

Genero Business Development Language

技术手册

(Version 5.1)



T 系列产品中心

Genero Business Development Language

技术手册

作者/ T 系列产品系统工程部
发行/ 鼎捷集团 集团研发 T 系列产品中心
地址/ 鼎新计算机 新北市新店区中兴路一段 222 号
鼎捷软件 上海市闸北区共和新路 4666 弄 1 号
网址/ <http://www.dsc.com.tw>
客服/ tiptop@dsc.com.tw
出版/ 2004 年 07 月初版、2021 年 04 月 5.2 版

声明/ 本书繁简中文版、英文版版权为鼎捷集团所有，
未经本公司授权任意拷贝、引用均属违法。

书中引用之商标及产品名称分属各原产公司所有，本书引用纯
属介绍之用，并无侵害之意。

(版权所有，翻印必究)

製造業解決方案

完整產業 e 化解決方案，攜手客戶邁入協同運作 ERP II 新世紀



製造業 e 化完整解決方案

- ◆ TIPTOP e-Business Total Solution
- ◆ Workflow ERP II Total Solution
- ◆ 鼎新 SmartERP 企業資源規劃系統
- ◆ EasyFlow 電子流程管理
- ◆ e-B Chain B2B 電子商務應用系統
- ◆ 鼎新 CRM 客戶關係管理系統
- ◆ V-Point BI 商業智慧
- ◆ V-Point EIS 主管資訊系統
- ◆ 鼎新 BSC 平衡記分卡
- ◆ Smarteam PLM 產品生命週期管理
- ◆ SCM 供應鏈管理解決方案
- ◆ MES 工廠營運管制系統

Genero BDL 技术手册目录

第一章 Genero BDL 架构

| | |
|---------------------------------------|-------|
| Genero BDL 概观..... | 1 - 1 |
| Genero BDL 程序组成..... | 1 - 2 |
| 编译（Compile）、连结（Link）执行（Execute） | 1 - 3 |

第二章 开发辅助工具 Genero Studio 与画面档（FORM）设计

| | |
|--------------------------------------|--------|
| Genero Studio..... | 2 - 1 |
| Configuratio（系统设定） | 2 - 2 |
| Genero Studio Preferance（偏好设定） | 2 - 5 |
| Database Browser..... | 2 - 6 |
| File Browser..... | 2 - 10 |
| Code Editor..... | 2 - 11 |
| Graphical Differential..... | 2 - 12 |
| Form Designer..... | 2 - 13 |
| 画面档（FORM） | 2 - 14 |
| 画面档设计 | 2 - 17 |
| Container（容器物件） | 2 - 18 |
| Grid..... | 2 - 19 |
| ScrollGrid..... | 2 - 19 |
| MFArray..... | 2 - 20 |
| Table..... | 2 - 21 |
| GroupBox..... | 2 - 21 |
| PageControl..... | 2 - 22 |
| HRec..... | 2 - 22 |
| 版面排列物件..... | 2 - 23 |
| Widgets（元件） | 2 - 24 |
| Edit..... | 2 - 24 |
| TextEdit..... | 2 - 25 |
| ButtonEdit..... | 2 - 25 |
| DateEdit..... | 2 - 26 |
| ComboBox..... | 2 - 26 |
| RadioGroup..... | 2 - 27 |
| CheckBox..... | 2 - 27 |
| Silder..... | 2 - 28 |
| SpinEdit..... | 2 - 28 |

| | |
|---|--------|
| TimeEdit..... | 2 - 28 |
| Image..... | 2 - 29 |
| ProgressBar..... | 2 - 29 |
| Label..... | 2 - 30 |
| Button..... | 2 - 31 |
| Canvas..... | 2 - 31 |
| HLine..... | 2 - 31 |
| 其他项目..... | 2 - 32 |
| TM (TopMenus) | 2 - 32 |
| TB (Toolbars) | 2 - 33 |
| AD (Action Defaults) | 2 - 34 |
| SR (Instructions: Screen Records) | 2 - 35 |
| Style (Styles) | 2 - 36 |
| Tab Index (文件排列顺序) | 2 - 37 |
| Alignment (对齐) | 2 - 38 |
| DataControl..... | 2 - 38 |
| 画面档验证、编译、预览作业流程..... | 2 - 42 |

第三章 变量及运算

| | |
|--|--------|
| 变量的定义..... | 3 - 1 |
| 变量的型态..... | 3 - 2 |
| 变量的集合 (Records) | 3 - 4 |
| 变量的设定..... | 3 - 5 |
| 初始化一组变量的值..... | 3 - 5 |
| 预定义完成变量 (PRE-DEFINED Variable) | 3 - 6 |
| 常数的定义 (CONSTANT) | 3 - 6 |
| 预定义完成常数 (PRE-DEFINED Constant) | 3 - 6 |
| 表达式..... | 3 - 7 |
| 全域变量指定..... | 3 - 9 |
| 变量的生命周期..... | 3 - 10 |
| USING..... | 3 - 11 |

第四章 程控流程

| | |
|-------------------------|-------|
| Module 程序架构..... | 4 - 1 |
| MAIN () 函式..... | 4 - 1 |
| 一般 FUNCTION () 函式..... | 4 - 3 |
| 报表结构 REPORT () 函式..... | 4 - 3 |
| 注解符号..... | 4 - 3 |

| | |
|---------------|--------|
| CALL..... | 4 - 4 |
| RETURN..... | 4 - 4 |
| IF..... | 4 - 6 |
| CASE..... | 4 - 7 |
| FOR..... | 4 - 9 |
| WHILE..... | 4 - 10 |
| CONTINUE..... | 4 - 10 |
| EXIT..... | 4 - 11 |
| SLEEP..... | 4 - 12 |

第五章 WINDOWS 与 FORM

| | |
|---------------------|-------|
| WINDOWS 与 FORM..... | 5 - 1 |
| OPEN WINDOW..... | 5 - 1 |
| CLEAR..... | 5 - 2 |
| CLEAR FORM..... | 5 - 2 |
| CLOSE WINDOW..... | 5 - 3 |
| CURRENT WINDOW..... | 5 - 4 |
| OPEN FORM..... | 5 - 5 |
| DISPLAY FORM..... | 5 - 5 |
| CLOSE FORM..... | 5 - 5 |

第六章 功能表

| | |
|------------|-------|
| 建立功能表..... | 6 - 1 |
|------------|-------|

第七章 DISPLAY 与 INPUT

| | |
|-----------------------|-------|
| DISPLAY 与 INPUT..... | 7 - 1 |
| DISPLAY 指令..... | 7 - 1 |
| INPUT 指令..... | 7 - 3 |
| INPUT 架构..... | 7 - 5 |
| INPUT 控制点..... | 7 - 6 |
| WITHOUT DEFAULTS..... | 7 - 7 |
| PROMPT..... | 7 - 8 |

第八章 CURSOR 的应用

| | |
|-----------------------|-------|
| CURSOR 资料的查询、修改..... | 8 - 1 |
| CONSTRUCT..... | 8 - 1 |
| PREPARE..... | 8 - 3 |
| 资料的查询..... | 8 - 4 |
| SCROLLING CURSOR..... | 8 - 5 |

| | |
|-------------------------------|--------|
| FETCH 叙述..... | 8 - 6 |
| NON-SCROLLING CURSOR..... | 8 - 8 |
| FOREACH 叙述..... | 8 - 8 |
| 资料的锁定..... | 8 - 9 |
| LOCKING CURSOR..... | 8 - 9 |
| USING 的使用时机..... | 8 - 11 |
| TRANSACTION..... | 8 - 12 |
| 大量执行同一 SQL 指令 (EXECUTE) | 8 - 13 |
| 大量执行新增指令 (PUT...FLUSH) | 8 - 14 |
| 第九章 ARRAY 的应用 | |
| ARRAY (阵列) | 9 - 1 |
| INPUT ARRAY..... | 9 - 4 |
| DISPLAY ARRAY..... | 9 - 9 |
| 第十章 REPORT 撰写 | |
| REPORT 撰写..... | 10 - 1 |
| 启动 REPORT..... | 10 - 1 |
| REPORT FUNCTION 的组成..... | 10 - 3 |
| OUTPUT Section..... | 10 - 4 |
| ORDER BY Section..... | 10 - 4 |
| FORMAT Section..... | 10 - 5 |
| FORMAT 内可用的指令..... | 10 - 5 |
| 打印时可运用的表达式或函数..... | 10 - 6 |
| 群组函数..... | 10 - 6 |
| 第十一章 Debugger 的应用 | |
| Debugger 的使用..... | 11 - 1 |
| 除错模式的常用指令..... | 11 - 1 |
| 附录 A 相关参考信息..... | |
| 附录 B 上课地点位置图..... | |



Chapter 1

Genero BDL 架构

Genero BDL 概观

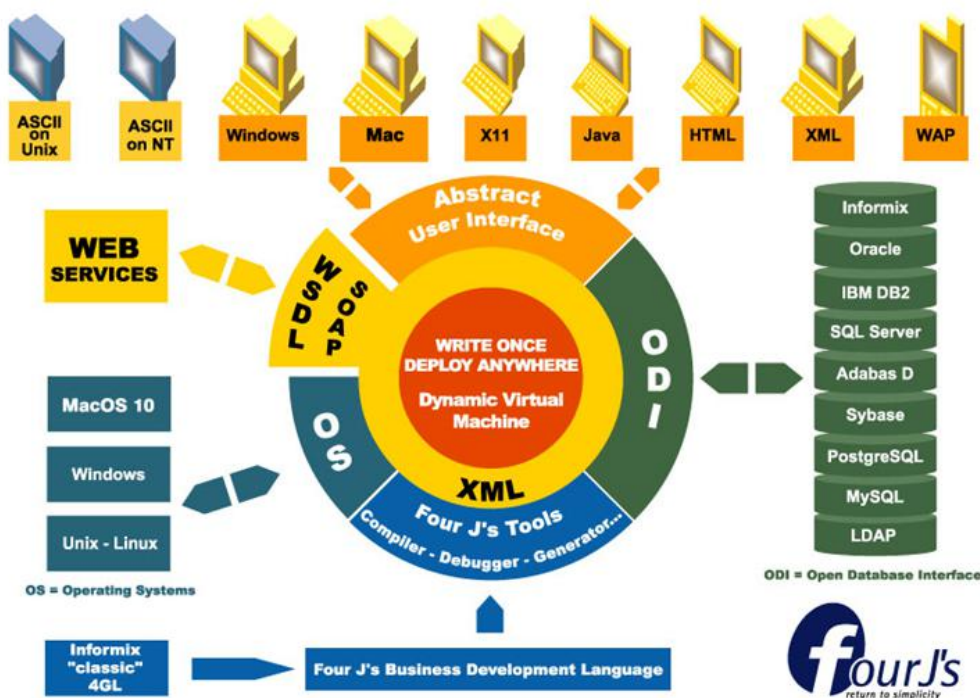
Genero BDL 语言，为法国的 FourJS (<http://www.fourjs.com>) 公司于 2004 年所提出。其前身即为 INFORMIX-4GL 语言。

INFORMIX-4GL 语言属于第四代架构的语言，其优点在于构成程序的语法和英文近似，可以大幅减少学习的时间，但仅能使用于 INFORMIX 数据库的控制上。

FourJS 则是取其优点，致力于扩张后端可连结的数据库种类，开发出 BDL 语言，使之可串接多种不同形式的数据库，如 IBM DB2、INFORMIX、MySQL、ORACLE、SQL Server 等

2004 年，因应图型化界面已成为计算机作业的主流模式，推出了 Genero BDL 语言，此语言除承袭旧有优点外，更加入一些新的特点：

- 切分为 Client、Server 架构（GDC 与 fgl），增进执行效率
- 以 XML Bsa 做为 Client 及 Server 端资料传递的架构
- 支援更多不同平台（OS）及数据库系统
- 可在执行阶段动态调整画面输出的格式（Layout Styles）
- 在部份新增的功能中引入基本的物件（Object）概念

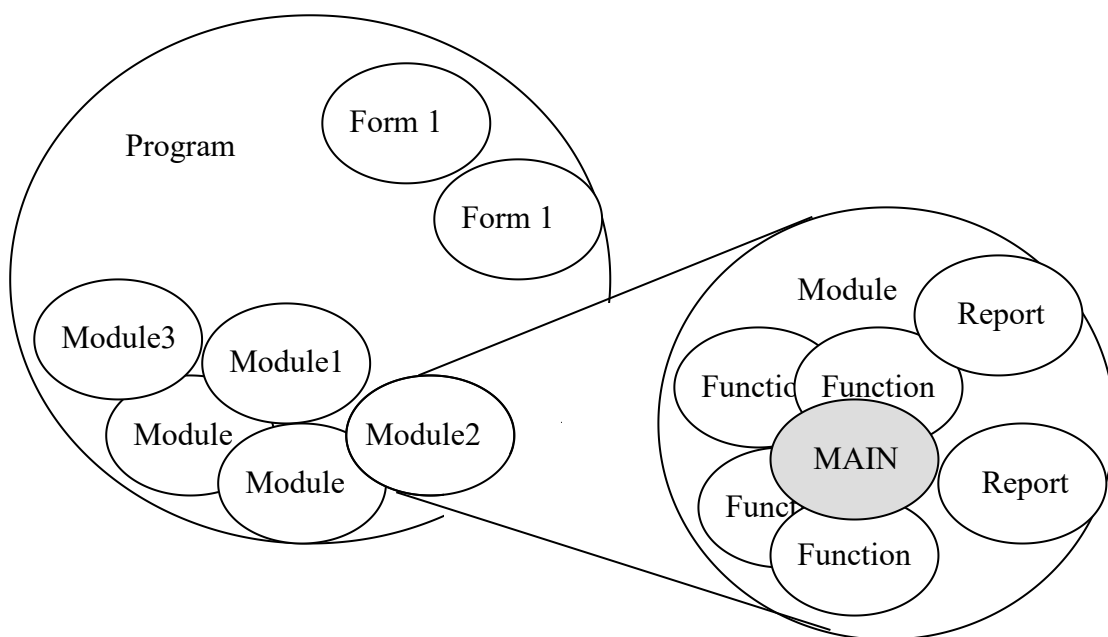


Genero 架构示意图（本图取材自 FourJS 网页）

Genero BDL 程序组成

在 Genero BDL 语言架构中，仍是将程序逻辑、与画面架构视为不同的控制项目，因此采行分别撰写的方式。一般通称一支一支的小程序（4GL）为 **Module**，称画面结构（PER）为 **Form**（注 1），组合后可独立执行的作业为 **Program**。

整理以上的说明，我们可以得到『Genero BDL 独立的 Program，组成可区分为 Module 和 Form。』的结论。



Genero BDL 作业（Program）组成架构概略示意图

由上图可知，**Program** 可由许多的 **Module** 与 **Form** 构成。**Module** 使用的副档名为『4GL』，**Form** 使用的则为『PER』，因此以下以 4GL 与 PER 来代称 **Module** 与 **FORM**。

单一的 4GL 由一个或一个以上的 **Function**、**Report** 组成。

一个完整的 **Program** 中，必需指定一个特定 **Function** 作为执行入口（注 2），此特定的 **Function** 即为『Main』（如上图中的 **Module 2**）。

