1 简单说明

内部交流、或可作培训使用。对用户作如下假定：

1、 能 COPY、修改、编译源代码（RPGLE、CLP），并能运行编译后的程序

2、 能 COPY、修改、编译文件（PF、LF、PRTF、DSPF）；

3、 对数据文件（PF）有简单的认识（FIELD → RECORD → PF），并知道 LF 与 PF 的对应关系。

2 程序代码行的编写

2.1 最简单的 RPGLE 程序

为便于理解，这里写一个最简单的 RPGLE 程序

```
CL0N01Factor1+++++++Opcode&ExtFactor2+++++++Result++++++++Len++D+HiLoEq
      *************** Beginning of data ************************************
0001.00 C     'HELLO WORLD'   DSPLY
0002.00 C                     RETURN
      ****************** End of data ***************************************
```

这个程序编译成功，并调用（CALL　程序名），就是在屏幕上反白显示“HELLO WORLD”字样。（其中，绿色字样，是系统自动显示的，下同）

与自由风格的 C 语言不同，RPGLE 中的编码，是有一定的格式，如果写错，将会在当前代码行上高亮反绿显示。初学者如果不太清楚从何处开始下手，可以使用“F4”键查看（F4 键只有用 2 进入的编辑状态才有效，用 5 进入的查看状态是无效的）

```
Level      N01 Factor 1              Operation        Factor 2        Result
                'HELLO WORLD'        DSPLY
                Decimal
Length     Positions      HI    LO    EQ     Comment
```

关于每一项所对应的内容代表什么意思，该如何填写，即如何写程序，将会在下面的具体讲解。

2.2 举例准备

列出表名，字段，以方便下面的举例。

假设有 PF 文件叫 PFFHS，文件的记录格式叫 FMTFHS

每条记录，都是由 FHS01、FHS02、FHS03 三个字段组成，每个字段都是两位长的字符

型变量。

逻辑文件 PFFHSL1 的键值为 FHS01

逻辑文件 PFFHSL2 的键值为 FHS02

逻辑文件 PFFHSL3 的键值为 FHS01、FHS02

注：

文件的记录格式，可以理解为给这个文件整条记录起的一个名字；或者是说将每条记录视都视做一个类型相同大变量，然后给这个大变量起的名字。所以文件的记录格式信息中，包含有一条记录由多少个字段组成，总计长度是多少这样的信息。

文件的记录格式，与各个字段同时定义。（写文件的源码时）

文件的记录格式在 RPGLE 的程序中，不能与文件名相同。

2.3 简单的程序流程

为方便起见，系统自动显示的就不再贴出来了，只贴代码段。

```
FPFFHS     UF     E                 DISK
C                 READ      FMTFHS
C                 EVAL      FHS01=”01”
C                 UPDATE    FMTFHS
C                 SETON                          LR
C                 RETURN
```

这个程序的意思，是说读 PFFHS 这个文件，然后将读到的第一条记录中的 FHS01 这个字段的值修改为“01”。

“SETON LR”，LR 的位置可在 HI、LO、EQ 中任选一处。意思是指将打开指示器*INLR，即赋值使指示器*INLR 的值等于 1。等价于 “ EVAL *INLR=’1’ ”，意思是强制将内存中的数据写到磁盘中。（基于效率因素，系统在修改文件时，会先将修改的结果先放在内存中，在同一程序中，读取数据也是先从内存中查询。）LR，取自是 Last Record

RETURN，表示程序结束，在后面“操作码”一节中，会有讲述。

如果不太明白，就记住

```
C           SETON                          LR
C           RETURN
```

或

```
C           EVAL        *INLR=’1’
C           RETURN
```

这两句话加在一起，表示程序结束就可以了。

从这个程序中，我们可以看到，RPGLE 的程序，大致上可以分为两个部分：

1、 声明、定义部分：声明程序中使用到的文件（F 行），定义程序中使用的变量（D 行）

2、 程序运行部分 ：即 C 行，也就是程序段。

在 RPGLE 程序中，F 行必须在 D 行前面，D 行必须在 C 行前面。

程序执行的起始顺序，将从定义部分之后，第一个 C 行开始，顺序向下执行。

程序中的 F 行、D 行都不是必须项，一个程序可以没有 F 行（如仅完成计算功能的公共函数，比如计算利息），也可以没有 D 行（没有需要特别定义的变量，或者所有变量都在 C 行进行定义），但不应该没有 C 行，因为 F 行与 D 行都属于非执行行，是起定义作用；C 行是执行行。没有 C 行的程序，是无执行意义的。

2.4 常见的程序流程

```
FPFFHS      UF    E          DISK       //声明文件 PFFHS
D   LSFLD01       S          2          //定义临时变量 LSFLD01
C                 EVAL       LSFLD01='01'  //给变量 LSFLD01 赋值
C                 EXSR       SUB#UPD    //执行子过程 SUB#UPD
C                 EVAL       LSFLD02='02'  //给变量 LSFLD02 赋值
C                 EXSR       SUB#UPD    //执行子过程 SUB＃UPD
C                 SETON                 LR      //数据写入磁盘
C                 RETURN                       //程序结束
C   SUB#UPD       BEGSR                 //子过程 SUB#UPD 开始
C                 READ       FMTFHS     //读 PFFHS 文件
C                 EVAL       FLD01=LSFLD01  //给字段 FLD01 赋值
C                 UPDATE     FMTFHS     //修改文件
C                 ENDSR                 //子过程结束
```

“//”后面的，只是简单的解释，如果自己动手写，不需要输入这些内容。

系统在运行这个程序时，是按如下的顺序来执行：

1. 首句 EVAL 赋值语句，直接执行；
2. 当系统发现操作码“EXSR”时，根据后面的变量名“SUB#UPD”，去查找对应的“SUB#UPD　　BEGSR”语句；
3. 然后从“SUB#UPD　　BEGSR”之后，顺序向下执行，直至“ENDSR”语句
4. 执行到“ENDSR”之后，将会再回到当初的“EXSR　SUB#UPD”处，继续向下执行，直到 RETURN 语句为止。

这里提出一点要注意，如果子过程中，又执行了自身，即在 SUB#UPD 程序中，又出现了“EXSR　　SUB#UPD”，是可以编译通过的，但在执行过程中，系统会因为无法定位，而出现死循环，直至报错异常中断退出。也就是 RPGLE 的程序中，子过程不允许出现递归。

2.5 F 行说明

2.5.1　内容说明

首位填上 F，然后按 F4，会出现如下内容：

File Filename **File Type** **File Designation** **End of File** **File Addition** **Sequence**

File Format **Record Length** **Limits Processing** **Length of Key Field** **Record Address Type**

File Organization **Device** **Keywords**

Comment

各项的含义分别是：

Filename：

需要声明的文件名，必须顶格，文件名必须唯一，也就是程序中对同样的文件名不能声明两次。

File Type：

声明文件的处理类型。必须填写。允许的选项有：

I： 输入型，即只读文件，对声明的文件只取其记录的值，不对记录进行修改

U： 修改型，即对声明的文件进行修改操作（删除记录属于修改操作的一种）

O： 输出型，即只写，对声明的文件只进行写操作。

C：混合型，用于对屏幕文件的定义。（混合型，即输入/输出型，以屏幕文件为便，也就是读取屏幕文件的一些输入字段信息，同时也可以输出一些字段的值到屏幕文件中，但不能对屏幕文件自身进行修改，所以与上面的 U 是有区别的）

File Designation：

文件的指定方式，允许的选项有：

不填：表示这是一个输出文件，即“File Type”项为“O”时，此项不填

P：表明声明的文件是主文件，这个很少用，cycle 相关

S：表明声明的文件是次文件，这个没用过，cycle 相关

R：Record address file，记录地址文件？没用过

T：数组或表文件？不懂，没用过

F：常用，具体含义不知道该如何翻译(Full procedural file)

简单来说，不考虑 cycle(循环控制)，这样理解就够了：

当“File Type”为 I，U，C 时，这里填“F”

当“File Type ”为 O 时，这里不填写

End of File：

程序结束前，对记录的处理方式。可以不填，或填“E”。但从英文解释上来看，不敢妄下定论，似乎不填，表示在程序结束前，要处理所有文件的所有记录（含 LF？）；填 E，表示只处理这个文件的所有记录？

总之，此项一般是不填。

File Addtion：

是否会增加文件中的记录，即是否会对文件进行写操作。

可以不填，或填“A”

当 File Type 为“O”时，系统自动默认此项为“A”，不必填写；

当 File Type 为“I”，或“U”时，这项内容可以填“A”，也可以不填。不填，即表示不会增加文件中的记录，也就是没有写操作；填“A”时，即表示会增加文件中的记录，也就是会对文件进行写操作。

Sequence：

针对 cycle 使用的，表示排序顺序。(Cycle 我没有用过，估计可能是使用控制起来，程序代码不那么直观，不利于上手和维护，所以现在已经不流行使用了。)

当定义为非 cycle 文件时，即“File Designation”项非“P”、“S”时，此项必须为空；

当定义为 cycle 文件时，即“File Designation”项为“P”、或“S”时，此项可填空、A、D。A 表示升序，D 表示降序。

因为 CYCLE 现在已不常用，所以通常不填。

File Format

文件格式，不能为空，允许的值有：

E： 声明的文件，是外部描述的文件（即文件在程序运行之前就已存在？）

F： 声明的文件，是一个程序描述文件？（不知道什么意思，没用过）

这里通常填“E”，即为外部描述文件

Record Length

“File Format”为“F”时，才需要填写。没用过

通常不填

Limit Processing

不懂。

通常不填。

Length of Key Field

查询时，索引键值的长度

如果“File Format”项等于“E”，即外部描述文件时，此项不填

如果“File Format”项等于“F”，便不需要按 KEY 值查询时，此项也不填

如果“File Format”项等于“F”，需要按 KEY 值查询时，此项填写 KEY 值的长度（1—2000）。

因为一般都使用外部描述文件，所以这里一般都不填写。

Record Address Type

记录寻址类型，好象是对文件键值的描述。允许的值如下：

空：不使用 KEY 值，在程序段中，不会对文件的查询定位操作，如“SETLL”、“CHAIN”操作码都不会用的时，该项填空。

K： 使用 KEY 值，即表示会对声明的文件进行查询定位操作，此时声明的文件必须有键值，即必须为逻辑文件（LF 文件），或在生成文件时，已加入了 KEY 值。

（下面的选项应该是程序描述文件才会使用）

A：KEY 值为字符型

D：KEY 值为日期型

F：KEY 值为数字型

G：KEY 值为非英文字符

P：KEY 值为压缩型数字

T：KEY 值为时间型

Z：KEY 值为 timestamp？

总之，如果要按照键值对声明的文件进行查询定位操作（即程序中使用了 CHAIN、

SETLL 操作码，则此项需要填写“K”；如不需要进行查询操作，则不填。），此项填“K”时，声明的文件必须含有 KEY 值。

File Organization

不知道，一般不填

Device

声明文件的存放位置，必须填写，允许的值有：

DISK：　　磁盘文件，即文件存储在磁盘上，最常见的；

PRINTER：　打印文件，提供打印输出描述，以及对打印设备访问。打印报表用这个；

WORKSTN：workstation，工作站，显示文件。屏幕文件(DSPF)的定义用这个值

（下面这两种我没用过的）

SEQ：　　磁带文件，文件存储在磁带上。

SPECIAL：　特殊文件，我现在也不是很清楚具体使用方式。据 blogliou 说，这种类型，是允许指定一种不能被 RPG 直接操作的输入/输出设备。比如可以通过 SPECIAL 文件，在 RPGLE 程序中实现象读写磁盘一样，对 DTAQ 进行程序间数据交换。

Keyword

可以不填，常用的值有（这里只列出几个常用的）：

COMMIT

该文件记录的数据操作进行日志处理（关于日志处理，后面会章节会讲到）

RENAME

对文件记录格式名进行重命名。比如说程序中需要同时声明 PFFHSL1，PFFHSL2 这两个逻辑文件。这两个逻辑文件的记录格式名都是一样（通常和 PF 一样，即都为 FMTFHS；不过也可以定义成不同。如果不同，当然就不需要使用 RENAME 键字了）。

那么，为了能让系统区分，就必须对其中一个的记录格式名进行重命名。RENAME 的语法：RENAME（旧记录格式名：新记录格名），如下：

```
FPFFHSL1   IF          E       DISK
FPFFHSL2   IF          E       DISK    RENAME（FMTFHS：FMTFHS2
```

新记录格式可以自由定义，只要在该程序中无同名的即可。RENAME 并不会真正的更改文件的记录格式名，仅是在当前运行程序中进行重命名。对同时运行的其它程序无影响

USROPN

对于声明的文件，由用户自行打开。如果不填写此关键字，系统将会在程序最最开始（执行第一句 C 行语句前），自动执行“OPEN 文件” 的操作，在程序结束后，自动执行“CLOSE 文件”的操作。而填写此关键字之后，OPEN，CLOSE 的操作将由用户在 C 行程序段中，自行处理。如果用户未执行 OPEN 操作，就执行 CHAIN、READ、SETLL 等语句，在编译程序时就会报错。程序在结束之前，必须关闭所有已打开的文件，所以用起来会比较繁琐。USROPN 常作用于对文件的解锁，在同一程序中打开同一文件的不同 MEMBER 等，属于一个较高级的用法，可在实际操作中慢慢体会。

OPEN，CLOSE 的操作码，对应的是文件名，不是记录格式名。即

```
C                   OPEN        PFFHSL1
C                   CLOSE       PFFHSL1
```

而不是

```
C                   OPEN        FMTFHS
```

Comment

注释说明。源自 RPG，在 RPG 中是有作用的，可以对程序作简短的说明，但在 RPGLE 中，其实已经没有作用了，此项不用填。（填了也没用）

2.5.2 常用例子

对文件进行只读的声明：

```
FPFFHS      IF      E               DISK
```

对文件进行修改的声明：

```
FPFFHS      UF      E               DISK
```

对文件进行只写的声明：

```
FPFFHS      O       E               DISK
```

对文件进行修改，以及增加记录的操作：

```
FPFFHS      UF A    E               DISK
```

对文件进行查询，增加记录的操作，并对文件进行查询操作：

```
FPFFHSL1    IF A    E               K DISK
```

声明两个记录格式相同的文件，并对其中之一进行重命名

```
FPFFHSL1    IF      E               K DISK
FPFFHSL2    IF      E               K DISK      RENAME(FMTFHS:FMTFHS2)
```

注：在声明时，两个文件不一定要上下紧接着；随便改哪一个文件对应的记录格式都可以；新旧记录格式名用冒号隔开，新记录格式名可自行定义，无规则。

对文件的修改操作进行日志处理：

```
FPFFHSL2    UF      E               K DISK      COMMIT
```

cycle 类文件的声明：

```
FPFFHSL2    IP      E               K DISK
```

这样文件声明为 P 之后，程序中不需要写循环读文件，也不需要写 RETURN，设指示器 INLR，也就是

```
FPFFHSL2    IP      E               K DISK
C                   READ    记录格式名
```

等价于

```
FPFFHSL2    IF      E               K DISK
C                   DOW     1 = 1
C                   READ    记录格式名                  EQ 指示器
C                   IF      EQ 指示器='1'
C                   LEAVE
C                   ENDIF
C                   ENDDO
C                   RETURN
```

2.5.3 补充说明

声明的文件，可以同时使用多个 keyword 关键字，并可以不在同一行（但必须紧接在声明的文件的下面），如下：

FPFFHSL2　IF　　E　　DISK　　RENAME（FMTFHS：FMTFHS2）
F　　　　　　　　　　　　　　COMMIT

即表示文件 PFFHSL2，同时使用了 RENAME、COMMIT 两个关键字。

如果写得下，也可以写在同一行，以空格键分开，如下

FPFFHSL2　IF　　E　　DISK　　COMMIT　RENAME(FMTFHS：FMTFHS2）

2.6 D 行说明

首行填“D”，然后按 F4，会出现如下内容：

			Declaration		To /
Name	**E**	**S/U**	**Type**	**From**	**Length**

Internal	Decimal	
Data Type	**Positions**	**Keywords**

Comment

2.6.1 内容说明

Name:

定义的变量的名字，该名字可以不顶格写。(即允许有缩进）

E:

标识定义的变量是否源自外部数据结构。可以不填，或填“E”

上面的解释可能有点饶口，其实这个地方的意思，就是说：

如果是程序内部自行定义一个临时变量，此处不填；

如果是引用的一个外部文件作为数据结构，那么这里就要填“E”；同时“Declaration Type”处，就要填“DS”，即定义为一个结构；“Keywords”处要使用 EXTNAME 关键字

所谓“引用一个外部文件作为数据结构”，也就是说定义一个结构，整个结构中的变量，参照外部文件来定义。

所谓结构，可以理解为一个“由多个变量组合而成的大变量”。

举例而言：

D MYDS　　E DS　　　　EXTNAME(PFFHS)

和

D MYDS　　　DS
D　FHS01　　　　1　2（1 在 From 项；2 在 To / length 项）

```
D     FHS02                    3     4
D     FHS03                    5     6
```

是等价的，都是定义一个结构变量 MYDS（名字可以自行定义），这个结构变量是由三个字符型变量 FHS01，FHS02，FHS03 拼成的。

第一种定义方法，就是引用外部文件“PFFHS”作为数据结构的定义，注意使用到了“EXTNAME”关键字，而且“E”项的值为“E”。

而第二种定义方法，就是直接定义一个结构“MYDS”。注意没有使用外部文件时，“E”项的值为空。

S/U：

不知道，一般都填空。

Declaration Type：

定义变量的类型，允许的值如下：

不填：　非以下内容：数据结构、常量、独立变量、数组、表。此项为空时，好象只能用来表示当前定义的变量是属于结构的一个变量。在下面会举例

DS：　数据结构，即定义一个结构变量，这个之前已讲过

C：　常量

常量只能使用字符，不需要定义常量的长度、类型。常量的内容写在“Keywords”处，并使用 CONST 关键字，在程序段中，不能对常量进行赋值操作。

```
D MYNUM              C                   CONST('abcdefghijklmn')
```

就是定义一个叫做 MYNUM 的常量，这个常量包含字母 a--n。

PI：　不知道，没有用过

PR：　不知道，没用过

S：　定义以下内容：独立变量、数组、表

定义一个叫 MYFIELD1 的变量，变量为 1 位长的字符型

```
D    MYFIELD1        S                1     //1 在“To/length”项
```

定义一个叫 MYARRAY 的数组，共含 3 条记录，每条记录为 1 位字符型

```
D    MYARRAY         S                1     DIM(3)   //DIM 在“Keywords”项
```

表的定义没有用过

总之，这一项，最常用的，就是“DS”、“S”与空。即结构体与独立变量，其它选项较少用到。

From：

当“Declaration Type”项为“S”时，表示独立变量、数组，此项不填

当“Declaration Type”项为“DS”时，表示结构，此项仍然不填

当“Declaration Type”项为空时，表示当前定义的变量，属于上面定义的结构，此时，此项可以填写，也可以不填写。

当填写时，“From”项表示变量在结构中的起始位置，右对齐；“To/length”表示变量在结构中的结束位置，也是右对齐。

当不填写时，“To/length”表示直接定义为变量长度。

举例：

```
D MYDS            DS
D    DSFLD01          1     2     //1 在“From”项，2 在“To/length”项
D    DSFLD02          3     4
```

与

```
D MYDS            DS
```

```
D    DSFLD01          2    //2 在“To/length”项
D    DSFLD02          2
```

其实是等价的，都是定义一个结构变量 MYDS，这个结构变量中，包含了两个变量 DSFLD01，DSFLD02，这两个变量都是两位长字符。所不同的是，第一种定义方法，是指定了变量在结构中的位置；而第二种方法，是直接指定变量的长度和类型

注意到上面的定义中，DSFLD01、DSFLD02 的 **Declaration Type** 为空，也就是表示这两个字段是属于上面定义的结构 MYDS。如果此项为“S”，即表示这个变量与结构无关

```
D MYDS           DS
D    DSFLD01          2    //2 在“To/length”项
D    DSFLD02     S    2
```

在这个定义中，变量 DSFLD02 就是一个独立的变量，与结构 MYDS 无关。

Length：

上面已讲述在定义结构时的使用方法。

在定义非结构时，此项的内容即为定义变量的长度。右对齐

Internal Data Type：

定义变量的类型，允许的值有：

空：变量定义为字符型、压缩型数字

A： 变量定义为字符型

B： 二进制？不知道

D： 变量定义为日期型

F： 变量定义为浮点型？

G： 变量定义为图型？（非英文？汉字？）

I： 变量定义为带符号的整数

N： 变量定义为指示器变量？（没用过）

P： 变量定义为压缩型数字

S： 变量定义为普通的数型

T： 变量定义为时间型

U： 变量定义为无符号的整数

Z： 变量定义为日期+时间型（格式：年-月-日-时.分.秒.微秒）

*： 变量定义为指针型

其实我最常用，就是不填，因为一般的程序，有字符和数字这两种类型变量，就足够了。

Decimal Positions：

当变量定义为数字型时，用来标志小数的位数。

当“To/Length”项为 3，“Internal Data Type”项为空时

此项为空，表示定义的变量为 3 位长的字符型

```
D    MYFLD01          S         3              //定义为 3 位字符型
```

此项不为空（右对齐），表示定义的变量为数字型

```
D    MYFLD01          S         3  2           //定义数字型变量，1 位整数，2 位小数（总长为 3 位）
```

Keywords：

关键字，可以不填，常用的值如下：（同样，这里我也只列出几个常用的，这里先不做详细说明，仅供参考，在后面的例子，看看就知道用法了）

CONST： 定义常量的值

DIM：　　定义数组
EXTNAME：引用外部文件作为数据结构变量
EXTFLD：　对引用了外部文件作为数据结构的某个变量，进行重命名
LIKE：　　定义变量时，参照已存在的变量定义
OCCURS：　定义结构体变量时，指定的结构体变量的记录条数
INZ：　　定义变量时，赋值初始值
DATFMT：　定义日期变量时，指定日期格式
*MDY (mm/dd/yy)
*DMY (dd/mm/yy)
*YMD (yy/mm/dd)
*JUL (yy/ddd)
*ISO (yyyy-mm-dd)
*USA (mm/dd/yyyy)
*EUR (dd.mm.yyyy)
*JIS (yyyy-mm-dd)

Comment

注释项，源自 RPG，不用填，因为填了也没用。

2.6.2　常用例子

定义一个 10 位长的字符型变量：

```
D MYFLD          S                10
```

定义一个 10 位长，其中含 2 位小数的字符型变量，并使其初始值为 1

```
D MYFLD          S                10    2     INZ(1)
```

定义一个每条记录为 5 位长字符型变量，共 10 条记录的数组

```
D MYFLD          S                 5          DIM(10)
```

定义一个 10 位长的字符型变量，再定义一个变量，参照前一变量定义

```
D MYFLD01        S                10
D MYFLD02        S                           LIKE(MYFLD01)
```

定义一个结构，由一个 3 位长的字符变量，和一个 10 位长，其中 2 位小数的数字变量组成

```
D MYDS           DS
D    MYDS01                        3
D    MYDS02                       10    2
```

定义一个结构变量，结构内容参照外部文件 PFFHS

```
D MYDS       E    DS                     EXTNAME(PFFHS)
```

定义一个结构变量，结构内容参照外部文件 PFFHS，并且将第二个字段重命名为 FHS999

```
D MYDS       E    DS                     EXTNAME(PFFHS)
D FHS999     E    DS                     EXTFLD(FHS02)
```

