的大小为一个柱面时，使用柱面为单位；当 CA 的大小小于一个柱面时，使用磁道为单位。如此，便可得到 LDS 的空间大小。由于空间是以柱面或磁道为单位进行分配的，因此该空间大小也是以柱面或磁道为数量单位的。

以上计算步骤中涉及到的 CI 所需的数量以及 CA 的大小是直接给出的。CA 的大小以柱面或磁道为单位，每一柱面包含 15 个磁道。因此，通过 CA 的大小也可以得到每一 CA 所包含的磁道数量。其余一些数据，如所需 CA 数量等，则是通过前面的步骤计算所得的。

以上第 3 个步骤中所涉及到的每一磁道上的 CI 数量，需要根据查表得到。表格中主要包含 CI 的大小，物理记录的大小，以及每一磁道上物理记录的数量这 3 个数据量。并且，这 3 个数据量的单位都是字节。相应表格如表 13.1 所示。

表 13.1 物理记录对应表（1）

CI 大小	物理记录大小	每一磁道上所包含的物理记录数量
512	512	49
1024	1024	33
1536	1536	26
2048	2048	21
2560	2560	17
3072	3072	15
3584	3584	13
4096	4096	12
4608	4608	10
5120	5120	9
5632	5632	9
6144	6144	8
6656	6656	7
7168	7168	7
7680	7680	6
8192	8192	6

可以看到，以上表格中 CI 的大小都是 512 字节的整倍数。同时，此表中 CI 的大小和物理记录的大小是相等的。下面一张表中的数据反映了 CI 的大小为 2048 字节的整倍数的情况。注意到该表中 CI 的大小和物理记录的大小不一定相等。相应表格如表 13.2 所示。

表 13.2 物理记录对应表（2）

CI 大小	物理记录大小	每一磁道上所包含的物理记录数量
10240	10240	5
12288	12288	4
14336	7168	7
16384	16384	3
18432	18432	3
20480	10240	5
22528	5632	9
24576	24576	2
26624	26624	2
28672	7168	7
30720	10240	6
32768	16384	3

实际上，CI 中所包含的逻辑记录最终还是由磁道上的物理记录所体现出来的。CI、物理记录以及磁道这 3 者之间的关系如图 13.6 所示。

下面通过一个具体的例子说明 LDS 的空间大小实际是如何计算的。假设所需计算的 LDS 有如下要求及属性。

- 该 LDS 需要 500 个 CI，并且每个 CI 的大小为 4KB。
- 该 LDS 中每个 CA 的大小为一个柱面。

- 该 LDS 存储在磁盘设备上，并且设备型号为 3390。

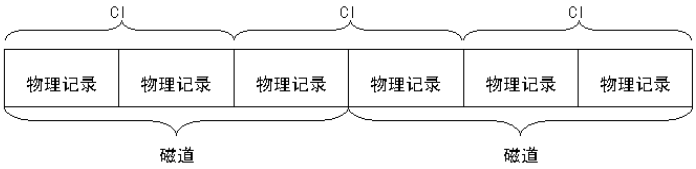

图 13.6　CI、物理记录与磁道示例关系图

根据以上信息，计算该 LDS 的空间大小步骤如下。

（1）计算每一 CI 中数据记录的数量。由于该 VSAM 数据集为 LDS，因此这一数量为 1。

（2）计算所需的 CI 数量。由所提供的信息，该数量为 500。

（3）计算每一 CA 中所包含 CI 的数量。注意到此处 CI 的大小为 4KB，即 4096 个字节。根据 CI 的大小查表得此时对应的物理记录大小也为 4096 字节，并且每一磁道上包含 12 个物理记录。因此，可得到每一磁道上所包含的 CI 数量也为 12。同时，由于此时 CA 的大小为一个柱面，因此每一 CA 包含 15 个磁道。有了这 2 个数据量，可通过以上公式计算所求结果如下。

```
每一 CA 中所含 CI 数量 = 12 * 15 = 180
```

（4）计算所需要的 CA 数量。通过第 2 个步骤可得到所需 CI 数量为 500。通过第 3 个步骤可得到每一 CA 中所含 CI 数量为 180。并且，由于该 VSAM 数据集为 LDS，因此 CA 中的自由空间为 0。有了以上这 3 个数据量，可通过公式计算所求结果如下。

```
所需 CA 数量 = 500 / (180 * (1- 0)) = 3 （注意：实际运算结果为 2.77…7，通过向上取整得到所求结果为 3）
```

（5）计算所需要的柱面或磁道数量。由于给定了 CA 的大小为一个柱面，并且通过以上第 4 个步骤得到所需 CA 数量为 3。因此，可通过公式计算所求结果如下，同时得到 LDS 空间大小为 3 个柱面。

```
所需柱面数量 = 1 * 3 = 3
```

13.3　VSAM 中的 ESDS

ESDS 是指进入顺序数据集（Entry Sequenced Data Set）。ESDS 中数据的存放必须依照先后顺序进行，并且新增的数据必须在原数据的末尾添加。该数据集与传统的 SAM File 比较类似。

13.3.1　ESDS 的结构及特征

ESDS 中同样也仅有数据部分，不过其中的数据并非线性的。ESDS 中的数据形成了相对独立的逻辑记录，并且每一记录主要通过相对字节位置 RBA 进行识别。在 ESDS 中包含有 RDF 和 CIDF 控制信息。如图 13.7 所示，为一个 ESDS 的基本结构。

ESDS 作为以先后顺序存放且组织数据记录的一种 VSAM 数据集，有着其一定的特征。这些特征分别如下。

- ESDS 中仅含有数据部分，并且数据形成了逻辑记录。
- 逻辑记录既可为定长记录，也可为变长记录。

- 逻辑记录不可删除，不过可以进行逻辑隐藏。
- 新的数据只可在原数据的末尾进行添加。
- 对逻辑记录既可进行顺序访问，也可进行直接访问。
- 当对逻辑记录进行顺序访问时，访问顺序为记录插入的顺序。
- 当对逻辑记录进行直接访问时，根据记录的 RBA 进行访问。
- 支持 Spanned Record 技术。
- 在创建时不能预留 Free Space。
- 可以更新数据记录的内容，但不能改变原有记录的长度。
- 不能改变资料记录的 RBA。

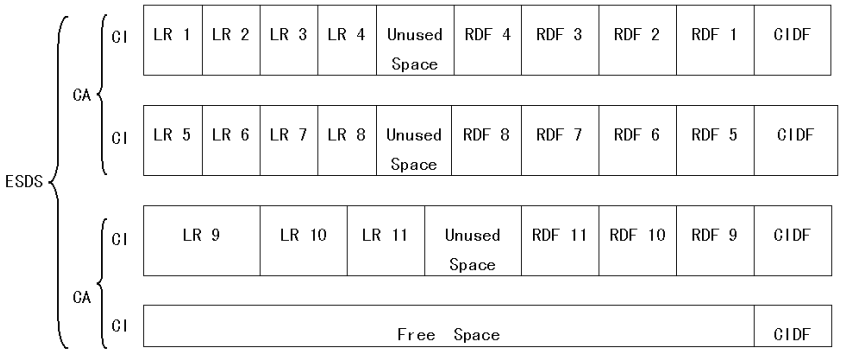

图 13.7　ESDS 基本结构图

13.3.2　ESDS 的访问方式

ESDS 与 LDS 不同，除可保存和备份数据外，也常用于在程序中进行处理和访问。可以存取 ESDS 文件中数据的程序设计语言通常有以下几种。

- COBOL
- ASSEMBLER（大型机汇编语言）
- PL/1
- RPG
- VS FORTRAN

在前面 ESDS 的特点中曾提到，ESDS 的访问方式主要有顺序访问方式和直接访问方式这两种。以下分别对这两种访问方式予以介绍。

1．顺序访问方式

顺序访问方式是对 ESDS 文件进行访问的最常用的一种方式。当对 ESDS 文件进行顺序访问时，所访问的数据记录将根据记录的存放顺序自动查找获得。

例如，对于上一小节所列出的 ESDS 文件，图中根据其逻辑记录 LR 的存放顺序进行了人为的编号。其中 LR 1 为存放的第一条逻辑记录，LR 2 为存放的第二条逻辑记录，其余依次类推。这些逻辑记录既可在 VSAM 文件设计时通过载入文件进行存放，也可通过此后的应用程序进行写入。

假设需要访问该 ESDS 文件中的逻辑记录 LR 10。如果通过顺序访问方式，访问过程依次如下。

- 从包含有逻辑记录 LR 1、LR 2、LR 3、LR 4 的 CI 开始访问。由于 LR 10 并不在该条 CI 内，因此继续访问下一条 CI。
- 访问包含有逻辑记录 LR 5、LR 6、LR 7、LR 8 的 CI。由于该条 CI 中也没有 LR 10，因此继续访问下一条 CI。
- 访问包含有逻辑记录 LR 9、LR 10、LR 11 的 CI。该条 CI 中包含有 LR 10，因此将该 CI 读入地址空间，并将其中的逻辑记录 LR 10 读入程序工作区域。

2．直接访问方式

直接访问方式是通过 ESDS 文件中逻辑记录的 RBA 进行访问的。关于 RBA，主要有以下几点需要注意的地方。

- 当对 ESDS 文件中的逻辑记录进行直接访问时，必须要提供该记录的 RBA。
- 逻辑记录的 RBA 只和该记录在 ESDS 文件中所在的位置有关。当一条新的记录被添加到 ESDS 文件中时，其 RBA 便同时得到确定。
- 第一条逻辑记录的 RBA 为 0。
- 除第一条逻辑记录外，其余逻辑记录的 RBA 为相对于第一条逻辑记录的偏移量。
- 可通过建立次索引的方式跟踪 RBA 的变化轨迹。
- RBA 总是为全字二进制整数（关于全字的概念，将在大型机汇编语言扩展一章中详细讲解）。

此处假设上一小节中所列出的 ESDS 文件中每条 CI 的大小为 4KB。则以上各条 CI 中第一个逻辑记录的 RBA 分别如下。

- 第一条 CI 中的第一个逻辑记录为 LR 1，RBA 数值为 0。
- 第二条 CI 中的第一个逻辑记录为 LR 5，RBA 数值为 4096。
- 第三条 CI 中的第一个逻辑记录为 LR 9，RBA 数值为 8192。
- 第四条 CI 中没有逻辑记录，因此不存在 RBA 数值。

同时，如果第三条 CI 中的逻辑记录 LR 9 的长度为 200 个字节，可得到 LR 10 的 RBA 为 8392。由于 LR 10 是紧接着 LR 9 存放的，因此可直接用 LR 9 的 RBA8192 加上 LR 9 的长度 200 得到。但对于 LR 9 而言，由于其和 LR 8 之间还有未使用空间及控制信息，因此不能如此计算。当通过直接访问方式访问 LR 10 时，可直接通过其 RBA 值 8392 进行访问了。

最后，关于 ESDS 文件中数据的访问及其相关操作，有以下几点需要特别注意。

- ESDS 文件提供两种访问方式，分别为顺序访问方式和直接访问方式。其中直接访问方式是根据逻辑记录的 RBA 进行访问的。
- ESDS 中可访问的逻辑记录包括定长记录、变长记录以及 Spanned Record。
- 对 ESDS 文件添加新的数据时，只可在该文件中原有数据的末尾进行添加。在原有数据的开头或中间进行添加都是不允许的；这一点同时也保证了 ESDS 文件中的数据均为顺序存放的，体现了 ESDS 的特征。
- ESDS 文件中逻辑记录的内容可以被更改，但逻辑记录的长度不可被更改。因为如果某

一逻辑记录的长度被更改，则将会影响到后面记录的 RBA 值。

- ESDS 文件中的逻辑记录不可在物理上将其删除。因为若将其从物理上删除了，相对于将该记录的长度更改为 0。这样仍然会影响后面记录的 RBA 值。不过，在应用程序中可将其内容视为空，以此从逻辑上将其删除，实现逻辑隐藏。

13.3.3 Spanned Record 技术

前面提到，ESDS 文件是支持 Spanned Record 技术的。关于 Spanned Record 技术，简单地说就是当逻辑记录长度大于一个 CI 的大小时，将其进行切分，并分别使用多个 CI 进行存放。如图 13.8 所示，反映了这一技术的特点。

图 13.8 Spanned Record 技术示例图

可以看到，以上这条逻辑记录被划分为 3 个逻辑记录分段，并依次存放在 3 个 CI 中。当程序需要访问该条逻辑记录时，需要将这 3 个 CI 同时读入地址空间。并将这 3 个 CI 中的逻辑记录分段整合为一条完整的逻辑记录，读入程序的工作区域。

关于 Spanned Record 技术，有以下几点需要注意。

- Spanned Record 技术仅适用于 ESDS 文件和 KSDS 文件。
- 使用 Spanned Record 技术处理的逻辑记录其长度应该大于一个 CI 的大小。
- 所处理的逻辑记录需要通过 AMS 中的 DEFINE CLUSTER 里的 SPANNED 参数进行定义。
- 所处理的逻辑记录以 CI 的大小为单位进行分割，并将分割后的内容分别存放在多个 CI 之中。
- 存放逻辑记录分段的每个 CI 中应有两个 RDF 控制信息，以保证其连贯性。
- 当文件中各条逻辑记录的长度变化范围较大时，Spanned Record 技术可节省数据集的存储空间。原因是使用 Spanned Record 技术后，CI 中未被使用的空间大小将相对减少。
- Spanned Record 技术只可跨越多个 CI 进行，而不可跨越 CA。

实际上，当逻辑记录的长度大于 CI 的大小减去 7 倍的 RDF 与 CIDF 大小之和时，将对其处理。此时，VSAM 文件管理系统会将其视做一个 Spanned Record，并使用一个以上的 CI 进行存放。

需要注意的是，使用 Spanned Record 技术划分的逻辑记录是从某一 CI 的开始位置进行存放的。同时，整个逻辑记录的最大长度为一个 CA 的大小减去其中控制信息的长度。

当在应用程序中访问 Spanned Record 时，主要有以下两点需要注意。

- 必须使用 MOVE 模式对其进行访问。

- 程序的工作区域以及 I/O 区域必须足够大。其空间大小至少应该大于数据集中最大的一个 Spanned Record 的长度。

13.3.4 计算 ESDS 的空间大小

计算 ESDS 的空间大小同计算 LDS 的空间大小类似，依次按照 5 个步骤进行的。这 5 个步骤分别如下。

（1）计算每一 CI 中逻辑记录的数量。

（2）计算所需要的 CI 数量。

（3）计算每一 CA 中所包含 CI 的数量。

（4）计算所需要的 CA 数量。

（5）计算所需要的柱面或磁道数量。

此外，对于包含有定长逻辑记录和变长逻辑记录的 ESDS，具体计算方式将略有不同。下面分别作介绍。

1．计算含定长记录的 ESDS 空间大小

假设所需计算的 ESDS 有如下要求及属性。

- 该 ESDS 的 CI 大小为 2KB。
- 该 ESDS 的 CA 大小为 5 个磁道。
- 该 ESDS 的逻辑记录大小为 200 个字节。
- 该 ESDS 所需逻辑记录的数量为 5000 个。
- 该 ESDS 存储在磁盘设备上，并且设备型号为 3390。

根据以上步骤，计算该 ESDS 的空间大小如下。

（1）计算每一 CI 中逻辑记录的数量。由于该 ESDS 中每个逻辑记录的大小为 200 个字节，因此为定长逻辑记录。对于定长逻辑记录，可以用 CI 中逻辑记录所占总空间的大小除以每一逻辑记录的大小得到所求数据。其中，CI 中逻辑记录所占总空间的大小由 CI 的大小减去控制信息及自由空间的大小得到。

当 CI 中存放的是定长逻辑记录时，每一 CI 中的控制信息将只包含 2 个 RDF 及 1 个 CIDF。由于 RDF 的大小为 3 个字节，CIDF 的大小为 4 个字节，因此控制信息总共为 10 个字节。

CI 自由空间的大小则由 CI 的大小乘以自由空间的比率得到。对于 ESDS 而言，自由空间比率为 0。此处只是为了统一反映 VSAM 数据集空间大小的计算。

根据以上分析，可得到在逻辑记录为定长记录的情况下，每一 CI 中记录数量的计算公式如下。

```
每一 CI 中逻辑记录的数量 = (CI 的大小 — CI 的大小 * CI 自由空间比率 - 10) / 逻辑记录的大小
```

需要注意的是，为保证 ESDS 的空间足够存放所需逻辑记录，对于以上计算结果需要向下取整。这点同计算所需 CI 数量及所需 CA 数量时的向上取整是不同的。

对于本例中的 ESDS，根据以上公式可计算本步骤所求数据如下。

```
每一 CI 中逻辑记录的数量 = (2048 — 2048 * 0 - 10) / 200 = 10
```

（2）计算所需要的 CI 数量。此处所需 CI 数量同计算 LDS 空间时不同，并没直接给出来。

不过，此处给出了所需逻辑记录的数量。因此，可以将所需逻辑记录的数量除以步骤 1 得到的每一 CI 中所含记录的数量得到所求结果。计算公式如下。

```
所需 CI 数量 = 所需逻辑记录数量 / 每一 CI 中数据记录的数量
```

为保证 ESDS 的空间足够存放所需逻辑记录，对于以上计算结果需要向上取整。对于本例中的 ESDS，根据以上公式可计算本步骤所求数据如下。

```
所需 CI 数量 = 5000 / 10 = 500
```

（3）计算每一 CA 中所包含 CI 的数量。CA 是以磁道或柱面为单位的，而磁道通过物理记录则是可以与 CI 对应起来的。同时，对于 3390 的设备，每一柱面为 15 个磁道。即当 CA 以柱面为单位时，也是可以转换为磁道的。因此，计算本步骤所求数据，最终也是以磁道为媒介的，计算公式如下。

```
每一 CA 中所含 CI 数量 = 每一磁道上的 CI 数量 * 每一 CA 所包含的磁道数量
```

以上公式同计算 LDS 的空间大小时对应步骤中的公式实际上是相同的。此外，对于本例中的 ESDS，每一磁道上的 CI 数量仍然需要通过查表得到。由于此时 CI 大小为 2KB，因此通过查表 13.1 可得到每一磁道上的 CI 数量为 21。因此对于本步骤所求的数据计算如下。

```
每一 CA 中所含 CI 数量 = 21 * 5 = 105
```

（4）计算所需要的 CA 数量。本步骤同计算 LDS 的空间大小时所对应步骤中的公式一致，其计算公式如下。

```
所需 CA 数量 = 所需 CI 数量 / (每一 CA 中所含 CI 数量 * (1- CA 中的自由空间))
```

注意以上公式中 CA 中的自由空间这一项对于 ESDS 而言仍然为 0。同时，为保证空间大小足够，计算结果仍然需要向上取整。对于本例中的 ESDS，此项数据计算如下。

```
所需 CA 数量 = 500 / (105 * (1- 0)) = 5
```

（5）计算所需要的柱面或磁道数量。本步骤同计算 LDS 的空间大小时所对应步骤中的公式一致，其计算公式如下。

```
所需柱面或磁道数量 = CA 的大小 * 所需 CA 的数量
```

由于此处 CA 是以磁道为单位的，因此需要计算所需磁道数量。最终可得到该 ESDS 的空间大小为 25 个磁道。计算如下。

```
所需磁道数量 = 5 * 5 = 25
```

2．计算含变长记录的 ESDS 空间大小

假设所需计算的 ESDS 有如下要求及属性。

- 该 ESDS 的 CI 大小为 4KB。
- 该 ESDS 的 CA 大小为 1 个磁道。
- 该 ESDS 的平均逻辑记录大小为 100 个字节。
- 该 ESDS 所需逻辑记录的数量为 8000 个。
- 该 ESDS 存储在磁盘设备上，并且设备型号为 3390。

对于存放变长记录的 ESDS 而言，空间大小的计算主要是第一个步骤同定长记录时的情

况不同。存放变长记录的 CI 中每一记录都有一个对应的 RDF，同时，整个 CI 中只有一个 CIDF。由于 RDF 的大小都为 3 个字节，而 CIDF 的大小为 4 个字节。因此，在变长记录情况下，计算每一 CI 中逻辑记录的数量公式如下。

```
每一 CI 中逻辑记录的数量 = (CI 的大小 - CI 的大小 * CI 自由空间比率 - 4) / (平均逻辑记录的大小 + 3)
```

对于本例中的 ESDS 而言，此项数据计算如下。

```
每一 CI 中逻辑记录的数量 = (4096 - 4096 * 0 - 4) / (100 + 3) = 39
```

注意到为保证 ESDS 有足够的空间，以上对计算结果进行了向下取整。后面的步骤与在定长记录的情况下类似。下面直接给出本例中每一步的计算过程。

```
所需 CI 数量 = 8000 / 39 = 206
每一 CA 中所含 CI 数量 = 12 * 1 = 12
所需 CA 数量 = 206 / (12 * (1-0)) = 18
所需磁道数量 = 1 * 18 = 18
```

为保证空间足够大，对于以上计算所需 CI 数量和所需 CA 数量的运算结果都进行了向上取整。最终可以得到，该含有变长记录的 ESDS 空间大小应该为 18 个磁道。

13.4 VSAM 中的 RRDS

RRDS 即相对记录数据集（Relative Record Data Set）。在 RRDS 中存在着许多固定长度的存储区域，称为 Slot。RRDS 中的数据便是存放在这些 Slot 之中。

13.4.1 RRDS 的结构及特征

RRDS 结构上最大的特点便是每一个 CI 中的数据部分都被划分为了等长的 Slot。这些 Slot 都被进行了编号，称之为相对记录号码 RRN（Relative Record Number）。如图 13.9 所示，为某一 RRDS 的基本结构。

图 13.9 RRDS 基本结构图

基于以上结构，RRDS 主要有以下特征。

- RRDS 中只有数据部分。
- RRDS 中的逻辑记录存放在预先设置好的 Slot 中。
- 每一个 Slot 都有一个对应的 RDF。该 RDF 用于指示该 Slot 中是否存在数据。

- 每一个 CI 中包含有相等数目的 Slot，其数目根据 CI 的大小及记录的长度决定。
- Slot 被编号，编号被称为相对记录号码 RRN，并且 RRN 唯一。
- 逻辑记录按 RRN 进行存放。
- 逻辑记录只能为定长记录。
- 逻辑记录可以删除，并将其所占 Slot 置为空状态。
- 新的逻辑记录只能在空的 Slot 上添加。
- 对逻辑记录既可进行顺序访问，也可进行直接访问。
- 当对 RRDS 进行随机载入数据时，需要通过用户程序进行。
- 在创建 RRDS 时不能预留 Free Space。
- 可以更新逻辑记录的内容，但不可更改逻辑记录的长度。
- 不能建次索引。
- 不能改变逻辑记录的 RRN。

13.4.2 RRDS 的访问方式

同 ESDS 文件一样，可以存取 RRDS 文件中数据的程序设计语言通常也有以下几种。

- COBOL
- ASSMBLER
- PL/1
- RPG
- VS FORTRAN

RRDS 的访问方式通常也包含有顺序访问方式和直接访问方式两种。对于顺序访问方式，需要注意以下两点。

- 访问的顺序依据 RRN 的值从小到大的顺序。
- 顺序访问时，对于空的 Slot，VSAM 文件管理系统将自动跳过。

当对 RRDS 中的数据记录进行直接访问时，将主要根据记录的 RRN 进行。对于直接访问方式，主要需要注意以下两点。

- VSAM 文件管理系统将在内部把所提供的 RRN 转换为 RBA，以对记录进行访问。
- 不可直接使用 RBA 访问 RRDS 中的数据。

此外，在 RRDS 中还可存在一种访问方式，称作跳跃顺序访问方式。该访问方式的处理过程类似于直接访问，也是通过 RRN 的值获得数据记录的。不过在该方式中，数据记录必须根据 RRN 值有小到大的顺序进行检索。

13.4.3 计算 RRDS 的空间大小

计算 RRDS 空间的大小同前面计算 ESDS 空间的大小类似。由于 RRDS 中的逻辑记录只能为定长的，因此 RRDS 空间大小的计算只存在这一种情况。

下面直接给出计算 RRDS 空间大小的各个步骤及相应的计算公式。

（1）计算每一 CI 中逻辑记录的数量。计算公式如下。

```
每一 CI 中逻辑记录的数量 = (CI 的大小 - CI 的大小 * CI 自由空间比率 - 4) / (逻辑记录的大小 + 3)
```

（2）计算所需要的 CI 数量。计算公式如下。

```
所需 CI 数量 = 所需逻辑记录数量 / 每一 CI 中数据记录的数量
```

（3）计算每一 CA 中所包含 CI 的数量。计算公式如下。

```
每一 CA 中所含 CI 数量 = 每一磁道上的 CI 数量 * 每一 CA 所包含的磁道数量
```

（4）计算所需要的 CA 数量。计算公式如下。

```
所需 CA 数量 = 所需 CI 数量 / (每一 CA 中所含 CI 数量 * (1- CA 中的自由空间))
```

（5）计算所需要的柱面或磁道数量，公式如下所示。

```
所需柱面或磁道数量 = CA 的大小 * 所需 CA 的数量
```

下面结合一个具体的实例，以便更好地说明 RRDS 空间大小的计算方式。假设所需计算的 RRDS 具有如下要求及属性。

- 该 RRDS 的 CI 大小为 1KB。
- 该 RRDS 的 CA 大小为 1 个柱面。
- 该 RRDS 的逻辑记录大小为 100 个字节。
- 该 RRDS 所需逻辑记录的数量为 5000 个。
- 该 RRDS 存储在磁盘设备上，并且设备型号为 3390。

根据以上步骤，对于该 RRDS 空间大小的完整计算过程如下。

```
每一 CI 中逻辑记录的数量 = (1024 - 1024 * 0 - 4) / (100 + 3) = 9
所需 CI 数量 = 5000 / 9 = 556
每一 CA 中所含 CI 数量 = 33 * 15 = 495
所需 CA 数量 = 556 / (495 * (1-0)) = 2
所需磁道数量 = 1 * 2 = 2
```

由此可见，该 RRDS 所需空间大小为 2 个柱面。需要注意的是，为使空间足够容下所有记录，对于以上计算过程中的除法运算结果分别处理如下。

- 计算每一 CI 中逻辑记录的数量时，对运算结果向下取整。
- 计算所需 CI 数量时，对运算结果向上取整。
- 计算所需 CA 数量时，对运算结果向上取整。

13.5 VSAM 中的 KSDS

KSDS 即索引顺序数据集（Key Sequenced Data Set）。KSDS 最主要的特点是通过索引来组织和管理其中的数据。KSDS 是 5 种 VSAM 数据集中最复杂的一种，同时也是在实际中用得最多的一种。因此，KSDS 是最重要的一种 VSAM 数据集。

13.5.1 KSDS 的结构及特征

在 KSDS 中，逻辑记录是按照记录关键字（Record Key）的升序进行存放的。KSDS 文件属于索引文件，这点同 ISAM 是类似的。不过，KSDS 的存取方式与 ISAM 是不同的，并且 KSDS 具有更高的存取效率。同时，KSDS 还提供了最大的存储变通性。

KSDS 的结构中包含两块内容。一块为索引部分的内容，另一块为数据部分的内容。例如，如图 13.10 所示，为 KSDS 的基本结构。

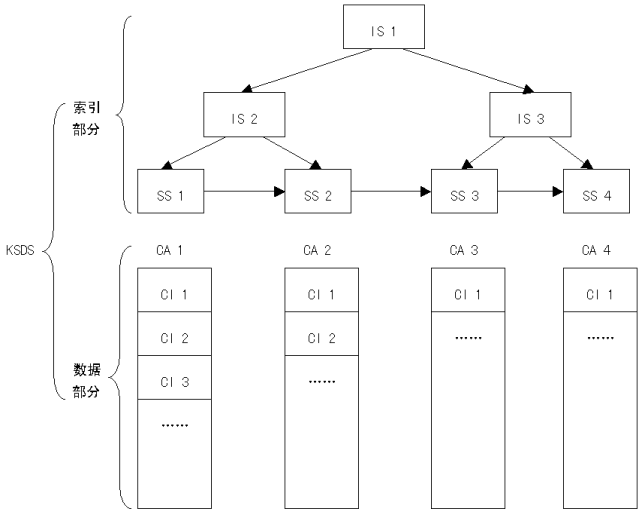

图 13.10　KSDS 基本结构图

以上结构图中的 IS（Indexed Set）为索引设施，SS（Sequence Set）为顺序设施。基于以上结构，KSDS 的特征主要如下。

- 同时拥有数据部分和索引部分。
- 逻辑记录既可为定长，也可为变长。
- 可对逻辑记录进行全方位的增加、修改、删除操作。
- 访问方式可以有顺序访问和直接访问。
- 新增逻辑记录可以在原数据中的任何一个位置添加。
- 可以改变原有逻辑记录的长度，即可增加或缩短原有记录的长度。
- 可以真正删除逻辑记录，并将删除逻辑记录所占的空间收回，作为 Free Space。
- 可建立次索引（Alternate Index），并以次关键字（Alternate Key）存取数据。
- 可以改变逻辑记录的 RBA 值。
- 能够在创建时预留 Free Space。
- 能够支持 Spanned Record 技术。

13.5.2　KSDS 中的 Key 及索引

在 KSDS 文件中，每条逻辑记录中的同一位置都有唯一的一个关键字（Key）。关于此处的 Key，有以下几点需要注意。

- Key 为定长数据，且位于每条逻辑记录中相同的位置。
- 逻辑记录根据 Key 值的升序排列存放在文件之中。
- 每一个 Key 的值必须唯一，以确定逻辑记录在文件中的排序序列。
- Key 的值必须按从小到大的顺序连续排列。
- 一旦 Key 的值被设定后，该值不可被更改。不过，可以将包含该 Key 的整条逻辑记录删除，这样同时也将该 Key 删除了。

通过上一小节的 KSDS 基本结构图可以看到，在 KSDS 中主要涉及索引部分和数据部分两块内容。在数据部分中，每一 CA 的大小尤其重要。

在索引部分，主要包含两类数据，分别为 IS 和 SS。其中 SS 是 Sequence Set 的缩写。关于 SS，需要注意如下几点。

- 每一 SS 存放在一个 CI 中，并且每一 SS 对应于一个 CA。
- SS 包含所对应 CA 中的每一个 CI 的指示器及 Key 的相关信息。
- SS 水平指向下一个 SS。

IS 为 Indexed Set 的缩写。关于 IS，需要注意如下几点。

- IS 为索引部分除 SS 外剩下的部分。
- IS 向下指向后续的 IS 或者指向 SS。
- 只有一条 IS 位于最高层。

最后需要注意的是，KSDS 的索引部分是在对其载入数据时自动建立的。并且，索引部分的 SS 包含所对应 CA 中每条 CI 的一个入口。该入口为对应 CI 中最大的 Key 值。而 IS 则相应的包含每条 SS 的一个入口。如图 13.11 所示，反映了 KSDS 中的 Key 值对应情况。

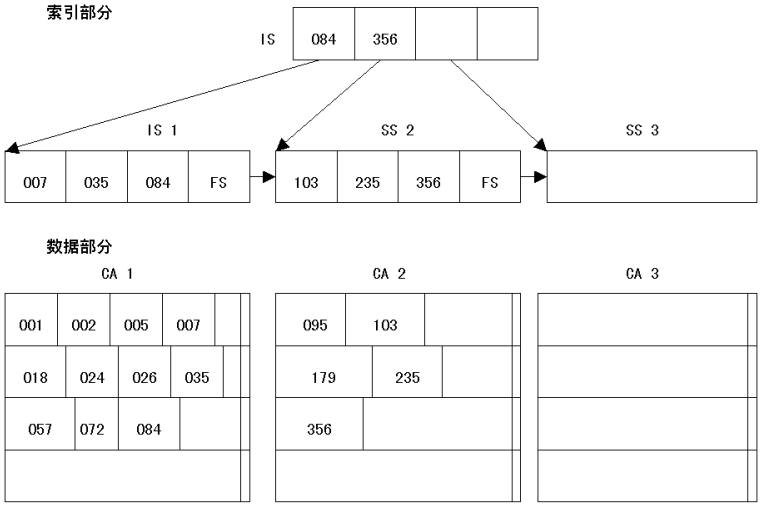

图 13.11　KSDS 中的 Key 值对应情况

上图中 CA 里的每一行代表其中的一个 CI。CI 中每一个单元格代表其中的一个逻辑记录，单元格中的数值表示该逻辑记录的 Key 值。由此，可以看出 KSDS 中索引部分与数据部分是如何通过 Key 值对应起来的。

13.5.3　KSDS 的访问方式

能对 KSDS 中的数据进行存取的程序设计语言通常有以下几种。

- COBOL
- ASMMBLER
- PL/1
- RPG

KSDS 的访问方式也包含有顺序访问方式和直接访问方式这两种。对于顺序访问方式，需要注意以下几点。

- 顺序访问通过 SS 中的 Key 值对数据进行访问。
- 访问顺序既可为升序，也可为降序。
- 当一个 CA 中的数据遍历完后仍未找到所求数据，将通过 SS 中的水平指针 FS 访问下一 CA。
- 可从 KSDS 文件中的任何一个位置开始顺序访问。

当对 KSDS 中的数据记录进行直接访问时，将从 KSDS 的索引部分开始进行访问。对于直接访问方式，需要注意以下几点。

- 直接访问从索引部分中最高一层的 IS 开始访问。
- 通过索引部分层级之间的向下指针搜寻可能包含有所求记录的 CA 及 CI。
- 检索可能包含有所求记录的整条 CI，以对记录进行访问，或判断该记录不存在。

同 RRDS 一样，KSDS 中也存在一种所谓的跳跃顺序访问方式。该访问方式根据 SS 搜寻所求记录，并且搜寻顺序是按照 Key 值的升序进行的。

13.5.4 CI 及 CA 分割技术

由于 KSDS 文件中逻辑记录必须按照其 Key 值由小到大的顺序排列。因此，当对 KSDS 文件中新增逻辑记录时，将可能会用到 CI 及 CA 的分割。

1. CI 的分割

此处结合一个具体的例子来讲解什么时候将会用到 CI 的分割，以及 CI 是如何分割的。假设某一 KSDS 中的一个 CA 及其对应的 SS 内容如图 13.12 所示。

如果需要新增一个 Key 值为 1024 的逻辑记录，根据 Key 值顺序，只能在 CI 1 中添加。并且，该逻辑记录应该添加在 Key 值为 1019 和 1350 的这两个逻辑记录之间。由于 CI 1 已没有更多的空间装下该条逻辑记录，将会把 CI 1 中的逻辑记录对等分割为两半。其中一半逻辑记录将存放在该 CA 中空白的 CI 3 中。添加该记录后的 KSDS 内容如图 13.13 所示。

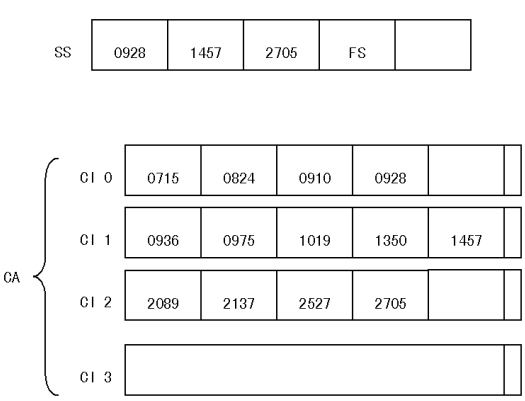

图 13.12 KSDS 原始数据

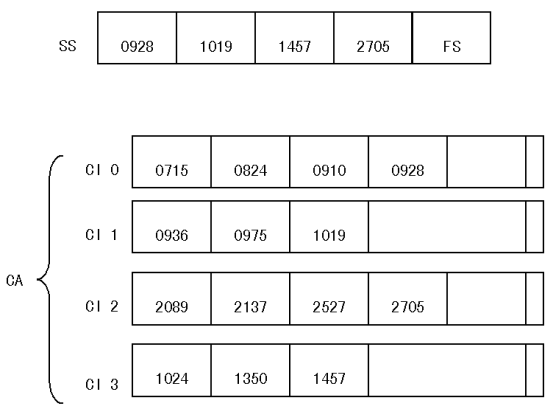

图 13.13 CI 分割后 KSDS 的内容

2. CA 的分割

如果对 KSDS 新增数据时，CA 中没有空白 CI 用于进行 CI 分割了，将进行 CA 分割。

例如，当在以上 KSDS 中再新增两个 Key 值分别为 2107 和 2348 的逻辑记录，KSDS 数据部分内容将如图 13.14 所示。

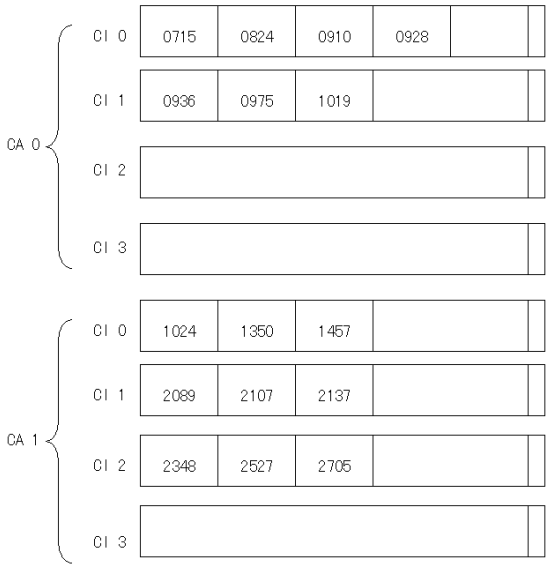

图 13.14　CA 分割后数据部分的内容

需要注意的是，由于此时存在两个 CA，因此索引部分中相应的也有两个 SS。同时，这两个 SS 的上层 IS 中的数值也应相应地改变。因此，CA 分割后，该 KSDS 的索引部分内容应该如图 13.15 所示。

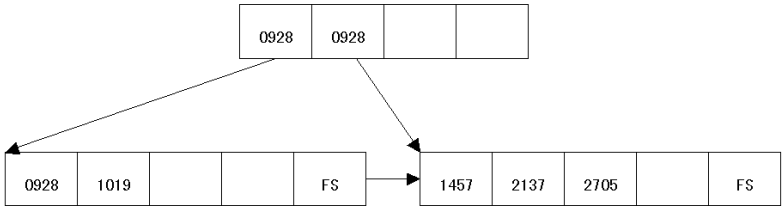

图 13.15　CA 分割后索引部分的内容

13.5.5　次索引技术

次索引（Alternate Index）简称为 AIX。注意，此处的 AIX 同 IBM 的 Unix 操作系统“AIX”是不同的。次索引为逻辑记录提供次级 Key，并可同主索引中的 Key 进行交互，以提高记录访问效率。次索引本身相当于一个特殊结构的 KSDS。

例如，对于一个 VSAM 数据集，若其中的每条记录为一个银行用户信息。并且，主索引为用户的账号。可建立次索引如下。

- 以用户姓名建立一个次索引。
- 以用户注册时间建立另一个次索引。

此时，若只知道用户的姓名，可通过第一个次索引查找该用户的记录了。同时注意到，次索引中的 Key 不一定唯一。例如，可以有相同姓名的用户，也可以有在相同时间注册的用

户。因此，次索引中不唯一的 Key 值将对应多条逻辑记录。

在每条次索引中所包含的数据信息分别如下。

- 标题信息。
- 次关键字（Alternate Key）。
- 一个或多个指向主数据集中逻辑记录的指针。

对于以上的次关键字，同主关键字一样，仍然为逻辑记录中固定位置上拥有固定长度的数据。对于主数据集及指向主数据集的指针，可以有以下两种情况。

- 主数据集为 KSDS，指针为主关键字（Prime Key）的值。
- 主数据集为 ESDS，指针为 RBA 的值。

13.5.6 计算 KSDS 数据部分的空间大小

首先需要说明的是，KSDS 数据部分的空间大小并非整个 KSDS 的空间大小。KSDS 中的逻辑记录既可以为定长记录也可以为变长记录。对应于这两种不同的情况，KSDS 数据部分的空间大小计算方式分别如下。

1．计算含定长记录的 KSDS 数据部分的空间大小

含定长记录的 KSDS 数据部分空间大小的计算同含定长记录的 ESDS 空间大小的计算基本类似。下面直接给出每一步骤所对应的计算公式。

```
每一 CI 中逻辑记录的数量 = (CI 的大小 - CI 的大小 * CI 自由空间比率 - 10) / 逻辑记录的大小
所需 CI 数量 = 所需逻辑记录数量 / 每一 CI 中数据记录的数量
每一 CA 中所含 CI 数量 = 每一磁道上的 CI 数量 * 每一 CA 所包含的磁道数量
所需 CA 数量 = 所需 CI 数量 / (每一 CA 中所含 CI 数量 * (1- CA 中的自由空间))
所需柱面或磁道数量 = CA 的大小 * 所需 CA 的数量
```

不过需要注意的是，在 KSDS 中是可以预留自由空间（Free Space）的。例如，以下为一个含有定长记录的 KSDS 相关要求及属性。

- 该 KSDS 的 CI 大小为 4KB。
- 该 KSDS 的 CA 大小为 1 个柱面。
- 该 KSDS 的逻辑记录大小为 100 个字节。
- 该 KSDS 所需逻辑记录的数量为 10000 个。
- 该 KSDS 的 CI 自由空间比率为 20%，CA 自由空间比率为 10%。
- 该 KSDS 存储在磁盘设备上，并且设备型号为 3390。

根据以上计算步骤，并结合以上实际数据，计算该 KSDS 数据部分的空间大小如下。

```
每一 CI 中逻辑记录的数量 = (4096 - 4096 * 20% - 10) / 100 = 32   /*此处对结果向下取整*/
所需 CI 数量 = 10000 / 32 = 313                                  /*此处对结果向上取整*/
每一 CA 中所含 CI 数量 = 12 * 15 = 180
所需 CA 数量 = 313 / (180 * (1- 10%)) = 2                        /*此处对结果向上取整*/
所需柱面或磁道数量 = 1 * 2 = 2                                   /*最终得到空间大小为 2 个柱面*/
```

2．计算含变长记录的 KSDS 数据部分的空间大小

含定长记录的 KSDS 数据部分空间大小的计算同 RRDS 以及含变长记录的 ESDS 的基本相似。下面直接给出每一步骤所对应的计算公式。

```
每一 CI 中逻辑记录的数量 = (CI 的大小 — CI 的大小 * CI 自由空间比率 - 4) / (平均逻辑记录的大小 + 3)
所需 CI 数量 = 所需逻辑记录数量 / 每一 CI 中数据记录的数量
每一 CA 中所含 CI 数量 = 每一磁道上的 CI 数量 * 每一 CA 所包含的磁道数量
所需 CA 数量 = 所需 CI 数量 / (每一 CA 中所含 CI 数量 * (1- CA 中的自由空间))
所需柱面或磁道数量 = CA 的大小 * 所需 CA 的数量
```

例如，以下为一个含有变长记录的 KSDS 相关要求及属性。

- 该 KSDS 的 CI 大小为 2KB。
- 该 KSDS 的 CA 大小为 10 个磁道。
- 该 KSDS 的平均逻辑记录大小为 200 个字节。
- 该 KSDS 所需逻辑记录的数量为 30000 个。
- 该 KSDS 的 CI 自由空间比率为 10%，CA 自由空间比率为 5%。
- 该 KSDS 存储在磁盘设备上，并且设备型号为 3390。

根据以上计算步骤，并结合以上实际数据，计算该 KSDS 数据部分的空间大小如下。

```
每一 CI 中逻辑记录的数量 = (2048 - 2048 * 10% - 4) / (200 + 3) = 9  /*此处对结果向下取整*/
所需 CI 数量 = 30000 / 9 = 3334                                     /*此处对结果向上取整*/
每一 CA 中所含 CI 数量 = 21 * 10 = 210
所需 CA 数量 = 3334 / (210 * (1- 5%)) = 28                          /*此处对结果向上取整*/
所需柱面或磁道数量 = 10 * 28 = 280                                  /*最终得到空间大小为 280 个磁道*/
```

13.6 VSAM 中的 VRRDS

VRRDS 即变长相对记录数据集（Variable-length Relative Record Data Set）。将 VRRDS 放在 KSDS 之后讲，是因为 VRRDS 实际上也相当于一种特殊的 KSDS。只是对于该 KSDS 的处理方式同 RRDS 比较类似。

13.6.1 VRRDS 的结构及特征

VRRDS 作为一种特殊的 KSDS，结构同 KSDS 基本类似。关于 VRRDS 的基本结构，可以参照图 13.10 所示的 KSDS 基本结构。

VRRDS 的处理方式同 RRDS 的比较类似。最主要的一点在于二者都是通过相对记录编号 RRN（Relative Record Number）处理其中的数据的。同时，VRRDS 的定义方式也同 RRDS 的比较类似。关于 VRRDS，主要有以下特征。

- VRRDS 中的逻辑记录都为变长记录。
- 每一逻辑记录都有唯一的一个 RRN，并且记录按 RRN 的升序排列。
- VRRDS 中的逻辑记录通过 RRN 存取和访问。
- VRRDS 中没有 Slot，这点和 RRDS 是不同的。
- VRRDS 中存在索引部分。
- 在创建 VRRDS 时可以预留自由空间（Free Space）。
- 定义 VRRDS 同定义 RRDS 基本类似。不过定义 VRRDS 时指明的平均记录长度和最大记录长度不能相等。

13.6.2 计算 VRRDS 数据部分的空间大小

关于 VRRDS 数据部分空间大小的计算同前面所讲解的 KSDS 中数据部分空间大小的计

算类似。下面直接给出每步对应的计算公式。

```
每一 CI 中逻辑记录的数量 = (CI 的大小 - CI 的大小 * CI 自由空间比率 - 4) / (平均逻辑记录的大小 + 3)
所需 CI 数量 = 所需逻辑记录数量 / 每一 CI 中数据记录的数量
每一 CA 中所含 CI 数量 = 每一磁道上的 CI 数量 * 每一 CA 所包含的磁道数量
所需 CA 数量 = 所需 CI 数量 / (每一 CA 中所含 CI 数量 * (1- CA 中的自由空间))
所需柱面或磁道数量 = CA 的大小 * 所需 CA 的数量
```

假设某一 VRRDS 具有以下相关要求及属性。

- 该 VRRDS 的 CI 大小为 4KB。
- 该 VRRDS 的 CA 大小为 5 个磁道。
- 该 VRRDS 的平均逻辑记录大小为 100 个字节。
- 该 VRRDS 所需逻辑记录的数量为 10000 个。
- 该 VRRDS 的 CI 自由空间比率为 10%，CA 自由空间比率也为 10%。
- 该 VRRDS 存储在磁盘设备上，并且设备型号为 3390。

对于该 VRRDS 数据部分的空间大小计算如下。

```
每一 CI 中逻辑记录的数量 = (4096 - 4096 * 10% - 10) / 100 = 36   /*此处对结果向下取整*/
所需 CI 数量 = 10000 / 36 = 278                                  /*此处对结果向上取整*/
每一 CA 中所含 CI 数量 = 12 * 5 = 60
所需 CA 数量 = 278 / (60 * (1- 10%)) = 6                         /*此处对结果向上取整*/
所需柱面或磁道数量 = 5 * 6 = 30                                  /*最终得到空间大小为 30 个磁道*/
```

13.7 VSAM 文件及其空间计算总结

以上分别详细讲解了 5 种类别的 VSAM 文件。这 5 类文件所在的 VSAM 数据集分别为 LDS、ESDS、RRDS、KSDS 以及 VRRDS。这部分内容构成了本章的主体内容。

现将以上这 5 种 VSAM 文件综合起来进行一个总结。这 5 类 VSAM 文件各自的特点及属性如表 13.3 所示。

表 13.3　　VSAM 文件性质对照总结表

LDS	ESDS	RRDS	KSDS	VRRDS
仅含数据部分	仅含数据部分	仅含数据部分	包含数据部分及逻辑部分	包含数据部分及逻辑部分
由程序划分逻辑记录	定长或变长逻辑记录或 Spanned Record	定长逻辑记录	定长或变长逻辑记录或 Spanned Record	变长逻辑记录
通常通过载入备份数据，不临时新增数据	只能在原数据末尾新增数据	只能在空的 Slot 中新增数据	可在任意位置新增数据	在空白区域新增数据
通常数据不用于修改	可改变逻辑记录的内容，但不可改变其长度	可改变逻辑记录的内容，但不可改变其长度	可改变逻辑记录的内容及长度	可改变逻辑记录的内容，但不可改变其长度
通常数据直接进行整体移除	无法真正删除逻辑记录，但可在逻辑上删除	可以真正删除逻辑记录，并将所占 Slot 置空	可以真正删除逻辑记录，并将所占空间收回	可以真正删除逻辑记录，并将所占空间置空
通常不用于数据访问	可顺序访问或直接访问。直接访问根据 RBA 进行	可顺序访问或直接访问。直接访问根据 RRN 进行	可顺序访问或直接访问。直接访问根据 Key 进行	可顺序访问或直接访问。直接访问根据 RRN 进行

续表

LDS	ESDS	RRDS	KSDS	VRRDS
通常不考虑 Free Space	不可预留 Free Space	不可预留 Free Space	可以预留 Free Space	可以预留 Free Space
不可建立次索引	可以建立次索引	不可建立次索引	可以建立次索引	不可建立次索引
通常不考虑 CI 与 CA 分割	不可进行 CI 与 CA 分割	不可进行 CI 与 CA 分割	可以进行 CI 与 CA 分割	不可进行 CI 与 CA 分割
通常不考虑 RBA 值	不可改变 RBA 值	不可改变 RBA 值	可以改变 RBA 值	不可改变 RBA 值
没有控制信息 RDF 及 CIDF	定长记录时含 2 个 RDF；变长记录时 RDF 与记录个数一致。含一个 CIDF	RDF 与 Slot 个数一致，用于表明 Slot 状态是否为空。含一个 CIDF	定长记录时含 2 个 RDF；变长记录时 RDF 与记录个数一致。含一个 CIDF	RDF 与记录个数一致。含一个 CIDF

在分别学完各类 VSAM 文件的基础上，通过上表可以有一个整体的回顾与总结。通过对比分析表中的各项内容，以巩固和加深对于 VSAM 文件的理解。

对于各类 VSAM 文件空间大小的计算实际上也都是比较类似的。需要注意的是，对于 VSAM 文件空间大小的计算都是针对其数据部分进行的。这点对于包含有索引部分的 KSDS 与 VRRDS 尤其需要注意。下面对其整体计算步骤及相关公式进行归纳，归纳内容如表 13.4 所示。

表 13.4　　VSAM 文件空间大小计算归纳

计算步骤顺序	所求数据	对应公式	
步骤 1	每一 CI 中逻辑记录的数量	LDS	1
		包含定长记录的 KSDS 与 ESDS	每一 CI 中逻辑记录的数量 = (CI 的大小 − CI 的大小 * CI 自由空间比率 − 10) / 逻辑记录的大小 （对于 ESDS 与 RRDS，其 CI 自由空间比率为 0）
		包含变长记录的 KSDS 与 ESDS，以及 RRDS 和 VRRDS	每一 CI 中逻辑记录的数量 = (CI 的大小 − CI 的大小 * CI 自由空间比率 − 4) / (平均逻辑记录的大小 + 3)
步骤 2	所需要的 CI 数量	所需 CI 数量 = 所需逻辑记录数量 / 步骤 1 的计算结果	
步骤 3	每一 CA 中所包含 CI 的数量	每一 CA 中所含 CI 数量 = 每一磁道上的 CI 数量 * 每一 CA 所包含的磁道数量 （每一磁道上的 CI 数量根据 CI 的大小查表得到）	
步骤 4	所需要的 CA 数量	所需 CA 数量 = 步骤 2 的计算结果 / (步骤 3 的计算结果 * (1- CA 中的自由空间)) （对于 ESDS 与 RRDS，其 CA 自由空间比率为 0）	
步骤 5	所需要的柱面或磁道数量	所需柱面或磁道数量 = CA 的大小 * 步骤 4 的计算结果 （最终所得空间大小的单位同给定的 CA 单位一致）	

13.8　通过 COBOL 操作 VSAM 文件

建立在系统学习过 VSAM 文件的基础上，本节将讲解 COBOL 对 VSAM 文件的一些常见操作。VSAM 文件在 COBOL 程序中的应用，在 COBOL 对文件的处理中占有一定的比重。

13.8.1 在程序中指定 VSAM 文件

对于 COBOL 程序而言，在对 VSAM 文件进行处理之前，需要在环境部和数据部中对其进行指定。例如，以下为在 COBOL 程序中指定 VSAM 文件的一段代码。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  VSAM-PROCESS.
 AUTHOR.      XXX.
*
 ENVIRONMENT  DIVISION.
 INPUT-OUTPUT  SECTION.
 FILE-CONTROL.
    SELECT  KS-FILE  ASSIGN  TO  VKSDS
            ORGANIZATION    IS   INDEXED
            ACCESS  MODE   IS   DYNAMIC
            RECORD  KEY    IS   TEST-KEY
            FILE    STATUS   IS   TEST-STAT.
*
 DATA  DIVISION.
 FILE  SECTION.
 FD  KS-FILE
     LABEL  RECORD  IS  STANDARD
     DATA   RECORD  IS  KSRCD.
     COPY   KSRCD.
 ......
*
 PROCEDURE  DIVISION.
      ......
 STOP  RUN.
```

该 VSAM 文件在系统中的名称为 VKSDS，在本程序中的名称为 KS-FILE。对于在环境部中对 KS-FILE 文件所指定的各项参数分别介绍如下。

- ORGANIZATION：用于指定 VSAM 文件的逻辑结构。通常可以为 SEQUENTIAL 或 INDEXED，默认时为 SEQUENTIAL。
- ACCESS MODE：用于指定对该 VSAM 文件的访问模式。通常可以为 SEQUENTIAL、RANDOM 或 DYNAMIC。其中 SEQUENTIAL 对应顺序访问方式，RANDOM 对应直接访问方式，DYNAMIC 既可为顺序访问方式又可为直接访问方式。
- RECORD KEY：用于指定 Key 在程序中的名称。
- FILE STATUS：用于反映 VSAM 文件被读取或写入的状态。

对于 VSAM 文件在数据部中的指定，主要需要注意以下几点。

- 通过 LABEL RECORD IS(ARE)STANDARD 指明该 VSAM 文件是否含有标号(label)。
- 通过 DATA RECORD IS（ARE）定义该 VSAM 文件中逻辑记录在程序中的名称。

通过 COPY 命令将逻辑记录复制到程序工作区域中。

VSAM 文件在数据部中默认 BLOCK CONTAINS 0 RECORDS 一项。

可默认 RECORD CONTAINS n CHARACTERS 一项。

13.8.2 实现对 VSAM 文件的操作

在对 VSAM 文件进行任何处理之前，首先仍然需要打开该文件。并且，为保护文件数据，

在程序结束后也应该关闭该文件。打开和关闭文件仍然是通过 OPEN 和 CLOSE 命令进行的。以下为打开和关闭 VSAM 文件的相应代码。

```
......
 PROCEDURE  DIVISION.
       OPEN  INPUT      KS-FILE01, KS-FILE02
             I-O        RR-FILE
             OUTPUT     ES-FILE.
        ......
       CLOSE  KS-FILE01, KS-FILE02, RR-FILE, ES-FILE.
         STOP  RUN.
```

当对 VSAM 文件中的数据进行读取时，通常有 3 种读取方式。这 3 种方式分别对应于 VSAM 文件境部 ACCESS MODE 中的定义，分别如下。

- 顺序读取方式（Sequential Access）
- 随机读取方式（Random Access）
- 动态读取方式（Dynamic Access）

其中第 3 种方式为前两种方式的综合，因此关键在于前两种方式。对于顺序读取方式，相应代码通常如下。

```
READ  KS-FILE01  NEXT  RECORD
                 AT  END  GO TO 100-TEST-END
END-READ.
```

对于随机读取方式，相应代码通常如下。

```
READ  KS-FILE02  KEY  IS  TEST-RCD-KEY
                 INVALID  KEY  GO TO 200-TEST-ABEND
END-READ.
```

此外，可以通过 WRITE 命令或 REWRITE 命令对 VSAM 文件中的数据进行写入或改写。需要注意的是，WRITE 命令可用于以 OUTPUT 或 I-O 模式打开的 VSAM 文件。而 REWRITE 命令则只能用于以 I-O 模式打开的 VSAM 文件。同时，所写入或改写的名称应该为记录名，并不是文件名。相关代码分别如下。

```
WRITE  KS01-RCD  FROM  TEST-RCD01.
......
REWRITE  RR-RCD  FROM  TEST-RCD02.
```

对于以上两条语句，也都可以在其后加上 INVALID KEY 从句。该从句用以指明当写入或改写失败时进行何种处理。

当需要对 VSAM 文件中的记录进行删除时，可以使用 DELETE 命令。例如，以下为使用 DELETE 命令删除 VSAM 文件中记录的相应代码。

```
DELETE  KS-FILE01  RECORD
         INVALID  KEY  DISPLAY  'DELETE FAILED'.
```

当所处理的 VSAM 文件为索引文件时（如 KSDS），在 COBOL 中还可提供指针定位的功能。提供指针定位功能的相应代码如下。

```
START  KS-FILE02   KEY  <  TEST-VALUE
                   INVALID  KEY
                   DISPLAY  'NO  SUCH  RECORD'.
```

最后，简要介绍一下 VSAM 文件在实际不同类型处理方式中各自需要注意的地方。此处所说的不同类型的处理方式分别是指在线（ONLINE）方式和批处理（BATCH）方式。关于这两种方式下 VSAM 文件的处理特点分别如下。

- ONLINE 方式：此时 VSAM 文件不用在 COBOL 程序中进行打开和关闭。所用到的文件将在 CICS 下打开。同时，该方式下可以实现多终端对文件的操作，而不必等待用户或资源。关于此处提到的 CICS，将在后面章节中进行详细讲解。
- BATCH 方式：VSAM 文件必须在 COBOL 程序中进行打开和关闭。同时，每次只能有一个作业可以对该文件进行操作。其余作业需要等到该作业结束后，才可对其所占用的 VSAM 文件进行操作。

13.9 本章回顾

本章主要讲解了 VSAM 文件的概念及相关特性。VSAM 文件是大型机系统中十分常见的一类文件。无论从事开发还是系统工作，都会经常接触到此类文件。VSAM 文件所在的数据集为 VSAM 数据集。关于 VSAM 数据集，需要清楚该类数据集根据其数据组织结构通常共分为 5 种类型。这 5 种类型的 VSAM 数据集分别为 LDS、ESDS、RRDS、KSDS 以及 VRRDS。

首先，介绍了 VSAM 的基本概念。关于 VSAM 的基本概念，除需要了解其分类外，还应了解 VSAM 数据集中 3 个比较重要的概念。这 3 个概念分别为 CI 的概念、CA 的概念以及 RBA 的概念。

然后，分别对 5 类 VSAM 文件进行了详细讲解。该部分内容为本章的主体内容。学习该部分内容，需要深入理解各类 VSAM 文件的结构及其特征，掌握各类 VSAM 数据集数据部分空间大小的计算方式，理解对于 ESDS、RRDS、KSDS 中数据的访问方式，理解 Spanned Record 技术、CI 及 CA 分割技术、次索引技术各自的特点及适用范围。

在分别讲解完毕各类 VSAM 文件之后，本章对所讲解的 VSAM 文件进行了一个总结。总结包括各类 VSAM 文件的性质以及 VSAM 空间大小的计算。此处的总结主要体现在两张表格上，通过这两张表格有利于从整体上把握本章所讲解的主要内容。

在系统讲解完毕 VSAM 文件的基础之上，简要介绍了 COBOL 对 VSAM 文件的常用操作。这些操作包括在程序的环境部和数据部中对 VSAM 文件进行指定，打开和关闭 VSAM 文件，对 VSAM 文件中的数据进行读取、写入、改写、删除以及对含有索引部分的 VSAM 文件实现指针定位功能。

第 14 章 JCL 扩展

JCL（Job Control Language）即作业控制语言。在大型机中，作业（Job）是用户在完成某任务时要求计算机所做工作的集合。其中也包括了对于 COBOL 程序的编译运行，以及 COBOL 源代码所在数据集的管理等。因此，无论从事大型机哪方面的工作，都是应该掌握 JCL 的。

14.1 基本概念

JCL 是用户与大型机操作系统的接口。用户让操作系统完成的绝大多数任务都是通过 JCL 进行安排的。JCL 实现了用户和操作系统之间的通信，通信所传递的内容是由 JCL 语句指定的。在学习具体的 JCL 语句之前，首先有必要了解一下 JCL 的基本概念。

14.1.1 作业与作业步

前面提到，用户在完成某任务时要求计算机所做工作的集合是通过作业表示的。对于用户而言，JCL 使用户获得作业所需的资源，并按自己的意图控制作业的执行；对于作业而言，JCL 为被执行的任务引导操作系统，并注释所需要的全部 I/O 设备。

JCL 对作业的控制具体而言是通过 JCL 语句来实现的。对于任何一个作业，通常至少都应该包含以下 3 种 JCL 语句。

- JOB 语句：作业语句。该语句标志着一个作业的开始，并提供该作业执行时所必要的一些运行参数。
- EXEC 语句：执行语句。该语句标志着作业中的一个作业步的开始，并指明本作业步执行的程序名或过程名。当该语句在过程中时，标志一个过程步的开始。此处所说的过程是与程序类似的一个概念，只是过程中的参数在调用时是可以重新设置的。
- DD 语句：数据定义语句。该语句用于定义应用程序所需的数据文件。这种数据文件通常为一个数据集，不过有时也可以为其他内容，如显示屏等。

以上这 3 种 JCL 语句也称为 JCL 的基本语句。注意，其中 EXEC 语句是用于标志一个作

业步的开始。关于一个作业步，对应着一个作业中一次程序的执行。一个作业既可由一个作业步组成，也可以多个作业步组成。由一个作业步组成的作业称为单步作业；由多个作业步组成的作业称为多步作业。例如，以下为一个单步作业的例子。