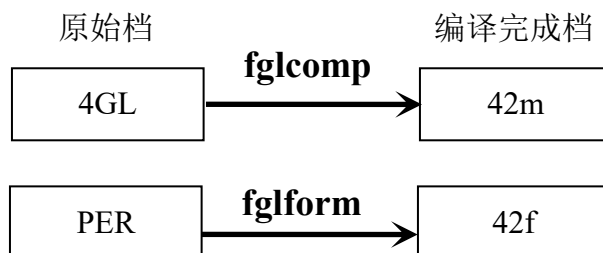
注 1：BDL 预设画面原始档为纯文字格式的『PER』档；当系统采用 Genero Studio 作为辅助开发工具时，编辑的画面始支援 XML 格式的『4FD』档。

注 2：完整的作业中只能存在单一的『Main』，若有一个以上时，即会造成编译或连结的错误。

编译（Compile）、连结（Link）、执行（Execute）

当程序（4GL）及画面（PER）编写完成后，还需经过编译（Compile）、连结（Link）。

编译流程



程序（Module）的编译（Compile）

指令：fglcomp

```
> fglcomp test1.4gl
```

程序的原始档附档名必为【4gl】（小写），经过编译后会产生附档名为【42m】的档案。

画面（Form）的编译（Compile）

指令：fglform

```
> fglform test.per
```

画面的源文件附档名为【per】（小写），经过编译后会产生附档名为【42f】档案。

注：Genero Studio 编辑的画面原始档，附档名为【4fd】，编译后亦为【42f】档案。

指令：gsform

```
> gsform [-i] [-keep] test[.4fd]
```

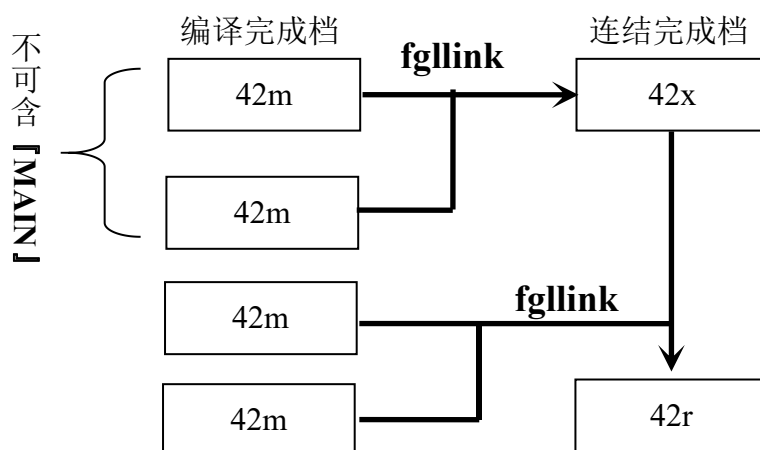
-keep 选项会保留编译过程中产生的【per】，-i 会强制覆写存在的【per】档。

连结流程

若该作业为仅靠一支 4GL 原始档程序即编写完成时，则可以略过连接（Link）的程序，直接参考『执行（Execute）』说明来执行作业。

若该作业被切分为许多的子程序（或称子程序、SubFunction；TIPTOP GP 即属此情形），则可以在执行连结作业前，预先打包成为一组动态连结函式库（注）（DLL，Dynamic Link Library）（附档名必需为【42x】），待打包完成后，再与原始主作业进行连结，以成为一支完整作业。

注：被打包成 42x 格式的所有原始 4GL 中，均不可含有【MAIN()】函式。



程序（Module）的连结（Link）

指令：fgllink

```
> fgllink -o test.42r test1.42m test2.42m
```

```
> fgllink -o test.42x test1.42m test2.42m
```

Link 需指定产生附档名为【42r】的档案，或产生附档名为【42x】的函式库档。

画面档不需要执行连结。

执行程序

指令: **fglrun**

```
> fglrun test.42m
```

不管是『42r』或『42m』, 均可以利用此方式直接执行作业, 唯一需注意的, 就是此执行作业『仅能含有单一的 **MAIN()** 函式』。

作业执行前, 需先开启使用者端的 Genero 桌面客户端软件 (GDC: Genero Desktop Client) (注一), 以令『fglrun』可以与客户端的『GDC』进行沟通 (注二)。

MEMO:

注一: GDC 为 Genero 的桌面端应用程序, 安装时必须视作业系统不同而安装不同版本, 现提供 MS Windows (98 以上版本适用)、Linux KDE、MAC OS X 等版本。如果不在上列范围, 则需使用 Web 方式登入, 须在主机端 Web Server 安装 GDC ActiveX 版, 以用 Web Plug-in 方式开启 GDC。

注二: fgl 与 GDC 均有各自独立的版本编号, 本手册以 fgl 版本『2.02.02』为参考范例, GDC 搭配上以『2.02.02』为主, 基本大原则为两软件间前两码大版本号须为一致即可互相搭配, 如『2.02』或『2.03』。

fgl 版本可利用『fpi』进行确认:

```
>fpi
Four J's Development Tools

Genero Business Development Language
Version 2.00.1a
Build 872.101
(c) 1989-2006 Four J's Development Tools
```

.

执行程序画面范例

以下画面取用自 TIPTOP GP 系统的『一般订单维护作业 (axmt410)』，我们将藉由此一作业画面，定义后续将看到的一些荧幕区块名称。

最上方标示作业名称即为 Window Title



红色区块即为
Ring Menu 区块，系统
会将映对不到 ToolBar 的
ACTION 列于此处（参考下一章），
圆圈处即为滚动钮



Chapter 2

开发辅助工具 **Genero**
Studio

与

画面档（**FORM**）组成

Genero Studio

Genero Studio 是由 Four J's 发布的 Genero 整合开发环境作业环境（注一）。

此工具由下列七种界面环境组成（注二）：

Database Browser：档案结构浏览器

Project Manager：项目管理工具

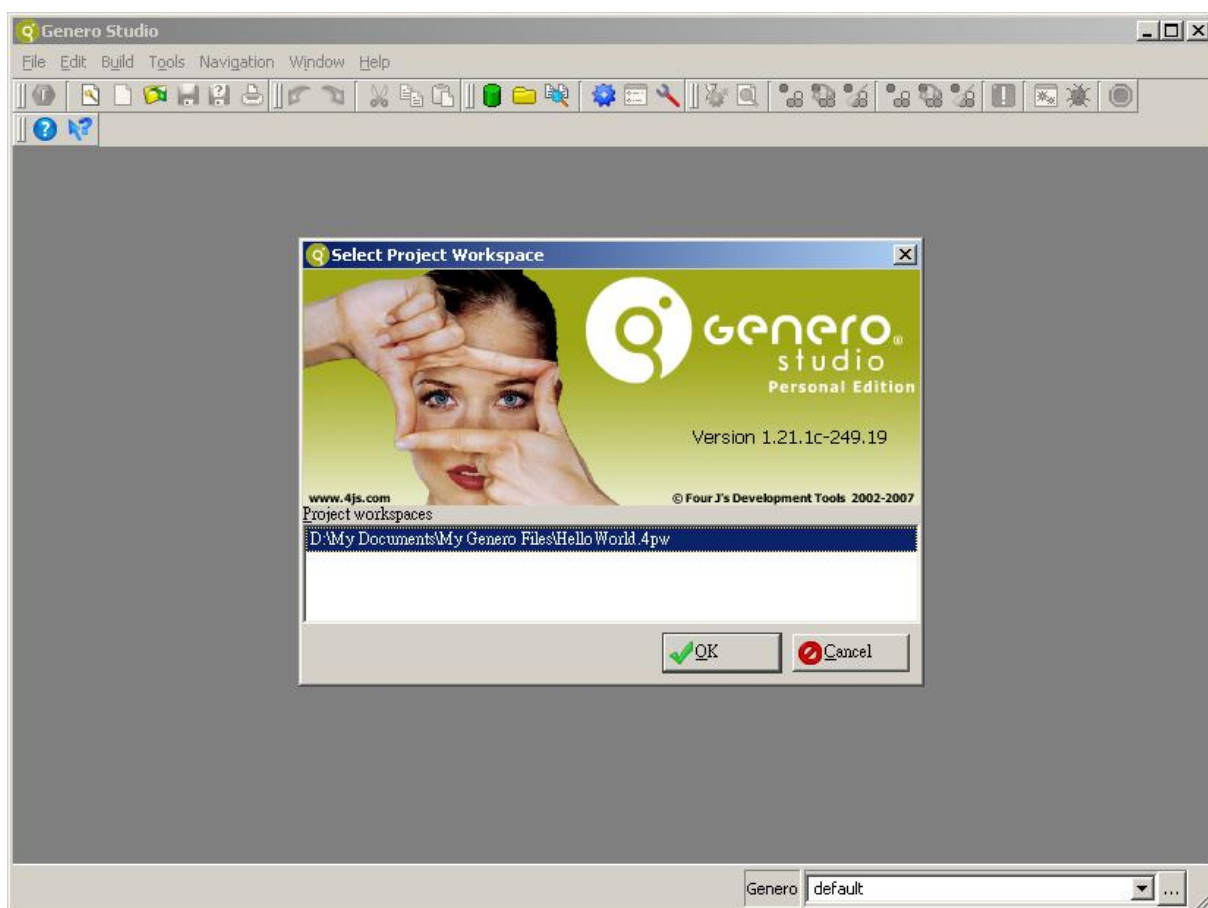
File Browser：档案浏览器

Code Editor：程序码编辑器

Graphical Debugger：图形化除错工具

Graphical Differential：程序码差异比对工具

Form Designer：画面设计工具



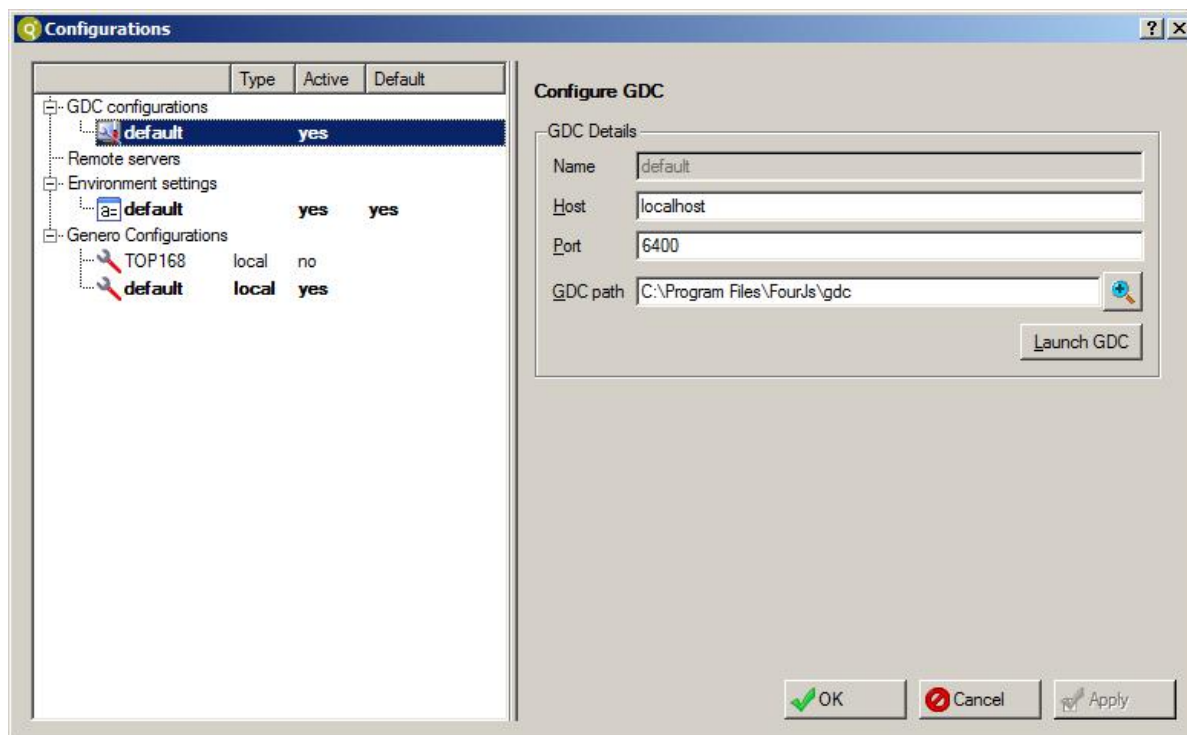
注一：编译程序时需要搭配 Genero License

注二：在 TIPTOP GP 中，编辑画面档会使用到的工具为 **Form Designer**、**Database Browser**、**File Browser**。

Configurations (系统设定)

初次进入系统时，需要对 Studio 进行系统设定：包含 **GDC 安装路径指定**、**环境变量指定**，以及**编译环境指定**等。

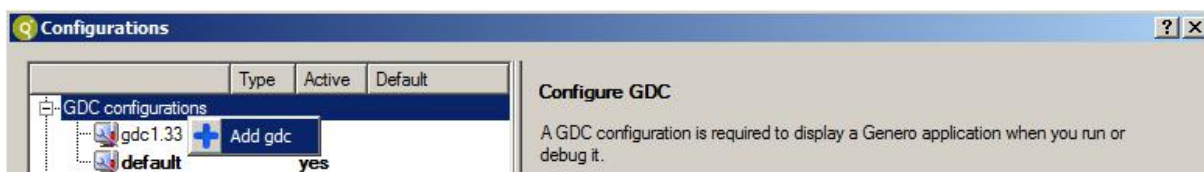
进行设定时，请按下工具列中的按钮 ，系统即会开启以下画面提供设定。



图一： GDC configurations 设定画面

■ GDC configurations:

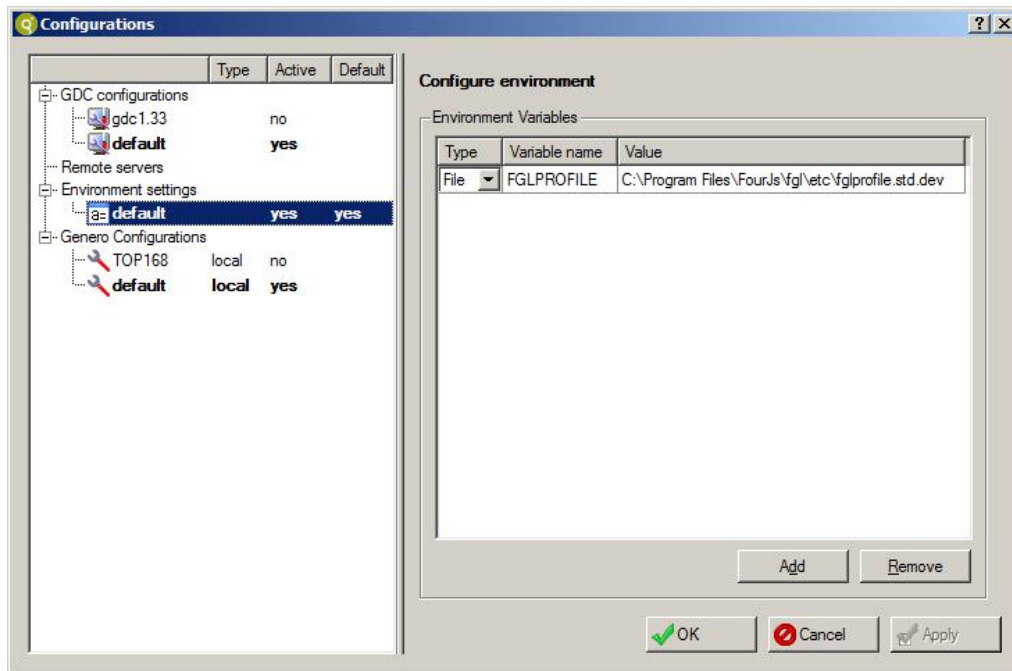
本选项主要在指定 GDC 安装位置 (GDC Path) 及预设使用的通讯埠 (Port)。本选项允许建立多组参数 (但 default 值一定要输入)，当建立多组参数后 (如下图二)，在执行选单『Tools』→『Launch GDC...』时，系统会开启对话框 (如下页图一)，询问要使用哪一份 GDC。
(若选择『Launch GDC』，则以 default 项目进行开启)。