定义一个日期型变量，格式为 yyyy-mm-dd

```
D MYDATE         S                   D    DATFMT(*ISO)
```

2.6.3 补充说明

变量的定义，除了在 D 行定义之外，还可以在 C 行通过赋值语句直接定义

如

```
D    FLD01        S                2    INZ('01')
```

与

```
C                 MOVE       '01'      FLD01        2     //2 在 length 处，右对齐
```

是等价的

定义结构之后，可以将结构变量视为一个普通的变量进行赋值来改变结构变量的值，也可以通过对组成结构变量的变量进行赋值，来达到修改结构变量的值的目的。

如：

```
D MYDS           DS
D    MYFLD01                  2
D    MYFLD02                  2
```

在 C 行中，这两句是等价的

```
C                 EVAL       %SUBST(MYDS:3:2)='01'
C                 EVAL       MYFLD02='01'
```

第一句是直接改结构变量 MYDS 的后两位的值（当然，此时 MYFLD02 的值也变化了）

第二句是对 MYFLD02 进行赋值，同样，赋完值之后，MYDS 的后两位也变为'01'

在需要频繁进行数字与字符之间转换时，偷懒的人会通过定义这样的结构来达到目的：

```
D MYDS           DS
D    MYFLD01             1    8
D    MYFLD02             1    8 0
```

比如说，给 MYFLD01 赋值为'20070208'之后，MYFLD02 也就自动等于 20070208；然后给 MYFLD02 加 1 之后，MYFLD02 等于 20070209，MYFLD01 的值也自动等于'20070209'。可以认为结构变量 MYDS 是字符型（即一直等于 MYFLD01 的值）

这种方法，当需要字符型变量时，就使用 MYFLD01；当需要数字变量时，就使用 MYFLD02，不过我总觉得有点类似于作弊，一般没用。

关于数组、结构体的内容，因为要说起来内容还颇多，而也属于略为高级一些的用法，所以将在后面专设章节讲述。

2.7 入口参数

程序可以通过“*ENTRY”定义入口参数，或称之为接口参数，来传递数据。

假设有程序 FHS01ILE，其中入口参数的定义如下：

```
C          *ENTRY     PLIST
C                     PARM                    FLD01          3
C                     PARM                    FLD02          4
```

其中：

*ENTRY 在“Factor 1”项；

PLIST 在“Operation”项；

PARM 在“Operation”项；

FLD01、FLD02 都在“Result”项

上述定义，表示这个程序通过两个字段与其它外部程序沟通。那么别的程序(如FHS02ILE)在调用程序 FHS01ILE 时，就要带上两个字符型变量，如

```
C       CALL        'FHS01ILE'
C                   PARM            FHSFLD01    3
C                   PARM            FHSFLD02    4
```

在两个程序里，这两个变量名可以不同（比如说一边叫 FHSFLD01，FHSFLD02；一边叫 FLD01，FLD02），但长度，类型必须匹配。

如果在 FHS02ILE 中，FHSFLD01 等于'123'，FHSFLD02 等于'abcd'，那么系统在运行 CALL 语句，执行程序 FHS01ILE 时，将会对字段 FLD01 初始化赋值，使其一开始就等于'123'，字段 FLD02 等于'abcd'。

如果 FHS01ILE 程序中，对 FLD01、FLD02 进行了改动，比如 FLD01 最后等于'789'，FLD02 最后等于'efgh'，那么程序 FHS02ILE 在调用完 FHS01ILE 之后，FHSFLD01、FHSFLD02 这两个字段也同样会改变，成为'789'，和'efgh'

也就是入口参数的变化是可以传递的，其实应该很好理解吧。

入口参数的定义，可以写在程序的任何一处，而程序的执行，始终是从 C 行的顺序第一行开始执行，与入口参数所在的位置无关。

FHS02ILE 也可以使用一个大变量来调用 FHS01ILE，只要总长相等即可（这种方法仅限于被调用的程序 FHS01ILE 的入口参数全部为字符型才可使用，仅仅只是不会错，不建议这样使用。

```
C       CALL        'FHS01ILE'
                    PARM            FHSFLD01    7
```

其实从上面的例子可以看出，入口参数可以使用结构的形式来表达，所以下面这种写法也不会有错。（如果被调用程序有数字型变量，只要在定义结构时也定义为数字型即可）

```
D   MYDS        DS
D       DS01        3
D       DS02        4
C       CALl        'FHS01ILE'
C                   PARM            MYDS
```

不过要注意，如果 RPG 程序调用 C 程序，那么入口参数必须严格按照 C 程序中的来，比如 C 程序中带了两个字符型参数，那么 RPG 程序中也必须是两个字段入口参数，不能使用由两个字符变量组成的结构。原理可以自行想想。

既然可以使用结构做为入口参数，当然，也可以参照外部文件来定义结构做为入口参数

```
D MYDS      E   DS                  EXTNAME(PFFHS)
C       CALL        'FHS01ILE'
```

```
C                   PARM            MYDS
```
与
```
C         CALL      'FHS01ILE'
C                   PARM            FHS01     2
C                   PARM            FHS02     2
C                   PARM            FHS03     2
```
是等价的。

可以看到，参照外部文件定义结构做为入口参数时，可以有效的节省代码行，而且不会出现遗漏。所以在实际使用中，常会看到，将一些公共程序的入口参数定义成一个 PF 文件。而调用它的程序，就参照这个 PF 文件，定义结构做为调用的接口参数。

当接口参数不一致时，如 FHS02ILE 中漏了第二个参数时：
```
C         CALL      'FHS01ILE'
C                   PARM            FHS01     3
```
此时，并不是一开始运行 FHS01ILE 程序，系统判断入口参数不符就报错；实际上，此时，FLD01 的值还是正确的，但 FLD02 的值就处于一个未初始化的状态。于是，当代码执行到与 FLD02 有关的操作码时，才会报错；如果 FHS01ILE 在运行的过程中，因为逻辑判断（如 IF 条件判断）的关系，而未执行任何与 FLD02 有关的操作码，那么程序会正常运行完毕，不会有报错。

这时，FHS02ILE 调用了程序 FHS01ILE 之后，程序中原有的接口参数的数据就可能因为这次调用程序而发生错位，从而导致数据的错误、混乱。数据的错误、混乱其实还不是最大的问题，更大的问题在于**“这时我们不知道数据已经出错了”**。解决之道，也是如上所说，对于调用频繁，且入口参数较多的公共程序，考虑将其入口参数写成一个 PF 文件。这样调整入口参数时，只要修改 PF 文件并重新编译，再编译相关程序即可（至少发生遗漏时，程序会报错异常中断，不会出现错误的数据而不自知）

2.8 C 行说明

2.8.1 写在前面

终于说到程序的执行部分，也是我们写程序的平时接触的最多的一部分：“C”行了。在这里，我想先说一下我个人的看法：

400 系统，提供了一些语法，可以大大减少程序代码行数。但是如果这个用法并不普遍，那么并不建议使用（当然自己用来练习无妨），否则会给其它读代码的人带来困难，同时也会给自己带来麻烦（比如出了问题，别人看不懂，自然会打电话来问原作者）

基于这样的道理，同样，我认为 FREE 格式的程序，虽然可以自由书写，有缩进等优势，但是除非整个项目组所有成员都已熟练掌握 FREE 格式的程序，或已进行过完善、系统的 FREE 格式的培训，才能正式使用。如果只是知道几个与 RPG，RPGLE 对应的语法就用来进行实际处理，可能会造成的维护的不便，尤其是在出现一些不那么明显的错误之后。

至于 cycle，不知道是不是基于这个原因，现在用得也比较少了。感觉 RPGLE 中，至少有一半的内容是与 cycle 相关的。

2.8.2 内容说明

Level N01 Factor 1 Operation Factor 2 Result

Decimal
Length Positions HI LO EQ Comment

Level：

和 cycle 相关，没研究过，一般不填

N01：

这个含义比较丰富，我只用过其中一种：

首位不带 N，后面填写 01—99 的数字时，表示相应的指示器打开时，执行后面的操作，如：

```
C    12                EVAL      FHS01='01'
```

等价于

```
                       IF        *IN12='1'
                       EVAL      FHS01='01'
                       ENDIF
```

首位带 N，后面填写 01—99 的数字，表示相应的指示器关闭时，执行后面的操作

要注意，该项内容仅作用于该行操作码。如果指示器打开后，需要执行多条语句，那么每条语句前面，该项都要赋值。

即

```
C                      IF        *IN12='1'
C                      EVAL      FHS01='01'
C                      EVAL      FHS02='02'
C                      ENDIF
```

如果用这种方式来表达，就要写作

```
C    12                EVAL      FHS01='01'
C    12                EVAL      FHS02='02'
```

所以说，根据指示器状态来执行的语句，在执行少量操作码时，可以使用这种方法；如果语句较多，修改起来不方便，还是直接用 IF—ENDIF 的判断语句比较合适。

该项还有针对其它指示器的用法，看上去似乎又是与 CYCLE 相关，暂不介绍了。

Factor 1：

操作内容一，将在后面与操作码一起讲

Operation：

操作码，后面有专门章节讲解操作码

Factor 2：

操作内容二，同上

Result：

操作结果，同上

Length：

长度。

变量的定义，除了在 D 行定义之外，还可以在 C 行通过赋值语句直接定义

如

D　FLD01　　S　　2　INZ(‘01’)

与

C　　MOVE　　‘01’　　FLD01　　2　//2 在 length 处，右对齐

是等价的

一个变量，在整个程序中，只要定义一次就可以了，对定义的顺序没有强制要求。

Decimal Positions：

与 length 相呼应，当此项有值时，表示定义的是一个数字型变量，该项表示小数位长度。

如

C　　Z-ADD　　2　　FLD02　　3　2

即是说，将 FLD02 定义为一个 3 位长，其中 1 位整数，2 位小数的数字变量，并赋值为 2.00

HI、LO、EQ

这是三个指示器位置项。可赋值的内容是从 01—99，在以后的说明中，如果 HI 项填写 10，LO 项填写 20，EQ 项填写 30，那么我所说的 HI 指示器，即是指*IN10，LO 指示器即是*IN20，EQ 指示器即是*IN30，依此类推。（也就是说，HI 指示器，并不是*INHI，事实上，也没有*INHI 这个指示器）

Comment：

注释行，源自 RPG，不用填，填了也没用。

2.8.3　ILE 操作码分类：

1. **程序流程控制**

   DO、DOU、DOUxx、DOW、DOWxx、ITER、LEVAE

   IF、ELSE、ELSEIF、IFxx、ORxx、ANDxx

   SELECT、WHEN、WHENxx、OTHER、

   ENDxx、

   GOTO、TAG 、

   EXSR、BEGSR、ENDSR

   CABxx

2. **初始化操作**

   CLEAR、RESET

3. **文件操作**

   OPEN、CLOSE、

   CHAIN、SETGT、SETLL、

   READ、READC、READE、READP、READPE、

   DELETE、UPDATE、WRITE 、UNLOCK

   ROLBK、COMMIT、

   EXFMT、

   ACQ、EXCEPT、FEOD、FORCE、NEXT、POST、REL

4. **程序调用**

   CALL、CALLB、CALLP、PARM、PLIST、RETURN

5. **赋值语句**

MOVE、MOVEA、MOVEL、
EVAL

6. **字符操作**
   CAT、CHECK、CHECKR、SCAN、SUBST、XLATE
7. **数字操作**
   ADD、DIV（除）、MULT（乘）、MVR（除法取余）、SQRT（开方）、SUB、XFOOT、Z-ADD、Z-SUB
8. **数组操作符**
   LOOKUP、MOVEA、SORTA、XFOOT
9. **数据区操作(没用过)**
   IN、OUT、UNLOCK
10. **日期操作**
    ADDDUR、EXTRCT、SUBDUR、TEST
11. **指示器操作**
    SETOFF、SETON
12. **信息操作（前两个没用过）**
    DUMP、SHTDN、TIME、DSPLY
13. **内存管理操作(完全没用过)**
    ALLOC、DEALLOC、REALLOC
14. **位操作（没用过）**
    BITOFF、BITON、TESTB

2.8.4 ILE 操作码

2.8.4.1 A--C

ACQ {(E)} (Acquire)

取地址位。其实 400 的程序中也有指针型变量，那么也就会有地址位，这个命令是取地址位的。我试过，不过不知道取出了地址位能干嘛，所以没有实际运用过。

ADD {(H)} (Add) **加法操作**

1. 基本语法：

Factory 1	Operation	Factory 2	Result	
FHS01	ADD	FHS02	FHS03	// RPG 的语法

等价于

	EVAL	FHS03=FHS01+FHS02	//RPGLE 的语法

FHS01、FHS02、FHS03 必须都为数字型变量（P 型，或 S 型），或者就是数字
意思是将 Factory 1 项的数据，加上 Factory 2 项的数据，赋值到 Result 项上

2. 语法二：

如果这样写的话：

Factory 1	Operation	Factory 2	Result
	ADD	FHS02	FHS03

就等价于：

	EVAL	FHS03=FHS03+FHS02

即 Factory 1 项未填时，就表示将 Result 项的数据，加上 Factory 2 项的数据，然后赋值到 Result 项上

3. 四舍五入：

（H）表示四舍五入，如 FHS02=12.50（4，2），FHS03=3（2，0），那么

ADD(H) FHS02 FHS03

执行之后，FHS03=16（因为进位了）

而

ADD FHS02 FHS03

执行之后，FHS03=15

不过实际使用中，我们都尽可能使相加的字段小数位数相等，所以 ADD 操作码一般都没有使用到四舍五入的功能。

4. 可以在 ADD 操作时，对 Result 项的变量进行定义

Factory 1	Operation	Factory 2	Result	
FHS01	ADD	FHS02	FHS03	10 2

EVAL 语句不能对在赋值的同时，对变量进行定义。

5. **关于结果数据超长时的问题：**

当加出的结果超长，比如 FHS03 定义为 3，2（1 位整数，2 位小数，下同）时，再假设 FHS01=10，FHS02=4。那么 FHS01+FHS02 本来应该等于 14，但用 ADD 操作之后，系统会自动将超长位截去，即 FHS03 最后等于 4；

而用 EVAL 语句时，系统判断超长后，会直接报错“The target for a numeric operation is too small to hold the result”，然后异常中断。

使用 ADD 操作码程序不会异常中断，但有可能发生了错误的数据我们也不知道；使用 EVAL 操作码可以避免产生错误的数据，但程序会异常中断，需要进行人工干预。可根据实际情况选择使用 ADD 还是 EVAL.

ADDDUR {(E)} (Add Duration)　　日期时间相加

1. 对日期型变量进行加操作，比如说已定义日期型变量 MYDATE1，MYDATE2，将 MYDATE1 的日期加上 3 天，赋值到 MYDATE2 中：

Factory 1	Operation	Factory 2	Result
MYDATE1	ADDDUR	3:*D	MYDATE2

其中，Factory 1，Result 项，都必须为日期型变量（即在 D 行，Internal Data Type 项为“D” ）

2. 与 ADD 操作码相同，Factory 1 项为空时，表示直接在 Result 项上进行日期相加，如将 MYDATE1 直接加上 3 个月(即结果也是赋值到 MYDATE1 中)：

Factory 1	Operation	Factory 2	Result
	ADDDUR	3:*M	MYDATE1

3. 日期型变量的参数含义：

*D 表示天，也可用*DAYS

*M 表示月，也可用*MONTHS

*Y 表示年，也可用*YEARS

4. 除了日期型之外，还有时间型，日期时间型，都可以使用 ADDDUR 操作码.

在 D 行，Internal Data Type 定义为“T”，表示时间型（时、分、秒）

Internal Data Type 定义为“Z”，表示日期时间型（年、月、日、时、分、秒、微秒）

在使用 ADDDUR 操作码，时间型的参数如下：

*H　表示小时，也可用*HOURS
*MN 表示分钟，也可用*MINUTES
*S　表示秒，　也可用*SECONDS
而日期时间型，除了可以使用*Y、*M、*D、*H、*MN、*S（以及相应的全称）之外，还可以对微秒进行处理，参数为*MS，或*MSECONDS

5. Factory 2 项中的数字，可以使用负数，使用负数时，表示减去相应的年、月、日，不过通常我会使用 SUBDUR 这个操作码来进行日期的减法，语法与 ADDDUR 相同
6. 既然说到这里，就顺便说一下对于日期型变量（时间型，日期时间型也类似）的处理。在实际运用时，常需要将日期型变量与 8 位数字型变量转换。这时要使用 MOVE 操作码。如 MOVE　日期型变量　数字型变量，反之也是。不能用 Z-ADD。

ALLOC {(E)} (Allocate Storage)

好象是给指针型变量分配空间的，没有用过

ANDxx (And)　条件判断语句—“与”

1. 在 RPG 的用法中，有 ANDEQ，ANDNE 之类的，与 IF 语句一起联用。

Factory 1	Operation	Factory 2	Result
FLD01	IFEQ	‘A’	
FLD02	ANDEQ	‘B’	
	。。。处理内容		
	ENDIF		

2. AND 后面跟的 xx，可以有如下用法：

EQ	Factory 1 = Factory2
NE	Factory 1 <> Factory2
GT	Factory 1 > Factory2
LT	Factory 1 < Factory2
GE	Factory 1 >= Factory 2
LE	Factory 1 <= Factory 2

3. 不过在实际上，因为 RPGLE 可以使用类似自由语言的风格，而且可以使用括号来处理逻辑判断先后顺序（这个是我用 RPG 写起来比较麻烦的，尤其是 ANDxx 与 ORxx 的关系，很复杂的判断写起来就不那么顺畅），所以一般都不会这么写，而是用如下的风格：

Factory 1	Operation	Factory 2	Result
	IF	FLD>FLD2　AND　FLD2<FLD3　AND (FLD3>=FLD4 OR FLD4<>FLD5)	
	。。。处理内容		
	ENDIF		

可以看出，逻辑判断的内容，可以用括号括起来以区分先后顺序；
判断的语句，允许有空格：FLD1　>　FLD2，与 FLD1>FLD2 是相同的
但 AND 前后，必须要有空格

当逻辑判断条件太长，一行写不下要分行写时，AND 在上一行，还是下一行，都没有关系。

Operation 项用 EVAL 操作码时，再按 F4，会看到 Factory2 项变成了一个很长的 Extended Factory2 项，而原来的 Result、Length、Decimal Positions、HI、LO、EQ 项都没了。

我们的逻辑判断语句，就都写在这个 Extended Factory 2 项中

BEGSR (Beginning of Subroutine)　　子过程的开始处

Factory 1	Operation	Factory 2	Result
	BEGSR	子过程名	

在前面，讲述程序流程时，已经对子过程进行了解释。这里，BEGSR 本身用法没什么特别的，只是要注意有 BEGSR 语句，就一定要有 ENDSR 对应。否则程序编译会报错。所以这里建议，如果是自己从头到尾写一个程序，最好写了 BEGSR 语句后，马上写一个 ENDSR，然后再来写中间的内容，避免遗漏。

在程序中，可以写一个子过程，但是不调用它。（也允许就是子过程没有相应的 EXSR 语句）

也允许写一个空的子过程。（也就是 BEGSR 语句与 ENDSR 语句之间没有别的执行语句了）

BITOFF (Set Bits Off)

没用过

BITON (Set Bits On)

没用过

CABxx (Compare and Branch)

没用过

CALL {(E)} (Call a Program)　　调用外部程序

Factory 1	Operation	Factory 2	Result
	CALL	‘外部程序名’	

1. 如果是直接调用外部程序，那么程序名称需要用单引号括起来，且该程序必须存在。如

	CALL	‘FHSILE01’	

就表示调用“FHSILE01”这个外部程序

2. 如果没有用单引号，如

	CALL	FHSILE01	

就表示，FHSILE01 这时是个字符型变量（即并非调用“FHSILE01 这个程序），调用的是变量内容所代表的程序。如：

FHSILE01=’FHS01’时，表示调用“FHS01”这个外部程序；

FHSILE01=’FHS02’时，表示调用“FHS02”这外外部程序

也就是说，CALL 操作码调用的程序名，可以是一个变量名。

3. 被调用的外部程序如果有接口参数，那么 CALL 操作码之后，也需要有“PARM”操作码相对应。详细内容可参考之前的“入口参数”一节。
4. 这一点要注意：虽然 400 的程序段代码中，是不区分大小写；但调用的程序名，要区分大写小。即’FHSILE01’，与 fhsile01，表示的是两个不同的程序。在使用时要注意，尤其是程序名使用字符变量来表达时，可能会因为大小写的问题而导致 CALL 不到想 CALL 的程序，从而导致程序异常中断。

CALLB {(D | E)} (Call a Bound Procedure)

也没用过，不过有不少人用，望举例说明

CALLP {(M | R | E)} (Call a Program or Procedure)

也没用过，不过有不少人用，望举例说明

CASxx (Conditionally Invoke Subroutine)　　带条件的调用子过程

1. 表示根据 xx 项对 Factory 1 与 Factory 2 进行判断，当符合条件时，执行 Result 处

的子过程。需要配合“END”或“ENDCS”语句来表示条件判断的结束。不需要“SELECT”操作码。

2. 举例如下：

Factory 1	Operation	Factory 2	Result
FLD01	CASEQ	‘1’	SUB01
FLD01	CASEQ	‘2’	SUB02
	CAS		SUB03
	ENDCS		

表示当 FLD01 等于’1’时，执行子过程 SUB01；

当 FLD01 等于’2’时，执行子过程 SUB02；

当不满足以上两个条件时，执行子过程 SUB03

最后的“ENDCS”必须要有，表示条件判断结束；但不需要 SELECT。

上面这段语句，与下面这一段是等价的：

Factory 1	Operation	Factory 2	Result
	SELECT		
	WHEN	FLD01=’1’	
	EXSR	SUB01	
	WHEN	FLD01=’2’	
	EXSR	SUB02	
	OTHER		
	EXSR	SUB03	
	ENDSL		

3. 可以看出来，CASxx 这种语句，是用于逻辑判断仅一个条件时的分支处理，这样的写法在代码的阅读上会很直观。而当逻辑判断大于一个条件时，这个语句就不适用了。

CAT {(P)} (Concatenate Two Character Strings)　　字符连接

1. 基本语法：

Factory 1	Operation	Factory 2	Result
FLD01	CAT	FLD02:0	FLD03

这句话的意思，是将 FLD02 拼在 FLD01 后面，中间没有空格，然后将拼出的结果赋值到 FLD03 中。

FLD02：0，表示 FLD02 与 FLD01 之间的空格数为 0，依此类推

FLD02：1，就表示 FLD01 后面加一个空格，再拼上 FLD02

FLD02：3，就表示 FLD01 后面加三个空格，再拼上 FLD02

Factory 1 项，与 Factory 2 项可以是字符型变量，也可以就是字符。当是字符时，需要用单引号将字符括起来

2. 其实根据 RPG 的语法，大家应该也可以想得到，Factory1 项如果不填值，就表示将 Factory 2 项的内容直接拼在 Result 项上，这里就不举例了。

3. 字段 FLD01 如果内容后面有空格，如“ABC　　”，那么在 CAT 操作时，系统会自动将后面的空格截去，只取’ABC’。举例：

FLD01=’ABC　　‘　（8 位字符型），

FLD02=’1’　　（1 位字符型），

FLD03=’’　　（8 位字符型）

那么，执行了下述语句之后

FLD01 CAT FLD02:0 FLD03

FLD03 就等于’ABC1 ’