```
//JOB1   JOB …
//STEP1  EXEC …
//DD1    DD …
//
```

以下为一个多步作业的例子，共包含 3 个作业步。

```
//JOB2   JOB …
//STEP1  EXEC …
//DD20   DD …
//STEP2  EXEC …
//DD21   DD …
//DD22   DD …
//STEP3  EXEC …
//DD23   DD …
//
```

一个作业中的各个作业步是按顺序依次执行的。因此，上一个作业步的输出可以作为下一个作业步的输入。并且，有的时候某一作业步是否执行也是根据上一作业步的执行情况而定的。

此外，在系统内部，一个作业的具体执行通常是由 3 个相对独立的部分顺次组成的。这 3 个部分的内容分别如下。

- 编译：将源程序语句（也称作源模块）转换为目标模块。
- 连接：把目标模块同子程序库中的其他程序连接得到可执行模块。
- 运行：运行可执行模块，得到最终结果。

14.1.2 JCL 语法规则

JCL 有一套严格的语法规则。用户必须按照其语法规则编写相应的 JCL，否则系统将会产生错误信息，甚至产生不可预知的后果。

在 JCL 语句中，通常只允许出现系统所指定的字符，其他字符将不被系统识别。这些特定的字符称作 JCL 字符集，JCL 字符集包含以下字符。

- 26 个英文字母：A ~ Z。
- 10 个阿拉伯数字：0 ~ 9。
- 6 个关系字符：GT、GE、LT、LE、EQ、NE。
- 3 个通配符：@、$、#。
- 10 个特殊字符：“,”、“.”、“/”、“ (”、“)”、“*”、“&”、“+”、“–”、“=”。

JCL 语句除/*语句外均以第一、二列的“//”符号作为开始标志。实际上，凡是在首两列以“//”符号开始的语句通常都是 JCL 语句。系统规定每行的长度为 80 列，当某一 JCL 语句需要超过 80 列时，可以通过续行完成。逻辑上将这 80 列划分为 5 个区域，这 5 个区域分别如下。

- 标志符区：用于标志 JCL 语句。通常情况下，该区域的符号为“//”，位于语句的首两列。

在某些特殊情况下，该区域的符号也可以为“/*”或“//*”。

- 名字区：用于对一条语句进行命名，以便于系统控制块或其他语句识别。名字区后必须跟一个或多个空格，用于将名字和其他操作符区分开来。其中语句的名字可以由 1～8 个字母、数字以及通配符组成。但名字的第一个字符必须为字母或通配符，并且必须从第三列开始。通常应该选择比较有意义的名字，如选择 STEPn（n 为自然数）命名 EXEC 语句等。
- 操作符区：用于指定语句的类型。这些语句的类型通常使用一些操作符指定，即 JOB、EXEC、DD、PROC、PEND 或操作员命令。
- 参数区：用于指定语句中所要用到的参数。这些参数既包括位置参数，也包括关键字参数。关于 JCL 语句中的参数，将在后面的小节中详细讲解。
- 注释区：用于对相应语句进行注释说明，可以为任何注释信息。通常仅当语句中含有参数时，才能书写注释信息，否则容易与参数相混淆。注释区后必须跟一个空格。

为更好地体现 JCL 语句中各个区域的划分，下面结合具体的 JCL 语句进行说明。首先，给出一条示例 JCL 语句。

```
//EXAMPLE  JOB  , TOM, CLASS=S              THIS IS A COMMENT
```

对于以上示例 JCL 语句，其各个区域的内容分别如表 14.1 所示。

表 14.1　　JCL 语句各区域内容

标志符区	//	参数区	, TOM, CLASS=S
名字区	EXAMPLE	注释区	THIS IS A COMMENT
操作符区	JOB		

关于 JCL 语句中的续行，有以下要求。

- JCL 只允许在参数区和注释区有续行。
- 当前行的第 71 列之前的参数或子参数，以及参数后的逗号必须是完整的。
- 新一行的首两列必须为“//”符号，第三列为空格。续行内容只能从 4～16 列开始。当续行内容超过 16 列时，将会被认为是注释内容，而非 JCL 语句的续行部分。

例如，以下为一个续行的例子。

```
//DD1    DD   DSNAME = ADCDA.MYDATA,
//  DISP = (NEW, ,KEEP),
//  UNIT = 3390, VOL=SER=WORK01,
//  DCB = (RECFM=FB, LRECL=80, BLKSIZE=4000),
//  SPACE = (TRK , (10,5))
```

最后，通过对比几条合法的 JCL 语句与非法的 JCL 语句，以加深对 JCL 语法规则的熟悉。首先来看几条非法的 JCL 语句。

```
  //TEST    JOB  2008, CLASS=A            /*标志符区中的//符号没有顶格写*/
//1STEP   EXEC  PGM=IEBGENER              /*语句名字首字符不能为数字*/
//SYSPRINT01  DD  SYSOUT = *              /*名字超过 8 个字符*/
//SYSINDD    DUMMY                        /*名字后没有空格*/
//DD*     DD    DSN=TEST.DATA, DISP=SHR   /*名字中包含非法字符*/
//PTEST   PROC     COMMENT                /*没有参数时不能书写注释信息*/
//DD1     DD    DSN=TEST.DATA01           /*续行时当前行最后的参数缺少一个逗号*/
//               DISP=SHR                 /*续行时新行不在 4～16 列*/
```

关于不遵守 JCL 语法规则的情况还有很多，以上只是列举了其中的几种。下面再来看几条合法的 JCL 语句，在对比的同时，加强正确的印象。

```
//TEST    JOB  2008, CLASS=A                FOR TEST
//*  THESE ARE CORRECT JCL STATEMENTS
//STEP1   EXEC  PGM=IEBGENER
//SYSPRINT  DD  SYSOUT = *
//SYSIN   DD    DUMMY
//DD1     DD    DSN=TEST.DATA, DISP=SHR
//PTEST   PROC
//STEP2   EXEC PGM=MYPGM
//SYSIN    DD  *
          DSN = MYDATA
/*
//  PEND
//  PEND
```

14.1.3 JCL 语句类型

JCL 是通过语句实现对作业的控制的。一个使用 JCL 编写的作业由多条 JCL 语句所组成。其中每条 JCL 语句既可占一行，也可占多行。JCL 语句没有特定的结束标志，下一条语句的开始就象征着本条语句的结束。

JCL 语句除前面讲解的 3 种基本语句：JOB 语句、EXEC 语句和 DD 语句外，还包含 6 种附加语句。这 6 种附加语句分别如下。

- //语句：当某一 JCL 语句仅有标志符区中的“//”符号时，表示该语句为空语句。空语句常用来标志一个完整作业的结束。
- /*语句：当标志符区中的前两列字符为“/*”时，表示该语句对应两种情况之一。其中第一种情况用来表示流内数据的结束，流内数据通常由 DD *或 DD DATA 引导。第二种情况用来调用 JES 控制语句，JES 控制语句是相对于 JCL 的另一种作业控制语句。
- //*语句：当标志符区中的前三列字符为“//*”时，表示该语句为注释语句。注释语句中的内容可以为任何注释信息。
- PROC 语句：当操作符区中的字符为 PROC 时，表示一个过程的开始。该过程既可以为流内过程（In-stream Procedure），也可以为编目过程（Cataloged Procedure）。
- PEND 语句：当操作符区中的字符为 PEND 时，表示一个流内过程的结束。
- Command 语句：该语句中的内容为操作员在输入流中写入的操作命令。这些命令包括 SORT、MERGE、REPRO 等。

由于一条 JCL 语句可以占一到多行，也没有特定的结束标志，因此需要仔细地对其进行划分。正确地划分 JCL 语句是读懂 JCL 代码的前提。

14.1.4 参数的类别及书写规则

JCL 语句中通常都包含有众多的参数，各参数之间是通过逗号隔开的。实际上，学习 JCL 很大程度上就在于了解 JCL 语句中各项参数的意义及用法。JCL 语句中的参数根据其性质通常可以分为两大类别，分别如下。

- 位置参数：参数的意义与其位置有关。
- 关键字参数：参数由关键字和等号后面的数据组成，其意义与位置无关。

若要判断一个参数究竟是位置参数还是关键字参数，主要可以通过观察该参数中是否含有等号。通常情况下，不含等号的就是位置参数，含有等号的就是关键字参数（少数情况也有例外）。例如，对于以下这条 JCL 语句。

```
//JPARM  JOB  2008, ROBIN, CLASS=A, MSGCLASS=M, MSGLEVEL=(1,0)
```

该语句中的位置参数分别如下。

- 2008
- ROBIN

该语句中的关键字参数分别如下。

- CLASS=A
- MSGCLASS=M
- MSGLEVEL=(1,0)

当某一语句中既有位置参数又有关键字参数时，通常关键字参数都应位于位置参数之后。例如，以上语句中的 3 个关键字参数就均位于位置参数之后。

此外需要注意的是，某些位置参数和关键字参数还是可以包含子参数的。子参数写在括号之内。例如，以下就为几条包含有子参数的 JCL 语句。

```
//TEST1  JOB  (A254-8874, '14/02/08', ITD)
//TEST2  JOB  MSGLEVEL=(0,1)
//TEST3  DD   DSN=TEST.DATA,
//   DISP=(NEW,CATLG),
//   DCB=(RECFM=FB, LRECL=120, BLKSIZE=480, DSORG=PS),
//   SPACE=(TRK, (2, 1)),
//   UNIT=3390, VOL=SER=WORK04
```

第一条 JOB 语句 TEST1，包含了一个位置参数。这一位置参数由 3 个子参数所组成。并且，这 3 个子参数也都为位置参数。因此，这 3 个子参数的书写也是有顺序要求的。

第二条 JOB 语句 TEST2，包含了一个关键字参数。该关键字参数的关键字为 MSGLEVEL。并且，由于该参数中等号后的数据是以括号表示的，因此可以将其视作该关键字参数的子参数。并且，由于这些子参数中没有等号，因此这些子参数为位置参数。位置参数根据不同的位置而有不同的意义，因此下面这两条语句中的参数意义是不同的。

```
//EXP01  JOB  , MSGLEVEL=(0,1)
//EXP02  JOB  , MSGLEVEL=(1,0)
```

以上语句中的第三条为 DD 语句。该语句中共包含了 4 个关键字参数。其中第 2 个关键字参数中包含了两个位置子参数，第 3 个关键字参数中包含了 4 个关键字子参数，第 4 个关键字参数中包含了两个位置子参数，并且第二个位置子参数又包含了两个位置子参数。

由于关键字参数的意义与其位置无关，因此，对于关键字参数的书写顺序是不做要求的。即使该关键字参数为一子参数，其书写顺序仍然是任意的。因此，对于以上 DD 语句，以下这种写法也是等效的，该写法如下。

```
//TEST3  DD   DSN=TEST.DATA,
//   DISP=(NEW,CATLG),
```

```
//   SPACE=(TRK, (2, 1)), UNIT=3390, VOL=SER=WORK04,
//   DCB=(DSORG=PS, LRECL=120, BLKSIZE=480, RECFM=FB)
```

无论是位置参数还是关键字参数，其子参数做多只能有两级。即用以表示子参数的括号最多只能有两层。

由于位置参数与其位置息息相关，因此当缺省某一位置参数时，必须用逗号指明其所在位置。这样做的目的是防止因某一位置参数的默认，而影响其他位置参数的位置。例如，以下几条语句中就默认了部分位置参数，并用逗号指明了默认的位置。

```
//EXP03  JOB  (A254-8874, , ITD),AUTHOR
//EXP04  JOB  ,AUTHOR
//EXP05  JOB  MSGLEVEL=( ,0)
//EXP06  DD   DSN=TEST.DATA, DISP=(, CATLG, DELETE),
//       UNIT=SYSDA, VOL=SER=WORK04
//EXP07  DD   DSN=TEST.DATA, DISP=(, , CATLG),
//       UNIT=SYSDA, VOL=SER=WORK06
```

当默认的位置参数为最后一个参数时，由于不会影响到其他位置参数的位置，因此可以省略逗号。当语句中没有任何位置参数时，不必标明任何提示内容，直接在其后书写关键字参数。例如，以下几条语句就默认了最后一个位置参数或所有的位置参数。

```
//EXP08  JOB  (A254-8874, '14/02/08' )
//EXP09  JOB  A254-8874
//EXP10  JOB  CLASS=A
//EXP11  DD   DSN=TEST.DATA, DISP=SHR
```

最后需要注意的是，参数中是可以存在 JCL 字符集中的特殊字符的。当特殊字符并不起到某种特定的语法功能，只是作为一个单纯的字符时，必须用撇号将该参数括起来。例如，以下几条语句中的部分参数就包含了特殊字符，并使用撇号括了起来。

```
//EXP12  JOB  (A254-8874, '14/02/08' )
//EXP13  JOB  ,'SIMON&JASON'
//EXP14  EXEC PGM=MYPGM, ACCT='12.25+22.26'
```

14.2 JOB 语句

JOB 语句用于表明一个作业的开始，任何一个作业的第一条语句必须为 JOB 语句。同时，JOB 语句还为作业分配作业名并设置相应的各项参数。这些参数既包括位置参数，也包括关键字参数。下面主要就该语句中这两种不同类别的参数分别进行讲解。

14.2.1 JOB 语句中的位置参数

在讲解 JOB 语句的位置参数之前，先关注一下 JOB 语句的语句名。JOB 语句的语句名即为所标明作业的作业名。作业名作为 JCL 语句名字区中的内容，必须遵循该区域中的语法规则。此处的语法规则主要体现在以下两点。

- 作业名的第一个字符必须为字母或通配符。
- 作业名的长度不能超过 8 个字符。

此外，由于系统不能同时运行同名的作业，因此每一个作业都应指定为唯一的名字。通

常情况下，作业名使用用户的账号加上数字或字母组成。这样做的目的是便于进行统一管理，同时方便作业提交后在 SDSF 中观察其执行的情况。

关于 JOB 语句中的位置参数，主要有两个。其中一个参数为记账信息，另一个参数为程序员名。记账信息参数位于操作符 JOB 之后，程序员名参数位于记账信息参数之后。

1．记账信息参数

记账信息参数主要用于记录和作业相关的一些信息。其中包括用户账号、时间、部门号、房间号等。记账信息参数可以包含多个子参数，每一子参数用来提供以上某一条信息。记账信息参数及其子参数最多不得超过 143 个字符。这些字符包括各子参数之间的逗号，但不包括将子参数括起来的括号。例如，以下为几条包含记账信息参数的 JOB 语句。

```
//TEST01  JOB  (B867-4092, '15/3/08',BSS)
//TEST02  JOB  B867-4092
//TEST03  JOB  (2008, ,2)
```

2．程序员名参数

程序员名参数用于标明该作业的所有者信息。该信息既可为该作业所有者的真实姓名，也可为其编号，内容不受限制。程序员名参数最多不得超过 20 个字符。并且这些字符中包括 JCL 字符集中的特殊字符在内。例如，以下为几条包含程序员名参数的 JOB 语句。

```
//TEST10  JOB  ,ERIC
//TEST20  JOB  ,ST262
//TEST30  JOB  ,A.B.C
```

需要注意的是，由于记账信息参数为第一个位置参数，而程序员名为最后一个位置参数。因此，当默认程序员名参数时，不必在记账信息参数后写上逗号。而当默认记账信息参数时，则必须在程序员名参数前写上逗号。

根据 JOB 语句中的位置参数，共有 4 种不同书写方式的 JOB 语句。这 4 种不同书写方式的 JOB 语句分别如下。

- 同时包含记账信息参数和程序员名参数的 JOB 语句，如下所示。

```
//TEST11  JOB   2008,ERIC
```

默认记账信息参数的 JOB 语句，如下所示。

```
//TEST22  JOB   ,ERIC
```

默程认序员名参数的 JOB 语句，如下所示。

```
//TEST11  JOB   2008
```

- 不含任何位置参数的 JOB 语句，如下所示。

```
//TEST11  JOB
```

14.2.2 JOB 语句中的关键字参数

JOB 语句中的关键字参数有很多。这些关键字参数分别用于指定作业的类别、存储空间、运行时间、运行方式等。以下就 JOB 语句中几个常用的关键字参数分别进行讲解。

1．CLASS 参数和 MSGCLASS 参数

CLASS 参数用于指定作业的类别。相同类别的作业将具有相同的处理属性，并在 JES 子系统中处于同一输入队列等待执行。作业的类别由一个字符所表示，该字符既可以为 A～Z 这 26 个字母，也可以为 0～9 这 10 个阿拉伯数字。例如，以下为几条包含 CLASS 参数的 JOB 语句。

```
//TEST01  JOB  2008, ERIC, CLASS=A
//TEST02  JOB  ,ERIC, CLASS=9
//TEST03  JOB  CLASS=G
```

当 CLASS 参数默认时，将由 JES 子系统为作业指定一个默认的 CLASS 值。该默认的 CLASS 值是在系统安装时所定义的。

MSGCLASS 参数与 CLASS 参数类似，其数据也是 26 个字母或 10 个数字中的一个字符。不过，该参数指定的是作业日志的输出类别，而非作业的处理类别。作业日志主要是在作业提交后产生的作业执行情况相关信息。当 MSGCLASS 参数默认时，系统也会为其指定一个默认的值。例如，以下为几条包含 MSGCLASS 参数的 JOB 语句。

```
//TEST10  JOB  ,ERIC, MSGCLASS=9
//TEST20  JOB  CLASS=A, MSGCLASS=9
//TEST30  JOB  MSGCLASS=A, CLASS=A
```

2．MSGLEVEL 参数

MSGLEVEL 参数用于指定作业清单输出的内容及方式。该参数中包含有两个位置子参数，其中第一个子参数主要用于控制作业清单输出的内容，取值内容及对应信息如下。

- 取值为 0：表示仅输出作业中的 JOB 语句。
- 取值为 1：表示输出作业中的所有 JCL 语句。
- 取值为 2：表示输出作业中包含输入流中控制语句在内的所有语句。

第二个子参数主要用于控制作业清单输出的方式。该子参数的取值内容及对应信息如下。

- 取值为 0：表示只有在作业异常终止时，才输出相关信息。这些信息包括 JCL、JES、操作员以及 SMS 的各种处理信息。
- 取值为 1：表示无论作业是否异常终止，都输出以上相关信息。

需要注意的是，由于该关键字参数中的这两个子参数为位置参数，因此必须按照顺序书写。同时，各子参数的取值内容也必须在规定的内容之内。当第一个参数默认时，必须用逗号表示；当第二个参数默认时，不必写逗号，并且可去掉括号。例如，以下为几条包含 MSGLEVEL 参数的 JOB 语句。

```
//TEST10  JOB  MSGLEVEL=(0,1)
//TEST20  JOB  CLASS=A, MSGLEVEL=(,0)
//TEST30  JOB  MSGLEVEL=2
```

此外需要特别注意的是，切不可将这两个参数的取值范围弄混淆了。例如，以下这种写法就混淆了二者的取值范围，是错误的。

```
//TEST10  JOB  MSGLEVEL=(1,2)          /*第二个子参数只能取值为 0 或 1，不能取值为 2*/
```

最后需要说明的是，MSGLEVEL 参数中的两个位置子参数默认值都为 1。当这两个参数

默认时，系统将按照其默认值进行处理。

3．ADDRSPC 参数和 REGION 参数

ADDRSPC 参数用于指定作业所需的存储类型。该参数的数据有两种取值。这两种取值及其表示的信息分别如下。

- 取值为 VIRT：表示作业请求的存储类型为虚拟页式存储。
- 取值为 REAL：表示作业请求的是实存空间。

该参数的默认值为 VIRT。即当该参数默认时，系统会认为该作业请求的存储类型是虚拟页式存储。以下为两条包含 ADDRSPC 参数的 JOB 语句。

```
//TEST01  JOB  ADDRSPC=VIRT
//TEST02  JOB  ,BOB,ADDRSPC=REAL
```

REGION 参数参数则用于指定作业所需的虚存或实存的空间大小。该空间的大小通常用两种单位进行表示，一种为 K（KB），另一种为 M（MB）。

系统是以每 4KB 为一个存储单元分配空间的。因此，对于以 K 为单位的表示方式，前面的数值应该为 4 的倍数。当其不为 4 的倍数时，系统会将其增至为一最接近的 4 的倍数的值。例如，当指定该参数为 14KB 时，系统会将其视作 16KB 进行处理。对于以 M 为单位的表示方式，当系统未定义具体的上限值时，其空间大小最大不能超过 16MB。

以下为几条包含 REGION 参数的 JOB 语句。

```
//TEST10  JOB  REGION=16K
//TEST20  JOB  123,BOB,
//   ADDRSPC=REAL,
//   REGION=280K
//TEST30  JOB  , ANDY, CLASS=A,REGION=2M
```

在作业处理过程中，系统会在该作业的每一作业步中用到 REGION 参数所指定的值。当 JOB 语句中的 REGION 参数默认时，系统将采用每条 EXEC 语句中定义的 REGION 参数。当 EXEC 语句中的 REGION 参数也默认时，系统将会采用安装时的默认值。

REGION 参数可以默认，但却不能指定为 0。同时，该参数也不可以被指定为任何大于系统极限值的数值。否则，以上两种做法都将引发存储问题。

总之，ADDRSPC 参数和 REGION 参数都是和作业的存储有关的。其中 ADDRSPC 参数用以指定存储的类型，而 REGION 参数则用以指定存储空间的大小。

4．TIME 参数

TIME 参数用于指定作业的最长运行时间。当作业运行时间超过该参数所指定的值时，系统将会终止该作业。TIME 参数的数据有 4 种类型，分别如下。

- （minutes, seconds）：两个位置子参数。第一个子参数表示分钟数，第二个子参数表示秒数。
- MAXIMUM：该数据表示作业的运行时间为 357912 分钟。
- 1440：1440 相当于分钟数，将其转换为小时数即 24 小时。因此，该数据表示的意义是作业的运行无时间限制。
- NOLIMIT：该数据同样表示作业的运行是无时间限制的，与 1440 数据是等效的。

以下为几条包含 TIME 参数的 JOB 语句。

```
//TEST01  JOB  ,WENDY,TIME=(8, 10)
//TEST02  JOB  ACCT428, TIME=(, 40)
//TEST03  JOB  , JESSICA, TIME=5
//TEST04  JOB  , TRACY, TIME=NOLIMIT
```

TIME 参数默认时的处理情况同 REGION 参数类似。当 JOB 语句中的 TIME 参数默认时，系统将采用每条 EXEC 语句中定义的 TIME 参数。当 EXEC 语句中的 TIME 参数也默认时，系统将会采用 JES 默认的作业步时间限制值。

此外需要注意的是，JOB 语句中的 TIME 参数指定的是整个作业的最长运行时间。因此，对于作业中的各作业步来说，既要满足各自的运行时间，其总和又要满足整个作业的运行时间。例如，以下为一个包含有两个作业步的作业。

```
//JOBTIM  JOB  ,TINNA,TIME=5
……
//STEP1  EXEC  PGM=MYPGM1, TIME=3
……
//STEP2  EXEC  PGM=MYPGM2, TIME=3
……
```

对于以上作业，整个作业的运行时间被限定为 5 分钟。同时，该作业中每一作业步的运行时间被限定为 3 分钟。任何一个作业步的运行时间超过 3 分钟，该作业都将异常终止。同时注意到，整个作业的运行时间最长为 5 分钟。因此，若第一个作业步运行时间为 2.58 分钟，则第二个作业步的运行时间将不能超过 2.42 分钟。否则，该作业同样会异常终止。

5．NOTIFY 参数、TPYRUN 参数和 PRTY 参数

以上 3 种关键字参数也是在 JOB 语句中经常会用到的。特别是 NOTIFY 参数，在平常用于练习的 JOB 中都应包含该参数。这 3 种参数表示的意义分别介绍如下。

NOTIFY 参数：用于在作业运行完毕后发送返回信息给指定用户，其数据为一个 TSO 用户账号。如果该用户当前没有登录系统，则将会在下次登录时收到系统发送的返回信息。通常可使用&SYSUID 数据表示将返回信息发送给自身用户。

- TPYRUN 参数：主要用于对所写的 JCL 进行语法检测。通常其数据指定为 SCAN，此时提交作业后将不运行该作业，而只是对其进行语法检测。当不确定所写的 JCL 是否正确，而又不能影响系统的正常执行时，通常使用该参数进行调试。
- PRTY 参数：该参数用于指定作业的优先级。优先级使用数字表示，数字越大，优先级越高。优先级的取值范围为 0～15。

以下为几条包含有以上 3 个参数的 JOB 语句。

```
//TEST01  JOB  2008, NOTIFY=ADCDA
//TEST02  JOB  2009, NOTIFY=&SYSUID
//TEST03  JOB  2010, NOTIFY=&SYSUID, TPYRUN=SCAN
//TEST04  JOB  2020, PRTY=10, NOTIFY=&SYSUID, TPYRUN=SCAN
```

最后需要补充说明的是，关于通过 NOTIFY 参数得到的返回信息实际上概括了作业的运行情况。在该返回信息中，重点需要观察其中 MAXCC 的值。当 MAXCC 的值为 0 时，表示作业运行正常。此外，通过该条返回信息还可推断出作业运行异常的部分原因，如作业所在数据集空间是否足够等。

14.2.3 JOB 语句中参数的综合应用

前面分别讲解了 JOB 语句中的位置参数和关键字参数。其中位置参数仅有两个，分别是记账信息参数和程序员名参数，而关键字相对要多得多。除以上讲解的几个关键字参数外，JOB 语句中还存在如下一些关键字参数。

- BYTES
- PASSWORD
- COND
- GROUP
- PERFORM
- RESTART
- USER
- RD
- SECLABEL

以上参数在实际中用得并不算多，此处不再进行详细讲解。同时需要注意的是，JOB 语句中的关键字参数并不止这些。对于其他一些参数，可以在具体的工作中再进一步学习。

下面根据前面所讲解的各项参数，给出一条对作业中的各项内容指定较为详细的 JOB 语句。该语句实现了各项参数的综合应用，语句如下。

```
//SAMPLE0  JOB  (ST226, '15/8/08', HUST), MARK.XU,
//   CLASS=A, MSGCLASS=X,
//   MSGLEVEL=(1,1),
//   ADDRSPC=VIRT, REGION=240K,
//   TIME=(,30),
//   PRTY=6,
//   NOTIFY=&SYSUID
```

当该作业在正式运行之前需要对其进行 JCL 语法检测时，可以在该语句中加上 TPYRUN=SCAN 参数。以上这条 JOB 语句对作业中各项内容的指定是十分详细的。通常情况下，在实际编写的 JCL 中往往默认了以上语句中的部分参数。

在编写一系列的 JCL 作业时，往往有一些作业的设置是相同的。如作业类别都为 A，返回信息都发送给&SYSUID 等。同时，还有一些作业之间虽然参数的具体数据不同，但参数的种类和个数却是一致的。如都包含有 CLASS 参数，MSGCLASS 参数和 NOTIFY 参数等。

对于以上所说的情况，可以认为各作业中 JOB 语句的结构是相同的。未避免重复书写相同结构的 JOB 语句，可以将该 JOB 语句单独列出来。通常将相关 JCL 作业存放在一个后缀名为 CNTL 的 PDS 分区数据集中。而单独列出来的 JOB 语句同时也作为该 PDS 中的一个成员，其成员名通常为 JOBCARD。当编写新的 JCL 作业时，可以直接将 JOBCARD 复制为该作业的 JOB 语句。

14.3 EXEC 语句

EXEC 语句用于表明作业或过程中一个作业步的开始，同时为该作业步设置相关参数。分析一段 JCL 代码的功能，实际上就是分析其中每一作业步的功能。下面同 JOB 语句类似，

依然根据 EXEC 语句中不同的参数类型分别进行讲解。此外，该语句中还有一个比较复杂的关键字参数——COND 参数。对于该参数将单独用一个小节进行讲解。

14.3.1 EXEC 语句中的位置参数

在讲解 EXEC 语句中的位置参数之前，先大致了解一下关于 EXEC 语句本身的一些注意事项。这些注意事项归纳起来主要有以下两条。

- 一个作业中最多只能有 255 个作业步。并且，这些作业步还包含在本作业中通过 EXEC 语句调用的过程中的所有作业步。即一个作业中最多只能含有 255 条 EXEC 语句。并且，其中还包括在该作业中所调用的过程中的 EXEC 语句。
- 作业步名必须在该作业，及该作业所调用的过程中是唯一的。根据 JCL 语句名字区中的规定，作业步名也是由字母或通配符开头的 1～8 个字符和数字所组成。此外，作业步名是可以默认的。

EXEC 语句中的位置参数有两个，分别为 PGM 参数和 PROC 参数。这两个位置参数相对其他位置参数比较特殊，因为这两个位置参数是带等号的。每条 EXEC 语句中必须有一个位置参数，并且仅能有一个位置参数。

1．PGM 参数

作业步是通过调用程序或过程以实现其具体功能的。PGM 位置参数便用于指定该作业步所调用的程序名。PGM 参数数据的表示方法通常有以下 3 种。

- PGM=program-name：直接指定程序名 program-name，实现直接调用方式。其中程序名是由 1～8 个以字母，或通配符开头的字符和数字组成。
- PGM=*.stepname.ddname：调用的程序在本作业步之前名为 stepname 的作业步中指定。stepname 作业步中名为 ddname 的 DD 语句中的 DSN 参数具体决定了本作业步调用的程序。这种情况下实现的是间接调用。
- PGM=*.stepname.procstepname.ddname：调用的程序在本作业步之前名为 stepname 的作业步所调用的过程中指定。该过程的过程步 procstepname 中的 DD 语句 ddname 的 DSN 参数决定了该程序。这种情况下实现的也属于间接调用。

例如，以下为一段包含有多个作业步，以及一个流内过程的作业。

```
//JOB01  JOB  CLASS=A
//STEP1  EXEC  PGM=MYPGM
//DD1    DD    DSN=ADCD.LINKLIB(PGM2),
//     DISP=OLD
//STEP2  EXEC  PGM=*.STEP1.DD1
//PROC1  PROC
//JSTEP1  EXEC  PGM=PCPGM
//JDD1    DD    DSN=SAMPLE.LINKLIB(P03), DISP=SHR
//  PEND
//PSTEP   EXEC  PROC=PROC1
//STEP3   EXEC  PGM=*.PSTEP.JSTEP1.JDD1
//
```

对于以上作业步 STEP1，其调用的是名为 MYPGM 的程序，进行的是直接调用。作业步 STEP2 调用的程序由作业步 STEP1 中名为 DD1 的 DD 语句决定。该 DD 语句中的 DSN 参数

指定了一个分区数据集 ADCD.LINKLIB 中的成员 PGM2。该成员即为作业步 STEP2 所调用的程序，进行的是间接调用。同样，作业步 STEP3 调用的程序为分区数据集 SAMPLE.LINKLIB 中的成员 P03，也是间接调用。

通过 PGM 参数所指定的程序通常只能为以下几种情况，分别如下。

- 分区数据集（PDS）或扩充分区数据集（PDSE）的成员。
- 系统库的成员。
- 私有库的成员。
- 临时库的成员。

2．PROC 参数

EXEC 语句中的位置参数 PROC 用来指定该作业步所调用的过程名。过程名同程序名一样，也是由 1～8 个以字母或通配符开头的字符和数字组成。

PROC 参数所指定调用的过程大体上可以分为两种类型。一种为编目过程，此时 PROC 参数的数据可为该编目过程的成员名或别名。另一种为通过 PROC 语句定义的流内过程，PROC 参数的数据为该过程的过程名。

需要注意的是，当 PROC 参数指定的过程为流内过程时，该过程必须在本作业步之前进行定义。例如，以下为一段调用流内过程的代码。

```
//JOB01  JOB  CLASS=A
//INPROC  PROC                          /*以下为一段流内过程*/
//JSTEP1  EXEC  PGM=INPGM
//JDD1    DD    DSN=SAMPLE.PS2
//  PEND                                /*该流内过程在此定义结束*/
//STEP1   EXEC  PROC=INPROC             /*本作业步调用该流内过程*/
//
```

此外，PROC 参数还是可以默认的。当 PROC 参数默认时，只用给出一个该参数调用的过程名。因此，以下两条 EXEC 语句是等效的。

```
//STEP01  EXEC  PROC=PTEST
//STEP02  EXEC  PTEST
```

总之，PGM 参数用于调用程序，而 PROC 参数用于调用过程。这两个参数都为 EXEC 语句中的位置参数。

14.3.2　EXEC 语句中的关键字参数

EXEC 语句中的部分关键字参数同 JOB 语句中的类似，如 REGION 参数、ADDRSPC 参数等。不过 JOB 语句中的参数是针对整个作业而言，而 EXEC 语句中的参数都是针对该作业步而言的。EXEC 语句中的关键字参数包含一个名为 COND 的参数。该参数相对较为复杂，将在下一小节中单独讲解。

1．ADDRSPC 参数和 REGION 参数

首先来看 EXEC 语句中的关键字参数 ADDRSPC 和 REGION。这两个参数实际上和在 JOB 语句中的用法是一致的。不过此时二者都是指具体某一作业步的存储空间类型和空间大

小的。以下为几条包含这两个参数的 EXEC 语句。

```
//STEP01  EXEC  PGM=PGM1, ADDRSPC=REAL
//STEP02  EXEC  PROC=PROC1, REGION=10K
//STEP03  EXEC  PGM=PGM2,
//   ADDRSPC=VIRT, REGION=50K
```

2．TIME 参数

关于 EXEC 语句中的关键字参数 TIME，其用法也同在 JOB 语句中的大体相同。不过在 JOB 语句中该参数的数据是不可为 0 的。而在 EXEC 语句中，该参数数据可以为 0。当某一作业步的 TIME 参数为 0 时，表示该作业步执行的最大时间为前面作业步执行后的剩余时间。例如，以下为一个包含有 3 个作业步的作业。

```
//JOBTIM  JOB  ,TINNA,TIME=6
......
//STEP1  EXEC  PGM=MYPGM1, TIME=2
......
//STEP2  EXEC  PGM=MYPGM2, TIME=3
......
//STEP3  EXEC  PGM=MYPGM3, TIME=0
......
```

以上第 3 个作业步 STEP3 的 TIME 参数被设置为 0。注意到整个 JOB 的 TIME 参数值为 6。因此，当 STEP1 执行时间为 1.5 分钟，STEP2 执行时间为 2 分钟时，则 STEP3 执行的时间最多将为 2.5 分钟。此时相当于将该作业步中的 TIME 参数值设置为 2.5。

3．ACCT 参数

EXEC 语句中的记账信息是通过关键字参数 ACCT 表示的。这点和 JOB 语句中通过位置参数表示记账信息是不同的。ACCT 参数包括其子参数在内最多不能超过 142 个字符。这些字符包括子参数之间用于分割的逗号，但不包括子参数列表的括号。例如，以下为几条包含 ACCT 参数的 EXEC 语句。

```
//S1  EXEC  PGM=IEFBR14, ACCT=123456
//S2  EXEC  PGM=IEBCOPY, ACCT='123+456'
//S3  EXEC  PROC02, ACCT=(12, 34, 56)
```

4．PARM 参数

除了 ACCT 参数以外，在 EXEC 语句中还存在一个相对 JOB 语句比较特殊的参数，即 PARM 参数。该参数用于向本作业步所调用的程序传递相关的变量信息。同时，本作业步所调用的程序中也必须有相应的语句用于接收并使用这些信息。

PARM 参数包括所有字符在内的总长度不得超过 100 个字符。对于 PARM 参数中包含有特殊字符的子参数，需要用撇号将其括起来。例如，以下为几条包含 PARM 参数的 EXEC 语句。

```
//SP1  EXEC  PGM=P01, PARM=ABCDE
//SP2  EXEC  PGM=P02, PARM=(AB, CDE)
//SP3  EXEC  PGM=P03, PARM=('A+B', CDE)
//SP4  EXEC  PGM=P04, PARM='A+B, C=DE'
```

5．关键字参数的覆盖

最后说说关于 EXEC 语句中关键字参数的覆盖问题。当 EXEC 语句的位置参数为 PGM 时，该语句中的关键字将不存在覆盖问题。但如果 EXEC 语句中的位置参数为 PROC 时，则会产生覆盖现象。此时，该语句中的关键字参数将覆盖所调用过程中的各条 EXEC 语句中的关键字参数。例如，以下为一条 EXEC 语句。

```
//STEP0  EXEC  PROC=MYPROC, ACCT=2008
```

该条 EXEC 调用了一个名为 MYPROC 的过程。并且，该过程中每条 EXEC 语句的 ACCT 参数的值将被覆盖为 2008。

设该过程的 JCL 代码如下。

```
//MYPROC  PROC
……
//STEP1  EXEC  PGM=MYPGM1, ACCT=2007
……
//STEP2  EXEC  PGM=MYPGM2, ACCT=2007
……
//STEP3  EXEC  PGM=MYPGM3, ACCT=2009
……
```

此时在调用中也可仅将该过程中 STEP2 中的 ACCT 参数值覆盖为 2008。实现这一方式可通过以下这条 EXEC 语句对该过程进行调用。

```
//STEP0  EXEC  PROC=MYPROC,
//   ACCT.STEP2=2008
```

在使用 EXEC 语句调用过程时，经常会涉及到类似于上面的对某些具体过程步中的参数进行覆盖。需要注意的是覆盖的参数关键字是写在前面的，而该参数所在的过程步名则是写在后面的。

14.3.3 COND 参数

COND 参数是 EXEC 语句中一个比较特殊的关键字参数。该参数不仅涉及到本作业步，还涉及到本作业步之前的一些作业步。其作用是根据对先前作业步执行后的返回码进行指定的条件判断，以决定本作业步是否执行。

在 COND 参数中，是通过将先前作业步的返回码，与指定的测试码比较以对其进行条件判断的。其中测试码的取值范围为 0～4095。而先前作业步的返回码通常为 0、4、8、16，以及异常终止码 ABEND CODE。对返回码与测试码进行比较的操作符为以下 6 个关系字符。

- GT：大于。
- GE：大于等于。
- LT：小于。
- LE：小于等于。
- EQ：等于。
- NE：不等于。

需要特别注意的是，当测试条件不满足时，系统才执行本作业步。而当测试条件满足时，

系统则跳过该作业步不对其执行。这一点和人的正常思维习惯是有所不同的，容易混淆。例如，下面为一段 JCL 作业的部分代码。

```
//JOBA  JOB  CLASS=A, MSGCLASS=A, MSGLEVEL=(1,1),
//      NOTIFY=&SYSUID
//STEP1  EXEC  PGM=MYPGM1
……
//STEP2  EXEC  PGM=MYPGM2, COND=(4, GE, STEP1)
……
//STEP3  EXEC  PGM=MYPGM3, COND=((16, EQ),(20, LT, STEP1))
……
```

以上作业中包含有 COND 参数的作业步有两个，分别为 STEP2 和 STEP3。STEP2 在作业步 STEP1 的返回码大于 4 的情况下执行。STEP3 的执行需要满足两个条件。一个条件是该作业步之前的所有作业步的返回码都不等于 16，另一个条件是 STEP1 的返回码小于等于 20。

通过以上示例可以看到，COND 参数判断条件里返回码位置上的作业步名是可以默认的。当作业步名默认时，表示该判断条件应用于本作业步之前所有作业步的返回码。同时，COND 参数中还可以有多个判断条件，每一个判断条件相当于一个子参数。当存在多个判断条件时，只有全部条件都不满足，本作业步才执行。只要有一个条件满足，本作业步就不执行。

此外，COND 参数中还存在两个子参数，分别为 EVEN 和 ONLY，二者不能同时出现。当 COND 参数中包含有这两个子参数之一时，表示在先前作业步有异常终止的情况下也进行条件判断。否则，在此情况下本作业步将不被执行。

当 EVEN 或 ONLY 子参数设定时，系统都将不去测试先前任何异常终止作业步的返回码。其中 EVEN 子参数表示无论先前作业步是否异常终止，本作业步都将继续根据判断条件决定是否执行。而 ONLY 子参数则表示只有当先前作业步异常终止时，本作业步才可能执行。如表 14.2 所示，反映了这两个子参数和作业执行情况之间的关系。

表 14.2　　包含 EVEN/ONLY 子参数的作业执行情况

	先前作业步是否异常终止	测试条件是否满足	本作业步是否执行
EVEN	是	是	否
EVEN	是	否	是
EVEN	否	是	否
EVEN	否	否	是
ONLY	是	是	否
ONLY	是	否	是
ONLY	否	是	否
ONLY	否	否	否

需要注意的是，COND 参数的子参数最多只能为 8 个。因此，当不含 EVEN 或 ONLY 子参数时，COND 参数最多可以有 8 个判断条件。当包含 EVEN 或 ONLY 子参数时，COND 参数最多只能有 7 个判断条件。

例如，以下为一段 JCL 作业的部分代码。

```
//JOBB  JOB  CLASS=B, MSGCLASS=A, MSGLEVEL=(1,1),
//      NOTIFY=&SYSUID
//STEP0  EXEC  PGM=MYPGM0
```

```
……
//STEP1  EXEC  PGM=MYPGM1,COND=(0, NE, STEP0)
……
//STEP2  EXEC  PGM=MYPGM2, COND=((4, GT, STEP1), EVEN)
……
//STEP3  EXEC  PGM=MYPGM3, COND=((8, LE, STEP2), ONLY)
……
```

对于以上作业中的 STEP1 而言，该作业在 STEP0 的返回码为 0 的情况下将执行。同时，若 STEP0 异常终止了，则不进行条件判断，也不执行 STEP1 作业步。

对于 STEP2 而言，无论此前的作业步是否异常终止都将进行条件判断。并且，只要满足 STPE1 的返回码大于等于 4 的条件，该作业步都将执行。实际上，如果 STEP0 异常终止了，则 STEP1 将不被执行。STEP1 不执行，也就不存在对其返回码进行条件判断了。因此，只要 STEP0 异常终止了，STEP2 都必将会执行。

对于 STEP3 而言，只有在此前的作业步存在异常终止的情况下才进行条件判断。同时，该作业的执行还需满足 STEP2 的返回码小于 8 的条件。

最后，COND 参数还可用于调用过程的 EXEC 语句中。此时，COND 参数作为关键字参数，可覆盖到该过程的各过程步之中。此外，COND 参数中的判断条件也可以为与先前作业步所调用过程中的过程步进行的比较。例如，以下为几条与过程有关的 EXEC 语句。

```
//STEP3    EXEC   PROC=MYPROC,
//    COND.PROSTP1=(10, LT, STEP1),
//    COND.PROSTP2=((20,GE, STEP2, SPROC), EVEN)
//STEP4    EXEC    PGM=MYPGM,
//    COND=((50,GE),(8, LE, STEP3,PROSTP2))
```

以上 EXEC 语句调用的过程为 MYPROC。其中 PROSTP1 和 PROSTP2 为该过程中的 2 个过程步。而 PROSTP2 的 COND 参数中的 SPROC 则为之前的作业步 STEP2 所调用过程中的一个过程步。STEP4 作业步的执行情况，则涉及对过程步 PROSTP2 返回码的判断。

14.4 DD 语句

DD 语句是 JCL 3 种基本语句中最复杂的一种。该语句主要用于定义数据集及设置与该数据集相关的各种属性。下面分别从各方面对 DD 语句进行讲解。

14.4.1 DD 语句的语句名

同 JOB 语句和 EXEC 语句一样，DD 语句的语句名首先也必须满足 JCL 语句名字区中的要求。即 DD 语句的语句名也必须为一段由字母或通配符开头的 1～8 位字符。

注意到在一个 JOB 语句表示的作业中，可包含多条 EXEC 语句表示的作业步。与之类似，在一个 EXEC 语句表示的作业步中，也可以包含多条 DD 语句表示的数据定义。同一 JOB 语句中的 EXEC 语句名必须唯一。同样，同一 EXEC 语句中的 DD 语句名也必须惟一。不过，不同作业步中的 DD 语句名则是可以重复的。例如，以下各 DD 语句的语句名都是允许的。

```
//ST226DD  JOB  , ROBIN, CLASS=A, MSGCLASS=X
//    NOTIFY=&SYSUID
//STEP1    EXEC  PGM=A
```

```
//DD1      DD  ……
//DD2      DD  ……
//DD3      DD  ……
//STEP2    EXEC  PGM=B
//DD1      DD  ……
//DD2      DD  ……
```

虽然以上作业步 STEP2 中的两条 DD 语句和 STEP1 中的 DD 语句有重名现象，但这是允许的。当需要引用重名的某条 DD 语句时，可在该 DD 语句名之前加上其所在的作业步名以进行指定。

与 JOB 语句和 EXEC 语句所不同的是，DD 语句中的语句名还可以由系统指定。由系统命名的 DD 语句常用于指定一些特殊的参数以及 JCL 实用程序中。本节在最后将有讲解的几条特殊 DD 语句的语句名就是由系统指定的，这些语句名具体如下。

- JOBCAT
- JOBLIB
- STEPCAT
- STEPLIB
- SYSIN

以上名为 SYSIN 的 DD 语句常用于 JCL 的实用程序中。在 JCL 实用程序中，除该条 DD 语句外，通常还有以下几条由系统命名的 DD 语句。

- SYSPRIN
- SYSUT1
- SYSUT2

还有一些由系统命名的 DD 语句虽然在初级开发中不常会用到，但也是应该了解的。一起介绍如下。

- SYSMDUMP
- SYSUDUMP
- SYSCHK
- SYSCKEOV
- SYSABEND

以上为通常几条由系统所定义的 DD 语句名。当用户书写自己的 JCL 时，注意避免自己定义的 DD 语句与系统定义的相重复。

最后，当为应用程序输入输出文件定义数据集时，其命名规则将取决于程序所使用的开发语言。以下为大型机中各语言类型及其所命名规则的指定方式。

- COBOL 语言：由 ASSIGN 指定。
- 大型机汇编语言：由 DCB 宏命令指定。
- PL/1 语言：由 DECLARE 语句指定。
- FORTRAN 语言：由 READ 或 WRITE 语句中的通道号构成。

14.4.2 DD 语句中的位置参数

DD 语句中共有 3 种位置参数，这 3 种位置参数是可选的，但每次至多只能出现一种。

关于这 3 种位置参数分别介绍如下。

1．“*”参数

“*”参数用于表示一个流内数据集的开始。并且该参数所引导的流内数据集中的首两列不得为“/*”符号或“//”符号。因为前一个符号用于表示流内数据集的结束。而后一个符号则用于表示一个新的 JCL 语句的开始，同时也表示该流内数据集的结束。例如，以下为包含有位置参数“*”的 DD 语句使用示例代码。

```
//STEP1       EXEC  MYPROC
//CRT.DSA      DD   UNIT=3390
//CRT.INP1      DD  *
             123
             ABC
             CREATE
//PRT.DSB      DD   UNIT=180
//PRT.INP2     DD    *
              TEST
              TEST
              PRINT
/*
```

以上 CRT.INP1 所标明的 DD 语句所定义的输入数据将被编目过程中的过程步 CRT 使用。而 PRT.INP2 所标明的 DD 语句所定义的输入数据将被编目过程中的过程步 PRT 使用。以上两组数据都为通过 DD 语句中的参数“*”所引导的流内数据集中的内容。

2．DATA 参数

DD 语句中的位置参数 DATA 同样也用于引导一个流内数据集。与参数“*”不同的是，DATA 所引导的流内数据集中的首两列是可以为“//”符号的。此时，系统并不会将其看作一条新的 JCL 语句，而只将其视作流内数据集中的一段数据。因此，通过 DATA 引导的流内数据集只能通过/*语句表示结束。例如，以下为包含有位置参数 DATA 的 DD 语句使用示例。

```
//STEP2       EXEC  MYPROC
//CREATE.DSA   DD   UNIT=3390
//CRT.INP2     DD  DATA
//            123
            ABC
//INDATA       DD  DSN=TEST.DATA, SPACE=(TRK, (2, 1)), DISP=SHR
/*
//PRT.DSB      DD   UNIT=180
//PRT.INP2     DD    DATA
//            TEST
            TEST
            PRINT
/*
```

需要注意的是，以上第一组输入数据流中的最后一行并非一条 DD 语句。该行内容仅为其流内数据集中的一段数据。

此外，若输入数据流中要求在首两列中也包含有“/*”符号，则需要在 DD 参数后通过 DLM 设置。DLM 用来标明一个由用户任意指定的流内数据集结束标志，该结束标志通常为

两个字符。例如，以下 DD 语句在 DATA 参数后，通过 DLM 将流内数据集的结束标志指定为了“AA”。

```
//DLMAA   DD  DATA, DLM=AA
//BEGIN DATA
/*
TEST TEST TEST
……
TEST TEST TEST
/*
//END DATA
AA
```

3．DUMMY 参数

DD 语句中的最后一个位置参数 DUMMY 通常主要有两种用途。一种用途是作为空的内容填充某些 JCL 实用程序中相应参数的数据，以保证格式的完整。另一种用途是可以对 DD 语句中的各项参数进行语法检测。例如，下面为一条包含 DUMMY 参数的 DD 语句。

```
//TEST1   DD   DUMMY,
//  DSN=TEST.DS, DISP=OLD,
//  UNIT=3370, VOL=WORK01, SPACE=(TRK, (5, 2, 2)),
//  DSORG=PO, RECFM=FB, LRECL=80
```

以上语句定义的数据集并不会被创建，因为该 DD 语句中含有位置参数 DUMMY。该语句中除 DUMMY 参数以外的各项参数并不起实际作用，但将接受系统的语法检测。若语法检测无误，并且需要实际执行时，直接将 DUMMY 参数去掉便可。

4．位置参数的综合应用

实际上，以上 3 种位置参数最常应用于 JCL 实用程序中的输入 DD 语句及控制 DD 语句。以下为这 3 种位置参数在 JCL 实用程序中的用法，包含其用法的作业代码如下。

```
//SAMPLEA   JOB , CLASS=A, NOTIFY=&SYSUID
//STEP1  EXEC   PGM=IEBCOPY
//SYSPRINT  DD  SYSOUT=*
//INDS   DD  DSN=A.B.C
//OUTDS DD  DSN=X.Y.Z
//SYSIN  DD  *                              /*此处在实用程序 IEBCOPY 的控制语句中用到了参数“*”*/
  COPY  INDD=INDS
       OUTDD=OUTDS
      EXCLUDE MEMBER=(MEM1, MEM2)
//STEP2  EXEC  PGM=IEBGENER
//SYSPRINT  DD  SYSOUT=*
//SYSUT1  DD  DATA, DLM=ZZ                  /*此处在实用程序 IEBGENER 输入语句中用到了参数 DATA*/
//INJOB  JOB, CLASS=B, NOTIFY=&SYSUID       /*以下为流内数据集中的内容*/
//INS1  EXEC  PGM=MYPGM
//INDD  DD   *
      TEST
      TEST
/*
//OUTDD  DD  SYSOUT=*
ZZ                                          /*流内数据集在此处结束*/
```

```
//SYSUT2  DD   DSN=MYJOB.TEST.SAMPLEA, DISP=(, CATLG),
//         UNIT=SYSDA, VOL=SER=44321
//SYSIN    DD  DUMMY                          /*此处在 IEBGENER 的控制语句中用到了参数 DUMMY*/
```

以上两个作业步各自调用了两个 JCL 实用程序，并分别用到了 DD 语句中的 3 种位置参数。此处只需对这 3 种位置参数通常的作用途径有一个大致的印象，暂不必考虑各作业步的实际作用。

14.4.3 DD 语句中与数据集相关的关键字参数

DD 语句的关键字参数比较多，不过总的来说可以分为两大类：一类为与数据集相关的关键字参数，主要涉及到逻辑上的概念；另一类为与设备相关的关键字参数，主要涉及到物理上的概念。本节将主要讲解与数据集相关的关键字参数。

1．DSNAME 参数

DSNAME 参数用来指定一个数据集的名字，是 DD 语句中最常用的参数。当该数据集存在时，表示对原有数据集的引用；当该数据集不存在时，表示对新数据集的命名。此外，参数名“DSNAME”在实际应用中也常简写为“DSN”。例如，以下为几条包含有 DSNAME 参数的 DD 语句。

```
//DD1  DD  DSNAME=MYDATA.OLD.A, DISP=SHR
//DD2  DD  DSNAME=MYDATA.NEW.A, DISP=(NEW, CATLG,CATLG)
//     UNIT=3390, VOL=SER=WORK04
//DD3  DD  DSN=MYDATA.OLD.B, DISP=OLD
//DD4  DD  DSN=MYDATA.NEW.B, DISP=(, CATLG)
//     UNIT=3390, VOL=SER=WORK06
```

DSNAME 参数也可用来指定分区数据集中的成员。指定方式为将成员名写在其所在分区数据集名字之后的括号内。同时，DSNAME 参数后的数据集名还可为一个仅含部分内容的抽象数据集名。例如，下面为包含以上用法的几条 DD 语句。

```
//DDA  DD  DSN=A.B.MYLIB(MEMBER1), DISP=SHR
//DDB  DD  DSN=&&TEMP
//DDC  DD  DSN=&&LIB(MEM2), DISP=SHR
```

以上第一条 DD 语句引用了分区数据集 MYLIB 中的成员 MEMBER1。第二条 DD 语句创建了一个临时数据集。该临时数据集名字的最后一个部分为“TEMP”。第三条 DD 语句引用了一个分区数据集中的成员 MEM2。该分区数据集名字的最后一个部分为“LIB”。实际上，以上后两条 DD 语句所指定的抽象数据集名的表示形式分别如下。

```
&&TEMP: 用户 ID.作业名.作业 ID.数据集号.TEMP
&&LIB: 用户 ID.作业名.作业 ID.数据集号.LIB
```

最后，DSNAME 参数也可对之前作业步中 DD 语句里指定的数据集进行引用。例如，以下 STEP2 中名为 DD02 的 DD 语句就引用了 STEP1 中 DD01 语句所指定的数据集。而 STEP3 中的 DD03 语句则引用 STEP02 中 DD02 语句指定的数据集，实际引用的为同一数据集。相应 JCL 语句如下。

```
//STEP1  EXEC  PGM=P01
//DD01  DD  DSN=DATA01.LEVEL01.GROUP01, DISP=(, PASS),
```

```
//      UNIT=3390, VOL=SER=WORK04
//DD1   DD  DSN=TEST
//STEP2  EXEC  PGM=P02
//DD02  DD  DSN=*.STEP1.DD01,
//      DISP=(OLD,PASS,DELETE)
//STEP3  EXEC  PGM=P03
//DD03  DD  DSN=*.STEP2.DD02,
//      DISP=(OLD, CATLG, DELETE)
```

2．DISP 参数

DISP 参数用于描述数据集的状态，并指定该数据集在作业或作业步结束后如何进行相应处理。DISP 参数中含有 3 个位置子参数：其中第一个参数用于描述数据集的状态；第二个参数指定作业或作业步正常结束时如何处理该数据集；第三个参数指定异常结束时的处理方式。

用于描述数据集状态的子参数通常有以下 4 种取值情况。

- NEW：表示该数据集为一个新创建的数据集。
- OLD：表示该数据集已经存在，并且当前作业步以独占方式使用该数据集。其他作业或作业步不能同时对其进行使用。
- SHR：表示该数据集已经存在，并且当前作业步以共享方式使用该数据集。其他作业或作业步此时也可对其进行使用。
- MOD：若该数据集已经存在，并且为顺序数据集，记录将被添加到该数据集的末尾。如果该数据集不存在，则创建一个新的数据集。以上两种情况中，当前作业步都以独占方式方式使用该数据集。

用于对数据集进行处理的子参数通常有以下几种取值情况。

- DELETE：删除该数据集。
- KEEP：保留该数据集。
- CATLG：将该数据集进行编目。
- UNCATLG：将该数据集解除编目。
- PASS：将数据集保留传递给同一作业的后续作业步中使用。该参数数据仅用于正常结束的情况。

此外，DISP 参数及其子参数是可以默认的。当该参数或其部分子参数默认时，所对应的默认取值如下。

默认整个 DISP 参数：DISP=(NEW, DELETE, DELETE)

```
DISP=NEW: DISP=(NEW, DELETE, DELETE)
DISP=OLD: DISP=(OLD, KEEP, KEEP)
DISP=SHR: DISP=(SHR, KEEP, KEEP)
DISP=MOD:DISP=(MOD, KEEP, KEEP)
DISP=(, CATLG): DISP=(NEW, CATLG, CATLG)
```

3．DCB 参数

DCB 全称为 Data Control Block，DCB 参数即数据控制块参数。该参数主要用在新创建的数据集上，用于描述数据集的属性。这些属性包括数据集的组织结构、记录格式、逻辑记

录长度、记录块大小等。DCB 参数中主要有以下几个关键字子参数，分别如下。

- DSORG 参数：指定数据集的组织结构。其中参数数据 PS 或 PSU 表示顺序数据集；PO 或 POU 表示分区数据集；DA 或 DAU 表示直接定位数据集；IS 或 ISU 表示索引数据集。PS 和 PO 这两个参数数据是最常用到的。该参数默认时为 PS。
- RECFM 参数：指定数据集的记录格式。其中参数数据 FB 表示定长组块记录格式；F 表示定长不组块记录格式；VB 表示变长组块记录格式；V 表示变长不组块记录格式。参数数据 FB 是最常用到的。
- LRECL 参数：指定数据集的逻辑记录长度，以字节为单位。对于定长记录，该参数数值为实际记录的长度；对于变长记录，该参数数值为变长记录中最大的记录长度再加上 4 个字节的控制信息。
- BLKSIZE 参数：指定数据集的记录块大小，以字节为单位。其中最大的记录块大小不得超过 32K。并且，对于定长组块的记录，该参数数值必须为 LRECL 参数数值的偶数倍。当该参数数值为 0 时，将由系统指定一个最适当的记录块大小。

以上这些子参数既可写在 DCB 参数后面的括号内，也可单独列出来。例如，以下为几条包含 DCB 参数的 DD 语句。

```
//DD1  DD   DSN=DATA1, DISP=(, CATLG),
//    UNIT=3380, VOL=SER=WORK10, SPACE=(23472, (200, 40)),
//    DCB=(DSORG=PS, RECFM=FB, LRECL=80, BLKSIZE=800
//DD2  DD   DSN=DATA.LIB, DISP=(NEW, KEEP),
//    UNIT=3380, VOL=SER=WORK10, SPACE=(23472, (200, 40)),
//    RECFM=FB, LRECL=100, DSORG=PO
```

4．SYSOUT 参数

SYSOUT 参数主要用于将相应的数据集标志为一个系统输出数据集。该参数的参数数据可以为一个输出类。输出类同作业类一样，也是由 A～Z 或 0～9 中的一个字符所表示。当参数数据为“*”号时，表示输出类与 JOB 语句中的 MSGCLASS 参数定义相同。该参数常用于 JCL 实用程序中名为 SYSPRINT 的 DD 语句中，用法如下。