图二：以右键进行新增设定



图一：选单『Tools』→『Launch GDC...』时，系统会开启对话框询问



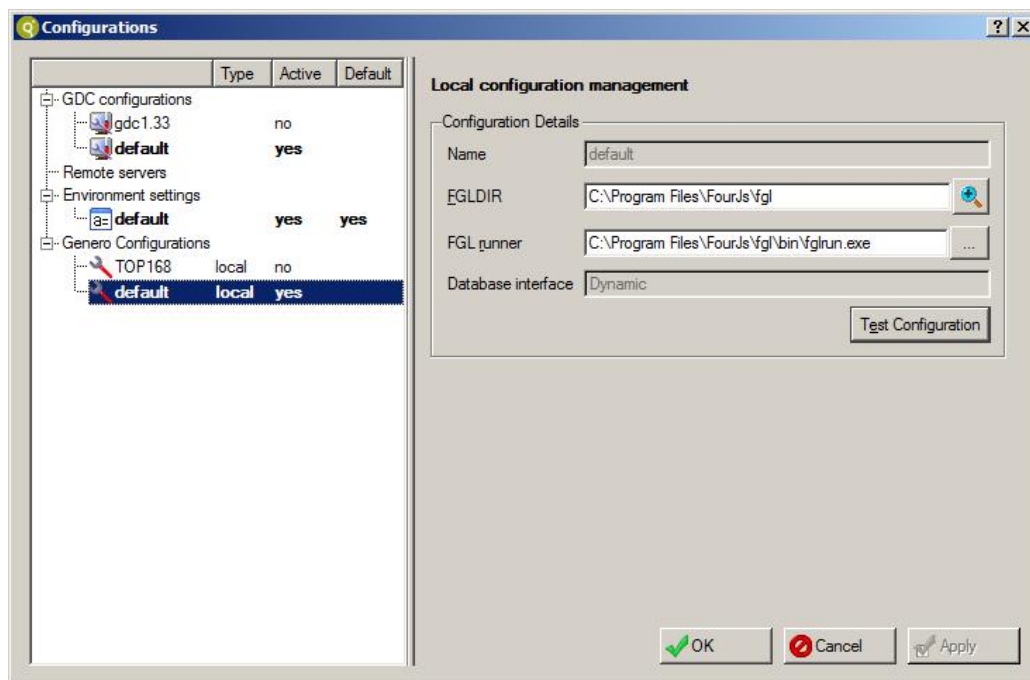
图二：Environment settings

■ Environment Settings:

本选项主在设定系统开启时，必须应用的环境变量项目（如 FGLPROFILE）。

设定时必须了解：该变量需纪录的型态是 Value、File、Directory、Path（如 URL）、或 Inherited。因为不同的选项，会令 Value 栏位中开窗功能对应不同的项目。

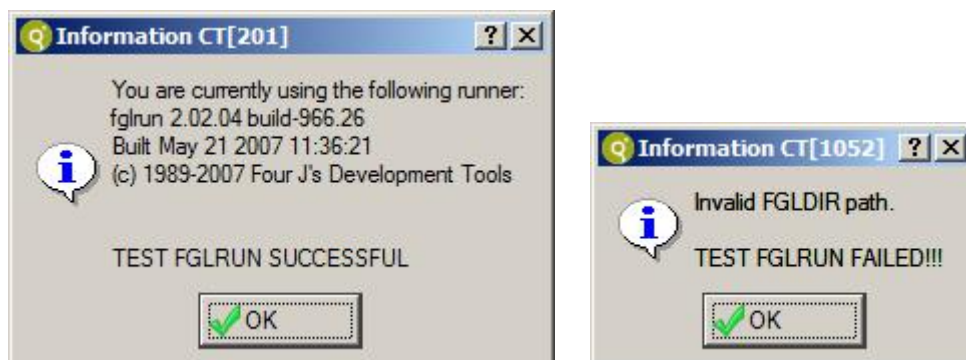
此处环境变量不可设定『FGLDIR』，因为此变量属于『Genero Configurations』所设定，若在此处设定，系统会出现错误讯息。



图一：Genero Configurations

■ Genero Configurations:

本选项在设定 Genero Compiler 位置, 本设定一样可以允许使用者设定多组编译器 (注)。设定完成后, 可利用右侧的『Test Configuration』来验证设定的系统环境是否已经可以完成程序的编译。

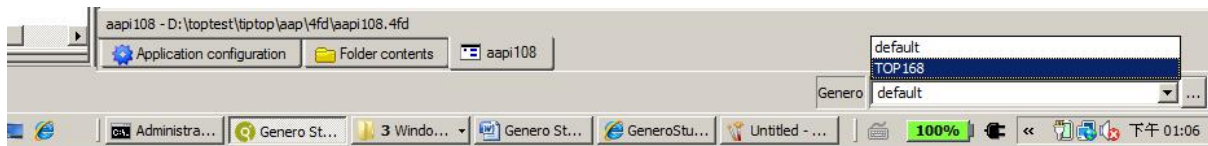


图二：测试正确（左图）、测试失败（右图）范例

注：由于编译时需要使用 Genero License。若指定的 fgl 并未安装 License 时, 进行『Test Configuration』均不会成功。


TIPTOP GP 5.0 编译相关程序及画面时, 均有本身作业之流程。若为编译 TIPTOP 相关作业时, 请参照标准作业流程进行。

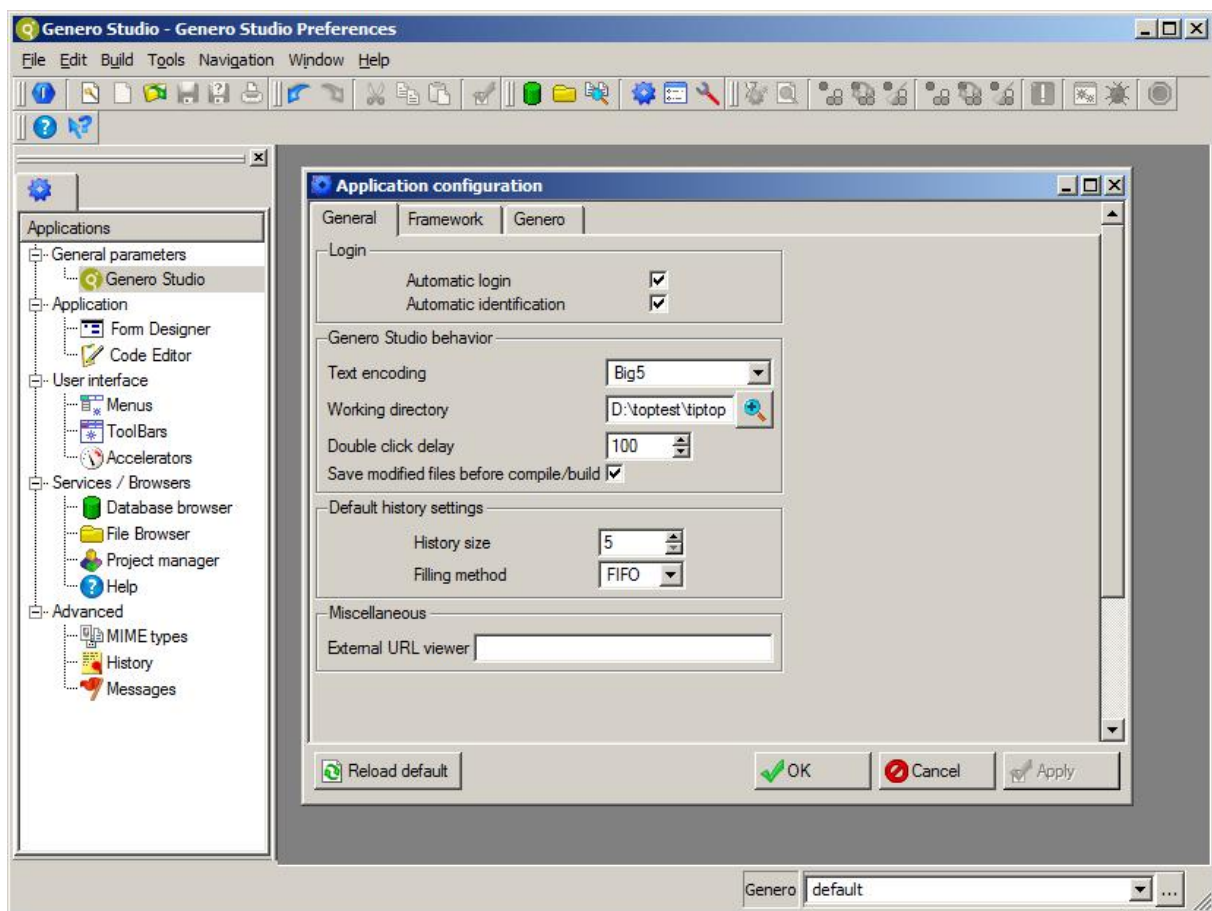
在编译过程中，可以由画面的右下角（如下图）进行设定调整。



图三：如右侧处，编译程序时可以指定使用的 fgl 环境




Genero Studio Preference（偏好设定）

除本身环境变量需要设定外，Studio 也提供其他内部 Application 的参数设定界面。启动此功能可以直接用工具列上的  功能开启，或由下拉式选单中选择『Window』→『Application configuration』开启。



开启后，可以由左侧的 panel 区（如上图所示）选取需要调整参数的 Application 后，在右侧的设定区定值。若需要回复为出厂设定时，请以『Reload default』功能回复。

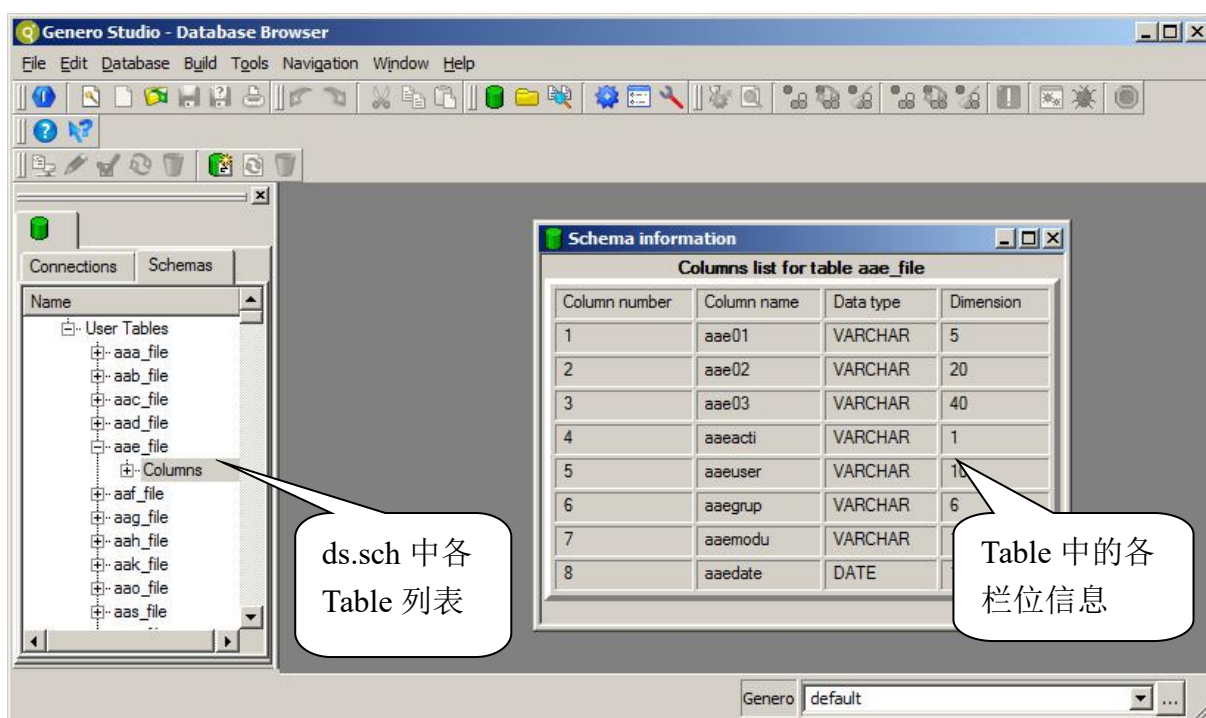
由于此部分依原系统出厂设定值即可使用，建议若有特别需求时再进行调整即可。相关的参数设定，可以按下『F1 键』，查阅在线说明。

Studio 以七种界面环境组成，但有部分界面工具是由系统自动判断启动（如点开 4gl 档案时，系统会呼叫 Code Editor；而点开 4fd 时，系统会打开 Form Designer 界面），因此在工具列上，只有『Database Browser 』、『File Browser 』、及『Graphical Differential 』等三种界面图示。

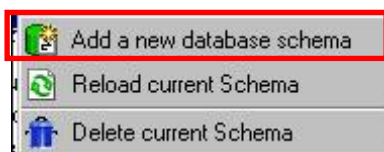
Database Browser

Database Browser 界面在提供设定 Studio 连接数据库，及检视数据库结构（Schema）的工具。

编辑或编译时会用到的 schema，都必须透过此界面先加到 Genero Studio 中。Studio 再将 schema 储存在 XML 设定档案内。这些设定档可被 Studio 下的其他 Applications 使用（如：Code Editor、Form Designer、Project Manager）。



从.sch 档建立 schema



从选单选取『Database』→『Add a new database schema』，或是在浏览面板的 Schemas 页签中按鼠标右键，并在弹出式选单中选取『Add a new database schema』。

出现如右下方的对话视窗时，即开始进行新增作业。

Schema name 请输入程序中需使用的 database name，建议参照相关设定档定名。（TIPTOP 预设设为 ds）

* 『Create new connection』选项主要在新建一个与数据库的连线，以便于让 Studio 可以自动登入数据库进行档案结构的撷取。

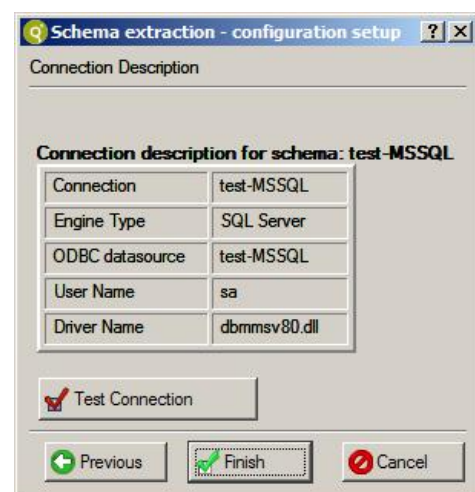
使用此方法必须先要在系统上建立 ODBC 的 DSN 连线，再搭配相关系统参数，让 Studio 可以登入。

参数中须指定要使用的 driver name，请参照下表所示：

| Database Type | Driver library prefix | Example |
|---------------|-----------------------|--------------|
| Informix | dbmifx | dbmifx9x.dll |
| ORACLE | dbmora | dbmora92x.so |