而如果执行：

FLD01 CAT FLD02:1 FLD03

FLD03 就等于’ABC 1 ‘ （C 与 1 之间有一个空格）

4. 表示空格个数时，可以使用数字型的变量来表达，如

EVAL N=1

FLD01 CAT FLD02:N FLD03

5. CAT 操作码，其实也可以通过 EVAL 来实现部分。比如将 FLD01 与 FLD02 拼起来，中间无空格，赋值到 FLD03 中，也可以写做：

EVAL FLD03=FLD01 + FLD02

6. EVAL 操作码的优势，在于可以在一行语句中，就把多个字符拼在一起，如：

EVAL FLD05=FLD01+FLD02+FLD03+FLD04

如果要用 CAT 写，就要写多行，不简洁。

7. EVAL 操作码的不足之处，在于只能进行最简单的拼接，无法自动将字符后面的空格去除，如：

FLD01=’ABC ‘ （8 位字符型），

FLD02=’1’ （1 位字符型），

FLD03=’’ （8 位字符型）

在执行语句

EVAL FLD03=FLD01+FLD02

后，

FLD03=’ABC ‘

因为 FLD01 是 8 位，FLD03 也是 8 位，在操作时，不能去掉 ABC 后面的 5 位空格，此时后面再拼 FLD02 已无意义，所以 FLD03 仍是’ABC ‘

CHAIN {(N | E)} (Random Retrieval from a File)　按键值对文件记录进行查询定位

1. 基本语法：

举例，对逻辑文件 PFFHSL1 进行定位操作。逻辑文件 PFFHSL1，是以 FHS01 为键值，文件记录格式名叫 FMTFHS

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
FHS01	CHAIN	FMTFHS		17	18	

这个例子中，FHS01 应该是一个与文件 PFFHSL1 中键值（FLD01）类型、长度都相等的一个字符型变量，或者字符。

Factory 2 项中，要填文件的记录格式名，而不是文件名

HI 指示器表示是否查询到相应记录，查询不成功时，打开 HI 指示器

LO 指示器表示查询时，文件是否被锁。文件被锁时，打开 LO 指示器

也就是说：

*IN17=’0’，表示查询到了目标记录。

*IN17=’1’, *IN18=’0’，表示无相应的目标记录

*IN17=’1’, *IN18=’1’， 表示查询时，文件被锁（不确定有没有相应的目标记录）

2. 用修改方式声明的文件，当查询成功后，目标记录被锁定，其它程序无法以修改的方式定位到当前目标记录上。（但可以用只读的方式定位）
3. LO 指示器，仅于用修改方式声明的文件。对于只读方式声明的文件其实无意义
4. 如果用修改方式声明文件，但在 LO 处未填写指示器，且有程序锁住目标记录时，

当前程序会根据 PF 中定义的 WAITRCD(Maximum record wait time)参数，等待相应的秒数(如 WAITRCD 处为 10，即表示锁表时，等待 10 秒；如果填的是*IMMED，就表示不等待，一判断目标记录被锁就结束操作)；如果在等待时间内，对方仍未解锁，当前程序就会异常中断退出；如果 LO 处填写指示器，那么程序就不会中断退出，且 LO 指示器打开。

5. 当 FHS01 键值，在文件 PFFHSL1 中，对应有多条记录时（即键值不唯一），程序将会按照文件的内部记录号由小到大的顺序，定位到符合条件的第一条记录。
6. 当一个逻辑文件，KEY 值有多项时，可以使用 KLIST 操作码先定义一个组合键值，然后再用这个组合键值来进行 CHAIN 操作。需要注意，组合键值中，必须每一个成员字段都与目标记录所对应的字段相等，才能查询成功。（所以组合键值，通常使用 SETLL 定位较多）
7. 当用修改方式声明文件，但希望进行不锁记录的查询操作时，可以将 CHAIN 操作码写为 CHAIN（N），这个括号（N），就表示当前的查询定位，不对记录进行锁定操作（也就是用只读的方式来处理目标记录，所以只能取出目标记录的信息，不能做修改；如果要修改目标记录的话，还必须进行一个 CHAIN 操作）
8. **这一条要特别注意：**

   当没有填写 HI 指示器时，RPGLE 是允许编译通过的。而在某些情况之下（以前见过），运行程序时，如果 CHIAN 操作成功，找到了符合条件的记录时，没有任何问题；但如果 CHAIN 操作没有找到符合条件的记录时，实际上系统会按照键值的排序以及内部记录号，定位到下一条记录上，这时再取出来的数据，就统统都是下一条记录的数据。所以，除非有绝对的把握，CHAIN 操作肯定成功，否则就一定要养成填写 HI 指示器的良好习惯。

   （不好意思，这里好象又写错了，几经测试仍未测出这种情况，不过印象中的确出现过这种找到错误记录的情况，可能具体情况/条件记混了）。

CHECK {(E)} (Check Characters)　　检查目标变量中的字符

1. 基本语法：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
FLD01	CHECK	FLD02:2	N			42

语句 CHECK 的意思，是指字段 FLD02 中，从第二位字符开始，向右查找(含第二位字符)，是否包含有 FLD01 中没有的字符。

如果的确有在 FLD01 中没有的字符，将字符所在位置赋值到变量 N 中，同时打开 EQ 指示器；

如果从第二位字开始，向右读取的所有字符，都可以在变量 FLD01 中找到，那么 N 为 0，EQ 指示器处于关闭状态。

“FLD02:2”表示从变量 FLD02 的第二位开始，向右作比较。如果仅仅只是“FLD02”，那么就表示从变量 FLD02 的首位开始，向右查找

2. 实例

   假设 FLD01 为 8 位长字符，且当前值为’12345678’

   而 FLD02 为 5 位长字符，且当前值为’A3456’

   那么执行上述 CHECK 操作后，N=0， *IN42=’0’（从第二位开始，3456 都存在于变量 FLD01 中）

假设 FLD01 为 8 位长字符，且当前值为’12345678’

而 FLD02 为 5 位长字符，且当前值为’34ABDC’

那么执行 CHECK 操作后，N=3，*IN42=’1’ (即第三位“A”，不存在于变量 FLD01 中)

CHECKR {(E)} (Check Reverse)　反向检查目标变量中的字符

1. 基本语法：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
FLD01	CHECKR	FLD02:3	N			42

语句 CHECKR 的意思，是指字段 FLD02 中，从第三位字符开始，向左查找(含第三位字符)，是否包含有 FLD01 中没有的字符。（基本与 CHECK 类似，所不同的，就是 CHECK 是自左向右的查找；而 CHECKR 是自右向左查找）

如果有，将字符所在位置赋值到变量 N 中，同时打开 EQ 指示器；

如果从第二位字开始，向左读取的所有字符，都可以在变量 FLD01 中找到，那么 N 为 0，EQ 指示器处于关闭状态。

“FLD02:3”表示从变量 FLD02 的第三位开始，向左作比较。如果仅仅只是“FLD02”，那么就表示从变量 FLD02 的末位开始，向左查找

2. 实例

假设 FLD01 为 8 位长字符，且当前值为’12345678’

而 FLD02 为 5 位长字符，且当前值为’23A56’

那么执行上述 CHECK 操作后，N=0， *IN42=’0’（从第三位开始，向左，23 都存在于变量 FLD01 中）

假设 FLD01 为 8 位长字符，且当前值为’12345678’

而 FLD02 为 5 位长字符，且当前值为’BC3B45’

那么执行 CHECK 操作后，N=2，*IN42=’1’ (即第二位“C”,不存在于变量 FLD01 中)

3. 计算字符实际长段

根据 CHECKR 的这个操作码的用法，我们可以使用它来取出变量截去尾部空格后的实际长度：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
‘ ‘	CHECKR	FLD02	N			42

Factory 1 项，是一个空格字符。

N 表示的意思是：从变量 FLD02 的尾部，向前数第一个非空格字符，在 FLD02 中所处的位置。那么这个 N 当然就是变量 FLD02 的实际长度；

如果指示器未打开，那么说明整行都是空，实际长度为 0，也没错。

有趣吧。

CLEAR (Clear)　　清除内容

1. 基本语法

Factory 1	Operation	Factory 2	Result
	CLEAR		目标名

这个目标名，可以是程序中定义的结构、文件的记录格式名。

所谓文件的记录格式名，包括了程序中声明的磁盘文件、打印报表文件、屏幕文件

CLEAR 操作的意思，就是将目标所对应的所有变量/字段都赋上空值。

2. 对结构体的清空

关于结构体，后面会另设章节专门讲述，这里只说明对结构体初始化清空

Factory 1	Operation	Factory 2	Result
	CLEAR	*ALL	目标名

CLOSE {(E)} (Close Files)　关闭文件

1. 基本语法

Factory 1	Operation	Factory 2	Result
	CLOSE	目标文件名	

2. CLOSE 所对应的，是文件名，不是文件的记录格式名，要注意
3. CLOSE 操作码，仅适用于声明文件时，keyword 使用“USROPN”关键字的文件
4. 每一个 CLOSE 的操作，在之前都必须有 OPEN 文件的操作。也就是，文件必须打开了之后，才能关闭。不能关闭未打开的文件
5. 允许使用*ALL 变量，来表达关闭所有已打开的文件：

	CLOSE	*ALL	

COMMIT {(E)} (Commit)　　日志事务处理的确认操作

1. 基本语法

Factory 1	Operation	Factory 2	Result
	COMMIT		

2. 该操作码无其它参数，就是指对事务处理进行确认操作。
3. ILE 程序中，COMMIT 操作可随时进行，也允许在没有声明 COMMIT 类型的文件的情况下，仍进行 COMMIT 操作（对该进程这前的事务进行确认处理）f
4. 关于日志的确认操作，在后面会另设专门章节讲述。

COMP (Compare)　　比较

1. 基本语法：

将 Factory 1 与 Factory 2 进行比较。

当 Factory 1 > Factory 2 时，打开 HI 指示器；

当 Factory 1 = Factory 2 时，打开 LO 指示器；

当 Factory 1 < Factory 2 时，打开 EQ 指示器。

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
FLD01	COMP	FLD02		56	57	58

当 FLD01=2，FLD02=1 时，*IN56=’1’，*IN57=’0’，*IN58=’0’

当 FLD01=2，FLD02=2 时，*IN56=’0’，*IN57=’0’，*IN58=’1’

当 FLD01=1，FLD02=2 时，*IN56=’0’，*IN57=’1’，*IN58=’0’

字符也可以进行比较，好象是按字母排序，然后将内码相加，再比较（类似于 ASCII 码一样，不过不是特别清楚这个规律，所以一般没用）

坦白说，我觉得这个操作码有点无聊。

2.8.4.2　D--E

DEALLOC {(E | N)} (De-allocate Storage)

没用过，好象是对定义的指针型变量，对其分配地址空间之后，用这个操作码可以回收空间。

DEFINE (Field Definition)

根据已知的字段，来定义新字段，如下：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
*LIKE	DEFINE	FLD01	FLD02			

这句话的意思，就是说“象定义字段 FLD01 一样，定义字段 FLD02 的类型、长度”

这个字段 FLD01，必须是个已定义的字段/变量。

DELETE {(E)} (Delete Record)

删除当前记录，语法如下：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	DELETE	文件记录格式名				

这里，在做 DELETE 操作前，必须先定位到具体的记录上（使用 CHAIN、READ 等语句）；同时，文件在 F 行必须使用修改的方式来声明。

当文件定位到目标记录时，目标记录会被锁定，此时，目标记录无法被其它程序修改；如果执行了 DELETE 操作，自然会解锁，不过别的程序也找不到目标记录了（因为被删除）

实际使用中，通常并不会对文件中的记录进行物理删除，而是修改某个标识状态的字段的值，所以使用这条命令要慎重。

DIV {(H)} (Divide)　　数学运算—除

DIV 操作码，表示数学运算的“除”，常与操作码 MVR 一起来使用。

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
FLD01	DIV	FLD02	N			
	MVR		M			

上面两句话的意思，是说，用 FLD01 来除 FLD02，将商赋值到变量 N 中，将余数，赋值到变量 M 中。(N，M 都是数字型变量)

再具体一点，如果 FLD01 = 10,　FLD02 = 3，执行完这两句操作码之后

N = 3　　（10 / 3 的商）

M = 1　　（10 / 3 的余数）

DO (Do)　　循环

我最常用的循环方法之一，适用于已知循环次数的循环，与 ENDDO 搭配使用，每一个 DO，必须要配合一个 ENDDO。基本语法如下

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
1	DO	10				
	循环中处理的内容					
	ENDDO					

上面的意思，就是说固定循环 10 次。

不过呢，在实际使用中，我们常常需要知道当前循环到了第几次处理，这里，可以：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
1	DO	10	N			
	处理过程					
	ENDDO					

这个 N，是一个整数型变量（小数位为 0 的 P 型吧），它所表示的就是循环的次数。如第一次循环，N=1；第二次循环，N=2

所以，　　1　　DO　　1，就表示只循环一次，而不是表示死循环。

DOU {(M | R)} (Do Until)　　还是循环

也是循环，不过当满足 Extend Factory 2 项的条件之后，结束循环。如下：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	DOU	FLD01>FLD02				
	处理过程					
	ENDDO					

上面这句话，就是说当 FLD01 小于或等于 FLD02 时，进行循环；当满足条件，即 FLD01 大于 FLD02 时，结束循环。

在 RPGLE 的写法中，这个条件可以写得很复杂。当然，很复杂的时候，要记得多用括号，以免得逻辑判断与设计不一致。

DOUxx (Do Until)　　又是循环

这应该是 RPG 的写法，上面的循环也可以写做：

```
FLD01     DOUGT     FLD02
          ENDDO
```

不过，正如果前面所说的“ANDxx”操作码一样，RPGLE 的表示手法可以比 RPG 更直观，更好维护，所以这里也是一样的原理，有了“DOU”之后，我们就可以不用“DOUxx”了。

DOW {(M | R)} (Do While)　　循环，又见循环

这个循环的意思，与 DOU 正好相反，它的意思是满足 Extend Fatcory 2 项的条件时，才进行循环，不满足条件时，不循环

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	DOW	FLD01>FLD02				
	处理过程					
	ENDDO					

上面这句话，就是说当 FLD01 小于或等于 FLD02 时，不进行循环循环；当满足条件，即 FLD01 大于 FLD02 时，才进行循环。

在 RPGLE 的写法中，这个条件可以写得很复杂。当然，很复杂的时候，要记得多用括号，以免得逻辑判断与设计不一致。

注意：在实际使用过程中，我常常只使用 DO，DOW 这两种循环。而且使用 DOW 时，常常使其条件永远成立（即死循环，比如说 1=1 就永远成立，那么　DOW　　1=1 就是个死循环），然后在循环体中用 IF 语句判断何时退出循环（使用 LEAVE 语句），我认为这样会比较直观，而且很少会在逻辑上出错。还是那句话，我宁愿多写几行代码，而免去每次读到这里的时候都要想想逻辑上有没有问题的麻烦。

DOWxx (Do While)　　最后一个循环了

跟 DOU 与 DOUxx 的关系一样，有了 DOW 之后，就不需要 DOWxx 了，不多说了。

DSPLY {(E)} (Display Function)　　屏幕显示

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
FLD01	DSPLY					

就是在屏幕上，显示 FLD01 的内容。FLD01 可以是字符、数字型变量。也可以直接就是字符、数字。

DUMP (Program Dump)

没用过

ELSE (Else)　　逻辑判断语句

常见用法：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	IF	条件判断				
	处理内容一					
	ELSE					
	处理内容二					
	ENDIF					

不满足条件判断的时候，就执行 ELSE 与 ENDIF 之间的处理内容二；

满足条件判断时，执行 IF 与 ELSE 之间的处理内容一。

IF 与它最临近的 ELSE、ENDIF 配对；

同理，ELSE 也是与它最临近的 IF、ENDIF 配对。

注意多层 IF 嵌套时的逻辑判断，这个要自己多尝试，不讲了。

ELSEIF {(M | R)} (Else/If)　　逻辑判断语句

与 ELSE 类似，不过将条件判断语句也写在了同一行。

ELSEIF 不需要相应的 ENDIF 语句。即

IF　　条件判断 1

处理内容一

ELSEIF　条件判断 2

处理内容二

ELSE

处理内容三

ENDIF

当程序满足条件判断 1，将执行 IF 与 ELSEIF 之间的处理内容一；

当程序不满足条件判断 1，满足条件判断 2，执行 ELSEIF 与 ELSE 之间的处理内容二；

当程序连条件判断 2 的内容都不满足时，执行 ELSE 与 ENDIF 之间的内容处理内容三。

也就是说，这时的 ELSE，是和 ELSEIF 进行配对的，要注意。

ENDyy (End a Structured Group)

ENDIF　对应　IF、IFxx

ENDSR　对应　BEGSR

ENDDO　对应　DO、DOW、DOU、DOUxx、DOWxx

ENDSC　对应　CASxx

ENDSL　对应　SELECT

ENDFOR 对应　FOR

ENDSR (End of Subroutine)　　子过程结束语句

EVAL {(H | M | R)} (Evaluation)　赋值语句

1. 基本语法：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	EVAL	FLD01=FLD02				

赋值，使用 FLD01 的值，等于 FLD02 的值。FLD01 与 FLD02 的类型必须相同（即同为字符型，或同为数字型，但长度可以不同。

2. 当 FLD01、FLD02 同为字符型时，EVAL 语句其实等价于：

	EVAL	FLD01=*BLANKS				

MOVEL　　FLD02　　FLD01

即先清空 FLD01（当 FLD01 长度长于 FLD02，且有初始值时，这个操作就有意义了），然后再将 FLD02 左对齐赋值到 FLD01 中。所谓左对齐的含义，就是说将 FLD02 至左向右（含空字符），依次向 FLD01 字段中赋值。当 FLD02 长度大于 FLD01 时，右边的字符在赋值时将会被截去。

3. 当 FLD01、FLD02 同为数字型时，且 FLD02 的值，超过 FLD01 的长度时，EVAL 语句会直接报错，程序异常中断；
4. 当 FLD01、FLD02 同为字符型时，且 FLD01 的长度大于 FLD02 时，将会将 FLD02 的值左对齐赋值到 FLD01 中，并且将后面内容全部清空。
   比如 FLD01=’12345678’（8 位字符型），FLD02=’54321’（5 位字符型）
   执行了　EVAL　　FLD01=FLD02　　之后，FLD01=’54321　‘
5. 当 FLD01、FLD02 同为字符型，且 FLD01 的长度小于 FLD02 时，会将 FLD02 的内容左对齐，再赋值到 FLD02 中。
6. 字符的拼接：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	EVAL	FLD01=FLD02+FLD03+’AAA’+’BB’+FLD04				

FLD01、FLD02、FLD03、FLD04 都必须是字符。FLD02、FLD03 如果字符未满，有空格，EVAL 操作码不会将空格去掉。如果太长，可以分多行写，“+ ”在上在下都可。即

```
EVAL        FLD01 = FLD02 +
                    FLD03
```

等价于

```
EVAL        FLD01 = FLD02
                    + FLD03
```

7. 数学运算：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	EVAL	FLD01 =(((LD02+FLD03）* FLD04) / 2)　- FLD05				

加、减、乘、除都在了。注意多用括号。
如果运算公式太长，一行写不完，也可以分行写，运算符在上行、下行都可以。

8. 四舍五入：
   对于数字运算，EVAL(H)，是四舍五入的运算，如。

```
EVAL(H)     FLD01 = FLD02 / FLD03
```

而 EVAL，仅仅只是四舍，没有五入。

EVAL {(M | R)} (Evaluation Right Adjust)　不知道

EXCEPT (Calculation Time Output)　不知道

没用过

EXFMT {(E)} (Write/Then Read Format)　显示屏幕文件并等待输入

这个操作码的意思，是显示屏幕，并等待输入，语法为：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	EXFMT	记录格式名				

当程序执行到这一句时，表示显示“记录格式名”所对应的子文件（由 DSPF 定义，显示在屏幕上），同时等待用户进行输入操作。

当用户输入预定义的热键（F3，F4 等等）、或执行输入操作后(按下输入键)，程序才会继续向下执行。

EXSR (Invoke Subroutine)　　执行子过程

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	EXSR	子过程名				

子过程名必须存在。没有特别的

EXTRCT {(E)}(Extract Date/Time)　取日期、时间型变量中的具体内容

首先，必须要有一个日期或时间型变量，这个可以在 D 行定义，然后使用该变量中有值，比如

```
D FLD01      S                D
D FLD02      S                T
C            MOVE   *DATE     FLD01
C            TIME             FLD02
```

这样，就有了一个日期型变量 FLD01，一个时间型变量 FLD02，且两个变量中都有值。再假设变量 FLDYEAR 为四位字符型、FLDHOUR 为两位数字型

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	EXTRCT	FLD01:*Y	FLDYEAR			
	EXTRCT	FLD02:*H	FLDHOUR			

这时，FLDYEAR 表示当前的年份；FLDHOUR 表示当前的小时

所以说，EXTRCT 操作码对应的目标变量，可以是字符型，也可以是数字型，但长度一定要与预期的目标长度相等（比如年份可以是字符，也可以是数字，但长度必须为 4 位；月份必须为 2 位，依此类推）

而冒号后面所跟的参数，与 ADDDUR 操作码一样，如下：

对应于日期型变量：

*D 表示天，也可用*DAYS

*M 表示月，也可用*MONTHS

*Y 表示年，也可用*YEARS

对应于时间型变量

*H　表示小时，也可用*HOURS

*MN 表示分钟，也可用*MINUTES

*S　表示秒，　也可用*SECONDS

2.8.4.3　F--N

FEOD {(E)} (Force End of Data)

没用过

FOR　(For)　又是循环

FOR 操作码，也是一种指定次数的循环，与“DO”循环有异曲同工之妙。

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	FOR	N=1　TO　10				
	处理语句					
	ENDFOR					

等价于

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
1	DO	10	N			

处理语句
ENDDO

都是循环 10 次，并且将当前循环次数赋值到数字型变量 N 中

FORCE (Force a Certain File to Be Read Next Cycle)

没用过

GOTO (Go To)　　跳转语句

GOTO，跳转语句。跳转到指定的标签处，需要用“TAG”语句来定义，如：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	处理语句 1					
	IF	条件判断				
	GOTO	FHSTAG				
	ENDIF					
	处理语句 2					
FHSTAG	TAG					

在这里，执行完处理语句 1 时，如果“条件判断”成立，将会直接跳至“FHSTAG TAG”处，不执行处理语句 2。

主过程不能跳转至子过程；（主过程我用来表示是与子过程相对应的词，指 C 行顺序第一行，至程序结束语句之间的语句）

子过程不能跳转至其它子过程，只能在本子过程内部跳转，或跳转到主过程；

技术可以使用跳转语句来实现循环，不过不建议。

IF {(M | R)} (If)　　条件判断

条件判断语句，必须与 ENDIF 一起使用

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	IF	((FLD01=FLD02) OR (FLD01>FLD03)) AND (FLD03=FLD04)				
	处理语句					
	ENDIF					

当条件判断成立时，才会执行下面的处理语句。

条件判断可以使用括号，OR，AND，来表达复杂的逻辑关系

IF 必须有对应的 ENDIF，所以建议写程序时，写无 IF 就加个 ENDIF，然后再来写中间的处理语句，以免遗漏。

IF 语句，还可以这种用法：

```
IF      *IN(20)
ENDIF
```

这时，等价于

```
IF      *IN20='1'
ENDIF
```

不过，暂时没看出这种写法有什么实用性。

IFxx (If)　　条件判断

与 ANDxx 类似，这是 RPG 的语法，一行代码只能表示一个条件，如

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
FLD01	IFEQ	FLD02				
FLD01	ORGT	FLD03				