```
//STEP1   EXEC   PGM=IEHLIST
//SYSPRINT  DD  SYSOUT=*          /*此处在实用程序 IEHLIST 中用到了 SYSOUT 参数*/
//DD1  DD  DISP=OLD, UNIT=SYSDA, VOL=SER=VLTEST
//SYSIN  DD  *
 LISTVTOC  FORMAT, VOL=SYSDA=VLTEST
 DSNAME=ES10.MVS74.LIST
```

14.4.4 DD 语句中与设备相关的关键字参数

DD 语句中与设备相关的关键字参数通常主要有 3 种功能。一种用于指定数据集所在的设备；另一种用于指定数据集所在设备上的卷；还有一种用于指定设备上分配给数据集的物理空间。下面分别对这 3 种功能相应的关键字参数进行介绍。

1．UNIT 参数

UNIT 参数用于为数据集指定物理设备。物理设备通常使用设备类型或设备组名进行描述。其中设备类型通常由数字组成。以下分别为磁带机和磁盘机中包含的设备类型。

- 磁带机的类型：3480、3422、……
- 磁盘机的类型：3390、3380、3375、3340、……

设备组名则由 1～8 个英文字母组成，表示一台或一组设备。通常使用的设备组名有 SYSDA、DASD、TAPE、CART 等。

2．VOLUME 参数

VOLUME 参数用于指定数据集所在物理设备上的卷，该参数也可简写为 VOL。VOLUME 参数中包含两个关键字子参数，分别如下。

- SER 子参数：该参数用来直接指定卷标号，卷标号由 1～6 位数字、字母、通配符或特殊字符所组成。
- REF 子参数：该参数通过从其他已知数据集中间接获得卷标号。不过该参数所指定的数据集不能是 GDG 世代数据集及其成员。

此外，数据集还允许跨卷存放。数据集的跨卷存放可以通过在 VOLUME 参数中同时指定多个卷标号实现。以下为几条包含有 VOLUME 参数以及 UNIT 参数的 DD 语句。

```
//DD1  DD  DSN=TEST.A, DISP=(MOD, KEEP),
//     UNIT=3390,
//     VOLUME=SER=WORK01
//DD2  DD  DSN=TEST.B,
//     UNIT=3480,
//     VOL=REF=*.DD1
//DD3  DD  DSN=TEST.C, DISP=(NEW, CATLG),
//     UNIT=SYSDA,
//     VOL=SER=(85412, 45875)
//     SPACE=(TRK, (2, 1))
```

需要注意的是，对于新创建的数据集，通常都应指定 UNIT 和 VOLUME 参数。但如果该数据集是由 SMS（存储管理子系统）管理的，则不必指定这两个参数。即使指定了这两个参数，系统也会根据 SMS 中的设置来进行分配。

3．SPACE 参数

SPACE 参数用于对数据集分配存储空间。该参数中的第一个位置子参数用于表示所分配存储空间的单位。存储空间的单位通常有以下几种。

- TRK：表示以磁道为单位进行分配。
- CYL：表示以柱面为单位进行分配。
- 0～65535 中的一个数字：表示以字节为单位进行分配。其中在非 SMS 环境下表示数据的平均块长度，在 SMS 环境下表示数据的平均记录长度。

SPACE 参数中的第二个位置子参数中又包含有 3 个位置子参数。其中第一个子参数用于指定初次分配的数量，第二个子参数用于指定再次追加的数量。系统在初次分配量不足时将进行追加。其中对于非 VSAM 数据集最多追加 15 次，对于 VSAM 数据集最多追加 122 次。

第三个子参数仅用于分区数据集中，表示分区数据集目录区的数据块个数。其中每一数据块为 256 字节，可包含 5 个成员名。

此外，在 SPACE 参数中还包含有其他子参数，如 RLSE、CONTIG、ABSTR 等。这些子

参数相对来说并不常用，因此不再详细讨论。以下为包含有 SPACE 参数的几条 DD 语句，注意其中每个子参数的意义。

```
//DDA  DD  DSN=TEST.PS.A, DISP=(NEW, CATLG),
//     SPACE=(TRK, (10, 2)),
//     UNIT=SYSDA, VOL=SER=ABCDE
//DDB  DD  DSN=TEST.PDS.B, DISP=(, CATLG),
//     SPACE=(CYL,(2,1,1)),
//     UNIT=3390, VOL=SER=12345
//DDC  DD  DSN=TEST.PS.C, DISP=(, CATLG),
//      SPACE=(32768, (5, 2)),
//     UNIT=SYSDA, VOL=SER=ABCDE
//DDD  DD  DSN=TEST.PS.D, DISP=(, CATLG),
//      SPACE=(TRK, 8),
//     UNIT=SYSDA, VOL=SER=ABCDE
```

14.4.5 特殊的 DD 语句

此处所说的特殊的 DD 语句是指由系统命名的 DD 语句。由前面第一小节的内容可以知道由系统命名的 DD 语句是很多的。此处仅对其中常用的 5 种语句进行讲解。

1．JOBCAT 和 JOBLIB

语句名为 JOBCAT 的 DD 语句用于为作业定义一个私有的编目。系统将在搜索主编目和用户编目之前，首先搜索本语句中定义的私有编目。

语句名为 JOBLIB 的 DD 语句用于为作业创建或指定一个私有的库。系统将首先在该私有库中搜索 EXEC 语句中的 PGM 参数所指定调用的程序。私有库中的每一个成员都应该为一个可执行程序。系统只有在私有库中未搜索到相应程序时，才会去搜索系统库。

此外，这两条 DD 语句在作业中是有位置要求的。其中语句名为 JOBCAT 的 DD 语句应位于 JOB 语句之后，第一条 EXEC 语句之前。当作业中同时包含有以上两条语句时，语句名为 JOBLIB 的 DD 语句应该在前。以下为这两条语句在实际中的用法。

```
//TEST01   JOB  CLASS=A
//JOBLIB   DD   DSN=PRIV.MYLIB, DISP=SHR
//JOBCAT   DD   DSN=MYCAT, DISP=SHR
//STEP     EXEC  PGM=MYPGM
......
```

对于以上作业，系统先从私有编目 NYCAT 中获取该作业的相关信息。同时，系统将从私有库 PRIV.MYLIB 中搜索 STEP 作业步所调用的程序 MYPROC。

2．STEPCAT 和 STEPLIB

以上两条 DD 语句的功能和前面讲解的两条基本类似。只是这两条语句的作用范围是一个作业步，而不是整个作业。

语句名为 STEPCAT 的 DD 语句指定的私有编目将会在作业步中覆盖 JOBCAT 所指定的私有编目。并且，该条 DD 语句可以出现在作业步中的任何位置。以下为该语句在实际中的用法。

```
//TEST02   JOB  CLASS=A
//JOBCAT   DD   DSN=MYCAT, DISP=SHR
```

```
//STEP1    EXEC  PGM=TEST02
//STEP2    EXEC  PGM=MYPGM02
//STEPCAT  DD
//    DSN=STPCAT, DISP=SHR
//DD1   DD  DSN=ST253.TEST.DATA, DISP=SHR
```

以上作业通过 JOBCAT 定义了一个私有编目 MYCAT。而对于该作业中的作业步 STEP2 而言，则通过 STEPCAT 定义了另一个私有编目 STPCAT。在该作业步中，编目 STPCAT 将覆盖编目 MYCAT。

语句名为 STEPLIB 的 DD 语句同样也可以在作业步中覆盖 JOBLIB 指定的私有库。并且，在 STEPLIB 中可以通过多条 DD 语句定义多个私有库。以下为该语句在实际中的用法。

```
//TEST02   JOB  CLASS=A
//JOBLIB   DD   DSN=ST253.LIB03.GRP01, DISP=(OLD, PASS)
//STEP1   EXEC  PGM=TEST01
//STEP2   EXEC  PGM=TEST02
//STEPLIB  DD
//      DSN=TEST.LIB, DISP=SHR
//STEP3   EXEC  PGM=TEST03
//STEPLIB  DD
//      DSN=*.STEP2.STEPLIB, DISP=(OLD, KEEP)
//STEP4   EXEC  PGM=TEST04
//STEPLIB  DD    DSN=MULTI.TEST.LIB01, DISP=(OLD, PASS)
//         DD   DSN=MULTI.TEST.LIB02, DISP=(OLD, KEEP),
//          UNIT=3390, VOL=SER=AB12CD
//         DD   DSN=MULTI.TEST.LIB03, DISP=(OLD, KEEP)
//STEP5   EXEC  PGM=TEST05
```

对于以上作业，系统将先在 ST253.LIB03.GRP01 中搜索程序 TEST01，在 TEST.LIB 中搜索程序 TEST02 以及 TEST03，在 ST253.LIB03.GRP01 中搜索程序 TEST05。对于程序 TEST04，系统将先在下面几个私有库中进行搜索。这几个私有库依次为 MULTI.TEST.LIB01、MULTI.TEST.LIB02、MULTI.TEST.LIB03。

3．SYSIN

语句名为 SYSIN 的 DD 语句常用于定义一个内部数据集。同时，该语句在调用 JCL 的实用程序中经常出现，用于输入相关控制信息。以下为该语句在实际中的用法。

```
//TEST03   JOB  CLASS=A
//STEP1   EXEC  PGM=READ01
//SYSIN   DD  *
    READ  READ  READ
//OUT01   DD  SYSOUT=A
//STEP2   EXEC  PGM=WRITE01
//SYSIN  DD  DATA, DLM=00
***********    INSTREAM  DATA  BEGIN***********
//INSTP  EXEC  INPROC
//INDD   DD    DSN=TEST.LIB, DISP=SHR
***********    INSTREAM  DATA  END  ***********
00
//SPUTL   EXEC  IEBGENER
//SYSPRINT  DD  SYSOUT=*
//SYSUT1   DD  DSN=TEST.DATA.PS01, DISP=SHR
```

```
//SYSUT2   DD  DSN=TEST.DATA.PS02, DISP=SHR
//SYSIN    DD  DUMMY
```

14.5 JCL 实用程序

前面所讲的 JCL 语句是本章的基础，此处要讲解的 JCL 实用程序则是本章的重点。日常使用 JCL 完成的各项功能几乎都是通过调用相应的 JCL 实用程序实现的。JCL 实用程序也很多，此处只讲解其中常用的几种。

14.5.1 IEFBR14 实用程序

IEFBR14 实用程序相当于一个空的程序，其本身并不实现任何特定功能。不过，用户可在调用该程序的作业步中通过 DD 语句中的相应参数实现数据集的新建与删除等功能。例如，以下作业分别创建了一个顺序数据集 BR14.PS.GRP01，与一个分区数据集 BR14.PDS.GRP01。

```
//ALBR14  JOB  'BR14,ALLOCATE', JENNY,
//    CLASS=C, MSGCLASS=M, NOTIFY=&SYSUID
//ALOT    EXEC  PGM=IEFBR14
//ALPS    DD    DSN=BR14.PS.GRP01, DISP=(NEW, CATLG),
//        UNIT=3390, VOL=SER=WORK01,
//        SPACE=(TRK, (5,2)),
//        DCB=(RECFM=FB, LRECL=80, BLKSIZE=2400, DSORG=PS)
//ALPDS   DD    DSN=BR14.PDS.GRP01, DISP=(, CATLG),
//        UNIT=3390, VOL=SER=WORK01,
//        SPACE=(TRK, (10, 3, 2)),
//        RECFM=FB, LRECL=80, DSORG=PO
```

可以看到，当创建新的数据集时，关键需要对 DD 语句中的 DISP 参数进行设置。该参数中的第一个子参数必须为 NEW，以表明其为一个新的数据集。第二个子参数通常为 CATLG，这样在作业步正常结束时将会对所创建的数据集进行编目。同时，此处也需要指定新建数据集相应的设备及属性参数。

当在 IEFBR14 中删除数据集时，DISP 的第一个位置参数通常为 OLD。第二个位置参数则必须为 DELETE。例如，以下作业删除了多个不再使用的数据集。

```
//DEBR14  JOB  'BR14,DELETE', JENNY,
//    CLASS=C, MSGCLASS=M, NOTIFY=&SYSUID
//DELT    EXEC  PGM=IEFBR14
//DD1     DD    DSN=BR14.DATA.TEMP01, DISP=(OLD, DELETE)
//DD1     DD    DSN=BR14.DATA.TEMP02, DISP=(OLD, DELETE)
//DD1     DD    DSN=BR14.DATA.TEMP03, DISP=(OLD, DELETE)
```

14.5.2 IEBGENER 实用程序

IEBGENER 实用程序主要用于创建、复制、及打印顺序数据集。其中顺序数据集为该实用程序的主要针对对象，复制则为其默认的操作。

IEBGENER 实用程序中涉及到两个数据集，其中一个作为源数据集，另一个作为目标数据集。该程序实现的功能便是将源数据集中的数据复制到目标数据集中。如果目标数据集为新的数据集，则将在复制之前创建该数据集。此时相当于实现了数据的备份。例如，以下作业将顺序数据集 TEST.DATA 中的数据复制到了 TEST.DATA.BACKUP 之中。

```
//BACK01  JOB  , JESSIE,
//    CLASS=C, MSGCLASS=M, NOTIFY=&SYSUID
//BACUP  EXEC  PGM=IEBGENER
//SYSPRINT  DD  SYSOUT=*
//SYSUT1   DD  DSN=TEST.DATA, DISP=SHR
//SYSUT2   DD  DSN=TEST.DATA.BACKUP, DISP=(, CATLG),
//       UNIT=3390, VOL=SER=WORK01,
//       SPACE=(TRK, (5,2)),
//       RECFM=FB, LRECL=80, BLKSIZE=2400, DSORG=PS
//SYSIN    DD  DUMMY
//
```

可以看到，在 IEBGENER 中，是通过名为 SYSUT1 的 DD 语句指定源数据集的。而目标数据集则通过名为 SYSUT2 的 DD 语句指定。

此外，IEBGENER 也可实现顺序数据集和分区数据集中一个成员的相互复制。例如，以下作业便实现了这一功能。

```
//BACK02  JOB  , JESSIE,
//    CLASS=C, MSGCLASS=M, NOTIFY=&SYSUID
//STEP1  EXEC  PGM=IEBGENER
//SYSPRINT  DD  SYSOUT=*
//SYSUT1   DD  DSN=TEST.DATA.PDS(MEM1), DISP=SHR
//SYSUT2   DD  DSN=TEST.DATA.PS1, DISP=SHR
//SYSIN    DD  DUMMY
//STEP2  EXEC   PGM=IEBGENER
//SYSPRINT  DD  SYSOUT=*
//SYSUT1   DD  DSN=TEST.DATA.PS2, DISP=SHR
//SYSUT2   DD  DSN=TEST.DATA.PDS(MEM2), DISP=SHR
//SYSIN    DD   DUMMY
//
```

当需要在创建一个新的顺序数据集的同时对该数据集赋予数据时，也可使用 IEBGENER 实现。此时，可将由 SYSUT1 指定的源数据集定义为一组流内数据。当新的数据集被创建时，其数据内容将为该流内数据。例如，以下作业创建了一个顺序数据集 TEST.NEW，并且该数据集中的数据内容为 3 行“TEST”。

```
//CREAT   JOB  , JESSIE,
//    CLASS=C, MSGCLASS=M, NOTIFY=&SYSUID
//CRT    EXEC  PGM=IEBGENER
//SYSPRINT  DD  SYSOUT=*
//SYSUT1   DD  *
   TEST
   TEST
   TEST
//SYSUT2   DD  DSN=TEST.PS.NEW, DISP=(, CATLG),
//       UNIT=3390, VOL=SER=WORK01,
//       SPACE=(TRK, (5,2)),
//       RECFM=FB, LRECL=80, BLKSIZE=2400, DSORG=PS
//SYSIN    DD  DUMMY
//
```

IEBGENER 中名为 SYSIN 的 DD 语句用于引导控制语句。当对数据集进行打印输出时，可通过控制语句指定数据记录的输出格式。以下作业便进行了这一操作。

```
//PRINT   JOB  , JESSIE,
```

```
//    CLASS=C, MSGCLASS=M, NOTIFY=&SYSUID
//PRT  EXEC  PGM=IEBGENER
//SYSPRINT  DD  SYSOUT=*
//SYSUT1   DD  DSN=TEST.DATA.PRNT, DISP=SHR
//SYSUT2   DD  SYSOUT=*
//SYSIN     DD   *
 GENERATE  MAXFLDS=3
 RECORD  FIELD=(4,1,,1)
 RECORD  FIELD=(5,6,,10)
 RECORD  FIELD=(3,12,,6)
/*
```

关于以上控制语句中的各项内容分别介绍如下。

- GENERATE：表示调用编辑功能。
- MAXFLDS：通过其后的数值指定下面最多分为多少个数据区域进行描述。
- RECORD：表示一块数据区域。

FIELD：用于对数据区域进行描述。其后括号内的第一个子参数表示数据的长度；第二个子参数表示数据在输入数据中的起始列；第三个参数表示数据转换方式，默认为不进行转换；第四个子参数表示数据在输出数据中的起始列。

此外，以上控制语句也常写为如下形式。

```
......
//SYSIN     DD   *
 GENERATE  MAXFLDS=3
 RECORD  FIELD=(4,1,,1), FIELD=(5,6,,10), FIELD=(3,12,,6)
/*
```

通过控制语句还可以将一个顺序数据集中的数据进行切分，分别复制到分区数据集的多个成员中。例如，以下作业便根据顺序数据集 TEST.DATA.PS 中的数据创建了一个新的分区数据集。该分区数据集为 TEST.DATA.PDS，所包含的两个成员分别为 MEM1 和 MEM2。作业如下。

```
//TOMEM   JOB  , JESSIE, CLASS=C, MSGCLASS=M, NOTIFY=&SYSUID
//CPMEM  EXEC  PGM=IEBGENER
//SYSPRINT  DD  SYSOUT=*
//SYSUT1   DD  DSN=TEST.DATA.PS, DISP=SHR
//SYSUT2   DD  DSN=TEST.DATA.PDS, DISP=(, CATLG), UNIT=3390, VOL=SER=WORK01,
//    SPACE=(TRK,(5,2,2)), RECFM=FB, LRECL=80, DSORG=PO
//SYSIN     DD   *
 GENERATE  MAXNAME=3, MAXGPS=2
 MEMBER NAME=MEM1
           RECORD IDENT=(4,'GAPS',1)
 MEMBER NAME=MEM2
```

根据以上控制语句，可以知道 TEST.DATA.PS 中的数据是根据其中的字符“GAPS”进行划分的。该字符为 4 个字节，位于源数据中的第一列。该字符及其以上的数据被复制到 MEM1 中，该字符以下的数据被复制到 MEM2 中。

14.5.3 IEBCOPY 实用程序

IEBCOPY 的默认操作仍然为复制。不过，IEBCOPY 针对的对象主要是分区数据集。下

面分别对 IEBCOPY 在实际中通常的作用方式进行讲解。

IEBCOPY 最基本的作用就是复制分区数据集。通过复制，可以实现对数据集的备份。以下作业便调用该实用程序，实现了对分区数据集 TEST.PDS 的备份。

```
//BKPDS   JOB  , MARK,
//    CLASS=A, MSGCLASS=M, NOTIFY=&SYSUID
//BAKPDS  EXEC  PGM=IEBCOPY
//SYSPRINT  DD  SYSOUT=*
//SYSUT1  DD  DSN=TEST.PDS, DISP=SHR
//SYSUT2  DD  DSN=TEST.PDS.BACKUP, DISP=(,CATLG),
//   UNIT=3390, VOL=SER=SYSDA, SPACE=(TRK,(2,1,2))
//SYSIN    DD  DUMMY
```

此外，在 IEBCOPY 中也可通过其他 DD 语句指定源数据集和目标数据集。不过此时需要在控制语句中对这两条 DD 语句进行输入输出方向的指定。其中输入方向通过 INDD 指定，对应源数据集；输出方向通过 OUTDD 指定，对应目标数据集。相应作业如下。

```
//CTLIO   JOB  , MARK,
//    CLASS=A, MSGCLASS=M, NOTIFY=&SYSUID
//MULTCP  EXEC  PGM=IEBCOPY
//SYSPRINT  DD  SYSOUT=*
//DD1  DD  DSN=TEST.DATA.PDS01, DISP=SHR
//DD2  DD  DSN=TEST.DATA.PDS02, DISP=SHR
//DD3  DD  DSN=TEST.DATA.PDS03, DISP=SHR
//OUTPUT  DD  DSN=TEST.PDS.GROUP, DISP=OLD
//SYSIN    DD  *
 COPY  OUTDD=OUTPUT
       INDD=(DD1, DD2, DD3)
```

以上作业是分别将 3 个分区数据集复制到了 1 个分区数据集中。该目标分区数据集将包含 3 个源分区数据集中所有的成员。

由于分区数据集是由该数据集中的各个成员组成的。因此，对于分区数据集的复制实际上就是对该数据集中成员的复制。默认情况下，IEBCOPY 对数据集中的每个成员都将进行复制。而通过控制语句，则可以选择部分成员进行复制或者不复制。

例如，以下作业对分区数据集 ADCDA.TEST.PDS01 进行复制。但不对该数据集中的成员 GAP1、GAP2 和 GAP3 进行复制。相应作业如下。

```
//EXMEM   JOB  , MARK,
//    CLASS=A, MSGCLASS=M, NOTIFY=&SYSUID
//PART01  EXEC  PGM=IEBCOPY
//SYSPRINT  DD  SYSOUT=*
//INSET  DD  DSN=ADCDA.TEST.PDS01, DISP=SHR
//OUTSET  DD  DSN=ADCDA.TEST.PDSGP01, DISP=OLD
//SYSIN    DD  *
 COPY  OUTDD=OUTSET
       INDD=INSET
       EXCLUDE  MEMBER=(GAP1, GAP2, GAP3)
```

以下作业则只对分区数据集 ADCDA.TEST.PDS02 中的成员 MEM1、MEM2 和 MEM3 进行了复制。并且，该作业将覆盖目标数据集中的成员 MEM2（默认情况不覆盖）。同时，在目标数据集中将复制过来的成员 MEM3 重命名为 NEWMEM3。相应作业如下所示。

```
//SMEM   JOB  , MARK,
//    CLASS=A, MSGCLASS=M, NOTIFY=&SYSUID
//MULTCP  EXEC  PGM=IEBCOPY
//SYSPRINT  DD  SYSOUT=*
//INSET  DD  DSN=ADCDA.TEST.PDS02, DISP=SHR
//OUTSET  DD  DSN=ADCDA.TEST.PDSGP02, DISP=OLD
//SYSIN    DD  *
 COPY  OUTDD=OUTSET
        INDD=INSET
        SELECT MEMBER=(MEM1, (MEM2, , R), (MEM3, NEWMEM3, R))
```

当对一个分区数据集中的成员进行多次的创建和删除后，该数据集将会存在大量的剩余空间。此时，可以通过 IEBCOPY 对该数据集进行压缩。压缩的方式为将 IEBCOPY 中的源数据集和目标数据集同时指定为该数据集。相应作业如下。

```
//ZIP   JOB  , MARK,
//    CLASS=A, MSGCLASS=M, NOTIFY=&SYSUID
//COMPRESS  EXEC  PGM=IEBCOPY
//SYSPRINT  DD  SYSOUT=*
//SYSUT1  DD  DSN=ADCDA.ZIP.PDS, DISP=SHR
//SYSUT2  DD  DSN=ADCDA.ZIP.PDS, DISP=SHR
//SYSIN    DD  DUMMY
```

14.5.4 ICEMAN 实用程序

ICEMAN 用于对数据集中的数据进行排序与合并。该实用程序同前面学习的 COBOL 中的 SORT 和 MERGE 语句比较类似。实际上，在大型机中有关数据的排序与合并都是建立在 DFSORT 机制上的。

例如，以下作业调用 ICEMAN 对顺序数据集 ADCDB.SOURCE.DATA 进行了排序操作。排序后的结果输出到数据集 ADCDB.OUTPUT.DATA 中。相应作业如下。

```
//DSSORT   JOB  , STONE, NOTIFY=&SYSUID
//ICESORT   EXEC  PGM=ICEMAN
//SYSOUT  DD   SYSOUT=*               /*注意，此处并非 SYSPRINT*/
//SORTIN  DD  DSN=ADCDB.SOURCE.DATA, DISP=SHR
//SORTOUT  DD  DSN=ADCDB.OUTPUT.DATA, DISP=(, CATLG),
//     UNIT=3390, VOL=SER=ABC123,
//     SPACE=(TRK,(5,2)),
//     RECFM=FB, LRECL=80, BLKSIZE=1600
//SYSIN    DD  *
 SORT  FIELDS=(10, 8, CH, A, 22, 5, CH, D)
/*
```

以上控制语句表示将原始数据中，从第 10 列开始的 8 个字符数据进行升序排列。同时，将原始数据中从第 22 列开始的 5 个字符数据进行降序排列。

使用 ICEMAN 进行数据的合并时，需要将控制语句中的 SORT 命令替换为 MERGE 命令。同时，可通过 SORTIN01～SORTIN16 定义多个合并输入文件。其他地方与排序基本类似，此处不再赘述。

14.5.5 IEBPTPCH 实用程序

IEBPTPCH 实用程序主要用于对数据集的打印输出。使用该程序进行打印输出的除顺序

数据集和分区数据集外，还可以为其他内容。比如分区数据集中的部分成员，数据集中的部分数据，或者分区数据集的目录等。

以下作业对顺序数据集 ST100.PRT.PS 进行了打印输出，并设置了数据记录打印的格式。同时，该作业还另外打印了两个标题。作业如下。

```
//PRTPS   JOB  , MIKE, NOTIFY=&SYSUID
//PRT01   EXEC  PGM=IEBPTPCH
//SYSPRINT  DD   SYSOUT=*
//SYSUT1  DD  DSN=ADCDB.PRT.PSDATA, DISP=SHR
//SYSUT2  DD  SYSOUT=*
//SYSIN    DD  *
  PRINT
  TYPORG=PS
  MAXFLDS=3
  TITLE ITEM=('   DATALIST   REPORT    ', 30)
  TITLE ITEM=('ROW  DATA    SIGN', 25)
  RECORD  FIELD=(5,2,,1), FIELD=(3,10,,8), FIELD=(10,15,,12)
```

以上控制语句中的 MAXFLDS、RECORD 和 FIELD 的意义同前面在 IEBGENER 中讲解的一致。此外，PRINT 命令用于标明打印控制信息的开始；TYPORG 用来指定数据集的类型；TITLE ITEM 用来输出标题。其中 ITEM 的第一子参数为标题的内容，第二个子参数为标题的起始列。

以下作业则对分区数据集 ST100.PRT.PDS 中的成员 CAT 和 DOG 进行了打印输出。其中控制语句中涉及到了与分区数据集成员相关的参数 MAXNAME 和 MEMBER NAME。作业如下。

```
/PRTPDS   JOB  , MIKE, NOTIFY=&SYSUID
//PRT01   EXEC  PGM=IEBPTPCH
//SYSPRINT  DD   SYSOUT=*
//SYSUT1  DD  DSN=ADCDB.PRT.PODATA, DISP=SHR
//SYSUT2  DD  SYSOUT=*
//SYSIN    DD  *
  PRINT
  TYPORG=PO
  MAXNAME=2
  MEMBER NAME=CAT
  MEMBER NAME=DOG
```

14.5.6 IEBCOMPR 实用程序

IEBCOMPR 实用程序常用于对两个数据集进行比较。对数据集的比较操作常用于确定备份数据集与原数据集是否一致。

判断两个顺序数据集一致，需要满足两个条件：一个条件是这两个数据集的 LRECL（逻辑记录长度）一致；另一个条件是两者所包含的数据记录一致。

判断两个分区数据集一致，也需要满足两个条件：一个条件是这两个数据集对应的成员包含的记录数量一致；另一个条件是两者所包含的数据记录一致。

例如，以下作业通过调用 IEBCOMPR 对两个顺序数据集进行了比较。

```
/COMP   JOB  , CLARK, NOTIFY=&SYSUID
//COMSTP   EXEC  PGM=IEBCOMPR
```

```
//SYSPRINT  DD   SYSOUT=*
//SYSUT1  DD  DSN=ADCDB.COMP.PS01, DISP=SHR
//SYSUT2  DD  DSN=ADCDB.COMP.PS02, DISP=SHR
//SYSIN    DD  DUMMY
```

当两个数据集不一致时，通过 IEBCOMPR 将会输出以下信息。

- 定义数据集的 DD 语句的语句名。
- 数据集的数据记录及记录块的数量。
- 不一致的记录，并且最多输出 10 条。

14.5.7 IEHLIST 实用程序

IEHLIST 主要用于对一些系统信息进行列表。其中包括对编目的列表、对 VTOC 的列表以及对位于同一个卷上的分区数据集的目录进行列表等。

以上提到的 VTOC 是一个顺序数据集，记录了所在存储设备上所有数据集的属性。这些属性包括数据集的名称、长度、记录格式、定位信息、使用情况、建立日期等。下面作业便调用 IEHLIST 程序对 VTOC 进行了列表。

```
//LSVTOC   JOB  , SIMON, NOTIFY=&SYSUID
//LISTVT   EXEC  PGM=IEHLIST
//SYSPRINT  DD   SYSOUT=*
//DDA    DD
//   DISP=OLD, UNIT=SYSDA, VOL=SER=ACA0
//DDB    DD
//   DISP=OLD, UNIT=SYSDA, VOL=SER=VVTTRC
//SYSIN    DD  *
 LISTVTOC  FORMAT, VOL=SYSDA=ACA0
 LISTVTOC  FORMAT, VOL=SYSDA=VVTTRC,
 DSNAME=(ES10.MVS07.LIST, ES15.VM08.LIST)
```

以上名为 DDA 与 DDB 的 DD 语句用于分配给 LISTVTOC 操作的各个 FORMAT 包。第一条控制语句 LISTVTOC 请求编辑 VTOC 的一个 FORMAT 包 ACA0。第二条 LISTVTOC 请求编辑两个数据集的 VTOC 的 FORMAT 包。这两个数据集分别为 ES10.MVS07.LIST 和 ES15.VM08.LIST。

对于以上调用 IEHLIST 的作业中的控制语句，有以下两点需要注意。

- 控制语句中 VOL 参数的第一个等号之后的内容为前面 DD 语句中 UNIT 参数所指定的内容。第二个等号后的内容则为前面 DD 语句中 SER 子参数所指定的内容。
- 控制语句中的 DSNAME 不可效仿 JCL 语句中的参数而简写为 DSN。

14.6 JCL 的过程

JCL 的过程相当于包含一段 JCL 语句的模块。该模块可被其他 JCL 作业调用，并可以进行重载。通常可将经常用到的一段 JCL 语句编入一个过程，以便于其他作业的编写。JCL 的过程通常分为两大类别，分别如下。

- 流内过程: 直接在 JCL 作业中编写。其中使用 PROC 语句表示该过程的开始，使用 PEND 语句表示该过程的结束。

- 编目过程：存放在分区数据集的一个成员中。其中 PROC 语句在该过程中是可选的，而 PEND 语句通常则不应用于该过程。

一个作业中最多只能包含有 15 个流内过程。流内过程位于 JOB 语句之后，并且，流内过程是必须要求有过程名。例如，以下为一段包含有流内过程的作业。

```
//INSPRC  JOB  , WENDY, CLASS=A, NOTIFY=&SYSUID
//MYPRC  PROC                    /*MYPRC 为该流内过程的过程名*/
//PSTEP1  EXEC  PGM=IEFBR14
//PDD1    DD   DSN=ADCDA.PROC.TEST, DISP=(, CATLG),
//     UNIT=3390, VOL=SER=WORK04, SPACE=(TRK, (2, 1)),
//     DCB=(RECFM=FB, LRECL=120, BLKSIZE=1200)
//PDD2    DD   DSN=ADCDA.PROC.DEL01, DISP=(OLD, DELETE)
//PDD3    DD   DSN=ADCDA.PROC.DEL02, DISP=(OLD, DELETE)
//   PEND
//STEP1   EXEC  PROC=MYPRC       /*本作业步实现对流内过程的调用*/
//
```

对于编目过程，其过程名是可以默认的。当默认过程名时，作业中的 EXEC 语句通过指定该过程所在分区数据集的成员以对其进行调用。

过程在调用中是可以进行重载的，这一点实际上过程最主要的用途。通常，用户可以通过如下方式对过程进行重载。

- 在过程中增加新的 DD 语句。
- 增加、覆盖或置空 EXEC 语句及 DD 语句中的参数。
- 为过程中的符号参数赋值。

需要注意的是，重载后的过程只适用于对其调用的本作业步，原过程是不被改变的。例如，如果编目过程 PRCM 中的内容如下。

```
//PRCM   PROC
//PSTP1   EXEC  PGM=PONE
//PDD2    DD    DSN=PDATA, DISP=SHR
//PSTP2   EXEC  PGM=PTWO, PARM='MF,COBOL'
```

则以下作业对该过程调用的同时，对其进行了重载。包括在 PSTP1 作业步中新增一条 DD 语句，为该作业步的 EXEC 语句新增一个 ACCT 参数，将名为 PDD2 的 DD 语句中的 DSN 参数内容覆盖为 JDATA2，将名为 PSTP2 的 EXEC 语句中的 PARM 参数置空。相应作业如下。

```
//MODPRC  JOB  , WENDY, CLASS=A, NOTIFY=&SYSUID
//STEP1   EXEC  PRCM,
//ACCT.PSTP1='TEST',
//PSTP1.PDD1  DD  DSN=JDATA1, DISP=SHR,
//PSTP1.PDD2  DD  DSN=JDATA2,
//PARM.PSTP2=
```

不过，对于过程中 EXEC 语句里的 PGM 参数是不允许通过以上方式进行重载的。可以使用过程中的符号参数来实现类似功能。过程中的符号参数也是一个十分重要的概念。在过程中是使用“&”符号表示符号参数的。例如，以下为一个包含有符号参数的过程。

```
//SYMP  PROC
//PSTP1  EXEC  PGM=&SPGM
```

```
//PDD    DD    DSN=&SDATA, DISP=&SDISP,
//   UNIT=3390, VOL=SER=&SVOL,
//   SPACE=(TRK, (3, 1))
//  DCB=(RECFM=&SRCF, LRECL=80, BLKSIZE=1600)
```

以上过程中共有 5 个符号参数，均由“&”符号进行表示。对以上过程进行调用时，必须对该过程中的每一个符号参数进行赋值。例如，以下为一段调用该过程的作业。

```
//SYMBOL  JOB  , WENDY, CLASS=A, NOTIFY=&SYSUID
//STEP1   EXEC  PROC=SYMP
//ACCT.PSTP1='TEST',
//   SPGM=IEFBR14,
//   SDATA=ADCDA.NEW.PS,
//   SDISP=(, CATLG),
//   SVOL=WORK01,
//   SRCF=FB
//
```

同时，也可在过程的开头对该过程中的符号参数赋予初值。当对该过程进行调用时，调用的作业可以不对符号参数赋值，此时符号参数将为过程中所赋的初值。例如，可在以上过程中对其中的 5 个符号参数赋予初值，形式如下。

```
//NEWSYMP  PROC  SPGM=IEFBR14, SDATA=ADCDA.NEW.PS,
//     SDISP=(, CATLG), SVOL=WORK01, SRCF=FB
//PSTP1  EXEC  PGM=&SPGM
//PDD    DD    DSN=&SDATA, DISP=&SDISP,
//   UNIT=3390, VOL=SER=&SVOL,
//   SPACE=(TRK, (3, 1))
//  DCB=(RECFM=&SRCF, LRECL=80, BLKSIZE=1600)
```

此时对以上过程进行调用的相应作业可以写作如下。

```
//NEWSYM  JOB  , WENDY, CLASS=A, NOTIFY=&SYSUID
//STEP1   EXEC  PROC=NEWSYMP
//
```

14.7 通过 JCL 管理 VSAM 数据集

在 JCL 中主要是通过 IDCAMS 实用程序实现对 VSAM 数据集的管理的。IDCAMS 实用程序是一个功能十分强大的程序。该实用程序中常用的控制语句及其所对应的功能如下。

- DEFINE：创建一个 VSAM 数据集。
- ALTER：修改 VSAM 数据集的属性。
- DELETE：删除 VSAM 数据集。
- REPRO：实现 VSAM 数据集的复制。
- LISTCAT：对编目信息进行列表。
- PRINT：对数据集进行列表。
- IMPORT：实现载入操作。
- EXPORT：实现卸载操作。

下面主要讲解如何使用 JCL 中的 IDCAMS 实用程序实现对 VSAM 数据集的创建和复制。

根据前面章节的讲解，可以知道 VSAM 数据集主要分为 5 类。这 5 类 VSAM 数据集分别是 LDS、ESDE、RRDS、KSDS 以及 VRRDS。以下作业便调用 IDCAMS 创建了一个 LDS。

```
//ADCDV01  JOB  ,ERIC, CLASS=C, NOTIFY=&SYSUID
//DFLDS    EXEC   PGM=IDCAMS
//SYSPRINT  DD  SYSOUT=*
//SYSIN     DD   *
  DEFINE   CLUSTER  -
             (NAME (SAMPLE.LDS)  -
             LINEAR  -
             VOLUMES (VSPAK5)  -
             TRACKS (3  3))
  LISTCAT   ENTRIES  -
              (SAMPLE.LDS)  ALL
```

该作业定义的 VSAM 数据集名为 SAMPLE.LDS，通过 LINEAR 参数指明了其为 LDS。以下作业则通过类似方式创建了一个名为 SAMPLE.ESDS 的 ESDS。相应作业如下。

```
//ADCDV02  JOB  ,ERIC, CLASS=C, NOTIFY=&SYSUID
//DFESDS  EXEC   PGM=IDCAMS
//SYSPRINT  DD  SYSOUT=*
//SYSIN     DD   *
  DEFINE   CLUSTER  -
             (NAME (SAMPLE.ESDS)  -
             NIXD  -
             VOLUMES (VSPAK5)  -
             TRACKS (5  5))
  LISTCAT   ENTRIES  -
              (SAMPLE.ESDS)  ALL
```

可以看到，以上两个作业中创建 LDS 和 ESDS 的方式基本类似。所不同的是创建 LDS 时是通过参数 LINEAR 指定的，而创建 ESDS 时则是通过参数 NIXD 指定的。创建 RRDS 的作业则相对要复杂一些，该作业如下。

```
//ADCDV03  JOB  ,ERIC, CLASS=C, NOTIFY=&SYSUID
//DFESDS  EXEC   PGM=IDCAMS
//SYSPRINT  DD  SYSOUT=*
//SYSIN     DD   *
  DEFINE   CLUSTER  -
             (NAME (SAMPLE.RRDS)  -
             NUMBERED  -
             VOLUMES (VSPAK5)  -
             RECORDSIZE (80  80)  -
             CISZ(4096))  -
            DATA  -
             (NAME(SAMPLE.RRDS.DATA)  -
             CYLINDERS(3  1))
  LISTCAT   ENTRIES  -
              (SAMPLE.RRDS)  ALL
```

需要注意的是，以上作业中 RECORDSIZE 中的第一个子参数表示的是数据集记录的平均长度。而第二个子参数则表示记录的最大长度。由于 RRDS 中的记录都是等长的，因此这两个参数数值相等。而该作业中的 CISZ 参数则是用于表示数据集的 CI 大小。

VRRDS 的创建方式和 RRDS 的基本类似。不过在创建 VRRDS 时，RECORDSIZE 中的

两个子参数数值是不相等的。KSDS 是在实际中用得最多的一种 VSAM 数据集。以下作业通过分别指定文件、数据以及索引 3 部分的内容创建了一个 KSDS。

```
//ADCDV04  JOB  ,ERIC, CLASS=C, NOTIFY=&SYSUID
//DFKSDS  EXEC   PGM=IDCAMS
//SYSPRINT  DD  SYSOUT=*
//SYSIN     DD   *
  DEFINE   CLUSTER  -
             (NAME (SAMPLE.KSDS01)  -
             INDEXED  -
             VOLUMES (VSPAK5))  -
            DATA  -
             (NAME(SAMPLE.KSDS01.DATA)  -
             FREESPACE(20 10)  -
             RECORDSIZE(100  100)  -
             CISZ(4096)  -
             CYL(3  1)  -
             KEYS(7  20))  -
            INDEX  -
             (NAME (SAMPLE.KSDS01.INDEX))
  LISTCAT   ENTRIES  -
              (SAMPLE.KSDS01)  ALL
```

此外，在创建 KSDS 时也可仅定义文件项目，并将所有相关的参数都在文件项目中指定。此时系统将自动根据其属性定义其余部分中的内容。例如，以下这种定义方式也是正确的。

```
//ADCDV05  JOB  ,ERIC, CLASS=C, NOTIFY=&SYSUID
//DFKSDS  EXEC   PGM=IDCAMS
//SYSPRINT  DD  SYSOUT=*
//SYSIN     DD   *
  DEFINE   CLUSTER  -
             (NAME (SAMPLE.KSDS02)  -
             INDEXED  -
             VOLUMES (VSPAK5)  -
             FREESPACE(20 10)  -
             RECORDSIZE(120  120)  -
             CISZ(4096)  -
             CYL(2  1)  -
             KEYS(8  5))
  LISTCAT   ENTRIES  -
              (SAMPLE.KSDS01)  ALL
```

通过 IDCAMS 实用程序中的 REPRO 控制语句可以实现对 VSAM 数据集的复制。以下作业便将 SAMPLE.INPUT.KSDS 中的内容复制到了新的数据集 SAMPLE.OUTPUT.KSDS 中。

```
//ADCDV0? JOB  ,ERIC, CLASS=C, NOTIFY=&SYSUID
//CPVS   EXEC   PGM=IDCAMS
//SYSPRINT  DD  SYSOUT=*
//INVS    DD  DSN=SAMPLE.INPUT.KSDS, DISP=OLD
//OUTVS  DD  DSN=SAMPLE.OUTPUT.KSDS,
//        DISP=(NEW, CATLG), DATACLAS=DCKSDS
//SYSIN   DD  *
  REPRO  INFILE(INVS)  OUTFILE(OUTVS)
```

14.8 本章回顾

本章讲解了与当前 COBOL 的主要运行环境——大型机所相关的 JCL 语言。从事大型机方面的工作，无论是使用 COBOL 进行开发还是系统管理维护，都是必须要求了解 JCL 的。可以说，JCL 是进入大型机领域所要掌握的最基本的知识。

本章首先介绍了 JCL 的基本概念。关于 JCL 的基本概念，重点需要理解作业与作业步的概念，掌握 JCL 的语法规则，理解 JCL 语句中关键字参数与位置参数的概念。

本章其后依次介绍了 JCL 中的 3 种基本语句。其中 JOB 语句用于定义一个作业，EXEC 语句用于执行一个作业步，DD 语句用于定义数据集及设置数据集的相关属性。每条语句中都涉及到很多的参数。学习这部分内容，重点需要理解这些参数的用法及意义。

本章接下来介绍了 JCL 中一些常用的实用程序。这部分内容是本章的重点。学习该部分内容，重点需要理解各项实用程序的功能，并掌握在 JCL 作业中调用这些程序的格式。

在讲解完 JCL 的实用程序之后，本章介绍了 JCL 中过程的概念及用法。对于 JCL 中的过程，需要理解流内过程和编目过程的概念，掌握如何在作业中调用并重载过程，掌握过程中符号参数的用法。

本章最后介绍了如何通过 JCL 管理 VSAM 数据集。在 JCL 中主要是通过 IDCAMS 实用程序实现对 VSAM 数据集的创建、复制、列表及删除等功能的。此处重点需要掌握如何使用 IDCAMS 创建各种类型的 VSAM 数据集。

第 15 章 DB2 扩展

运行于大型机上的数据库通常有两种，DB2 和 IMS。其中 DB2 比 IMS 更常用。由于 COBOL 程序主要用于处理大量的商务数据，因此会涉及到数据库的应用。同时，DB2 本身也可以作为一块独立的行业领域。

15.1 基本概念

DB2 作为一种独立的数据库，与 Oracle 数据库是属于一个概念范畴的。DB2 不仅可以用于大型机，也可以用于其他一些操作系统平台，如 UNIX、Windows 等。DB2 本身的知识十分庞大，此处主要对其在大型机上的应用进行简要的讲解。

15.1.1 关系数据库的概念

首先需要明确的是，DB2 数据库在结构和原理上是属于关系数据库的。因此，在学习 DB2 之前，有必要先了解一下关系数据库的概念。

关系数据库也称关系型数据库，是数据库结构原理发展阶段的第三个模型。关系型数据库的前一个模型为层次型数据库。IMS 数据库便属于层次型数据库。下面首先给出一个层次型数据库的基本结构，以便通过对比了解关系型数据库的特点。层次型数据库结构如图 15.1 所示。

由此可见，层次型数据库的结构是通过层级之间的指针建立数据的关系。层次型数据库的结构实际上与数据结构中的树比较类似。

层次型数据库在数据的管理与查找上并不是很方便。同时，在层次型数据库中也会存在比较多的数据冗余。而关系型数据库相对于层次型数据库在各方面都有了很大的改善。在关系型数据库中主要是以表（Table）为单位组织数据的。需要注意的是，此处所说的表是数据库上的概念，同前面讲解的 COBOL 中的表是不同的。关系型数据库中表的结构如图 15.2 所示。

可以看到，关系型数据库中的表实际上是由不同的行和列所组成的。关于表中的行和列分别具有以下属性。

- 关系数据库表中的行：对应一条完整的数据记录。各条数据记录都包含有相同数目的属

性，每一属性对应该行中的一个单元格。

- 关系数据库表中的列：对应各条数据记录中相同类型的属性。每一列中所包含的各个数据项分别描述不同行中数据记录的相关属性。

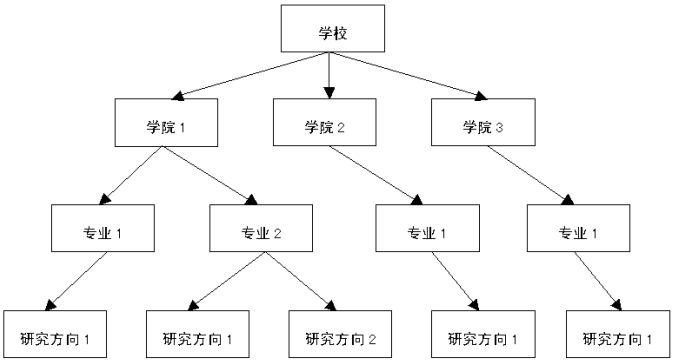

图 15.1　层次型数据库结构

学院编号	学院名称	专业设置	院系主任	学生人数
0122	计算机学院	计算机科学与技术，	黄贝贝	504
0130	机械学院	机械设计与自动化	张玉	480
0232	电气学院	电气工程	国江	263
0245	能源学院	热能与动力工程	谢德华	472

图 15.2　关系型数据库中的表

在关系数据库的表中，还有一个很重要的概念：主关键字（Primary Key）的概念。主关键字是每条记录中的一个属性，有时也称作主键或主码。主关键字是人为指定的，记录中的任何属性都可以作为主关键字。不过，通常是将比较有明显特征的属性作为主关键字的。例如，对于以上示例中的表，通常是将学院编号作为主关键字。

关系数据库中包含有多张表，每张表之间通过其中的属性数据项建立直接或间接的联系。例如，对应以上数据库中的表，如图 15.3 所示，为其中的另一张表。

课程编号	学院编号	课程名称	任课教师	课程学分
0A40	0122	系统结构	林益	4
0B12	0130	机械制图	邱峰	5
1A35	0122	组成原理	高建生	5
0C07	0245	工程化学	徐安	3

图 15.3　关系数据库中的另一张表

对于这张表，通常将课程编号作为主关键字。同时，注意到在该表中也存在学院编号这

一属性。由于学院编号为前一张表中的主关键字，因此学院编号属性在此处属于外来关键字（Foreign Key）。通过该外来关键字，可以在以上两张表之间建立起联系。

建立在表的基础上，在关系数据库中还存在视图（View）和索引（Index）的概念。视图相当于一张虚拟的表。通常将多表连接查询的结果作为视图，以方便数据查询。而索引则是直接建立在表上的，相当于一个目录，也是主要用于方便数据查询的。

对数据库的处理过程是以事务为单位进行的。事务是指作为单个逻辑工作单元执行的一系列操作。这些逻辑工作单元需要具有以下 4 个属性，并统称为 ACID 特性。

- 原子性（Atomicity）：事务中的所有操作要么全做，要么全不做。
- 一致性（Consistency）：事务执行的结果必须是使数据库从一个一致性状态，变为另一个一致性状态。
- 隔离性（Isolation）：一个事务的执行不能被其他事务干扰。
- 持续性（Durability）：一个事务一旦提交，对数据库中数据的改变就应该是永久的。

最后，对于关系数据库，还应该大致了解一下关于范式的概念。通常情况下在关系数据库中共有 5 种范式，目前最高可达到 6 种。不过一般情况下，数据库只需满足第三范式就足够了。下面对前 3 种范式分别介绍如下。

- 第一范式（1NF）：指数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值。例如，对于以上第一张表而言，是不能把学院编号和学院名称归并为一列的。
- 第二范式（2NF）：满足第二范式的数据划分首先必须满足第一范式。此外，第二范式同时要求实体的属性完全依赖于主关键字。例如，若将第二张表中的课程学分作为主关键字，则该表将不满足第二范式。因为存在着相同学分的课程，课程名称等属性并非根据课程学分而定的。
- 第三范式（3NF）：满足第三范式的数据划分首先也必须满足第二范式。此外，第三范式同时要求实体中的属性不依赖于其他非主属性。例如，在第二张表中，是不能将第一张表中的非主属性如院系主任等包含进来的。

可以看到，范式的划分主要是根据不同的数据依赖而来的，并且层级越高的范式要求越严格。在数据库设计中，遵照范式进行的目的是为了减少或避免以下现象的发生。

- 数据冗余。
- 插入异常。
- 更新异常。
- 删除异常。

15.1.2 DB2 简介

DB2 是大型机上主要应用的一种关系数据库。学习 DB2，首先应该了解 DB2 中主要包含有哪些系列的产品。其中每一个 DB2 产品对应于一个不同的系统平台。这些产品分别如下。

- OS/390 与 z/OS 平台下的 DB2。
- VSE 与 VM 平台下的 DB2。
- iservers 下的 DB2。
- Windows 平台下的 DB2。
- Unix 平台下的 DB2。

以上前两种产品都属于运行于大型机上的产品。大型机的型号可以为 S/390 或者 z Series。这两种产品实际上是 DB2 应用得最多的产品，尤其是第一种。

对 DB2 数据库进行访问的模式通常为客户机/服务器模式，即通常所说的 C/S 模式。其中服务器主要包括 SQL 应用程序接口（API），以及数据库引擎（Database Engine）。关于 SQL，为数据库中最基本的一项操作语言，将在后面的章节中对其进行详细的讲解。如图 15.4 所示，反映了对 DB2 进行访问的客户机/服务器模式。

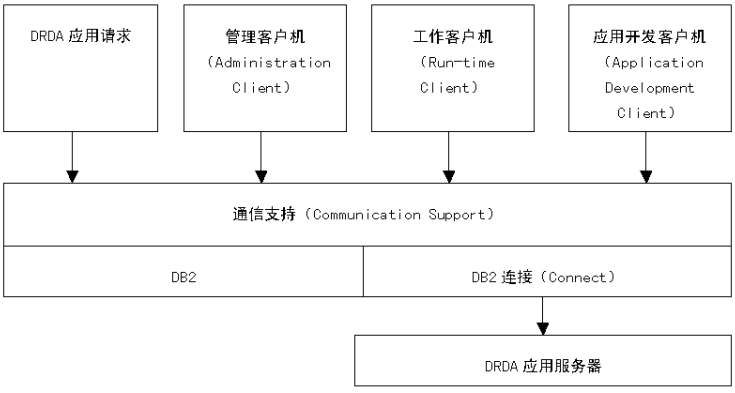

图 15.4　DB2 的 C/S 模式图

上图中的 DRDA（Distributed Relational Database Architecture）是指分布式关系数据库体系结构。DB2 正是属于分布式关系数据库。

对于应用于大型机中的 DB2，需要了解其物理环境。如图 15.5 所示，为 DB2 在大型机中的物理环境，为 DB2 学习中最常见到的一个图。

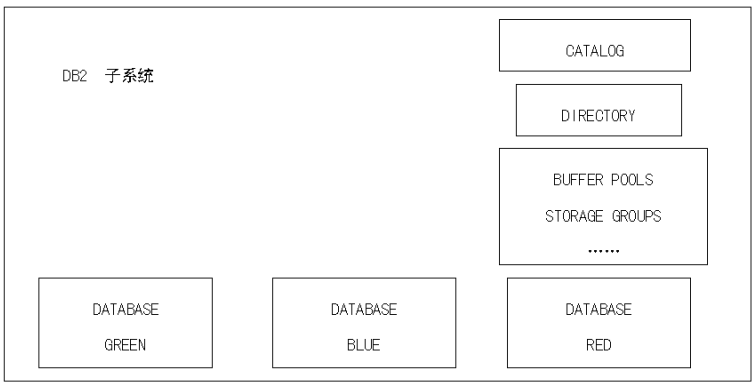

图 15.5　DB2 在大型机中的物理环境

此外，DB2 作为数据库，还应了解关于数据库管理系统的基本概念。数据库管理系统即 DBMS（Database Management System）。不同数据库的 DBMS 功能是不同。关于 DB2 的 DBMS，主要有以下几项功能。

- 对关系数据库中表的基本管理（Tables）。
- 提供优化功能（Optimizer）。

- 提供锁的管理功能（Lock Manager）。
- 可进行日志记录（Logging）。
- 提供持续操作（Continuous Operation）。
- 提供安全性（Security）。
- 提供数据的完整性（Integrity）。
- 提供数据的可恢复性（Recovery）。

从事关于 DB2 不同方面的工作，将会用到 DB2 不同方面的内容。例如，对于数据库管理员 DBA 和应用程序开发人员所主要接触到的 DB2 的内容就不同。对于开发人员而言，主要需要关注 DB2 以下这几方面的内容。

- DB2 SQL 性能分析（SQL Performance Analyzer）。
- DB2 表的编辑（Table Editor）。
- DB2 绑定管理（Bind Manager）。
- DB2 路径检查（Path Checker）。
- DB2 实用程序（Utilities）。
- DB2 请求监控（Query Monitor）。
- DB2 管理工具（Administration Tool）。
- DB2 Web 请求工具（Web Query Tool）。

最后，利用 DB2 数据库进行开发时，通常需要遵循一定的步骤进行。遵循这些步骤，主要是为了规范化开发流程，同时也可减少一定的项目预算风险。这些步骤依次如下。

（1）建立测试环境。
（2）建立 SQL 原型。
（3）绑定 Packages 和 Plans。
（4）编写 SQL 语句。
（5）确认目标变化。
（6）查询目标属性。
（7）应用性能分析。
（8）编辑表中数据。
（9）载入实际数据。

注意 此处是指整个利用 DB2 进行开发的流程，而不是指单纯地在 COBOL 程序中如何访问 DB2。如何在 COBOL 的源代码中访问 DB2，将在后面的章节中详细讲解。

15.1.3 DB2 的组织结构及创建步骤

通过前面的讲解可以知道，DB2 数据库对数据的组织与管理是以表为基本单位的。而在 DB2 系统中，表是存放在表空间（Table Space）之中的。一块表空间可以有一个到多个表。表空间则存放在数据库（database）之中。一个数据库也可以有一个到多个表空间。需要注意的是，此处所说的数据库是一个数据空间上的概念。如图 15.6 所示，为此结构特征。

同时，DB2 在硬件平台上还将涉及到存储组（Storage Group）和卷（Volume）。结合前面所讲解的视图和索引的概念，DB2 及其硬件平台上中各单元的关系，如图 15.7 所示。

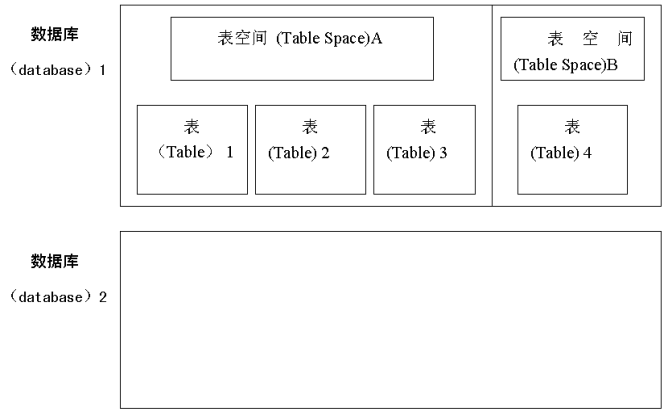

图 15.6　DB2 的数据组织结构

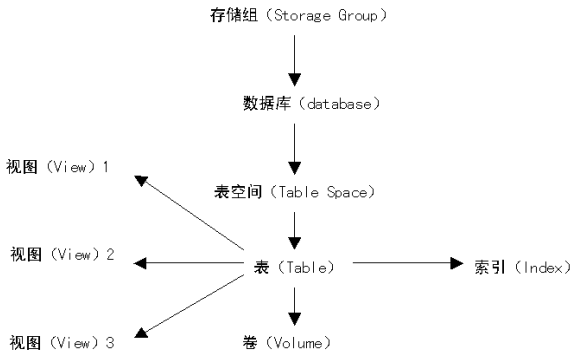

图 15.7　DB2 及硬件平台上各单元的关系

关于上图所示的存储组，属于一组直接访问存储设备（DASD）上的卷。这些 DASD 卷用于分配 DB2 中的表空间及索引空间。

在 DB2 中，所有数据库服务进程的集合被称作一个 Instance。一个 Instance 中有多个数据库，以及一个数据库管理配置文件（DBM CONFIG FILE）。关于 Instance，主要有以下几种操作。

- db2icrt: 建立一个 instance（并不激活）。
- db2ilist: 将多个 instance 进行列表。
- db2idrop: 删除一个 instance。
- db2start: 启动一个 instance（进行激活）。

同系统的 IPL（Initial Program Loading）类似，DB2 数据库的创建也是需要经历一定的步骤的。这些步骤依次如下。

（1）创建数据库存储空间。

（2）建立数据字典（catalog）及数据库恢复日志（recovery log）。

（3）建立数据库配置文件（configuration file），并配置相应的默认值。

（4）绑定（bind）数据库实用程序（utilities）到数据库中。

（5）定义 3 个特殊的表空间：SYSCATSPACE、TEMPSPACE、USERSPACE。需要注意的是，对于这 3 个表空间，用户只能管理其中的 USERSPACE。

（6）将数据库编目在本地数据字典与系统数据字典上，用于用户查询。其中本地数据字典在 DB2 所设的硬盘上。

（7）分配代码集（code set）、区域信息（territory）及比较对照顺序（collating sequence）。其中区域信息如当地的日期、时间等。

（8）建立 SYSCAT、SYSFVN、SYSIBM、SYSSTAT 及 scheme。

（9）建立优先级（privilege）。

关于上面提到的 scheme，主要需要了解 scheme name 的概念。关于 scheme name，是指实体名称中的高位标志字段（high-level qualifier）。该字段主要用于标志整个名称。当建立一个用户名时，会产生一个同名的 scheme。例如，以下将 scheme 设置为了“PAYROLL”。

```
SET  CURRENT  SCHEME  =  'PAYROLL'
```

在此设置的基础上，以下第一条 SQL 语句在实际执行中相当于第二条 SQL 语句。

```
SELECT  *  FROM  EMPLOYEE
SELECT  *  FROM  PAYROLL.EMPLOYEE
```

最后，在大型机中实际运行含有 DB2 的程序，并查看运行结果，主要是通过 SPUFI 产品进行的。SPUFI（SQL Processor Using File Input）即使用到文件输出的 SQL 处理器。使用 SPUFI 处理 DB2 程序的流程如图 15.8 所示。

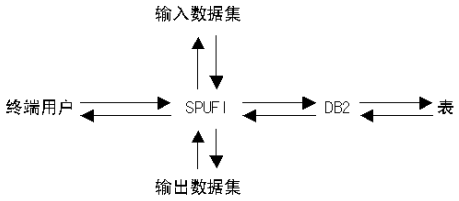

图 15.8　SPUFI 在 DB2 中的应用

15.2　DB2 的基本应用

DB2 在 COBOL 中的应用与普通的 COBOL 编码是不同的。涉及到 DB2 的代码，实际上是以 COBOL 为宿主语言，并嵌入到 COBOL 代码中编写的。同时，在编码之后，将其编译连接为可执行程序，也是需要经历特定的过程的。以下分别对这两方面的内容进行讲解。

15.2.1　DB2 在 COBOL 中的编码

由于 DB2 作为关系数据库，因此对其操作仍然是通过 SQL 语句进行的。SQL（Structured Query Language）即结构化查询语言的意思，任何关系数据库通常都会用到。关于 SQL 语句，将在后面的章节中详细讲解。此处只考虑在 COBOL 中是如何利用 SQL 语句访问和操作 DB2 数据库的。在 COBOL 中调用 SQL 语句的格式如下。