设定成功时，即会出现右侧对话框，表示 Studio 已经和数据库连线成功。并随即开始下载指定资料库的结构（schema）。

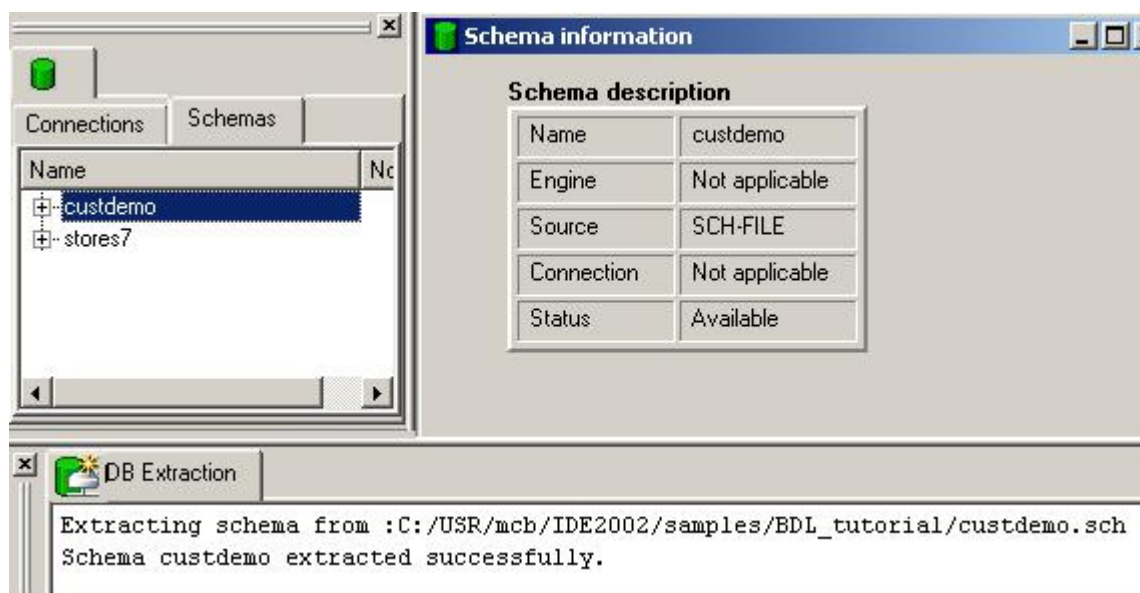
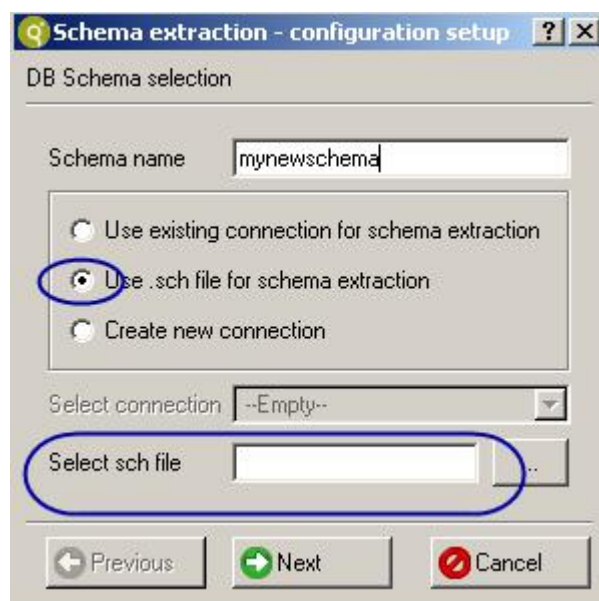
* 『Use existing connection for schema extraction』，主要在使用已建立的连线，来读入不同 database 的 schema。



- * 『Use .sch file for schema extraction』
选项，然后在 Select sch file 按右边的按钮选取.sch 档。

按下『Next』后，会出现确认的对话视窗，此时按『Finish』就会开始产生 schema

产生完成后，在输出面板会显示讯息（如下图），并且在 Schemas 页签可以看到产生的 schema。



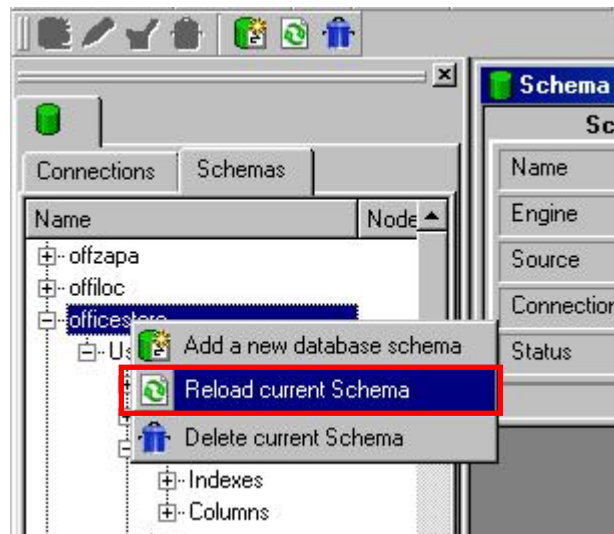
注：TIPTOP GP 中，因为系统安全性考量，建议不要使用由 Studio 直接连线到资料库抓取 schema 的方式。

透过下载 TIPTOP 主机上『\$TOP/schema』目录下的 ds.sch 档，以『读入.sch 档』的方式，建立 schema。

重新载入 schema

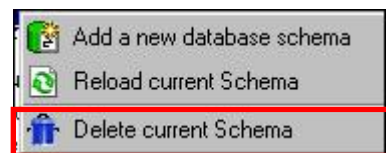
在浏览面板的 Schemas 页签中按鼠标右键，在选单中选取『**Reload current Schema**』执行。

接下来会出现与新增 schema 时相同的对话视窗，执行『**Next**』后再执行『**Finish**』。



删除 schema

在浏览面板的 Schemas 页签中按鼠标右键，在选单中选取『**Delete current Schema**』，或是选取 Database 选单中的『**Delete current Schema**』，或是选取图示列上的小图示执行。



接着会出现删除的提示视窗（如右图）

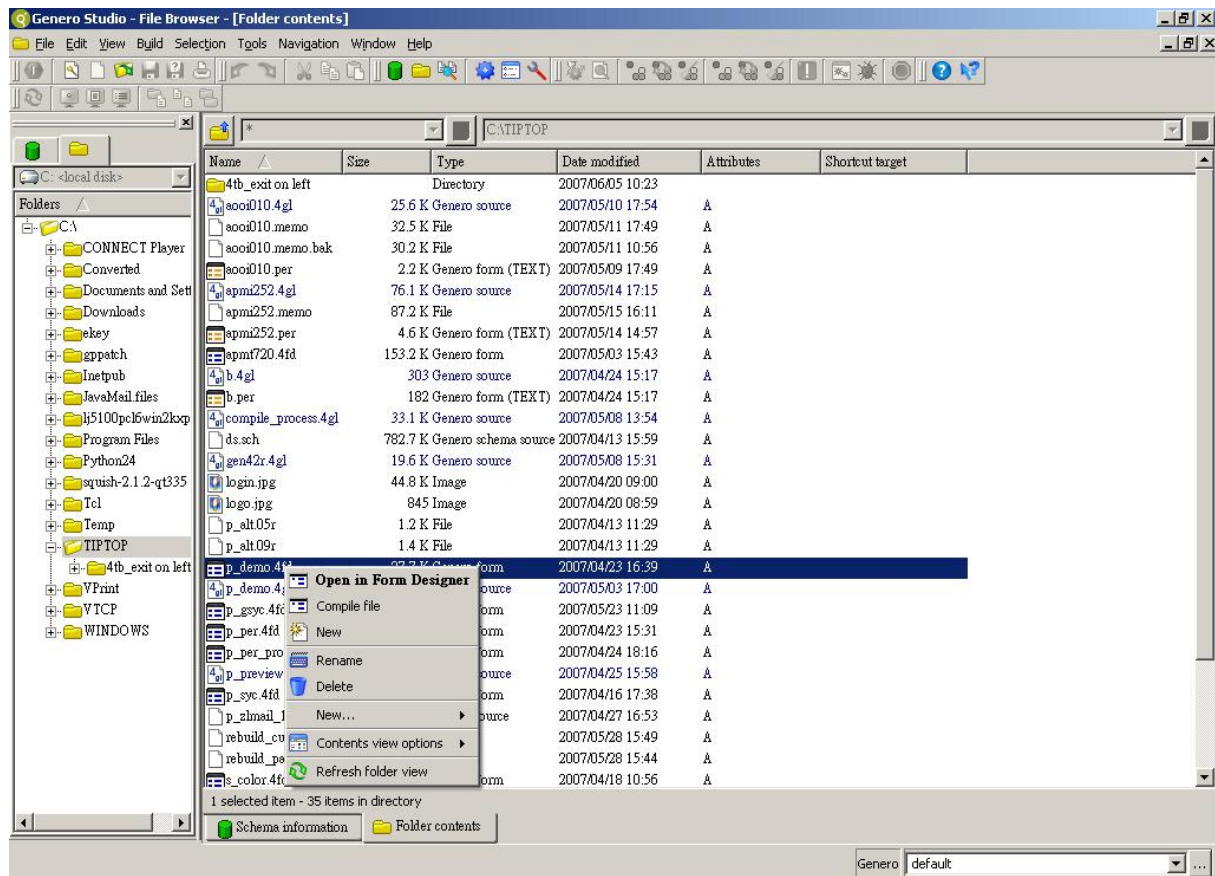
选『**OK**』确定删除。



注：若是以下载档案方式读入 schema，建议在每次要重新作业时均做一次删除，并重新读入（或是 reload）的动作，以避免因资料表异动，而造成开发的画面或程式档无法移入 TIPTOP 主机使用。

File Browser

此界面为 Genero Studio 所提供的档案浏览器。在选取的档案上按鼠标右键时，会弹出功能选单。功能选单会根据附档名的不同而提供不同的功能。一般编辑 4fd 档时会由此处开启。从 per 档汇入到 4fd 档也是在此执行。



从 per 档汇入到 Form Designer

per 档案可透过 Genero Studio 汇入到 Form Designer (以右键选择『Import in Form Designer』, 注)。

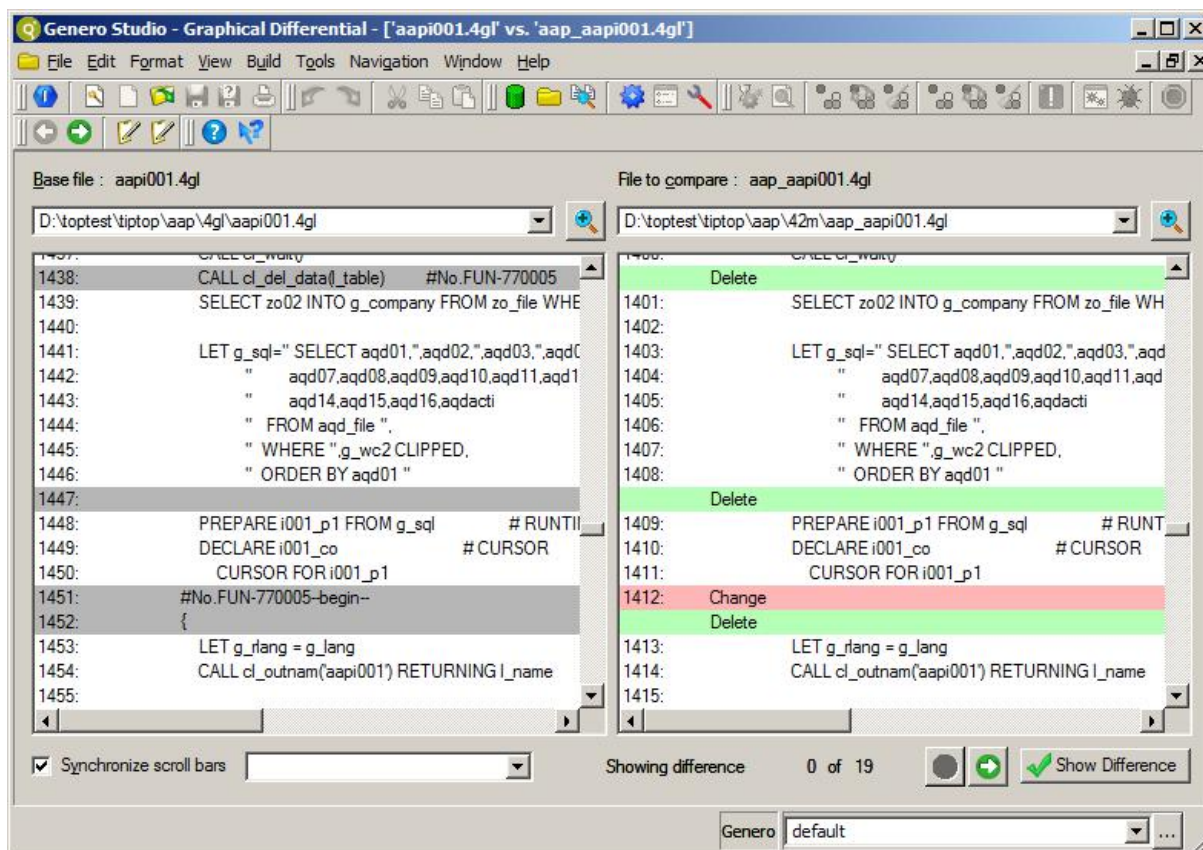
汇入后, per 档会备份为 ~per。4fd 档编译所产生 42f 档与 per 档编译所产生的 42f 档相同。所有 per 档的特性在 4fd 档中都会被保留。在 per 档中的注解会被存到 4fd 档中 Form 物件的 PER Comment 属性。



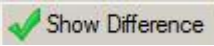
注: 若是以双击方式直接点开 per 档, 会以『Code Editor』进行编辑。

从 File Browser 中选出编辑的档案后，双击该档案，如果是 4gl 或 per 档等，Studio 就会开启 Code Editor 让使用者进行编修。