处理语句
ENDIF

与 AND 和 ANDxx 类似，由于 IF 语句在表达复杂的逻辑关系时更直观，所以通常我只使用 IF，而不是 IFxx 了。

IN {(E)} (Retrieve a Data Area)　　对数据区？的操作

没用过，看书上的意思，是对数据区？的操作。数据区有什么具体优势，实用性如何，是否可被普通的程序替代，都希望有人来补充。

ITER (Iterate)　循环中，不处理此语句之后的语句

怎么解释都费劲，直接举例吧：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
1	DO	10	N			
	处理语句 1					
	IF	条件判断				
	ITER					
	ENDIF					
	处理语句 2					
	ENDDO					

等价于

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
1	DO	10	N			
ITERTAG	TAG					
	处理语句 1					
	IF	条件判断				
	GOTO	ITERTAG				
	ENDIF					
	处理语句 2					
	ENDDO					

不知道这个例子能不能让人明白，都是在说当条件判断成立时，就不执行处理语句 2，直接从循环头开始处理。

ITER 语句执行后，程序相当于跳转至循环体中的第一句，重新向下运行（即执行循环）。

当有多层循环嵌套时，ITER 仅作用于最里层的循环。

当循环中执行了子过程时，子过程中不能对该循环进行 ITER 操作。也就是说：DO，ENDO，ITER 这些语句必须在一块（即要么都在主过程中，要么都在同一个子过程中）

KFLD (Define Parts of a Key)　　定义组合键值中的字段

这个其实应该与 KLIST 一起来讲，因为这两个操作是在一起使用的。

一个 PF 对应的逻辑文件 LF，可能会有多个 KEY 值，那么，我们要根据多个 KEY 值对于该逻辑文件进行查询定位操作时，就需要使用 KLIST、KFLD 这两个操作码：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
FHSKEY	KLIST			//定义组合键值 FHSKEY		
	KFLD		FLD01	//组合键的首个字段为 FLD01		
	KFLD		FLD02	//组合键的第二个字段 FLD02		

组合键值的使用方式，就是将组合键视为一个变量，进行各种操作，如：

FHSKEY	CHAIN	FMTFHS	

或

FHSKEY SETLL FMTFHS

意思，就是说按组合键值，查询定位到逻辑文件 PFFHSL3 中去。

这里要注意，PFFHSL3 的键值是 FHS01、FHS02（即 PF 文件中的字段），而程序中的组合键值为 FLD01、FLD02 两个程序中定义的变量，只要类型、长度与键值 FHS01、FHS02 相等即可。同时，对文件进行查询定位操作后，在没有对 FLD01、FLD02 赋值之前，这两个字段的值是不会发生变化的（即不会更改组合键的键值）；

但是，组合键中的字段，也可以直接定义为 PF 文件中的字段，如

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
FHSKEY	KLIST					
	KFLD		FHS01			
	KFLD		FHS02			

此时，可以将 PF 文件中的字段 FHS01、FHS02 先视为两个普通的变量，进行赋值，然后用组合键 FHSKEY 对 PFFHSL3 进行查询定位；当执行完 CHAIN，或 READ 语句后，因为读取到了文件中的数据，所以 FHS01、FHS02 的值将发生变化，也即是组合键 FHSKEY 的键值发生了变化。

基于此，所以通常我不会将组合键的字段设置为查询文件中的字段，而是定义为几个独立的临时字段，以免需要过于频繁的考虑键值的更改。

KLIST (Define a Composite Key)　　以上面的 KFLD 操作码

LEAVE (Leave a Do Group)　退出当前循环

退出当前循环，语法和注意事项基本与“ITER”相同：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
1	DO	10	N			
	处理语句 1					
	IF	条件判断				
	LEAVE					
	ENDIF					
	处理语句 2					
	ENDDO					

等价于

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
1	DO	10	N			
	处理语句 1					
	IF	条件判断				
	GOTO	LEAVETAG				
	ENDIF					
	处理语句 2					
	ENDDO					
LEAVE	TAG					

这两个例子都是在说当条件判断成立时，就不执行剩余的循环体，直接跳出循环。

LEAVE 语句执行后，程序相当于跳转至 ENDDO 循环结束处，然后向下执行 ENDDO 之后的循环外的执行语句。

当有多层循环嵌套时，LEAVE 仅作用于最里层的循环。

当循环中执行了子过程时，子过程中不能对该循环进行 LEAVE 操作。也就是说：DO，ENDO，LEAVE 这些语句必须在一块（即要么都在主过程中，要么都在同一个子过程中）

LOOKUP (Look Up a Table or Array Element)　　对数组的查询定位

假设已定义数组 DIMDATA，每条记录四位长，共三条记录的数组，且数组中已赋值如下：

DIMDATA(1) = '0001'
DIMDATA(2) = '0002'
DIMDATA(3) = '0003'

那么，下面的语句就是对数组的查询定位操作：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	EVAL	N=1				
'0002'	LOOKUP	DIMDATA(N)				54

首句 N=1，表示从第一条记录开始查询；如果 N=2，即表示从第 2 条记录开始查询；如果 N 大于数组定义时的总记录数（如 N=4），则下面的 LOOKUP 语句会报错，跳出程序；

第二句，表示查询　数组 DIMDATA 中，内容为'0002'的，是第几条记录；其中，54 为 EQ 指示器。当找到记录时，打开指示器，*IN54 = '1'；当未找到记录时，　指示器处于关闭状态，*IN54='0'。与操作码 CHINA 的指示器含义刚好相反；

在这个例子中，执行完 LOOKUP 语后，*IN54='1'，　N = 2（即数组 DIMDATA 中，有内容为'0002'的记录，是第二条记录。

当数组中有多条内容相同的记录时，N 为顺序查询到的第一条记录

当数组中无相应记录时，*IN54='0'，　N = 初始值，此例中，即 N = 1

MHHZO (Move High to High Zone)

没用过

MHLZO (Move High to Low Zone)

没用过

MLHZO (Move Low to High Zone)

没用过

MLLZO (Move Low to Low Zone)

没用过

MOVE　(Move)　　赋值语句

1. 基本语法：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	MOVE	FLD01	FLD02			

FLD01、FLD02 可以是不同的类型。即允许一个为字符，一个为数字

2. 允许在 MOVE 操作中，对 Result 字符进行定义。
3. MOVE 操作码，使用**右对齐赋值**。所谓右对齐，即是将 Factory 2 的最右边一个字符，赋值到 Result 的最右边的一个字符；然后再将 Factory 2 向左取一位字符，赋值到 Result 从最右边起，左移一位的字符是，依此类推。直到取完 Factory 2 的值，或 Result 已赋满值。
4. 由上述其实已可看出，MOVE 操作码在对字符赋值时，使用的是覆盖方式赋值。即如果 FLD02 将在赋值的字符处，原来有值，将会将原来的值覆盖；如果已有值时，未执行赋值操作，将仍然保留原值。
5. 当 FLD01、FLD02 都为字符型：

如果 FLD01 的长度为 5 位，FLD02 的长度为 8 位。FLD01 的内容为'12345'，FLD02 的内容为'ABCDEFGH'，当执行了 MOVE 操作后，FLD02 的内容为'ABC12345'，可以

看到，后 5 位由’DEFGH’变成了 FLD01 的’12345’，而前三位仍然保留原来的值’ABC’；**要注意的是，在字符型变量中，空字符也会参与赋值**。

如果 FLD01 的内容为’12　‘，即后三位是空时，那么执行 MOVE 操作后，FLD02 的值将会是’ABC12　‘。看到了没有，最后三位也与 FLD01 一样，是空；而前三位 FLD01 的值保持不变，仍为’ABC’。

反之，当 FLD01 的长度为 8 位，’12345678’,而 FLD02 的长度为 5 位’ABCDE’时，执行完 MOVE 操作后，FLD02 的值为’45678’，即从最右边开始，向左赋值；

MOVE 操作表示右对齐，空也算字符。

6. 数字赋值到字符的处理：

当 FLD01 为数字（5，2），FLD02 为字符（8）时，MOVE 操作将会把数字的小数点去掉，将数字转换为字符，右对齐赋值到字符变量中。数字型变量如果未满，前面都视为空，而不是 0；

如果 FLD01 为 123.45，FLD02 为 ’ABCDEFGH’，赋值之后，FLD02=’ABC12345’，首三位保持 FLD02 原有的值。

如果 FLD01=123.45，FLD02 原来为空，赋值之后，FLD02=’　12345’；（字符型变量首三位保持原来的空值）

如果 FLD01=3.45，FLD02 原来为空，　赋值之后，FLD01=’　345’。（数型变量未满足，前面也视为空，而不是 0）

如果数字的长度大于字符的长度，系统将会自动从左边开始，将大于的部分截去。如 FLD01 长度为 10，2，等于 12345678.90 时，赋值后，FLD02 会等于’34567890’

7. 字符赋值到数字的处理：

当 FLD01 为字符(8)，FLD02 为数字(5,2)时，MOVE 操作会将字符右对齐赋值到数字中，如果字符后面有空，系统会自动做补零处理；

如果 FLD01 为’12345678’，FLD02 为 0，赋值之后，FLD02=456.78；

如果 FLD01 为’12345　‘，即后三位为空时，赋值之后，FLD02=450.00，自动补 0；

当数字长度大于字符的长度，如 FLD02 为(10,2)时，数字最高位在赋值时，将会保持原值，如 FLD02=12345678.90，FLD01=’87654321’，赋值之后，FLD02=12876543.21，注意最高位的 12 保持未变

所以通常，字符转数字时，长度最好保持一致，以免得赋值时有意外。

8. 日期、时间型变量与字符、数字的转换

日期型变量与字符的转换，会带上分隔符。如果日期型变量为’2007-02-12’时，字符型变量也需要定义为 8+2 位，当执行完赋值操作时，字符型变量即为’2007-02-12’，注意，是有分隔符的；时间型变量也一样，只是长度要需要 6+2 位。反之，当字符型变量赋值到日期型变量中，也是需要有这个分隔符的；

日期型变量与数字的转换，将会自动去掉分隔符（非常好吧），如果日期型变量为’2007-02-12’时，赋值到一个 8 位的数字型变量，该变量就等于 20070212；反之，数字赋值到日期型变量中时，也是需要 8 位纯数字；时间型变量也一样处理，只要长度需要为 6 位。

MOVEA {(P)} (Move Array)　　数组赋值

该操作码仅作用于数组。

假设已定义数组 DIMDATA，每条记录二位长，共三条记录的数组，且数组中已赋值如下：

DIMDATA(1) = ‘01’

DIMDATA(2) = ‘02’
DIMDATA(3) = ‘03’

当执行完 MOVEA 操作后：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	MOVEA	*BLANK	DIMDATA			

DIMDATA 中所有的要素都变成了空值。

MOVEA 还可以将一个变量拆分赋值到数组中，仍是上例，假设有一个 6 位字符型变量 FLD01，其值为’ABCDEF’，那么执行语句：

MOVEA FLD01 DIMDATA

后，数组中的值将是如下：

DIMDATA(1) = ‘AB’
DIMDATA(2) = ‘CD’
DIMDATA(3) = ‘EF’

当变量的长度，小于（数组单条记录长度*总记录数）时，超出部分的数组的值，保持原值。仍是上例，假设 FLD01 为 4 位长的字符变量，值为’ABCD’时，执行 MOVEA 操作后，数组中的值如下：

DIMDATA(1) = ‘AB’
DIMDATA(2) = ‘CD’
DIMDATA(3) = ‘03’

注意，最后一个记录 DIMDATA(3)的值仍保持原’03’不变，没有被清空。

MOVEL {(P)} (Move Left)　　左对齐赋值

语法与 MOVE 操作码相同，所不同的是**对齐方式为左对齐**，即是将 Factory 2 的最左边一个字符，赋值到 Result 的最左边的一个字符；然后再将 Factory 2 向右取一位字符，赋值到 Result 从最左边起，右移一位的字符，依此类推。直到取完 Factory 2 的值，或 Result 已赋满值。

在进行长度不同的字符之间赋值时，可按照上面这个原则，根据 MOVE 操作码中的案例，想想不同情况的结果，这里不做一一列举。

要注意的是，当使用 MOVEL，由字符转为数字时，后面的空格会自动补 0，如’123‘（后面 5 位都为空），转到数字型变量中，就变成了 123000.00。因为大部分情况下，我们定义的标准，都是字符型左对齐，数字型右对齐。所以此时字符转数字要注意，以免无端扩大倍数。

MULT {(H)} (Multiply)　　数学运算—乘

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
FLD01	MULT	FLD02	FLD03			

等于

EVAL FLD03 = FLD01 * FLD02

可以对当前行对 FLD03 进行定义；

Factory 1 可以不填写；

必须保证 FLD03 的位数足够大，否则超长时，程序会报错异常退出。

MVR (Move Remainder)　数学运算—除法运算取余

这个操作码，在讲操作码 DIV 时，已讲过：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ

FLD01	DIV	FLD02	FLD03
	MVR		FLD04

当 FLD01=11，FLD02=时 4，

FLD03=2　　（商）

FLD04=3　　（余）

NEXT {(E)} (Next)

没用过

2.8.4.4　O--R

ON-ERROR (On-Error)

没用过

OPEN {(E)} (Open File for Processing)　打开文件

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	OPEN	文件名				

OPEN 后面的目标，必须是在当前程序中已声明的文件名（不是文件的记录格式名），而且在 OPEN 操作之后，在程序结束之前之前，必须有对应的 CLOSE 操作。

使用 OPEN 操作，文件在声明时，必须使用 USROPN 关键字（详见 D 行说明）。

ORxx (Or)　逻辑判断—或

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
FLD01	IFGT	FLD03				
FLD01	OREQ	FLD02				

等价于

	IF	FLD01>FLD03 OR FLD01=FLD02				

与 IF、IFxx，AND、ANDxx 类似，RPGLE 的写法 OR，比 RPG 的写法 ORxx 要灵活，而且可以用来表达一些复杂的逻辑关系。有鉴于此，所以通常 IF 语句中，我会以 OR 为主，基本不用 ORxx。如果在编程序方面，公司/项目组无硬性要求，那我觉得还是少用 ORxx 吧，总觉得这种写法的逻辑关系看起来不直接，尤其是有很复杂的 AND,OR 时。

OTHER (Otherwise Select)　分支语句的判断

与分支语句 SELECT 一起使用，表示不符合上述所有条件时的操作，如下：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	SELECT					
	WHEN	条件判断 1				
	处理语句 1					
	WHEN	条件判断 2				
	处理语句 2					
	OTHER					
	处理语句 3					
	ENDSL					

在这个例子中，当满足条件判断 1 时，运行处理语句 1，运行结束后跳至 ENDSL 处；如果不满足条件判断 1，则程序继续向下执行，判断是否满足条件判断 2。

当满足条件判断 2 时，运行处理语句 2，跳至 ENDSL；当不满足

当不满足条件判断 2 时，程序继续向下执下，当读到 OTHER 操作码时，无条件运

行处理语句 3（即当程序当前不满足以上所以条件判断时，则执行 OTHER 之后的语句。

处理语句允许有很多句；

条件判断可以写得很复杂，也允许对不同的字段进行判断；比如说 C 语言也有分支语句 switch，但是这个语句只能对一个字段进行分支判断，ILE 语言与它不同，允许对不同的字段进行判断

就我目前掌握的测试情况，上述的 SELECT—WHEN--OTHER—ENDSL，其实也可以写做：

```
IF          条件判断 1
处理语句 1
ELSEIF   条件判断 2
处理语句 2
ELSE
处理语句 3
ENDIF
```

即 WHEN 与 ELSEIF 是类似的，这样说，应该可以明白了吧。

总之，SELECT—ENDSL 是一个很好用的语法，尤其是在表示很多不同的分支处理时。

OUT {(E)} (Write a Data Area)

没用过，讲数据域的。

PARM (Identify Parameters)　定义入口参数

Factory 1	Operation	Factory 2	Result	HI	LO	EQR
*ENTRY	PLIST					
	PARM		FLD01			

关于具体内容讲解，详见前面所说“入口参数”一章。

允许做为入口参数的有：普通变量、结构变量、数组变量

关于 PARM、PLIST，还有一种在 Factory 1， Factory 2 也填写变量或指示器的用法，不过我不知道它具体表示什么意思，也不知道该怎么用。请用过的来补充。

PLIST (Identify a Parameter List)　同上

POST {(E)} (Post)

没用过

READ {(N | E)} (Read a Record)　读取记录

1. 基本语法：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	READ	文件记录格式名			45	46

READ 后面跟的，必须是声明的文件记录格式名；

LO 指示器表示锁表指示器，当在指定的时间（CHGPF，WAITRCD 项可看到），需要读取的记录仍被锁，将会打开 LO 指示器，即*IN45=’1’；

EQ 指示器为是否读到指示器。当未读到任何记录时，打开 EQ 指示器，即*IN46=’1’

2. 当文件在程序中，是用只读的方式声明时，READ 操作并不会造成锁表；

如果文件在程序中是用修改的方式声明，READ 操作成功后，该记录被锁；直到执行解锁操作（UNLOCK，或 UPDATE），或 READ 该文件的其它记录，才会解锁

如果文件是用修改的方式声明，但希望 READ 操作不锁表时，那么就用 READ（N），即

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	READ(N)	文件记录格式名			45	46

这样读文件，就不会锁记录，但是同时也不能修改记录。如果需要修改记录，那么在修改之前（包括对文件字段赋值之前），还必须再对该记录进行一次定位操作（比如 CHAIN、READ 语句均可）。也就是说，如果要修改记录，必须先锁住当前记录（很合理吧）

3. 当执行 READ 操作时，程序是根据游标当前在文件中所指向的位置，顺序读取下一条记录。关于游标是如何指向，还不是一个很简单的问题，所以将会在下一章“数据库相关知识”中具体讲解。
4. 执行 READ 操作时，允许声明的文件没有键值。（即 PF 文件）

READC {(E)} (Read Next Changed Record)

没用过，读下一次修改过的记录？

READE {(N | E)} (Read Equal Key)　　读取键值相等的记录

语法与 READ 操作码大致一样，这里不再重复，只说不同的：

假设程序中已声明逻辑文件 PFFHSL3（键值为 FHS01+FHS02）

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
FHSKEY	KLIST					
	KFLD		FLD01			
	KFLD		FLD02			
FHSKEY	SETLL	FMTFHS				
	DOW	1=1				
FHSKEY	READE	FMTFHS			15	
	IF	*IN15=’1’				
	LEAVE					
	ENDIF					
	ENDDO					

这段话的意思，就是定义组合键值 FHSKEY，然后根据这个 FHSKEY 在逻辑文件 PFFHSL3 中去定位，循环读取 PFFHSL3 中，FHS01、FHS03 与 FLD01、FLD02 相等的记录。当读取记录结束，或键值不等时，退出循环（*IN15 是 EQ 指示器）。如果将 READE 操作码换成 READ 操作码的话（当然，Factory 1 处也就不能有值），就没有“**键值不等时退出循环**”这一层意思，只是读不到记录时就退出循环，但有时我们使用逻辑文件，仅仅是需要它的排序，而不需要读不到键值相等的记录就退出循环。所以说，使用 READ 操作码，还是 READE 操作码，需要根据实际的要求来决定。

以上的 Factory 1 处填写值的系统处理，当 READE 操作码在 Factory 1 处未填写值时，系统实际上是将当前的值与读到的上一条记录的关键字进行比较，而不是与 SETLL 时的键值做比较（**读第一条记录不做比较！**），如果键值不等时，置 EQ 指示器为 1。。也就是说，如果没有与 FHSKEY 键值相同的录，那么系统并不是直接找开 EQ 指示器，而是会一直保持正常地往下读，直到找到与读到的第一条记录关键字不同的记录，才会打开 EQ 指示器，所以要注意。

READP {(N | E)} (Read Prior Record)　　读取记录—游标上移

简单来说，READ、READE 操作时，游标在数据文件中，是下移的；即读完第一条记录，游标指向第二条记录；读完第二条记录，游标指向第三条记录，依此类推，直至最后一条记录。但 READP 则正好相反，游标是上移的，即读完第三条记录后，游标指向第二条记录；读完第二条记录后，游标指向第一条记录，直至读完第一条记录。

一般来说，用 READ、READE 的概率会比 READP、READPE 的概率高得多，不过在某些情况下，使用 READP 操作，又的确会很省事，这个一时间想不起例子来，大家可在编程序时多实践。

READPE {(N | E)} (Read Prior Equal)

虽然我没用过，但猜想它应该就是指游标上移，按键值去读取文件。与 READP 的关系，就类似于 READE 与 READ 的关系。

REALLOC {(E)} (Re-allocate Storage)

没用过

REL {(E)} (Release)

没用过

RESET {(E)} (Reset)

将数据结构赋值成为初始值。

注意是初始值，不是清空。

如定义结构：

```
D FHSDS          DS
D   FHS01             10   INZ（'ABCD'）
D   FHS02             5    INZ（'EFGH'）
```

那么，不管对该结构如何赋值，当执行语句：

```
C                RESET                FHSDS
```

之后，FHS01 将会变成'ABCD，FHS02 将会变成'EFGH'，即恢复成为初始值。

RETURN {(H | M | R)} (Return to Caller)

RETURN 是程序结束。

在前面，“简单的程序流程”中，我们讲过，“SETON　LR” 与 RETURN 这两句话一起，做为程序的结束。这里，再详细解释一下两者之间的区别，以及关系：

如果不写 RETURN，只写“SETON　LR”，程序执行完最后一句之后，将会再从第一句开始执行，造成死循环。在简单的程序流程这个例子中，程序原来只想修改读到的第一条记录，而**如果没有 RETURN 的话，将会把所有的记录都修改掉，直到最后找不到可修改的记录，然后系统报错，异常中断**。（这种离奇的现象现在又测试不到了，可能是当时写错程序了？把 F 写成了 P？不管它，当是我写错了，总之 RETURN 是表示程序结束，没有 RETURN，主程序无可执行的语句时，它也会结束；如果 RETURN 出现在主程序的中间，那么 RETURN 后面的语句将不会执行）

如果只写 RETURN，不打开指示器*INLR，根据 blogliou 所说 “*程序不会强制将内存中的数据写到磁盘中。400 缺省的是 BLOCK 输出，即数据记录满一个 BLOCK 块时才会将这一组记录写到磁盘上。那么如果这时 BLOCK 没满，数据信息不会立刻写到磁盘上。之后有其它作业用到该文件，读取的数据就不完整。*”

但如果文件有唯一键字，或记录日志，必须同步写时，其实 BLOCK 实际被忽略，也就是此时不会有错。目前我们用的是 MIMIX 备份，客户实际上将所有的文件都列入日志，这时不写也不会出现上述错误。但为避免一些潜在的问题，养成良好的编程风格，建议将 SETON LR 与 RETURN 一同，做为程序结束的标志。当然，如果某个程序频繁被调用，且不涉及文　　操作时，可考虑不打开指示器*INLR，仅用 RETURN 作为结束，这样程序不会被 PURGE 出内存，可提高调用效率。

如果没写 RETURN，也没有打开指示器*INLR，在编译时，系统将会报 40 级错，说找不到程序结束的语句，所以大可放心。

ROLBK {(E)} (Roll Back)