```
EXEC  SQL
     SQL  statements
END-EXEC.
```

可以看到，在 COBOL 中调用 SQL 语句主要是通过 EXEC SQL 和 END-EXEC 对其进行

指定的。这两个命令之间便为通常意义的 SQL 语句。此处的 SQL 语句实际上可以看作是以 COBOL 为宿主语言，并嵌入到 COBOL 源码中编写的。

同时，还应了解 SQLCA 的基本概念。SQLCA（SQL Communication Area）即 SQL 通信区域的意思。SQLCA 主要用于实现 COBOL 应用程序和 DB2 数据库之间的交互。当 COBOL 程序中需要访问或操作 DB2 数据库时，需要在工作存储节中将 SQLCA 包含进来。并且，还应将数据记录也相应地包含进来。相关代码如下。

```
......
DATA  DIVISION.
WORKING-STORAGE  SECTION.
  EXEC  SQL  INCLUDE  SQLCA    END-EXEC.
  EXEC  SQL  INCLUDE  TESTRCD  END-EXEC.
......
```

执行 SQL 语句后，会得到一个称作 SQLCODE 的返回码。SQLCODE 存放于 SQLCA 之中。当 SQLCODE 为 0 或者 100 时，表示该条 SQL 语句执行成功。当 SQLCODE 为负数时，表示该条 SQL 语句在执行时出现错误。

实际上，COBOL 程序中关于 DB2 方面的代码几乎都是涵盖在 EXEC SQL 和 END-EXEC 之中的。以下为一段应用到 DB2 的完整的 COBOL 程序代码。此处仅需通过该段代码对 DB2 在 COBOL 中的应用有一个大致的印象。相应代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.    PGMDB99.
 AUTHOR.        XXX.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
   EXEC  SQL  INCLUDE  SQLCA  END-EXEC.
   EXEC  SQL  INCLUDE  EMP99RCD  END-EXEC.
 01  ERROR-MESSAGE.
     03  SQL-CODE  PIC  X(10).
     03  ERR-LEN   PIC  S9(4)  USAGE COMP
                               VALUE  IS  +560.
     03  E-MESS1   PIC  X(80).
     03  E-MESS2   PIC  X(80).
     03  E-MESS3   PIC  X(80).
     03  E-MESS4   PIC  X(80).
     03  E-MESS5   PIC  X(80).
     03  E-MESS6   PIC  X(80).
     03  E-MESS7   PIC  X(80).
     03  E-MESS8   PIC  X(80)  VALUE  IS  'FIX  AND  RETRY'.
     03  REC-LEN   PIC  S9(8)  USAGE  COMP
                               VALUE  IS  +80.
 01  WRKFLDS.
     03  RECKEY    PIC  X(6)  VALUE  IS  '000000'.
     03  DBERR-SW  PIC  X(1)  VALUE  IS  'N'.
     03  K          PIC  S9(4)  COMP.
     03  OUTPUT-MSG  PIC  X(80).
*
 PROCEDURE  DIVISION.
```

```
       EXEC  SQL  DECLARE  EMP  CURSOR  FOR
                  SELECT   *
                  FROM     TSOUD99.EMPLOYEE
       END-EXEC.
       EXEC  SQL  OPEN  EMP  END-EXEC.
       MOVE  SQLCODE  TO  SQL-CODE
       IF  SQLCODE  <  0
            DISPLAY  'I1:'  SQL-CODE
            PERFORM  DBERR-RETURN
       END-IF.
   *****  START  OF  LOOP  *****
       PERFORM  VARYING  K  FROM  1  BY  1
                UNTIL  K  IS > 100
         MOVE  LOW-VALUES  TO  DCLEMPLOYEE
         EXEC   SQL   FETCH  EMP  INTO :DCLEMPLOYEE
         END-EXEC
         MOVE  SQLCODE  TO  SQL-CODE
         IF  SQLCODE  =  100
                DISPLAY  'I2:' > SQL-CODE
                DISPLAY  'END  OF  DATA'
                GO TO  CLOSING
         ELSE
                DISPLAY  K '=' DCLEMPLOYEE
         END-IF
       END-PERFORM.
   *****   END  OF  LOOP  *****
    CLOSING.
       EXEC  SQL  CLOSE  EMP  END-EXEC.
       IF  SQLCODE < 0
          GO TO  DBERR-RETURN
       END-IF.
       DISPLAY  'PROGRAM  ENDED  NORMALLY!'.
       STOP  RUN.
    DBERR-RETURN.
       MOVE  SQLCODE  TO  SQL-CODE.
       DISPLAY  SQL-CODE.
       DISPLAY  'PROGRAM  ABEND!'.
       CALL  'DSNTIAR'  USING  SQLCA  ERROR-MESSAGE  REC-LEN.
       DISPLAY  ERROR-MESSAGE.
       GO BACK.
```

以上程序首先定义了一个 DB2 中的游标（CURSOR）。该游标的名称为 EMP。程序通过该游标将输入数据集 TSOUD99.EMPLOYEE 中的数据记录选择并输出到显示屏上。数据的输入及输出都需要借助 SPUFI 产品进行。

此外还需注意的是，在嵌入到 EXEC SQL 与 END-EXEC 之间的 DB2 相关变量是不能有中划线的。不过这些变量却可以包含下划线。这点同 COBOL 正好相反。COBOL 中的变量可以包含中划线，但却不可以有下划线。

15.2.2 含 DB2 的 COBOL 编译过程

涉及到 DB2 的 COBOL 程序的编译过程相对单纯的 COBOL 程序要复杂一些。如图 15.9 所示，反映了其编译及连接的大体过程。

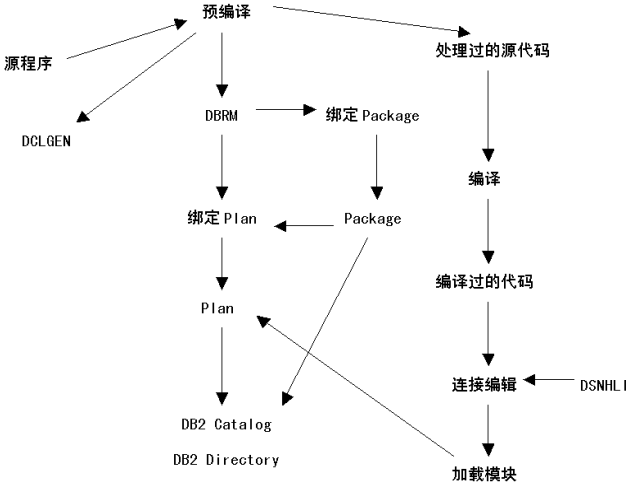

图 15.9　涉及 DB2 的 COBOL 程序编译过程

实际上，在应用程序中主要是通过 SQL 语句访问和操作 DB2 数据库的。因此，此处所说的涉及到 DB2 的 COBOL 程序通常也可以看作是含有 SQL 语句的 COBOL 程序。由图 15.9 可知，对此类程序在编译前，先要进行一个预编译。预编译所起的作用是将程序的源代码分为以下两部分，并对其分别进行不同的处理。

- SQL 语句部分：使用 DBRM 对其进一步处理。
- COBOL 代码部分：使用 COBOL 编译器对其进一步处理。

关于 DBRM（Database Request Module），即数据库请求模块的意思。DBRM 实现的功能主要有以下两条。

- 绑定 Package。
- 绑定 Plan。

此处所说 Package 即 SQL 语句处理之后所生成的结果，存放在 DB2 之中。实际上，Package 同 COBOL 代码在编译连接后，生成的加载模块（Load Module）是比较类似的。Package 的集合在 DB2 中被称作 collection。而与 Package 相应的 Plan 正是反映了 DB2 搜索 collection 的顺序。

上图所示的 DB2 Catalog 在前面的 DB2 物理环境（图 15.5）中也曾出现过。DB2 Catalog 即 DB2 的数据字典。所谓数据字典，是一张系统表，用来存放数据库所用的有关信息。对于用户而言，这张表是只读的。

DB2 Directory 的功能与 DB2 Catalog 在某种意义上比较类似。不过 DB2 Directory 的执行性能要比 DB2 Catalog 更加好。DB2 Directory 是以 VSAM 文件的形式组织存放数据的。在 DB2 Directory 中共包含 5 个结构（Structure），分别如下。

- SYSUTILX：存放实用程序（Utilities）。
- SYSLGRNX：存放活动的（active）及归档的（archive）日志。
- SPT01：存放 Plan。
- SCT02：存放 Package。
- DBD01：存放描述数据库的相关信息。

此外，当 DB2 Catalog 和 DB2 Directory 中的数据不一致时，需要使用 REPAIR DBD 语

句进行同步。当将二者进行同步时，将以 DB2 Directory 中的数据为准。

上图中的 DCLGEN（Declarations Generator）即声明生成程序。DCLGEN 在 DB2 中的应用主要有以下两条。

- 产生通过 SQL 语句声明的表。
- 产生通过 COBOL 代码声明的对应于表或视图的主变量。

上图中的 DSNHLI 是 COBOL 程序与 SQL 语句的接口。DSNHLI 中含有相关参数，并在 COBOL 代码的连接编辑（Link Editor）步骤中参与进来。

15.3 常用 SQL 语句

前面多次提到，在 COBOL 应用程序中对 DB2 数据库进行操作最主要是通过 SQL 语句进行的。SQL 语句通常可以分为 3 种类型，分别为 DML、DDL 以及 DCL。下面分别对这 3 种类型的 SQL 语句予以讲解。

15.3.1 DML 类别的 SQL 语句

DML（Data Manipulation Language）即数据操作语言的意思。DML 类别的 SQL 语句通常有以下几种。

- SELECT 语句
- INSERT 语句
- UPDATE 语句
- DELETE 语句

DML 类别的 SQL 语句是作为应用程序开发人员用得最多的一类 SQL 语句。下面分别对这 4 种语句进行讲解。

1．SELECT 语句

SELECT 语句主要用于数据的查询。SELECT 语句是以上 4 种 DML 类别的 SQL 语句中用得最多的一条，也是在整个 SQL 语句里用得最多的一条。下面给出 DB2 中的一张表，设表名为 Q.COURSE，如图 15.10 所示。

CODE	NAME	INSTRUCTOR	DAYS
ES07	JCL	Simon	2
ES15	z/OS Facilities	Daisy	3
SS06	SMS	Tony	3
SS83	VSAM and AMS	Jason	2
ES28	JES2	Simon	2
ES19	RACF	Tony	2

图 15.10 Q.COURSE 表

基于以上表中数据，使用 SELECT 语句对其最基本的操作是全选表中所有的数据。实现全选的语句如下。

```
SELECT  *
FROM   Q.COURSE
```

在实际应用中，更多的情况是只选择表中的部分数据，如特定数据记录中的特定属性。例如，以下 SELECT 语句将选择由 Simon 任课的所有课程的名称。

```
SELECT  NAME
FROM   Q.COURSE
WHERE  INSTRUCTOR = 'Simon'
```

以上语句执行后，将选择出以下数据。

```
JCL
JES2
```

由此可见，使用 SELECT 语句进行数据查询选择时，通常是根据 WHERE 从句指定选择条件的。WHERE 从句中的条件不仅可以包含通常的关系运算，还可以包含很多其他的比较条件。例如，以下 SELECT 语句将选择课时小于 3，并且课程编号以 E 开头的课程编号及名称。

```
SELECT  CODE, NAME
FROM   Q.COURSE
WHERE  DAYS > 2
       AND
       CODE  LIKE  'E%'
```

以上语句执行后，将选择出以下数据。

```
ES07   JCL
ES28   JES2
ES19   RACF
```

在 SELECT 语句中，还可以存在一些功能子句，通常的功能子句及其功能分别如下。

- SUM: 得到所选数据的总数。
- AVG: 得到所选数据的平均数。
- MIN: 得到所选数据中的最小数。
- MAX: 得到所选数据中的最大数。
- COUNT(*): 得到所选数据的个数。

例如，以下 SELECT 语句将通过功能子句得到由 Tony 任课的课程数及总课时。

```
SELECT   COUNT(*)  AS  COUNT,
         SUM(DAYS)  AS  TOTAL DAY
FROM     Q.COURSE
WHERE   INSTRUCTOR = 'Tony'
```

以上语句执行后，COUNT 以及 TOTAL_DAY 这两个变量中的数据将分别如下。

```
COUNT: 2
TOTAL_DAY: 5
```

在 SELECT 语句中，有时还会用到分组选择的功能。分组选择的功能是通过 GROUP BY 从句实现的。例如，以下为一条使用到分组选择功能的 SELECT 语句。

```
SELECT   CODE, INSTRUCTOR, DAYS
FROM     Q.COURSE
WHERE    CODE  <>  'ES07'
GROUP    BY  INSTRUCTOR
HAVING   MAX(DAYS) >= 3
ORDER    BY  1  DESC
```

以上语句首先通过 WHERE 从语排除了课程编号为“SS83”的数据记录。其后，该语句使用了 GROUP 从句根据任课老师对剩下的数据记录进行了分组。分组后的情况如下。

```
ES07   Simon  2
ES28   Simon  2

ES15   Daisy   3

SS06   Tony   3
ES19   Tony   2
```

可以看到，通过 GROUP BY 从句将数据分为了 3 组，其中每组的任课教师都相同。注意到该 SELECT 语句只对课程编号、任课老师、课时这 3 个属性进行了选择。因此，在以上分组数据中并没有课程名称这一数据项。

接下来，该语句通过 HAVING 从句对分组进行了选择。需要注意的是，HAVING 从句是和 GROUP BY 语句紧密对应的。HAVING 对于 GOURP BY，相当于 WHERE 对于 SELECT。以上 HAVING 从句将每组中最大课时大于 3 天的组选择了出来，选择后的数据如下。

```
ES15   Daisy   3

SS06   Tony   3
ES19   Tony   2
```

最后，该语句通过 ORDER BY 从句对结果数据进行了排序。排序依据所选数据的第 1 列属性，即课程编号进行排序，并且通过 DESC 指定为降序排列。由于课程编号为字符型数据，因此其大小的比较根据前面讲解的字符串的比较方式进行。最终结果如下。

```
ES15   Daisy   3
ES19   Tony   2
SS06   Tony   3
```

以上讲解了 SELECT 语句的一些基本用法。此外，在 SELECT 语句中还存在着多表查询、嵌套查询等内容，作为基础教程，此处不再一一讲解。

2．INSERT 语句、UPDATE 语句以及 DELETE 语句

这 3 种 DML 类别的 SQL 语句相对于 SELECT 语句用得较少，用法也较为简单。这 3 种语句的功能分别如下。

- INSERT 语句：实现数据的插入功能。
- UPDATE 语句：实现数据的更新功能。
- DELETE 语句：实现数据的删除功能。

仍然以上面的 Q.COURSE 表为例，以下使用 INSERT 语句添入了几行新的数据记录。需要注意的是，此处尚未安排任课老师，因此该项属性应为空（NULL）。相应 INSERT 语句如下。

```
INSERT  INTO  Q.COURSE
VALUES  ('AD40', 'COBOL', NULL, 3),
        ('ES26', 'SMP/E', NULL, 2),
        ('ES52', 'REXX', NULL, 2)
```

以上语句也可以直接指定相应的属性项，而不用通过 NULL 表示。这种情况下的 INSERT 语句如下。

```
INSERT  INTO  Q.COURSE
(CODE, NAME, DAYS)
VALUES  ('AD40', 'COBOL', 3),
        ('ES26', 'SMP/E', 2),
        ('ES52', 'REXX', 2)
```

UPDATE 语句通常需要结合 SET 从句对数据进行更新。同时，在 UPDATE 语句中也可以经常使用 WHERE 从句指明修改何处数据。例如，以下 UPDATE 语句将编号为“ES07”课程的学时改为了 3 天。

```
UPDATE  Q.COURSE
SET   DAYS = 3
WHERE  CODE = 'ES07'
```

DELETE 语句常用于删除整行数据记录。在 DELETE 语句中，也是通过 WHERE 从句指定删除哪条数据记录的。例如，以下 DELETE 语句将任课老师为 Tony 的所有课程信息都删除了。

```
DELETE  FROM  Q.COURSE
WHERE  INSTRUCTOR = 'Tony'
```

需要注意的是，如果此处没有 WHERE 从句，将删除 Q.COURSE 表中的所有记录。

15.3.2 DDL 类别的 SQL 语句

DDL（Data Definition Language）即数据定义语言的意思。DDL 类别的 SQL 语句有以下几种。

- CREATE 语句。
- ALTER 语句。
- DROP 语句。

其中，GREATE 语句主要用于创建 DB2 数据库中的表。使用 GREATE 语句创建表时，需要指定表的名称，以及表中数据记录的各属性名称（即表的列名）。通常还需指定各属性的数据类型、限制条件，以及表中的主关键字与外来关键字等。例如，以下语句使用 GREATE 语句创建了一个名称为 TBTEST 的表。

```
CREATE   TABLE  TBTEST
(TEST NO         CHAR (5)      NOT  NULL,
 TEST NAME      CHAR(20)       NOT  NULL,
 FOREIGN_NO     CHAR(3)        NOT  NULL,
 TEST ATTR1     CHAR(10),
 TEST ATTR2     INT,
 TEST_ATTR3     DATE          NOT  NULL,
 TEST ATTR4     VARCHAR(100),
 TEST_ATTR5     TIMESTAMP     NOT  NULL
```

```
                              WITH   DEFAULT,
 PRIMARY  KEY     (TEST_NO),              /*定义该表主关键字为 TEST_NO*/
 FOREIGN  KEY     (FOREIGN NO)            /*定义外来关键字 FOREIGN NO 来自于表 FOREIGN*/
                  REFERENCES   FOREIGN )
```

CREATE 语句还可用于创建建立于表上的视图和索引。例如，以下语句创建了一个基于表 TBTEST 的视图，名为 VITEST。

```
CREATE  VIEW  VITEST
((TEST NO, TEST NAME)   AS
SELECT  TEST NO > 'A00'
FROM  TBTEST)
```

使用 CREATE 语句建立索引也是以表为基础的。例如，以下语句创建了一个基于表 TBTEST 的索引，名为 INXTEST。

```
CREATE  INDEX  INXTEST
ON  TBTEST(TEST NO)
INCLUDE  (TEST_NAME, TEST_ATTR1)
```

还可使用 CREATE 语句创建 DB2 中的一些其他元件，如存储组、数据库、表空间等。其中创建存储组的代码如下。

```
CREATE  STOGROUP  ……
        VOLUMES ……
        VCAT ……                        /*此处为相应编目 Catalgue 的名称*/
        PASSWORD ……
```

创建数据库（存储意义上的数据库）的代码如下。

```
CREATE   DATABASE ……
         STOGROUP ……
         (CCSID  [ASCII / EBCDIC] )   /*此处指明数据库中数据的默认 scheme 编码*/
```

创建表空间的代码如下。

```
CREATE  TABLESPACE ……
        IN ……                         /*此处为所在数据库的名称*/
        USING  STOGROUP ……
```

最后，使用 CREATE 语句还可以用来创建触发器（trigger）。触发器通常是指当对数据进行某一项操作时，将会触发另一项由该触发器所指定的操作。例如，以下为使用 CREATE 定义的一个触发器。

```
CREATE   TRIGGER  TESTTRI
         AFTER  UPDATE  OF  TEST NO  ON  TBTEST      /*触发器被触发的时机*/
         REFERENCING  NEW  AS  N
         FOR  EACH  ROW  MODE  DB2SQL
         WHEN  (N.TEST NO  <  'A0000')              /*触发器被触发的条件*/
         INSERT  INTO  TESTTRI  VALUES
               (N.ITEMNO, CURRENT  TIMESTAMP)       /*ITEMNO 指定了触发器要进行的操作*/
```

ALTER 语句主要用于处理表中的属性。使用 ALTER 语句可以新增、修改及删除表中的属性。以下代码分别反映了这 3 种用法。

```
ALTER  TBTEST  ADD  COLUMN  (NEW_ATTR   CHAR(10))    /*增加一列新的属性*/
ALTER  TBTEST  ALTER  (TEST_ATTR1   INT)             /*更改一列原有属性的数据类型*/
ALTER  TBTEST  DROP  COLUMN  TEST_ATTR2              /*删除一列原有属性*/
```

DROP 语句可以用来删除整张表，以及建立在表上的视图和索引。注意之前讲解的DELETE 语句针对的是具体的数据记录，而 DROP 语句则是针对的整个数据库元件。以下代码反映了 DROP 语句的常见用法。

```
DROP  TABLE  TBTEST          /*删除一张表*/
DROP  VIEW   VITEST          /*删除一个视图*/
DROP  INDEX  INXTEST         /*删除一个索引*/
```

15.3.3 DCL 类别的 SQL 语句

DCL（Data Control Language）即数据控制语言的意思。DCL 类别的 SQL 语句主要有以下几种。

- GRANT 语句。
- REVOKE 语句。
- ROLLBACK 语句。
- COMMIT 语句。

GRANT 语句用于赋予一个用户或一个用户组对数据库的某些访问权限。以下代码反映了该语句的一些常见用法。

```
GRANT  ALL ON TBTEST  TO  STU226              /*赋予用户 STU226 对 TBTEST 表的所有权限*/
GRANT  SELECT ON VITEST  TO  GROUP STUGRP     /*赋予用户组 STUGRP 对 VITEST 视图的选择权限*/
GRANT  SELECT ON TBTEST  TO  PUBLIC           /*赋予全体用户对 TBTEST 表的选择权限*/
```

REVOKE 语句相当于 GRANT 语句的逆运算。该语句用于收回通过 GRANT 语句赋予的权限。以下代码反映了该语句的常见用法。

```
REVOKE  ALL  ON  TBTEST  FROM  STU226
REVOKE  SELECT  ON  VITEST   FROM  GROUP  STUGRP
REVOKE  SELECT   ON  TBTEST  FROM  PUBLIC
```

ROLLBACK 语句和 COMMIT 语句主要用于控制数据操作的执行流程。其中 ROLLBACK 语句表示将之前所做的操作进行重做。由于之前所有的操作可能过多，因此通常创建检查点以配合 ROLLBACK 语句的实际使用。当存在有检查点时，ROLLBACK 语句只将到检查点为止的所有操作进行重做。检查点由 SAVEPOINT 所指定。

COMMIT 语句表示将到目前为止所有做过的操作进行提交。提交之后的结果将写入 DB2 数据库中，不可以被更改。通常情况下，每做完 500 条操作，就使用 COMMITE 语句提交一次。等全部操作做完之后，再一起提交或每做完几条操作就提交一次都是不可取的。以下代码表示了 ROLLBACK 语句和 COMMIT 语句的大致使用方式。

```
......
SAVEPOINT  SP  ON  ROLLBACK
INSERT   INTO  TBTEST  (TEST_NO,  TEST_NAME)  VALUES  ('00001', 'TEST_001')
INSERT   INTO  TBTEST  (TEST_NO,  TEST_NAME)  VALUES  ('00002', 'TEST_002')
INSERT   INTO  TBTEST  (TEST NO,  TEST NAME)  VALUES  ('00003', 'TEST 003')
......
INSERT   INTO  TBTEST  (TEST_NO,  TEST_NAME)  VALUES  ('00500', 'TEST_500')
IF  some conditions
   ROLLBACK  TO  SAVEPOINT  SP
COMMIT
```

15.4 嵌入式 SQL

在 COBOL 源程序中编写的用于访问及操作 DB2 数据库的 SQL 语句，都属于嵌入式 SQL 语句。前面曾提到，SQL 语句是通过首尾的分割符 EXEC SQL 和 END-EXEC 嵌入到 COBOL 源代码中的。除此之外，关于嵌入式的 SQL 语句，还有其他一些需要了解的地方，下面分别进行介绍。

15.4.1 主变量

嵌入式 SQL 语句中的主变量（Host Variables）对于使用 COBOL 操作 DB2 而言是十分重要的。主变量直接在 COBOL 和 DB2 的数据之间建立起了联系。主变量中的数值既可以为 COBOL 所用，也可以为嵌入其中的 SQL 语句所用。

例如，仍然以上一节中的 Q.COURSE 表（见图 15.10）为实验模型。当直接通过 SQL 语句插入一行新的数据记录时，该语句可以如下。

```
INSERT  INTO   Q.COURSE
( CODE, NAME )
VALUES  ('ES52', 'REXX')
```

可以看到，以上插入的新的一行记录只包含课程编号和课程名称两个数据，其余属性为空。当在 COBOL 中通过嵌入式 SQL 语句实现同样的功能时，便需要用到主变量。实现的方法是在 COBOL 中将数值 MOVE 到主变量中，再将主变量应用于 SQL 语句。相关代码如下。

```
......
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
    EXEC SQL  INCLUDE  SQLCA  END-EXEC.
    EXEC SQL  INCLUDE  COURSERCD  END-EXEC.
 01  COURSE-DETAIL.
    03   HCODE    PIC  X(4).
    03   HNAME    PIC  X(20).
    03   HINSTR   PIC  X(10).
    03   HDAYS    PIC  9.
......
 PROCEDURE  DIVISION.
      MOVE  'ES52'  TO  HCODE.
      MOVE  'REXX'  TO  HNAME.
*
      EXEC  SQL
        INSERT   INTO  Q.COURSE
        ( CODE, NAME )
        VALUES  ( :HCODE, :HNAME )
      END-EXEC.
*
      ......
       STOP  RUN.
```

由此可见，在 COBOL 程序中，主变量实际上同其他变量一样使用。在嵌入式 SQL 语句中，是通过在主变量名称前加上冒号“:”对其进行引用的。

主变量不仅可以用于在嵌入式 SQL 语句中进行数据的插入，也可以用于数据的更新。例如，以下代码便通过主变量将编号为“SS06”课程的任课老师更新为了“Simon”。

```
.....
 PROCEDURE  DIVISION.
     MOVE  'SS06'  TO  HCODE.
     MOVE  'Simon'  TO  HINSTR.
*
     EXEC  SQL
       UPDATE  Q.COURSE
       SET  INSTRUCTOR  =  :HINSTR
        WHERE  CODE  =  :HCODE
     END-EXEC.
*
     ......
```

无论是插入数据，还是更新数据，都是在嵌入式 SQL 语句中将主变量作为了数据的发送者。与之相应，主变量同时也可作为数据的接受者。例如，以下代码便将课程编号为“ES15”的所有课程信息都存放在了相应的主变量中。

```
.....
 PROCEDURE  DIVISION.
     MOVE  'ES15'  TO  HCODE.
*
     EXEC  SQL
       SELECT  CODE, NAME, INSTRUCTOR, DAYS
          INTO  :HCODE, :HNAME, :HINSTR, :HDAYS
          FROM  Q.COURSE
          WHERE  CODE = :HCODE
     END-EXEC.
*
     ......
```

15.4.2 指示变量

嵌入式 SQL 语句中的指示变量（Indicator Variable）也是比较重要的一个概念。指示变量的形式同主变量类似，也是通过冒号“:”引用的。指示变量紧接着主变量之后，用于指示主变量所接收的数据是否为空值。当主变量接收的数据为空值 NULL 时，指示变量中的数值将为负数。基于指示变量的这一特性，通常将其用作以下两个方面。

- 测试所选择的数据是否为空值。
- 将空值传递给 DB2 中的数据。

为更好地结合实际对指示变量进行讲解，需要创建一张用于测试的表。创建该测试表的 SQL 语句如下。

```
CREATE  TABLE  TESTTB
( TESTNO      CHAR (5)    NOT NULL,
 TESTNAME    CHAR (20)   NOT NULL,
 TESTATTR1   CHAR (10),
 TESTATTR2   CHAR (10),
 TESTATTR3   CHAR (10))
```

通过以上 SQL 语句可以看出，所创建的 TESTTB 表中仅有两项数据不能为空。其余数

据都有可能为空。当使用嵌入式 SQL 语句对表中数据进行选择时，可根据指示变量对空值数据进行相应处理。例如，以下嵌入式 SQL 语句将接受到空值的主变量统一设置为了“UNKNOW”。

```
……
  EXEC  SQL
      SELECT  TESTNO, TESTATTR1, TESTATTR2, TESTATTR3
      INTO  :TESTNO, :TESTATTR1:IND1, :TESTATTR2:IND2, :TESTATTR3:IND3
      FROM  TESTTB
      WHERE  TESTNO = :TESTNO
  END-EXEC.
……
  IF IND1 < 0   MOVE  'UNKNOW'  TO  TESTATTR1.
  IF IND1 < 0   MOVE  'UNKNOW'  TO  TESTATTR2.
  IF IND1 < 0   MOVE  'UNKNOW'  TO  TESTATTR3.
……
```

当需要将空值传递给 TESTTB 表中的相应数据时，也可以通过指示变量进行。首先要在 COBOL 代码中将指示变量赋为一个负值。然后，再将包含有指示变量的主变量整个赋值到表中相应数据项中。例如，以下语句通过 IND1 将 TESTNO 为“000A1”的数据记录所对应的 TESTATTR1 设置为了空值。

```
……
  MOVE  -1  TO  IND1.
  MOVE  '000A1'  TO  TESTNO.
……
  EXEC  SQL
    UPDATE  TESTTB
    SET  TESTATTR1  =  :TESTATTR1:IND1
    WHERE  TESTNO  =  :TESTNO
  END-EXEC.
……
```

最后需要说明的是，指示变量除用于指示主变量中是否为空值时，还有一些其他的用途。例如，当主变量与 DB2 表中的属性数据所定义的类型或长度不符合时，指示变量便会产生相应信息。当属性数据无法存入主变量空间时，指示变量中的数值将为–2。

15.4.3 SQLCA

在前面章节中曾提到过 SQLCA 的概念。SQLCA 作为 DB2 与应用程序的通信区域，也是嵌入式 SQL 语句中一个重要的概念。SQLCA 实际上相当于应用程序与 DB2 之间的一个接口，如图 15.11 所示，为其主体功能。

在 SQLCA 中，除 SQLCODE 外，还存在一个 SQLSTATE。这两者都是用于由 DB2 反馈给应用程序的 SQL 语句的执行情况，各自的特点分别如下。

- SQLCODE：由数字组成，描述信息更加具体，不过有可能会依赖于不同的硬件平台。
- SQLSTATE：由字符组成，描述信息不如 SQLCODE 具体，但不依赖于硬件平台。

实际上，SQLSTATE 是由 5 位字符所组成的。其中前 2 位字符用于表示概要信息，后 3 位字符用于表示更具体一些的信息。关于 SQLCODE 和 SQLSTATE 在不同情况下的取值情况，如表 15.1 所示。

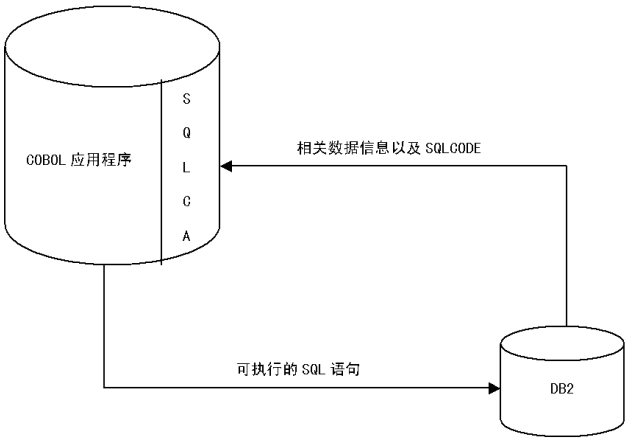

图 15.11　SQLCA 主体功能图

表 15.1　　SQLCODE 和 SQLSTATE 的取值情况

SQL 语句执行后的情况	SQLCODE 的取值	SQLSTATE 的取值
警告	大于 0 且不等于 100	“01nnn”
错误	小于 0	大于“02nnn”
数据没找到	100	“02nnn”
成功	0	“00000”

最后，关于 SQLCA 中存在的警告信息，通常有以下几条。

- SQLWARN1：传递给主变量的字符串被截断了。
- SQLWARN2：对属性列赋值时将空值 NULL 排除了。
- SQLWARN3：属性列的个数大于主变量的个数。
- SQLWARN4：UPDATE 语句或 DELETE 语句中缺少 WHERE 从句。
- SQLWARN6：日期（DATE）或时间戳（TIMESTAMP）被调整以改正由运算得到的非法日期。
- SQLWARN8：不可转换的字符被其他字符所替代。
- SQLWARN9：在属性列的功能处理中忽略掉了错误的算术运算表达式。
- SQLWARNA：将 SQLCA 某区域中的字符数据进行转换时，检测到一条转换错误。

15.5　动态 SQL

动态 SQL 通常是指将 SQL 语句首先读入某一变量中。然后，直接通过嵌入式 SQL 语句执行该变量中所包含的 SQL 语句。动态 SQL 中的执行变量既可以包含 SELECT 语句，也可以包含非 SELECT 语句。在非 SELECT 语句中，又可以分为两种情况。一种是不含参数的非 SELECT 语句，另一种是含有参数的非 SELECT 语句。此处将仅对这两种情况分别进行讲解。

15.5.1　不含参数的非 SELECT 语句

执行不含参数的非 SELECT 语句应该是动态 SQL 中最简单的一种情况。如图 15.12 所示，

为动态 SQL 从编写到执行的流程。

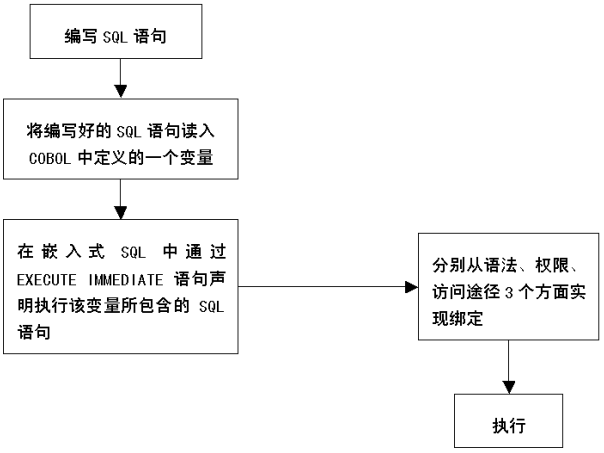

图 15.12　动态 SQL 中不含参数的非 SELECT 语句执行流程

例如，假设在 COBOL 程序中将用于动态 SQL 的执行变量定义为如下形式。

```
01  STMT.
    49   STMT-LEN    PIC S9(4)  COMP  VALUE  +255.
    49   STMT-TEXT   PIC X(255).
```

定义完成之后，可以将如下这条不含参数的非 SELECT 语句读入该变量中。

```
DELETE   FROM  TESTTB
```

此时，可以在 COBOL 中通过动态 SQL 的方式执行上条 SQL 语句。执行方式如下。

```
EXEC  SQL
   EXECUTE  IMMEDIATE  :STMT
END-SQL.
```

由此可见，以上动态 SQL 的执行方式是直接通过执行变量 STMT 中所包含的 SQL 语句进行的。同时需要注意的是，在执行完成后，还应检查一下 SQLCA 中的内容。

15.5.2　含有参数的非 SELECT 语句

含有参数的非 SELECT 语句比不含参数的情况要略微复杂一些。如图 15.13 所示，为动态 SQL 从编写到执行的流程。

例如，假设在 COBOL 程序中将用于动态 SQL 的执行变量，相关参数定义为如下形式。

```
01  STMT.
    49   STMT-LEN    PIC S9(4)  COMP  VALUE  +255.
    49   STMT-TEXT   PIC X(255).
01  PARMA           PIC X(5).
01  PARMB           PIC X(20).
```

定义完成之后，可以将如下这条不含参数的非 SELECT 语句读入该变量中。

```
DELETE  FROM  TESTTB
        WHERE  TESTNO = ?
        AND    TESTNAME = ?
```

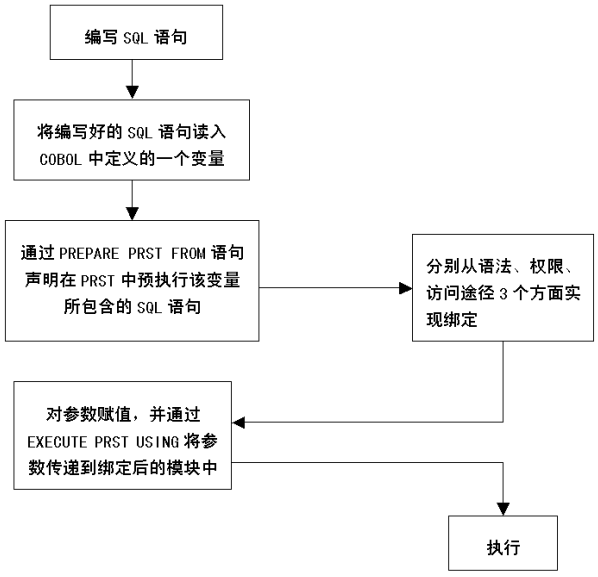

图 15.13　动态 SQL 中含有参数的非 SELECT 语句执行流程

此时，首先在 COBOL 中通过动态 SQL 方式对以上语句进行预执行。预执行是在 PRST 中进行的。PRST 为 DB2 中的一块工作区域，并不在 COBOL 程序中定义。在预执行完成后，也应该检查一下 SQLCA。预执行的方式如下。

```
EXEC  SQL
   PREPARE  PRST  FROM  :STMT
END-SQL.
```

预执行完成后，该 SQL 语句已经在 DB2 中进行绑定了。不过，此时该语句中还缺少两个参数的内容。因此，最终需要将参数赋值后，传递到 PRST 中以完整地执行。此外，在执行之后，也应对 SQLCA 进行检查。完成参数赋值和传递的代码如下。

```
MOVE  'A0023'  TO  PARMA.
MOVE  'James F. Ross'  TO  PARMB.
EXEC  SQL
   EXECUTE   PRST  USING  :PARMA,  :PARMB
END-EXEC.
```

15.6　DB2 中的游标

当对 DB2 的表中的多行数据记录进行处理时，需要用到游标（Cursor）。游标用来对以行为单位的数据记录的定位功能。在 DB2 中，除基本的游标外，还有可以回滚（Scrollable）的游标。可以回滚的游标中又分为静态（Static）可回滚的游标与动态（Dynamic）可回滚的游标两种类型。下面分别进行讲解。

15.6.1　游标的基本定义及用法

在前面章节的示例程序中曾出现过游标的定义及用法。该程序中对游标的使用方式只是其基本使用方式之一。此处将分别对游标的定义和游标的用法进行更为详细的讲解。

1．游标的定义

在 COBOL 程序中，游标是通过 DECLARE CURSOR 语句进行定义的。在定义游标时，也可以指定该游标只用于哪些特定的行，以及只关注这些行中哪几项属性。

例如，对于 TESTTB 表而言，要求所定义的游标 P1 只用于 TESTATTR1 属性为“A”的行，并且，该游标只关注这些行中的 TESTNO 和 TESTNAME 属性。假设属性值“A”已保存在主变量 HATTR1 中，则对该游标的定义方式如下。

```
EXEC  SQL
     DECLARE  P1  CURSOR  FOR
     SELECT  TESTNO,  TESTNAME
     FROM  TESTTB
     WHERE  TESTATTR1 = :HATTR1
END-EXEC.
```

此外，在定义游标时还能指定游标对数据记录搜寻的顺序。如果未指定，游标在使用时将根据数据记录在表中的位置依次进行搜寻。指定之后，游标便能根据某一属性中的数值顺序进行搜寻了。例如，对于 TESTTB 表，当要求所定义的游标能根据 TESTNO 的值由小到大搜寻时，可定义如下。

```
EXEC  SQL
     DECLARE  P2  CURSOR  FOR
     SELECT  TESTNO,  TESTNAME
     FROM  TESTTB
     ORDER  BY  TESTNO
END-EXEC.
```

在实际生产环境中，数据库中满足程序选取条件的数据记录往往是相当多的。可以通过在定义游标时，要求该游标将满足条件的所有记录进行分屏显示。例如，通过以下定义的游标将在每屏中，只显示满足程序选取条件的 20 行数据记录。

```
EXEC  SQL
     DECLARE  C1   CURSOR  FOR
     SELECT  *  FROM  TESTTB
     OPTIMIZE  FOR  20  ROWS
END-EXEC.
```

对于满足条件较多的数据记录，也可以只选取其中的部分进行处理。例如，以下定义的游标将只选取满足条件的前 200 行数据记录。

```
EXEC  SQL
   DECLARE  C2   CURSOR  FOR
   SELECT  *  FROM  TESTTB
   FETCH  FIRST  200  ROWS   ONLY
END-EXEC.
```

2．游标的用法

当定义完成游标之后，便可以对所定义的游标进行实际使用了。需要注意的是，在使用游标之前，需要将其打开。使用完成后，也需要将游标关闭。关于游标在 COBOL 程序中完整的用法结构代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.    TEST99.
 AUTHOR.       XXX.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  HOSTLS.                                        /*以下定义主变量*/
     05  HNO    PIC  X(5).
     05  HNAME  PIC  X(20).
  ......
*
 PROCEDURE  DIVISION.
       EXEC  SQL  DECLARE  TST  CURSOR  FOR         /*以下定义游标 TST*/
                  SELECT  TESTNO,  TESTNAME
                  FROM  TESTTB
                  WHERE  TESTATTR1 = :HATTR1
                  ORDER  BY  TESTNO
      END-EXEC.
      EXEC  SQL  OPEN  TST  END-EXEC.              /*此处打开游标*/
      ......
      EXEC  SQL                                    /*以下对游标进行使用*/
         FETCH  TST  INTO  :HNO, :HNAME
      END-EXEC.
      ......
      EXEC  SQL  CLOSE  TST  END-EXEC.             /*此处关闭游标*/
       STOP RUN
```

可以看到，在 COBOL 程序中主要是通过 FETCH 语句实现对游标的使用的。FETCH 语句实现的功能便是将游标所指向的数据记录中相应属性的值传递到主变量中。FETCH 语句每传递一行值后，游标便向下移动一行。通过循环结构，便可实现多行数据的传递。

同时，使用游标也可实现对多行数据的删除与更新。例如，以下代码在循环结构中便可实现对多行数据的依次删除。

```
EXEC  SQL  FETCH  TST  INTO  :HNO,  :HNAME          /*首先通过 FETCH 语句将游标定位*/
END-EXEC.
......
EXEC  SQL  DELETE  FROM  TESTTB                    /*删除当前游标所指向的一行数据记录*/
       WHERE  CURRENT  OF  TST
END-EXEC.
```

当通过游标实现多行数据的更新时，还需要在定义游标时声明该游标可以用来更新哪一属性值。例如，以下代码可通过游标实现对多行数据中的 TESTNAME 属性值进行更新。

```
EXEC  SQL  DECLARE  TST  CURSOR  FOR
           SELECT  TESTNO, TESTNAME
           FROM  TESTTB
           FOR  UPDATE  OF  TESTNAME            /*此处声明该游标可用于更新 TESTNAME 属性值*/
......
EXEC  SQL  FETCH  TST  INTO  :HNO,  :HNAME
END-EXEC.
......
EXEC  SQL  UPDATE  TESTTB                       /*以下将该属性值更新为主变量 HNAME 中的数据*/
```

```
            SET   TESTNAME = :HNAME
            WHERE  CURRENT  OF  TST
END-EXEC.
```

15.6.2 回滚游标的概念及指向方式

回滚（Scrollable）游标是指可以在表中循环移动、逆向移动以及直接指向某一特定行的游标。回滚游标通常可以分为静态回滚游标和动态回滚游标两种类型。当在程序中打开回滚游标后，通常会得到一张结果信息表。结果信息表中的数据为原表中满足选择条件的数据记录。

对于结果信息表中的数据记录，可以结合 FETCH 语句实现游标对其多种方式的指向。如图 15.14 所示，为常用的几种回滚游标指向方式。

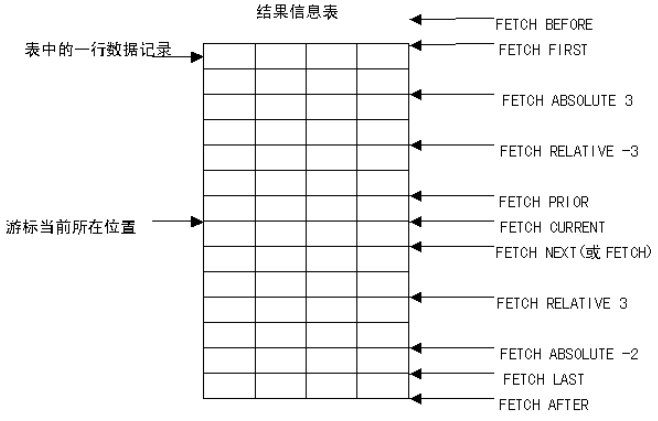

图 15.14　多种游标指向方式

15.6.3 静态回滚游标

静态回滚游标属于传统的回滚游标。当在程序中打开静态回滚游标后，将会根据游标定义时的选择条件，创建一张临时的表，作为结果信息表。在静态回滚游标中，又分为不敏感的静态回滚游标和敏感的静态回滚游标两种，其中对于不敏感的静态回滚游标的定义方式如下。

```
EXEC  SQL
   DECLARE  S1  INSENSITIVE  SCROLL  CURSOR
        FOR   SELECT  ……
        FROM  ……
        WHERE  ……
END-EXEC.
```

关于不敏感的静态回滚游标，主要有以下两点需要注意。

- 使用 FETCH 在结果信息表中操作该游标时，对于原表中相应数据的更新不敏感。
- 打开该游标时所创建的结果信息表的所包含的记录行数及数据内容不能被更改。

由于此时结果信息表中的信息不会被更改，因此当原表中的数据更新时，将会产生数据的不一致。于是，需要引入敏感的静态回滚游标。该游标的定义方式如下。

```
EXEC  SQL
  DECLARE  S2  SENSITIVE  STATIC  SCROLL  CURSOR
       FOR   SELECT  ……
```

```
        FROM  ……
        WHERE  ……
END-EXEC.
```

与不敏感的静态回滚游标相对应，敏感的静态回滚游标主要有以下特点。

- 使用 FETCH 在结果信息表中操作该游标时，对于原表中相应数据的更新敏感。
- 结果信息表的记录行数不能被更改，但数据内容是可以被更改的。

下面结合一个具体实例来讲解敏感的静态回滚游标是如何针对不同情况进行相应处理的。例如，如图 15.15 所示，为一张课程信息表 CTABLE。

CNO	CNAME	CTEACHER	CCREDIT
0010	C 语言程序设计	卢飞	3.5
0022	数据结构	殷胜	3.5
0015	汇编语言程序设计	曹升	3.0
1023	数理方程与特殊函数	李亮	2.5
0057	数值分析	张平	2.5
1048	复变函数与积分变换	王刚	2.5
0105	微电子器件与 IC 设计	刘涛	3.5
0036	面向对象程序设计	杨恒	2.0
0179	模拟电子技术（二）	赵晓	3.0

图 15.15　课程信息表 CTABLE

以下代码定义了基于该表的敏感的静态回滚游标。

```
EXEC  SQL
  DECLARE  CS  SENSITIVE  STATIC  SCROLL  CURSOR
       FOR   SELECT  CNO, CNAME
       FROM  CTABLE
       WHERE  CNO < '1000'
END-EXEC.
```

在程序中打开以上定义的游标时，将产生一个相应的结果信息表 CRESULT，如图 15.16 所示。

假设在原表中将数值分析的课程编号 CNO 进行了更新，在结果信息表中也将会有相应反映。此时，在结果信息表中的相应记录位置将存在一个更新空洞（Update Hole）。当使用 FETCH 语句将游标指向此处时，根据 CNO 更新后的不同取值，将存在以下两种处理结果。

- 当更新后的 CNO 仍然满足游标选择条件，即仍小于“1000”时，更新空洞将被填充。此时，结果信息表中的相应数据将被刷新。并且，通过游标将返回刷新后的正确数据。
- 当更新后的 CNO 不满足游标选择条件，即更新后大于“1000”时，更新空洞仍然保留。此时，DB2 将返回 SQLCODE 为+222。并且通过游标不能得到所期望的返回数据。

CNO	CNAME
0010	C 语言程序设计
0022	数据结构
0015	汇编语言程序设计
0057	数值分析
0105	微电子器件与 IC 设计
0036	面向对象程序设计
0179	模拟电子技术（二）

FETCH
SENSITIVE

图 15.16　相应的结果信息表 CRESULT

15.6.4　动态回滚游标

动态回滚游标是相对于静态回滚游标而言比较新的概念。在动态回滚游标下，对于原表中数据的更新及添加都是可视的（visible）。因此，可以说动态回滚游标都是敏感的。动态回滚游标的定义方式通常如下。

```
EXEC  SQL
  DECLARE  D1  SENSITIVE  DYNAMIC  SCROLL   CURSOR
     FOR   SELECT  ……
     FROM  ……
     WHERE  ……
END-EXEC.
```

关于动态回滚游标，主要有以下两点需要注意。

- 动态回滚游标并不为结果信息表创建一块临时的存储空间。使用 FETCH 语句操作游标时，该游标将直接在原表上移动。
- 结果信息表中数据记录的行数及数据内容都是可以被更改的。

15.6.5　利用游标同时处理多行记录

之前所讲解的利用游标处理多行记录并不是同时进行的。实际上，该种处理方式每次仍然只对一行记录进行处理。只是将其放在循环结构内，便可实现多行记录的处理。这里要介绍的是不通过循环结构，直接同时处理多行记录。

例如，假设某张表中共有 100 行记录，且游标 C1 当前指向第 50 行记录。以下代码将同时对第 50、51、52 行这 3 行数据记录进行处理。

```
EXEC  SQL
   FETCH  CURRENT  ROWSET  FROM  C1  FOR  3  ROWS  INTO ……
END-EXEC.
```

以下代码则以当前游标所在位置为基准，分别同时处理与该位置相关的多行数据记录。

```
EXEC  SQL
   FETCH  PRIOR  ROWSET  FROM  C1  FOR  5  ROWS  INTO ……
```

```
END-EXEC.
*
EXEC  SQL
   FETCH  NEXT  ROWSET  FROM  C1  FOR  10  ROWS  INTO ……
END-EXEC.
*
EXEC  SQL
   FETCH   ROWSET  STARTING  AT  RELATIVE  -7
   FROM  C1  FOR  7  ROWS  INTO ……
END-EXEC.
```

当然，也可不考虑当前游标所在位置，而以记录的绝对位置作为起始位置同时处理多行记录。以下代码反映了这一方式。

```
EXEC  SQL
   FETCH   ROWSET  STARTING  AT  ABSOLUTE  30
   FROM  C1  FOR  10  ROWS  INTO ……
END-EXEC.
```

当通过游标同时对多行数据进行处理时，相应的主变量也应定义为数组的形式。即此时程序中的主变量实际上应通过 OCCURS 语句定义为 COBOL 中的表。例如，以下代码实现了在 COBOL 中同时，将 10 行记录 FETCH 到主变量中的功能。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.    MUTIROW.
 AUTHOR.        XXX.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  HOST-VARY.
    05  HOST-ID        PIC  9(5)  USAGE  COMP
                              OCCURS  10   TIMES.
    05  HOST-NAME    OCCURS  10  TIMES.
        49   NAME-LEN     PIC  9(2)  USAGE  COMP.
        49   NAME-TEXT    PIC  X(20).
 01  IND-VARY.
     10    INDICATE1       PIC  S9(4)  USAGE  COMP
                                OCCURS  10  TIMES.
     10    INDICATE2       PIC  S9(4)  USAGE  COMP
                                OCCURS  10  TIMES.
 ……
*
 PROCEDURE  DIVISION.
      EXEC  SQL
        DECLARE  MTR  SCROLL   CURSOR
          WITH  ROWSET   POSITIONING
          FOR  SELECT  ID, NAME
          FROM  TESTTABLE
      END-EXEC.
      EXEC  SQL   OPEN  MTR  END-EXEC.
      EXEC  SQL
        FETCH  FIRST  ROWSET
        FROM   MTR  FOR  10  ROWS
```

```
          INTO  :HOST-ID:INDICATE1, :HOST-NAME:INDICATE2
      END-EXEC.
 ……
      EXEC  SQL  CLOSE  MTR  END-EXEC.
       STOP  RUN.
```

15.7 DB2 中的锁

DB2 中的锁相当于是为了控制数据操作的流程而建立的一种机制。通过 DB2 中的锁，可以保证数据操作的一致性，并使得数据资源得到合理而有序的利用。关于锁的属性，通常是从以下这几个方面予以考虑的。

- 锁的类型
- 锁的作用范围
- 锁的持续时间

任何数据库中的锁都分为很多不同的类型，DB2 也不例外。在 DB2 中，通常情况下锁的类型及其各自的意义分别如下。

- IN（Intent None）锁：空锁，也可以表示不上锁。
- Z（super exclusive）锁：超排他锁。该锁对于整张表结构及表中数据都上锁。
- S（Share）锁：共享锁。锁的拥有者可以对该锁控制范围内的数据进行读取操作。S 锁可以通过以下 SQL 语句来描述。

```
SELECT  *  FROM  T      /*其中 T 为表名，下同*/
```

- U（Update）锁：更新锁。锁的拥有者可以对该锁控制范围内的数据进行更新操作。U 锁可以通过以下 SQL 语句来描述。

```
SELECT  *  FROM  T
FOR  UPDATE  OF  ……
```

- X（Exclusive）锁：排他锁。锁的拥有者可以对该锁控制范围内的数据进行读写操作。X 锁可以通过以下 SQL 语句来描述。

```
DELETE / UPDATE / INSERT / SELECT
```

- IS（Intent Share）锁：共享意向锁。当锁的拥有者获得了 S 锁后，便可以对 IS 锁控制范围内的数据进行读取操作。IS 锁可以通过以下 SQL 语句来描述。

```
SELECT  *  FROM  T
WHERE ……
```

- IX（Intent Exclusive）锁：排他意向锁。当锁的拥有者获得了 X 锁后，可以对 IX 锁控制范围内的数据进行读写操作。IX 锁可以通过以下 SQL 语句来描述。

```
(DELETE / UPDATE / INSERT / SELECT) ……
WHERE ……
```

- SIX（Share with Intent Exclusive）锁：共享排他意向锁。该锁相当于结合了 S 锁及 IX 锁，并且 S 锁在 IX 锁之前处理。SIX 锁可以通过以下两条 SQL 语句来描述。

```
SELECT …… FROM  T                /*先根据 S 锁进行此步处理*/
(DELETE / UPDATE / INSERT) ……    /*再由 IX 锁进行此步处理*/
WHERE ……
```

锁的作用范围是关于锁的另一个需要考虑的属性。在 DB2 中，锁的作用范围通常情况下，主要有以下几块。

- 作用范围为整个表空间。
- 作用范围为整张表。
- 作用范围为表中数据记录的某几行。

对于作用范围为整张表的锁，通常可以为 IS 锁、IX 锁、S 锁、U 锁、X 锁。对于作用范围为表中某几行数据记录的锁，通常可以为 S 锁、U 锁、X 锁。不过，对于行锁而言，还需考虑到该行所在表的锁，二者的对应情况分别如下。

- 当行锁为 S 锁时，其所在的表至少应该拥有 IS 锁。
- 当行锁为 U 锁时，其所在的表至少应该拥有 IX 锁。
- 当行锁为 X 锁时，其所在的表至少应该拥有 IX 锁。

关于锁的持续时间，通常是和锁的相关处理模式有关的。在 DB2 中，锁的相关处理模式通常有以下几种。

- RS：当提交（Commit）时放锁。
- RR：当提交时放锁，且该模式通常只用于 S 锁。
- CS：每读完一条记录后便放锁，该模式下的并发性能（Concurrency）比较高。
- UR：表示不用于提交的读取，因此不上锁。

以上这 4 种模式通常是可以和游标同时定义的，定义方式如下。

```
EXEC  SQL
  DECLARE  C1  CURSOR  FOR   /*CI 为游标名称*/
           SELECT  ……
     WITH  RR/RS/UR/CS
END-EXEC.
```

对于以上模式，需要注意的是，UR 是可以转换为 CS 的。当在定义 UR 的同时加上 FOR UPDATE OF 语句，便可得到 CS。

最后，简要介绍一下关于 DB2 中死锁的概念。死锁也就是两个及其以上的并发进程互相等待对方的资源释放而产生的无限等待的情况。在数据控制流程中加上以上所讲解的 S 锁、X 锁等也是主要为了防止死锁的产生。同时，在 DB2 中，解决死锁的方式通常有以下几种。

- 将锁进行解除（Lock Escalation）。
- 将锁进行转换（Lock Conversion）。
- 使用可重复读模式（Repeatable Read）。
- 使用非一致的访问方式（Inconsistent Access）。
- 修改数据字典的内容（Catalog Modification）。
- 实施参照约束（Referential Constraint Enforcement）。

15.8 访问路径以及 EXPLAIN

访问路径（Access Path）以及 EXPLAIN 都是和应用程序在 DB2 上的运行性能有关的。其中访问路径是建立在索引的基础之上的，用于决定对数据的访问方式。EXPLAIN 则主要

用于对 DB2 进行优化操作。下面分别予以讲解。

15.8.1 访问路径

使用 SELECT 语句对 DB2 中的数据进行查找访问是最常用的一种操作方式。因此，提高查找访问的效率将从很大程度上提高整个应用程序在 DB2 上的运行性能。

DB2 的访问路径决定了对数据的访问方式，因此对数据的查找访问效率有着直接的影响。访问路径分为不同的类型，以下为几种典型的访问路径。

- Relation Scan。
- Matching Index Scan。
- Non-Matching Index Scan。
- Index-Only Access。

其中，Relation Scan 为以上 4 种典型访问路径中最基本的一种。Relation Scan 对应的选择条件并不包含建有索引的属性。该访问路径将直接顺次查找满足条件的数据。例如，对于 TBTEST 表，不妨假设以 TEST_NO 和 TEST_NAME 建立索引。则以下两条 SQL 语句将对应 Relation Scan。

```
SELECT  *  FROM  TBTEST
SELECT  *  FROM  TBTEST  WHERE  TEST_ATTR1  LIKE  '%TEST%'
```

如图 15.17 所示，结合 TBTEST 表，反映了 Relation Scan 的执行路径。

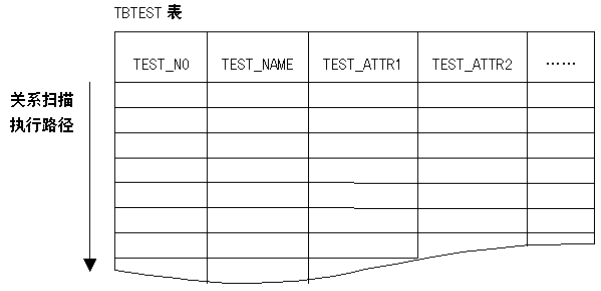

图 15.17 Relation Scan 执行路径

Matching Index Scan 对应的选择条件根据建有索引的属性进行判断。同时，所选取的属性中还包含有其他的属性。以下 SQL 语句将对应 Matching Index Scan。

```
SELECT  TEST_NO, TEST_NAME, TEST_ATTR1
FROM  TBTEST
WHERE  TEST_NO  LIKE  'T%'
```

在 Matching Index Scan 中，需要考虑的数据组织结构通常分为 3 层。第一层为根页（Root Page），包含一级索引，与 VSAM 中 KSDS 里的 IS 类似；第二层为叶子页（Leaf Page），包含二级索引，与 KSDS 里的 SS 类似；第三层即数据页（Data Page），为所有的实际数据。

当进行 Matching Index Scan 时，需要从根页开始，经过叶子页，最终在数据页访问到指定数据。见图 15.18 所示，为 Matching Index Scan 的执行路径。

Non-Matching Index Scan 对应的选择条件根据建有第二索引的属性进行判断。其余情况与 Matching Index Scan 相类似。以下 SQL 语句将对应 Non-Matching Index Scan。

```
SELECT  TEST NO, TEST NAME, TEST ATTR1
FROM  TBTEST
WHERE  TEST_NAME  =  'TNAME'
```