Graphical Differential

Graphical Differential 是一个可以让程序开发人员进行比对『修改前』与『修改后』程序码差异的工具。



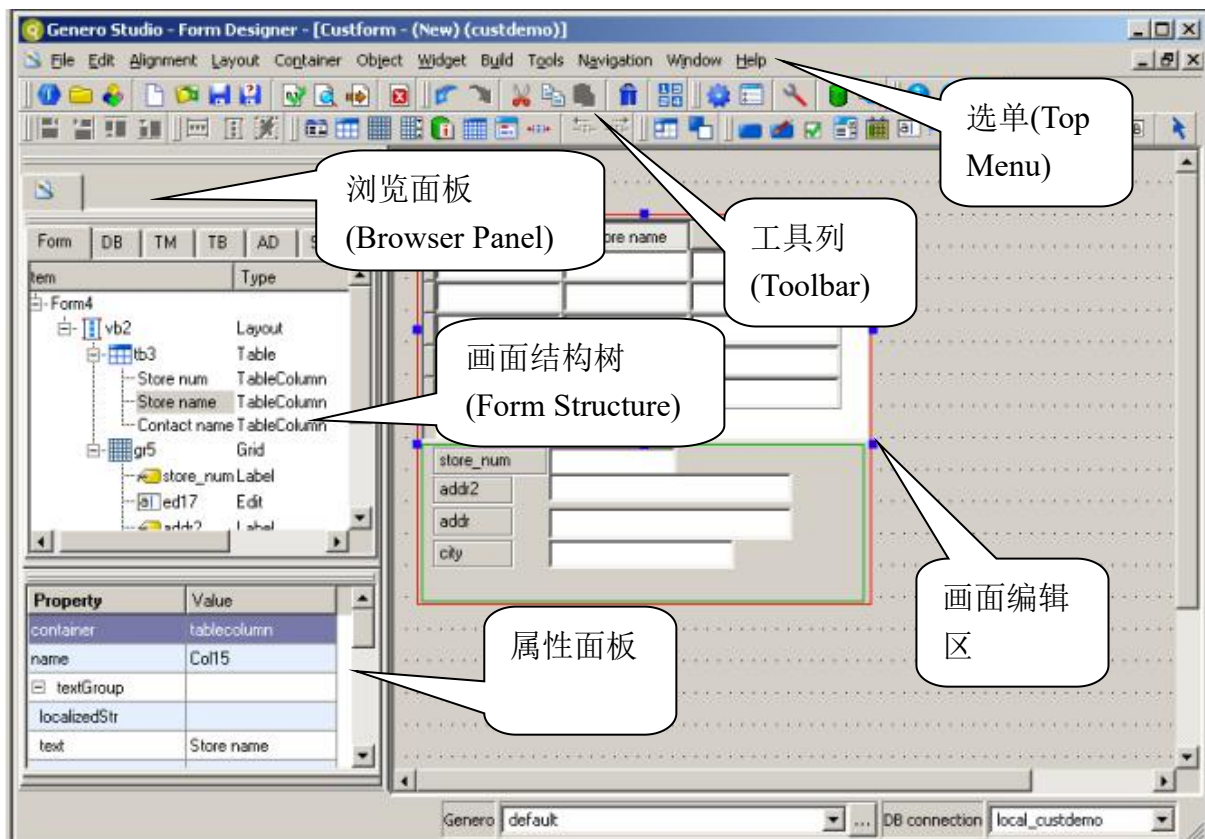
点选工具列上的『』钮后，即可开启此作业。执行时先将要比对的左右两侧档案选出（分别以左右的工具找出要比较的档案），再押键，系统即会比对两个档案内容，并将差异处以不同色块表现出来。

如上图所示，除每行开头附有行号之外，删除行以绿色表示、红色表示该行有字符上的不同。

Form Designer

Form Designer 是 Genero Studio 中用来设计画面档的工具。Form Designer 设计的画面与最后产生的画面相当接近。

Form Designer 所储存的附档名是『.4fd』，是以 XML 的格式储存画面元件在设计版面的各种信息。



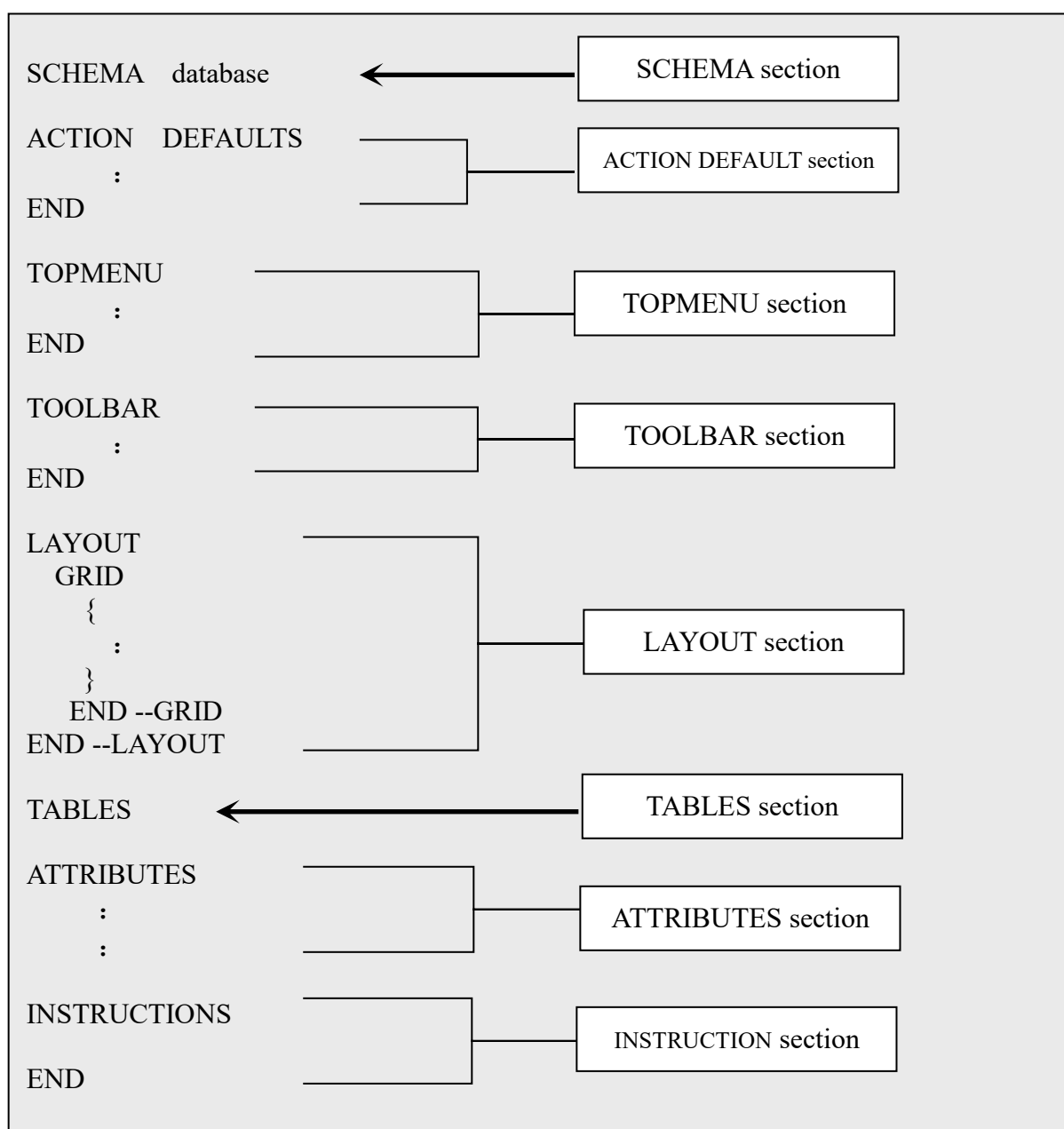
右边是表单设计区，左边则是浏览面板。浏览面板上有 Form、DB、TM（Top Menus）、TB（Toolbars）、AD（Action Defaults）、SR（Screen Records）、Style 七个页签。使用者可从下拉式选单，或是图示列选取元件（widget/object/container）放置于画面上，如上图所示。

画面档（FORM）

Genero BDL 预设开发画面原始档为 **「per」** 格式，此为纯文字档格式，可用文字编辑器直接编辑。缺点则是程序码可读性差。

Genero Studio 则因需提供较接近物件化的画面原始档编辑工具，因此新增 **「4fd」** 格式，4fd 档案采用 XML 格式纪录资料，因此必须用 Studio 工具（Form Designer），始得进行修改或编辑的工作。

PER FORM 主要由以下几个 SECTION 组成（有顺序性、大小写均可，建议保留字以大写字显示为佳），画面档的附档名必需为 **「per」**：



1. SCHEMA 【非必要 SECTION】

设定设计画面时，所需要引用到的数据库。

如果在『per』中没有指定 SCHEMA，系统预设 **FORMONLY**，即表示此不引用数据库中栏位型态、长度等资料。若未设定，则对后续 SECTION 影响有下列二点：

- 不得使用 TABLES SECTION。
- 需于 ATTRIBUTES SECTION 中指定栏位型态。

注：为了与 BDL 旧版本兼容，SCHEMA 指令也可使用 DATABASE 代替，两者意义与使用情形相同。

2. ACTION DEFAULTS 【非必要 SECTION】

定义 ACTION 显示于画面上的名称值及快速键设定。

3. TOPMENU 【非必要 SECTION】

定义画面中的 TOPMENU 结构。

4. TOOLBAR 【非必要 SECTION】

定义画面中的 TOOLBAR 结构。

5. LAYOUT 【必要 SECTION】

定义画面样式的主要 SECTION，在本 SECTION 中须采用物件的概念，任何的设定均应以 Container 视之，在结束处需加上 END 注记以标示结束。

6. TABLES 【非必要 SECTION】

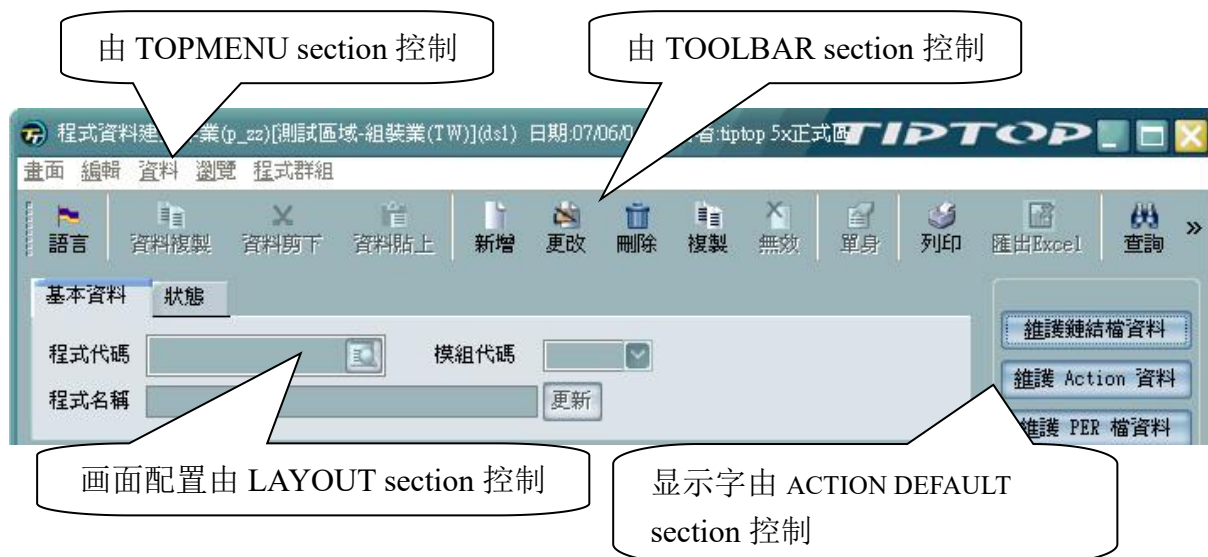
设定荧幕画面的显示栏位所对应的数据库的 TABLE 名称。

7. ATTRIBUTES 【非必要 SECTION】

在 LAYOUT SECTION 中出现的栏位、TAG 等，均需在此段中定义其形态或属性。

8. INSTRUCTIONS 【非必要 SECTION】

定义荧幕阵列。当使用荧幕阵列时才需定义。



各 section 控制区域概略说明图

4FD FORM 必须透过 Genero Studio 内含的 Form Designer 界面始得以进行编辑作业。整体的设计已经和 Microsoft©所提出的 Visual Studio™操作方式相当接近，以拖拉方式将所需要的元件（Widget）放到定位上，并填入所需要的属性（Attribute）。

但部分设计概念仍旧承袭原有的 PER 架构（如：Grid 画布、区块的位置排列等概念），因此在设计时也必须对 PER 架构有基本的了解。

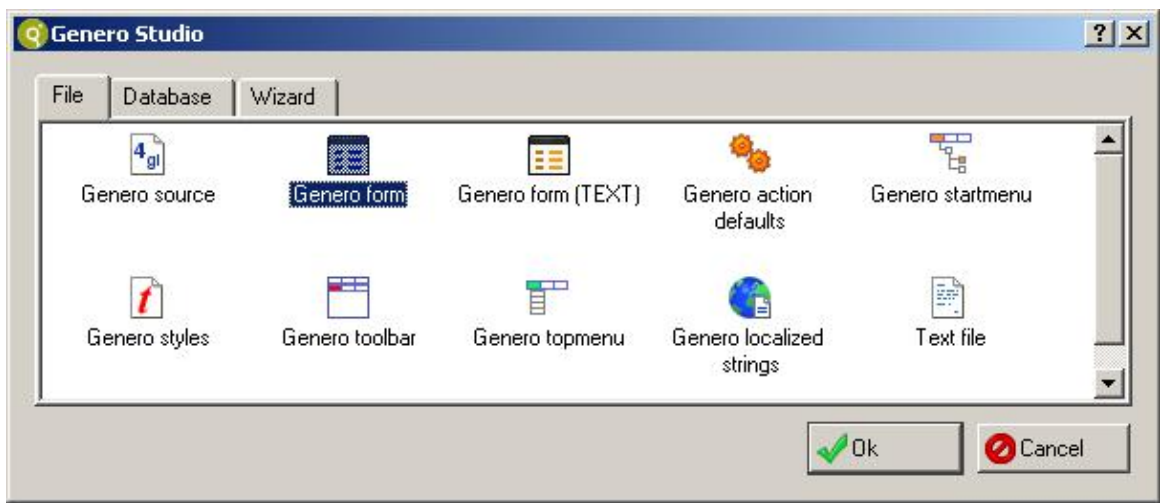
由于自 **TIPTOP GP 5.0** 版本起，均改为以 **4FD** 档为画面档使用格式，因此后续介绍画面档编写方式时，均以 **4FD** 为介绍目标；并以 **Genero Studio** 的 **Form Designer** 界面作为预设开发工具。

画面档设计

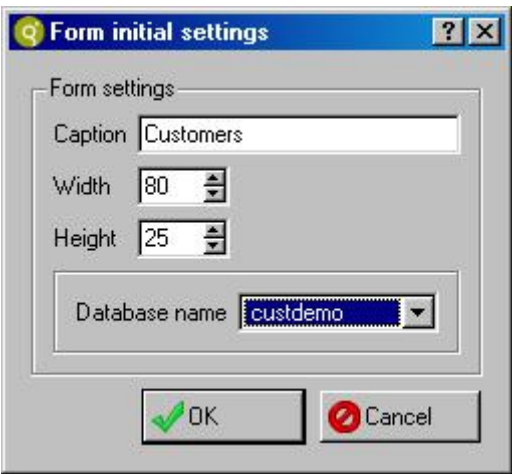
进入 Genero Studio 后，由选单选择『File』→『New...』，即会出现下列视窗，其中在画面档部分有两个类型：Genero form 与 Genero form (TEXT)。

- Genero form：Studio 预设画面档，即为 4FD 档，以 Form Designer 编辑。
- Genero form (TEXT)：为 fgl 预设画面档，纯文字模式，即为 PER 档，以 Code Editor 进行编辑。

此处选择『Genero form』格式进行作业。



按下『Ok』后会显示下面的对话视窗：



Form 物件的属性：

| | |
|--------------|--|
| Caption | 视窗标题列显示文字 |
| Width | 视窗宽度（每行字符数），最大值 300 . |
| Height | 视窗高度（行数），最大值 200 |
| Databasename | 参考的数据库名称，等于 per 档中的 SCHEMA 指令。此名称为在 Database Browser 中所设定的 Schema 名称 |











widget/object/container 图示选取规则：

- 在图示（icon）上单击鼠标左键：单次放置元件到 Form 上。
- 在图示（icon）上双击鼠标左键：可连续放置多个目前指定的元件到 Form 上，直到选取其它元件的小图示或是按 DeselectArrow（蓝色箭头小图示）。