1. 基本语法

   Factory 1　Operation　Factory 2　Result

   ROLBK
2. 该操作码无其它参数，就是指对事务处理进行回滚操作。
3. ILE 程序中，ROLBK 操作可随时进行，也允许在没有声明 COMMIT 类型的文件的情况下，仍进行 ROLBK 操作（对该进程这前的事务进行确认处理）f
4. 关于日志的确认回滚操作，在后面会另设专门章节讲述。

2.8.4.5　S--Z

SCAN {(E)} (Scan Character String)　扫描字符串

扫描字符或字符串 Factory 1 在目标字符串 Factory 2 中是否存在

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
FLD01	SCAN	FLD02	N			26

FLD01 可以是字符，也可以是字符变量；可以是一位长，也可以是多位长。

当 FLD01 在 FLD02 中存在时，EQ 指示器打开，即*IN26=’1’，同时将 FLD02 中的起始位置，赋值给 N；

当 FLD01 在 FLD02 中不存在时，EQ 指示器保持关闭状态，即*IN26=’0’，同时 N=0

允许从 FLD02 中的指定位置开始检查：

FLD01	SCAN	FLD02:2	N			26

如上句，即表示从 FLD02 的第 2 位，开始扫描。

在实际使用中，比如说我们判断某个字符是否为数字，就可以先定义一个 0—9 的常量，然后将要判断的字符去 SCAN 一下这个常量

SELECT (Begin a Select Group)　分支语句

在操作码“OTHER”中讲过，为方便读者，列出简单语法如下：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	SELECT					
	WHEN	条件判断 1				
	处理语句 1					
	WHEN	条件判断 2				
	处理语句 2					
	OTHER					
	处理语句 3					
	ENDSL					

要注意，SELECT 操作码，必须有对应的 ENDSL 操作码，否则编译无法通过。

SETGT {(E)} (Set Greater Than)　定位操作—大于

举个例子吧，假设文件中有一个字段，是标识顺序号的，1、2、3、4。即该字段为 1，表示第一条记录，该字段为 2，表示第 2 条记录。那么：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ

```
2           SETGT       文件记录格式名
            READ        文件记录格式名
```

这个 READ 操作，READ 到的，是第 3 条记录。也就是说，SETGT 操作码，会将游标定位到大于键值的第一条记录前。

在实际使用中，如果我们是按逻辑文件读取，而且读了一条记录之后，对其键值相同的记录都不需要再读取时，就可以用 SETGT，不过需要注意，Factory 1 项，需要是与键值相同的变量，即如果文件是使用多个字段做为键值，那么我们也需要先定义一个组合键值的变量，然后 Factory 1 处填写这个组合键值的变量名。

当声明文件的键值有多项时，Factory 1 项的键值，允许小于文件的键值，但顺序必须一致。即声明的文件如果键值为：FHS01、FHS02、FHS03，那么我们在程序中定义三个类型与之相同的变量 FLD01、FLD02、FLD03，以下写法都是有效的

```
FLDKEY      KLIST
            KFLD        FLD01
            KFLD        FLD02
            KFLD        FLD03
FLDKEY      SETGT       文件记录格式名

FLDKEY      KLIST
            KFLD        FLD01
            KFLD        FLD02
FLDKEY      SETGT       文件记录格式名

FLD01       SETLL       文件记录格式名
```

SETLL {(E)} (Set Lower Limit)　定位操作—小于

语法与 SETGT 相同，含义与 SETGT 不同。SETLL 操作码，会将游标定位到与键值相等的第一条记录之前，仍是上例，如果是

```
2           SETLL       文件记录格式名
            READ        文件记录格式名
```

那么 READ 操作码读到的记录，就是第 2 条记录，看到了吧，和 SETGT 不同。

SETLL 操作码还可以用来简单判断当前键值是否存在有记录，以 PFFHSL3 为例（键值为 FHS01、FHS02）

```
Factory 1   Operation   Factory 2       Result          HI  LO  EQ
FHSKEY      KLIST
            KFLD                        FLD01
            KFLD                        FLD02
            EVAL        FLD01='01'
            EVAL        FLD02='02'
FHSKEY      SETLL       文件记录格式名                          44
```

当文件中有相应记录时，EQ 指示器打开，即*IN44='1'

当文件中无相应记录时，EQ 指示器关闭，即*IN44='0'（与 CHAIN 正好相反，要注意）

而在这种用法中，SETLL 与 CHAIN 的区别在于，CHAIN 是定位读取了记录，而 SETLL 仅仅只是判断该记录是否存在。所以用 SETLL 操作，不能修改记录，也无法取出记录的值。只能判断记录是否存在。如果要修改记录，或取出记录的值，还需要有一个读取定位的操作，

如 READ，或 READE、READP 等（最常用的，应该就是 READ 操作）

SETOFF (Set Indicator Off)　　关闭指示器

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	SETOFF			10	11	12

等价于

```
EVAL    *IN10='0'
EVAL    *IN11='0'
EVAL    *IN12='0'
```

在 SETOFF 这个操作码中，指示器填在 HI、LO、EQ 哪里都没关系，都是表示要被关闭的指示器

SETON (Set Indicator On)　　打开指示器

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	SETON			10	11	12

等价于

```
EVAL    *IN10='1'
EVAL    *IN11='1'
EVAL    *IN12='1'
```

在 SETON 这个操作码中，指示器填在 HI、LO、EQ 哪里都没关系，都是表示要被关闭的指示器

SHTDN (Shut Down)

没用过

SORTA (Sort an Array)

没用过

SQRT {(H)} (Square Root)　开方

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
9	SQRT	3	N			

这时，N=3（因为 3 的平方为 9）

9、3 都可以是数字型变量，或者直接是数字

SUB {(H)} (Subtract)　减法操作

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
FLD01	SUB	FLD02	FLD03			
	SUB	FLD02	FLD03			

看过前面的 ADD、MULT 操作码，这里不用解释也应该明白是什么意思了吧。那就不多说了。

SUBDUR {(E)} (Subtract Duration)　日期相减

1. 减日期

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
FLD01	SUBDUR	N:*Y	FLD02			

表示将日期型变量 FLD01 减去 N 年，赋值到日期型变量 FLD02 中；

N 可以是一个数字型变量，也可以就是一个数字，N 允许为负数

*Y，*M，*D（还有其它的参数值，可见 ADDDUR，其中有详细解释）

2. 判断两个日期型变量之间的天/月/年数

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
FLD01	SUBDUR	FLD02	N:*D			

这时，N 做为一结果变量，表示日期型变量 FLD01 与 FLD02 之间的天数

SUBST {(P | E)} (Substring)　　取字符/字符串

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
2	SUBST	FLD01:3	FLD02			

表示从字段 FLD01 的第 3 位开始，取 2 位，左对齐赋值到字段 FLD02 中。

要求字段 FLD01 的长度必须大于或等于 3+2 位，否则程序会报错。

可以尝试用%SUBST 语句，也是等价的，如下

EVAL　　FLD02=%SUBST(FLD01:3:2)

表示的是同样的意思。

起始位数 3，取的长度 2，在两种写法之下，都可以使用数字型变量来表达。

相比较之下，%SUBST 还有一种用法，就是对字符的指定位置赋值，这个就厉害了：

EVAL　　%SUBST(FLD02:3:2)='01'

看到了吧，这句话就是说，使字段 FLD02 的第 3、4 位（即从第三位开始，两位长）等于“01”

TAG (Tag)　　定义标签，与 GOTO 同用

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
FHSTAG	TAG					

TEST {(D | T | Z | E)} (Test Date/Time/Timestamp)

没用过

TESTB (Test Bit)

没用过

TESTN (Test Numeric)

没用过

TESTZ (Test Zone)

没用过

TIME (Time of Day)　--取当前系统时间

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	TIME		FLD01			

FLD01 可以是时间型或数字型变量

UNLOCK {(E)} (Unlock a Data Area or Release a Record)　　解锁

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	UNLOCK	文件记录格式名				

UNLOCK 是解锁操作，在某种程度上，可以将 UNLOCK 视为 ROLBK，将 UPDATE 视为 COMMIT。即如果锁定某条记录，并对其字段进行赋值之后，使用 UPDATE 语句，将会把修改后的结果保存下来，即修改文件，而 UNLOCK 语句则不会修改文件，即否认了之前对文件字段做的赋值修改。

从程序的执行效率上来讲，UNLOCK 的执行效率是高于 UPDATE 的，因为 UPDATE 操作时，系统需要对文件的每一个字段进行确认处理(DEBUG 时可以看到)，而 UNLOCK 就是简单的解锁而已。

UPDATE (Modify Existing Record)　　修改记录

语法与 UNLOCK 一样。

WHEN {(M | R)} (When)　　分支判断语句中的条件判断

在操作码“OTHER”，“SELECT”中都讲过，仍列出简单语法如下：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	SELECT					
	WHEN	条件判断 1				
	处理语句 1					
	WHEN	条件判断 2				
	处理语句 2					
	OTHER					
	处理语句 3					
	ENDSL					

WHENxx (When True Then Select)

上面的语法，是 RPGLE 的语法，WHENxx 是 RPG 的语法，也就是

	SELECT	
FLD01	WHENEQ	FLD02
	处理语句 1	
	……..	

这样的语法，在表达复杂的逻辑关系时，必须与 ANDxx，ORxx 一起使用，所以我不使用 WHENxx 这个操作码。

WRITE (Create New Records)　写记录

常用的方式：

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	CLEAR		文件记录格式名			
	EVAL	文件字段 1=xxxx				
	EVAL	文件字段 2=xxxx				
	WRITE	文件记录格式名				

表示在文件中写入一条新记录。文件需要声明为可写的。

通常会在给文件字段赋值之前，作一次 CLEAR 操作来进行初始化，以避免不必要的麻烦。

XFOOT {(H)} (Sum the Elements of an Array)

没用过，看帮助，是表示对数组字段的累加统计。

假设 DIMDATA 定义为一个数字型的数组变量，FHS01 为一个足够大的数字型变量

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	XFOOT	DIMDATA	FHS01			

就表示将数组 DIMDATA 中的所有记录的值都取出来，汇总相加，赋值到数字变量 FHS01 中

XLATE {(P | E)} (Translate)

将一个字符串中指定的字符，更换成另外的字符。

举例：如 MYCHAR1， MYCHAR2 都是两个 20 位长的字符型变量

C		MOVEL	'ABCAAAC123'	MYCHAR1
C	'A':'9'	XLATE	MYCHAR1	MYCHAR2

执行过这个语句之后，MYCHAR2 就等于”9BC999C123’，即将字符串 MYCHAR1 中所有的“A”都变成了“9”；

XLATE 也可能指定起始位置。如上句更改为：

C	'A':'9'	XLATE	MYCHAR1：4	MYCHAR2

则 MYCHAR2 等于“ABC999C123”，指从第 4 位开始（含第 4 位），将“A”变成“9”赋值。

Z-ADD {(H)} (Zero and Add)　　向数字型变量赋值

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	Z-ADD	FLD01	FLD02			

将数字型变量 FLD01，赋值到数字型变量 FLD02 中。

Z-ADD、MOVE 虽然同是赋值操作码，但 Z-ADD 的用法就远没有 MOVE 那么变化多端，只能在数字型变量之间赋值。所以也没有什么可说的了。zero

如果要对数字型变量赋初值，使用*ZERO

	Z-ADD	*ZERO	FLD02

Z-SUB {(H)} (Zero and Subtract)　用 0 减

Factory 1	Operation	Factory 2	Result	HI	LO	EQ
	Z-SUB	FLD01	FLD02			

等价于

0	SUB	FLD01	FLD02

等价于

	EVAL	FLD02=FLD01*(-1)

***ALL**

*ALL 是个很有意义的变量，举例：

	EVAL	FLD01=*ALL’0’

表示将字符型变量 FLD01 赋值为全’0’

而

	CLOSE	*ALL

就表示关闭所有文件，意义与上面是不同的

%LEN

取字符串的长度，举例：

(MYLEN 为数字型变量，FLD01 为字符型变量)

	EVAL	MYLEN = %LEN(FLD01)

这句话的意思，是指取字符串 FLD01 的长度，不过通常这样用是没意义的，因为直接用%LEN 操作码，取到的是字符串的总长度，不是有效字符的长度，也就是说 FLD01 长度为 2，那么 MYLEN 就恒等于 2，不会变。就算变量 FLD01 中没有值，取出的 MYLEN 也等于 2.。

所以，%LEN 通常会与%TRIM 或是%TRIMR 一起使用，语法在下面介绍。

%TRIM，%TRIMR

都是去字符串变量中的空字符意思，%TRIM 是去字符串左边的空字符；%TRIMR 是去字符串右边的空格。

通常我们在写程序中，都是默认字符串变量左对齐，所以我们使用%TRIMR 操作码的概率应该高一点。举例：

	EVAL	MYLEN = %LEN(%TRIMR(FLD01))

这时的 MYLEN，就是指变量 FLD01 中的有效长度（前提条件是 FLD01 中如果有值，是左对齐）。如果 FLD01 为空，那么 MYFLEN 为 0；如果 FLD01 首位有值，第二位为空，那么 MYLEN 为 1；如果 FLD01 两位都不为空，那么 MYLEN 就等于 2。

如果字符串左对齐，那么就使用%TRIMR

还有一种用法，假设字符串 FLD04 为 4 位长的字符，FLD01，FLD02 都是 2 位长的字符，且 FLD01 等于“A ”，FLD02 等于“BC”，那我们执行：

```
EVAL        FLD04 = FLD01 + FLD01 + FLD02
```

FLD04 就等于“A A ”，也就是第二位与第四位都是空的，最后加的 FLD02 其实无效。

而如果执行

```
EVAL        FLD04 = %TRIMR(FLD01) + %TRIMR(FLD01) + FLD02
```

则 FLD04 就等于“AABC”，也就是说，在这里，%TRIMR(FLD01)，是等价于单字符“A”的

MONITOR

监控程序信息。据说是可以屏蔽掉出错信息，避免程序异常中断，未经测试。

3 和程序相关的数据库知识

3.1 LF（逻辑文件）

3.1.1 逻辑文件概念

LF，logic file，直译就是逻辑文件。我的理解，LF 文件即是 PF 文件的一种视图，所以 LF 文件自身是不会有数据的，也就是 LF 中没有 MEMBER。我们只能通过 LF 来修改 PF 中的 MEMBER 中的数据。（为了简单起见，“PF 中的 MEMBER 中的数据”，也可以就直接说是 PF 中的数据）

也就是说，我们建立 LF，来对 PF 中的字段进行排序，以便于我们程序中的查找，定位。

就数据操作而言，我们在程序中声明的文件，无论是 PF，还是 LF，只要对其进行了数据操作（WRITE、UPDATE、DELETE），最终其实都是操作的 PF 中的 MEMBER。LF 只是对其的一个排序视图。

LF 的基本写法如下：

```
A           R     文件记录格式名                PFILE(对应的物理文件名)
A           K     键值字段
```

LF 的文件记录格式名必须定义得与物理文件中的记录格式名一致，否则编译时会报错。

一、编写逻辑文件时，指定字段与不指定字段

在定义 LF 时，可以不指定字段，只指定键值，这样的话，默认 LF 文件中继承 PF 文件中的所有字段，如：

```
A           R     FMTFHS                PFILE(PFFHS)
A           K     FHS01
```

也可以指定字段名（写法与编写 PF 文件相同），如下：

```
A            R    FMTFHS                  PFILE(PFFHS)
A                 FHS01
A                 FHS02
A            K    FHS01
```

这样的话，这个 LF 将会只包含指定的字段名，在上例中，也就是 LF 文件中将不包含字段 FHS03。

如果程序中对该逻辑文件进行声明，并使用到物理文件中有，逻辑文件中未指定的字段—FHS03，程序在编译时将会报 7030 的错，即该字段不存在。

所以，在使用逻辑文件，或者说是设计逻辑文件时要注意规划，避免出现逻辑文件中字段不足的问题。当然，如果为了省事，所有的逻辑文件也可以都不指定字段，仅指定键值，这样就不会出错。但是在执行上似乎没有指定字段更有效率，尤其是 PF 文件的字段较多（如三、四十个字段，数据量较大时）。

二、带检索条件的逻辑文件

编译逻辑文件时，可以指定条件，过滤掉不需要的记录，如下例：

```
A            R    FMTFHS                   PFILE(PFFHS)
A            K    FHS01
A            S    FHS02                    COMP(EQ '01')
```

即表示这个逻辑文件以字段 FHS01 为键值，同时这个逻辑文件中只包含字段 FHS02 的值等于“01”的记录。

在关键字 COMP 中，常用的逻辑判断有 EQ、NE、GE、LE、GT、LT，这些判断符的含义在操作码中都有解释。

三、指定唯一键值

当逻辑文件中使用 UNIQUE 关键字时，如下：

```
A                                     UNIQUE
A            R    FMTFHS              PFILE(PFFHS)
A            K    FHS01
```

则表示在整个物理文件 PFFHS 中，FHS01 的值必须唯一。当试图写入文件中已含有的 FHS01 的值的记录时，程序将会异常中断退出，并报错：

Attempt to write a duplicate record to file 程序中声明的文件名

如果我们使用的逻辑文件中没有唯一键值，但程序运行时仍出现上述错误，那也就是说，这个物理文件还含有其它的逻辑文件是有唯一键值的。可用 DSPDBR + 物理文件名，来查看该物理文件对应的逻辑文件。

如果在逻辑文件中使用了 UNIQUE 关键字，同时还指定了多个字段，那即是表示在物理文件中，这几个字段联合起来，整体来看，值必须唯一。比如说

```
A                                     UNIQUE
A            R    FMTFHS              PFILE(PFFHS)
A            K    FHS01
A            K    FHS02
```

那么以下记录是允许同时存在的：

```
FHS01        FHS02
```

00 00
00 01
01 02
00 02

也就是说，只要没有 FHS01、FHS02 都相同的记录就可以了。

3.1.2 逻辑文件对效率的影响

当 PF 文件达到一定级别之后，新增一个逻辑文件需要考虑建立逻辑文件的时间。根据测试，一个有着二千万条记录的 PF 文件，在 640 上新增一个逻辑文件（非唯一键值，否则将会更慢）约需要 25 分钟，同时期间不能有其它程序来抢占资源，否则时间也会增加。

系统在新增记录时，会根据 PF 文件对应的所有逻辑文件，一一进行排序；如果有唯一键值的逻辑文件（UNIQUE 键值），系统还会判断是否有重复键值。

所以一个物理文件对应的逻辑文件越多，数据量越大，进行写操作时就会越慢。

根据实际测试，640 的机器在处理无逻辑文件的 PF 文件，进行写操作时，可达到每秒三万条；如果增加一个唯一键值的逻辑文件，速度将会锐减，而且越往后越慢。在总记录数为 200 万至 300 万时，所需要时间约为无逻辑文件的时间的 3—4 倍。似乎先向一个无逻辑文件的 PF 中写记录，然后再建立逻辑文件，比起先建立逻辑文件再写记录要快一些，具体差异未测试。

3.2 MEMBER

MEMBER，不知道怎么翻译更合适，用“成员”显得似乎有些生硬，这里直接用英文可能更合适些。

下面关于 PF 中多 MEMBER 的用法，实际适用范围并不广，这里我主要是谈一下 MEMBER 的概念，用时需谨慎。

1. 一个 PF 文件中，允许存放多个 MEMBER；
   PF 文件的数据实际上是存放在 MEMBER 中的；
   LF 文件中没有 MEMBER
2. PF 文件中的 MEMBER，可以通过命令
   “WRKOBJPDM 库名 PF 文件名”，然后选择 12，来查看。
3. 在程序开发时，为便于维护，便于理解，减少不必要的差错，通常一个 PF 都只有一个 MEMBER。PF 中可含有的 MEMBER 总数，是在编译物理文件时指定，（MAXMER），可以用 CHGPF 命令修改。
4. 通常，PF 文件默认的 MEMBER 是与 PF 文件重名的。
   当一个 PF 文件含有多个 MEMBER 时，通常系统会将与 PF 文件同名的，无同名时则是顺序排序第一个 MEMBER 做为默认的 MEMBER。此时我们在 SQL 操作，RPGLE 程序中操作的，将都是这个 MEMBER 的数据；如果要操作其它 MEMBER 的数据，需要使用 OVRDBF 命令，或在 RPGLE 程序中指定 MEMBER 名。
   如果要在程序中指定 MEMBER 名，必须与 USROPN 关键字一起使用。
5. 对于 RPGLE 程序而言，当声明文件时，如未用 USROPN 关键字，系统在运行程序时，一开始就会将声明的文件打开，并开始准备处理默认的 MEMBER 中的

数据。如果声明的文件重新编译过，而程序未重新进行编译，那么程序将不会运行，直接报错跳出；

而如果声明文件时，使用了 USROPN 关键字，系统在运行程序时，只有当执行到“OPEN 文件名”语句时，才会将声明的文件打开，并开始准备处理默认或指定的 MEMBER 中的数据。在这种情况下，如果声明的文件重新编译过，而程序未重新编译，那么**程序并不是一开始就报错**，而是运行到“OPEN 文件名”语句时，才会报错，异常中断。如果之前处理过其它的文件，并且其它的文件未进行日志处理，那么之前的处理将会生效，而这通常是我们所不愿意看到的（也就是 OPEN 之前的语句仍然会执行，这很不好），所以要小心。

6. PF 文件对应多个 MEMBER 的用法：

在实际处理中，可用于 FTP 上，来实现并发接收 / 发送格式相同的文件数据。

3.3 游标

3.3.1 游标的概念

游标，也就是系统中一个隐含的，指向某条记录的标志？

其实我以前一直把这东西叫做“指针”，不过好象这么说不太专业，因为指针的含义，是指某个变量的地址位（C 语言里面是这么说的），而且 RPGLE 程序中，也的的确确是有指针型变量，完全不是指向数据库中某条记录的意思。其实不管叫什么，总之能明白这个词代表的意思就行了。

当游标指向某条记录时，也就意味着系统对于该条记录进行了定位读取，此时可以取出当前记录的值，可以修改当前记录的值；

当游标指向某条记录之前，或之后，意味着系统仅仅对该条记录进行了定位，但并未进行读取，此时不知道该条记录的值，也无法对该条记录进行修改。

3.3.2 不同操作码对应的游标的处理

CHAIN 操作时，游标直接指向当前记录。

SETLL 操作时，游标指向相应记录之前，**有时（其实是很多时候），我们选用的逻辑文件的键值的顺序，未必如我们所预想的那样**（也就是我们会用错逻辑文件），这个我就不举例了。因为举例来说，可能大家都看得懂，而在实际操作中却不一定能把握好，这个只有靠自己在遇到问题时多想想才行。如果发现逻辑文件的排序不如意，只有更换逻辑文件，或更换程序代码的写法（如指定每一个键值的值，再去 CHAIN，而不是用 SETLL+循环 READ）。当然也可以

SETGT 操作时，游标指向相应记录之后

READ 操作时，游标指向下一条记录

READP 操作时，游标指向上一条记录

3.3.3 “有且仅有”的游标

在程序中，声明的每个文件，都**有且仅有**一个游标来定位记录。

这个“有”的意思，也就是说声明的每个文件，无论是物理文件还是逻辑文件，都会有游标。如程序中声明了 PFFHSL1，PFFHSL2 这两个文件，它们都是 PFFHS 的逻辑文件，系统在运行程序时，这两个文件都有对应的游标来定位记录。（当然，之前说过，如果 PFFHSL1、PFFHSL2 的记录格式名相同的话，必须要对其中之一进行 RENAME 处理）