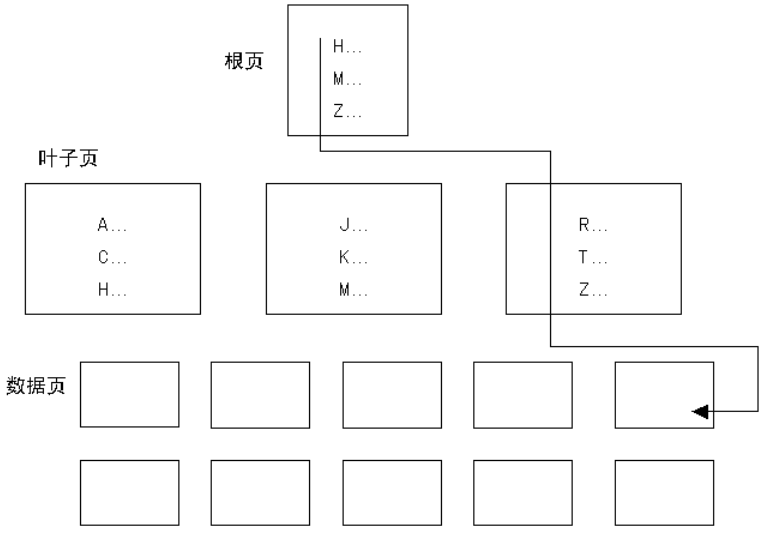

图 15.18　Matching Index Scan 执行路径

当进行 Non-Matching Index Scan 时，并不考虑根页，而直接扫描叶子页，并在数据页得到指定数据。如图 15.19 所示，反映了这一执行路径。

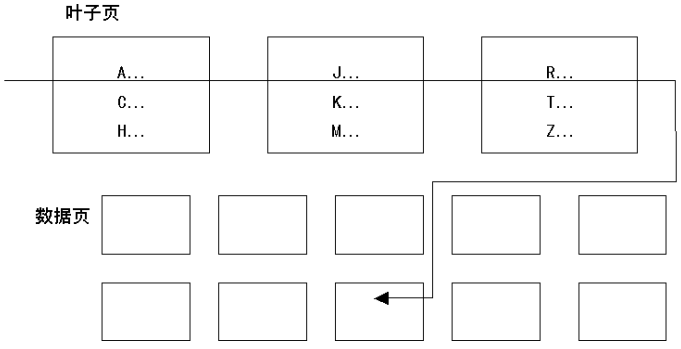

图 15.19　Non-Matching Index Scan 执行路径

Index-Only Access 所对应的选择条件根据建有索引的属性进行判断。同时，所选取的属性将全部为建有索引的属性。以下 SQL 语句将对应 Index-Only Access。

```
SELECT  TEST NO,  TEST NAME
FROM  TBTEST
WHERE  TEST_NO  LIKE  'R%'
```

Index-Only Access 实际上就是只对索引部分进行访问。当进行 Index-Only Access 时，将直接在叶子页中得到相应数据，而不必考虑数据页中的内容。如图 15.20 所示，反映了这一执行路径。

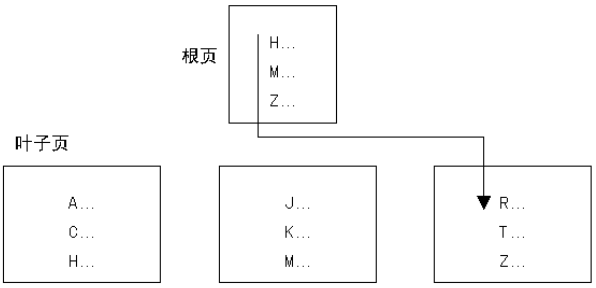

图 15.20　Index-Only Access 执行路径

15.8.2　EXPLAIN 优化工具

EXPLAIN 是 DB2 上的一项工具，主要是用来对基于 DB2 的应用程序的性能进行优化的。DB2 中的优化通常包含以下 3 个部分。

- 对 SQL 语句的优化。
- 对日志统计信息的优化。
- 对配置参数的优化。

使用 EXPLAIN，通常是遵循优化→分析→更改→再优化→再分析→再更改……这一循环步骤的。在 z/OS 大型机操作系统上使用 EXPLAIN，通常还会涉及到一个 PLAN_TABLE 的概念。PLAN_TABLE 实际上为一张表。在基于 DB2 的应用程序的绑定阶段，需要将访问路径的相关信息添加到 PLAN_TABLE 中。以下代码反映了在绑定阶段使用 EXPLAIN 的相应情况。

```
……PACKAGE / PLAN
MEMBER (DBRM1)
EXPLAIN (YES)
```

需要注意的是，DB2 将会在每一新的绑定或重绑定后重做 EXPLAIN。这种处理流程直到遇到 EXPLAIN（NO）时为止。

前面章节曾提到，基于 DB2 的应用程序的绑定通常涉及到两方面的内容。这两方面的内容分别为对 Plan 的绑定和对 Package 的绑定。对于这两种不同的绑定，读取 PLAN_TABLE 的方式将有所不同。以下代码反映了在绑定 Plan 时，对 PLAN_TABLE 的选择读取。

```
SELECT  *   FROM  PLAN TABLE
   WHERE  APPLNAME  =  'PLAN01'
   AND    PROGNAME  =  'DBRM01'
   ORDER  BY  QUERYNO, QBLOCKNO, PLANNO, MIXOPSEQ
```

以下代码反映了在绑定 Package 时，对 PLAN_TABLE 的选择读取。

```
SELECT  *   FROM  PLAN_TABLE
   WHERE  PROGNAME  =  'PKG01'
   AND    VERSION   =  'PROD'
   ORDER  BY  QUERYNO, QBLOCKNO, PLANNO, MIXOPSEQ
```

EXPLAIN 语句同时也可以存在于 SQL 语句中。此时，EXPLAIN 语句将可嵌入于 COBOL 程序代码中，并且可以在 SPUFI 或 QMF 中执行。例如，以下为一段包含有 EXPLAIN 语句

的代码。

```
EXPLAIN  ALL
    SETQUERYNO  =  50
    FOR  SELECT  TEST_ATTR1, TEST_ATTR2
    FROM  TBTEST
    WHERE   TEST_NO = 'A0005'
```

需要注意的是，以上代码中的 QUERYNO 参数既可以由用户指定，也可以由 DB2 系统所设置。此外，在以上 SQL 语句中不可以包含主变量。主变量在此必须使用问号“？”代替。

EXPLAIN 主要是根据 PLAN_TABLE 实现性能优化的。而 PLAN_TABLE 中保存的信息为访问路径的相关信息。由于访问路径通常有 4 种类型，因此 EXPLAIN 在处理 PLAN_TABLE 时也将相应的有 4 种方式。

最后，对于 EXPLAIN 的输出结果以及 PLAN_TABLE，最好应该予以保留。原因在于比较两条 EXPLAIN 往往比检查一条 EXPLAIN 要容易得多。

15.9 本章回顾

本章主要讲解了大型机上最常使用的数据库——DB2 在 COBOL 应用程序开发方面的相关知识。在所涉及的数据量较大的软件项目中，必然需要使用到数据库的。

本章首先介绍了 DB2 的相关基本概念及应用。DB2 作为一种关系型数据库，既有关系型数据库的共性，也有其自身的特性。学习这部分内容，需要了解关系数据库的概念，了解 DB2 的基本概念及组织结构，掌握在 COBOL 中如何应用 DB2，以及理解基于 DB2 的 COBOL 程序的编译过程。

本章接下来介绍了关系数据库中最基本的操作语言——SQL。对于 SQL 的讲解，分别包含了常用 SQL 语句、嵌入式 SQL、以及动态 SQL 的讲解。其中嵌入式 SQL 和动态 SQL 对于涉及到 DB2 的应用程序而言是两个比较重要的概念。学习这部分内容，需要牢固掌握各种常用 SQL 语句的格式及功能，理解嵌入式 SQL 在 COBOL 中的格式，理解嵌入式 SQL 中主变量和指示变量的概念及用途，理解 SQLCA 的概念以及其在嵌入式 SQL 中的用途，理解动态 SQL 的执行流程及代码实现。

本章之后介绍了 DB2 中的游标。关于游标，也是在基于 DB2 的应用程序中经常会用到的一个概念。学习这部分内容，需要掌握游标的定义方式，掌握如何使用 FETCH 语句对游标进行实际操作，理解回滚游标的概念及特征，理解两种类型的静态回滚游标的区别与联系，了解动态回滚游标的概念，掌握如何利用游标同时处理多行数据记录。

本章最后介绍了 DB2 中的锁，以及 DB2 中的访问路径和 EXPLAIN。学习这部分内容，需要理解各种锁的特性及用途，了解锁的不同作用范围，了解不同模式下锁的持续时间，理解访问路径的典型分类及各自的特征，了解 EXPLAIN 的基本概念及用途。

第 16 章 CICS 扩展

CICS 是大型机上的一个中间件，提供了一个面向事务处理的联机应用环境。在应用程序开发方面，CICS 同 DB2 类似，通常也是以 COBOL 作为宿主语言的。对于要求拥有界面的在线交互式程序开发，会需要用到 CICS。

16.1 基本概念

CICS 作为一个独立的子系统，其本身的知识涵盖是十分丰富的。同时，CICS 在大型机程序开发方面的应用也是十分广泛的。实际上，很大一部分 COBOL 程序都是涉及到 CICS 的。这里仅从应用开发的角度，简要介绍一下 CICS 的相关基本概念。

16.1.1 CICS 简介

CICS 的英文全称为 Customer Information Control System，即客户信息控制系统的意思。CICS 最初是 S/370 上的一个程序产品，迄今已有将近 40 年的发展历史。CICS 同 DB2 一样，也是支持多种操作系统的，但目前仍然主要用于大型机的操作系统 z/OS。

CICS 在大型机上的版本依次经历了 1.7 版、2.1 版、3.1.1 版、3.1.2 版、3.3 版、4.1 版等。当 CICS 发展到 5.x 版本后，便被称作 CICS TS（CICS Transaction Server）。虽然先后经历了多个版本。但 CICS 本身仍然为通用的数据库/数据通信（DB/DC）系统，主要是通过 PL/x 语言编写的，该语言同 PL/1 语言比较类似。CICS 在应用程序方面最显著的特征是提供了界面功能和交互功能。基于其强大的交互能力，CICS 广泛应用于各种大型商务领域。以下列举了其中的几项常见应用领域。

- 银行 ATM 交易处理系统
- 航空订票系统
- 保单处理系统
- 大型 ERP 系统
- 在线图书馆

作为 CICS 中的联机处理，是同批处理相对应的一个概念。其中批处理是指在用户不直接干预的情况下，系统对批量资源在规定时间内，进行例行处理的过程。批处理主要有以下特点。

- 所有需要用到的 I/O 区和工作区都应在程序中进行定义。
- 由程序读入批量的输入数据。
- 输入数据必须在处理开始前准备就绪，在处理过程中不得再次插入。
- 程序直接向操作系统发出 I/O 指令。
- 如果出现故障，处理可重新进行，或从故障点继续向后处理。

联机处理是指在用户直接干预的情况下，系统根据用户的输入在短时间内进行交互式处理的过程。联机处理主要有以下特点。

- 用户可以在不同地点，通过不同的终端使用同一台主机。
- 数据可以随时输入到系统中，而无须积累成批量后再输入。
- 对终端的处理请求具有实时性的响应。
- 输出信息通常直接在用户所在终端上显示。
- 可以对同一个文件同时进行多种操作。
- 用户可以在任何时候通过终端启动应用程序，而无须经过操作员的调度安排。

CICS 作为一种中间件，是存在于操作系统和应用程序之间的一个子系统。CICS 实际上是在操作系统控制下的一个分区中作为一个主程序运行。而其他联机应用程序则是在 CICS 的控制下运行的。如图 16.1 所示，为包含有 CICS 的系统架构。

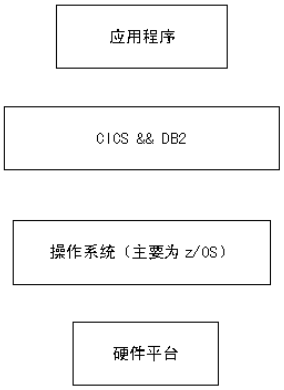

图 16.1　包含有 CICS 的系统架构

有上图可以看到，CICS 通常是结合 DB2 使用的。DB2 同 CICS 一样，也属于一种中间件。CICS 所处的位置是操作系统和应用程序之间的事务管理层。借助 CICS，应用程序不必直接同操作系统打交道，由此可以减轻操作系统的负担。同时，由于操作系统的负担得以减轻，因此也可以满足更多潜在的用户和要求处理的事务。

最后，CICS 作为一个子系统，为运行于其上的应用程序提供了类似于操作系统的管理功能。这些管理功能主要有以下几项。

- 任务管理
- 文件管理
- 程序管理
- 队列管理
- 终端管理
- 系统服务
- 恢复机制
- 外部安全管理

16.1.2　CICS 中的交易和任务

交易（Transaction）和任务（Task）是 CICS 中的两个最基本的概念。这两个概念实际上

也是基于 CICS 面向事务处理的特点而产生的。

1．CICS 中的交易

在 CICS 中，一个交易是指一组相关联的操作序列或为了完成一个特定功能的一组步骤。交易通常产生于终端和数据库之间，属于一种应用过程。一个交易中既可能只有一个操作，也可能存在一组操作。例如，在 ATM 机上完成的取款过程便可视作一个交易，该交易所含的一组操作如下。

（1）验证用户账号和密码信息。

（2）读取用户取款请求。

（3）检查用户卡上余额是否充足。

（4）提取用户所请求数量的现金。

（5）询问用户是否需要打印凭单。

在 CICS 中，每个交易是由一个相应的交易 ID（TRANSID）来标识的。每个 TRANSID 由 4 个字符所组成。同时，交易通常还具有以下两个特点。

- 交易处理的对象主要包含两个：一个是终端用户，另一个是被处理的数据。
- 每个交易处理过程是由终端用户或程序提交一个简单请求而启动的。

在 CICS 系统内部，交易是根据一定的顺序执行的。这种执行顺序在基于 CICS 的应用开发中也将有一定的反映。交易的执行顺序通常如下。

（1）输入：输入一个 TRANSID 以标识相应的交易。

（2）创建：由 CICS 创建一个任务处理该交易，任务将处于准备状态。

（3）分派：确定 CICS 中处于准备状态的哪一个任务将运行，并对其进行分派。

（4）执行：被分派的任务启用相应的程序以运行。

（5）处理：任务所启用的程序调用 CICS 服务时，任务本身将释放 CPU 并等待服务完成。

（6）重分派：服务完成后，任务回到准备状态，CICS 再次对其中的任务进行分派。

（7）返回：当交易中的所有操作都完成后，交易程序通过 RETURN 命令，将控制权交还给 CICS。

（8）终止：CICS 终止任务，并回收相关资源。

实际上，定义一个交易时，往往需要定义相应的程序，并将其与该交易建立关联。此外，应用程序所访问的资源也都需要在 CICS 中进行定义。以下为几个常见的资源及其访问位置。

- 终端，通过 TCT（Terminal Control Table）访问。
- 交易，通过 PCT（Programming Control Table）访问。
- 程序，通过 PPT（Processing Program Table）访问。
- 文件，通过 FCT（File Control Table）访问。

2．CICS 中的任务

CICS 中的任务是指操作员或用户请求的特定交易的一个实例。一个任务实际上就是一个交易的一次执行过程。如果将交易看作一个程序，则任务就相当于是一个进程。

关于 CICS 中的任务，首先需要了解任务控制区域 TCA（Task Control Area）的概念。每一个 TRANSID 标识了一个交易，而每一个 TCA 则代表了一个任务。

同时，每一个任务中还有一个唯一的执行接口块 EIB（Execution Interface Block）。EIB 是 CICS 用来和用户程序进行通信的一组字段。EIB 字段信息在 COBOL 的过程部中将可能被使用到。如图 16.2 所示，为 CICS 任务中 TCA 与 EIB 的大体结构。

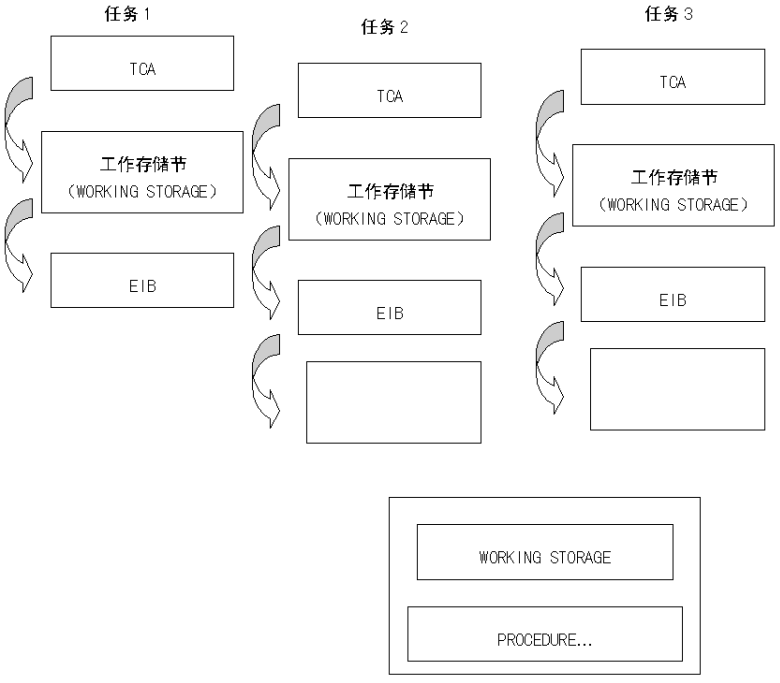

图 16.2　CICS 任务中的 TCA 和 EIB 结构图

需要注意的是，由于 CICS 是支持并发性的，因此多个任务是可以同时执行的。并且，这些任务可以对应同一个交易的执行过程。例如，当 100 个用户在不同的终端上同时输入同一个 TRANSID 时。此时系统中可以只有一个交易，并且对应该交易有 100 个任务。

此外，关于 CICS 中的任务，还有以下几点需要注意。

- 每一个任务都有各自的工作存储节。工作存储节中的数据相对独立。一个任务在执行过程中其本身的数据不能同时被另一个任务使用和修改。
- 多个任务可以作为同一个交易的执行过程，因此也可以调用同一个程序。
- 当任务终止后，CICS 将释放掉该任务所占用的所有内存。

最后需要补充的是，CICS 中的交易是需要进行定义的，而任务则直接通过交易的执行来创建，因此不用定义。并且，交易是可以自己调用自己的，而任务则不能自己调用自己。

16.1.3　CICS 的基本操作

关于 CICS 的基本操作，首先需要清楚 CICS 的开启与关闭。作为一个子系统，CICS 的开启和关闭都是通过在终端，或控制台输入相关命令实现的。在 PCOM 主界面登录 CICS 系统，则与登录 TSO 类似，通常是使用以下命令登录的。

```
L  CICS
```

CICS 的重启与 JES2 的重启类似，也是分为 3 种类型的重启的。关于这 3 种类型的重启

的名称及方式分别如下。

- 冷启动：正常关闭 CICS，并且全新安装 CICS。该重启方式将会清除某些信息。
- 热启动：正常关闭 CICS，但在关闭之前要求所有正在运行的任务依次完成。这种重启方式实际上为最好的一种方式。
- 紧急启动：非正常关闭 CICS，通常是由于电力故障等外在原因造成的。此时，为保证数据的一致性，CICS 在重启后，将回滚所有未完成的逻辑工作单元。

在 CICS 运行过程中，CICS 还提供了许多用于处理系统事务的系统交易。通过这些交易所生成的任务构成了 CICS 日常操作的主体部分。

例如，以下将使用相应的系统交易进行 CICS 的签到和退出操作。

```
CESN                /*进行签到操作*/
CESF                /*进行退出操作*/
```

使用系统交易还可与位于控制台的中心操作员进行通信。例如，以下操作将把“TEST”信息发往控制台。

```
CWTO  'TEST'
```

如果权限足够，也可以将消息发往其他的终端用户上。例如，以下操作将把“CALL TEM1”消息发往编号为“TEM1”的终端上。

```
CMSG  'CALL  TEM1',  R=TEM1,  S
```

以下操作则将把“CALL　ALL”消息发往所有的终端上。

```
CMSG  'CALL  ALL',  R=ALL,  S
```

此外，CICS 中的一些其他用于日常操作的系统交易分别如下。

- CMAC：用于对消息进行查找。
- CECI：用于对指定的 CICS 命令进行解释。
- CECS：用于对 CICS 命令进行语法检查。
- CEBR：用于浏览临时存储队列。

最后，CICS 还提供了 CEDF 和 CEDX 用于进行交互式的调试，其中主要包括以下几项功能。

- 确定并更正程序中的错误。
- 跟踪每一条 CICS 命令。
- 进行异常条件模拟。
- 为异常结束提供更为详细的信息。
- 强迫产生 DUMP。
- 在源程序不可用时，提供检查代码的方法。

以上所说的 CEDF 是在程序调试中最常用到的一项工具。与之相关的还有 CEDA 和 CEMT，分别用于资源的定义和资源的查询与设置。这两项系统交易也是在应用开发中经常用到的。对于这 3 者将在后面章节中详细讲到。

16.2　CICS 编译处理过程

涉及到 CICS 的程序在编译连接时与普通程序是有所不同的。此外，关于 CICS 在实际中所用到的各种资源，都是需要在 CICS 子系统上进行定义的。CICS 任务的执行及程序的调试

也同样是在 CICS 子系统上进行的。下面分别对以上相关内容进行讲解。

16.2.1 CICS 程序编译流程

涉及到 CICS 的程序从源代码到可加载模块之间的编译流程通常需要经历 3 个处理步骤。这 3 个步骤依次如下。

（1）CICS 转换（CICS Translation）。

（2）编译（Compilation）。

（3）连接编辑（Link Edit）。

在以上每一个步骤中，又分别包含有多个相应的具体操作，如图 16.3 所示，共包括 3 个步骤及其所包含的具体操作。

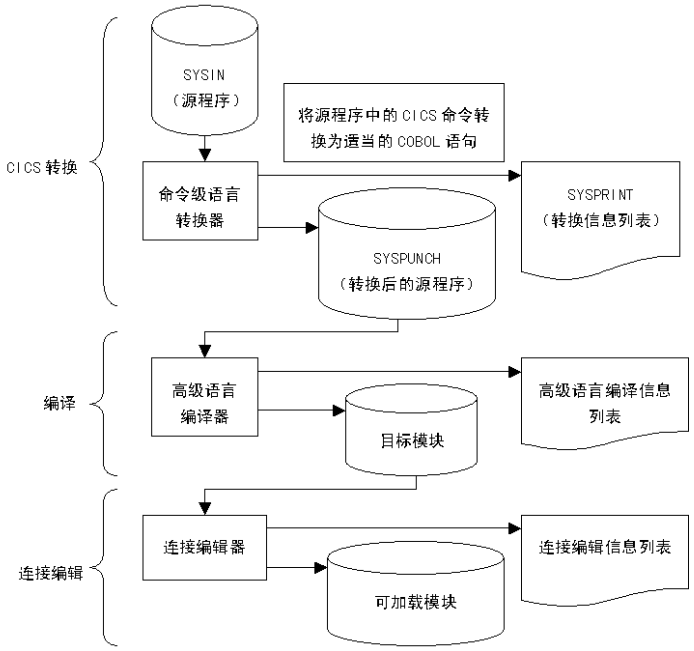

图 16.3 CICS 程序基本编译流程图

前面曾讲到，CICS 还可以结合 DB2 使用的。当作为宿主的 COBOL 程序中既包含有 CICS，又包含有 DB2 时，其编译流程如图 16.4 所示。

可以看到，当包含有 DB2 时，将先进行 DB2 的预编译，之后再进行 CICS 的转换。在 DB2 预编译中，将使用 COBOL 中的 PERFORM 和 CALL 语句替换 EXEC SQL 语句。在 CICS 转换中，将使用 MOVE 和 CALL 语句替换 EXEC CICS 语句。

16.2.2 使用 CEDA 定义资源

CICS 子系统中包含有很多类型的资源。对于不同的应用，将会用到其中不同部分的资源。这些资源通常有以下几种。

- 交易
- 程序

- MAPSET
- 文件
- 队列
- 数据库
- 终端

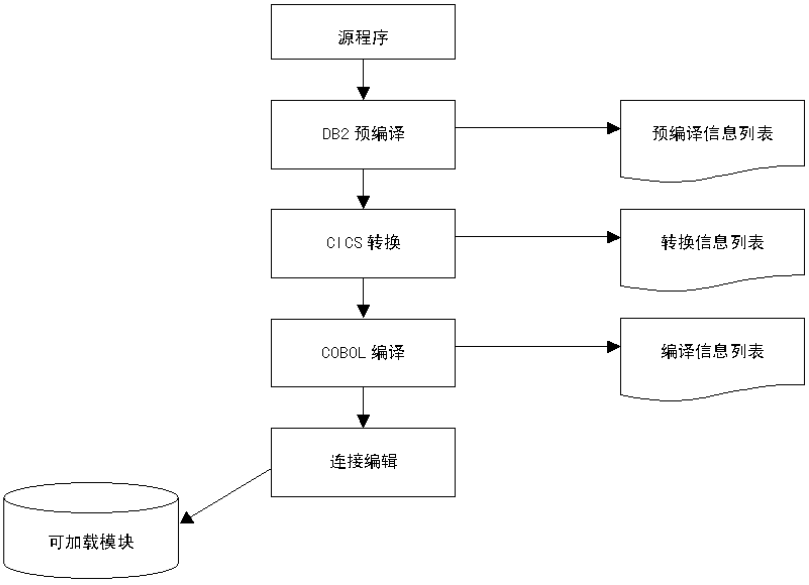

图 16.4　含有 DB2 的 CICS 程序编译流程图

以上这些资源一般来说需要在 CICS 子系统中先定义之后方能使用。CICS 中的资源主要是通过 CEDA 进行定义的。前面曾提到，CEDA 实际上是系统中所预设的一个系统交易。用户可以通过该交易生成的任务，完成定义 CICS 资源的系统事务。

在实际应用中，通常首先需要将在 ISPF 中所开发的程序在 CICS 子系统中进行定义。定义程序的相应操作如下。

```
CEDA  DEFINE  PROGRAM(TESTPGM)   GROUP(TESTGRP)
```

以上操作在 CICS 中定义了程序名为“TESTPGM”的程序。同时，该操作还将这一程序定义在了名为“TESTGRP”的组中。组在 CICS 中是用来将各种相关资源存放在一起的，其本身严格来说并不属于一种 CICS 资源。

以上操作执行后，CICS 系统接下来将会出现一个列表，用于对程序更详细的属性进行定义。这些详细定义中有很多参数都是由系统默认给出的。用户在此可根据实际需要修改这些默认值。以下为其中几条比较常见的程序属性。

- 程序由何种语言所编写。通常，这些语言可以为 COBOL、大型机汇编语言、PL/1、C、RPG 或 LE370。
- 在某个时刻，可以有多少个任务并发使用该程序。
- 该程序的存储位置为何处（通常为主存或是磁盘）。

如果定义成功，在该列表界面倒数第 2 行的首列位置将会显示“DEFINE SUCCESSFUL”

的提示信息。通常情况下，多数提示信息都是在该位置显示的。

当成功定义完程序后，接下来通常需要定义交易。定义交易时，关键需要将该交易同程序相关联起来。定义交易的相应操作如下。

```
CEDA  DEFINE  TRANS(TST1)   PROGRAM(TESTPGM)   GROUP(TESTGRP)
```

以上操作定义了交易名（TRANSID）为“TST1”的一个交易。同时，该操作还将 TST1 交易同之前定义的程序 TESTPGM 相关联了起来。TST1 交易和 TESTPGM 程序一样，也是存放在 TESTGRP 组中的。

以上操作执行后，CICS 系统仍然会出现一个列表，用于对交易进行更详细的定义。不过，此处通常只须关注交易名和该交易关联的程序，其余则采用系统默认值。

通常情况下，以上定义完成后，便可以在安装之后直接通过交易名启用任务完成一定功能了。不过，有时程序中还用到其他一些资源，如文件和终端等。其中定义文件的操作可以如下。

```
CEDA  DEFINE  FILE(TESTFILE)  GROUP(TESTGRP)
```

在定义文件时系统所给出的详细列表中，通常需要关注以下几个关于文件的属性。

- 文件的逻辑名称同物理数据集的对应关系。
- 文件类型。
- 文件访问方式。
- 文件访问权限。
- 文件的密码。
- 文件的保护机制。

此外，当定义终端时，关键需要给出终端的终端号。例如，以下操作将在 TESTGRP 组中定义一个名称为“TESTTEM”的终端。

```
CEDA  DEFINE  TERMINAL(TESTTEM)  GROUP(TESTGRP)
```

当定义完成资源后，在实际应用之前，还需将所定义的资源进行安装。所谓安装，实际也就是将该资源所包含的所有数据读入内存。原因在于 CPU 是只能执行读入内存的程序的。安装也是使用 CEDA 进行的，以下为几段相应的安装操作。

```
CEDA  INSTALL   PROGRAM(TESTPGM)  GROUP(TESTGRP)
CEDA  INSTALL   PROG(*)   GROUP(TESTGRP)
CEDA  INSTALL   GROUP(TESTGRP)
```

以上第一条操作是将 TESTGRP 组中的程序 TESTPGM 进行了安装，第二条操作则将安装 TESTGRP 组中的所有程序，第三条操作将安装 TESTGRP 组中的所有资源。

同时，注意到第二条操作中将“PROGRAM”简写为了“PROG”。实际上，在 CICS 操作中经常会存在着简写。部分常见的原字符和简写后的字符对应关系如下。

- DEFINE：可以简写为 D。
- PROGRAM：可以简写为 PROG。
- TRANSACTION：可以简写为 TRANS。
- FILE：可以简写为 F。
- TERMINAL：可以简写为 TE。
- GROUP：可以简写为 G。

- ALTER：可以简写为 AL。
- INQUIRE：可以简写为 I。
- SET：可以简写为 S。

使用 CEDA 也可对已定义后的资源的各种属性进行修改。例如，可以将某一交易的关联程序改为其他程序，或者将该交易所在的组改为其他组，等等。以下操作可以对 TESTPGM 组中的 TST1 交易的相关属性进行修改。

```
CEDA  ALTER  TRANS(TST1)  G(TESTGRP)
```

该操作执行后，系统将给出一个列表，列表中包含有所指定的 TST1 交易的各种属性信息。当需要修改某一属性值时，直接在该列表中相应的属性位置修改即可。

此外，使用 CEDA 还可对所定义的资源信息进行列表。列表通常可以分为两种情况。一种是对某一资源进行详细列表，另一种是对某一组中的所有资源进行概括的列表。例如，以下操作将对 TESTGRP 组中的 TST1 交易进行详细列表。

```
CEDA  DISPLAY  TRANS(TST1)  G(TESTGRP)
```

以下操作执行后，将会把 TESTGRP 组中的所有资源进行列表。其中所列表的资源属性主要包含资源名称、类型、所在组名以及创建时间。相应操作如下。

```
CEDA  EXPAND  GROUP(TESTGRP)
```

16.2.3 使用 CEMT 查询和设置资源

CEMT 主要用于对定义后的资源进行查询以及设置，通常和 CEDA 的联系最为紧密。例如，以下操作将查询在 CICS 上定义的所有程序资源。

```
CEMT  INQUIRE  PROGRAM(*)
CEMT  I         PROG(*)    /*此条为上条的简写方式，当熟练后通常采用此条表达方式，下同*/
```

以上操作执行后，将会把 CICS 上所有定义过的程序列表出来。该列表相对于第一种方式中的列表信息要简略一些。对于该操作而言，将主要只显示程序的名称、长度、访问权限等相关内容。同时，在列表中的某行程序旁输入“I”，便可以直接对该程序进行安装了。

同时，在实际中常常也使用 CEMT 对当前运行在 CICS 上的所有任务进行查询。对任务进行查询的相应操作如下。

```
CEMT  INQUIRE  TASK(*)
CEMT  I         TAKS(*)
```

最后需要特别注意的是，当原程序在 ISPF 中被修改后，在 CICS 中也应通过 CEMT 进行相应设置。此处的设置为更新设置。若不进行设置，由于之前已将程序安装到了 CICS 内存中，因此 CICS 将仍然运行修改之前的程序。设置方式如下。

```
CEMT  SET  PROGRAM(TESTPGM)  NEW
CEMT  S    PROG(TESTPGM)  NEW
```

以上操作中的“SET”表示此时 CEMT 用于对资源进行设置，而“NEW”则表示该设置为更新设置。当使用以上方式在 CICS 中更新程序后，可以通过此时屏幕上显示的程序长度来判断是否更新成功。通常情况下，如果程序长度有变化，说明更新成功了。

实际上，在 CICS 中最常使用的操作便是通过 CDEA 定义程序和交易。同时，当程序有

修改后，使用 CEMT 在 CICS 中对相应程序资源进行更新。

16.2.4 使用 CEDF 调试程序

当定义并安装完成相关资源后，便可以通过输入交易号以生成任务，启动并运行所编写的应用程序了。最后所剩下的工作便是对程序进行调试。调试程序主要是通过 CEDF 实现的。使用 CEDF 调试程序与使用 CEMT 更新程序往往是交替进行的。同时，CEDF 还可调试在另一终端上正在运行的程序。

CEDF 相当于 CICS 中的一个程序调试器。使用 CEDF 时，首先在屏幕左上角输入“CEDF”，然后清空屏幕，输入被调试程序所在交易的交易号。此时，系统将首先显示该交易所生成的任务最初的 EIB 信息。使用功能键 PF7 和 PF8 可上翻或下翻输出信息，使用右 Ctrl 键可执行下一步操作。

CEDF 的下一步操作将根据程序中的 CICS 命令采用单步执行的方式以进行调试。在单步执行中，屏幕上将只显示 CICS 命令，而不显示作为宿主语言的 COBOL 语句。例如，以下可以为在单步执行中显示的一条 CICS 命令。

```
EXEC  CICS  SEND  MAP
  MAP('TETS')
  MAPONLY
  LENGTH(8)
  CURSOR
  TERMINAL
  FREEKB
  ERASE
```

以上命令实现的功能是在屏幕上显示一张 MAP。关于 MAP，将在后面的章节中详细讲解。此处需要注意的是，对于该命令中的 MAP 名称“TETS”，在 CEDF 中是可以直接更改的。例如，可以在此将其改为“TEST”，从而显示另一张 MAP。

即使用 CEDF 不仅可以显示程序在单步执行时的信息，也可以在执行过程中对其进行干预。这种干预体现了 CEDF 的交互式调试功能。除以上干预方式外，CEDF 通常还有以下几种干预方式。

- 更改异常条件。
- 通过“NOOP”或“NOP”跳过某些 CICS 命令。
- 异常终止（ABEND）一个任务。

当程序中没有 CICS 命令时，同样也可以使用 CEDF 进行调试跟踪。此时，需要在程序中设置相应的单步断点。以下为在 CICS TS 中的设置方式。

```
EXEC  CICS  ENTER  TRACENUM(nn)  END-EXEC.        /*nn 为断点编号，通常用数字表示*/
```

总之，在 CEDF 中可随时查看当前 EIB 信息、任务工作存储节的信息、以及任意地址空间的信息。这些信息对于程序的调试都是十分有用的。

16.3 CICS 在 COBOL 中的基本应用

CICS 在实际应用方面的功能最终还是通过程序实现的。这些程序同操作 DB2 的程序类似，也是将某一程序语言作为宿主语言，并将 CICS 命令嵌入其中。相关宿主语言可以为

COBOL 语言、大型机汇编语言、PL/1 语言等。目前最常用的还是 COBOL 语言。

16.3.1 基本程序结构

基于 CICS 的 COBOL 程序同样是通过 EXEC 标识符将 CICS 命令嵌入其中的。以下为 CICS 命令在 COBOL 中的表示方式。

```
EXEC  CICS
      ……                              /*此处为嵌入的相关 CICS 命令*/
     END-EXEC.
```

此外，基于 CICS 的 COBOL 程序在结构上同普通 COBOL 程序略有不同。在基于 CICS 的 COBOL 程序中，和文件有关的节都是不应存在的。程序中用到的文件资源都将由 CICS 统一定义。以下为基于 CICS 的 COBOL 程序的基本结构。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  XXX.
*
 ENVIRONMENT  DIVISION.
 × CONFIGURATION  SECTION.          /*不应有配置节*/
 × INPUT-OUTPUT  SECTION.           /*不应有输入输出节*/
*
 DATA  DIVISION.
 × FILE  SECTION.                   /*不应有文件节*/
 WORKING-STORAGE  SECTION.
 LINKAGE  SECTION.                  /*通常需要有连接节*/
*
 PROCEDURE  DIVISION.
```

此外，当程序中的 CICS 逻辑部分结束时，需要使用 RETURN 命令将控制权移交回系统。在程序的最后，通常使用 GOBACK 表示终止，而并非 STOP RUN。这样可避免一些编译方面的警告或错误。完整的程序框架如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  XXX.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 ……
 LINKAGE  SECTION.
 ……
*
 PROCEDURE  DIVISION.
      ……
       EXEC  CICS  RETURN  END-EXEC.
      GOBACK.
```

16.3.2 使用 CICS 进行输入输出

基于 CICS 的 COBOL 程序最基本的功能便是输入和输出。在 CICS 中，输入是通过 RECEIVE 命令实现的，输出是通过 SEND 命令实现的。下面分别进行讲解。

1．使用 RECEIVE 命令进行输入

使用 RECEIVE 命令进行输入时，关键需要指定输入的存储位置以及输入数据的长度。例如，以下代码将通过终端接受输入数据，并存放在变量 INAREA 中。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  INPGM.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  INFLDS.
     05   INLEN     PIC  S9(4)  USAGE  IS  COMP.
 01  INAREA.
     05   DATA-1   PIC X(5).
     05   FILLER   PIC XX.
     05   DATA-2   PIC X(5).
 ......
*
 PROCEDURE  DIVISION.
       MOVE   12   TO  INLEN.
       EXEC  CICS  RECEIVE  INTO(INAREA)
                            LENGTH(INLEN)
                            END-EXEC.
......
       EXEC  CICS  RETURN  END-EXEC.
       GOBACK.
```

以上 RECEIVE 命令中，LENGTH 选项用于指定所接受数据的最大长度。当输入数据长度小于该长度时，CICS 将接受输入数据的实际长度；当输入数据长度大于该长度时，CICS 将会产生一个异常。

在 VS COBOL Ⅱ中，LENGTH 选项则不用指定。VS COBOLⅡ中有一个专门的地址寄存器，CICS 将根据该寄存器判断接受数据的长度。

2．使用 SEND 命令进行输出

同 RECEIVE 命令类似，使用 SEND 命令进行输出关键需要指定输出数据的存储位置，以及数据长度。例如，以下代码将输出一段“HELLO WORLD”字符。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  OUTPGM.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 ......
01  MSG    PIC  X(80).
*
 PROCEDURE  DIVISION.
      ......
       MOVE   'HELLO  WORLD'   TO  MSG.
```

```
        EXEC  CICS  SEND    FROM(MSG)
                            LENGTH(20)
                            END-EXEC.
        ......
        EXEC  CICS  RETURN  END-EXEC.
        GOBACK.
```

需要注意的是，使用 SEND 命令进行输出时，LENGTH 选项中应该为实际数据，而并非变量。这点和 RECEIVE 命令是有所不同的。此外，输出数据的起始位置在默认情况下，将为当前光标在屏幕中的位置。关于 CICS 屏幕中的光标，将在后面详细讲解到。

16.3.3 输入过程中的异常处理

CICS 中的异常处理有很多种情况，此处主要只针对在输入过程中的异常处理。异常处理是通过在相应的 CICS 命令中使用 RESP 选项实现的。RESP 选项后为一个 8 位长度的有符号数，用以表明异常的种类。例如，以下代码在输入数据大于指定长度时，将对所抛出的异常进行处理。处理方式为在屏幕上输出相应提示信息。代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  RESPGM.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  WKFLDS.
     05   INLEN      PIC  S9(4)  USAGE  IS  COMP.
     05   ERR-CODE  PIC  S9(8)  COMP.
 01  INAREA.
     05   DATA-1   PIC X(5).
     05   FILLER   PIC XX.
     05   DATA-2   PIC X(5).
 ......
*
 PROCEDURE  DIVISION.
     MOVE   12   TO  INLEN.
     EXEC  CICS  RECEIVE  INTO(INAREA)
                          LENGTH(INLEN)
                          RESP(ERR-CODE)          /*此处捕获异常*/
                          END-EXEC.
     IF    ERR-CODE  =  DFHRESP(LENGERR)
           MOVE  'INPUT  DATA  TOO  LONG'  TO  MSG
     EXEC  CICS  SEND    FROM(MSG)
                         LENGTH(20)
                         END-EXEC
      ......
      EXEC  CICS  RETURN  END-EXEC.
       GOBACK.
```

由以上代码可以看到，判断该异常的类型实际上是通过以下代码实现的。

```
IF    ERR-CODE  =  DFHRESP(LENGERR)
     ......
```

其中 ERR-CODE 是由用户定义的一个变量。该变量通过 RESP 选项可得到所抛出异常的

编号。将该异常编号与 CICS 系统中的 DFHRESP 相应异常编号进行比较，便可以判断出该异常的类型。除以上代码中出现的异常外，通常还有以下几种异常类型。

```
IF   ERR-CODE  =  DFHRESP(TERMERR)        /*终端错误异常*/
……
IF   ERR-CODE  =  DFHRESP(EOC)            /*链结束（End of Chain）异常*/
     ……
IF   ERR-CODE  =  DFHRESP(ERROR)          /*错误异常*/
     ……
IF   ERR-CODE  =  DFHRESP(NORMAL)         /*无异常*/
     ……
```

需要注意的是，DFHRESP（ERROR）异常属于一种类型的异常，并不代表全部异常。判断一条 CICS 命令中存在任何异常，可以使用如下方式进行。

```
IF   ERR-CODE  NOT  EQUAL  DFHRESP(NORMAL)    /*当存在任何异常时，执行相关操作*/
     ……
```

16.3.4 输出过程中的光标定位

前面曾讲到，CICS 中默认的输出是以当前光标所在位置作为起始位置的。不过，光标的实际位置在程序中也是可以进行更改的。若要在程序中更改光标的位置，需要提供更改后光标的位置值。在 CICS 支持的标准终端屏幕上，光标的位置值如图 16.5 所示。

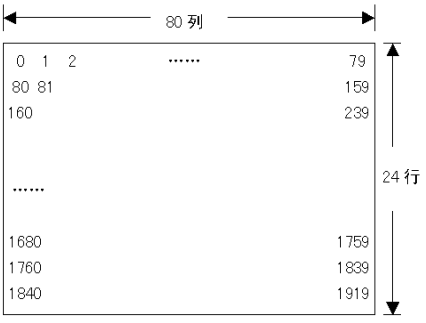

图 16.5 CICS 标准终端屏幕上光标的位置值

由上图可以看出，光标的位置值是一个一维数字，而不是由行数和列数形成的一个二维向量。通常在程序要求中只给出直观的行列数，需要通过计算将其转换为程序中的光标位置值。例如，以下代码将从屏幕中倒数第二行的第一列开始输出相应信息。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  CURSORPGM.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 ……
 01  MSG    PIC  X(80).
*
 PROCEDURE  DIVISION.
      ……
```

```
        MOVE   'MESSAGE  OUTPUT :'   TO  MSG.
        EXEC  CICS  SEND  CONTROL   CURSOR(1760)          /*此处为光标定位*/
                                   END-EXEC
        EXEC  CICS  SEND             FROM(MSG)
                                    LENGTH(20)
                                    END-EXEC.
        ......
         EXEC  CICS  RETURN  END-EXEC.
          GOBACK.
```

这样，输出信息的位置便可由程序任意指定，而不必受之前光标所在位置的影响了。此外，如果在 SEND 命令中同时加上 ERASE 选项，则屏幕将被清屏，光标位置将被刷新。刷新后的光标位置为其初始位置，相应位置值为 0。

16.3.5 获取 CICS 的终端信息

在第一节中曾讲到，每一个任务都有一个与之对应的 EIB。CICS 的终端信息正是通过 EIB 得到的。

需要注意的是，由于任务可以在不同的终端上启动，因此得到的终端信息也是不同的。即终端信息是基于任务而言的，这点同每一任务中 EIB 的惟一性是相对应的。

此处所说的终端信息，主要是指终端在 CICS 中的设备编号。终端编号是通过 EIB 中的信息 EIBTRMID 提供的。以下程序将直接通过 EIBTRMID 得到相应的终端设备编号，并从屏幕第二行的第一列开始将其输出。

```
     IDENTIFICATION  DIVISION.
     PROGRAM-ID.  TERMPGM1.
    *
     ENVIRONMENT  DIVISION.
    *
     DATA  DIVISION.
     WORKING-STORAGE  SECTION.
    01  MSG1    PIC  X(80).
    *
     PROCEDURE  DIVISION.
         STRING  'TERMINAL  ID  IS:  '  EIBTRMID      /*此处通过EIBTRMID得到终端设备编号*/
            DELIMITED  BY  SIZE  INTO  MSG1
         EXEC  CICS  SEND  CONTROL  CURSOR(80)
                                   END-EXEC
         EXEC  CICS  SEND           FROM(MSG1)
                                   LENGTH(20)
                                   END-EXEC
         EXEC  CICS  RETURN  END-EXEC.
         GOBACK.
```

不过，以上这种直接输出结果信息的方式并不能体现出 CICS 的交互功能，在实际中较为少见。通常情况下，实际的 CICS 程序往往是根据用户输入的请求而执行相应操作的。例如，以下程序将只在用户输入“TERM”的请求之后，方获取并显示相应终端信息。

```
     IDENTIFICATION  DIVISION.
     PROGRAM-ID.  TERMPGM2.
    *
     ENVIRONMENT  DIVISION.
    *
```

```
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  WKAREA.
    05   LEN         PIC  S9(4)  USAGE  IS  COMP.
    05   ERR-CODE   PIC  S9(8)  COMP.
 01  INAREA.
    05   TRANID     PIC   X(4).
    05   FILLER      PIC   X.
    05   REQUEST    PIC   X(4).
 01  MSG2    PIC  X(80).
*
 PROCEDURE  DIVISION.
     MOVE  9  TO  LEN.
     EXEC  CICS  RECEIVE          INTO(INAREA)
                                  LENGTH(LEN)
                                  RESP(ERR-CODE)
                                  END-EXEC
     IF   ERR-CODE  =  DFHRESP(LENGERR)
          MOVE  'INPUT  DATA  TOO  LONG: MAX  IS  9  CHARS'
          TO  MSG2
    EXEC  CICS  SEND  CONTROL   CURSOR(1760)
                                END-EXEC
    EXEC  CICS  SEND            FROM(MSG2)
                                LENGTH(40)
                                END-EXEC
    EXEC  CICS  RETURN  END-EXEC.
   IF   REQUEST  =  'TERM'
   STRING  'TERMINAL  ID  IS:  '  EIBTRMID
      DELIMITED  BY  SIZE  INTO  MSG2
   EXEC  CICS  SEND  CONTROL   CURSOR(80)
                               END-EXEC
   EXEC  CICS  SEND            FROM(MSG2)
                               LENGTH(20)
                               END-EXEC
    EXEC  CICS  RETURN  END-EXEC.
     GOBACK.
```

需要注意的是，将以上程序关联到交易中后，用户首先需要输入相应的交易编号以启动一个任务。因此，在以上代码定义的输入区域 INAREA 中包含两个输入变量。其中前一个变量 TRANID 对应输入的交易编号，后一个变量 REQUEST 对应输入的请求。

同时，在以上代码中存在两处 RETURN 命令。由于 RETURN 命令表示结束任务逻辑，将控制权移交 CICS 系统。因此，当以上程序运行时满足第一个条件，并执行第一个 RETURN 命令后，任务将直接结束。此时，程序将不会执行第一条 RETURN 命令之后的内容。

16.3.6 获取 CICS 的时间信息

获取 CICS 的时间信息也是关于 CICS 的一项最基本的应用。该应用主要分为两步操作。其中第一步是通过 ASKTIME 得到时间的数据。第二步是通过 FORMATTIME 得到时间的格式。例如，以下代码将根据用户的请求而获得相应的时间信息。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  TIMEPGM.
*
```

```
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  WKAREA.
     05   LEN        PIC  S9(4)   USAGE  IS  COMP.
     05   ERR-CODE   PIC  S9(8)   COMP.
     05   GETTIME    PIC  S9(15)  COMP-3.      /*该变量用于接受时间数据*/
     05   TIMEOUT    PIC  X(8).                /*该变量用于接受时间格式，并输出时间信息*/
 01  INAREA.
     05   TRANID     PIC   X(4).
     05   FILLER     PIC   X.
     05   REQUEST    PIC   X(4).
 01  MSG    PIC  X(80).
*
 PROCEDURE  DIVISION.
       MOVE  9  TO  LEN.
       EXEC  CICS  RECEIVE          INTO(INAREA)
                                    LENGTH(LEN)
                                    RESP(ERR-CODE)
                                    END-EXEC
       IF   ERR-CODE  =  DFHRESP(LENGERR)
            MOVE  'INPUT  DATA  TOO  LONG: MAX  IS  9  CHARS'
              TO  MSG
       EXEC  CICS  SEND  CONTROL  CURSOR(1760)
                                  END-EXEC
       EXEC  CICS  SEND           FROM(MSG)
                                  LENGTH(40)
                                  END-EXEC
         EXEC  CICS  RETURN  END-EXEC.
       IF   REQUEST  =  'TIME'
       EXEC  CICS  ASKTIME       ABSTIME(GETTIME)
                                 END-EXEC
       EXEC  CICS  FORMATTIME    ABSTIME(GETTIME)
                                 TIMESEP
                                 TIME(TIMEOUT)
                                 END-EXEC
       STRING  'TIME  IS:  '  TIMEOUT
          DELIMITED  BY  SIZE  INTO  MSG
       EXEC  CICS  SEND  CONTROL   CURSOR(80)
                                   END-EXEC
       EXEC  CICS  SEND            FROM(MSG)
                                   LENGTH(20)
                                   END-EXEC
       EXEC  CICS  RETURN  END-EXEC.
       GOBACK.
```

假设在 CICS 中将以上程序关联到 TRANSID 为“TM01”的交易。则用户若要启动相应任务并提出显示时间的请求，可输入以下一段命令。

```
TM01 TIME
```

此时，系统将根据输入的“TM01”启动相应交易的一个任务。该任务便调用了以上程序。之后，系统再根据输入的“TIME”执行程序中相应的分支部分，最终显示出系统当前的时间。时间以小时制表示，以下为运行后屏幕上的一种显示情况。

```
TM01 TIME
TIME  IS:  20:05:12
```

在以上代码中，关于时间信息的获取，主要是通过两条 CICS 命令实现的。这两条 CICS 命令是该程序的关键部分，现将其提取如下。

```
......
EXEC  CICS  ASKTIME       ABSTIME(GETTIME)
                          END-EXEC
EXEC  CICS  FORMATTIME    ABSTIME(GETTIME)
                         TIMESEP
                         TIME(TIMEOUT)
                         END-EXEC
    ......
```

关于第一条 CICS 命令 ASKTIME，主要有以下几点需要注意。

- 该命令含有一个 ABSTIME 选项，该选项中的内容应为一个 15 位长度的有符号整型数变量。该命令将得到的时间信息数据存放在这一变量中。
- 该命令可以更新 CICS 中的时间信息。
- 该命令可以更新 EIB 中的信息 EIBTIME 以及 EIBDATE。其中 EIBTIME 反映了时间信息，而 EIBDATE 则反映了日期信息。关于日期信息，将在下一小节中讲到。
- 当任务启动后，该命令将反映时间戳（time stamp）的信息。

第二条 CICS 命令 FORMATTIME 在此处用来表示接收的为时间信息，同时设置时间信息的格式。对应于以上用法，该命令总共需要包含有 3 个选项。这 3 个选项分别为 ABSTIME、TIMESEP 以及 TIME。其各自的功能分别如下。

- ABSTIME：用来指明时间信息数据的来源。在以上程序中，该选项表示时间信息数据来源于变量 GETTIME。
- TIMESEP：用来表明接收的为时间信息。这一选项十分重要。若指明的类型错误，系统有可能会接收日期信息，从而得不到预想的结果。
- TIME：用来设置时间信息的格式，同时结合时间数据，一并存入变量 TIMEOUT 中。此时，该变量中的内容将为所求的结果信息。

16.3.7 获取 CICS 的日期信息

获取 CICS 的日期信息同获取时间信息比较类似。日期信息也是依次通过 CICS 的命令 ASKTIME 和 FORMATTIME 得到的。不过在 FORMATTIME 中，需要指明获取的为日期信息，同时设置日期信息的相应输出格式。例如，以下程序将用于获取日期信息。

```
IDENTIFICATION  DIVISION.
PROGRAM-ID.  DATEPGM.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  WKAREA.
     05   LEN         PIC  S9(4)   USAGE  IS  COMP.
     05   ERR-CODE    PIC  S9(8)   COMP.
     05   GETDATE     PIC  S9(15)  COMP-3.     /*该变量用于接受日期数据*/
```

```
       05   DATEOUT     PIC  X(10).              /*该变量用于接受日期格式，并输出日期信息*/
    01  INAREA.
       05   TRANID      PIC   X(4).
       05   FILLER      PIC   X.
       05   REQUEST     PIC   X(4).
    01  MSG    PIC  X(80).
   *
    PROCEDURE  DIVISION.
        MOVE  9  TO  LEN.
        EXEC  CICS  RECEIVE          INTO(INAREA)
                                     LENGTH(LEN)
                                     RESP(ERR-CODE)
                                     END-EXEC
        IF   ERR-CODE  =  DFHRESP(LENGERR)
             MOVE  'INPUT  DATA  TOO  LONG: MAX  IS  9  CHARS'
               TO  MSG
        EXEC  CICS  SEND  CONTROL  CURSOR(1760)
                                   END-EXEC
        EXEC  CICS  SEND           FROM(MSG)
                                   LENGTH(40)
                                   END-EXEC
          EXEC  CICS  RETURN  END-EXEC.
        IF   REQUEST  =  'DATE'
        EXEC  CICS  ASKTIME        ABSTIME(GETDATE)
                                   END-EXEC
        EXEC  CICS  FORMATTIME     ABSTIME(GETDATE)
                                   DATESEP
                                   YYYYMMDD(DATEOUT)
                                   END-EXEC
        STRING  'DATE  IS:  '  DATEOUT
           DELIMITED  BY  SIZE  INTO  MSG
        EXEC  CICS  SEND  CONTROL   CURSOR(80)
                                    END-EXEC
        EXEC  CICS  SEND            FROM(MSG)
                                    LENGTH(20)
                                    END-EXEC
        EXEC  CICS  RETURN  END-EXEC.
         GOBACK.
```

以上程序中，将日期的输出格式设置为了“YYYYMMDD”。其中“YYYY”代表 4 位数字的年份，“MM”代表 2 位数字的月份，“DD”代表 2 位数字的日期。在 CICS TS 版本中，一共支持以下几种日期输出格式。

- YYYYMMDD
- YYYYDDMM
- DDMMYYYY
- MMDDYYYY
- YYYYDDD

需要注意的是，最后一种格式使用“DDD”表示该日期在一年中的绝对天数。例如，对于平年的 12 月 31 日，此处使用“365”进行表示。

最后，以下这段程序综合了终端信息、时间信息、日期信息的获取与输出。将该程序连接到相应交易后，用户便可利用这一个交易，通过输入不同的请求而得到不同的信息。该程

序代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  MIXPGM.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  WKAREA.
     05   LEN         PIC  S9(4)   USAGE  IS  COMP.
     05   ERR-CODE    PIC  S9(8)   COMP.
     05   TSTAMP      PIC  S9(15)  COMP-3.
     05   TIMEOUT     PIC  X(8).
     05   DATEOUT     PIC  X(10).
 01  INAREA.
     05   TRANID      PIC   X(4).
     05   FILLER      PIC   X.
     05   REQUEST     PIC   X(4).
 01  MSG    PIC  X(80).
*
 PROCEDURE  DIVISION.
     MOVE  9  TO  LEN.
     EXEC  CICS  RECEIVE          INTO(INAREA)
                                  LENGTH(LEN)
                                  RESP(ERR-CODE)
                                  END-EXEC
     IF   ERR-CODE  =  DFHRESP(LENGERR)
          MOVE  'INPUT  DATA  TOO  LONG: MAX  IS  9  CHARS'
            TO  MSG
     EXEC  CICS  SEND  CONTROL  CURSOR(1760)
                                END-EXEC
     EXEC  CICS  SEND           FROM(MSG)
                                LENGTH(40)
                                END-EXEC
     EXEC  CICS  RETURN  END-EXEC.
*
      IF   REQUEST  =  'TERM'
      STRING  'TERMINAL  ID  IS:  '  EIBTRMID
         DELIMITED  BY  SIZE  INTO  MSG
      PERFORM  100-OUTPUT-RESULT
        EXEC  CICS  RETURN  END-EXEC.
*
      IF   REQUEST  =  'TIME'
      EXEC  CICS  ASKTIME       ABSTIME(GETTIME)
                                END-EXEC
      EXEC  CICS  FORMATTIME    ABSTIME(GETTIME)
                                TIMESEP
                                TIME(TIMEOUT)
                                END-EXEC
      STRING  'TIME  IS:  '  TIMEOUT
         DELIMITED  BY  SIZE  INTO  MSG
      PERFORM  100-OUTPUT-RESULT
         EXEC  CICS  RETURN  END-EXEC.
*
```

```
        IF   REQUEST  =  'DATE'
        EXEC  CICS  ASKTIME        ABSTIME(GETTIME)
                                   END-EXEC
        EXEC  CICS  FORMATTIME     ABSTIME(GETTIME)
                                   DATESEP
                                    DDMMYYYY(DATEOUT)
                                    END-EXEC
        STRING  'DATE  IS:  '  DATEOUT
            DELIMITED  BY  SIZE  INTO  MSG
        PERFORM  100-OUTPUT-RESULT
          EXEC  CICS  RETURN  END-EXEC.
            GOBACK.
    100-OUTPUT-RESULT.
        EXEC  CICS  SEND  CONTROL  CURSOR(80)
                                   END-EXEC
        EXEC  CICS  SEND           FROM(MSG)
                                   LENGTH(20)
                                   END-EXEC.
```

16.4 伪会话程序

伪会话程序是 CICS 开发中十分重要的一个概念，绝大多数 CICS 上的程序都是采用的伪会话程序。伪会话程序充分体现了 CICS 的交互能力，形成了区别于批处理作业最大的特征。同时，伪会话程序也避免了长时间的独占任务资源，从而提供了程序的并发执行能力。

16.4.1 伪会话程序的基本概念

伪会话程序是相对于会话程序发展而来的。会话程序将会不间断地等待从终端接受数据，直到程序结束。而伪会话程序则只在终端有输入时才激活，并不会始终占用资源以等待从终端接受数据。如图 16.6 所示，是这二者之间的区别。

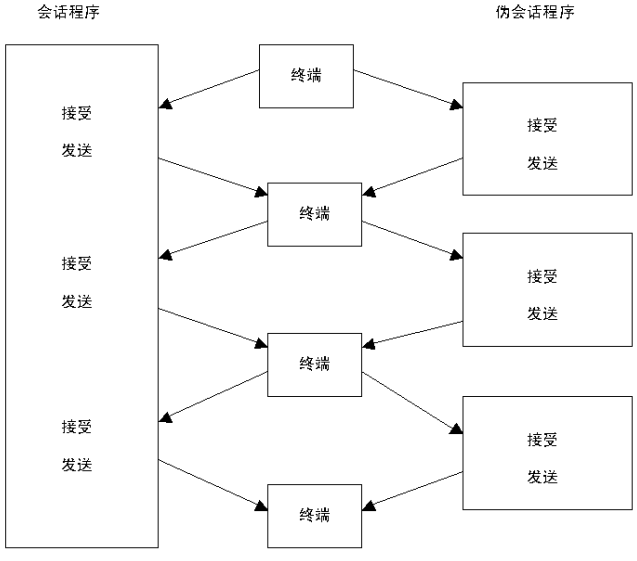

图 16.6 会话程序与伪会话程序的区别

由此可见，伪会话程序相对于会话程序最大的好处在于不会占据任务资源等待用户的输入。例如，在银行 ATM 自动取款机取款时，用户输入密码后有时会考虑究竟取多少现金。在这一用户考虑的过程中，系统便可以执行其他操作，而不必等待用户的输入。当用户考虑好后输入取款数目时，系统再重新启动一个任务继续完成该项事务。