Container（容器物件）

容器物件均具有本身的属性及特殊用途，物件间可相互包覆（基本物件除外），以呈现不同的效果。

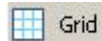
以下是列出在 Genero Studio 中可以使用的物件，及简易说明列表。

| Container | Widget | Build | Tools | Navigation |
|---|---------------------|-------|--------------|------------|
|  | MFArry | | Ctrl+Alt+A | |
|  | PageControl | | Ctrl+Alt+P | |
|  | Table | | Ctrl+Alt+T | |
|  | Grid | | Ctrl+Alt+D | |
|  | ScrollGrid | | Ctrl+Alt+G | |
|  | GroupBox | | Ctrl+Alt+U | |
|  | HRec | | Ctrl+Alt+I | |
|  | DataControl... | | Ctrl+Alt+J | |
|  | Add Spacer to Left | | Ctrl+Shift+L | |
|  | Add Spacer to Right | | Ctrl+Shift+R | |

物件（Container）摘要如下：

| 名称 | 功能说明 | 可使用的下层容器物件 |
|-----------------------------|---------------|--|
| Grid | 简易空白画布 | ScrollGrid、Table、GroupBox |
| ScrollGrid | 有卷轴的空白画布 | ScrollGrid、Table、GroupBox |
| Table | 以表格方式显示阵列资料 | 无 |
| MFArry | 以画布方式显示阵列资料 | 无 |
| GroupBox | 将外层加上框线 | VBOX、HBOX、GroupBox、PageControl、Grid、ScrollGrid、Table |
| PageControl | 以分页方式显示资料 | VBOX、HBOX、GroupBox、PageControl、Grid、ScrollGrid、Table |
| VBOX (Vertical layout) | 将内含的物件以垂直方式排列 | VBOX、HBOX、GroupBox、PageControl、Grid、ScrollGrid、Table |
| HBOX (Horizontal layout) | 将内含的物件以水平方式排列 | VBOX、HBOX、GroupBox、PageControl、Grid、ScrollGrid、Table |

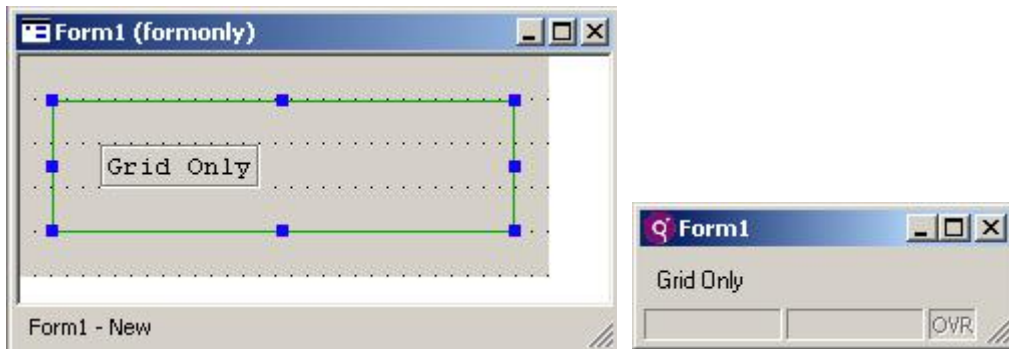
Grid



Grid

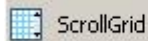
可将此容器视为一块空白的画布，布置在上方的元件都可以显示在画面相对位置上。Grid 只能处理『非阵列资料』。Grid 不可以被安排在其他 Grid 容器之内。

拉动此容器时，画面上以绿色单线框呈现。



图左：编译画面 图右：执行预览画面

ScrollGrid

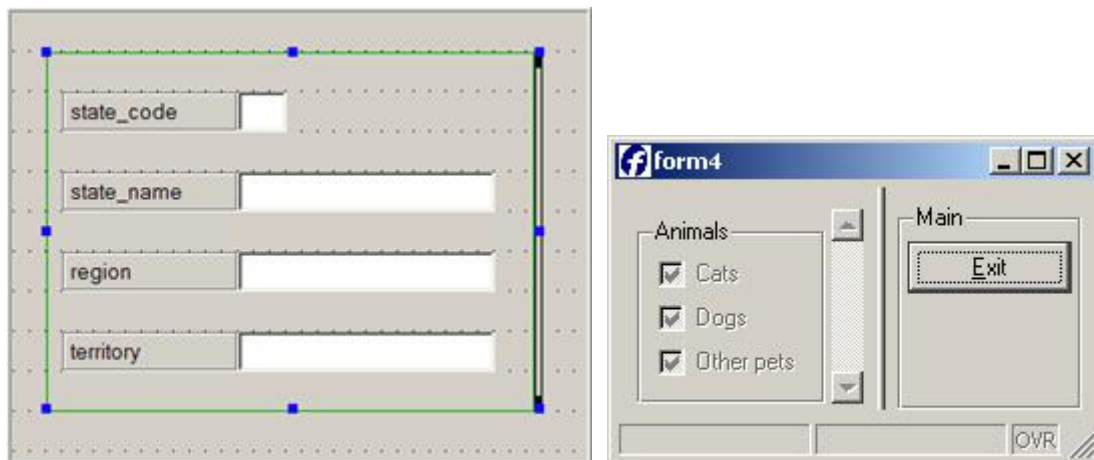


ScrollGrid

在 Genero Studio 中，此容器与 Grid 相同，均作为处理『非阵列资料』用。与 Grid 容器的差异仅在可使用卷轴，可以卷动画面。不能用于显示阵列资料。

〔注：在 PER 档格式中，SCROLLGRID 容器是处理阵列资料用，该类容器与本节说明 ScrollGrid 完全不同，PER 档的 SCROLLGRID 性质应等同于 4FD 格式中的 MFArray。〕

拉动此容器时，画面上以绿色单线但右侧带有卷轴的方框呈现。



图左：编译画面 图右：执行预览画面

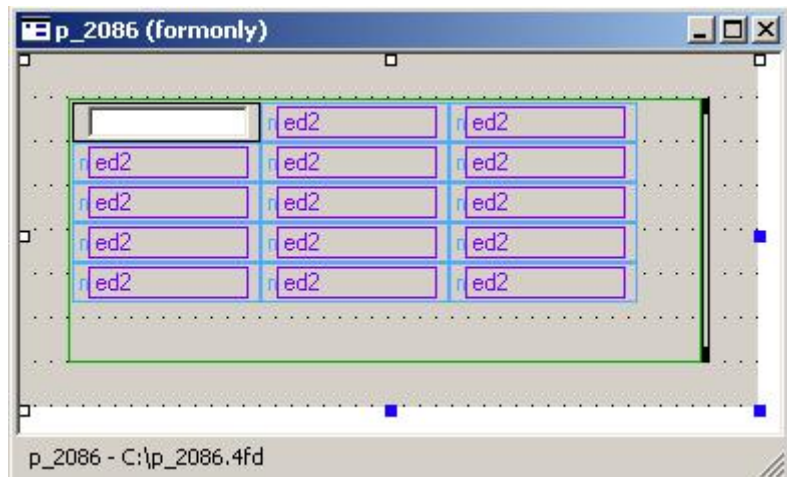
Multi Field Array (MFArray)



4FD 格式中的 MFArray 功能与 PER 格式中的 SCROLL-GRID 容器相同；都是用来显示（或存放）阵列资料。

MFArray 简化了设计。只要在第一个栏位（黑色外框）放置元件；之后再设定 Row Count 与 Column Count 属性，就会在设计画面上出现与浅蓝外框的栏位，这些栏位可以视作与第一个栏位相同的复本。

调动第一个栏位时，其他栏位会跟着第一个栏位的内容自动调整。因此使用者不用自行做复制调整，在设计上简便许多。



图上：编译画面 图下：执行画面

若栏位在 Row 与 Column 方向均有设定重复次数时，系统显示资料时先以同 Row 为主。（参照右上范例图）

拉动此容器时，画面上以绿色单线但右侧带有卷轴的方框呈现，最外层框与 ScrollGrid 相同，但内部会有一至数个黑色或浅蓝色小框。

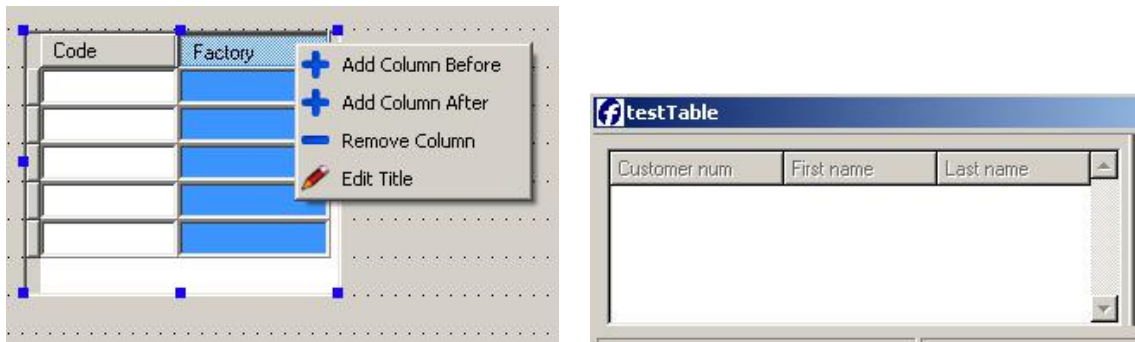
编制时须注意：

- 编制完需到左方浏览面板的 SR 页签去设定 Screen Record。
- 阵列资料除非程序规格需要，请尽量勿使用 MFArray，而尽量改用 TABLE 方式呈现。
- 使用时若需加上栏位说明文字，在特别情况下应采用『Static Label』显示说明。

Table



使用 TABLE 即是以表格方式显示阵列资料，此方式有许多的优点，这些优点都是系统提供的，不需要额外再撰写程序码即可使用；包含：动态排序、栏位隐藏、显示或移动等。在设计时期改变 Table 高度时，会自动增减资料的行数。在 Table 物件上按鼠标右键，在弹出式选单可以新增或删除栏位。另外可以直接以鼠标拖曳改变栏位的顺序。

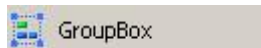


图左：编译画面 图右：执行预览画面

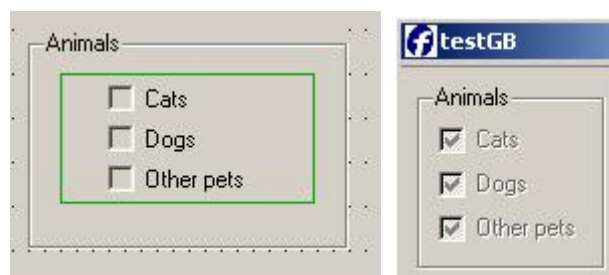
编制时须注意：

- 使用 TABLE 物件时，资料（Record）一定是横列，没有直垂直排列。
- 编制时须到各栏位的属性中进行形态、对应数据库等资料的设定或变更。

GroupBox



相关栏位可用 GroupBox 包覆，以让相近性质的栏位可以明确的群聚于同处。

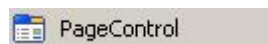


图左：编译画面 图右：执行预览画面

编制时须注意：

- 若不需要 Group 名称，则可以不要设定『textGroup』→『Text』属性。
- 若要分组但不要框线，请改用 Grid。

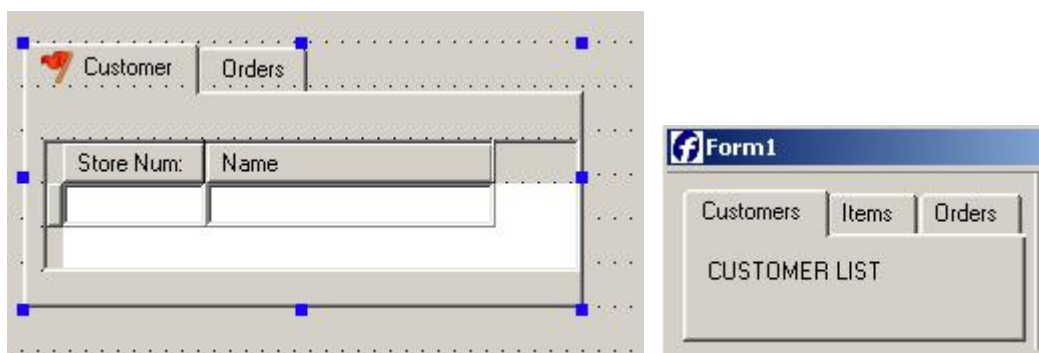
PageControl



当摆放元件空间不足时（或是需要卷动画面，操作上较麻烦时），即可使用切页的功能，以资料夹的形式将资料性质相近的栏位，切分在同样的 page 当中。

制作时可在页签位置以鼠标右键新增、删除页面。编译时若该页签内没有任何元件，则编译会失败，且系统会显示有空白页签存在。

页签上显示字符串仍需在属性视窗指定。

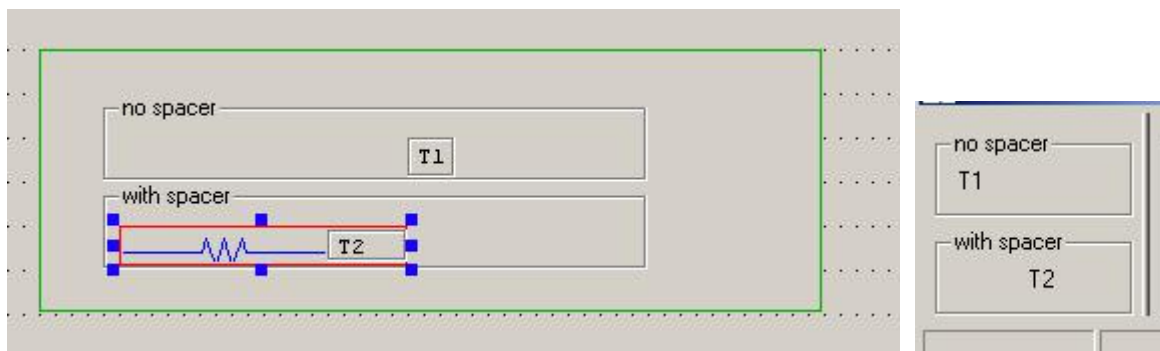


图左：编译画面 图右：执行预览画面

HRec



HRec 用 spacer 来保留画面一定的空格符数。当元件在 GroupBox 中时，若不使用 HRec，则在预览时会发现想要保留的空格符，实际上并没有被保留。点选 HRec 里面的元件或是 spacer 时，可以在元件的左右加入 spacer（**Spacer left** and **Spacer right**）。



图左：编译画面 图右：执行预览画面

编制时须注意：

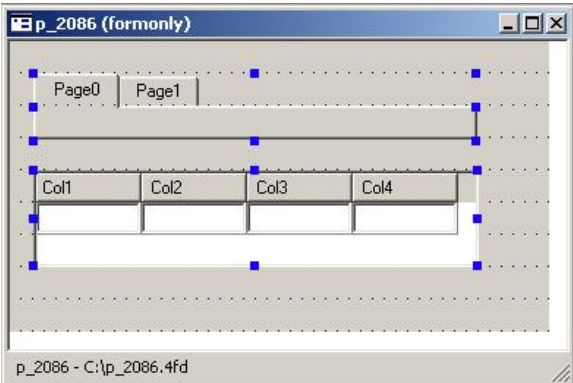
- HRec 只容许放置单行元件。
- 空白的 Label 也可以达到相同的功能，请尽量以空白 Label 取代此物件。