这个“仅有”也很重要，举例而言，如果在程序我们只声明了一个文件 PFFHSL1，有时会在处理某条记录 A 的时候，还会处理 A 所对应的另外一条记录 B（这个 B，也是物理文件 PFFHS 中的记录，在程序中，通过 PFFHSL1 来处理它），那么我们在处理了这个 B 时，PFFHSL1 仅有的游标就已经由 A 指向了 B，也就是说游标此时已与 A 记录无关系了。之后的 READ 操作，READ 的是 B 记录之后的记录，而不是 A 记录之后的。如果程序中仅仅只是处理 A 这一条记录，倒没什么影响；但如果是循环读取数据并做处理时，将会造成游标指向的异常，可能会漏掉记录（比如 B 按排序，在 A 记录之后），或造成死循环（B 按排序，在 A 记录之前），或者数据处理无误，但效率降低。

对于上述问题，常见的解决之道，是声明两个文件，（要注意，同一个文件不能声明两次），用一个文件用来进行循环读取并做普通处理，当读到记录 A，发现要额外处理记录 B 时，再用另一个文件来定位 B，并进行修改。不过这种方法，要求定位 B 的文件必须有唯一键值，否则可能会定位到键值相同的记录 C 上面去。

也可以使用保留 A 记录的键值到临时变量中，在处理完 B 记录之后，再用这个键值定位回 A 记录的方法来解决，同样，保留的键值也必须是唯一键值。

3.3.4 LOVAL、HIVAL 对应的游标操作

如果程序中，对声明文件 **Record Address Type** 项使用了“K”，那么在 SETLL 操作时，除了使用键值定位之外，还可以使用*LOVAL，与*HIVAL，即

```
*LOVAL      SETLL       记录格式名
```

或

```
*HIVAL      SETLL       记录格式名
```

其中：

*LOVAL，是指对于键值而言的一个最小值，也就是**将游标定位到所有记录之前**，常用在循环 READ 之前，通常表示接下来，准备读取所有记录。不过要注意，如果*LOVAL 写到了循环里面，那就表示每次循环，都会由第一条记录开始读起，如果是读完所有记录才跳出循环，那么这样写将会造成死循环。举例如下：

常用的循环读取文件的写法

```
*LOVAL      SETLL       记录格式名
            DOW         1=1
            READ        记录格式名                    58
            IF          *IN58='1'
            LEVAE
            ENDIF
            ENDDO
```

如果写成了

```
          DOW       1=1
*LOVAL    SETLL     记录格式名
          READ      记录格式名                          58
          IF        *IN58='1'
          LEVAE
          ENDIF
          ENDDO
```

那就是一个标准的死循环。这种错误很多人都犯过，特此指出。

*HIVAL 与之相反，意思就是指对于键值而言的一个最大值，也就是将游标定位到所有记录之后，常与 READP 一起使用，即先将游标定位到文件最末尾，然后倒序，向上读取。

3.4 事务处理 -- COMMIT

3.4.1 概念描述

程序中，常会看到声明文件时，使用的“COMMIT”关键字，有不少人对于这个关键字的含义可能不是特别清晰。COMMIT 关键字，可以实现事务处理。简单的来说，事务处理，就类似于游戏中的 SAVE / LOAD 一样。当我们觉得结果不满意时（比如程序运行时，发现有错误，不能满足正常的业务流程），我们可以使用类似于取档的功能，使用数据恢复到原先的初始状态（ROLLBAK，回滚操作），也就是使数据恢复成为到程序执行之前的数值；而当我们对当前结果满意（或者是程序正常运行完毕）时，就执行类似于存档的功能，将数据彻底写死，使其无法再回滚。

举个实际的例子，如果程序中，以修改的方式声明了四个文件 A、B、C、D，其中 A、B、C 都使用了 COMMIT 关键字，而 D 未使用 COMMIT 关键字；在程序执行过程中，首先更改了 A、B、D 的值

在接下的处理中，如果业务流程判断，逻辑有误，不再执行，此时可以进行回滚操作，此时 A、B 的数据恢复成为修改之前的数值；D 的数据，因为在声明文件时未使用 COMMIT 关键字，所以回滚操作对它无效，即 D 的数据仍然保持修改之后的值；

而如果业务流程判断正常，程序顺利执行完毕，此时需要进行一次确认操作（COMMIT），来落实数据的修改。

就我的习惯而言，确认操作（也可称之为落实操作），与回滚操作，统称为“事务处理”

3.4.2 使用方法

（参考 hanyu 的《转 Commit/Rollback 概念》）

要实现事务处理的功能，需要以下的准备工作：

1、 生成一个日志接收器（CRTJRNRCV）

2、 生成一个日志，并将该日志连接之前建立的日志接收器（CRTJRN）

3、 将需要声明的文件，增加到上面已建立的日志中（STRJRNPF），只增加 PF 文件即可，不需要增加相应的 LF 文件；一个 PF 文件只用增加一次。

4、 对于当前进程，执行启用日志的命令 STRCMTCTL LCKLVL（*ALL）（对于同一进

程，执行一次即可，通常在签到时由公共程序来执行）

5、 在程序中，对声明的文件，填加 COMMIT 关键字

6、 在执行完修改操作语句后，执行事务处理。

步骤 1、2、3 通常由系统管理人员来执行，对于普通程序员而言，不需要理会。不过需要注意，一个 PF 文件如果重新编译之后，系统会自动将它从日志中去掉，所以在重新编译过 PF 文件之后，需要执行 STRJRNPF 命令将它再加到日志中去。（当然，新增的 PF 文件，如果需要进行日志处理，也是要加到日志中去的）

而步骤 4，一般是隐含在登录程序中执行的。登录程序可以 CHGUSRPRF 命令来查看。再引申一下，对于使用 SBMJOB 命令提交后台执行的程序而言，因为 SBMJOB 是提交产生了一个新进程，这个新进程未执行签到程序，所以如果运行的程序有用到事务处理的话，将会报错。解决之道是在这个新进程中，执行业务处理程序之前，先执行一次签到程序，或执行 STRCMTCTL 命令。

3.4.3 注意事项

1、 在实际使用过程中，普通的业务处理程序，通常会将所有的声明为“修改”方式打开的文件，都加上 COMMIT 关键字，所谓一损俱损，一荣俱荣，只要业务逻辑判断上有任何一处不符合要求，所有已修改的文件，都恢复成为修改之前的值，这样的处理是完全正确，也是应该的；但对应于批处理程序，尤其是循环处理某个表，或某几个表中的多笔记录时，通常会采取处理完一笔记录，执行一次事务处理的方式，因为我们不希望因为一笔记录不符合要求，就导致其它所有记录都不再处理。当然，有特殊要求的除外。

2、 为了便于排错，跟踪数据，通常日志性的文件（仅执行写操作），是不会定义 COMMIT 关键字的。

3、 落实操作与回滚操作，有两种方式来实现，一种方式是在程序中使用操作码，对应操作码为 COMMIT 与 ROLBK；第二种方式是使用命令，对应的命令为 COMMIT 与 ROLLBACK。这两种方式效果是相同的。

4、 事务处理仅针对当前进程的，对其它进程无效。

5、 声明的文件如果定义了 COMMIT 关键字，那么对其进行修改后，修改的记录在执行事务操作之前，会一直保持锁定状态。锁定的记录越多，其它程序的执行效率将会越低，同时当前进程在执行回滚操作时的时间就越长，所以原则上，我们通常在完成了一次处理之后，尽快执行事务操作。

6、 在交互式画面中，我们签退时（SIGNOFF），系统会默认执行一次 ROLLBACK 操作（这个好象可选，可以改为默认执行 COMMIT 操作，待查）。所以有时如果发现数据莫名其妙的恢复了，或非所预期的结果，不要首先怀疑系统故障或公共程序，请先回想一下自己的操作是否合理。就概率而言，在平常开发、测试时，建议多用 ROLLBACK 命令，但运行时间超长的大程序不适用此原则。

7、 SQLRPGLE 程序中，对于 SQL 语句修改的文件，程序默认对文件进行了 COMMIT 声明，也就是执行了 SQL 语句之后，需要进行事务处理操作。否则执行完程序后，如果仅检查数据更新无误，在未执行事务处理操作的情况下就签退的话，系统将会默认执行 ROLLBACK 操作，导致修改无效。（关于 SQLRPGLE，似乎也有参数可改，待查）

3.5 关于锁表的问题 LCKW

400 的设置应该是锁记录。

程序内部调用之一：

如果 A 程序调用 B 程序，而且 A、B 程序都用 U 的方式打开同一个文件，更改同一条记录时，那么在调用 B 程序之前，需要有一个 UPDATE 或者 CLOSE 的动作，否则在被调用的 B 程序将会锁表。

程序内部调用之二：

单个程序内，无论有无定义 COMMIT，只要做了 UPDATE 的操作，都可以多次对同一条记录进行定位操作（CHAIN、READ 等）的；只要没做 UPDATE 操作，下一次做定位操作时，就会锁表。

程序之间的调用：

A 程序定义了 COMMIT，在定位操作语句（CHAIN、READ 等）之后，UPDATE 操作之前，其它程序查找该条记录，仍然是原始的数据；在 UPDATE 操作之后，事件处理语句之前（COMMIT、ROLBK），其它程序查找该条记录，则是更改后的数据了。如果之后再进行 ROLLBACK 操作，其它程序查到的就又是原始的数据。这样在统计时就会出现错误，所以说统计程序一定要在事务处理语句之后进行。

当 A 程序在执行 UPDATE 操作之后，事务处理语句之前，其它程序以修改的方式读取该条记录时，是会锁表的。也就是当声明的文件定义了 COMMIT 关键字时，在打开一条记录之后，除了进行 UPDATE 或 UNLOCK 语句解锁之外，还必须进行事务处理操作，才能正式解锁。所以 400 的系统处理，在逻辑上看还是很严谨的，不会出现 UNIX 上类似于 VI 编辑时，后者覆盖前者的问题，当然这个比方打得可能有点不合适，总之就是数据的处理在逻辑上不会出现混乱，在排障时，不用在这方面想得太过复杂。

4 DEBUG 调试以及常见出错信息

4.1 写在前面

关于 DEBUG，其实也是一个老生常谈的问题了。不断有人讲，也不断有人问。而本文作为一个入门级普及型的文档，当然应该对 DEBUG 进行介绍。

很多初学者，一碰到错误，或是程序运行出错，或是运行结果不正确，第一个反应就是“找个人问问”；其实出现问题是好事，在错误中才能总结经验嘛，找人问就等于是浪费掉了这样的机会。当然，对于初学者而言，可能还不知道 DEBUG 这个调试方法。

平时我们调用程序时，执行完 CALL 的命令之后，系统会自动将程序由头运行到尾，我们只能知道程序运行是否正常运行完毕；或者在程序运行完毕之后，查看结果是否正确。如果不正常，再回过头仔细检查程序。

但很多情况下可能我们看来看去，始终觉得程序并没有错误。这时，我们就可以使用

DEBUG 来进行跟踪调试。所谓跟踪调试，就是指系统运行时，我们可以逐步地运行程序，并且运行每一步时，都可以观察到程序中变量的值，这样就可以知道程序为什么运行得不理想。通常而言，一个程序的错误只要是可以重现，（也就是在相同的初始条件下，始终会出现同样的错误），那我们就一定可以将错误 DEBUG 找出来；这时最重要的，就是找出程序出错的规律来。（当然，不排除会有很厉害的人写得很强的 BUG），所以说，会有“DEBUG 是王道”的说法。就算没错误的时候，对正常运行的程序进行 DEBUG，也可以更好地帮助理解程序的运行方式。

总而言之，DEBUG 是个很有用的调试方法，做为一个 RPGLE 程序员，不能不掌握它。

4.2 常规用法

4.2.1 程序编译

如果想对程序进行 DEBUG，必须在编译时指定参数：

编译程序时，先按 F4，再按 F10，修改编译参数的值(可能需要翻页)：

RPGLE	程序：	Debugging views	(DBGVIEW)	选项填“*LIST”，或“SOURCE”
RPG	程序：	Source listing option	(OPTION)	选项填“*LSTDBG”
		Generation options	(GENOPT)	选项填“*LIST”

如果编译的程序未指定这个参数，那么执行 STRDBG 命令时，将会毫无效果。

所以编译时的参数非常重要，一定要注意。

4.2.2 执行 DEBUG 命令

在命令行，输入

STRDBG + 程序名

然后按 F4，这两个参数要选为“*YES” ：

Update production files （UPDPROD）

OPM source level debug （OPMSRC）

接下来，就可以进入 DEBUG 模式。也就是我们可以看到这个程序编译后的可执行代码。比起我们写的源代码而言，通常会多出声明文件的字段部分，一般在最末尾。

然后我们要做的，是设置断点。所谓断点，也就是当我们在 DEBUG 环境下执行 CALL 程序是，系统执行到断点处将会停止运行，等待我们输入命令（如 F10 是单步向下执行、F3 是直接退出运行等）。那么，如果我们没有设置断点，在 DEBUG 环境下执行 CALL 命令，系统将会直接将程序运行完毕。

DEBUG 模式中设置断点的方法，是在目标代码行按 F6 键。当代码行被设置断点后，将会反白显示。如果觉得断点设置得不对，想取消，那么在该行再按一下 F6 键，代码行将会取消反白显示，也即是取消该行断点。

4.2.3 运行程序

设置好断点之后，按 F12 键（退出当前环境），或 SHIFT+F9 键（在 DEBUG 状态下调

出命令行）都可以，总之都是在交互式命令行上执行：

CALL　+　程序名

这时，程序会执行到首个断点处，并显示出 DEBUG 模式。

4.2.4　在 DEBUG 模式中进行调试

在 DEBUG 模式中，可做如下操作：

F3　退出程序（不执行下面的语句了）

F6　定位断点（定位断点处，会反白显示）/ 已定位断点处再按 F6，即是取消断点

F10 单步向下执行程序

F11 查看当前光标处的变量的值

F12 程序运行到下一断点处（若无断点，则程序会运行完毕）

F21 调出交互式命令行

在 DEBUG 模式的命令行中执行：

(注意，是 DEBUG 模式的命令行，不是用 F21 调出的交互式命令行)

EVAL　变量名，查看当前变量的值；(如 EVAL　FLD01，就是查看字段 FLD01 的值)

EVAL　变量名=''，直接更改当前变量的值（通常不要使用这个功能）

F 字符串　顺序向下查找字符串（如 F　FHS，光标将会跳至顺序向下，代码段中首个 FHS 字符串处

F16　继续查找下一个字符串（紧接在 F　字符串后使用）

4.2.5　跟踪被当前程序调用的程序

在 DEBUG 过程中，有时我们希望当前程序在 CALL 另一个程序时，还可以跟踪被 CALL 的程序，比如说当前跟踪的程序是 FHS01R，FHS01R 中 CALL 了 FHS02R，我们跟踪到 FHS01R 中，运行了一部分代码，现在想看看 FHS02R 被调用时，是如何运行，这时我们可以通过如下操作来增加跟踪调试的程序：

一、SHIFT+F2，出现如下画面：

Opt	Program/module	Library	Type	
		*LIBL	*PGM	
	FHS01R	FHSLIB	*PGM	
	FHS01R		*MODULE	Selected

二、增加程序

在光标所指向的第一行，OPT 项输入 1（即表示增加跟踪的模块），Program/module 项输入 FHS02R，然后确认，画面将会变成：

Opt	Program/module	Library	Type	
	fhs02r	*LIBL	*PGM	
	FHS02R	FHSLIB	PGM	
	FHS02R		MODULE	
	FHS01	SLIB	PGM	
	FHS01R		MODULE	Selected

三、进入增加的程序

在第三行，也就是

FHS01R		MODULE

这一行，选择“5”,就进入了程序 FHS02R 中。

四、跟踪调试程序 FHS02R

进入程序 FHS02R 之后，仍然是设置断点

接下来按 F10，或 F12，系统执行到程序 FHS01R 中的“CALL ‘FHS02R’”的语句时，就会进行我们设置的程序 FHS02R 的断点处。

当程序 FHS02R 执行完毕之后，程序还会退回到最开始我们 DEBUG 的程序中，也即是 FHS01R 中。

补充说明：

以上的跟踪被当前程序所调用的程序，仅限于两个程序都是 RPGLE 程序。

4.2.6 一定要退出 DEBUG 模式

当我们结束 DEBUG 调试之后，必须输入“ENDDBG”用来结束 DEBUG 调试模式。否则无法进行下次 DEBUG 操作，而且可能会带来一些不可预知的错误。

4.2.7 补充

对于程序中未用到的字段，用 EVAL 是看不到它的值。举例而言，比如 PF 文件 PFFHS 中有若干条记录，我们在程序代码中只对字段 FHS01 进行了操作（如将 FHS01 赋值到另一变量中，或是将 FHS01 赋成某个值，总之就是有效代码行中出现了 FHS01），未对 FHS02 进行操作（也就是有效代码行中未出现 FHS02），那么我们在 DEBUG 时，就算读到 PFFHS 中的记录，FHS02 是有值的，EVAL 出来的结果，也会是空（字符型变量为空，数字型变量为全 0）。

不过如果程序中，是以修改的方式声明的 PFFHS 文件，并且有效代码行中出现了 UPDATE 的操作，那么系统其实隐含了对 PFFHS 中每个字段都进行赋值的操作（也就是在 UPDATE 操作时，系统其实是会将 PFFHS 中每个字段都重新赋一次值），所以此时，就算有效代码行中未出现对 FHS02 字段的操作，DEBUG 状态下也可以看到它的值。

4.3 跟踪批处理程序(From qingzhou)

通常我们对于批处理程序，在交互式的状态下测试好以后，再正式开始使用提交后台的方式，让程序进行批处理运行，但有时交互式环境与批处理环境的不同，会导致程序运行异常，此时使用以下方法跟踪批处理程序会更为直接

1. 以 HOLD(*YES)参数提交 JOB 到 QBATCH JOB 中，让 JOB 暂时挂起；
2. 使用 WRKSBMJOB 查看所提交的 JOB 的以下 3 个参数值：
   *Job id
   *User Name
   *Job Number
3. 执行 STRSRVJOB，填入第 2 步骤获得的 3 个参数进行 QBATCH JOB 服务过程；
4. 执行 STRDBG 开始 DEBUG；
5. 利用 F21 键切换到命令行，在命令行执行 WRKSBMJOB，使用 6=Release 释放挂起的第 1 步骤提交的 JOB，然后系统允许你按 F10 输入 DEBUG 命令（注意：不要键入执行，否则在设立断点之前键入执行，程序就会运行，因而无法进行 debug 断点设置）；
6. 在 OS/400 命令行窗口；执行 DSPMODSRC 后，可通过 F6 设置断点；然后按 F3 退出，再按 F12 退出命令行；
7. 键入执行释放挂起的 JOB；程序将在断点中停留；可以使用交互式 DEBUG 使用 DEBUG 命令进行处理
8. 一旦程序或者 JOB 结束，使用 ENDDBG 和 ENDSRVJOB 结束操作。

补充：

如果在批处理作业的 RPGIV 程序中出现交互语句，如：DSPLY；显示文件输入输出语句程序的调用，如：EXFMT，作业将会被挂起处于 MESSAGE WAIT 状态，这是因为批处理作业无法处理显示信息而引起的。

4.4 常见的出错信息

4.4.1 编译程序时的出错信息

一、进入编译后的脱机文件中

在编译程序时如果出错，在命令行执行“WRKSPLF”，然后用 SHIFT+F6 去到最末尾，可以看到名称为“程序名”的脱机文件（SPOOL FILE），用 5 进入，查看出错信息。

File . . . . . : FHS02R Page/Line 1/1
Control . . . . . Columns 1 - 75
Find . . . . . .

二、了解当前编译的错误信息的类型、数量

如上例，即假设程序 FHS02R 编译时出错，进入到编译后的脱机文件中，先在“Control”处，填“B”，确认，到最末尾，可看到类似于这样的信息：

F i n a l S u m m a r y

Message Totals:

Information (00) . . . . . : 1

Warning (10) . . . . . : 0

Error (20) . . . . . : 0

Severe Error (30+) . . . . . : 2

---------- -------

Total . . . . . : 3

这就表示当前程序编译之后，有 2 个 30 级的错误，1 个 00 级的错误，总共有 3 个错误。

00 级的错误仅是信息级别(Information)，可以不理会；

10 级的错误是警告，也可以不理会；

20、30 级别的错误就是正式的错误，也就是不解决它们，程序就无法编译通过

除此之外，还有 40 级，以及传说中 50 级的错误。40 级的错误通常是程序中声明的文件不存在；50 级的我没见过，只是有人吹嘘他曾经写出过，表示他的 RP 与众不同。

总之，级别越大，就表示错误的问题越严重，所以排错的顺序，是先大后小。即先排除 40 级错误，再来查 30 级错误，然后才是 20 级。

举个例子，如果在写代码时，将声明的文件名写错，那么 30 级错误可能会有上百个，凡是涉及到与该文件中字段有关的语句，肯定都会报错；此时排除 40 级错误（文件名声明错误）之后，会发现 30 级错误将会大减少。

所以说，排错顺序是先大后小。

三、查找出错信息对应的代码行

脱机文件的最后一页，是错误的汇总信息，向上翻页，查看具体错误信息，仍是上例：

M e s s a g e S u m m a r y

Msg id Sv Number Message text

***RNF7086 00 1 RPG handles blocking for the file. INFDS is updated only blocks of data are transferred.**

***RNF7030 30 1 The name or indicator is not defined.**

***RNF7515 30 1 The move operation has operands that are not compatible.**

*** * * * * E N D O F M E S S A G E S U M M A R Y * * * * ***

5722WDS V5R1M0 010525 RN IBM ILE RPG FHSLIB/FHS02R

注意看，这里有两个 30 级错误，RNF7030，RNF7515。

以 7030 为例，在“Find”处，输入 7030，然后 SHIFT+F4 查找（也就是在脱机文件中查找 7030），多找几下，就可以看到具体的错误信息：

Msg id Sv Number Seq Message text

***RNF7030 30 12 000009 The name or indicator FHS20 is not defined.**

可以很清楚地看到，就是说代码第 12 行，字段“FHS20”没有定义。

这里，Number 项的值，就表示代码行数。当然，这个代码并不是我们源代码中的行数，而是指编译之后的执行代码中的行数。可以从脱机文件中的程序代码处从头开始看起，如：

12 C MOVE ‘ABCD’ FHS20

这个最左边的 12，就表示这是代码执行的第 12 行。

当我们查看到是错误行的语句之后，就可以回到源代码中，按图索骥地找到错误的代码，然后修正它。

系统对于错误的提示很准，不用怀疑。（与 C 不同，C 编译后，只是说疑似某行出错）

再一次提起要注意，系统提示的，是系统编译时的代码行行数，不是我们自己写的源代码的行数，要注意比对。Number 项标识错误的代码行行数

四、常见编译出错信息：

RNF2120

声明的文件不存在

RNF7030

变量未定义，通常随着如果变量未定义，那么与该变量有关的每一句话，都会报错，并且除了 7030 之外，还会有诸如类型不匹配这些的错误，所以排错时，一般都是先修改 7030 的错误。