由上图还可以看到，会话程序在此只启动了 1 个任务，而伪会话程序则启动了 3 个任务。当然，由于这 3 个任务并没有等待用户在终端上的输入，因此总的执行时间反而较少。

伪会话的多个任务之间存在着一个先后顺序。其中前一个任务将把参数传递给后一个任务。在 CICS 系统中，存在着一块特殊的区域，专用于实现参数的传递。该区域称作 COMMARE。在接受参数的程序中，相应的存在一块称作 DFHCOMMAREA 的区域，用于接受传来的数据。如图 16.7 所示，反映了伪会话程序中参数的传递情况。

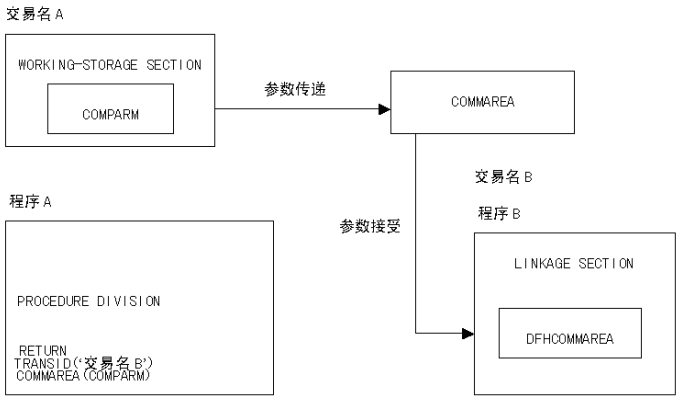

图 16.7　伪会话程序中的参数传递

由上图可以看出，程序 A 通过 RETURN 命令中的 TRANSID 选项指明下一个交易为交易 B。同时，程序 A 还通过 RETURN 命令中的 COMMAREA 选项将 COMPARM 作为参数进行传递。COMPARM 首先被传递到 COMMAREA 区域中。然后，交易 B 所关联的程序 B 再从该区域接受所传递的参数，并存放在 DFHCOMMAREA 中。

下面对程序 A 和程序 B 分别给出具体代码，以便更好地反映实际情况。其中程序 A 与参数传递相关的部分代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  PGMA.                      /*此处使用 PGMA 表示程序 A*/
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  COMPARM.                            /*COMPARM 为所要传递的参数，其名称可以任意*/
     05  S-STATUS        PIC  X.
     05  S-GRP.
         10  S-ONE      PIC  X(49).
         10  S-TWO      PIC  X(30).
 ......
 PROCEDURE  DIVISION.
     ......
```

```
       MOVE  data  TO  COMPARM.
       ......
       EXEC  CICS  RETURN  TRANSID('TRNB')     /*此处使用 TRNB 表示交易名 B*/
                           COMMAREA(COMPARM)
                           LENGTH(80)
                           END-EXEC.
         GOBACK.
```

程序 B 与之相关的部分代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  PGMB.                          /*此处使用 PGMB 表示程序 B*/
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
......
 LINKAGE  SECTION.
 01  DFHCOMMAREA.
     05  R-STATUS        PIC  X.
     05  R-GRP.
         10  R-ONE      PIC  X(49).
         10  R-TWO      PIC  X(30).
......
 PROCEDURE  DIVISION.
      ......
       Process  data  from  DFHCOMMAREA.
       ......
         GOBACK.
```

需要注意的是，COMMAREA 是 CICS 系统上的一块区域。而 DFHCOMMAREA 则是在接受参数的程序的连接节中定义的一块区域。在参数传递过程中，CICS 将自动把 COMMAREA 中的参数传递到 DFHCOMMAREA 之中。

最后需要说明的是，上例中的程序 A 与程序 B 既可以为不同的程序，也可以为相同的程序。对于这两种情况在实际中的应用，将在下面两小节中分别进行讲解。

16.4.2 RETURN 到不同的程序

结合前面所讲的伪会话程序的基本概念，下面直接通过具体实例说明先后任务对应不同程序的情况。不妨假设 CICS 中某两个交易及其各自关联的程序分别如下。

- 交易 TRN0，关联的程序为 PGM0。
- 交易 TRN1，关联的程序为 PGM1。

以上面这两个交易为例，如图 16.8 所示，反映了伪会话程序中先后任务对应不同程序的情况。

下面给出程序 PGM0 和 PGM1 的具体示例代码，以说明该方式在实际中通常的应用途径。假设 PGM0 的代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  PGM0.
*
 ENVIRONMENT  DIVISION.
*
```

```
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  COMSTART.
    05  STATUS-I      PIC  X.
    05  PASSWORD-I    PIC  X(4).
 01  MSG0             PIC  X(80).
*
 PROCEDURE  DIVISION.
      MOVE  'Y'   TO  STATUS-I.
      MOVE  'TEST'  TO  PASSWORD-I.
      MOVE  'PLS  ENTER  THE  PASSWORD: '  TO  MSG0.
      EXEC  CICS  SEND  CONTROL   CURSOR(80)
                                  END-EXEC
      EXEC  CICS  SEND            FROM(MSG0)
                                   LENGTH(40)
                                   END-EXEC
      EXEC  CICS  RETURN  TRANSID('TRN1')
                          COMMAREA(COMPARM)
                          LENGTH(5)
                          END-EXEC.
        GOBACK.
```

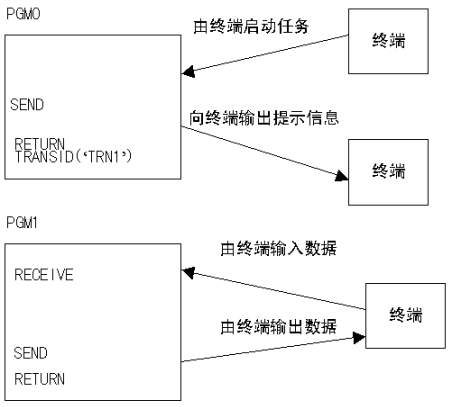

图 16.8　先后任务对应不同的程序

该程序所对应的任务启动后，将在屏幕第二行显示相应提示信息，要求用户输入一个密码。然后，该任务便结束，并通过 RETURN 命令指定下一个调用的任务所对应的交易为 TRN1。TRN1 将在用户输入数据时自动启动相应任务。

假设交易 TRN1 所关联的程序 PGM1 的代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  PGM1.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  PW-GET   PIC  X(4).
 01  INL       PIC  S9(4)   USAGE  IS  COMP.
 01  MSG1     PIC  X(80).
 LINKAGE  SECTION.
```

```
 01  DFHCOMMAREA.
     05  STATUS-C       PIC  X.
     05  PASSWORD-C     PIC  X(4).
*
 PROCEDURE  DIVISION.
      MOVE  4  TO  INL.
      IF  STATUS-C = 'Y'
         EXEC  CICS  RECEIVE       INTO(PW-GET)
                                   LENGTH(INL)
                                   END-EXEC
         IF  PW-GET = PASSWORD-C
            MOVE  'PASSWORD  CORRECT; WELCOME! '  TO  MSG1
         ELSE
            MOVE  'WRONG PASSWORD!'  TO  MSG1
         END-IF
      ELSE
         MOVE  'SYSTEM  NOT  AVAILABLE  NOW'  TO  MSG1
      END-IF.
      EXEC  CICS  SEND  CONTROL    CURSOR(160)
                                   END-EXEC
      EXEC  CICS  SEND             FROM(MSG1)
                                   LENGTH(80)
                                   END-EXEC
      EXEC  CICS  RETURN  END-EXEC.
        GOBACK.
```

以上程序利用了从 PGM0 中传来的参数，并且分 3 种不同的情况进行了相应的结果输出。这 3 种情况分别为密码正确、密码错误以及系统当前不可用。

16.4.3 RETURN 到相同的程序

该方式就是将同一个交易中的关联程序既作为参数的发送者，又作为参数的接受者。假设某一交易为 ROLL，其关联的程序为 ROLLPGM。根据这一条件，如图 16.9 所示，反映了伪会话程序中先后任务对应同一个程序的情况。

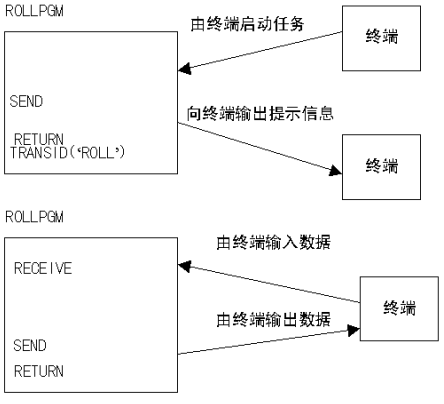

图 16.9　先后任务对应相同的程序

需要注意的是，在该方式下，参数的传递将形成一个闭环。同时，在第一次启动任务时，是没有参数传递进来的，因此此时 COMMAREA 中的内容将为空。当 COMMAREA 中的内

容为空时，该任务的 EIB 信息之一 EIBCALEN 将为 0。在实际应用中，需要根据判断 EIBCALEN 的数值将任务的第一次启动和后续启动区分开来。

例如，以下程序采用了 RETURN 到相同程序（即自身）的方式。通过该程序所对应的交易，将可实现任意次输出系统时间的功能。用户每次输入“TIME”时，便显示出当前时间。该操作可循环进行，直到用户输入“EXIT”退出该项事务。相应代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  ROLLPGM.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  COMSTART.
     05  STATUS-I  PIC  X.
 01  REC-AREA.
      05  INL       PIC  S9(4)   USAGE  IS  COMP.
      05  REQUEST   PIC   X(4).
 01  TIME-AREA.
      05  GETTIME      PIC  S9(15)  COMP-3.
      05  TIMEOUT      PIC  X(8).
 01  MSG         PIC  X(80).
 LINKAGE  SECTION.
 01  DFHCOMMAREA.
     05  STATUS-C  PIC  X.
*
 PROCEDURE  DIVISION.
***  FIRST START  ***
       IF  EIBCALEN = 0
       MOVE  'Y'  TO  STATUS-I
       MOVE  'PLS  ENTER  THE  REQUEST'  TO  MSG
        PERFORM   100-SHOW-MSG
       EXEC  CICS  RETURN  TRANSID('ROLL')
                           COMMAREA(COMSTART)
                           LENGTH(1)
                           END-EXEC.
***  SUBSEQUENT START  ***
       IF  STATUS-C = 'N'
          MOVE  'TASK  ENDED'   TO  MSG
          PERFORM   100-SHOW-MSG
          EXEC  CICS   RETURN  END-EXEC.
*
       IF  STATUS-C = 'Y'
           MOVE  4   TO  INL
           EXEC  CICS  RECEIVE       INTO(REQUEST)
                                     LENGTH(INL)
                                     END-EXEC
           IF  REQUEST  =  'TIME'
              MOVE  'Y'  TO  STATUS-I
              EXEC  CICS  ASKTIME        ABSTIME(GETTIME)
                                         END-EXEC
               EXEC  CICS  FORMATTIME     ABSTIME(GETTIME)
                                          TIMESEP
                                         TIME(TIMEOUT)
```

```
                                        END-EXEC
           STRING  'TIME  IS:  '  TIMEOUT
              DELIMITED  BY  SIZE  INTO  MSG
            PERFORM   100-SHOW-MSG
         END-IF
         IF  REQUEST  =  'EXIT'
            MOVE  'N'  TO  STATUS-I
         END-IF
         EXEC  CICS  RETURN  TRANSID('ROLL')
                             COMMAREA(COMSTART)
                             LENGTH(1)
                             END-EXEC.
         GOBACK.
100-SHOW-MSG.
         EXEC  CICS  SEND  CONTROL   CURSOR(80)
                                     END-EXEC
         EXEC  CICS  SEND             FROM(MSG)
                                     LENGTH(40)
                                     END-EXEC.
```

16.5 CICS 中的程序调用

在大型软件项目中，应用系统往往由多个程序所共同组成。程序之间必然存在着一个调用关系。在基于 CICS 的 COBOL 程序中，通常使用 LINK 和 XCTL 命令实现程序调用。下面分别进行讲解。

16.5.1 使用 LINK 命令进行程序调用

使用 LINK 命令调用程序时，两程序将位于一个任务之下。同时，通过 LINK 命令中的选项 COMMAREA 可以实现在调用过程中的参数传递。例如，在以下程序 PGMA 中，将通过 LINK 命令调用程序 PGMB。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  PGMA.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  COMSTART.
     05   STATUS-I       PIC X.
     05   DATA-I         PIC X(19).
 ......
*
 PROCEDURE  DIVISION.
      ......
       EXEC  CICS  LINK  PROGRAM('PGMB')
                         COMMAREA(COMSTART)
                         LENGTH(20)
                         END-EXEC
     ......
     GOBACK.
```

需要注意的是，被调用的程序 PGMB 通过 RETURN 命令将返回到程序 PGMA 中。因此，通过 LINK 命令进行程序调用时，将改变 CICS 当前执行的逻辑层次。如图 16.10 所示，反映了使用 LINK 命令时 CICS 中不同逻辑层次之间的关系。

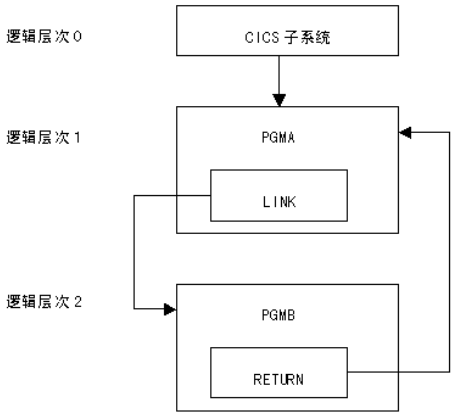

图 16.10　LINK 命令下的逻辑层次关系

16.5.2　使用 XCTL 命令进行程序调用

使用 XCTL 命令调用程序时，两程序仍然处于同一个任务之下。并且，同样也可以通过 COMMAREA 在两程序间传递参数。以下为在程序 PGMC 中，通过 XCTL 调用程序 PGMD 的部分代码。

```
IDENTIFICATION  DIVISION.
PROGRAM-ID.  PGMC.
……
PROCEDURE  DIVISION.
     ……
     EXEC  CICS  LINK      PROGRAM('PGMD')
                           COMMAREA(COMSTART)
                           LENGTH(20)
                           END-EXEC
     ……
```

同时需要注意的是，此时被调用程序 PGMD 通过 RETURN 命令将直接返回到 CICS 系统中。即使用 XCTL 命令调用程序时，将不改变 CICS 当前执行的逻辑层次。如图 16.11 所示，为 XCTL 命令下 CICS 的逻辑层次关系。

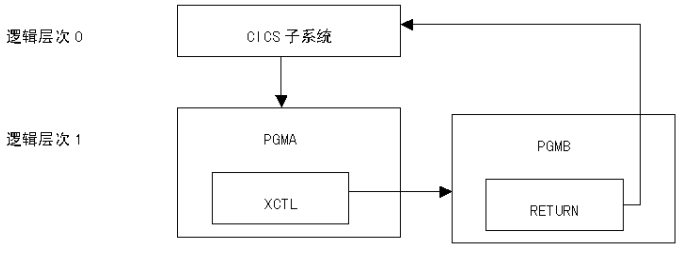

图 16.11　XCTL 命令下的逻辑层次关系

16.6 CICS 中的 MAP

CICS 中的 MAP 实现了 CICS 应用程序的界面。由于 CICS 程序主要用于交互式的处理，因此界面对于 CICS 而言十分重要。在实际开发中，CICS 程序往往会用到 MAP。MAP 可以说是 CICS 产品的一大特色。

16.6.1 MAP 的基本概念

MAP 是形成基于 CICS 程序的界面的一个实体，主要由 BMS（Basic Mapping Support）提供支持。MAP 存放于 MAPSET 中，一个 MAPSET 可以存放一到多个 MAP。通常情况下，一个 MAPSET 中仅存放一个 MAP。因此，有时也可以使用 MAP 指代其所在的 MAPSET。

同时，MAP 实际上是分为两种类型的。其中一种类型为物理 MAP，另一种类型为符号 MAP。理解 MAP 的基本概念，关键是要理解这两种类型 MAP 的区别与联系。物理 MAP 主要用于在屏幕上显示界面信息。其中 BMS 使用物理 MAP 既进行输入操作，也进行输出操作。例如，假设 CICS 中的某一 MAP 如图 16.12 所示。

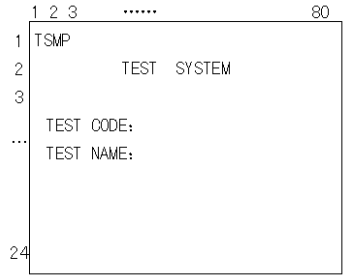

图 16.12 CICS 中的某一个 MAP

假设该 MAP 的名称为“TESTMA”，所存放的 MAPSET 的名称为“MAPSETA”。则该 MAP 所对应的源代码如下。

```
MAPSETA   DFHMSD  TYPE=&SYSPARM, MODE=INOUT, TERM=ALL,
                  LANG=COBOL, TIOAPFX=YES, CTRL=(FREEKB)
TESTMA    DFHMDI   SIZE=(24, 80), LINE=1, COLUMN=1
          DFHMDF   POS=(1, 1), LENGTH=4, INITIAL='TSMP', ATTRB=ASKIP
          DFHMDF   POS=(2, 30), LENGTH=12, INITIAL='TEST  SYSTEM', ATTRB=ASKIP
          DFHMDF   POS=(4, 2), LENGTH=12, INITIAL='TEST CODE:', ATTRB=ASKIP
CODE      DFHMDF   POS=(4, 15), LENGTH=10, ATTRB=(UNPROT, NUM, IC)
          DFHMDF   POS=(4, 26), LENGTH=1, ATTRB=PROT
          DFHMDF   POS=(5, 2), LENGTH=10, INITIAL='TEST NAME:', ATTRB=ASKIP
NAME      DFHMDF   POS=(5, 13), LENGTH=20, ATTRB=UNPROT
          DFHMDF   POS=(5, 34), LENGTH=1, ATTRB=PROT
MSG       DFHMDF   POS=(24, 1), LENGTH=75, ATTRB=ASKIP
          DFHMSD   TYPE=FINAL
          END
```

以上这段代码实际上为一段汇编宏代码。其物理 MAP 所包含内容的汇编宏指令分别如下。

- DFHMSD：指定 MAPSET。
- DFHMDI：指定 MAP。
- DFHMDF：指定物理 MAP 中的常量和变量。其中每行前面有标号的对应 MAP 中的变量，没有标号的对应 MAP 中的常量。常量仅用于输出，其属性和内容不可改变。变量既可用于输出，也可用于输入，其属性和内容可以改变。

- DFHMSD：表明该 MAP 定义结束。

符号 MAP 为一组 COBOL 数据，用于在 COBOL 程序中进行处理。例如，对于以上定义的 MAP，可在 COBOL 程序中使用“COPY MAPSETA”将符号 MAP 复制进来。该命令被编译后，将得到如下符号 MAP。

```
COPY  MAPSETA.
01  TESTMAI.                          /*以下为符号 MAP 中的输入信息*/
    02  FILLER  PIC  X(12).           /*此处表示 TIOA（Terminal Input/Output Area）前缀*/
    02  CODEL   COMP  PIC  S9(4).     /*以下为变量 CODE 的相关输入信息*/
    02  CODEF   PICTURE  X.
    02  FILLER  REDEFINES  CODEF.
      03  CODEA   PICTURE  X.
    02  CODEI   PIC  X(10).
    02  NAMEL   COMP  PIC  S9(4).     /*以下为变量 NAME 的相关输入信息*/
    02  NAMEF   PICTURE  X.
    02  FILLER  REDEFINES  NAMEF.
      03  NAMEA   PICTURE  X.
    02  NAMEI   PIC  X(20).
    02  MSGL   COMP  PIC  S9(4).      /*以下为变量 MSG 的相关输入信息*/
    02  MSGF   PICTURE  X.
    02  FILLER  REDEFINES  MSGF.
      03  MSGA  PICTURE  X.
    02  MSGI  PIC  X(75).
01  TESTMAO  REDEFINES  TESTMAI.      /*以下为符号 MAP 中的输出信息*/
    02   FILLER  PIC  X(12).
    02   FILLER  PICTURE  X(3).
    02   CODEO  PIC  X(10).
    02   FILLER   PICTURE  X(3).
    02   NAMEO  PIC  X(20).
    02   FILLER   PICUTRE  X(3).
    02   MSGO   PIC  X(75).
```

关于符号 MAP 中各变量的后缀，简要说明如下。

- 后缀“L”：半字边界区域，用于保存输入数据的长度（关于半字的概念，将在大型机汇编语言扩展一章中详细讲解）。
- 后缀“F”：当遇到 EOF（Erase to end of Field key）情况时，该区域内容为十六进制数“80”。此时后缀为“L”的区域内容将为 0。
- 后缀“I”：保存实际输入的数据。
- 后缀“A”：包含数据的属性，可以对其进行更改；其中常用属性有：DFHBMUNP（可输入）、DFHBMPRO（不可输入）、DFHBMBRY（高亮显示）、DFHBMDAR（不显示）。
- 后缀“O”：保存实际要输出的数据。

最后，如图 16.13 所示，分别反映了物理 MAP 和符号 MAP 从定义到生成的实际过程。

16.6.2 MAP 的创建

当前 MAP 主要是通过 SDFⅡ（Screen Define Facility Ⅱ）工具所创建的。SDFⅡ工具是 z/OS 上的一个可选产品。该产品自动生成 MAP 的源代码，实现了对于 MAP 所编即所见的功能。当然，在没有提供 SDFⅡ工具的情况下，也可以手工编写源代码生成 MAP。

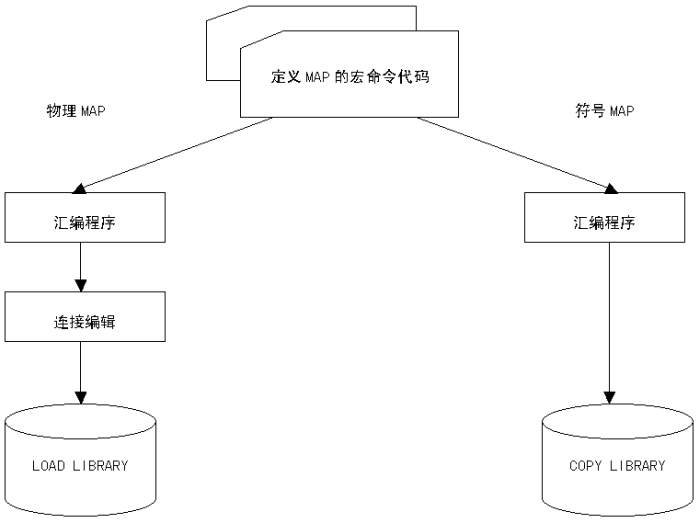

图 16.13　物理 MAP 和符号 MAP 的生成过程

下面只重点介绍如何使用 SDF Ⅱ 工具创建 MAP。此处所说的 MAP，既包含有物理 MAP，也包含有符号 MAP。创建步骤如下。

（1）配置 MAP 的环境。该步骤通常由以下几步操作所完成。

- 从 ISPF 菜单上进入 SDF Ⅱ 工具，通常是输入 9.6。
- 进入 SDF Ⅱ 工具后，在工具主菜单上输入 8（specify lib）用以指定创建该 MAP 所要用到的库。
- 退回主菜单，输入 1，开始编辑 MAP。

（2）编辑 MAP 时通常采用的是所编即所见的方式。此外，在编辑 MAP 的菜单中，还存在着一些相应的功能选项。关于其中常用的几个功能选项分别介绍如下。

- 选项 1：用于指定一些系统参数。如指定 MAP 的长度和宽度等。
- 选项 2：进入该选项后将直接用所编即所见的方式绘制 MAP。
- 选项 3：用于指定 MAP 中的一些特殊字符，对应符号 MAP 中的变量。
- 选项 4：将特殊字符用于 MAP 之中。
- 选项 5：查看在 MAP 中所设定的字符串（即符号 MAP 中的变量）的属性。这些属性包括其层数、长度、类型等。
- 选项 7：用于预览所编辑的 MAP。

（3）生成 MAP。编辑完成 MAP 之后，返回到 SDF Ⅱ 主菜单。通过主菜单中的选项 6 或选项 7 可以生成 MAP 的源代码。通过 ISPF 菜单中的 3.4 选项可以查看到 MAP 源代码所在的数据集。进入该数据集后，可以手工修改 MAP 源代码。接下来，对所生成的源代码进行提交，便可以得到相应的 MAP 了。其中物理 MAP 将存放于 LOAD LIBRARY 中，而符号 MAP 则存放于 COPY LIBRARY 中。

（4）在 CICS 中定义并安装 MAP。MAP 作为 CICS 中的一项资源，同交易与程序一样，也是需要定义和安装的。由于 MAP 在逻辑上是存放于 MAPSET 中的，因此在 CICS 中实际上是对 MAPSET 的定义与安装。假设某一 MAPSET 的名称为“MAPSETA”，则定义与安装

该 MAPSET 的操作如下。

```
CEDA   DEF   MAPSET(MAPSETA)   GROUP(TESTGRP)     /*定义 MAP*/
CEDA   INS    MAPSET(MAPSETA)   GROUP(TESTGRP)    /*安装 MAP*/
```

当定义并安装完成 MAP 后，便可以通过以下操作在 CICS 中看到所创建的 MAP 了。

```
CECI   SEND  MAP(TESTMA)  MAPONLY
```

16.6.3 MAP 的应用

在 COBOL 中对 MAP 的应用，主要可以分为发送 MAP 和接受 MAP 两种情况。其中发送 MAP 就是将 MAP 在 CICS 屏幕中显示出来。发送 MAP 时可选择只发送物理 MAP 还是只发送符号 MAP。其中只发送物理 MAP 的代码如下。

```
EXEC  CICS  SEND  MAP('TESTMA')
                  MAPSET('MAPSETA')
                  MAPONLY                    /*此处指定只发送物理 MAP*/
                  END-EXEC.
```

只发送符号 MAP 的代码如下。

```
EXEC  CICS  SEND  MAP('TESTMA')
                  MAPSET('MAPSETA')
                  DATAONLY                   /*此处指定只发送符号 MAP*/
                  END-EXEC.
```

如果不指定“MAPONLY”和“DATAONLY”选项，CICS 将把物理 MAP 和符号 MAP 合并发送。同时，在实际发送 MAP 中，通常还会指定一些其他选项，如清屏、释放存储空间、指定光标位置等。以下为通常对整个 MAP 进行发送的代码。

```
EXEC   CICS  SEND  MAP('TESTMA')
                   MAPSET('MAPSETA')
                   CURSOR(720)
                   ERASE
                   FREEKB
                   END-EXEC
```

对于 MAP 的接收，主要是用于接收用户在 MAP 中输入的数据。这些数据实际上存放在符号 MAP 的变量中。以下为接收 MAP 的代码，其中涉及到对接收时的异常捕获。

```
EXEC  CICS   RECEIVE  MAP('TESTMA')
                    MAPSET('MAPSETA')
                    RESP(RCVERR)
                    END-EXEC
```

下面程序综合应用了 MAP 的发送和接收。当用户在 MAP 中输入完成数据时，该数据将高亮显示，同时输出相应提示信息。需要注意的是，此处使用了伪会话，但作为一个单独的程序，并没利用到参数传递的功能。假设该程序对应的交易为“TRMP”，则程序代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  MAPPGM.
*
 ENVIRONMENT  DIVISION.
*
```

```
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01   RCVERR       PIC S9(8)   COMP.
 01   COM-FLDS.
     05   T-CODE    PIC  X(10).
     05   T-NAME      PIC  X(20).
 COPY  DFHAID.
 COPY  DFHBMSCA.
 COPY  MAPSETA.
 LINKAGE  SECTION.
 01  DFHCOMMAREA.
    05  C-CODE  PIC  X(10).
    05  C-NAME    PIC  X(20).
*
 PROCEDURE  DIVISION.
      IF  EIBCALEN  =  0
         EXEC  CICS  SEND  MAP('TESTMA')
                           MAPSET('MAPSETA')
                           MAPONLY
                           ERASE
                           FREEKB
                           END-EXEC
         EXEC  CICS   RETURN  TRANSID('TRMP')
                              COMMAREA(COM-FLDS)
                              LENGTH(30)
                              END-EXEC.
*
      IF  EIBAID  =  DFHCLEAR
         MOVE  LOW-VALUES  TO  TESTMAO
         MOVE  'CLEAR KEY PRESSED: SESSION ENDED'  TO  MSGO
         EXEC  CICS  SEND   MAP('TESTMA')
                            MAPSET('MAPSETA')
                            DATAONLY
                            FREEKB
                            END-EXEC
         EXEC  CICS  RETURN  END-EXEC.
*
      IF  EIBAID  =  DFHENTER
          EXEC  CICS   RECEIVE  MAP('TESTMA')
                                MAPSET('MAPSETA')
                                RESP(RCVERR)
                                END-EXEC
          IF  RCVERR  =  DFHRESP(MAPFAIL)
                PERFORM  100-ERROR-ROUTINE.
          MOVE  CODEI  TO  CODEO
          MOVE  NAMEI   TO  NAMEO
          MOVE  DFHBMBRY  TO  CODEA
          MOVE  DFHBMBRY  TO  NAMEA
          MOVE  'SESSION  COMPLETED!'  TO  MSGO
          EXEC  CICS  SEND   MAP('TESTMA')
                             MAPSET('MAPSETA')
                             DATAONLY
                             FREEKB
                             END-EXEC
          EXEC  CICS  RETURN  END-EXEC.
```

```
            GOBACK.
*
 100-ERROR-ROUTINE.
          MOVE  'PLS  ENTER  DATA'  TO  MSGO
          EXEC  CICS  SEND  MAP('TESTMA')
                            MAPSET('MAPSETA')
                            DATAONLY
                            FREEKB
                            END-EXEC
          EXEC  CICS   RETURN  TRANSID('TRMP')
                               COMMAREA(COM-FLDS)
                               LENGTH(30)
                               END-EXEC.
```

16.7 CICS 对于文件的操作

在 CICS 中所处理的数据除用户输入的以外，绝大多数来自于外部文件。CICS 所处理的文件主要为 VSAM 文件。下面分别对文件的读取和文件的写入进行讲解。

16.7.1 读取文件

CICS 所支持的 VSAM 文件主要有 KSDS、ESDS、RRDS。此外，文件在 CICS 中由 FCT（File Control Table）所统一管理。因此，在相应的 COBOL 程序中不用手工定义和打开文件。假设某一文件的文件名为“TESTF”，则在程序中通过 CICS 读取该文件的代码通常如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  FREADPG.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 01  FILE-AREA.
     05  RECL    PIC  S9(4)  COMP.
     05  RECKEY   PIC  X(6).
 01  REC-FIELD.
     05   F-CODE    PIC X(10)  VALUE  SPACES.
     05   F-SBA     PIC X(3)   VALUE  SPACES.
     05   F-NAME    PIC X(20)  VALUE  SPACES.
     05   F-NOTUSE  PIC X(47)   VALUE  SPACES.
 ......
*
 PROCEDURE  DIVISION.
      ......
      MOVE  80  TO  RECL.
      MOVE  value   TO  RECKEY.
      EXEC  CICS  READ  FILE('TESTF')
                        INTO(REC-FIELD)
                        RIDFLD(RECKEY)
                        LENGTH(RECL)
                        END-EXEC.
      ......
```

以上代码中，RECL 变量保存文件中逻辑记录的长度，RECKEY 则保存所要读取记录的关键字。对于 KSDS，RECKEY 为该记录的 KEY；对于 ESDS，RECKEY 为该记录的 RBA；对于 RRDS，RECKEY 为该记录的 RRN。

此外，CICS 还可对文件进行连续读取，相对于对文件进行浏览。对文件进行浏览的部分程序代码如下。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  FBROWSEPG.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 ......
*
 PROCEDURE  DIVISION.
      ......
      EXEC  CICS  STARTBR  FILE('TESTF')
                           RIDFLD(RECKEY)
                           RESP(ERR-CODE)
                           END-EXEC.
      IF  ERR-CODE  EQUAL  DFHRESP(NORMAL)
           PERFORM  UNTIL  ERR-CODE  EQUAL  DFHRESP(ENDFILE)
              EXEC  CICS  READNEXT  FILE('TESTF')
                                    INTO(REC-FIELD)
                                    LENGTH(RECL)
                                    RIDFLD(RECKEY)
                                    RESP(ERR-CODE)
                                    END-EXEC
              ......
           END-PERFORM
           EXEC  CICS  ENDBR  FILE('TESTF')  END-EXEC
       ......
```

16.7.2 写入文件

与使用 READ 命令读取文件相对应，可以使用 WRITE 命令将数据写入文件。不过通常是使用 REWRITE 命令对文件中的数据进行写入更新的。在使用 REWRITE 命令之前，需要通过 READ UPDATE 命令指定更新数据所在的位置。同时，记录中的关键字是不可被更改的。以下代码反映了 REWRITE 命令的大致用法。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  FWRITEPG.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 ......
*
 PROCEDURE  DIVISION.
      ......
```

```
      EXEC  CICS  READ  FILE('TESTF')
                        UPDATE              /*此处需要包含 UPDATE 选项*/
                        RIDFLD(RECKEY)
                        INTO(REC-FIELD)
                        LENGTH(RECL)
                        RESP(ERR-CODE)
                        END-EXEC.
      ......
      EXEC  CICS  REWRITE  FILE('TESTF')
                           FROM(REC-FIELD)
                           LENGTH(RECL)
                           RESP(ERR-CODE)
                           END-EXEC.
      ......
```

此外，通过 COBOL 程序在 CICS 中也可删除 KSDS 和 RRDS 文件中的记录。删除记录实际上也相当于是对文件数据的一种更新操作。以下代码反映了在程序中通过 DELETE 命令，删除文件中指定记录的大致方式。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  FDELPG.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
 ......
*
 PROCEDURE  DIVISION.
      ......
      EXEC  CICS  DELETE  FILE('TESTF')
                          RIDFLD(RECKEY)
                          LENGTH(RECL)
                          RESP(ERR-CODE)
                          END-EXEC.
      ......
```

16.8 CICS 中的队列

CICS 中的队列相当于一块临时缓冲区，用于以队列的形式存放临时数据。在 CICS 的队列中提供两种数据组织机制。这两种机制分别为瞬时数据（Transient Data）和临时存储（Temporary Storage）。下面只重点介绍临时存储。

将数据写入临时存储队列中时，是通过 WRITEQ 命令实现的。例如，以下代码将把文件“TESTF”中的数据依次写入到临时存储队列之中。

```
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  WTSPGM.
*
 ENVIRONMENT  DIVISION.
*
 DATA  DIVISION.
 WORKING-STORAGE  SECTION.
```

```
 ......
*
 PROCEDURE  DIVISION.
      ......
       EXEC  CICS  STARTBR   FILE('TESTF')
                             RIDFLD(RECKEY)
                             RESP(RCVERR)
                             END-EXEC
       IF RCVERR = DFHRESP(NORMAL)
              STRING EIBTRMID EIBTRNID
                 DELIMITED BY SIZE INTO TS-ID
              MOVE 1 TO TS-ITEMNO
              PERFORM UNTIL RCVERR = DFHRESP(ENDFILE)
                EXEC  CICS  READNEXT   FILE('TESTF')
                                       INTO(REC-FIELD)
                                       RIDFLD(RECKEY)
                                       LENGTH(RECL)
                                       RESP(RCVERR)
                                       END-EXEC
                IF RCVERR = DFHRESP(NORMAL)
                      EXEC CICS WRITEQ TS QUEUE(TS-ID)
                                          FROM(REC-FIELD)
                                          LENGTH(RECL)
                                          ITEM(TS-ITEMNO)
                                          RESP(TS-RCVERR)
                                          END-EXEC
                      ADD 1 TO TS-ITEMNO
                END-IF
              END-PERFORM
              EXEC  CICS  ENDBR  FILE('TESTF')  END-EXEC
       ......
```

需要注意的是，对于临时存储队列，对其进行操作时通常会涉及到一个称作 ITEM 的选项。ITEM 按顺序记录了队列中每一数据记录的编号，可以用于对记录进行直接访问。

对临时存储队列同样也可进行读取和删除。其中读取队列中指定数据记录（通过 ITEM 编号指定）的 CICS 命令如下。

```
EXEC  CICS  READQ  TS  QUEUE(TS-ID)
                       INTO(TS-REC)
                       LENGTH(TS-RECL)
                       ITEM(TS-ITEMNO)
                       RESP(TS-ERR-CODE)
                       END-EXEC.
```

对整个临时存储队列进行删除的 CICS 命令如下。

```
EXEC   CICS  DELETEQ  TS  QUEUE(TS-ID)
                         END-EXEC.
```

16.9 本章回顾

本章主要讲解了 CICS 的概念及应用。CICS 通常以 COBOL 作为宿主语言，提供了界面功能和交互能力，因此在实际开发中应用十分广泛。

本章首先讲解了 CICS 的基本概念及其编译处理过程。学习这部分内容，需要明确联机处理程序和批处理程序的概念及其区别，理解 CICS 中交易与任务的基本概念，掌握 CICS 中的日常操作，理解基于 CICS 的 COBOL 程序的编译流程，掌握实际开发 CICS 应用程序的步骤，其中包括对于资源的定义和安装，以及对于程序的更新和调试。

本章之后介绍了 CICS 在 COBOL 中的基本应用。其中主要包括 CICS 的输入输出以及各种系统信息的获取。学习这部分内容，需要掌握基于 CICS 的 COBOL 程序基本结构；掌握如何使用 CICS 进行输入输出处理，其中包括输入过程中的异常处理和输出过程中的光标定位；掌握获取 CICS 系统基本信息的方式，其中包括获取终端编号信息、系统时间信息、系统日期信息。

本章接下来重点介绍了 CICS 中的一些特殊应用，其中包括伪会话程序、程序调用、MAP。其中伪会话程序和 MAP 是 CICS 中的重点内容，一定要牢固掌握。学习这部分内容，需要理解伪会话程序与会话程序的概念及区别，掌握如何编写两种不同类型的伪会话程序，理解程序调用中 LINK 命令和 XCTL 命令的概念及区别，理解物理 MAP 和符号 MAP 的概念，掌握 MAP 的创建步骤，掌握在 COBOL 程序中发送 MAP 和接受 MAP 的基本应用。

本章最后介绍了 CICS 中关于文件和队列的操作。学习这部分内容，需要了解 CICS 通常处理的文件类型以及 CICS 中队列的基本概念，了解在 CICS 中对于文件以及队列的读写操作。

总之，CICS 对于使用 COBOL 进行实际的应用软件开发而言十分重要。在本书最后的 COBOL 实战篇中，将会涉及到对于 CICS 更为广泛和深入的应用。

第 17 章 大型机汇编语言扩展

COBOL 语言主要是应用于大型机，而大型机上的汇编语言则更加贴近其内部结构。因此，学习汇编语言，可以更加深入地理解 COBOL 语言。同时，汇编语言能够从更加底层的角度，反映大型机程序的工作原理。因此，若要从事大型机上更为高级的应用开发或系统管理工作，也是需要掌握汇编语言的。

17.1 基本概念

汇编语言是针对不同的机型而言的。例如，PC 上的 80x86 汇编语言和大型机上的 S/390 汇编语言就是两个不同的概念。本章只讲解大型机上的汇编语言。关于大型机的汇编语言，主要需要理解以下几个方面的概念，下面分别进行讲解。

17.1.1 主存组织

计算机中最小的存储单位是一个位（bit）。位是用 0 和 1 这两个二进制数表示的，其中 8 个位形成一个字节（Byte）。存储空间通常是以字节为基本单位的，由字节组成字。字所占的存储空间大小称为字长。字节在任何机器上都是由 8 位组成的，但字在不同的机器上则有着不同的字长。关于字及其相关概念，在大型机的主存中有以下 3 种组织形式。

- 半字（HALF WORD）：由 2 个字节组成，占 16 位存储空间。
- 全字（FULL WORD）：由 4 个字节组成，占 32 位存储空间。其中全字也就是所谓的字。
- 双字（DOUBLE WORD）：由 8 个字节组成，占 64 位存储空间。

此外，在表示大型机的主存地址空间时，除了二进制数外，通常还会应用到十六进制数。十六进制数中的数字 0～9 分别表示十进制数中的 0～9。而十六进制数中的数字 A～F 则分别表示十进制数中的 10～15。例如，以下为一组十进制数。

```
0
2
10
```

```
45
100
```

则以上数列对应的二进制数如下。

```
0
10
1010
101101
1100100
```

以上数列对应的十六进制数如下。

```
0
2
A
2D
64
```

实际上，引入十六进制数的概念也是为了更方便地表示二进制数。可以看到，2^4 正好等于 16，因此一个十六进制数正好可以表示 4 个二进制数。由于一个字节是由 8 个位组成的，而每个位又由一个二进制数表示。因此，一个字节也可以通过两个十六进制数来表示。

大型机主存中的普通地址空间（Common Area）和私有地址空间（Private Area）一共是 16MB。由于 1MB=2^{20}，因此 16MB 大小的地址空间可以用 24 个 bits 表示，同时也可以使用 6 个十六进制数表示。如图 17.1 所示，反映了使用十六进制数表示的一段主存空间。

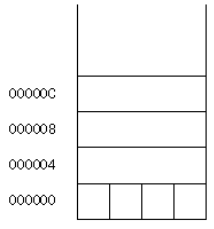

图 17.1　主存空间示例图

上图中每一行为 4 个字节，表示一个全字空间。这 4 个字节只在最下面一行划分了出来，以上各行相同。根据该示例图，需要理解以下两个重要概念。

- 半字空间的边界为偶数地址。
- 全字空间的起始地址应该能被 4 整除。

17.1.2　数码表示

同其他汇编语言一样，在大型机的汇编语言中通常也涉及到以下 3 个关于数码表示的概念。这 3 个概念分别如下所示。

- 原码：即该数据自身。例如，以下分别为不同进制中示例数据与原码的关系。

```
十进制数中：10 的原码为 10
二进制数中：1010 的原码为 1010
十六进制数中：A 的原码为 A
```

- 反码：通过将数据在二进制的表示中各位取反得到。以下为不同进制中示例数据与反码的关系。

```
十进制数中：10 的反码为 5
二进制数中：1010 的反码为 0101
十六进制数中：A 的反码为 5
```

- 补码：通过将数据在二进制的表示中各位取反后加 1 得到。以下为各示例数据与反码的关系。

```
十进制数中：10 的补码为 6
二进制数中：1010 的补码为 0110
十六进制数中：A 的补码为 6
```

在大型机中，以上所说的反码通常也叫做 1 的补码，而以上所说的补码则通常叫做 2 的补码。实际上，所谓 n 的补码，也就是在 n－1 的补码基础上再加 1 得到的。

对于大型机中的整数数据，是以 32 位全字空间作为存储单元的。其中最左边的 1 位表示符号位，正号用 0 表示，负号用 1 表示。以下为大型机上两组不同整数的表示方式。

- $0 \sim 2^{31}-1$ 之间的整数：直接使用该整数的原码表示。
- $-1 \sim -2^{31}+1$ 之间的整数：使用该整数绝对值编码形式的 2 的补码表示。

例如，对于整数−2，在大型机中的表示方法如下。

```
11111111 11111111 11111111 11111110
```

此外，关于大型机中的数码，还有以下几点需要注意。

- 负数的 2 的补码就是该数绝对值的原码，该绝对值的原码的 2 的补码为负数本身。
- 系统内部进行减法运算的过程是先得到减数的 2 的补码，再与被减数相加。

17.1.3 寄存器与程序状态字

在汇编语言中，寄存器可以看作是一块单独的存储空间。该存储空间通常用于存储汇编语言中的操作数。在 S/390 大型机上，有以下几种类型的寄存器。

- General Purpose Registers（GPRs）：通用寄存器。用于在程序中作为存储地址、工作空间或者计数器。
- Floating Point Registers（FPRs）：浮点寄存器。用于进行浮点运算。
- Control Registers：控制寄存器。用于操作系统。
- Access Registers：通道（访问）寄存器。用于操作系统。
- Vector Registers：向量寄存器。用于向量运算。

其中通用寄存器（GPRs）在汇编语言程序中经常要用到。大型机汇编语言中共有 16 个通用寄存器，分别用 R0～R15 表示。每个通用存储器的大小为一个全字空间。在汇编语言程序中，使用 0～15 这 16 个数字分别表示以上 16 个通用寄存器。如果在源程序最后通过 EQU 特别指定，也可以使用字符 R0～R15 来表示以上寄存器。使用 EQU 指定的方式如下。

```
R0   EQU   0
R1   EQU   1
R2   EQU   2
 ......
R15  EQU  15
```

程序状态字（Program Status Word）简称为 PSW。PSW 位于硬件上的一块特定区域，用于保存中央处理器（CPU）当前的状态。实际上，PSW 也是一个寄存器，并且常用于程序开发之中。扩展模式下的 PSW 共有 64 位，首位地址从 0 开始。其中以下几位表示的内容经常会用到。

- 31 位：表示程序是否产生异常。
- 34～35 位：表示条件码（Condition Code）当前的值。
- 40～63 位：表示下一条指令的地址。

关于条件码，也可以将其简称为 CC。CC 常用于程序流程控制中。通常条件运算的结果和 CC 的对应关系如下。

- 运算结果等于 0 时，CC 为 0，对应的二进制数为 00。
- 运算结果小于 0 时，CC 为 1，对应的二进制数为 01。
- 运算结果大于 0 时，CC 为 2，对应的二进制数为 10。
- 运算结果溢出时，CC 为 3，对应的二进制数为 11。

程序在执行过程中必然要用到 PSW。实际上，程序的每步执行都是根据 PSW 中的内容而来的。关于程序的具体执行步骤如下。

（1）程序第一条指令的绝对地址被装入 PSW，实现程序执行的初始化。其中具体取多少字节装入是由指令的类型决定的。

（2）机器从 PSW 指向的存储地址中检查出程序指令以执行。

（3）机器更新 PSW 的内容，使其指向下一条指令地址。并且，如果上条指令是转移分支，则 PSW 中的相应内容将指向跳转地址。

17.1.4 操作数的主存地址表示方式

与 COBOL 不同，汇编语言程序并不是按照语句执行的，而是按照指令来执行的。COBOL 中的一条语句往往对应汇编语言中的多条指令。例如，以下为一条在 COBOL 中实现算术加运算的语句。

```
ADD  A TO B.
```

该条 COBOL 语句对应的汇编指令如下。

```
PACK  X,A
PACK  Y,B
AP    X,Y
UNPK  B,X
```

汇编中的一条指令通常是由操作符和操作数共同组成的。其中操作符一般在指令的左边，操作数一般在指令的右边。例如，对于以上第一条指令，PACK 就为操作符，而 X 和 A 则分别为两个操作数。

以上指令中的操作数是直接用变量表示的。实际上，操作数也可直接通过其地址表示，并且变量最终也会转换为操作数的地址形式。操作数的地址通常有以下两种表示方式。为观察方便，此处不妨假设已通过 EQU 将 R0～R15 指定为 0～15，直接通过 R0～R15 表示寄存器。

1．基址+偏移地址表示方式

以上表示方式的格式如下。

```
D(B)
```

其中 B 代表基址，D 代表相对于该基址的偏移地址。基址也就是一个基准地址，是通过基址寄存器表示的。偏移地址表示相对基址的偏移量，是通过十进制数表示的。该十进制数的范围为 $0～2^{17}-1$。例如，以下为一个使用基址+偏移地址表示的操作数。

```
20(R13)
```

以上是将通用寄存器 R13 作为基址寄存器，并将十进制数 20 作为相对于基址的偏移量。如果 R13 中的内容如下。

```
(R13) = 010000A0
```

则以上操作数的地址是将偏移量 20 加上 R13 中右边 24 位表示的基址得到的。因此，该操作数的地址如下。

```
20(R13) = 0000B4
```

2．基址+偏移地址+索引地址表示方式

以上表示方式的格式如下。

```
D(X, B)
```

其中 X 为索引地址，通过索引寄存器表示。基址和偏移地址的要求和上一种表示方法相同。例如，以下为使用基址+偏移地址+索引地址表示的操作数。

```
10(R1, R15)
```

以上操作数是将通用寄存器 R15 作为基址寄存器，而将 R1 作为索引寄存器，10 作为偏移量。如果 R1 和 R15 中的内容分别如下。

```
(R1) = 00000100
(R15) = 10A051A1
```

以上操作数的地址如下。

```
10(R1, R15) = A052AB
```

此外需要注意的是，在汇编语言程序中操作数地址表示中的寄存器是可以省略不写的。例如，以下这几种表示方式都是正确的。

```
25
25( , R15)
25(R15, )
```

以上操作数地址中，第一个使用的是 D（B）表示方式，后两个使用的则是 D（X, B）表示方式。当寄存器省略不写时，可以认为省略处的寄存器内容为 0。

另外，在 16 个通用寄存器中，R15 通常作为基址寄存器。而 R0 既可作为基址寄存器，也可以作为索引寄存器。并且，当在操作数地址中使用 R0 时，将直接用 0 进行计算，而不理会 R0 寄存器中的实际内容。以下例子充分说明了这一点。

```
(R0) = 00100101
26(R0) = 1A
```

由此可见，虽然此时 R0 中的内容为 00100101，但 26（R0）所表示的地址并不理会 R0 中的内容。此时是将 R0 看作 0，直接通过计算十进制数 26 加上 0 后所对应的十六进制数得到的最终地址。

最后，关于指令中操作数地址的表示方法，还有以下两点需要注意。

- 地址都是为正值的，即使其首位为 1。
- 地址的表示不能超出相应的范围。

17.1.5 程序基本结构

在汇编语言源程序中，通常是以 CSECT 标志符表示程序的开始。程序名写在 CSECT 的左边，代码写在 CSECT 的下面。当涉及到子程序调用时，子程序使用 DSECT 替代 CSECT。当程序编写完成后，使用 END 加上程序名表示结束。

在汇编语言程序主体代码中，通常分为两块内容。其中一块为程序逻辑部分，另一块为数据定义部分。同 COBOL 类似，当程序代码中某行首列为星号时，表示该行是注释行。例如，下面为一段汇编语言程序的基本结构。

```
TEST    CSECT
        ……                  /*程序模块化代码，只在 z/OS 环境下使用*/
*****  BEGIN   LOGIC  ***** /*此行为注释，下同*/
        ……                  /*此处为程序逻辑部分代码*/
*****   END    LOGIC  *****
        ……                  /*程序模块化代码，只在 z/OS 环境下使用*/
*****   DATA   AREA  *****
        ……                  /*此处为数据定义部分代码，若在 z/OS 环境下，还包括 DCB 参数的定义*/
        END  TEST
```

大型机汇编语言通常可以在两种环境下运行。一种是在 ASSIST 软件下运行，另一种是直接在 z/OS 大型机操作系统上运行。例如，以下是一段运行于 ASSIST 软件下的程序代码。

```
TRANS1     CSECT
*****  BEGIN   LOGIC  *****
         USING   TRANS1, 15
         XREAD  CARD1, 80
         BC      B '0100', EXIT
         XPRNT  FIELD1, 81
         XREAD  CARD2, 80
LOOP    BC      B '0100', EXIT
        XPRNT   FIELD2, 80
        BC      B '1111', LOOP
EXIT     BCR     B '1111', 14
*****   END    LOGIC  *****
*****   DATA    AREA  *****
FIELD1   DC      C'1'
CARD1    DS      CL80
FIELD2   DC      C'0'
CARD2    DS      CL80
         END    TRANS1
```

以上程序实现的功能是依此读取 CARD 指向的每条记录，并将其显示输出，直至全部读完。在 z/OS 系统上具有与之类似功能的程序代码（包括相关 JCL 语句）如下。

```
……      /*相关 JCL 语句，根据不同的系统配置而有所不同*/
TRANS2  CSECT
          STM    R14, R12, 12(R13)
          BALR   R12, R0
          USING  *, R12
          LR      R11, R13
          LA      R13, SAVEAREA
          ST      R11, 4(, R13)
          ST      R13, 8(, R11)
```

```
*****  BEGIN  LOGIC  *****
         OPEN  (INFILE, (INPUT), OUTFILE, (OUTPUT))
 LOOP    EQU   *
          GET   INFILE, INAREA
          MVC   OUTREC, INAREA
          PUT   OUTFILE, OUTAREA
          B     LOOP
 ENDDATA  EQU *
          CLOSE (INFILE , , OUTFILE)
*****   END   LOGIC  *****
 RETURN  EQU  *
          L     R13, SAVEAREA+4
          RETURN (14,12), RC=0
*****   DATA   AREA  *****
 INAREA    DS    CL80
 OUTAREA  DS    0CL133
 OUTASA    DC   C'0'
 OUTREC    DS   CL80
          DC   CL52' '
INFILE    DCB   DDNAME = SYSIN, RECFM = FB, LRECL = 80, DSORG = PS,
                MACRF = GM, EODAD = ENDDATA
OUTFILE   DCB   DDNAME = SYSPRINT, RECFM = FBA, LRECL = 133, DSORG = PS,
                MACRF = PM, BLKSIZE = 6650
SAVEAREA   DC    18F'0'
           PRINT GEN
           YREGS
           LTORG
           END  TRANS2
//G.SYSIN    DD  DSN = ADCDA.ASM.DATA, DISP = SHR
//G.SYSPRINT  DD  SYSOUT = *
```

以上程序实现的功能是读取 z/OS 系统上数据集 ADCDA.ASM.DATA 中的每条记录并将其输出。对于以上两段程序，只需了解其大体框架结构，对汇编语言程序建立一个感性认识。至于其中的具体内容，此处不必深究。

17.2 指令类型与机器码

COBOL 语言属于高级语言，而汇编语言属于低级语言。二者编写的程序分别需要经过编译或汇编转换为机器语言后，计算机才能理解和执行。汇编语言更加贴近于机器语言，因此其执行效率也更高。汇编语言的指令根据其转换后的机器码类型可以分为不同的格式，下面分别进行介绍。

17.2.1 RR 指令类型及其机器码

当某一指令中的两个操作数都为寄存器时，该指令属于 RR 类型的指令。例如，以下这几条指令都为 RR 类型的指令。

```
AR  12, 0
SR  3, 1
LR  4, 5
CR  2, 3
```

RR 指令类型的格式及其对应的机器码分别如下。

```
OP r1, r2      /*RR 指令格式*/
h0h0hr1hr2     /*RR 指令机器码*/
```

关于以上机器码中的内容，分别介绍如下。

- h_0h_0：该指令的操作符 OP，通过两位十六进制数表示。
- h_{r1}：该指令的第一个操作数 r1，为一个寄存器，通过一位十六进制数表示。
- h_{r2}：该指令的第二个操作数 r2，为一个寄存器，通过一位十六进制数表示。

由此可见，RR 指令所占存储空间为 1 个半字空间。其中操作符占 1 个字节，第一个操作数和第二个操作数各占 0.5 个字节。对于以上第一条指令，由于其操作符 AR 的机器码为 1A。因此，可以写出该条指令对应的机器码如下。

```
1AC0
```

对于该类型中的其余各条指令，只要知道了指令中操作符的机器码，就可以写出整条指令的机器码。这点对于以下各类型的指令也是适用的。

17.2.2 RX 指令类型及其机器码

当某一指令的第一个操作数为寄存器，第二个操作数为主存地址时，该指令属于 RX 类型的指令。例如，以下这几条指令都为 RX 类型的指令。

```
L  2, 10(1, 15)
ST  1, 4(8)
A  2, 14( , 14)
S  3, 10(2, )
C  2, 8(0, 13)
LA  2, 5(3, 5)
STC  3, 0(6)
```

RX 指令类型的格式及其对应的机器码分别如下。

```
OP r, D(X, B)
h0h0hrhX hBhDhDhD
```

关于以上机器码中的内容，分别介绍如下。

- h_0h_0：该指令的操作符 OP，通过两位十六进制数表示。
- h_r：该指令的第一个操作数 r，为一个寄存器，通过一位十六进制数表示。
- h_X：该指令第二个操作数 D(X, B)中的索引寄存器 X，通过一位十六进制数表示。
- h_B：该指令第二个操作数 D(X, B)中的基址寄存器 B，通过一位十六进制数表示。
- $h_Dh_Dh_D$：该指令第二个操作数 D(X, B)中的偏移量 D，通过三位十六进制数表示。

由此可见，RX 指令所占存储空间为 2 个半字空间。其中操作符占 1 个字节，第一个操作数占 0.5 个字节，第二个操作数占 2.5 个字节。对于以上第一条指令，由于其操作符 L 的机器码为 58。因此，可以写出该条指令对应的机器码如下。

```
5821F00A
```

需要注意的是，当索引寄存器省略未写或使用 D（B）方式表示地址时，h_X 位为 0。这点对于其他类型的指令也是相同的。

17.2.3 RS 指令类型及其机器码

当某一指令的前两个操作数为寄存器，第三个操作数为主存地址时，该指令属于 RS 类型的指令。例如，以下这几条指令都为 RS 类型的指令。

```
LM 0, 15, 300(3)
STM 14, 1, 24(5)
BXLE  1, 2, 10(4)
```

RS 指令类型的格式及其对应的机器码分别如下。

```
OP r1, r2, D(B)
h0h0hr1hr2 hBhDhDhD
```

关于以上机器码中的内容，分别介绍如下。

- h_0h_0：该指令的操作符 OP，通过两位十六进制数表示。
- h_{r1}：该指令的第一个操作数 r1，为一个寄存器，通过一位十六进制数表示。
- h_{r2}：该指令的第二个操作数 r2，为一个寄存器，通过一位十六进制数表示。
- h_B：该指令第三个操作数 D(B)中的基址寄存器 B，通过一位十六进制数表示。
- $h_Dh_Dh_D$：该指令第三个操作数 D(B)中的偏移量 D，通过三位十六进制数表示。

由此可见，RS 指令所占存储空间为 2 个半字空间。其中操作符占 1 个字节，第一个操作数和第二个操作数各占 0.5 个字节，第三个操作数占 2 个字节。对于以上第一条指令，其操作符 LM 的机器码为 98。因此，可以写出该条指令对应的机器码如下。

```
980F312C
```

需要注意的是，该指令中使用主存地址表示的第三个操作数只能使用 D（B）方式。即此处不能含有索引寄存器。

17.2.4 SI 指令类型及其机器码

当某一指令的第一个操作数为主存地址，第二个操作数为直接数时，该指令属于 SI 类型的指令。其中有的地方也将直接数称做立即数。以下这条指令为 SI 类型的指令。

```
MVI  10(1), C'*'
```

RS 指令类型的格式及其对应的机器码分别如下所示。

```
OP D(B), I
h0h0hIhI hBhDhDhD
```

关于以上机器码中的内容，分别介绍如下。

- h_0h_0：该指令的操作符 OP，通过两位十六进制数表示。
- h_Ih_I：该指令的第二个操作数 I，为一个直接数，通过两位十六进制数表示。
- h_B：该指令第一个操作数 D(X, B)中的基址寄存器 B，通过一位十六进制数表示。
- $h_Dh_Dh_D$：该指令第一个操作数 D(X, B)中的偏移量 D，通过三位十六进制数表示。

由此可见，SI 指令所占存储空间为 2 个半字空间。其中操作符占 1 个字节，前两个操作数共占 1 个字节，第三个操作数占 2 个字节。对于以上第一条指令，其操作符 MVI 的机器码为 92，并且立即数“*”的机器码为 5C。因此，可写出该条指令对应的机器码如下。

```
925C100A
```

需要注意的是，该条指令中第一个操作数为主存地址，第二个操作数为直接数。然而在该指令对应的机器码中，则将第二个操作数的机器码写在前面，而将第一个操作数写在后面。

17.2.5 SS 指令类型及其机器码

当某一指令的两个操作数都为主存地址时，该指令属于 SI 类型的指令。例如，以下这几条指令都为 SS 类型的指令。

```
MVC 24(5),12(6)
SP 0(4), 12(3)
AP 12(3), 0(4)
PACK 10(1), 20(2)
UNPK 20(1), 10(2)
```

SS 指令类型的格式如下。

```
OP D1(B1), D2(B2)
```

SS 指令类型的格式实际上共分为两种情况。第一种情况是两操作数的长度相等，第二种情况是两操作数的长度不等。这两种情况对应的机器码分别如下。