版面排列物件

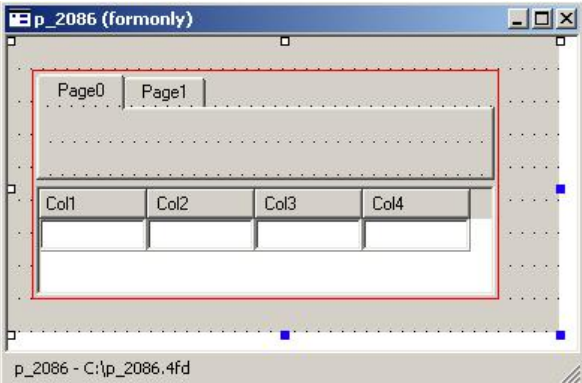


由于 Genero Studio 使用的 4FD 档案格式是承袭 Genero fgl 的 PER 档格式而来，因此有部分观念也需要延用到 4FD 档案中。

例如：当有多个容器（Container）同时并存在同一层画面上，设计画面时就必须再对这多个容器进行位置排列的指定（对应到 PER 档格式中即为 HBOX 及 VBOX 两种容器）。



图左：未加上排列



图右：已加上排列

在设计视窗中，以多选物件的方式，选取两个以上的物件后，工具列上的『Layout 排列工具项』（Create Layout）即会被 enable，直接选取工具列中的『垂直排列』功能（Vertical）或『水平排列』功能（Horizontal），出现红色外框时，即完成排列方式指定。



若设定完成后发现设定错误，必须取消原先设定后才能在设定新的排列方式。此时點選该红色外框，上方『取消排列』（Break Layout）即可进行取消作业。

功能说明：

| | |
|--|----------------------------|
| | 『垂直排列』功能 Vertical Layout |
| | 『水平排列』功能 Horizontal Layout |
| | 『取消排列』功能 Break Layout |

Widgets（元件）

4FD 档在编辑时，就需要视需求将指定的元件布置在画面上。Widget 项目如右图所示，每个项目均可以鼠标拖拉至画面上的指定位置做处理。

但若仅需要进行元件项目代换时，也可以透过属性页处提供的功能，直接进行转换。

以下分项目说明各种元件的差异。

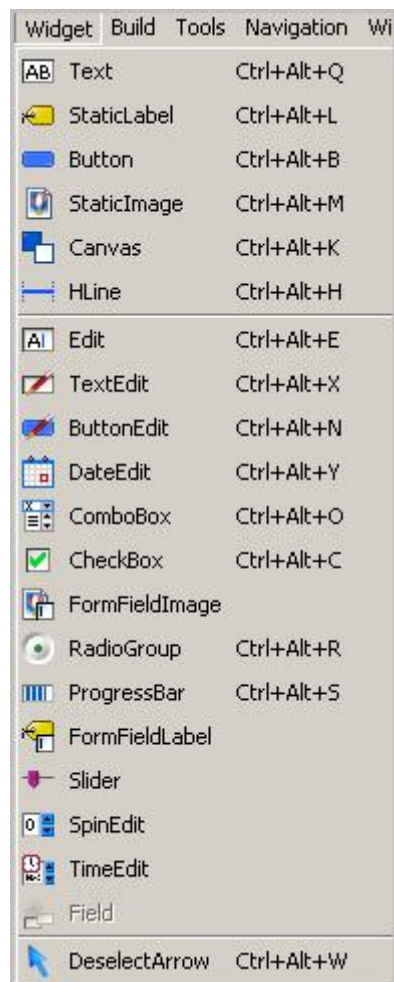
Edit



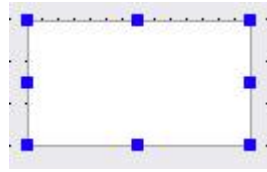
定义一个编辑栏位。属于 FormField 物件，可设定与资料栏位的关联。

常用属性：

- commentGroup → comment: 设定说明内容，当鼠标移过时会以符动视窗方式显示内容。
- constraints → noEntry: 设定此属性后即无法进入此栏位编辑资料。
- constraints → notNull: 不可在此栏位输入 NULL 值或空字符串。
- constraints → required: 不可在此栏位输入 NULL 值、空字符串或纯空格符。
- appearance → case: 令输入字符串自动转换大/小写。
- appearance → scroll: 卷动，若数据库栏宽大于画面预留栏宽时有效，当设定此属性时可藉卷动方式输入。



TextEdit



定义可编辑多行的栏位，输入长度当超过画面预留长度时，会自动出现卷轴。属于 FormField 物件，可设定与资料栏位的关联。

常用属性：

- `commentGroup` → `comment`：设定说明内容，当鼠标移过时会以符动视窗方式显示内容。
- `constraints` → `noEntry`：设定此属性后即无法进入此栏位编辑资料。
- `constraints` → `notNull`：不可在此栏位输入 NULL 值或空字符串。
- `constraints` → `required`：不可在此栏位输入 NULL 值、空字符串或纯空格符。
- `appearance` → `case`：令输入字符串自动转换大/小写。
- `scrollBars`：可设定卷轴出现的位置，有 NONE、VERTICAL、HORIZONTAL、BOTH 等。
- `wantTabs`：当设定此属性时，允许在输入框中输入 TAB 键。
- `WantNoReturns`：当设定此属性时，不允许在输入框中输入 ENTER 键。

ButtonEdit

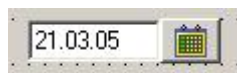


定义一个编辑栏位的元件，可透过右侧按钮以触发某一事件。通常用在串连与此栏位输入时有关的动作，例如查询合法可用资料等。此元件为 FormField 物件，可设定与数据库中资料表的栏位相关联，将 `data` 属性设为 `TABLE_COLUMN` 时可额外设定 `tableName` 及 `columnName` 这两个属性。

常用属性：

- `action`：设定当按下按钮时要触发 4GL 中何组 ON ACTION 段、此处定义其 `action-id`。
- `commentGroup` → `comment`：设定说明内容，当鼠标移过时会以符动视窗方式显示内容。
- `Image Group` → `image`：设定出现在 BUTTON 上的图片，其来源可参照 ACTION DEFAULT 段说明。
- `constraints` → `noEntry`：设定此属性后即无法进入此栏位编辑资料。
- `constraints` → `notNull`：不可在此栏位输入 NULL 值或空字符串。
- `constraints` → `required`：不可在此栏位输入 NULL 值、空字符串或纯空格符。
- `appearance` → `case`：令输入字符串自动转换大/小写。
- `appearance` → `scroll`：卷动，若数据库栏宽大于画面预留栏宽时有效，当设定此属性时可藉卷动方式输入。

DateEdit

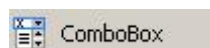


定义一个日期编辑，按右侧钮可带出 Client 端万年历选择视窗。日期显示格式由主机端 DBDATE 环境变量控制。属于 FormField 物件，可设定与资料栏位的关联。

常用属性：

- commentGroup → comment: 设定说明内容，当鼠标移过时会以符动视窗方式显示内容。
- constraints → noEntry: 设定此属性后即无法进入此栏位编辑资料。
- constraints → required: 不可在此栏位输入 NULL 值、空字符串或纯空格符。

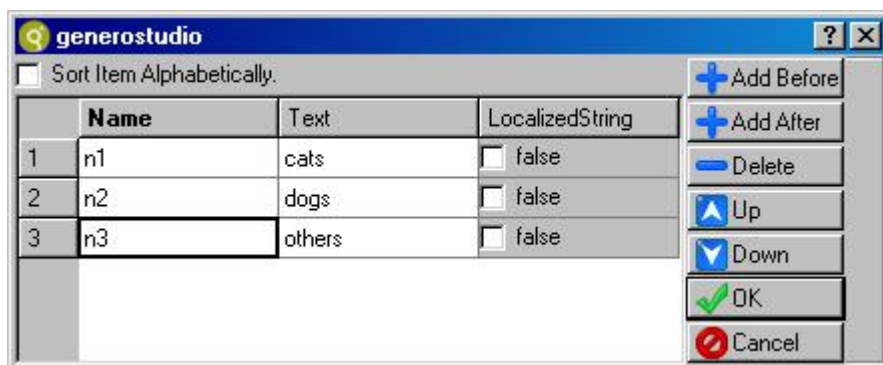
ComboBox



定义一个可利用下拉功能选值的编辑栏位，若输入资料只有几种值可供选择时，建议采用 RadioGroup 方式来限缩使用者可输入的内容（参阅 RadioGroup）。属于 FormField 物件，可设定与资料栏位的关联。

选项对话视窗

可管理 ComboBox 的选项，也可以按字母顺序排列选项的 Text。



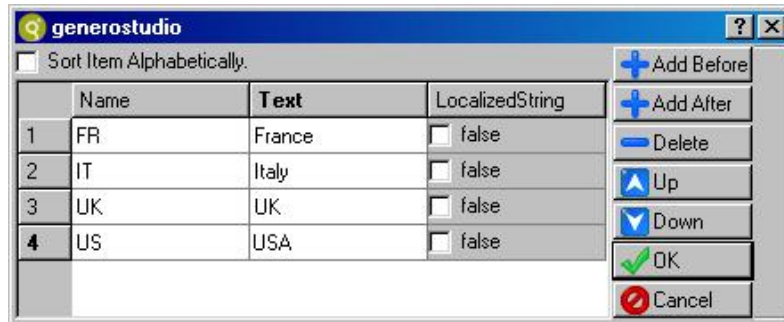
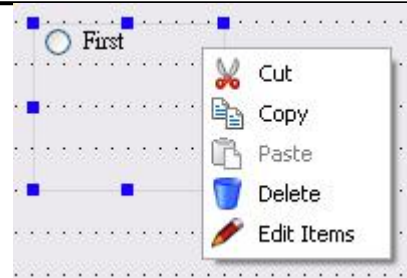
常用属性：

- commentGroup → comment: 设定说明内容，当鼠标移过时会以符动视窗方式显示内容。
- items: 定义可选择的选项，用对话视窗设定。
- constraints → noEntry: 设定此属性后即无法进入此栏位编辑资料。
- constraints → notNull: 不可在此栏位输入 NULL 值或空字符串，可抑制 NULL 选项出现。
- constraints → required: 不可在此栏位输入 NULL 值、空字符串或纯空格符。
- queryEditable: 当设定此属性后，若为 CONSTRUCT 模式下即可开放 USER 自行输入。
- case → 令输入字符串自动转换大/小写。

RadioGroup



定义一个可用选择方式输入资料的输入栏位，此种选择方式会将选项清单展示在画面上（ComboBox 不会展开显示，可参照 ComboBox 说明），故若需要采用此输入形态，要注意画面空间是否足够。



常用属性：

- commentGroup → comment: 设定说明内容，当鼠标移过时会以符动视窗方式显示内容。
- items: 定义可选择的选项，依范例方式设定。【必要属性】
- constraints → noEntry: 设定此属性后即无法进入此栏位编辑资料。
- constraints → notNull: 不可在此栏位输入 NULL 值或空字符串。
- constraints → required: 不可在此栏位输入 NULL 值、空字符串或纯空格符。
- orientation: 可定义选项排列方式为垂直排列（vertical）或水平排列（horizontal）。

注：使用 ComboBox 可动态定义 items，而 RadioGroup 不可动态定义 items。

CheckBox

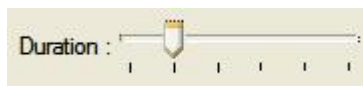


当栏位选项只有两种选择时（如：是或否，0 或 1，100 或 200 等），就可以采用此种输入型态执行输入。属于 FormField 物件，可以设定与资料栏位相关联。

常用属性：

- commentGroup → comment: 设定说明内容，当鼠标移过时会以符动视窗方式显示内容。
- constraints → noEntry: 设定此属性后即无法进入此栏位编辑资料。
- constraints → notNull: 不可在此栏位输入 NULL 值或空字符串，可抑制 NULL 选项出现。
- constraints → required: 不可在此栏位输入 NULL 值、空字符串或纯空格符。
- textGroup → text: 显示在选取格后面的说明字符串。
- valueChecked: 选取核取后传入 4GL 的值，可为数字、字符、字符串。【必要属性】
- valueUnchecked: 选取取消后传入 4GL 的值，可为数字、字符、字符串。【必要属性】

Slider

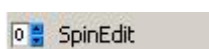


定义一个水平或垂直刻度的拖动条栏位。属于 FormField 物件，可设定与资料栏位的关联。显示的位置必须在有效范围内（在下限与上限之中）。

常用属性：

- **valueMin**: 定义控制数值的最小值。
- **valueMax**: 定义控制数值的最大值。
- **step**: 刻度的间距。

SpinEdit



定义一个旋转编辑栏位。属于 FormField 物件，可设定与资料栏位的关联。上下图示的按钮可以增加/减少数值，也可以直接在栏位中输入数值。

常用属性：

- **step**: 增加/减少数值的间距。

TimeEdit



定义一个时间编辑栏位。属于 FormField 物件，可设定与资料栏位的关联。

常用属性：

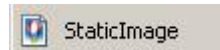
- **commentGroup** → **comment**: 设定说明内容，当鼠标移过时会以浮动视窗方式显示内容。
- **constraints** → **noEntry**: 设定此属性后即无法进入此栏位编辑资料。
- **constraints** → **required**: 不可在此栏位输入 NULL 值、空字符串或纯空格符。

Images

定义一个可显示图档的区域。

Image 分为下列两种 Widget:

1. **Static Image:** 显示静态图档。



2. **FormField Image:** 显示动态图档。



常用属性:

- **autoScale:** 自动依画面上现行留存框格为准, 调整(等比例缩放)显示在画面上的图片大小。
- **geometry → Width 或 Height:** 不以画框的方式指定大小, 而直接以画素指定的方式定宽或高。
- **stretch:** 定义是否要出现卷轴, 有 none、x、y、both 四种可选择。

说明:

- IMAGE 有区分动态或静态模式, 以上范例为动态模式, 影像图片在 CLEAR FORM 时会一并清除, 如同一般栏位设定。若图片属于 logo 类与资料无关、不会常常切换的话, 建议可改用静态图片的作法。
- 静态的以『Image Group -> image』属性以指定图片档来源。静态图片不会因 CLEAR FORM 被清除。

ProgressBar



定义一水平方向的进度显示表。

常用属性:

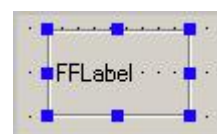
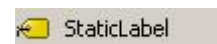
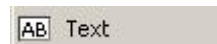
- **valueMin:** 定义控制数值的最小值。
- **valueMax:** 定义控制数值的最大值。

注: TIPTOP GP 中有已定义完整的 Progress Bar 处理函式, 可直接呼叫使用, 使用者不需自行定义 Progress Bar。

Labels

定义一显示值用的栏位，可区分为下列三类。

1. **Text:** 由字符数自动决定大小的基本静态文字标签。无法自行设定宽度。
2. **StaticLabel:** 静态文字标签，可设定宽度、前景色等。
3. **FormFieldLabel:** 动态标签元件，属于 FormField 物件，可设定与资料栏位的关联。



常用属性：

- textGroup → text: 显示说明字符串。

附注说明：

- TIPTOP GP 可支援动态画面多语言转换的功能，因此定有转换规则，符合下列规则的画面元件始能进行语系转换：

1. 输入栏位左侧说明依同名元件方式转换
2. TABLE 每栏栏位标题可转换
3. 各别元件有元件 ID 足供识别可转换【含 page、groupbox、static label 等】