RNF5177

使用了 DO、FOR、DOW、IF、SELECT 语句时，漏写了相应的 END 语句

4.4.2 运行时的出错信息

CPF4328

对声明的文件使用了 COMMIT 关键字，但该文件未加入到日志文件中

CPF4131

文件重新编译过，但之后未重新编译该程序

CPF128

这个比较麻烦，好像是说 PF 文件 damage（被破坏）。似乎只能恢复 PF 文件，或者是重新编译。

Decimal Error

通常是给数值型变量赋值时，超长溢出

Attempt to write a duplicate record to file

试图向文件中写入重复键值

5 CL、CMD

其实有关这一章，以及屏幕文件的，在网上已经有很多人写过了，想了想，还是说一下吧。

5.1 CL 程序

5.1.1 基本认识

简单的理解，CL 程序就是和 RPG 相对应的，是控制语言（Control Language）。类型为 CLP、CLLE 的源代码编译出来的程序，都属于 CL 程序。

可能还是不够直观，这么说吧，我们在交互式命令行上输入的命令，用程序的方式来执行，这个执行的程序，就是 CL 程序。

学过 UNIX 的会比较好懂，CL 程序有点类似于 SHELL，不过 SHELL 是可以直接执行的，而且不用编译；CL 程序需要编译，而且要用 CALL 的方式来执行。不过原理是接近的，

都是在程序中直接调用命令行的语句。

所以说，CL 程序其实很好写，只要会输入命令，就可以写 CL 程序了。在编辑 CL 程序时，也可以用“命令 ＋F4”的方式来写，不需要老老实实的整行输入。

CL 程序不像 RPGLE 程序，在编写时，可以使用自由格式书写；一行的内容如果太长要，在最末尾处用“+”表示换行

举个最简单的例子，比如说新建个名为 FHS01CL 的 CLP 源程序，代码如下：

```
PGM
WRKACTJOB
ENDPGM
```

编译此程序，然后 CALL 之，系统就会执行命令 WRKACTJOB，查看当前的活动作业，效果与在交互式命令行下输入 WRKACTJOB 是一样的。

当我们输入 F12，退出 WRKACTJOB 时，系统就会继续向下执行，发现是 ENDPGM，表示程序结束了，于是判定执行完毕，退出至交互式画面。

5.1.2 CL 程序的常用语法及命令：

一、程序的开始与结束：

```
PGM     PARM(&A   &B)      /* 开始 CL 程序 */
ENDPGM                     /* 结束 CL 程序 */
```

CL 程序，和 RPGLE 程序一样，也可以有程序的入口参数，而且程序的入口参数都是可传递的（也就是输入的参数如果在程序中被修改过，那么原调用的程序中的相应参数也会进行变化。不过 CL 的入口参数只能为字符型，或数字型的单个字段，不能象 RPGLE 程序中那么多样化（字段、结构、数组、指针）。

如果 CL 程序没有入口参数时，那么就可以不需要后面的 PARM 语句，直接写成

```
PGM
```

即可。

写 CL 程序时，不妨多使用 F4，看看系统的帮助，这样就不用记那么多命令的参数名。

二、变量及其定义

CL 程序中的所有变量，都使用&做为前缀，这一点与 RPGLE 程序不同。比如说

```
PGM     PARM（&A  &B）
```

就表示入口参数为 A、B 这两个变量

在 CL 程序中使用到的变量，都必须使用 DCL 语句来定义：

```
DCL         VAR(&FLD01)    TYPE(*CHAR)    LEN(10)
DCL         VAR(&FLD02)    TYPE(*DEC)     LEN(10 2)
```

上述语句表示：

定义变量 FLD01，10 位长的字符型变量

定义变量 FLD02，10 长，其中 2 位小数的数字型变量

除了字符、数字之外，CL 程序还可以定义逻辑变量(*LGL)，逻辑变量允许的值只

能为’1’或’0’。不过通常有字符与数字也就够了。CL 程序的主要功能在于进行命令处理，而不是处理字符串以及数据库

三、CL 常用命令：

CHGVAR -- 变量赋值

CHGVAR VAR(&FLD01) VALUE(‘ABCD’)

即是将变量 FLD01 赋值成为’ABCD’(左对齐)， 同理，VALUE 的括号中也可以填写一个变量，即表示将此变量的值赋值到变量 FLD01 中。

数字型变量的赋值同样也是使用 CHGVAR 语句。

当变量中只包含数字时（0—9），数字型变量与字符型变量可以使用 CHGVAR 语句进行转换，这一点与 RPGLE 中的 MOVE 语句比较类似。

IF -- 条件判断语句

```
IF      COND(&FLD01 *EQ '1')    THEN(CHGVAR VAR(&FLD02) +
          VALUE('0'))
```

当变量 FLD01 等于’1’时，将变量 FLD02 中的值更改为’0’

THEN 后面，即是当符合条件时，要执行的命令。写起来其实没有看上去那么复杂，多用 F4 就会发现 CL 程序写简单。

就比如上例，先写 IF，然后按 F4，在 Condition 处填写条件语句，然后在 Command 处填上 CHGVAR，再按 F4，再去填相应的处理语句，这样写，就比直接把整句抄下来就简单多了。

上面这种写法，只能在符合条件时，执行一条 CL 语句；如果要执行多条，就必须写做：

```
IF      COND(&FLD01 *EQ '1') THEN(DO)
    CHGVAR    VAR(&FLD02)    VALUE(‘0’)
    其它执行语句
ENDDO
```

也就是 THEN 后面，用 DO，表示接下来的语句都是在这个 IF 条件成立时才执行（DO）的。

然后结束处用 ENDDO，必须要有。ENDDO 在这里和循环没有任何关系，表示的是 ENDIF 的意思，但是 CL 语句里没有 ENDIF，只有 ENDDO。

IF 语句中，表示判断的关键字与 RPGLE 中的 Ifxx 操作码类似，有

***EQ *GT *LT *GE *LE *NE**

用来表示逻辑关系的关键字有

***AND, *OR, *NOT**

GOTO -- 跳转语句

GOTO 语句与 RPGLE 中的 GOTO 是一样的，都是跳转的意思。

```
FHSTAG：
GOTO  CMDLBL(FHSTAG)
```

注意，这里定义标签是用“：” 冒号

MONMSG -- 监控错误信息

我们使用 CL 语句时，执行的命令可能会报出一些异常错误，从而导致整个程序中断，需要手工干预。MONMSG 命令可以监控我们预定的错误信息，使 CL 程序正常的向下运行。举例而言，如果 CL 程序中有如下语句：

```
CALL    PGM(FHS01R)
MONMSG    MSGID(CPF4131)
```

则表示当系统调用程序 FHS01R 时，如果发现有 CPF4131（声明的文件重新编译过，但程序未重新编译）的错，那么 CL 程序将不会异常中断，仅仅只是不运行程序 FHS01R，然后继续向下执行 CL 程序

MONMSG 还可以用于在监控到错误信息之后，进行处理，如下：

```
CALL    PGM(FHS01R)
MONMSG      MSGID(CPF4131) EXEC(CHGVAR VAR(&FLD01) +
                VALUE('0'))
```

这句话就表示当发现有 CPF4131 的错误之时，将变量 FLD01 赋值成为’0’

如果要执行多句的话，和 IF 语句的方法类似，也是使用 DO 与 ENDDO

```
MONMSG      MSGID(CPF4131) EXEC(DO)
    CHGVAR      VAR(&FLD01)     VALUE(‘0’)
    其它处理语句
ENDDO
```

5.1.3 不常用的语法

%SST -- 取字符串

```
CHGVAR      VAR(&FLD01)   VALUE(%SST(&FLD02   3   1))
```

表示将字符变量 FLD02，从第 3 位开始，取 1 位，左对齐赋值到变量 FLD01 中。

%SST 的括号的参数中，第一个参数必须为字符型变量，第二个参数表示起始位，第三个参数表示要截取的长度。

***CAT -- 拼字符串**

```
DCL         VAR(&FLD01)         TYPE(*CHAR)     LEN(10)
CHGVAR      VAR(&FLD01)         VALUE('A'   *CAT    'B')
```

即表示将 10 位长的字符型变量赋值成为’AB ‘

‘A’， ‘B’，也可以使用变量，如

```
CHGVAR      VAR(&FLD01)         VALUE(&FLD02  *CAT   &FLD03)
```

要注意，*CAT 不能去掉字符串末尾的空，从效果上来看，有点类似于 RPGLE 中的 EVAL 操作码，而不是 CAT 操作码

+、-、*、/ -- 数学运算

数字型变量，可以进行数学运算

```
CHGVAR      VAR(&FLD01) VALUE(&FLD01 + &FLD02)
```

即等同于 RPGLE 程序中的 EVAL FLD01 = FLD01 + FLD02

同理，-、*、/ 分别对应减、乘、除

不过数学运行常用于 RPGLE 程序中，较少用在 CL 控制里面，这里只是介绍一下。

读取文件：（From　　Cuer：P1421）

```
DCL         VAR(&FLD01) TYPE(*CHAR) LEN(2)
DCLF        FILE(FHSLIB/PFFHS)
RCVF
CHGVAR      VAR(&FLD01)     VALUE(&FHS01)
```

以上这段 CL 的意思，就是在 CL 程序中读取 PFFHS 文件，然后将读到的第一条记录赋值到 CL 的临时变量 FLD01 中。

如果要将一个文件从头读到尾，则可以用如下语句来实现：

```
    DCLF        FILE(FHSLIB/PFFHS)
LOOP：
    RCVF
    MONMSG      MSGID(CPF0864) EXEC(GOTO CMDLBL(EXIT))
        读取到每条记录后的处理语句
    GOTO    CMDLBL(LOOP)
EXIT：
```

也就是说，信息 CPF0864，即表示未读取到记录。

在 CL 程序中声明文件使用 DCLF 语句，一个 CL 文件中只能声明一个文件，声明语句必须在 CL 控制语句之前。

使用声明的文件中的字段，同样需要在字段名前加上“&”；

CL 程序中，无法控制游标，无法对记录进行定位操作。所以 CL 程序对文件的操作是比较弱的，通常我最多只用来读取某些只含少量记录的参数文件。

5.2 CMD

CMD 是用来生成命令的，执行后可以像其它系统命令一样，直接输入命令，或是 F4，不需要像 CLP 一样，要 CALL 一下。

其实 CMD 本质上也是执行 CLP 或 RPGLE（在编译时指定），用起来，无非就是好看点，直接一些，除此之外的意义，似乎也就没什么了。

举个例子，比如我们查看一个文件中的内容时，可以使用 SQL 来查看，也可以使用命令 RUNQRY 命令来实现（RUNQRY　　QRYFILE（文件名））。但在我们要频繁查看文件时，这两种方式似乎都不是很爽，也就是说要输入的内容都不是最少的，那我们可以设计一个 CMD，譬如说叫 SEE，希望实现的最终效果，是在命令行输入“SEE　　文件名”，就可以查看 PF 文件中的记录。那么，我们按如下步骤来实现：

1. 建立一个 CLP 程序，比如叫 SEECLP，代码如下

```
PGM         PARM(&FILENAME)
RUNQRY      QRYFILE(&FILENAME)
ENDPGM
```

2. 编译此程序
3. 建立一个 CMD 程序（即源代码的属性为 CMD），代码如下：

```
CMD         PROMPT('  显示文件记录  ')
PARM        KWD(NAME) TYPE(*CHAR) LEN(10) MIN(1) +
            CHOICE('  显示文件记录内容  ') +
```

PROMPT('Display file record')

4. 编译此 CMD，用 F4，可见如下画面：

```
                                Create Command (CRTCMD)

 Type choices, press Enter.

 Command  . . . . . . . . . . . > SEE              Name
   Library  . . . . . . . . . . .>   FHSLIB       Name, *CURLIB
 Program to process command . . . > SEE            Name, *REXX
   Library  . . . . . . . . . . .>   *LIBL        Name, *LIBL, *CURLIB
 Source file  . . . . . . . . . . > FHSFILE         Name
   Library  . . . . . . . . . . .>   FHSLIB       Name, *LIBL, *CURLIB
 Source member  . . . . . . . . . > SEE             Name, *CMD
 Threadsafe . . . . . . . . . . .    *NO           *YES, *NO, *COND
```

其中，蓝色字体显示的，就是我们需要输入这个 CMD 要调用的程序名（默认值与 CMD 同名），这里我们将此项内容填为 SEECLP，表示 SEE 这个 CMD，调用的是 SEECLP 这个程序

5. 编译成功之后，我们在命令行执行“SEE　　文件名”，就可以看到指定文件的记录。也可以用 SEE ＋F4 的方式来使用
6. 要注意，CMD 中，PARM 表示的就是 CMD 命令的参数，参数的个数、类型、长度都必须与其调用的程序相匹配，但名称可以与其调用的程序中的参数名称不一样，而且名称前面不能有“&”字符。
7. 在 PARM 参数中，MIN(1)，表示该项参数必须有值（即最小的有效长度为 1），当参数无值时，将会自动出现 SEE ＋ F4 的效果，同时该项参数高亮显示。试一试就知道了

6 屏幕文件及使用（整理中）

```
A                                                      DSPSIZ(24 80 *DS3)
A            R HEAD
A                                                2     1DATE
A                                                        EDTCDE(Y)
A                                                2 69TIME
A                                                2 34'显示文件记录'
A*************************************************************
A            R SUBF                                SFL
A              FLD001          5A  O  9 16
A              FLD002         20A  O  9 28
A*************************************************************
A            R SUBFC                             SFLCTL(SUBF)
A                                                    SFLSIZ(9999)
```

```
     A                                                      SFLPAG(0010)
     A  30                                                  SFLDSP
     A  31                                                  SFLDSPCTL
     A  32                                                  SFLCLR
     A  33                                                  SFLEND(*MORE)
     A                                                      OVERLAY
     A                                                      CA12(12 'EXIT')
     A                                                7 15'字段 1'
     A                                                      COLOR(WHT)
     A                                                7 27'字段 2'
     A                                                      COLOR(WHT)
     A*************************************************************
     A            R FOOT
     A                                                      OVERLAY
     A                                               24 11'F12=EXIT'
     A                                                      COLOR(BLU)
****************** End of data ***************************************
```

(1) 第一个画面 HEAD 注解：

(a) 第一行表明显示尺寸，一般就这么写；

(b) 第二行的 R 表示接下来是一个记录，记录名是 HEAD，有点象 PF 的定义，不过在 DSPF 里的记录代表

一个画面；

(c) 第三行的 DATE 是系统关键字，表示日期，该句意思是在第 2 行第 1 列显示系统日期；

(d) 第四行的 EDTCDE 表示编辑字，EDTCDE(Y)表示系统日期按照“MM/DD/YY”格式显示；

(e) 第五行的 TIME 是系统关键字，表示时间，该句意思是在第 2 行第 69 列显示系统时间；

(f) 第六行意思是在第 2 行第 34 列显示字符窜"显示文件记录"。

(2) 第二个画面 SUBF 注解：

(a) 第一行的 R 和上面一样，也代表这是一个画面，名字是 SUBF；

后面的 SFL 是系统关键字，表示该画面是 SUBFILE；

(b) 第二行表示在画面第 9 行第 16 列显示长度为 5 的字段 FLD001，5 是长度，A 表示该字段是字符型，

O 表示该字段只用于输出；

(c) 第三行与上一行类似。

(3) 第三个画面 SUBFC 注解：

(a) 第一行的 R 和上面一样，也代表这是一个画面，名字是 SUBFC，

SFLCTL 是系统关键字，SFLCTL(SUBF)表示该记录是用来对画面 SUBF 显示的控制，

一般定义了 SUBFILE 之后，都要定义这个记录的；

(b) 第二行的 SFLSIZ 用来说明 SUBF 的记录数，这里我把它定义为最大 9999；

(c) 第三行的 SFLPAG 用来说明每页显示的记录条数，这里我把定义为 10 笔；

(d) 第四、五、六行的 SFLDSP 是用来显示记录的，SFLDSPCTL 用来控制显示的，

SFLCLR 用来清除显示的记录的，一般都需要给他们加上指示器，这里分别是 30、31、

32；

(e) 第七行的 SFLEND 用来声明每页下方的提示，这里用*MORE 来表示，如果还有下一页的话，下方

就显示”More……“，否则，就显示”Bottom“，这里也用个指示器 33；

(f) 第八行的 OVERLAY 表示显示该记录之前不要清除上一屏，即保留；

(g) 第九行的 CA12 表示在这个画面里我可以用键盘按键 F12，对应的指示器是 12，

这里你可以根据自己需要使用别的按键，例如 CA03；

(h) 第十一行的 COLOR 是系统关键字，用来给上面显示的字符窜设置显示颜色，

这里用 BLU，表示字符窜”字段 1“显示为蓝色；

(4) 第四个画面 FOOT 注解参考上面三点即可，在此不在累赘。

```
********** Beginning of data *************************************
     FMYPF          IF   E              K           DISK
     FMYDSPF   CF   E                           WORKSTN
     F
RRN

  KSFILE SUBF
     C              *IN12         DOWEQ'0'
     C                                WRITEHEAD
     C                                WRITEFOOT
     C                                EXSR CLRSFL
     C                                EXSR REDRCD
     C                                ENDDO
     C*
     C                                SETON                                        LR
     C*****************************************************************
     C              CLRSFL        BEGSR
     C*
     C                                Z-ADD0                  RRN        40
     C                                MOVEA'0011'       *IN,30
     C                                WRITESUBFC
     C*
     C                                ENDSR
     C*****************************************************************
     C              REDRCD        BEGSR
     C*
     C              *LOVAL        SETLLFMYPF
     C                                 READ FMYPF                          90
     C              *IN90         DOWEQ'0'
     C                                ADD   1             RRN
     C                                WRITESUBF
```

```
     C                       READ FMYPF                    90
     C                       ENDDO
     C*
     C                       SETON                         30
     C           RRN         IFEQ 0
     C                       SETOF                         30
     C                       ENDIF
     C                       MOVEA'10'     *IN,31
     C                       EXFMTSUBFC
     C*
     C                       ENDSR
****************** End of data ****************************************
```

只说一下，里面的 MOVEA'0011'　　*IN,30，这是数组附值，表示把’0011‘其中的’0‘给*IN30，’0

‘给*IN31，’1‘给*IN32，’1‘给*IN33。
还有为什么 ADD 1　RRN，再 WRITE，因为 RRN 这里面表示记录号，如果没有加 1，就会重复使用，系统会报

错。

以上是 1：简单的 SUBFILE 的应用部分。

下面是第 2 种，带窗口的 SUBFILE 的应用。

```
                                   DSPSIZ(24 80 *DS3)
     A                                 REF(*LIBL/ALGSYS)
     A                                 CHGINPDFT(CS)
     A                                 MSGLOC(24)
     A                                 PRINT
     A           R SCRN0
     A                                 WINDOW(10 40 12 34)
     A                                      WDWBORDER((*COLOR  WHT)
(*DSPATR RI)-
     A                                  (*CHAR '          '))
     A           R SCRN2                SFL
     A                                 KEEP
     A             SVSEL          1A  H
     A             S2SEL          1A  B  3  2COLOR(TRQ)
     A             S2DTA          1   B  3  5
     A             S2PRC          1   B  3  9
     A             DISYS      R       O  3 12COLOR(WHT)
     A                 DISYSN     R          O  3 23REFFLD(RALGSYS/DISYSN
```

```
*LIBL/ALGSYS)
      A                                      COLOR(PNK)
      A            R SCRN1                     SFLCTL(SCRN2)
      A                                      SFLSIZ(0007)
      A                                      SFLPAG(0006)
      A                                      WINDOW(SCRN0)
      A                                      CF01
      A                                      CF07
      A                                      KEEP
      A                                      BLINK
      A                                      OVERLAY
      A N50 51                              SFLDSP
      A N50                                  SFLDSPCTL
      A  50                                  SFLCLR
      A              SFSTRT          4S 0H     SFLRCDNBR
      A                                  1  4'Dta'
      A                                     COLOR(WHT)
      A                                  1  8'Prc'
      A                                     COLOR(WHT)
      A                                  2  2'X Y/N Y/N System       Name         '
      A                                     COLOR(WHT)
      A                                     DSPATR(UL)
      A            R SCRN3
      A*%%TS  SD  19950824  095816  KLUCK        REL-V3R1M0  5763-PW1
      A                                     WINDOW(SCRN0)
      A                                 10  2'F1=Return'
      A                                     COLOR(WHT)
      A                                 10 13'"X"=Select'
      A                                     COLOR(WHT)
      A                                 10 25'F7=Accept'
      A                                     COLOR(WHT)
      A                                 11 23'Roll Active'
      A                                     COLOR(WHT)
      A* ---------------------------------------------------------------------
      A* These records are required for seamless Window activation.
      A* No Program I/O shound ever be done to these records.
      A* ---------------------------------------------------------------------
      A            R SCRN4                     CLRL(*NO)
      A                                      OVERLAY
      A                                      FRCDTA
      A            R SCRN5                     ASSUME
      A                                      OVERLAY
      A                                      PUTOVR
      A                                  1  3' '
```

1 到 5 行，没什么好说的，只是其中的 MSGLOC（24）表示，信息显示固定在屏幕的 24 行处显示。
第 7 行，指定窗口的大小，第 8 行 WDWBORDER((*COLOR WHT) (*DSPATR RI)-
A (*CHAR ' '))
指定窗口边框显示的形式，具体效果自己可以去试，在这不多说了。

再重点说一下 SFSTRT 4S 0H SFLRCDNBR 这句的作用，SFLRCDNBR 表示子文件的记录号，这

个值要和在 RPG 中定义的那个
RRN KSFILE XXX，XXX 是指在显示文件里那个 SFLCTL。区分开来，这里的 SFSTRT 是用来显示 SUBFILE 的页

，就是说，RRN 在哪一页，就显示那一页。

概括的说就是指定要显示的子文件页是由这个字段的相对记录号的记录所在页。

7 其它

7.1 报表打印

在这里，简单说一下报表。其实据说 RPG 设计之初，主要就是为了解决报表问题。不过发展到现在，在我接触过的系统中，觉得报表在 RPG 编程之中反而退居其次，大部分程序都是对数据库中磁盘文件（即 PF 文件）的操作。

报表文件其实在某种意义上与普通的磁盘文件很类似，都是有记录格式（Record Format），都可以进行写操作（WRITE），RPGLE 的程序对它们的操作方式也比较雷同。所不同的是普通的磁盘文件的数据是存储在数据库中，而报表文件 WRITE 了之后，是以脱机文件（Spool File）的形式存在。所以在一个库中，磁盘文件具有唯一性，即不能有同名的磁盘文件；而在同一个输出队列中，同名的报表文件（即生成的脱机文件），允许有多个。

要做一个全新的报表打印，大致上可以分为画报表文件（PRTF），与编写打印程序这两部分。

一、画报表文件（PRTF）

1、 新建立一个属性为PRTF的文件，然后使用19进入报表编辑状态。（即STRRLU）
2、 定义一个新的记录格式（Record Format）：DR + F18 + F10，注意看下方的功能键说明
3、 在一个记录格式之下，追加内容时，要在新的编辑行前加“CLC”，表示此行的内容，是属于上面记录格式的。

4、 编辑行前加：

DC： 定义常量

CF： 使当前行的内容居中

5、 常用功能键：

F13：

标记 / 取消标记（光标所指的字段）。多试一下就知道使用方法，可以将同一行的多个相连的字段标记成为一个块（对首尾两个字段操作 F13 即可）。

高亮部分表示已被标记的块。

F14：

将已标记的块 COPY 到光标指定处。（其实这个我用得不多）

F15：

将已标记的块 MOVE 到光标指定处。（这个用得不少）

F16：

删除已标记的块。（这个用的频率也不少）

F11：

定义新变量

F23：

进入当前字段的功能菜单画面

6、 对新变量的定义，按 F11 之后，见到画面如下：

```
Edited length . . . . . :     1
Record format . . . . . :     RCD001
Number of keywords    . . . . . :     0
Number of indicators    . . . . . :     0

Type choices, press Enter.

  Field . . . . . .     FLD001          Name
  Option indicators . . . . . .                   01-99, N01-N99
    More indicators . . . . . .       N           Y=Yes, N=No
  Starting line . . . . . .                     1-255
  Starting position . . . . . .       6           1-255, +nn
  Length of data    . . . . . .       1           1-378, +nn, -nn
```

翻页，还有：

```
Data type . . . . . .     1               1=Character
                                            2=Zoned
                                            3=Floating point
                                            4=Open
                                            5=Graphic
                                            6=Date
                                            7=Time
                                            8=Time stamp
Decimal positions . . . . . .               0-31, +n, -n
Reference a field . . . . . .     N         Y=Yes, N=No
    Use referenced values . . . . . .       Y           Y=Yes, N=No
```

可以采用直接定义长度、类型的方法，即在第 1 页的最末尾，**Length of data** 处，填上字段长度；如果是字符型，就不需要再填其它内容；如果是数字型，在第 2 页 **Decimal positions** 处填上小数位数。

也可以采用参照字段的方法（即参照已存在的 PF 文件中的字段）。如果选用这种方法，就用需要在 **Length of data** 以及 **Decimal positions** 中填写内容，将 **Reference a field** 项填为“Y”，然后按确认键（好象 F10 键也可以），然后可以看到该项高亮显示。此时再按 F10，可进入该字段的功能菜单（也可以通过在报表编辑状态下，直接在当前字段处按 F23 进入）：

Opt	Keyword	Opt	Keyword	Opt	Keyword
	ALIAS		DFT		IGCCHRRTT
	BARCODE		DLTEDT		INDTXT
	BLKFOLD		EDTCDE		MSGCON
	CDEFNT		**EDTWRD**		PAGNBR
	CHRID		FLTFIXDEC		PRTQLTY
	CHRSIZ		FLTPCN		**REFFLD**
	COLOR		FNTCHRSET		**SKIPA**
	CPI		FONT		**SKIPB**
	CVTDTA		HIGHLIGHT		**SPACEA**
	DATE		IGCALTTYP		**SPACEB**
	DATFMT		IGCANKCNV		TEXT
	DATSEP		IGCCDEFNT		TIME

以上菜单中，标记为蓝色的，是我常用的几个选项。

在这些选项前面，选 2，即是对当前字段加注这些功能；选 4，即是去掉这些功能。

要参照已知字段定义变量，则在“**REFFLD**”选项前选 2，可见

Field . . . . . .