```
h0h0hLhLhB1hD1hD1hD1 hB2hD2hD2hD2
h0h0hL1hL2 hB1hD1hD1hD1 hB2hD2hD2hD2
```

可以看到，SS 指令所占存储空间为 3 个半字空间。其中以上机器码中关于操作符和操作数部分的内容和其他类型指令的机器码类似。这里仅重点介绍以下两点不同的地方。

- h_Lh_L：对应于指令中两操作数长度相等的情况，表示两者共同的长度大小。该长度通过两位十六进制数表示，因此对应的十进制数数值大小为 0 ~ 255。
- $h_{L1}h_{L2}$：对应于指令中两操作数长度不等的情况。其中 h_{L1} 表示第一个操作数的长度大小，h_{L2} 表示第二个操作数的长度大小。每个操作数的长度通过一位十六进制数表示，因此对应的十进制数数值大小为 0 ~ 15。

需要注意的是，由于长度为 0 的操作数是没有意义的。因此，在机器语言中操作数长度的显示数值都要比其真实数值小 1。例如，机器语言中长度显示为 0 时，表示的真实长度是 1；长度显示为 1 时，表示的真实长度为 2，等等。

以上第一条指令属于 SS 指令类型的第一种情况，即两操作数长度是相等的。不妨设该条指令中两操作数的长度均为 100 个字节，则反映到机器码中的数值应为 99。将 99 转换为十六进制数，为 63。此外，由于操作符 MVC 的机器码为 D2，因此可以写出第一条指令的机器码如下。

```
D2635018600C
```

以上第二条指令属于 SS 指令类型的第二种情况，即两操作数长度是不等的。不妨设第一个操作数的长度为 5 个字节，第二个操作数的长度为 1 个字节。则二者在相应的机器码中将分别为 4 和 0。由于操作符 SP 的机器码为 5B，因此可以写出第二条指令的机器码如下。

```
5B404000300C
```

17.3 数据的定义

在汇编语言程序中，各数据都是在程序末尾的数据定义部分通过 DC 和 DS 进行定义的。其中 DC 用于定义程序中的常量，DS 用于定义一段存储空间。下面分别予以讲解。

17.3.1 使用 DC 定义常量

通过前面的例子可以看出，在程序的数据定义部分经常会使用 DC 定义程序中的相关常量。DC 的使用格式如下。

```
constant  DC  dtLn'value'
```

现将以上格式中的各部分内容分别介绍如下。

- constant：代表该常量的名字，可省略。
- d：代表所定义内容的重复度，省略时表示重复度为 1。
- t：代表所定义常量的类型
- Ln：代表所定义常量的长度，以字节为单位表示。省略时其长度为所定义内容的全部长度。
- value：代表所定义的内容。

由于大型机中使用的是 EBCDIC 编码制度，因此任何数据反映到内存中都是 EBCDIC 码。为方便说明问题，如表 17.1 所示，为几个常用字符的 EBCDIC 码。

表 17.1　　常用字符 EBCDIC 码

字　符	EBCDIC 码	字　符	EBCDIC 码
A	C1	+	41
B	C2	−	60
C	C3	.（小数点）	4B
‘ ’（空格）	40	0～9	F0～F9
*	5C	−1～−9	D1～D9
,	6B		

此外，关于使用 DC 定义的数据类型，常用的共有以下几种。

- C：定义字符。
- X：定义十六进制数。
- B：定义二进制数。
- Z：定义分区十进制数（分区十进制数用于输入输出，将在后面章节中详细讲解。）。
- P：定义打包十进制数（打包十进制数用于算数运算，将在后面章节中详细讲解。）。
- H：定义半字定点数。
- F：定义全字定点数。

下面分别给出使用 DC 定义常量数据的部分示例。通过示例，可以更好地理解各常量在不同形式下的定义所对应的实际内容。示例如下。

```
CFLD1  DC   C'ABC'              /*转换为对应的 EBCDIC 码，实际内容为 C1C2C3*/
CFLD2  DC   3C'*'               /*重复度为 3，实际内容为 5C5C5C*/
```

```
CFLD3  DC   CL3'*'              /*不足部分在右填充空格，实际内容为 5C4040*/
CFLD4  DC   CL2'ABC'            /*溢出部分从右截断，实际内容为 C1C2*/
CFLD5  DC   2CL2'ABC'           /*溢出字符从右截断，同时重复度为 2，实际内容为 C1C2C1C2*/
*
XFLD1  DC   X'3A4F'             /*直接通过该十六进制数表示，实际内容为 3A4F*/
XFLD2  DC   XL3'3C'             /*不足部分在左填充 0，实际内容为 00003C*/
XFLD3  DC   2XL1'3C4D'          /*溢出部分从左截断，同时重复度为 2，实际内容为 4D4D*/
XFLD4  DC   X'12B'              /*长度需为完整字节数，此处应为 2 个字节，实际内容为 012B*/
XFLD5  DC   X'3C,ABC'           /*两个数值之间用逗号隔开，实际内容为 3C0ABC*/
*
BFLD1  DC   B'11010'            /*将二进制数转换为十六进制数表示，实际内容为 1A*/
BFLD3  DC   BL1'10101010101'    /*溢出部分从左截断，实际内容为 55*/
BFLD3  DC   2BL2'11010'         /*不足部分在左填充 0，同时重复度为 2，实际内容为 001A001A*/
*
ZFLD1  DC   Z'123'              /*转换为对应的 EBCDIC 码，实际内容为 F1F2F3*/
ZFLD2  DC   ZL5'123'            /*不足部分在左填充 F0，实际内容为 F0F0F1F2F3*/
ZFLD3  DC   3ZL2'1,-2,3'        /*溢出部分从左截断，同时重复度为 3，实际内容为 D2F3D2F3D2F3*/
*
PFLD1  DC   P'123'              /*正值在其后加 C，实际内容为 123C*/
PFLD2  DC   PL1'-123'           /*负值在其后加 D，溢出部分从左截断，实际内容为 3C*/
PFLD3  DC   PL3'123'            /*不足部分在左填充 0，实际内容为 00123C*/
PFLD4  DC   P'1,-2,3'           /*各数分别表示，实际内容为 1C2D3C*/
PFLD5  DC   P'12.25'            /*忽略小数点，实际内容为 01225C*/
*
HFLD1  DC   H'10'               /*将十进制数转换为十六进制数表示，半字占 2 个字节，实际内容为 000A*/
HFLD2  DC   H'-5'               /*负数用其绝对值原码的 2 的补码表示，实际内容为 FFFB*/
HFLD3  DC   H'10,-5'            /*各数分别表示，实际内容为 000AFFFB*/
*
FFLD1  DC   F'10'               /*全字占 4 个字节，实际内容为 0000000A*/
FFLD2  DC   F'-5'               /*负数用其绝对值原码的 2 的补码表示，实际内容为 FFFFFFFB*/
FFLD3  DC   2F'10'              /*重复度为 2，实际内容为 0000000A0000000A*/
```

需要注意的是，关于 H 和 F 类型的定点数常量，其类型中已暗含了存储空间的大小（半字或全字）。因此，对于定点数常量通常不提倡使用 Ln 定义其数据长度。同时，注意到打包十进制数中的小数点也是不占存储空间的。这点同 COBOL 语言中使用 V 表示的虚拟小数点是类似的。

此外，在程序代码中还可使用直接数（立即数）表示指令的操作数。直接数使用等号作为前缀标志，其后包括数据类型、长度及内容。例如，以下为几条含有直接数的指令。

```
MVC   FLD1, =C'ABC'
D     FLD2, =F'10'
AP    FLD3, =P'5'
MVC   FLD4, =CL132'*'
```

17.3.2 使用 DS 定义存储空间

使用 DS 定义存储空间与使用 DC 定义常量是类似的。不过，由于存储空间内的内容可以不断被改写，因此 DS 的定义中通常不含 value 项。使用 DS 定义存储空间的格式如下。

```
area  DC  dtLn
```

其中 area 为该段存储空间的名字。其余部分与 DC 定义格式类似，此处不再赘述。需要注意的是，在 DS 定义格式中，重复度 d 是可以为 0 的。当其重复度为 0 时，表示重复定义

相同的重复空间，类似于 COBOL 中的多层数据定义。例如，下面为一段使用重复定义方式定义的存储空间。

```
RECORD      DS   0CL36
FLDA        DS   CL12
FLDB        DS   CL12
FLDC        DS   0CL12
FLDC1       DS   CL4
FLDC2       DS   CL4
FLDC3       DS   CL4
```

以上所定义的存储空间如图 17.2 所示。

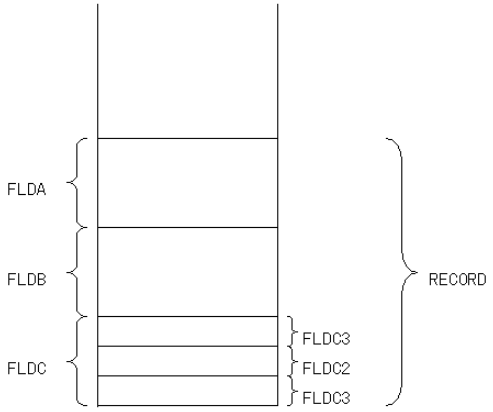

图 17.2　重复定义的存储空间图

DS 中定义的存储空间数据类型除上一小节在 DC 中介绍的各数据类型外，还包含有双字空间类型。各存储空间的数据类型及其长度，如表 17.2 所示。

表 17.2　　各存储空间类型及长度表

DS 中定义的符号	数 据 类 型	默认长度（单位：字节）
C	字符	1
X	十进制数	1
B	二进制数	1
Z	分区十进制数	1
P	打包十进制数	1
H	半字定点数	2
F	全字定点数	4
D	双字定点数	8

此外需要注意的是，在 DS 定义的存储空间中，重复度和长度是不能混为一谈的。例如，以下 3 段存储空间的大小都为 100 字节，定义方式分别如下。

```
A   DS  CL100      /*重复度为 1，长度为 100*/
B   DS  10CL10     /*重复度为 10，长度为 10*/
C   DS  100C       /*重复度为 100，长度为 1*/
```

以下 3 条指令将从 DATA1 中传递不同的数据量给各存储空间，分别如下。

```
MVC  A, DATA1     /*传递数据量为 100 字节*/
MVC  B, DATA1     /*传递数据量为 10 字节*/
MVC  C, DATA1     /*传递数据量为 1 字节*/
```

如果仍然要传递 100 个字节的数据给存储空间 C，可以通过以下指令实现。

```
MVC  C(100), DATA1
```

最后，无论是使用 DC 定义的常量，还是使用 DS 定义的存储空间，其名字都可作为实际操作数。如此，便可替代使用 D（B）或 D（X, B）方式表示的操作数。例如以上这 4 条指令中的操作数便直接使用的相关名字，而不是地址表示方式。

17.4 数据的传递

数据的传递是汇编语言程序中最常用到的数据操作。数据的传递既可在寄存器之间，也可在主存之间，还可在寄存器与主存之间。需要注意的是，此处所说的传递类似于复制，原空间中的数据不变。下面分别就几条最常用的相关指令进行讲解。

1. L、ST 和 LR 指令

L 指令将主存中的数据传递给寄存器，传递的数据量为一个全字空间，传递方向为载入方向（Load）。也就是说，LR 指令是将第二个操作数中的内容传递给第一个操作数。该指令属于 RX 类型的指令。例如，以下为一段使用 L 指令的代码。

```
      L  R2, FLD1
      ……
FLD1  DC  X'1A2B3C4D'
```

则该指令执行后，无论寄存器 R2 之前的内容为多少，此时 R2 的内容均为 1A2B3C4D。此外，当需要传递半字空间的数据量时，可使用 LH 指令。LH 指令和 L 指令仅是传递的数据量不同，其余基本类似，故此处不再赘述。

ST 指令与 L 指令相反，传递的方向为存入方向（Store）。也就是说，ST 指令是将第一个操作数中的内容传递给第二个操作数。ST 指令传递的数据量也为一个全字空间，属于 RX 类型的指令。例如，以下为一段使用 ST 指令的代码。

```
      ST  R2, FLD2
      ……
FLD2  DS  F
```

若此时寄存器 R2 中的内容为 1A2B3C4D，则 FLD2 表示的一段存储空间中的内容也将与之相同。此外，当需要传递半字空间的数据量时，也可使用 STH 指令。

LR 指令传递的数据量为一个全字空间，方向为载入方向。不过 LR 指令属于 RR 类型的指令，其数据的传递是在寄存器之间的。例如，以下为一条 LR 指令。

```
LR  R3, R2
```

若寄存器 R2 中的内容为 1A2B3C4D，则该指令执行后，R3 中的内容也将为 1A2B3C4D。

总之，L 指令是将主存中的数据载入到寄存器，ST 指令是将寄存器中的数据存入到主存，LR 指令则实现了寄存器间数据的传递。

2．LM 和 STM 指令

LM 指令与 L 指令类似，不过该指令实现的是多个寄存器的载入，属于 RS 类型的指令。例如，以下为一段使用 LM 指令的代码。

```
          LM  R2, R4, FLDS1
           ……
FLDS1     DC  F'1'
          DC  F'10'
          DC  F'100'
```

该指令是将以 FLDS 命名的起始地址后的 3 个全字空间中的数据依次载入寄存器 R2～R4。该指令执行后，以上 3 个寄存器中的内容分别如下所示。

```
(R2) = 00000001
(R3) = 0000000A
(R4) = 00000064
```

STM 指令与 LM 指令对应，是将多个寄存器中的数据载入一片主存空间。该指令也属于 RS 类型的指令。例如，以下为一段使用 STM 指令的代码。

```
          STM  R2, R4, FLDS2
          ……
FLDS2     DS  3F
```

该指令实现的功能是将寄存器 R2、R3、R4 中的数据依次存入以 FLDS2 命名的主存空间。

最后需要注意的是，在 LM 指令和 STM 指令中，第一个寄存器的编号是可以大于第二个的。当第一个寄存器为 R4，第二个为 R2 时，表示的是寄存器组 R4～R15 以及 R0～R2。

3．MVC 指令

MVC 指令用于主存之间数据的传递，属于 SS 类型的指令。该指令将第二个操作数中的内容传递给第一个操作数，传递的数据量由第一个操作数决定。因此，该指令对应 SS 类型指令中两操作数长度相等的情况，故最多可传递 256 个字节的数据。以下为一段使用 MVC 指令的代码。

```
        MVC  FLD3, FLD4
        MVC  FLD5, CL5'ABCDE'
……
FLD3  DS  CL3
FLD4  DC  C'123'
FLD5  DS  F
```

以上第一条 MVC 指令是将 FLD4 中的数据传递给 FLD3。由于 FLD3 被定义为 3 个字符空间大小，因此将传递 3 个字节的数据。该指令执行后，FLD3 中的数据将为 F1F2F3。而第二条指令则是将一个直接数传递给了 FLD5。由于 FLD5 被定义为 1 个全字空间大小，因此将传递 4 个字节的数据。该指令执行后，FLD5 中的数据将为 C1C2C3C4。其中直接数的最后一个字符 E 将不被传递。

17.5　数据的运算

数据的运算在程序逻辑部分中经常会用到。其中打包十进制数的运算和定点二进制数的运算是最常见的两种运算，下面分别进行讲解。

17.5.1 打包十进制数的运算

打包十进制数（Packed Decimal）是通过符号 P 描述，主要用于算术运算。其中常用的运算仍然是加减乘除四则运算。用于打包十进制数基本运算的指令都为 SS 类型的指令，并且都属于两操作数长度不等的情况。下面分别进行讲解。

1. 使用 AP 和 SP 指令进行加减运算

AP 指令的功能是实现两个打包十进制数的相加，并将相加的结果存放在第一个操作数中。例如，以下为一段使用 AP 指令的代码。

```
          AP  TOTAL, NUM1
          AP  TOTAL, NUM2
          AP  TOTAL, NUM3
          AP  COUNT, =P'3'
          ......
TOTAL     DC  P'0'
COUNT     DC  P'0'
NUM1      DC  P'1'
NUM2      DC  P'2'
NUM3      DC  P'3'
```

以上前 3 条 AP 指令实现了求 NUM1、NUM2、NUM3 这 3 个打包十进制数的总和。求得的结果被存入 TOTAL 中。TOTAL 也为一打包十进制数，该数据被初始化为 0，最终将为数值 6。第 4 条 AP 指令是将一直接数加入到打包十进制数 COUNT 中。并且，该直接数也为打包十进制数。该指令执行后，COUNT 将为数值 3。

此外，以 AP 指令为基础，还存在一个 ZAP 指令。该指令的功能是先将第一个操作数置为 0，再进行类似 AP 指令的操作。因此，通常可以使用 ZAP 指令实现打包十进制数的数据传递。

SP 指令的功能是实现两个打包十进制数的相减，并将相减的结果存放在第一个操作数中。例如，以下为一段使用 SP 指令的代码。指令执行后，TOTAL 中的数据将为 1，而 COUNT 中的数据将为 2。代码如下。

```
          SP  TOTAL, NUM4
          SP  COUNT, =P'1'
          ......
TOTAL     DC  P'6'
COUNT     DC  P'3'
NUM4      DC  P'5'
```

最后需要注意的是，AP 指令和 SP 指令将可能会改变 PSW 中的 CC 条件码。其中运算结果和 CC 的对应关系如前所述。

2. 使用 MP 和 DP 指令进行乘除运算

MP 指令的功能是实现两个打包十进制数的相乘，并将相乘的结果存放在第一个操作数中。例如，以下为一段使用 MP 指令的代码。

```
          ZAP PRODUCT, NUM1
          MP  PRODUCT, NUM2
```

```
          ......
NUM1      DC  PL1'3'
NUM2      DC   PL2'1000'
PRODUCT   DS   PL3
```

以上指令执行前，NUM1 中的数据为 3C，NUM2 中的数据为 3E8C。当执行第一条指令后，PRODUCT 中的数据为 00003C。当执行完第二条指令后，PRODUCT 中的数据为 00BB8C。

对于 MP 指令，有以下两点需要特别注意。

- 该指令中第二个操作数的长度不能超过 8 个字节，也不能超过第一个操作数的长度。
- 第一个操作数中的前缀 0 不能少于第二个操作数的长度。

如果不满足以上条件，将会产生异常。为避免这两类异常产生，通常可以定义一块专门用于存放相乘运算结果的存储区域。该存储区域的大小为 MP 指令中两操作数空间大小之和。当进行相乘运算前，先通过 ZAP 指令将某一操作数的内容传递给该存储区域。然后，再将该区域作为 MP 指令中的第一个操作数进行相乘运算。例如，以上代码中 PRODUCT 的空间大小为 NUM1 和 NUM2 的大小之和。

DP 指令的功能是实现两个打包十进制数的相除。第一个操作数为被除数，第二个操作数为除数。例如，以下为一段使用 DP 指令的代码。

```
                ZAP  RESULT, NUM3
                DP   RESULT, NUM4
                ......
NUM3            DS  PL5
NUM4            DS   PL3
RESULT          DS   0PL8
QUOTIENT        DS   PL5
REMAINDER       DS   PL3
```

实际上，DP 进行的是整除运算，除得的商和余数都保存在第一个操作数中。其中商在该操作数左边的空间，余数在该操作数右边的空间。余数的长度同除数的长度相等，商的长度则为被除数的长度减去除数的长度。类似于 MP 指令，此处通常也是先将被除数通过 ZAP 传递到一个更大的空间中，再执行 DP 指令。

以上指令执行后，QUOTIENT 中将为除得的商，REMAINDER 中将为除得的余数。被除数 NUM3 和除数 NUM4 中的数据可以在其余代码中通过数据传递得到。

最后需要注意的是，MP 指令和 DP 指令对条件码 CC 都是没有影响的。因此，不可以将打包十进制数的乘除结果作为流程控制中的判断条件。

17.5.2 定点二进制数的运算

定点二进制数的运算通常是以全字或半字为单位进行的运算。并且，此类运算指令中的操作数通常会用到寄存器。定点二进制数的运算也是以加减乘除四则运算作为基本运算，运算结果都保存在第一个操作数中。同打包十进制数的运算类似，该运算中的加减运算结果和 CC 相关，而乘除运算对 CC 没有影响。下面分别根据其中不同的类别进行讲解。

1. 寄存器与主存间的全字运算

此类运算是以全字为单位进行的运算。其中加减乘除四则运算分别使用 A、S、M、D 指令实现。这些指令都属于 RX 类型的指令。以下为这些指令使用方式的示例。

```
A    R3, FWD1
S    R5, FWD2
M    R4, FWD3
D    R2, FWD4
```

需要注意的是，对于乘除指令 M 和 D，其中的第一个操作数实际上为一个偶奇寄存器对。例如，以上 M 指令中，被乘数存放在 R5 中，乘得的结果则存放在 R4 和 R5 形成的双字空间中。对于以上 D 指令，被除数存放在 R2 和 R3 形成的双字空间中。除得的商存放在 R2 中，余数存放在 R3 中。这样做的目的是使保存乘除运算结果的操作数空间足够大，从而不会产生数据溢出。

2．寄存器与主存间的半字运算

此类运算是以半字为单位进行的运算。其中加减乘运算分别使用 AH、SH、MH 指令实现。没有半字除法运算。这些指令也都属于 RX 类型的指令。以下为这些指令使用方式的示例。

```
AH   R2, HWD1
SH   R4, HWD2
MH   R5, =H'20'
```

需要注意的是，由于运算的结果都保存在第一个操作数中，而第一个操作数都为寄存器。因此，半字运算的结果仍然为全字空间大小。此外，当使用 MH 进行半字乘法运算时，运算结果可能会产生数据溢出。当数据溢出时，超出 32 位的高位数据将直接被截断，而不产生任何相应提示信息。

3．寄存器之间的运算

此类运算是以全字为单位进行的运算。其中加减乘除运算分别使用 AR、SR、MR、DR 指令实现。这些指令都属于 RR 类型的指令。以下为这些指令使用方式的示例。

```
AR   R2, R3
SR   R5, R4
MR   R4, R7
DR   R2, R5
```

其中 MR 和 DR 指令同前面讲解的 M 和 D 指令类似，第一个操作数也为一个偶奇寄存器对。因此，这两条指令的运算结果也是都保存在双字空间中的。需要注意的是，对于以偶奇寄存器对表示的操作数，必须以偶数寄存器作为标志。此外，在程序中通常还使用以下这条指令对寄存器 Rn（n 为从 0 到 15 的整数）中的内容进行清空。

```
SR   Rn, Rn
```

17.6 数据的转换

在汇编语言中的数字型数据有 3 种形式，分别为二进制数、打包十进制数以及分区十进制数。其中二进制数和打包十进制数之间的转换通过 CVB 和 CVD 指令实现，打包十进制数和分区十进制数之间的转换通过 PACK 和 UNPK 指令实现。此外，还可以使用 ED 指令将打

包十进制数转换为指定的输出格式。下面分别进行介绍。

17.6.1 使用 CVB 和 CVD 指令转换数据

CVB 指令用于将打包十进制数转换为二进制数，属于 RX 类型的指令。其中用于转换的打包十进制数的大小要求为一个双字空间。转换后的二进制数大小为一个全字空间，并保存在寄存器中。例如，以下为一段使用 CVB 指令进行数据转换的代码。

```
        CVB      R1, PK1
        CVB   R2, PK2
         ......
PK1     DC     PL8'123'
PK2     DC     PL8'-123'
```

以上代码是将 PK1 中的正数 123 转换为二进制数存放在寄存器 R1 中。并将 PK2 中的负数−123 转换为二进制数存放在寄存器 R2 中。在汇编语言中，二进制数也可用十六进制数表示，其中每位十六进制数代表 4 位二进制数。于是，以上指令执行后，原始数据及转换后的结果分别对应如下。

```
PK1 = 00 00 00 00 00 00 12 3C       /*原始打包十进制数，为正数*/
(R1) = 00 00 00 B7                  /*转换后的二进制数，通过十六进制数表示*/
*
PK2 = 00 00 00 00 00 00 12 3D       /*原始打包十进制数，为负数*/
(R2) = FF FF FF 49                  /*转换后的二进制数，通过十六进制数表示*/
```

CVD 指令是将寄存器中的二进制数转换为主存中的打包十进制数，也属于 RX 类型的指令。其中转换后的打包十进制数大小也要求为一个双字空间。该指令相当于 CVB 指令的逆运算。以下为一段使用 CVD 指令进行数据转换的代码。

```
        CVD      R1, PK3
        CVD      R2, PK4
         ......
PK3     DS       PL8
PK4     DS      D
```

以上代码是将 R1 中的二进制数转换为打包十进制数，存放在以 PK3 开头的双字空间中。同时，将 R2 中的二进制数转换为打包十进制数存放在以 PK4 开头的双字空间中。由于存储空间只关注其空间大小，因此使用 PL8 表示和使用 D 表示是等效的，都为 8 个字节空间大小。

最后需要注意的是，该指令是将运行结果保存到第二个操作数中的。本章所讲解的指令只有 ST、STH、STM 这几条指令是与之类似的。其余指令都是将运行结果保存在第一个操作数中。

17.6.2 使用 PACK 和 UNPK 指令转换数据

PACK 指令用于将分区十进制数转换为打包十进制数。该指令为 SS 类型的指令，对应两操作数长度不等的情况。其中从文件中读入的数据通常为分区十进制数，而在程序中用于运算的则为打包十进制数。分区十进制数通过 EBCDIC 码表示，每个数字占一个字节。而打包十进制数则是每两个数字占一个字节，并在数据最后含有一个符号位。以下为一段使用 PACK 指令进行数据转换的代码。

```
        PACK  NUM1, FLD
        PACK  NUM2, FLD
        PACK  NUM3, FLD
……
FLD   DC X'F1F2F3F4F5F6'
NUM1  DS PL1
NUM2  DS PL2
NUM3  DS PL3
```

实际上，PACK 指令属于 SS 类型的指令，并且对应两操作数长度不等的情况。当转换后的打包十进制数空间过剩时，使用 0 在高位填充；当空间不足时，将高位部分截断。因此，对于以上代码，指令执行后各打包十进制数的内容如下。

```
NUM1 = 45 6F
NUM2 = 01 23 45 6F
NUM3 = 00 00 01 23 45 6F
```

可以看到，当使用 PACK 进行数据转换时，对于最后一个字节是将高 4 位和低 4 位进行的互换。例如，若原始数据的最后一个字节内容为 F6，则转换后的数据最后一个字节内容将为 6F。对于原始数据前面的内容，则是将连续两个字节中的数字部分提取，并转换为一个字节。例如，若原始数据非最后一位的两个连续字节中的内容为 F1F2，则转换后将为一个字节，内容为 12。

UNPK 指令用于将打包十进制数转换为分区十进制数，相当于 PACK 指令的逆运算。该指令也为 SS 类型的指令，对应两操作数长度不等的情况。以下为一段使用 UNPK 指令进行数据转换的代码。

```
        UNPK  CFLD, NUM
        OI      CFLD+7, X'F0'
        ……
CFLD  DS F
NUM   DC P'123'
```

打包十进制数中最后一个字节的低 4 位通过 C 和 D 表示其正负号。因此，当将其转换为分区十进制数进行输出时，通常需要通过 OI 指令对该 4 位进行处理。通过下面 CFLD 数据在以上指令执行过程中的不同内容可以看出进行处理的必要性。CFLD 数据在执行过程中的内容如下。

```
CFLD = 00 F1 F2 C3          /*在 UNPK 指令执行后的内容*/
CFLD = 00 F1 F2 F3          /*在 OI 指令执行后的内容*/
```

最后需要说明的是，此处的分区十进制数相当于 COBOL 中通过 PIC X 定义的字符型数据。并且字符均为数字字符。而打包十进制数则相当于 COBOL 中通过 PIC 9 定义的数值型数据，可用于算术运算。在较底层的汇编语言中，可以实现二者的转换。

17.6.3 使用 ED 指令转换数据

ED 指令用于将打包十进制数转换为指定的格式以进行输出。这点和 COBOL 语言中 Numeric Edited Fields 格式输出类型的应用是类似的。ED 指令属于 SS 类型的指令，对应两操作数长度相等的情况。该指令的使用格式如下。

```
ED  MASK, PK
```

以上指令中，PK 为用于转换的打包十进制数，MASK 为转换掩码。转换掩码用于指定转换后的格式。并且，该指令执行后，转换后的数据将存放在 MASK 中，覆盖 MASK 原有的转换掩码。

在 ED 指令中，还存在一个有效数指示器（Significance indicator）。当该指示器为开启状态时，直接转换为相应的数字或消息字符；当指示器为关闭状态时，将相应位置的数字或消息字符转换为填充字符。其中填充字符由转换掩码的第一个字节指定。

转换掩码除第一个字节用于指定填充字符外，其余部分的内容通常由以下字符组成。

- X’20’：对应打包十进制数中的一个数字，转换方式根据有效数指示器而定。
- X’21’：同 X’20’功能类似，不过在转换后将有效数指示器置为开启状态。
- X’22’：仅用于同时对多个打包十进制数进行转换的情况。相当于分隔字符，并将有效数指示器置为关闭状态。
- 消息字符：用于在数据中插入相应符号。常用的消息字符有“,”（6B）、“.”（4B）、“-”（60）。

进行数据转换时，有效数指示器初始状态为关闭。在转换中当遇到第一个非 0 数字时，指示器开启；当遇到正号时，指示器关闭。以下为一段使用 ED 进行数据转换的代码。

```
                MVC    PRTDATA1, MASK
                ED   PRTDATA1, PK1
                MVC    PRTDATA2, MASK
                ED   PRTDATA2, PK2
                MVC    PRTDATA3, MASK
                ED   PRTDATA3, PK3
……
PK1          DC   PL4'1234567'
PK2          DC   PL4'123'
PK3          DC   PL4'-123'
PRTDATA1     DS   CL11
PRTDATA2     DS   CL11
PRTDATA3     DS   CL11
MASK         DC   X'40206B2021204B202060'
```

以上代码主要用于说明转换掩码中包含有常用消息字符的基本转换方式。其中各原始数据及转换后的数据对应关系如下。

```
PK1 = 1234567C
PRTDATA1 = 12,345.67       /*直接对各有效数字进行转换，同时在相应位置插入消息字符*/
PK2 = 0000123C
PRTDATA2 =     1.23        /*前缀无效数字 0 及消息字符“, ”用填充字符空格取代*/
PK3 = 0000123D
PRTDATA3 =     1.23-       /*结尾非“+”号，指示器不关闭，对“-”号直接转换*/
```

注意到，以上前两个数据都为正数，因此末尾的“+”号使用填充字符空格转换，而不显示输出。这点同 COBOL 语言中算术符号格式里的 DR 与 CR 只在负数时才输出是类似的。当不关心数据的符号时，由于转换过程是从左至右的，因此也可以省略转换掩码中最右边的 60。

以下这段代码通过对比反映了转换掩码中 20 与 21 的用法。转换后 PRT1 中的数据将为“1”，PRT2 中的数据将为“.01”，PRT3 中的数据将为“0.01”。代码如下。

```
          ED  PRT1,  PK
          ED  PRT2,  PK
```

```
        ED  PRT3,  PK
        ……
PRT1    DC  X'4020204B2020'
PRT2    DC  X'4020214B2020'
PRT3    DC  X'4021204B2020'
PK      DC  PL2'1'
```

此外，填充字符除通常使用空格（40）表示外，有时也会使用“*”（5C）表示。例如，以下指令执行后，PRTFMT 中的数据将为“***012”。代码如下。

```
         ED  PRTFMT, PK0
         ……
PK0      DC  PL3'12'
PRTFMT   DC  X'5C202021202020'
```

最后需要注意的是，使用 ED 指令是会改变条件码 CC 的。其中转换的数据为 0 时，CC 为 0；转换的数据小于 0 时，CC 为 1；转换的数据大于 0 时，CC 为 2。

17.7 跳转指令与宏命令

跳转指令主要用于对程序流程进行控制。本节仅介绍两种常用的跳转指令，分别为 BC 指令和 B 指令。其中 BC 指令的格式如下。

```
BC  B'mask', addr
```

该指令第一个操作数中的 mask 由 4 位二进制数组成。其中第 1 位对应 CC 为 0 的情况，第 2 位对应 CC 为 1 的情况，第 3 位对应 CC 为 2 的情况，第 4 位对应 CC 为 3 的情况。当某位为 1 时，表示其对应的 CC 值满足时进行跳转。该指令的第二个操作数 addr 则表示将要跳转到的地址。例如，以下这条指令就表示当 CC 为 0 或 2 时跳转到 BRANCH 表示的地址处执行。指令如下。

```
BC B'1010', BRANCH
```

B 指令则相对要简单一些。该指令只含一个操作数，操作数中的内容为所要跳转到的地址。B 指令表示无条件跳转到该地址处执行。

汇编中的宏命令是由多条指令组成的，用于实现某一特定的功能，并用一条命令语句表示。常用的汇编语言宏命令如下。

- OPEN：用于打开一个文件。
- GET：用于从文件中读入数据。
- PUT：用于将数据写入文件。
- CLOSE：用于关闭文件。

以下这段代码综合应用了跳转指令和汇编宏命令。该段代码实现的功能是通过一个循环结构依次读取文件 INFILE 中的每条记录。同时，每读取一条记录，便将该记录中的数据输出到另一文件 OUTFILE 中，直至文件读取结束。代码如下。

```
         OPEN  (INFILE, (INPUT), OUTFILE, (OUTPUT))
LOOP     EQU   *
         GET   INFILE, INAREA
```

```
            MVC   OUTREC, INAREA
            PUT   OUTFILE, OUTAREA
            B     LOOP
ENDDATA     EQU *
            CLOSE (INFILE , , OUTFILE)
```

以上代码中的文件名 INFILE、OUTFILE 以及文件结束地址 ENDDATA 都是在 DCB 参数中指定的。关于 DCB 参数，将在下一节中进行讲解。

17.8 程序模块化与 DCB 参数

在 z/OS 环境中编写汇编语言程序时，通常需要在编写程序逻辑代码之前，对程序进行模块化处理。对程序进行模块化处理的目的是使其能够实现不同程序间的互相调用。这些不同的程序既包括汇编语言程序，也包括 COBOL 语言程序。其中模块化处理的代码如下。

```
STM    R14, R12, 12(R13)        /*该指令保存调用程序的寄存器*/
BALR   R12, R0                  /*以下两条指令建立本程序的地址*/
USING  *, R12
LR     R11, R13                 /*该指令保存调用程序的 SAVEAREA 地址*/
LA     R13, SAVEAREA            /*该指令将本程序的 SAVEAREA 地址存入 R13 中*/
ST     R11, 4(, R13)            /*以下两条指令将两程序的 SAVEAREA 地址链接起来*/
ST     R13, 8(, R11)
```

关于该段代码，有以下几条指令需要注意。

- BALR R12, R0：该指令执行后，R12 将包含下一条指令的地址。同时，程序跳转到 R0 所包含的地址处执行。
- USING *, R12：该指令将 R12 中的内容作为本程序的绝对地址。“*”为一个地址标志，通常在后面的代码中通过 EQU 将其具体指定。
- LA R13, SAVEAREA：该指令将 SAVEAREA 的地址传递给 R13。

其中 SAVEAREA 为 18 个全字空间，用于保存各寄存器中的数据及其他相关信息。实际上，SAVEAREA 可以认为是程序调用中用于保护现场的一段存储空间。并且，其名字可以任意指定，如 SAVE1、SAVE2 等。

DCB 即数据控制块（Data Control Block），主要用于指定 z/OS 环境下汇编语言程序用到的数据集。通常情况下，这些数据集对应于汇编语言程序中的输入输出文件。例如，以下分别为输入文件 INFILE 及输出文件 OUTFILE 的 DCB 宏命令。

```
INFILE    DCB     DDNAME = SYSIN, RECFM = FB, LRECL = 80, DSORG = PS,
                  MACRF = GM, EODAD = ENDDATA
OUTFILE   DCB     DDNAME = SYSPRINT, RECFM = FBA, LRECL = 133, DSORG = PS,
                  MACRF = PM, BLKSIZE = 6650
```

关于以上代码中 DCB 的各关键字参数，分别介绍如下。

- DDNAME：指定文件对应的外部数据集名。该数据集名需与其后 JCL 语句中的数据集名相对应。
- RECFM：指定数据集的记录格式。通常用于输入的数据集为 FB，用于输出的数据集为 FBA。
- LRECL：指定逻辑记录的长度。通常用于输入的数据集为 80，用于输出的数据集为 133。

- DSORG：指定数据集的组织形式。通常为顺序数据集，故使用 PS 表示。
- MACRF：指定访问数据集记录的宏命令的类型。通常用于输入的数据集为 GM（Get Macro），用于输出的数据集为 PM（Put Macro）。
- EODAD：指定文件读取结束后的地址标号。该参数只在用于输入的数据集中存在。
- BLKSIZE：指定物理记录的长度。该参数在用于输入的数据集中默认，在用于输出的数据集中通常为 6650。

以下为汇编语言程序之后的两条 JCL 语句。可以看到，第一条 JCL 语句中的“SYSIN”与 INFILE 文件的 DCB 参数 DDNAME 中的内容对应，第二条 JCL 语句中的“SYSPRINT”与 OUTFILE 文件的 DCB 参数 DDNAME 中的内容对应。相应 JCL 语句如下。

```
//G.SYSIN    DD   DSN = ADCDA.ASM.DATA, DISP = SHR
//G.SYSPRINT DD   SYSOUT = *
```

结合以上两段 DCB 宏命令，程序实际上将数据集 ADCDA.ASM.DATA 作为输入文件。而显示屏则为输出文件，数据直接输出到显示屏上。由此可见，DCB 的功能实际上同 COBOL 语言中环境部的功能相类似。

在 z/OS 环境下的汇编语言程序基本结构通常都包含有程序模块化代码和 DCB 参数。其中程序模块化代码在程序逻辑部分之前编写，DCB 参数在数据定义部分之后指定。具体形式可参照本章第 1 节的最后一个小节中程序基本结构的示例。

17.9 综合实例

前面分别对大型机汇编语言各方面的知识进行了讲解。下面通过两个汇编语言程序的综合实例，以加深对于汇编语言的掌握，增强实际应用能力。

17.9.1 输出商品报表实例

该程序实现的功能是读取商品信息输入文件的每条记录，并按指定格式生成商品信息报表。其中输入文件的商品记录数据存放在数据集 ADCDA.ASM.DATA01 中。每条商品记录为 80 列，每列数据占用 1 个字节。记录内容分别如下。

- 1 ~ 5 列：商品编号。
- 6 ~ 9 列：商品数量。
- 10 ~ 20 列：商品名称。
- 21 ~ 24 列：商品单价。
- 25 ~ 80 列：未用到的部分，用空格表示。

生成的报表要求反映真实数据，并且易于阅读。同时，报表中的每条记录中还应增加一个商品总价的记录项。并且，在报表最后一行显示所有商品的总价。报表内容直接在显示屏上输出。例如，若输入文件中的原始记录数据如下。

```
000010100SOAP        00350
000022000TOOTHBRUSH 00200
000030005BICYCLE     10000
000040025BREAD       00500
000050300CUP         02000
```

生成的报表应该如下。

```
00001     100     SOAP            3.50          350.00
00002    2000     TOOTHBRUSH      2.00        4,000.00
00003       5     BYCYCLE       100.00          500.00
00004      25     BREAD           5.50          137.50
00005     300     CUP            20.00        6,000.00
                                             10,987.50
```

实现以上功能，需要综合应用到前面所讲解的数据的传递、运算、转换等相关知识。完整的汇编语言程序代码如下。

```
……       /*相关 JCL 语句，根据不同的系统配置而有所不同*/
 LAB01    CSECT
          STM   R14, R12, 12(R13)
          BALR  R12, R0
          USING  *, R12
          LR     R11, R13
          LA     R13, SAVE01
          ST     R11, 4(, R13)
          ST     R13, 8(, R11)
*****  BEGIN  LOGIC  *****
          OPEN    (INFILE, (INPUT), OUTFILE, (OUTPUT))
          ZAP     SUM, =P'0'
 LOOP     EQU     *
          GET     INFILE, INAREA
          PACK    NUM1, INSUB2
          PACK    NUM2, INSUB4
          ZAP     NUM3, NUM1
          MP      NUM3, NUM2
          ZAP     NUM4, NUM3
          AP      SUM, NUM4
          MVC     RESULT, =X'4020206B2020214B2020'
          ED      RESULT, NUM4
          MVC     OUTREC1, INSUB1
          MVC     OUTREC2, =X'4020202020'
          ED      OUTREC2, NUM1
          MVC     OUTREC3, INSUB3
          MVC     OUTREC4, =X'402020214B2020'
          ED      OUTREC4, NUM2
          PUT     OUTFILE,OUTAREA
          B       LOOP
 ENDDATA  EQU     *
          MVC     OUTAREA, =CL132' '
          MVC     RESULT, =X'4020206B2020214B2020'
          ED      RESULT, SUM
          PUT     OUTFILE, OUTAREA
          CLOSE   (INFILE,,OUTFILE)
*****  END  LOGIC  *****
 RETURN    EQU         *
          L       R13, SAVEAREA+4
          RETURN  (14, 12), RC=0
*****  DATA  AREA  *****
          LTORG
 INAREA    DS      0CL80
```

```
INSUB1    DS      CL5
INSUB2    DS      ZL4
INSUB3    DS      CL11
INSUB4    DS      ZL5
INREST    DS      CL55
          ORG
OUTAREA   DS      0CL133
OUTASA    DS      C'0'
OUTREC1   DS      CL5
          DC      CL6' '
OUTREC2   DS      ZL5
          DC      CL4' '
OUTREC3   DS      CL11
          DC      CL5' '
OUTREC4   DS      ZL7
          DC      CL6' '
RESULT    DS      CL10
          DC      CL73' '
NUM       DS      0D
NUM1      DS      PL2
NUM2      DS      PL3
NUM3      DS      PL5
NUM4        DS    PL4
SUM       DS      PL4
INFILE    DCB     DDNAME = SYSIN, RECFM = FB, LRECL = 80, DSORG = PS,
                  MACRF = GM, EODAD = ENDDATA
OUTFILE  DCB DDNAME = SYSPRINT, RECFM = FBA, LRECL = 133, DSORG = PS,
                  MACRF = PM, BLKSIZE = 6650
SAVE01    DC      18F'0'
          PRINT GEN
          YREGS
          LTORG
          END  LAB01
//G.SYSIN     DD   DSN = ADCDA.ASM.DATA01, DISP = SHR
//G.SYSPRINT DD   SYSOUT = *
```

17.9.2 显示系统时间实例

大型机中的系统时间可以通过宏命令 TIME BIN 得到。该命令将得到的系统时间存放在寄存器 R0 中。不过，系统时间是以秒为单位的。现要求将其转换为日常小时制的时间，并在显示屏上依次输出当前系统时间转换后的小时、分钟和秒。

为实现以上功能，可使用两段汇编语言程序完成。其中主程序 LAB02 用来得到相应的小时、分钟和秒的数值。其后，该程序再调用子程序 PUTTIME，用以实现时间的输出。LAB02 主程序代码如下。

```
......
 LAB02  CSECT
           STM    R14, R12, 12(R13)
           BALR   R12, R0
           USING  *, R12
           LR      R11, R13
           LA      R13, SAVE02
           ST      R11, 4(, R13)
```

```
            ST     R13, 8(, R11)
*****  BEGIN  LOGIC  *****
            TIME  BIN
            LR     R3, R0
            SR     R2,R2
            D      R2, =F'100'
            SR     R2, R2
            D      R2, =F'3600'
            ST     R3,HOURS
*
            LR     R3, R2
            SR     R2, R2
            D      R2, =F'60'
            ST     R3, MINS
*
            ST     R2, SECS
*
            LA     R1, PARMLIST
            L      R15, =V(PUTTIME)
            BALR R14, R15
*****  END  LOGIC  *****
            ......
*****  DATA  AREA  *****
 HOURS      DS      F
 MINS       DS      F
 SECS       DS      F
 PARMLIST   DC      A(HOURS, MINS, SECS)
 SAVE02     DC      18F'0'
            PRINT GEN
            YREGS
            LTORG
            END  LAB02
```

以上程序执行后，以秒为单位的系统时间将被转换为日常小时制的时间。其中小时、分钟和秒分别存放在 HOURS、MINS、SECS 这 3 个全字空间中。此后，该程序将以上这 3 个变量作为参数，传递给子程序 PUTTIME，用以实现正确的输出。

子程序 PUTTIME 将传递的 3 个参数分别存放在寄存器 R5、R6、R7 中。之后，程序将寄存器中的数据转换为打包十进制数。通过输出格式转换后，程序最终将得到的有效数据通过 OUTBUF 写入显示屏进行输出。该程序代码如下。

```
......
 PUTTIME  DSECT
          STM    R14, R12, 12(R13)
          BALR   R12, R0
          USING  *, R12
          LR      R11, R13
          LA      R13, SAVE002
          ST      R11, 4(, R13)
          ST      R13, 8(, R11)
*****  BEGIN  LOGIC  *****
          LM      R2, R4, 0(R1)
          LM      R5, R7, 0(R2)
          CVD     R5, HD
```

```
         CVD      R6, MD
         CVD      R7, SD
         ED       EHOUR, HD
         ED       EMIN, MD
         ED       ESEC, SD
         MVC      HOUR, EHOUR
         MVC      MIN, EMIN
         MVC      SEC, ESEC
         MVC      OHOUR, HT
         MVC      OMIN, MT
         MVC      OSEC, ST
         OPEN (OUTAREA, OUTPUT)
         PUT      OUTAREA, OUTBUF
         CLOSE    (OUTAREA)
*****  END  LOGIC  *****
 RETURN  EQU      *
         L        R13, SAVEAREA+4
         RETURN   (14, 12), RC=0
*****  DATA  AREA  *****
 OUTBUF  DS   0CL133
         DC   CL1'1'
         DC   C'CURRENT HOURS:'
 OHOUR   DS   CL2
 FILLER     DC  CL5' '
         DC   C'CURRENT MINUTES:'
 OMIN    DS   CL2
 FILLER  DC  CL5' '
         DC   C'CURRENT SECONDS:'
 OSEC    DS   CL2
 FILLER  DS  CL70
 HOUR    DS   0CL16
         DS   CL14
 HT      DS   CL2
 MIN     DS   0CL16
         DS   CL14
 MT      DS   CL2
 SEC     DS   0CL16
         DS   CL14
 ST      DS   CL2
 EHOUR   DC   X'402020202020202020202020202020212121'
 EMIN    DC   X'402020202020202020202020202020212121'
 ESEC    DC   X'402020202020202020202020202020212121'
 HD      DS   D
 MD      DS   D
 SD      DS   D
 OUTAREA DCB  DDNAME = SYSPRINT, RECFM = FBA, LRECL = 133, BLKSIZE = 6650,
              MACRF = PM, DSORG = PS
 SAVE002 DC   18F'0'
         PRINT GEN
         YREGS
         LTORG
         END  PUTTIME
//G.SYSIN    DD   *
//G.SYSPRINT  DD   SYSOUT = *
```

最后，通过在主程序 LAB02 中调用该程序，便可实现本实例所指定的功能。若系统当

前时间为 10000s，以上两段程序执行后，在显示屏上输出的信息应该如下。

```
CURRENT HOURS:2    CURRENT MINUTES:46    CURRENT SECONDS:40
```

17.10 本章回顾

本章主要介绍了大型机中的汇编语言。由于 COBOL 语言主要是应用于大型机上的，而汇编语言从更底层上反映了大型机程序的工作原理。因此，学好汇编语言对于更深入地理解 COBOL 语言是有必要的。

本章首先介绍了大型机汇编语言所涉及到的一些基本概念。关于这些基本概念，需要牢固理解全字、半字和双字的意义；熟练掌握各种数据的原码、1 的补码（反码）和 2 的补码（补码）的表示方式；理解寄存器和程序状态字（PSW）的概念；掌握汇编指令中使用主存地址表示操作数的方式；了解分别运行于 ASSIST 软件下和 z/OS 系统下汇编语言程序的基本结构。

本章之后介绍了汇编语言程序中各种指令类型及所对应的机器码。其中大型机汇编语言的指令类型分别包括 RR 类型、RX 类型、RS 类型、SI 类型以及 SS 类型。学习指令类型及机器码，需要能够判断任意常用汇编指令的类型，并熟练写出该指令所对应的机器码。

本章接下来重点介绍了汇编语言程序中数据的定义、传递、运算以及转换。该部分内容是本章的重点，涉及到较多的汇编指令。学习该部分内容，需要能够熟练应用各种常用汇编指令，并能以此编写具有一定功能的汇编语言程序。

此后，本章分别介绍了汇编语言中的跳转指令、宏命令、程序模块化以及 DCB 参数的概念及应用。其中程序模块化和 DCB 参数主要应用于在 z/OS 环境下开发的汇编语言程序。对于该部分内容，需要大致了解其原理，以理解并编写 z/OS 环境下完整的汇编语言程序。

本章在最后给出了两个汇编语言程序的实例。通过实例，以综合理解和掌握汇编语言，并增强实际应用的能力。总之，学习本章需要结合前面所学内容，从更底层的角度理解大型机上程序的工作原理和设计开发。

第 18 章 开发小型银行账户管理信息系统

本章主要介绍如何使用 COBOL，以及 CICS 实际开发出一个小型的银行账户管理信息系统。该 MIS 系统共分为 5 个功能模块。这 5 个模块分别为主菜单模块和对账户的增、删、改、查功能模块。下面根据不同的模块分别进行介绍。

18.1 主菜单模块

主菜单界面通过 CICS 中的 MAP 绘制。用户可以在该界面上选择执行增、删、改、查操作中的某一项。同时，该界面中倒数第二行 MESSAGE LINE 也会根据用户的输入产生相应的提示信息输出。主菜单模块的界面如图 18.1 所示。

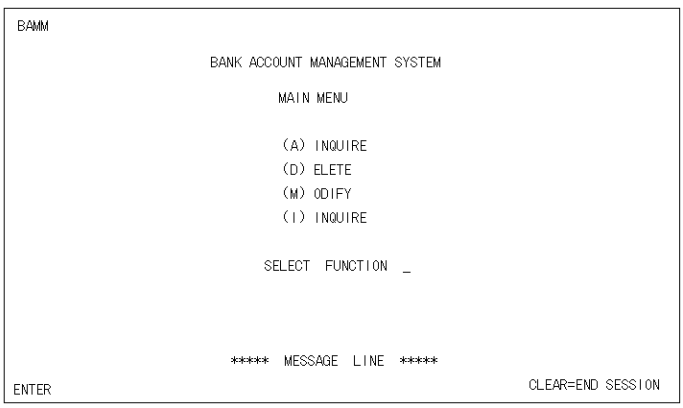

图 18.1 主菜单模块界面

需要注意的是，其他各个功能模块也都是有界面的，都为 CICS 中的 MAP。后面不再对此另行说明。

在以上界面中，下划线部分为用户输入区域，用以选择执行不同的功能。MESSAGE LINE 部分则为系统的信息输出区域。本模块中系统可输出的提示信息有以下几条。

- 提示信息 1：INVALID KEY PRESSED（表示用户按了非法的按键）。
- 提示信息 2：INVALID OPTION SELECTED（表示用户输入了非法的功能选项）。
- 提示信息 3：INPUT DATA REQUIRED（表示用户输入数据为空）。
- 提示信息 4：PROCESSING ERROR（表示系统在处理过程中发生错误）。
- 提示信息 5：MAIN SESSION ENDED（表示退出系统）。

在实际软件项目中，通常是需要按照严格的开发流程进行的。开发流程依次为项目计划、需求分析、可行性分析、概要设计、详细设计、编码、测试、维护，并且在每一个具体环节中都需要提供相应的文档。此处作为基础教程，仅简单的给出任务要求及相应的代码。其中主菜单模块的任务要求涵盖以下几个步骤。

（1）显示界面，并允许用户在界面上输入数据。

（2）根据用户在输入数据前后所按的不同功能按键，分别作出如下处理。

- “Enter”键：执行步骤 3。
- Pause/Break 键：在新页面首行输出提示信息 5，同时退出该系统。
- 其他键：输出提示信息 1，重新执行步骤 2。

（3）根据用户的输入数据，分别作出如下处理。

- 输入数据为空或者为空格，则输出提示信息 3，返回执行步骤 2。
- 输入数据非法，则输出提示信息 2，返回执行步骤 2。
- 输入数据合法，执行步骤 4。

（4）根据用户输入的合法数据，分别作出如下处理。

- 输入数据为“A”：执行添加账户功能模块。
- 输入数据为“D”：执行删除账户功能模块。
- 输入数据为“M”：执行修改账户功能模块。
- 输入数据为“I”：执行查询账户功能模块。

（5）如果在处理过程中遇到任何异常，则输出提示信息 4，同时退出该系统。

不妨设该模块所用到的各项资源名称分别如下。

- 交易名称：BAMM。
- 程序名称：BAMSMM。
- 界面所对应的 MAP 名称：BAMAPMM。
- MAP 所在的 MAPSET 名称：BAMPSMM。

同时，令符号 MAP 中的各变量如下。

- SELECT-M：对应“SELECT FUNCTION _”中的下划线。
- DISPL-M：对应系统信息输出部分。

实现该模块所有功能的完整程序代码如下。

```
IDENTIFICATION DIVISION.
PROGRAM-ID BAMSMM.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 COMSTART.
   05 STATUS-I  PIC X VALUE 'N'.
01 ERR-CODE     PIC S9(8) COMP.
```

```
 01 MSG1         PIC X(44) VALUE SPACES.
 01 OUT1         PIC X(79) VALUE SPACES.
 COPY DFHAID.
 COPY BAMPSMM.
 LINKAGE SECTION.
 01 DFHCOMMAREA.
     05 STATUS-C    PIC X.
 PROCEDURE DIVISION.
*** 当任务第一次执行时，直接输出MAP界面，并RETURN到自身 ***
      IF EIBCALEN = 0 THEN
          MOVE LOW-VALUES TO BAMAPMMO
          EXEC CICS SEND MAP('BAMAPMM')
                         MAPSET('BAMPSMM')
                         CURSOR(1337)
                         ERASE
                         FREEKB
                         END-EXEC
          EXEC CICS RETURN TRANSID('BAMM')
                           COMMAREA(COMSTART)
                           LENGTH(1)
                           END-EXEC.
*** 当用户按<Pause/Break>按键时，输出相应提示信息，并退出该MIS系统 ***
      IF EIBAID = DFHCLEAR THEN
        MOVE ' MAIN SESSION ENDED ' TO OUT1
        EXEC CICS SEND FROM(OUT1)
                       LENGTH(79)
                       ERASE
                       END-EXEC
        EXEC CICS RETURN END-EXEC.
*** 当用户按<Enter>按键时，接受用户输入的数据，同时进行异常处理 ***
      IF EIBAID = DFHENTER THEN
            EXEC CICS RECEIVE MAP('BAMAPMM')
                              MAPSET('BAMPSMM')
                              RESP(ERR-CODE)
                              END-EXEC
            IF ERR-CODE NOT EQUAL DFHRESP(NORMAL)
            MOVE ' PROCESSING ERROR ' TO OUT1
            EXEC CICS SEND FROM(OUT1)
                           LENGTH(79)
                           ERASE
                           END-EXEC
            EXEC CICS RETURN END-EXEC
            END-IF
*** 判断用户输入的数据，并根据判断结果进行不同的处理 ***
       MOVE LOW-VALUES TO MSG1
       EVALUATE TRUE
         WHEN  SELEC-MI  IS EQUAL TO SPACES OR
                            SELEC-ML  IS EQUAL TO ZERO
           MOVE 'INPUT DATA REQUIRED' TO MSG1
        WHEN SELEC-MI = 'A'
           EXEC CICS XCTL PROGRAM('BAMSAF')
                          END-EXEC
        WHEN SELEC-MI = 'D'
           EXEC CICS XCTL PROGRAM('BAMSDF')
                          END-EXEC
```

```
          WHEN SELEC-MI = 'M'
             EXEC CICS XCTL PROGRAM('BAMSMF')
                            END-EXEC
          WHEN SELEC-MI = 'I'
             EXEC CICS XCTL PROGRAM('BAMSMF')
                            END-EXEC
          WHEN OTHER
            MOVE ' INVALID OPTION SELECTED ' TO MSG1
        END-EVALUATE
*** 当用户按除 Pause/Break 和" Enter" 之外的按键时进行的相应处理 ***
       ELSE
          MOVE 'INVALID KEY PRESSED' TO MSG1
       END-IF
*** 输出界面，并 RETURN 到自身 ***
          MOVE LOW-VALUES TO BAMAPMMO
          MOVE MSG1  TO  DISPL-MO
          EXEC CICS SEND MAP('BAMAPMM')
                         MAPSET('BAMPSMM')
                         CURSOR(1337)
                         ERASE
                         FREEKB
                         END-EXEC
          EXEC CICS RETURN TRANSID('BAMM')
                           COMMAREA(COMSTART)
                           LENGTH(1)
                           END-EXEC.
             GOBACK.
```

最后需要注意的是，当用户输入“M”和“I”时将调用同一个程序。该程序对应修改账户功能模块。原因在于查询账户功能模块实际上是通过修改账户功能模块调用的。当在修改账户功能模块中输入账号的部分信息时，将进行模糊查找，并调用查询账户功能模块。

18.2 添加账户功能模块

在添加账户功能模块的界面上，用户可以输入要添加的账户账号和姓名。同时，用户也可以确定或撤销所要添加的账户。此外，用户还可在该界面上连续添加多个账户或者选择返回主菜单。添加账户功能模块的界面如图 18.2 所示。

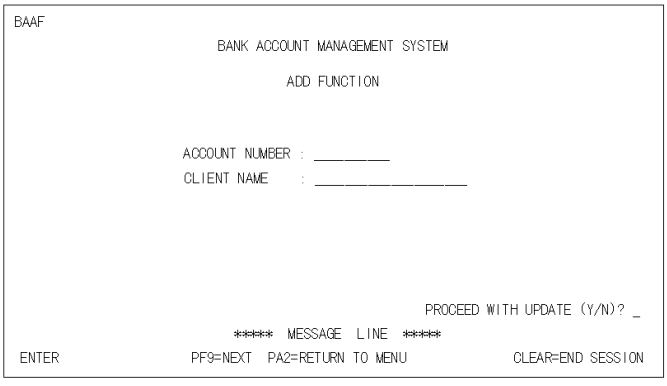

图 18.2 添加账户功能模块界面

在添加账户功能模块中，系统可输出的提示信息有以下几条。

- 提示信息 1：INVALID KEY PRESSED（表示用户按了非法的按键）。
- 提示信息 2：INPUT DATA REQUIRED（表示用户输入数据为空）。
- 提示信息 3：ACCOUNT NUMBER NOT NUMERIC（表示输入的账号不全为数字）。
- 提示信息 4：ACCOUNT ALREADY EXISTS（表示添加的账户已经存在）。
- 提示信息 5：ACCOUNT ADDED TO FILE（表示将账户成功添加到数据文件中）。
- 提示信息 6：UPDATE PROCESS CANCELED（表示用户撤销了添加账户操作）。
- 提示信息 7：ENTER “Y” OR “N”（要求用户输入确认或撤销信息）。
- 提示信息 8：PROCESSING ERROR（表示系统在处理过程中发生错误）。
- 提示信息 9：ADD SESSION ENDED（表示退出系统）。

添加账户功能模块的任务要求如下。

（1）显示界面，允许用户输入账号和姓名信息。但“PROCEED WITH UPDATE (Y/N)? _”以及“PF9=NEXT”不显示，且下划线位置不允许输入。

（2）根据用户在输入数据前后所按的不同功能按键，分别作出如下处理。

- “Enter”按键：执行步骤 3。
- Pause/Break 按键：在新页面首行输出提示信息 9，同时退出该系统。
- “PA2（F2）”按键：返回主菜单。
- 其他按键：输出提示信息 1，重新执行步骤 2。

（3）根据用户输入的账号和姓名信息，分别作出如下处理。

- 如果没有输入或输入的为空格，则输出提示信息 2。同时将光标定位到账号信息输入首位，并返回执行步骤 2。
- 如果输入的账号不全为数字，则输出提示信息 3。同时账号信息高亮显示，不允许输入，再返回执行步骤 2。
- 判断账号信息，如果在数据文件中存在该账号，则输出提示信息 4。同时将账号和姓名信息都高亮显示，不允许输入，再返回执行步骤 2。
- 如果数据文件中不存在该账号，则执行步骤 4。

（4）重新输出界面。此时将“PROCEED WITH UPDATE (Y/N)? _”高亮显示，并在下划线处允许输入。同时账号和姓名部分不允许输入。

（5）根据用户在输入数据前后所按的不同功能按键，分别作出如下处理。

- “Enter”按键：执行步骤 6。
- Pause/Break 按键：在新页面首行输出提示信息 9，同时退出该系统。
- “PA2（F2）”按键：返回主菜单。
- 其他按键：输出提示信息 1，重新执行步骤 5。

（6）根据用户输入的确认或撤销信息，分别作出如下处理。

- 如果没有输入或输入的为空格，则输出提示信息 2，返回执行步骤 5。
- 如果输入的为“Y”，则将账户信息写入数据文件，输出提示信息 5，执行步骤 7。
- 如果输入的为“N”，则输出提示信息 6，执行步骤 7。
- 如果输入的为其他字符，则输出提示信息 7，返回执行步骤 5。

（7）高亮显示“PF9=NEXT”，同时不显示“ENTER”。此时用户按下快捷键 F9，则返回

执行步骤 1；若按其他按键，包括 Enter 键，则输出提示信息 1，返回执行步骤 4。

（8）如果在处理过程中遇到任何异常，则输出提示信息 4，同时退出该系统。

其中本系统中用于存放数据的文件为 VSAM 文件，该文件的属性及每条记录所包含的数据项如下：

- 文件类型：KSDS。
- 逻辑记录长度：60。
- 主关键字：ACCOUNT NUMBER + ACCOUNT SBA CODE。
- 次关键字：CLIENT NAME。
- ACCOUNT NUMBER 数据项：10 字节，数字类型。
- ACCOUNT SBA CODE 数据项：3 字节，数字类型，内容全 0，用于分割账号和姓名。
- CLIENT NAME 数据项：20 字节，字符类型。
- NOT USED 数据项：27 字节，字符类型，内容为空。

设该模块所用到的其他各项资源名称分别如下。

- 交易名称：BAAF。
- 程序名称：BAMSAF。
- 界面所对应的 MAP 名称：BAMAPAF。
- MAP 所在的 MAPSET 名称：BAMPSAF。

令本模块中符号 MAP 里的各变量如下。

- ACC-N2：对应账号信息部分。
- CLI-N2：对应姓名信息部分。
- PROCE-2：对应“PROCEED WITH UPDATE (Y/N)?”字段。
- UPDATE-2：对应“PROCEED WITH UPDATE (Y/N)? _”中的下划线。
- NEXT-2：对应“PF9=NEXT”字段。
- ENTER-2：对应“ENTER”字段。
- DISPL-2：对应系统信息输出部分。

实现该模块所有功能的完整程序代码如下。