Record format . . . . . .

File . . . . . .

Library . . . . . . ***CURLIB**

各选项的含义：

Field：	当前变量所参照的字段
Record Format：	当前变量所参照字段，在 PF 文件中所属的记录格式
File：	PF 文件名
Library：	PF 文件所在的库名

7、 其它几个常用功能选项的含义：

SKIPA：Skip After， 在 WRITE 操作时，写该字段后，自动换页；

SKIPB：Skip Before 在 WRITE 操作时，写该字段前，自动换页

SPACEA：Space After 写该字段后，再打印一个空格；

SPACEB：Space Before 写该字段前，打印一个空格（可用来解决错行问题）

EDTCDE：定义当前字段的显示方式，比如说当前字段为数字类型时，是否显示逗号，前面是补零等等。根据 F1 键，看 Help 中的说明：

Edit Code Description **No Sign** **CR Sign** **- Sign(R)** **- Sign(L)**

Commas and zero balances	**1**	**A**	**J**	**N**
Commas	**2**	**B**	**K**	**O**
Zero balances	**3**	**C**	**L**	**P**
No commas or zero balances	**4**	**D**	**M**	**Q**
User defined edit codes	**5-9**			
Date field edit	**W**			
Date edit	**Y**			
Suppress leading zeros	**Z**			

二、编写打印报表的程序

1. 对报表文件的声明：

FEF4322P O E PRINTER OFLIND(*IN99)

可以看到，操作方式是“O”，即只写；

文件类型为“PRINTER”，即打印文件；

后面的 OFLIND 关键字表示该报表文件的换页指示器为 99；即写报表，当写满一页时，*IN99 自动变为 1；然后报表自动换页，*IN99 再自动变回 0；

其实这一项我觉得控制起来有点不爽，所以我通常都是自行控制换页，不用这个指示器来判断；我常使用的自行控制换页的方法在下面会说明。

2. 对报表文件的处理

和磁盘文件（DISK）一样，报表文件其实也有 OPEN，CLOSE，WRITE 的操作。不过使用 OPEN，CLOSE 操作时，不需要使用 USROPN 关键字。同时也因为生成的是脱机文件，所以不能进行 CHAIN、UPDATE 等定位、修改的操作。

如果在程序中，只需要生成一份报表，那么可以不使用 OPEN/CLOSE 操作，因为程序在运行之初，以及运行结束时，已默认打开，关闭了一次所有已声明的文件，包括报表打印文件。

但如果程序在运行时，需要生成多份报表，那么必须在每次生成报表前使用 OPEN 操作；在生成完报表后，使用 CLOSE 操作，以保证脱机文件的完整。

在生成完当前报表前，还可能需要使用 CHGPRTF 的命令，将报表生成到指定的输出队列中。如果不更改，那么报表会生成到当前用户默认的脱机文件存放处。当一个 RPGLE 中，生成多份同名报表时，常会在生成报表前使用 CHGPRTF，以便于管理，避免混乱。

报表的操作，也与磁盘文件（普通 PF 文件）类似，在 OPEN 与 CLOSE 之间，对各字段进行赋值，赋值完之后再通过“WRITE 记录格式名” 的语句，来写指定的记录格式。

3. 自行控制换页

在声明报表时，可以通过 OFLINE 关键字，定义换页指示器，不过可能是换页指示器的使用方法我摸索得还不够，所以使用起来总觉得不够自如。

我通常是先在报表文件中，定义一个记录格式(Record Format)，该记录格式下只有一个字段，同时这个字段的功能选项处要标识为“SKIPA”，然后该字段尽量做成不可见字符。在 RPGLE 程序中，当程序运行逻辑上判断需要换页时，WRITE 这个记录格式，就可以实现换页了（也就是通过字段的 SKIPA 选项来实现换页，当然大家也可以使用其它更便捷的方法，这里只是介绍我常用的方法，因为我觉得

这个方法比较稳定）。

至于如何判断换页，当然要看具体报表的需求是如何要求的。最重要的，是不要忘记定义一个数字型变量用来统计当前页已打印过的行数，当大于等于 60 行，要进行换页处理（我通常会留出几行来做为冗余，大家可以自己选择，但肯定不能大于 66 行，否则会自动换页）

三、几点补充

呵呵，找出我以前写的关于报表打印的贴子，做为使用时补充

1、 关于记录格式

其实这个本来没什么可说的，不过我就经常搞错，还是说说吧。

同一个记录格式之下，格式应该如下：

记录格式名

0001.00+　　打印内容

0002.00+　　打印内容

0003.00+

每行要有这个+，才归属于同一个记录格式。如果是个点的话，也就是说明这一行并不属于该记录格式，如果想让它属于这个记录格式，就要用 CLC 命令；

当我们用 19 编辑的时候，如果是 3 来 COPY 记录行，系统经常会自动在 COPY 处新增一个记录格式，所以要注意。（说来简单，但是我最开始画报表的时候，就曾经被这个问题折磨）

2、 关于变量的命名

不同的记录格式，可以使用相同的变量名。

有的时候，为了少写赋值语句，可以直接将变量命名得与 PF 中的字段相同。（不过我不太喜欢这种做法）

如果在 RPGLE 程序中，给某个变量赋值，但没有对相应的记录格式进行 WRITE 操作，编译会不通过，报 4030 字段未定义的错。

3、 关于打印时会错行的问题

有的时候，打印会错行，比如记录格式定义

0001.00+　　变量 A（比如说 10 个字符长）

0002.00+　　变量 B（比如说 5 个字符长）

变量 A 的起始处位于第一列

变量 B 在变量 A 的下一行，且起始处位于第三列

当变量 A='AAAAAAAAAA'， 变量 B='BBBBB'时，会打印出

AAAAAAAAAA

BBBBB

这很正常。

但当 A='　　A'或空时，本来我们是希望打印出：

A（或者这里就是一空行）

BBBBB

这样的效果，但是系统就会打印成为

BBBBBA 或

BBBBB　（没有打印出空行）

也就是说，当某一行的前面为空时，（好象空格也算空）如果下一行有数据，就会挤上来。解决办法是在将 A 字段的首位赋值为非空；如果想让客户看不见，可以考虑使用汉字指示器。想打印空行，也可以用这个办法。

又或者是将该字段的功能选项加上“SPACEB”，即打印前空一格，也可以解决这个问题。（报表的这个特点困扰我最久）

4、关于报表的测试

其实这个东西最简单，不过我以前一向很少去留意。

以前测试报表，尤其是测试换页的时候，都是老老实实的用打印机去打，打出来了再检查换页对不对，慢不说，还费纸。

其实我们进入到 SPLF 里，查看刚生成的报表文件，右上角就有当前的页数，行数，如第 1 页第 1 行，右上角就是 1/1，第 2 页第 3 行，就是 3/2，好象页数在后面。

在“position to”这里，+1 就是下一行，W+1 就是向右移一列，慢慢+，就可以看到换页是否正确了。

7.2 SAVF，备份与恢复

SAVF，全称 SAVE FILE，存储文件。可以将 SAVF 视为一个存储容器，它能够将指定的库，或指定的数据文件，或源代码保存在其中，有点类似于 UNIX 中的 TAR。

SAVF 只用于备份与恢复。虽然通常这些事情是系统管理员做的，但是如果环境许可的情况下，开发人员能了解这些命令，自行做做备份，就可以更好地对程序进行测试、保护源码。当然，在使用 RESTORE 命令时，一定要谨慎谨慎再谨慎，千万不能追求操作速度，切记切记。

1. 建立 SAVF

要使用 SAVF，首先我们当然需要建立一个 SAVF。如已有自己的 SAVF，可跳过此步。建立 SAVF 的命令是：

CRTSAVF FILE（库名/SAVF 名）

如果建立成功，就会在指定库中，生成一个空的 SAVF。

2. 清空 SAVF

使用 SAVF 前，必须保证 SAVF 是空的。SAVF 不能追加内容。新生成的 SAVF 一定的空的，不需要特别处理；

如果是已存在的 SAVF，需要使用 CLRSAVF 的命令，确保清空 SAVF

CLRSAVF FILE（库名/SAVF 名）

3. 将指定的库备份到 SAVF 中

备份库，使用下列命令来进行备份

SAVLIB LIB(要备份的库名) DEV(*SAVF) SAVF（SAVF 所在库名/SAVF 名）ACCPTH(*YES)

ACCPTH(*YES)，表示备份时，备份存储路径。也就是对应于数据文件，将其逻辑文件的相关信息也备份下来，会增加备份时间与备份空间；但恢复时，不需要对逻辑文件重新建立索引，可以省很多时间。所以在备份数据文件建议加上这个参数。当然，如果是备份源代

码，就不需要这个参数了。

4. 恢复已备份的库

RSTLIB SAVLIB（备份的库名） DEV(*SAVF) SAVF（SAVF 所在的库/SAVF 名） RSTLIB(恢复的库名)

RSTLIB 这个参数，表示恢复的库名，其默认值等于“备份的库名”。也就是说，如果我将 FHSLIB 整个库备份下来，再使用 RSTLIB 恢复，如果不更改 RSTLIB 中的值，那么将直接将 FHSLIB 整个都覆盖恢复；而如果指定 RSTLIB 的值为 OTHERLIB，那么将么把备份的 FHSLIB 的内容，覆盖恢复到指定的 OTHERLIB 库中。

5. 将指定的目标备份到 SAVF 中

SAVOBJ OBJ(目标名) LIB(目标所在的库) DEV(*SAVF) SAVF(SAVF 所在的库/SAVF 名)

一次可以备份多个目标。

如果是备份源码，那么 OBJ 就表示源码所在的 SRCFILE，MEMBER 项就是源码名。可以使用 F4 键，来备份更多的目标，或更多的源码。

6. 恢复目标

RSTOBJ OBJ(*ALL) SAVLIB(目标所在库) DEV(*SAVF) SAVF(SAVF 所在库名/SAVF 名) RSTLIB(恢复的库名)

与 RSTLIB 类似。当然，OBJ 选项使用*ALL，表示恢复 SAVF 中备份的所有的目标，也可以指定只恢复单个/多个目标。

7.3 菜单--MENU

如果有例子的话，做菜单其实也不复杂。我对菜单做得也不多，感觉对于一般开发人员来说，使用频率不是太高，还是简单说一下吧

随便找一个已存在的菜单，应该是包含 DDS 与 CMD 这两种 TYPE，COPY 一下；

然后用 17，或 19 进入 DDS。

用 19 进入时，可以像编辑报表一样，对菜单进行编辑；

用 17 进入时，在“Work with menu image and commands”选项中选择“Y”，然后用 F10 可以看到自行定义的每条输入项对应要运行的程序名，修改之；

用 17 进入后，再退出时，系统会自动提示编译菜单。

成功之后，查看属性为“CMD”的源，会看到里面也有自定义的输入项对应的要运行程序名。

GO 菜单名，就可以进入生成的菜单。

7.4 几个命令

WRKACTJOB

这个使用的频率应该是最高的吧，在这里只提一个比较有用的用法：Coding 的时候断线异常退出时，再登录之后，用 2 进入刚才编辑的程序，不是会报错“正在被使用”

嘛。除了等一会，一直等到它自动退出之外，还可以用 WRKACTJOB + 10，来查找刚才自己的那个进程(显示出来的命令应该是 STRSEU)，然后把它 KILL 掉就行了。

DSPFD

DSPFD + PF 名，查看文件的信息，包括文件的 MEMBER 数，总记录数，被删除的记录数，CCSID 等等。

DSPFD + LF 名，查看逻辑文件的键值，结合 DSPDBR，可以找出一个 PF 文件对应的所有 LF 文件的键值。

DSPFD + 库名 + *MBRLIST，可以看到该库之下所有的文件名。

这个命令可以使用*OUTFILE，将输出变成一个文件。灵活运用，将会是很多自行编写的工具的基础之一。

DSPFFD

注意，和上面的命令相差一个“F”，表示查看 PF 文件的字段信息。比如该 PF 文件共有多少个字段，每个字段的类型如何。

与 DSPFD 相似，DSPFFD 也可以使用*OUTFILE，将输出变成一个文件。同理，灵活运用，也是众多自行编写工具的基础。

MRGSRC

比较两个程序的差异。按照 A、B、A 的顺序，MRGSRC 时，A 程序在上面，B 程序在下面，白色的地方，即表示两个程序的差异。

F13 表示接受当前差异，F17 表示接受所有差异，F16 表示继续查找下一处差异。

在接受差异时，表示按照 B 程序，更改 A 程序。(即 B 程序不变)

把 SPLF 变成 MEMBER（源文件）

必须知道 SPLF 名，以及生成 SPLF 的 JOB、USR、NUMBER 名

```
CRTSRCPF     FILE(库名/SRCPF) RCDLEN(212) IGCDTA(*YES)
CHGPF        FILE(库名/SRCPF) CCSID(935)
CRTPF        FILE(库名/PF) RCDLEN(200) IGCDTA(*YES)
CPYSPLF      FILE(SPLF 名) TOFILE(库名/PF) +
                JOB(NUMBER 名 / USER 名 / JOB 名)    SPLNBR(*LAST)
CPYF         FROMFILE(库名/PF) TOFILE(库名/SRCPF) +
                TOMBR(MBR0001) MBROPT(*ADD) FMTOPT(*CVTSRC)
```

最后，SRCPF 下的 MBR001 就是 SPLF 转成的 MEMBER

当然，做为中间转换的临时文件，PF 文件会保留 SPLF 的信息。

把 MSGF 变成 SPLF

```
CHGSYSLIBL LIB(QSYS2989)
DSPMSGD      RANGE(*FIRST *LAST)    MSGF(MSGF 所在库名/MSGF 名) +
                DETAIL(*BASIC) OUTPUT(*PRINT)
CHGSYSLIBL LIB(QSYS2989) OPTION(*REMOVE)
```

这个命令，再结合上面的“把 SPLF 变成 MEMBER”，就可以把 MSGF 搞成 MEMBER，然后再 FTP 到 WINDOWS 上，就可以很方便地查找已定义的 MSG 信息了。

不过必须要有权限执行 CHGSYSLIBL 这个命令。

DSPOBJD

根据程序名，查找编译时源代码所在的库名、SRCFILE 名。

DETAIL 参数用“*SERVICE”，OBJTYPE 选项用“*PGM”

不过好象 RPGLE 的程序用这个命令查不出来，要用 DSPPGM 命令。

DSPPGM

根据目标

DSPPGM + RPGLE 程序名，参数用“*MODULE”，然后再选择 5，就可以看到编译时 RPGLE 程序时，源代码所在的库名，SRCFILE 名。

DSPDBR

根据物理文件查其对应的所有逻辑文件

DSPPGMREF

查找程序与 PF、LF 的关系。

PGM 参数用“*ALL”，OUTPUT 参数用“*OUTFILE”，OBJTYPE 参数用“*ALL”，然后执行，再输入输出文件名与所在库就可以了。然之后，就可以 SQL 在这个生成的文件中查找（生成的过程可能会有点慢，要耐心）。

在生成的文件中，字段 WHRFNM，表示记录格式名。举例来说，如果物理文件更改过，那么物理文件与逻辑文件就都重新编译过了，所以它们对应的的所有的程序都要重新编译，通常物理文件与逻辑文件使用相同的记录格式名（当然，如果不同就算了）这时就需要按记录格式名来查找。比如 SELECT * FROM 刚生成的文件 WHERE WHRFNM = “记录格式名”，找出所有涉及到这个记录格式名的程序。

在生成的文件中，字段 WHFNAM，表示文件名，包括物理文件与逻辑文件。比如说当我们只修改了某个逻辑文件时，那么当然是只需要重新编译与该逻辑文件有关的程序，也就是说只根据文件名来查找就足够了。

RTVCLSRC

如果编译 CL 程序时，不是刻意带*NONE 参数，那么一般来说 CL 程序都可以使用这个命令来反编译。具体用途试试便知

RGZPFM

重整文件，即回收已删除记录的空间。不过这个命令我没实际用过。

WRKMSGQ QSYSOPR

显示错误信息

CHGCMGDFT

更改某些命令的默认参数，如更改 CRTPF 的 WAITRCD 参数：

CHGCMDDFT CMD(QSYS/CRTPF) NEWDFT(‘WAITRCD(*IMMED)’)

SBMJOB

将任务提交后台处理。

普通程序的流程中，如果 A 程序 CALL 了 B 程序，那么 A 程序必须等待 B 程序运行结束之后，才会继续执行 CALL 之后的语句；

如果是使用 SBMJOB 的话，那么 A 程序就将 B 程序提交到后台去运行，不等待 B 程序结行完毕，直接继续向下运行。

SBMJOB 命令仅仅只是表示将任务提交给后台，所以此句运行完毕，也仅表示后台已开始运行，并不表示运行的程序结束。

Submit Job (SBMJOB)

Type choices, press Enter.

Command to run . . . . . .

..

Job name	*JOBD	Name, *JOBD
Job description	*USRPRF	Name, *USRPRF
Library		Name, *LIBL, *CURLIB
Job queue	*JOBD	Name, *JOBD
Library		Name, *LIBL, *CURLIB

如上所示，第一行 Command to run 处，填写需要提交后台运行的命令语句（通常是 CALL 某个程序）；

Job name 表示的，是显示在屏幕上的，运行的程序的名字，可以随便写，只是用来标识用的；

Job description， 填写对应的 JOBD（如果想将程序提交到指定的子系统下，那么此处填写子系统对应的 JOBD，Job queue 处填写子系统对应的 JOBQ），如果不填，即表示默认为当前用户的 JOBD，JOBQ

7.5 关于代码风格的几点想法

1、 绝大多数情况下，不做硬性要求，一切都以维护时程序的易读性为主导。

2、 一定要在程序最开始，简要说明程序实现的功能，输入输出参数，这个必须的。

3、 临时变量的命名：

这个应该算是最具有可规范性的了。总之最好就是一望之下，就知道这个临时变量是代表什么意思。变量名的长度可以不做局限，当然最好不要太长。通常项目开发对此都会有明文或潜在的规范，多参照即可。

4、 定义临时变量的位置：

在首次使用该临时变量前定义，同时加汉字注释，简单说明这个临时变量在什么情况下，应该赋什么样的值。（当然了，如果是多个用途相似的临时变量，可以用一行注释来搞定，不强求每个变量都要有一行注释）

见过的大部分规范里面，都是统一在程序开始处定义一大堆临时变量，规范的

同时，总觉得用起来不够方便，比如说不知道这个临时变量什么时候使用，代表什么意思，怎么使用，该怎么赋值等等。

5、 子过程的使用：

主程序最好只写主流程，将具体处理交给各子过程；同时在调用子过程之前，加注释，说明这个子过程的处理功能。而子过程代码最开始，也要有这个子过程的功能说明，最好详细一点。

把子过程尽量写成类似于 CALL　程序的感觉，也就是自己定一个输入输出接口。当然，这个并不是真的输入输出接口参数，因为子过程中的变量在整个程序中都可以用到，只是说类似。比如说，某个字段，是子过程中需要使用到的关键的一个字段，那我们就可以将其做为输入字段，并在整个子过程的代码中，尽量不要去改它的值；再比如说，子过程的功能是计算出一个金额，那我们就可以把这个金额字段做为输出字段，在使用子过程之前将其清零。然后，在子过程前，加注释说明输入输出字段。

这样做的意义在于：维护修改代码的便利性，COPY 代码的便利性。

6、 视觉上的分隔：

RPGLE 的程序，写出来都是一坨坨的，尤其是 IF 语句，用多了的确很难分清逻辑判断到底是怎样。我通常对代码都是采用视觉上的分隔，因为我觉得这样最直观。比如说，主代码段与子过程段之间，用全行“*”来做分隔；各段之内，也用长短不一的“*”来分隔，表示不同情况的处理，或表示当前是几层之内的“IF”。这个规律现在还没有总结出数字化的东西来，大部分情况下还是凭感觉，以后有时间做做统计看看。

总之最后的效果，是从视觉上，将不同的处理情况隔开。比如说

```
****************
**注释
C                   IF    XXXX
C                   ENDIF
****************
```

这种方式之下的 ENDIF，就比较好找了吧。

“*”当然还可以再打得长一点，如果层次多了，还可以用单横线，等号，诸如此类。

7、 注释

注释的重要性毋庸致疑，这里单指以下情况的注释：

如果写程序时，某些需求不是很明确，或自己对这个需求的理解不是很清晰，或者是用户特意要求某种情况之下不按常规方式处理，又或者是处理判断在逻辑上比较复杂比较绕，那么不妨把自己当时的想法或客户要求也写在注释中。事实上，有不少明显的 BUG，就是通过这种注释发现的。