```
IDENTIFICATION DIVISION.
PROGRAM-ID BAMSAF.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 COMSTART.
   05 STATUS-I    PIC X VALUE 'F'.
   05 ACCOUN-I    PIC X(10) VALUE SPACES.
   05 CLIENT-I    PIC X(20) VALUE SPACES.
01 DATA-FILED.
   05 S-ACCNO     PIC 9(10) VALUE ZERO.
   05 S-SBA       PIC 9(3)  VALUE ZERO.
   05 S-CNAME     PIC X(20) VALUE SPACES.
   05 S-NOTUSE    PIC X(27) VALUE SPACES.
01 REC-FILED.
   05 R-ACCNO     PIC 9(10) VALUE ZERO.
   05 R-SBA       PIC 9(3)  VALUE ZERO.
```

```
      05 R-CNAME     PIC X(20) VALUE SPACES.
      05 R-NOTUSE    PIC X(27) VALUE SPACES.
   01 RECL           PIC 9(2) COMP VALUE 60.
   01 RECKEY         PIC X(13).
   01 ERR-CODE       PIC S9(8) COMP.
   01 MSG2           PIC X(79) VALUE SPACES.
   01 OUT2           PIC X(79) VALUE SPACES.
   COPY DFHAID.
   COPY BAMPSAF.
   COPY DFHBMSCA.
   LINKAGE SECTION.
   01 DFHCOMMAREA.
      05 STATUS-C  PIC X.
      05 ACCOUN-C  PIC X(10).
      05 CLIENT-C   PIC X(20).
   PROCEDURE DIVISION.
  *** 当任务第一次执行时，直接输出MAP界面，并RETURN到自身 ***
      IF EIBCALEN = 0 THEN
           MOVE 'F' TO STATUS-I
           MOVE LOW-VALUES TO BAMAPAFO
           PERFORM PROCESS-SETMAP1
           EXEC CICS SEND MAP('BAMAPAF')
                         MAPSET('BAMPSAF')
                         CURSOR(593)
                         ERASE
                         FREEKB
                         END-EXEC
           EXEC CICS RETURN TRANSID('BAAF')
                            COMMAREA(COMSTART)
                            LENGTH(31)
                            END-EXEC.
  *** 当用户按<Pause/Break>按键时，输出相应提示信息，并退出该MIS系统 ***
           IF EIBAID = DFHCLEAR THEN
              MOVE 'ADD SESSION ENDED' TO OUT2
              EXEC CICS SEND FROM(OUT2)
                            LENGTH(79)
                            ERASE
                            END-EXEC
              EXEC CICS RETURN END-EXEC.
  *** 当用户按<F2>按键时，返回主菜单 ***
           IF EIBAID = DFHPF2
              EXEC CICS XCTL PROGRAM('BAMSMM')
                            END-EXEC.
  *** 当程序处于由用户输入新增账户的状态时所做的处理 ***
           IF STATUS-C = 'F'
              MOVE LOW-VALUES TO BAMAPAFO
              PERFORM PROCESS-SETMAP1
              IF EIBAID = DFHENTER
                  EXEC CICS RECEIVE MAP('MAMAPAF')
                                    MAPSET('MAMPSAF')
                                    RESP(ERR-CODE)
                                    END-EXEC
                  IF ERR-CODE NOT EQUAL DFHRESP(NORMAL)
                     MOVE ' PROCESSING ERROR ' TO OUT2
                     EXEC CICS SEND FROM(OUT2)
```

```
                        LENGTH(79)
                        ERASE
                        END-EXEC
          EXEC CICS RETURN END-EXEC
       END-IF
       IF ACC-N2I IS EQUAL TO SPACES OR   /*当输入数据为空或空格时进行的处理*/
                   CLI-N2I IS EQUAL TO SPACES OR
                   ACC-N2L IS EQUAL TO ZERO OR
                   CLI-N2L IS EQUAL TO ZERO
          MOVE 'INPUT DATA REQUIRED' TO MSG2
       ELSE IF ACC-N2I IS NOT NUMERIC    /*当输入账号不全为数字时进行的处理*/
          MOVE ' ACCOUNT NUMBER NOT NUMERIC' TO MSG2
          MOVE DFHBMBRY TO ACC-N2A
          MOVE DFHBMBRY TO CLI-N2A
       ELSE                              /*当输入数据合法时进行的处理*/
          MOVE ACC-N2I TO S-ACCNO
          MOVE CLI-N2I TO S-CNAME
          STRING S-ACCNO S-SBA           /*查找文件中是否已存在所要添加的账户*/
             DELIMITED  BY  SIZE INTO RECKEY
          EXEC CICS READ FILE('BAMSFKS') /*查找数据文件中是否存在所输入的账户*/
                      INTO(REC-FILED)
                      RIDFLD(RECKEY)
                      KEYLENGTH(13)
                      LENGTH(RECL)
                      RESP(ERR-CODE)
                      END-EXEC
          IF R-ACCNO = S-ACCNO           /*当文件中存在相应账户时进行的处理*/
               MOVE 'ACCOUNT ALREADY EXISTS' TO MSG2
          ELSE IF                        /*当文件中不存在相应账户时进行的处理*/
               ERR-CODE EQUAL DFHRESP(NOTFND)
               PERFORM PROCESS-SETMAP2
               EXEC CICS SEND MAP('BAMAPAF')
                           MAPSET('BAMPSAF')
                           CURSOR(1587)
                           ERASE
                           FREEKB
                           END-EXEC
               MOVE 'S' TO STATUS-I      /*转入确认/撤销操作的状态*/
               MOVE ACC-N2I TO ACCOUN-I
               MOVE CLI-N2I TO CLIENT-I
               EXEC CICS RETURN TRANSID('BAAF')
                            COMMAREA(COMSTART)
                            LENGTH(31)
                            END-EXEC
               END-IF                    /*结束文件中不存在相应账户时的处理*/
          END-IF                         /*结束输入数据非空时的处理*/
             END-IF                      /*结束对输入数据进行判断的处理*/
       END-IF                            /*结束用户按" Enter" 按键时的处理*/
ELSE                                     /*当用户按无效按键时进行的处理*/
    MOVE 'INVALID KEY PRESSED' TO MSG2
END-IF
MOVE MSG2 TO DISPL-2O                    /*输出以上处理完成后的界面信息*/
EXEC CICS SEND MAP('BAMAPAF')
              MAPSET('BAMPSAF')
              CURSOR(593)
```

```
                        ERASE
                        FREEKB
                        END-EXEC
        END-IF.
*** 当程序处于由用户确认/撤销操作状态时所做的操作 ***
        IF STATUS-C = 'S'
           PERFORM PROCESS-SETMAP2
           MOVE 'S' TO STATUS-I
           IF EIBAID = DFHENTER
               EXEC CICS RECEIVE MAP('BAMAPAF')
                                 MAPSET('BAMPSAF')
                                 RESP(ERR-CODE)
                                 END-EXEC
               IF ERR-CODE NOT EQUAL DFHRESP(NORMAL)
                  MOVE ' PROCESSING ERROR ' TO OUT2
                  EXEC CICS SEND FROM(OUT2)
                                 LENGTH(79)
                                 ERASE
                                 END-EXEC
               EXEC CICS RETURN END-EXEC
               END-IF
               PERFORM PROCESS-SETMAP2
               EVALUATE TRUE
                 WHEN UPDATE-2I IS EQUAL TO SPACES OR
                                 UPDATE-2L  IS  EQUAL TO ZERO
                   MOVE 'INPUT DATA REQUIRED' TO MSG2
                 WHEN UPDATE-2I = 'Y'                    /*用户确认添加账户操作*/
                   MOVE ACCOUN-C  TO S-ACCNO
                   MOVE CLIENT-C    TO S-CNAME
                   STRING S-ACCNO S-SBA
                      DELIMITED BY SIZE INTO RECKEY
                   EXEC CICS WRITE FROM(DATA-FILED)  /*将输入的账户信息添加到数据文件中*/
                                 FILE('BAMSFKS')
                                 RIDFLD(RECKEY)
                                 LENGTH(RECL)
                                 KEYLENGTH(13)
                                 END-EXEC
                   MOVE 'ACCOUNT ADDED TO FILE' TO MSG2
                   PERFORM PROCESS-SETMAP3
                   MOVE 'T' TO STATUS-I                  /*转入添加下一个账户的状态*/
                 WHEN UPDATE-2I = 'N'                    /*用户撤销添加账户操作*/
                   MOVE 'UPDATE PROCESS CANCELED' TO MSG2
                   PERFORM PROCESS-SETMAP3
                   MOVE 'T' TO STATUS-I                  /*转入添加下一个账户的状态*/
                 WHEN OTHER
                   MOVE 'ENTER "Y" OR "N" ' TO MSG2
               END-EVALUATE
           ELSE  MOVE 'INVALID KEY PRESSED' TO MSG2
           END-IF
           MOVE ACCOUN-C TO ACCOUN-I
           MOVE CLIENT-C TO CLIENT-I
           MOVE ACCOUN-C TO ACC-N2O
           MOVE CLIENT-C TO CLI-N2O
           MOVE MSG2 TO DISPL-2O
           EXEC CICS SEND MAP('BAMAPAF')
```

```
                          MAPSET('BAMPSAF')
                          CURSOR(1587)
                          ERASE
                          FREEKB
                          END-EXEC
        END-IF.
*** 当程序处于由用户选择添加下一个账户状态时所做的操作 ***
        IF STATUS-C = 'T'
           PERFORM PROCESS-SETMAP3
           IF EIBAID = DFHPF9
              MOVE  'F' TO STATUS-I                 /*转入由用户输入新增账户的状态*/
              PERFORM PROCESS-SETMAP1
              EXEC CICS SEND MAP('BAMAPAF')
                           MAPSET('BAMPSAF')
                           CURSOR(593)
                           ERASE
                           FREEKB
                           END-EXEC
           ELSE
              MOVE 'S' TO STATUS-I                  /*转入由用户确认/撤销操作的状态*/
              MOVE 'INVALID KEY PRESSED' TO DISPL-2O
              MOVE ACCOUN-C TO ACCOUN-I
              MOVE CLIENT-C TO CLIENT-I
              PERFORM PROCESS-SETMAP2
              EXEC CICS SEND MAP('BAMAPAF')
                           MAPSET('BAMPSAF')
                           CURSOR(1587)
                           ERASE
                           FREEKB
                           END-EXEC
           END-IF
        END-IF.
*** 各种状态操作结束后，RETURN到自身 ***
        EXEC CICS RETURN TRANSID('BAAF')
                         COMMAREA(COMSTART)
                         LENGTH(31)
                         END-EXEC.
          GOBACK.
*** 进行各项关于MAP变量输出属性的设置 ***
 PROCESS-SETMAP1.
        MOVE DFHBMDAR TO PROCE-2A
        MOVE DFHBMDAR TO NEXT-2A
        MOVE DFHBMPRO TO UPDATE-2A
        MOVE DFHBMDAR TO UPDATE-2A
        MOVE DFHBMUNP TO ACC-N2A
        MOVE DFHBMUNP TO CLI-N2A
        MOVE DFHBMPRO TO ENTER-2A.
 PROCESS-SETMAP2.
        MOVE DFHBMBRY TO PROCE-2A
        MOVE DFHBMPRO TO ACC-N2A
        MOVE DFHBMPRO TO CLI-N2A
        MOVE DFHBMPNL TO UPDATE-2A
        MOVE DFHBMUNP TO UPDATE-2A
        MOVE DFHBMDAR TO NEXT-2A
        MOVE DFHBMPRO TO ENTER-2A.
```

```
PROCESS-SETMAP3.
       PERFORM PROCESS-SETMAP2
       MOVE DFHBMDAR TO ENTER-2A
       MOVE DFHBMBRY TO NEXT-2A.
```

18.3 删除账户功能模块

在删除账户功能模块的界面上，用户可以通过输入所要删除的账户账号来对账户进行删除。系统会根据所输入的账号将相应的账户姓名显示出来。用户同样可确认或撤销删除操作、连续删除账户、以及返回主菜单。删除账户功能模块的界面如图 18.3 所示。

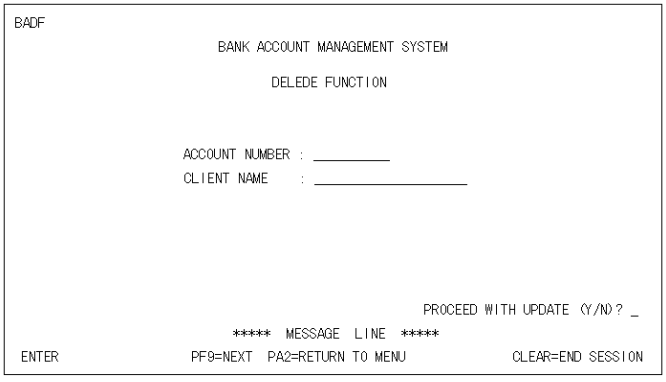

图 18.3 删除账户功能模块界面

在添加账户功能模块中，系统可输出的提示信息有以下几条。

- 提示信息 1：INVALID KEY PRESSED（表示用户按了非法的按键）。
- 提示信息 2：INPUT DATA REQUIRED（表示用户输入数据为空）。
- 提示信息 3：ACCOUNT NUMBER NOT NUMERIC（表示输入的账号不全为数字）。
- 提示信息 4：ACCOUNT NOT EXIST（表示所要删除的账户不存在）。
- 提示信息 5：ACCOUNT DELETED FROM FILE（表示将账户从数据文件中删除成功）。
- 提示信息 6：UPDATE PROCESS CANCELED（表示用户撤销了删除账户操作）。
- 提示信息 7：ENTER “Y” OR “N”（要求用户输入确认或撤销信息）。
- 提示信息 8：PROCESSING ERROR（表示系统在处理过程中发生错误）。
- 提示信息 9：DELETE SESSION ENDED（表示退出系统）。

删除账户功能模块的任务要求如下。

（1）显示界面，允许用户输入账号信息。但“PROCEED WITH UPDATE (Y/N)? _”、“PF9=NEXT”以及姓名信息位置不显示。并且以上各处下划线位置不允许输入。

（2）根据用户在输入数据前后所按的不同功能按键，分别作出如下处理。

- “Enter”按键：执行步骤 3。
- Pause/Break 按键：输出提示信息 9，同时退出该系统。
- “PA2（F2）”按键：返回主菜单。
- 其他按键：输出提示信息 1，重新执行步骤 2。

（3）根据用户输入的账号信息，分别作出如下处理。

- 如果没有输入或输入的为空格，则输出提示信息 2。同时账号信息高亮显示，不允许输入，再返回执行步骤 2。
- 如果输入的账号不全为数字，则输出提示信息 3。同时账号信息高亮显示，不允许输入，再返回执行步骤 2。
- 判断账号信息，如果在数据文件中不存在该账号，则输出提示信息 4。同时将账号信息高亮显示，不允许输入，再返回执行步骤 2。
- 如果数据文件中存在该账号，则执行步骤 4。

（4）重新输出界面。此时将“PROCEED WITH UPDATE (Y/N)? _”高亮显示，并在下划线处允许输入。同时显示与输入的账号对应的账户姓名，且二者所在位置都不允许输入。

（5）根据用户在输入数据前后所按的不同功能按键，分别作出如下处理。

- “Enter”按键：执行步骤 6。
- Pause/Break 按键：输出提示信息 9，同时退出该系统。
- “PA2（F2）”按键：返回主菜单。
- 其他按键：输出提示信息 1，重新执行步骤 5。

（6）根据用户输入的确认或撤销信息，分别作出如下处理。

- 如果没有输入或输入的为空格，则输出提示信息 2，返回执行步骤 5。
- 如果输入的为“Y”，则将相应账户信息从数据文件中删除，输出提示信息 5，执行步骤 7。
- 如果输入的为“N”，则输出提示信息 6，执行步骤 7。
- 如果输入的为其他字符，则输出提示信息 7，返回执行步骤 5。

（7）高亮显示“PF9=NEXT”，同时不显示”Enter”。此时用户按下快捷键 F9，则返回执行步骤 1；若按其他按键，包括 Enter 键，则输出提示信息 1，返回执行步骤 4。

（8）如果在处理过程中遇到任何异常，则输出提示信息 8，同时退出该系统。

设该模块所用到的除文件 BAMSFKS 以外的各项资源名称分别如下。

- 交易名称：BADF。
- 程序名称：BAMSDF。
- 界面所对应的 MAP 名称：BAMAPDF。
- MAP 所在的 MAPSET 名称：BAMPSDF。

同时，令本模块中符号 MAP 里的各变量如下。

- ACC-N3：对应账号信息部分。
- CLI-N3：对应姓名信息部分。
- PROCE-3：对应“PROCEED WITH UPDATE (Y/N)?”字段。
- UPDATE-3：对应“PROCEED WITH UPDATE (Y/N)? _”中的下划线。
- NEXT-3：对应“PF9=NEXT”字段。
- ENTER-3：对应“ENTER”字段。
- DISPL-3：对应系统信息输出部分。

则实现该模块所有功能的完整程序代码如下。

```
IDENTIFICATION DIVISION.
PROGRAM-ID BAMSDF.
```

```
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 COMSTART.
   05 STATUS-I    PIC X VALUE 'F'.
   05 ACCOUN-I    PIC X(10) VALUE SPACES.
   05 CLIENT-I    PIC X(20) VALUE SPACES.
01 DATA-FILED.
   05 S-ACCNO     PIC 9(10) VALUE ZERO.
   05 S-SBA       PIC 9(3)  VALUE ZERO.
   05 S-CNAME     PIC X(20) VALUE SPACES.
   05 S-NOTUSE    PIC X(27)  VALUE SPACES.
01 REC-FILED.
   05 R-ACCNO     PIC 9(10) VALUE ZERO.
   05 R-SBA       PIC 9(3)  VALUE ZERO.
   05 R-CNAME     PIC X(20) VALUE SPACES.
   05 R-NOTUSE    PIC X(27)  VALUE SPACES.
01 RECL           PIC 9(2) COMP VALUE 60.
01 RECKEY         PIC X(13).
01 ERR-CODE       PIC S9(8) COMP.
01 MSG3           PIC X(79) VALUE SPACES.
01 OUT3           PIC X(79) VALUE SPACES.
COPY DFHAID.
COPY BAMPSDF.
COPY DFHBMSCA.
LINKAGE SECTION.
01 DFHCOMMAREA.
   05 STATUS-C    PIC X.
   05 ACCOUN-C    PIC X(10).
   05 CLIENT-C    PIC X(20).
PROCEDURE DIVISION.
*** 当任务第一次执行时，直接输出 MAP 界面，并 RETURN 到自身 ***
  IF EIBCALEN = 0 THEN
         MOVE 'F' TO STATUS-I
         MOVE LOW-VALUES TO BAMAPDFO
         PERFORM PROCESS-SETMAP1
            EXEC CICS SEND MAP('BAMAPDF')
                           MAPSET('BAMPSDF')
                           CURSOR(588)
                           ERASE
                           FREEKB
                           END-EXEC
         EXEC CICS RETURN TRANSID('BADF')
                          COMMAREA(COMSTART)
                          LENGTH(31)
                          END-EXEC.
*** 当用户按 Pause/Break 按键时，输出相应提示信息，并退出该 MIS 系统 ***
      IF EIBAID = DFHCLEAR THEN
         MOVE 'DELETE SESSION ENDED' TO OUT3
         EXEC CICS SEND FROM(OUT3)
                        LENGTH(79)
                        ERASE
                        END-EXEC
         EXEC CICS RETURN END-EXEC.
*** 当用户按 “F2” 按键时，返回主菜单 ***
```

```
      IF EIBAID = DFHPF2
         EXEC CICS XCTL PROGRAM('BAMSMM')
                     END-EXEC.
*** 当程序处于由用户选择账户进行删除的状态时所做的处理 ***
      IF STATUS-C = 'F'
         MOVE LOW-VALUES TO BAMAPDFO
         PERFORM PROCESS-SETMAP1
         IF EIBAID = DFHENTER
            EXEC CICS RECEIVE MAP('BAMAPDF')
                           MAPSET('BAMPSDF')
                           RESP(ERR-CODE)
                           END-EXEC
            IF ERR-CODE NOT EQUAL DFHRESP(NORMAL)
                MOVE ' PROCESSING ERROR ' TO OUT3
                EXEC CICS SEND FROM(OUT3)
                               LENGTH(79)
                               ERASE
                               END-EXEC
              EXEC CICS RETURN END-EXEC
            END-IF
            IF ACC-N3I  IS  EQUAL TO SPACES  OR
                         ACC-N3L  IS  EQUAL TO ZERO
                MOVE 'INPUT DATA REQUIRED' TO MSG3
            ELSE IF ACC-N3I  IS  NOT NUMERIC
                 MOVE ' ACCOUNT NUMBER NOT NUMERIC' TO MSG3
                 MOVE DFHBMBRY TO ACC-N3A
                 ELSE
                    MOVE ACC-N3I TO S-ACCNO
                    STRING S-ACCNO S-SBA
                       DELIMITED BY SIZE INTO RECKEY
                    EXEC CICS READ FILE('BAMSFKS')
                                  INTO(REC-FILED)
                                  RIDFLD(RECKEY)
                                  KEYLENGTH(13)
                                  LENGTH(RECL)
                                  RESP(ERR-CODE)
                                  END-EXEC
                    IF ERR-CODE EQUAL DFHRESP(NOTFND)
                       MOVE 'ACCOUNT DOSE NOT EXIST' TO MSG3
                    ELSE IF  R-ACCNO = S-ACCNO
                         PERFORM PROCESS-SETMAP2
                         MOVE R-ACCNO TO ACC-N3O
                         MOVE R-CNAME TO CLI-N3O
                         EXEC CICS SEND MAP('BAMAPDF')
                                       MAPSET('BAMPSDF')
                                       CURSOR(1587)
                                       ERASE
                                       FREEKB
                                       END-EXEC
                         MOVE 'S' TO STATUS-I
                         MOVE ACC-N3I TO ACC-I
                         EXEC CICS RETURN TRANSID('BADF')
                                         COMMAREA(COMSTART)
                                         LENGTH(31)
                                         END-EXEC
```

```
                        END-IF
                     END-IF
                  END-IF
               END-IF
            ELSE
                MOVE 'INVALID KEY PRESSED' TO MSG3
            END-IF
            MOVE 'F' TO STATUS-I
         END-IF.
*** 当程序处于由用户确认/撤销操作状态时所做的操作 ***
         IF STATUS-C = 'S'
            PERFORM PROCESS-SETMAP2
            MOVE 'S' TO STATUS-I
            IF EIBAID = DFHENTER
               EXEC CICS RECEIVE MAP('BAMAPDF')
                                 MAPSET('BAMPSDF')
                                 RESP(ERR-CODE)
                                 END-EXEC
               IF ERR-CODE NOT EQUAL DFHRESP(NORMAL)
                   MOVE ' PROCESSING ERROR ' TO OUT3
                   EXEC CICS SEND FROM(OUT3)
                                  LENGTH(79)
                                  ERASE
                                  END-EXEC
                   EXEC CICS RETURN END-EXEC
               END-IF
               PERFORM PROCESS-SETMAP2
                   EVALUATE TRUE
                     WHEN UPDATE-3I IS EQUAL TO SPACES  OR
                                    UPDATE-3L IS EQUAL TO ZERO
                       MOVE 'INPUT DATA REQUIRED' TO MSG3
                     WHEN UPDATE-3I = 'Y'
                       MOVE ACCOUN-C TO S-ACCNO
                       STRING S-ACCNO S-SBA
                          DELIMITED BY SIZE INTO RECKEY
                       EXEC CICS DELETE FILE('BAMSFKS')
                                        RIDFLD(RECKEY)
                                        KEYLENGTH(13)
                                        END-EXEC
                       MOVE 'ACCOUNT DELETED FROM FILE' TO MSG3
                       PERFORM PROCESS-SETMAP3
                       MOVE 'T' TO STATUS-I
                     WHEN UPDATE-3I = 'N'
                       MOVE 'UPDATE PROCESS CANCELED' TO MSG3
                       PERFORM PROCESS-SETMAP3
                       MOVE 'T' TO STATUS-I
                     WHEN OTHER
                       MOVE 'ENTER "Y" OR "N" ' TO MSG3
                     END-EVALUATE
            ELSE
               MOVE 'INVALID KEY PRESSED' TO MSG3
            END-IF
               MOVE ACCOUN-C TO ACCOUN-I
               MOVE CLIENT-C TO CLIENT-I
               MOVE ACCOUN-C TO ACC-N3O
```

```
            MOVE CLIENT-C TO CLI-N3O
      END-IF
*** 当程序处于由用户选择删除下一个账户状态时所做的操作 ***
      IF STATUS-C = 'T'
         PERFORM PROCESS-SETMAP3
         IF EIBAID = DFHPF9
            MOVE 'F' TO STATUS-I
            PERFORM PROCESS-SETMAP1
            EXEC CICS SEND MAP('BAMAPDF')
                           MAPSET('BAMPSDF')
                           CURSOR(588)
                           ERASE
                           FREEKB
                           END-EXEC
            EXEC CICS RETURN TRANSID('BADF')
                             COMMAREA(COMSTART)
                             LENGTH(31)
                             END-EXEC
         ELSE
            MOVE 'S' TO STATUS-I
            MOVE 'INVALID KEY PRESSED' TO MSG3
            MOVE ACCOUN-C TO ACCOUN-I
            MOVE ACCOUN-C TO ACC-N3O
            MOVE CLIENT-C TO CLIENT-I
            MOVE CLIIENT-C TO CLI-N3O
         END-IF
      END-IF
*** 各种状态操作结束后，输出界面，并 RETURN 到自身 ***
      MOVE MSG3 TO DISPL-3O
      EXEC CICS SEND MAP('BAMAPDF')
                     MAPSET('BAMPSDF')
                     CURSOR(588)
                     ERASE
                     FREEKB
                     END-EXEC
      EXEC CICS RETURN TRANSID('BADF')
                       COMMAREA(COMSTART)
                       LENGTH(31)
                       END-EXEC.
               GOBACK.
*** 进行各项关于 MAP 变量输出属性的设置 ***
 PROCESS-SETMAP1.
         MOVE DFHBMCUR TO ACC-N3A
         MOVE DFHBMDAR TO PROCE-3A
         MOVE DFHBMDAR TO NEXT-3A
         MOVE DFHBMPRO TO UPDATE-3A
         MOVE DFHBMDAR TO UPDATE-3A
         MOVE DFHBMUNP TO ACC-N3A
         MOVE DFHBMUNP TO CLI-N3A
         MOVE DFHBMPRO TO ENTER-3A.
 PROCESS-SETMAP2.
         MOVE DFHBMBRY TO PROCE-3A
         MOVE DFHBMPRO TO ACC-N3A
         MOVE DFHBMPRO TO CLI-N3A
         MOVE DFHBMUNP TO UPDATE-3A
```

```
            MOVE DFHBMPRO TO ENTER-3A.
     PROCESS-SETMAP3.
            PERFORM PROCESS-SETMAP2
            MOVE DFHBMDAR TO ENTER-3A
            MOVE DFHBMBRY TO NEXT-3A.
```

18.4 修改账户功能模块

在修改账户功能模块的界面上，用户可以通过输入账号对所需修改的账户信息进行定位。同时，输入的账号可以为部分内容，以进行模糊查找。修改的对象为账户姓名。用户同样可以确认或撤销修改操作、连续修改账户以及返回主菜单。修改账户功能模块的界面如图18.4 所示。

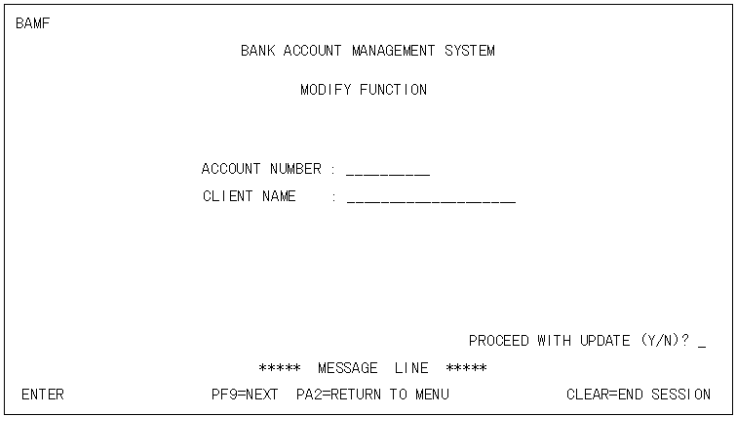

图 18.4 修改账户功能模块界面

在添加账户功能模块中，系统可输出的提示信息有以下几条。

- 提示信息 1：INVALID KEY PRESSED（表示用户按了非法的按键）。
- 提示信息 2：INPUT DATA REQUIRED（表示用户输入数据为空）。
- 提示信息 3：ACCOUNT NUMBER NOT NUMERIC（表示输入的账号不全为数字）。
- 提示信息 4：ACCOUNT NOT EXIST（表示所要修改的账户不存在）。
- 提示信息 5：CLIENT NAME MODIFIED（表示账户姓名修改成功）。
- 提示信息 6：UPDATE PROCESS CANCELED（表示用户撤销了修改账户操作）。
- 提示信息 7：ENTER “Y” OR “N”（要求用户输入确认或撤销信息）。
- 提示信息 8：PROCESSING ERROR（表示系统在处理过程中发生错误）。
- 提示信息 9：MODIFY SESSION ENDED（表示退出系统）。

修改账户功能模块的任务要求如下。

（1）显示界面，输出提示信息 3，允许用户输入账号或姓名信息。但“PROCEED WITH UPDATE (Y/N)？_”和“PF9=NEXT” 不显示，且下划线位置不允许输入。

（2）根据用户在输入数据前后所按的不同功能按键，分别作出如下处理。

- “Enter”按键：执行步骤 3。
- Pause/Break 按键：输出提示信息 9，同时退出该系统。

- “PA2（F2）”按键：返回主菜单。
- 其他按键：输出提示信息 1，重新执行步骤 2。

（3）根据用户输入的账号信息，分别作出如下处理。

- 如果没有输入或输入的为空格，则输出提示信息 2。同时将光标定位到账号信息输入首位，返回执行步骤 2。
- 如果输入的账号不全为数字，则输出提示信息 3。同时账号信息高亮显示，不允许输入，再返回执行步骤 2。
- 如果输入了完整的账号（长度为 10），则判断相应账户是否存在。如果存在，执行步骤 5；如果不存在，输出提示信息 4，光标定位到账号信息，高亮显示，允许输入，返回执行步骤 2。
- 如果输入的账号信息不完全（长度小于 10），则进行模糊查找，执行步骤 4。

（4）将所有满足条件的记录存放在临时存储队列中，队列名由 EIBTRMID 和 EIBTRNID 组成。将队列名作为其中一项参数传递给查询账户功能模块，并执行该模块。当查询账户功能模块返回本模块时，执行步骤 5。

（5）重新输出界面。此时账号信息部分不允许输入。姓名信息部分则输出相应的姓名，并允许在此进行改写。

（6）根据用户在输入改写的姓名数据前后所按的不同功能按键，分别作出如下处理。

- “Enter”按键：执行步骤 7。
- Pause/Break 按键：输出提示信息 9，同时退出该系统。
- “PA2（F2）”按键：返回主菜单。
- 其他按键：输出提示信息 1，重新执行步骤 6。

（7）重新输出界面。此时将“PROCEED WITH UPDATE (Y/N)? _”高亮显示，并在下划线处允许输入。账号和姓名部分同步骤 5 执行之后的内容一致。根据用户输入的确认或撤销信息，分别作出如下处理。

- 如果没有输入或输入的为空格，则输出提示信息 2，返回执行步骤 6。
- 如果输入的为“Y”，则改写数据记录，输出提示信息 5，执行步骤 8。
- 如果输入的为“N”，则输出提示信息 6，执行步骤 8。
- 如果输入的为其他字符，则输出提示信息 7，返回执行步骤 6。

（8）高亮显示“PF9=NEXT”，同时不显示“ENTER”。此时用户按下快捷键 F9，则返回执行步骤 1；若按其他按键，包括 ENTER 键，则输出提示信息 1，返回执行步骤 5。

（9）如果在处理过程中遇到任何异常，则输出提示信息 8，同时退出该系统。

不妨设该模块所用到的除文件 BAMSFKS 以外的各项资源名称分别如下。

- 交易名称：BAMF。
- 程序名称：BAMSMF。
- 界面所对应的 MAP 名称：BAMAPMF。
- MAP 所在的 MAPSET 名称：BAMPSMF。
- 临时存储队列名称：由 EIBTRMID 和 EIBTRNID 所组成。

同时，令本模块中符号 MAP 里的各变量如下。

- ACC-N4：对应账号信息部分。

- CLI-N4：对应姓名信息部分。
- PROCE-4：对应“PROCEED WITH UPDATE (Y/N)?”字段。
- UPDATE-4：对应“PROCEED WITH UPDATE (Y/N)? _”中的下划线。
- NEXT-4：对应“PF9=NEXT”字段。
- ENTER-4：对应“ENTER”字段。
- DISPL-4：对应系统信息输出部分。

则实现该模块所有功能的完整程序代码如下。

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID BAMSMF.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 COMSTART.
    05 STATUS-I    PIC X VALUE 'F'.
    05 ACCOUN-I    PIC X(10) VALUE SPACES.
    05 CLIENT-I    PIC X(20) VALUE SPACES.
 01 DATA-FILED.
    05 S-ACCNO     PIC 9(10) VALUE ZERO.
    05 S-SBA       PIC 9(3)  VALUE ZERO.
    05 S-CNAME     PIC X(20) VALUE SPACES.
    05 S-NOTUSE    PIC X(27)  VALUE SPACES.
 01 REC-FILED.
    05 R-ACCNO     PIC 9(10) VALUE ZERO.
    05 R-SBA       PIC 9(3) VALUE ZERO.
    05 R-CNAME     PIC X(20) VALUE SPACES.
    05 R-NOTUSE    PIC X(27) VALUE SPACES.
 01 RECL           PIC 9(2) COMP VALUE 60.
 01 RECKEY         PIC X(13).
 01 ERR-CODE       PIC S9(8) COMP.
 01 Q-NAME         PIC X(8).
 01 MSG4           PIC X(79) VALUE SPACES.
 01 OUT4           PIC X(79) VALUE SPACES.
 01 I-COUNT        PIC S9(4) COMP.
 COPY DFHAID.
 COPY BAMPSMF.
 COPY DFHBMSCA.
 LINKAGE SECTION.
 01 DFHCOMMAREA.
    05 STATUS-C    PIC X.
    05 ACCOUN-C    PIC X(10).
    05 CLIENT-C    PIC X(20).
 PROCEDURE DIVISION.
*** 当任务第一次执行时，直接输出 MAP 界面，并 RETURN 到自身 ***
       IF EIBCALEN = 0 THEN
         MOVE 'F' TO STATUS-I
         MOVE LOW-VALUES TO BAMAPMFO
         PERFORM PROCESS-SETMAP1
         MOVE 'ENTER ACCOUNT NUMBER' TO DISPL-4O
         EXEC CICS SEND MAP('BAMAPMF')
                        MAPSET('BAMPSMF')
                        CURSOR(589)
```

```
                        ERASE
                        FREEKB
                        END-EXEC
          EXEC CICS RETURN TRANSID('BAMF')
                          COMMAREA(COMSTART)
                          LENGTH(31)
                          END-EXEC.
*** 当用户按<Pause/Break>按键时，输出相应提示信息，并退出该 MIS 系统 ***
        IF EIBAID = DFHCLEAR THEN
          MOVE 'MODIFY SESSION ENDED' TO OUT4
          EXEC CICS SEND FROM(OUT4)
                        LENGTH(79)
                        ERASE
                        END-EXEC
          EXEC CICS RETURN END-EXEC.
*** 当用户按<F2>按键时，返回主菜单 ***
        IF EIBAID = DFHPF2
          EXEC CICS XCTL PROGRAM('BAMSMM')
                        END-EXEC.
*** 当程序处于由用户选择账户进行修改的状态时所做的处理 ***
        IF STATUS-C = 'F'
          MOVE LOW-VALUES TO BAMAPMFO
          PERFORM PROCESS-SETMAP1
          IF EIBAID = DFHENTER
             EXEC CICS RECEIVE MAP('BAMAPMF')
                               MAPSET('BAMPSMF')
                               RESP(ERR-CODE)
                               END-EXEC
             IF ERR-CODE NOT EQUAL DFHRESP(NORMAL)
                  MOVE ' PROCESSING ERROR ' TO OUT4
                  EXEC CICS SEND FROM(OUT4)
                                 LENGTH(79)
                                 ERASE
                                 END-EXEC
                  EXEC CICS RETURN END-EXEC
              END-IF
              IF ACC-N4I  IS  EQUAL TO SPACES OR
                            ACC-N4L  IS  EQUAL TO ZERO
                  MOVE 'INPUT DATA REQUIRED' TO MSG4
             ELSE IF ACC-N4I IS NOT NUMERIC
                  MOVE 'ACCOUNT NUMBER NOT NUMERIC' TO MSG4
                  MOVE DFHBMBRY TO ACC-N4A
                  ELSE
                     IF ACC-N4L IS NOT EQUAL 10
                        PERFORM PROCESS-TO-LIST
                     END-IF
                     IF ACC-N4L  IS  EQUAL 10
                       MOVE ACC-N4I  TO  S-ACCNO
                       STRING S-ACCNO S-SBA
                         DELIMITED BY SIZE INTO RECKEY
                         EXEC CICS READ FILE('BAMSFKS')
                                        INTO(REC-FILED)
                                        RIDFLD(RECKEY)
                                        KEYLENGTH(13)
                                        LENGTH(RECL)
```

```
                                    RESP(ERR-CODE)
                                    END-EXEC
                        IF ERR-CODE EQUAL DFHRESP(NOTFND)
                            MOVE 'ACCOUNT NOT EXIST' TO MSG4
                        ELSE IF R-ACCNO = S-ACCNO
                               MOVE R-ACCNO TO ACC-N4O
                               MOVE R-CNAME TO CLI-N4O
                               MOVE DFHBMPRO TO ACC-N4A
                               MOVE DFHBMUNP TO CLI-N4A
                               EXEC CICS SEND MAP('BAMAPMF')
                                             MAPSET('BAMPSMF')
                                             CURSOR(749)
                                             ERASE
                                             FREEKB
                                             END-EXEC
                               MOVE 'R' TO STATUS-I
                               MOVE ACC-N4I  TO ACCOUN-I
                               MOVE CLI-N4I   TO CLIENT-I
                               EXEC CICS RETURN TRANSID('BAMF')
                                              COMMAREA(COMSTART)
                                              LENGTH(31)
                                              END-EXEC
                             END-IF
                        END-IF
                    END-IF
                END-IF
            END-IF
          ELSE
              MOVE 'INVALID KEY PRESSED' TO MSG4
          END-IF
        MOVE 'F' TO STATUS-I
        MOVE MSG4 TO DISPL-4O
        EXEC CICS SEND MAP('BAMAPMF')
                       MAPSET('BAMPSMF')
                       CURSOR(593)
                       ERASE
                       FREEKB
                       END-EXEC
        END-IF.
*** 当程序处于由用户输入修改后的账户姓名的状态时所做的处理 ***
        IF STATUS-C = 'R'
           EXEC CICS RECEIVE MAP('BAMAPMF')
                             MAPSET('BAMPSMF')
                             END-EXEC
           MOVE CLI-N4I TO CLIENT-I
           MOVE ACCOUN-C TO ACCOUN-I
           PERFORM PROCESS-SETMAP2
           MOVE ACCOUN-C TO ACC-N4O
           MOVE DFHBMBRY TO PROCE-4A
           EXEC CICS SEND MAP('BAMAPMF')
                          MAPSET('BAMPSMF')
                          CURSOR(1597)
                          ERASE
                          FREEKB
                          END-EXEC
```

```
           MOVE 'S' TO STATUS-I
           EXEC CICS RETURN TRANSID('BAMF')
                          COMMAREA(COMSTART)
                          LENGTH(31)
                          END-EXEC
       END-IF.
*** 当程序处于由用户确认/撤销操作状态时所做的操作 ***
       IF STATUS-C = 'S'
           PERFORM PROCESS-SETMAP2
           MOVE 'S' TO STATUS-I
           IF EIBAID = DFHENTER
              EXEC CICS RECEIVE MAP('BAMAPMF')
                              MAPSET('BAMPSMF')
                              END-EXEC
              PERFORM PROCESS-SETMAP2
              EVALUATE TRUE
                   WHEN UPDATE-4I IS EQUAL TO SPACES  OR
                                   UPDATE-4L IS EQUAL TO ZERO
                     MOVE 'INPUT DATA REQUIRED' TO MSG4
                   WHEN UPDATE-4I = 'Y'
                     MOVE ACCOUN-C TO S-ACCNO
                     MOVE CLIENT-C  TO S-CNAME
                     STRING S-ACCNO S-SBA
                        DELIMITED BY SIZE INTO RECKEY
                     EXEC CICS READ FILE('BAMSFKS')
                                   INTO(REC-FILED)
                                   RIDFLD(RECKEY)
                                   KEYLENGTH(13)
                                   LENGTH(RECL)
                                   UPDATE
                                   RESP(ERR-CODE)
                                   END-EXEC
                     EXEC CICS REWRITE FILE('BAMSFKS')
                                      FROM(DATA-FILED)
                                      LENGTH(RECL)
                                      NOHANDLE
                                      END-EXEC
                     MOVE 'CLIENT NAME MODIFIED' TO MSG4
                     PERFORM PROCESS-SETMAP3
                     MOVE 'T' TO STATUS-I
                   WHEN UPDATE-4I = 'N'
                     MOVE 'UPDATE PROCESS DISCONTINUED' TO MSG4
                     PERFORM PROCESS-SETMAP3
                     MOVE 'T' TO STATUS-I
                   WHEN OTHER
                     MOVE 'ENTER "Y" OR "N" ' TO MSG4
              END-EVALUATE
           ELSE
              MOVE 'INVALID KEY PRESSED' TO MSG4
           END-IF
           MOVE ACCOUN-C TO ACCOUN-I
           MOVE CLIENT-C TO CLIENT-I
           MOVE ACCOUN-C TO ACC-N4O
           MOVE CLIENT-C TO CLI-N4O
           MOVE MSG4 TO DISPL-4O
```

```
          EXEC CICS SEND MAP('BAMAPMF')
                         MAPSET('BAMPSMF')
                         CURSOR(1597)
                         ERASE
                         FREEKB
                         END-EXEC
       END-IF.
*** 当程序处于由用户选择修改下一个账户状态时所做的操作 ***
       IF STATUS-C = 'T'
          PERFORM PROCESS-SETMAP3
          IF EIBAID = DFHPF9
          MOVE  'F' TO STATUS-I
          PERFORM PROCESS-SETMAP1
          EXEC CICS SEND MAP('BAMAPMF')
                         MAPSET('BAMPSMF')
                         CURSOR(589)
                         ERASE
                         FREEKB
                         END-EXEC
          ELSE
             MOVE 'S' TO STATUS-I
             MOVE 'INVALID KEY PRESSED' TO DISPL-4O
             MOVE ACCOUN-C TO ACCOUN-I
             MOVE CLIENT-C TO CLIENT-I
             PERFORM PROCESS-SETMAP2
             EXEC CICS SEND MAP('BAMAPMF')
                            MAPSET('BAMPSMF')
                            CURSOR(1597)
                            ERASE
                            FREEKB
                            END-EXEC
        END-IF
       END-IF.
*** 各种状态操作结束后，RETURN到自身 ***
       EXEC CICS RETURN TRANSID('BAMF')
                        COMMAREA(COMSTART)
                        LENGTH(31)
                        END-EXEC.
              GOBACK.
*** 进行各项关于MAP变量输出属性的设置 ***
PROCESS-SETMAP1.
            MOVE DFHBMDAR TO PROCE-4A
            MOVE DFHBMDAR TO NEXT-4A
            MOVE DFHBMPRO TO UPDATE-4A
            MOVE DFHBMDAR TO UPDATE-4A
            MOVE DFHBMUNP TO ACC-N4A
            MOVE DFHBMUNP TO CLI-N4A
            MOVE DFHBMPRO TO ENTER-4A..
 PROCESS-SETMAP2.
            MOVE DFHBMBRY TO PROCE-4A
            MOVE DFHBMPRO TO ACC-N4A
            MOVE DFHBMPRO TO CLI-N4A
            MOVE DFHBMUNP TO UPDATE-4A
            MOVE DFHBMDAR TO NEXT-4A
            MOVE DFHBMPRO TO ENTER-4A.
```

```
 PROCESS-SETMAP3.
           PERFORM PROCESS-SETMAP2
           MOVE DFHBMDAR TO ENTER-4A
           MOVE DFHBMBRY TO NEXT-4A.
*** 进行模糊查找时的处理 ***
 PROCESS-TO-LIST.
        MOVE ACC-N4I TO RECKEY
        MOVE 1 TO I-COUNT
        STRING EIBTRMID EIBTRNID
           DELIMITED BY SIZE INTO Q-NAME
        EXEC CICS READ FILE('BAMSFKS')
                       INTO(REC-FILED)
                       LENGTH(RECL)
                       RIDFLD(RECKEY)
                       KEYLENGTH(13)
                       GTEQ
                       END-EXEC
        EXEC CICS STARTBR FILE('BAMSFKS')
                       RIDFLD(RECKEY)
                       GTEQ
                       RESP(ERR-CODE)
                       END-EXEC
        IF ERR-CODE EQUAL DFHRESP(NORMAL)
            PERFORM UNTIL ERR-CODE EQUAL DFHRESP(ENDFILE)
            EXEC CICS READNEXT FILE('BAMSFKS')
                               INTO(REC-FILED)
                               LENGTH(RECL)
                               RIDFLD(RECKEY)
                               KEYLENGTH(13)
                               RESP(ERR-CODE)
                               END-EXEC
            EXEC CICS WRITEQ TS QUEUE(Q-NAME)
                               FROM(REC-FILED)
                               LENGTH(RECL)
                               ITEM(I-COUNT)
                               END-EXEC
            END-PERFORM
            EXEC CICS ENDBR FILE('BAMSFKS') END-EXEC
            EXEC CICS XCTL PROGRAM('BAMSCL')
                          COMMAREA(Q-NAME)
                          LENGTH(8)
                          END-EXEC.
```

18.5 查询账户功能模块

此处的查询账户功能模块相对前 3 个模块比较特殊。该模块是由修改账户功能模块进行模糊查找时所调用的，而不是由主菜单调用。在该模块的界面上，将把满足模糊查找条件的所有账户数据列表出来，并支持翻页功能，以供用户查询。查询账户功能模块的界面如图 18.5 所示。

查询账户功能模块的任务要求如下。

（1）读取由修改账户功能模块所传递的临时存储队列信息。

（2）输出界面，列出满足模糊查找条件（部分账号信息）的所有记录，且每页至多显示5条。同时允许用户通过在“TAG”下的相应下划线处输入“X”以确定所要查找并修改的记录。该记录即与所输入的“X”为同一行的数据记录。每次只能查找一条记录，因此只能输入一个“X”，如图18.5所示。

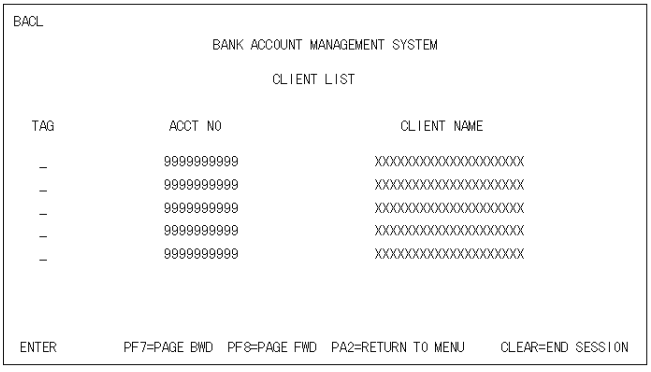

图18.5　查询账户功能模块界面

（3）根据用户在输入数据前后所按的不同功能按键，分别作出如下处理。

- “Enter”按键：执行步骤5。
- Pause/Break按键：输出提示信息“LIST SESSION ENDED”，同时退出该系统。
- “PA2（F2）”按键：返回主菜单。
- “PF7（F7）按键”：上翻界面记录信息。
- “PF8（F8）按键”：下翻界面记录信息。

（4）如果数据记录由多页显示（超过5条），并且用户选择了翻页功能，需遵循以下原则。

- 如果该页为第一页，则不显示“PF7=PAGE BWD”，不允许进行上翻操作。
- 如果该页为最后一页，则不显示“PF8=PAGE FWD”，不允许进行下翻操作。

（5）若用户选定了某一条记录，则将该记录内容传递到修改账户功能模块执行，同时退出本模块。设该模块所用到的各项资源名称分别如下。

- 交易名称：BACL。
- 程序名称：BAMSCL。
- 界面所对应的MAP名称：BAMAPCL。
- MAP所在的MAPSET名称：BAMPSCL。
- 临时存储队列名称：由修改账户功能模块的程序BAMSMF所传递。

同时，令本模块中符号MAP里的各变量如下。

- ACC-M：对应账号信息部分。
- CLI-M：对应姓名信息部分。
- PF7-M：对应“PF7=PAGE BWD”字段。
- PF8-M：对应“PF8=PAGE FWD”字段。
- MODI-M：对应“TAG”下面的下划线。

实现该模块所有功能的完整程序代码如下。

```
IDENTIFICATION DIVISION.
```

```
 PROGRAM-ID BAMSCL.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 COMSTART.
    05 STATUS-I    PIC X VALUE 'N'.
    05 ACCOUN-I    PIC X(10) VALUE SPACES.
    05 CLIENT-I    PIC X(20) VALUE SPACES.
 01 DATA-FILED.
    05 S-ACCNO     PIC 9(10) VALUE ZERO.
    05 S-SBA       PIC 9(3)  VALUE ZERO.
    05 S-CNAME     PIC X(20) VALUE SPACES.
    05 S-NOTUSE    PIC X(27)  VALUE SPACES.
 01 RECL           PIC 9(2)  COMP VALUE 60.
 01 RECKEY         PIC X(13).
 01 ERR-CODE       PIC S9(8) COMP.
 01 COUNTER        PIC 9(4)  VALUE ZERO.
 01 I-COUNT        PIC S9(4) COMP.
 01 Q-NAME         PIC X(8).
 01 OUTMSG         PIC X(79) VALUE SPACES.
 COPY DFHAID.
 COPY BAMPSCL.
 COPY BAMPSMF.
 COPY DFHBMSCA.
 LINKAGE SECTION.
 01 DFHCOMMAREA.
    05 STATUC-C    PIC X.
    05 ACCOUN-C    PIC X(10).
05 CLIENT-C        PIC X(20).
05 QNAME-C         PIC X(8).
 PROCEDURE DIVISION.
*** 当用户按<Pause/Break>按键时，输出相应提示信息，并退出该 MIS 系统 ***
    IF EIBAID = DFHPAUSE/BREAK
          MOVE 'LIST SESSION ENDED' TO OUTMSG
          EXEC CICS SEND FROM(OUTMSG)
                         LENGTH(79)
                         END-EXEC
          EXEC CICS RETURN END-EXEC.
*** 当用户按<F2>按键时，返回主菜单 ***
      IF EIBAID = DFHPF2
          EXEC CICS XCTL PROGRAM('BAMSMM')
END-EXEC.
*** 当任务第一次执行时，输出相应信息，并 RETURN 到自身 ***
      IF EIBCALEN = 0
        MOVE DFHRESP(NORMAL) TO ERR-CODE
        MOVE 0 TO COUNTER
        MOVE 1 TO I-COUNT
        MOVE LOW-VALUES TO BAMAPCLO
        PERFORM UNTIL ERR-CODE NOT = DFHRESP(NORMAL)
           IF COUNTER <= 5
                EXEC CICS READQ TS QUEUE(QNAME-C)
                                   INTO(DATA-FILED)
                                   LENGTH(RECL)
                                   ITEM(I-COUNT)
                                   RESP(ERR-CODE)
```

```
                                    END-EXEC
                ADD 1 TO I-COUNT
                ADD 1 TO COUNTER
                IF COUNTER <= 4
                   MOVE DFHBMUNP TO MODI-MA(COUNTER)
                   MOVE S-ACCNO TO ACC-MO(COUNTER)
                   MOVE S-CNAME TO CLI-MO(COUNTER)
                   MOVE DFHBMPRO TO ACC-MA(COUNTER)
                   MOVE DFHBMPRO TO CLI-MA(COUNTER)
                END-IF
             ELSE
                MOVE 'PF8=PAGE FWD' TO PF8-MO
                MOVE DFHRESP(ENDFILE) TO ERR-CODE
                MOVE 'N' TO STATUS-I
             END-IF
          END-PERFORM
          EXEC CICS SEND MAP('BAMAPCL')
                         MAPSET('BAMPSCL')
                         CURSOR(731)
                         ERASE
                         FREEKB
                         END-EXEC
       EXEC CICS RETURN TRANSID('BACL')
                          COMMAREA(COMSTART)
                          LENGTH(31)
                          END-EXEC.
*** 当用户进行翻页时所做的处理 ***
          MOVE 0 TO I-COUNT
          IF STATUC-C = 'N' AND EIBAID NOT EQUAL DFHENTER
             IF EIBAID = DFHPF8
                MOVE 'PF7=PAGE BWD' TO PF7-MO
                ADD 5 TO I-COUNT
             END-IF
             IF EIBAID = DFHPF7
                MOVE 'PF8=PAGE FWD' TO PF8-MO
                MOVE 6 TO I-COUNT
                SUBTRACT 5 FROM I-COUNT GIVING I-COUNT
             END-IF
             PERFORM UNTIL COUNTER = 5
                EXEC CICS READQ TS QUEUE(QNAME-C)
                                   INTO(DATA-FILED)
                                   LENGTH(RECL)
                                   ITEM(I-COUNT)
                                   END-EXEC
                MOVE S-ACCNO TO ACC-MO(COUNTER)
                MOVE S-CNAME TO CLI-MO(COUNTER)
                MOVE DFHBMPRO TO ACC-MA(COUNTER)
                MOVE DFHBMPRO TO CLI-MA(COUNTER)
                MOVE DFHBMUNP TO MODI-MA(COUNTER)
                ADD 1 TO COUNTER
                ADD 1 TO I-COUNT
             END-PERFORM
             EXEC CICS SEND MAP('BAMAPCL')
                            MAPSET('BAMAPCL')
                            CURSOR(731)
```

```
                          ERASE
                          FREEKB
                          END-EXEC
             EXEC CICS RETURN TRANSID('BACL')
                           COMMAREA(COMSTART)
                           LENGTH(31)
                           END-EXEC
         END-IF.
*** 当用户选定所要修改的账户信息时所做的处理 ***
         IF EIBAID = DFHENTER
             EXEC CICS RECEIVE MAP('BAMAPCL')
                              MAPSET('BAMPSCL')
                              END-EXEC
             MOVE 0 TO COUNTER
             PERFORM UNTIL COUNTER = 5
               EVALUATE TRUE
                 WHEN MODI-MI(COUNTER) = 'X'
                  MOVE COUNTER TO I-COUNT
                  EXEC CICS READQ TS QUEUE(QNAME-C)
                                     INTO(DATA-FILED)
                                     LENGTH(RECL)
                                     ITEM(I-COUNT)
                                     END-EXEC
                  EXEC CICS DELETEQ TS QUEUE(QNAME-C) END-EXEC
                  MOVE S-ACCNO TO ACCOUN-I
                  MOVE S-CNAME TO CLIENT-I
                  MOVE 'R' TO STATUS-I
               END-EVALUATE
               ADD 1 TO COUNTER
             END-PERFORM
             MOVE LOW-VALUES TO BAMAPMFO
             MOVE DFHBMPRO TO ACC-N4A
             MOVE DFHBMUNP TO CLI-N4A
             MOVE ACCOUN-I  TO ACC-N4O
             MOVE CLIENT-I    TO CLI-N4O
             EXEC CICS SEND MAP('BAMAPMF')
                            MAPSET('BAMPSMF')
                            CURSOR(749)
                            ERASE
                            FREEKB
                            END-EXEC
             EXEC CICS XCTL PROGRAM('BAMSMF')
                           COMMAREA(COMSTART)
                           LENGTH(31)
                           END-EXEC
             EXEC CICS RETURN END-EXEC.
                   GOBACK.
```

18.6 本章回顾

本章属于实战篇，主要介绍了如何通过 COBOL 开发一个小型模拟 MIS 系统。在该系统的源码中还结合了很多以 COBOL 作为宿主语言的 CICS 方面的功能。在 COBOL 的实际开发中，这种方式往往是应用得最多的。

本章所介绍的 MIS 系统为银行账户管理信息系统，主要功能是实现对账户的增删改查操作。其中账户数据由 VSAM 文件存放，用户操作界面为 CICS 中的 MAP。该系统在实现基本功能的同时，还提供了多种针对各种情况的系统提示信息，具有较强的健壮性。将本系统略作改动，可以很方便地生成其他各种 MIS 系统，如图书管理信息系统、员工管理信息系统、票务管理信息系统等。

此外，本系统在功能上也可以进一步完善，如实现以账户姓名（次关键字）作为查找依据的功能，实现对账户的各项统计功能，实现对账户的多项信息（包括注册时间、存款额、取款额等）进行管理的功能，实现将账户数据通过 DB2 进行管理，并使用 SQL 与之交互的功能等。同时，在健壮性及时间空间利用率方面，本系统还可以进一步地加强。

学习本章内容，关键是要对使用 COBOL 及与之相关的技术开发完整的系统有一个总体认识。通过此项实际案例，可以作为对前面各章所学知识的一个回顾。

最后，通过全书的学习，相信应该对 COBOL 及其相关知识有了一个比较全面的了解和掌握。并且，在此基础之上，能够顺利进入大型机开发领域，胜任相应的工作。