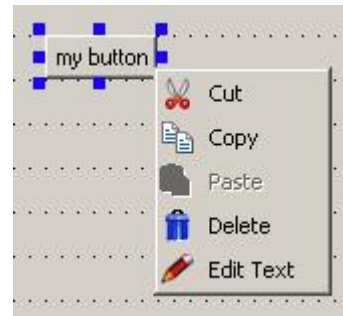
其余不符合上列规则者均无法于 TIPTOP GP 系统中完成多语言元件转换。

- TIPTOP GP 在设计 static label 时，要求其栏位名称必需以『dummy』为启始，以资识别为 static label。

Button



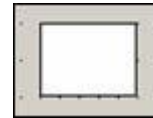
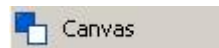
定义一个按钮以触发某一 4GL 中已写定的 ON ACTION 段。建议除了特别的 ACTION 有需要在 Layout 内布置按键外，一般作业可不必布置，令其出现在 Ring Menu 处即可。



常用属性：

- commentGroup → comment: 设定说明内容，当鼠标移过时会以符动视窗方式显示内容。
- Image Group → image: 设定出现在 BUTTON 上的图片，其来源可参照 ACTION DEFAULT 段说明。
- textGroup → text: 显示在 BUTTON 上面的说明字符串。

Canvas



定义一个绘图的区块以供 4GL 程序绘制图形用。

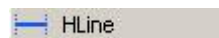
常用属性：

- commentGroup → comment: 设定说明内容，当鼠标移过时会以符动视窗方式显示内容。

说明：

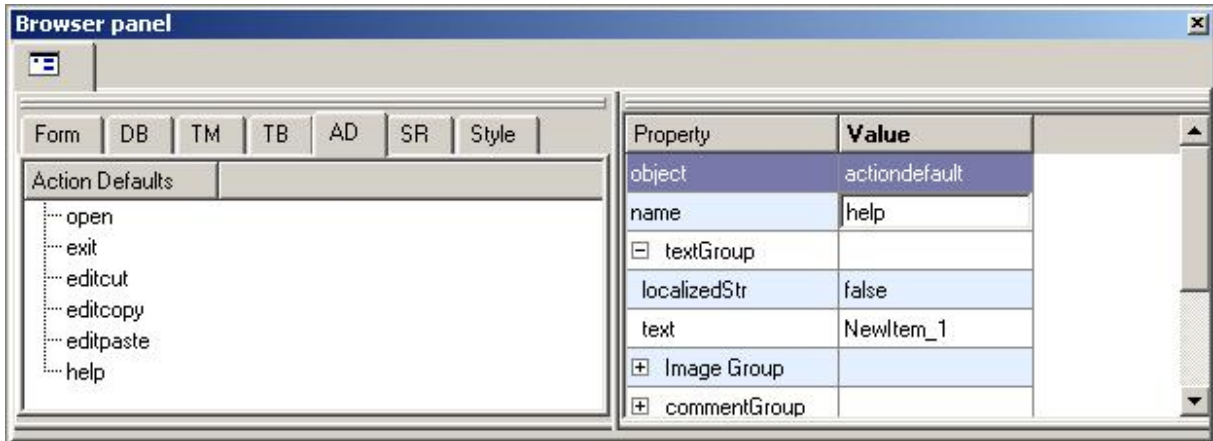
Genero BDL 可支援简易的绘图模式，但需藉由建立物件 `om.DomNode` 的方式来建立，此部份可参阅 Genero BDL 在线说明文件。

HLine



水平分隔线

其他项目

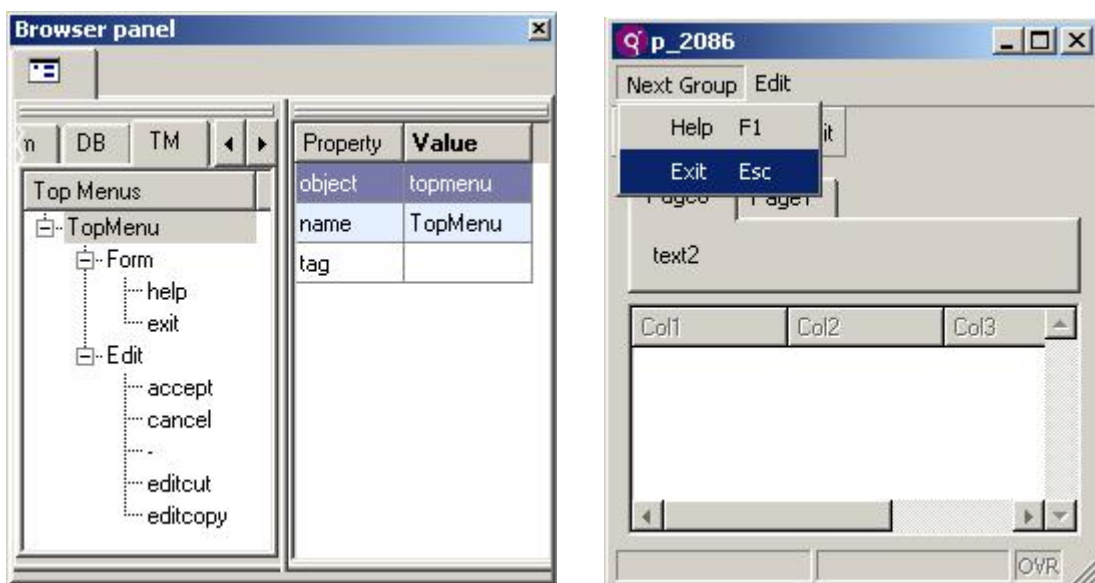


在 Studio 左侧属性页签上方，有一区『Form Browser』功能（如上图所示），此处各页页签均对应原有 PER 档个别设定区段，以下就各区段分别说明。

TM (TopMenus) (参考用)

TM 页签对应到 fgl 中 PER 档的 TOMMENU Section。主要功能在将 Action 依功能别群聚为一组一组，设定显示在画面上方 Menu Bar，让使用者利用 Pull-Down Menu 点选即可执行 Action。

范例：



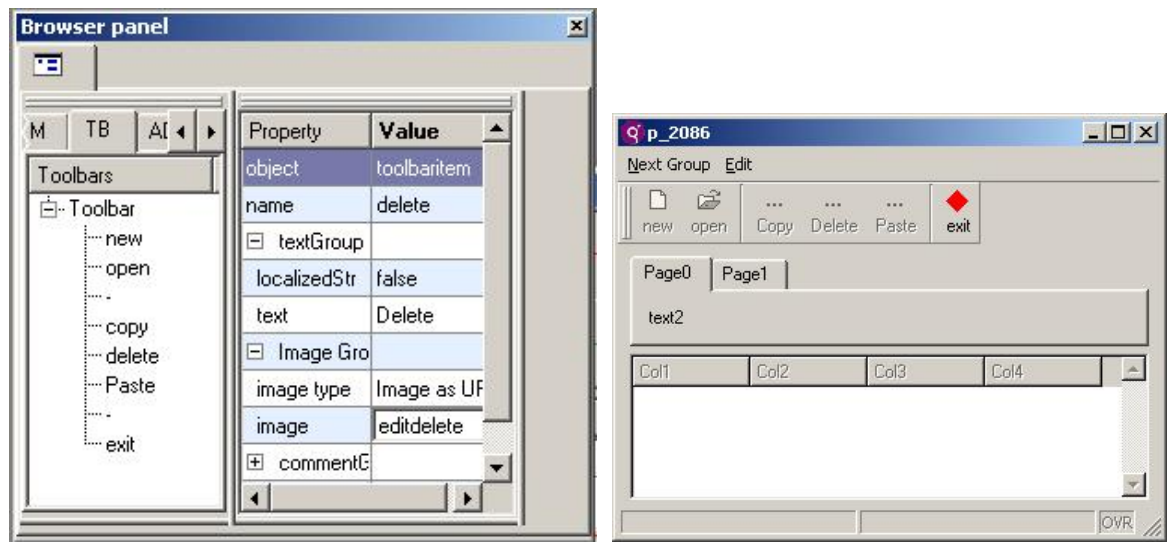
图左：编译画面 图右：执行预览画面

注：在 TIPTOP GP 系统中亦有使用 TM 功能，但已依 TIPTOP GP 特性进行设定，并未将资料定义于 4FD 档内，而是将资料另外储存于『\$STOPCONFIG』下，可参阅『TIPTOP GP 技术手册』。

TB (Toolbars) (参考用)

TB 段对应至原 PER 格式的 TOOLBAR Section，主要功能在定义将常用的 Action 显示在画面上方 TOOLBAR 处。

范例：



图左：编译画面 图右：执行预览画面

注：TOPTOP GP 将 TB 资料另外储存于『\$STOPCONFIG』下，可参阅『TIPTOP GP 技术手册』。

注：TIPTOP GP 中不使用 ACTION DEFAULT 与 TOPMENU SECTION，因为此二段设定会使『TIPTOP GP 的特性（画面显示语言别可动态更新）遭到破坏』。因此并未于画面档（PER）中设定。
TIPTOP GP 将此二段独立至『\$STOPCONFIG』目录下进行设定，详情请参照『TIPTOP GP 技术手册』。

AD (Action Defaults) (参考用)

AD 页签对应到原 PER 档的 ACTION DEFAULTS 区段，主要功能是定义功能按键 (Action) 的属性，包含显示在画面档上的文字、快速键 (1.31 以上版本允许设定至多三组)、显示图片 (注)、弹出式说明 (Hint、Comments) 等属性，须搭配 4GL 程序才会有作用。

注：图片抓取请参照『TOOLBAR Section』说明。

〔.4ad〕档

在某些特殊原因下【注】，可照规则编写一 XML 格式档案 (例如下面范例：act.4ad)。而在 4GL 程序中以呼叫 (CALL) `ui.Interface.loadActionDefaults("4ad-filename")`，传入参数是 4ad 档档名，附档名可不写；产生的结果也与在 PER 档编辑 ACTION DEFAULTS 段或在 4FD 编写 TM 功效一致。

范例：act.4ad

```
<ActionDefaultList>
  <ActionDefault name="add" text="Append" acceleratorName="CONTROL-V" />
  <ActionDefault name="del" text="Delete" acceleratorName="SHIFT-F2"/>
  <ActionDefault name="zoom" text="Zoom" comment="Open zoom window" />
  <ActionDefault name="quit" text="Quit" acceleratorName="SHIFT-F5" image="quit" />
</ActionDefaultList>
```

4GL 引入范例：act.4gl

```
MAIN
  DEFINE    ls_flow_pic          STRING

  CALL ui.Interface.loadActionDefaults("act")

  OPEN WINDOW act_w WITH FORM "/u1/topprod/tiptop/demo/edit"

  MENU ""
    ON ACTION add
      CALL act_a()
    ON ACTION del
      :
    ON ACTION zoom
      :
    ON ACTION quit
      EXIT Program
  END MENU
  :
```

编译之后可以得到的画面与前一页相同。

注：如 TIPTOP GP 系统即未采用在 PER 档编写 ACTION DEFAULT 段的方法，原因是若编写在 PER 档中，则于遭遇不同语系时，就需准备多组不同的 PER 档案，如此则会增加程序开发的困难。故采用 4ad 分离的方式以降低开发系统中会发生的错误情形。可参阅『TIPTOP GP 技术手册』。

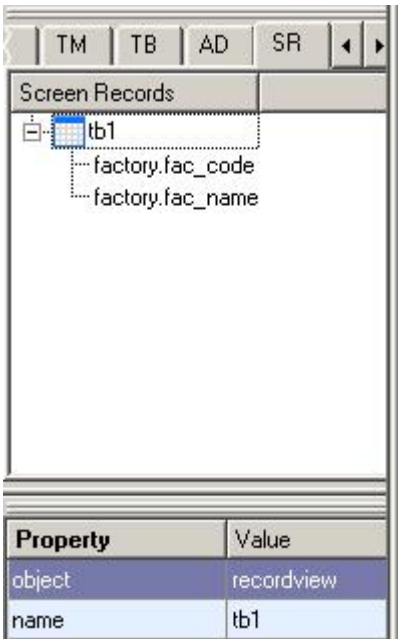
SR (Instructions; Screen Record)

SR 所建立的栏位相当于在 PER 档中的 INSTRUCTION Section 内的 Screen Record (荧幕变量组)。

INSTRUCTIONS

```
SCREEN RECORD tb1[10] (factory.fac_code,  
factory.fac_name)  
END INSTRUCTION
```

一个荧幕变量集合 (Screen Record) 可以包含来自不同资料表格的资料行，甚至是 formonly 的栏位，若该荧幕记录恰完全属于同一个表格 (table)，则可以使用『table_id.*』来定义。



每个 Table 元件都会自动建立 Default Screen Record，所有的变更都会自动反应到 Default Screen Record。(注)

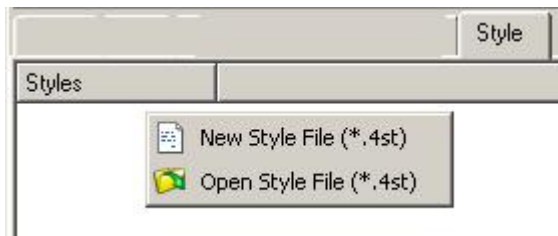
可以对 Table 建立不同栏位顺序的额外的 ScreenRecord。

但若使用 MFArray 容器时必须特别注意，需要手动自行建立 Screen record。

- 注：
- 使用 Table 时虽然会自行建立 SR 变量，但是却会以 tbXX (即 table 的名称) 直接命名，和 TIPTOP GP 对荧幕变量组命名规则不合，需要自行手动至属性页签处进行变更。命名规则部分可参阅『TIPTOP GP 技术手册』。
 - 栏位不可被重复定义，重复定义在编译时会产生编译错误。

Style (Styles) (参考用)

在浏览面板的 style 页签中，可以管理及编辑样式。

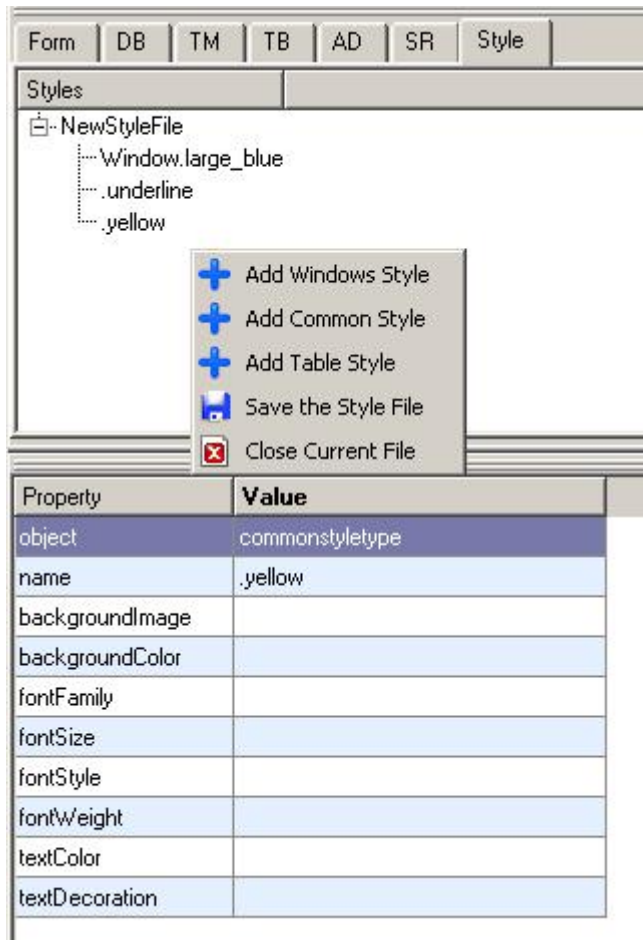


- Window Style: Window 可以使用的样式
- Table Style: Table 可以使用的样式
- Common Style: 所有元件都可以使用的样式。

使用样式

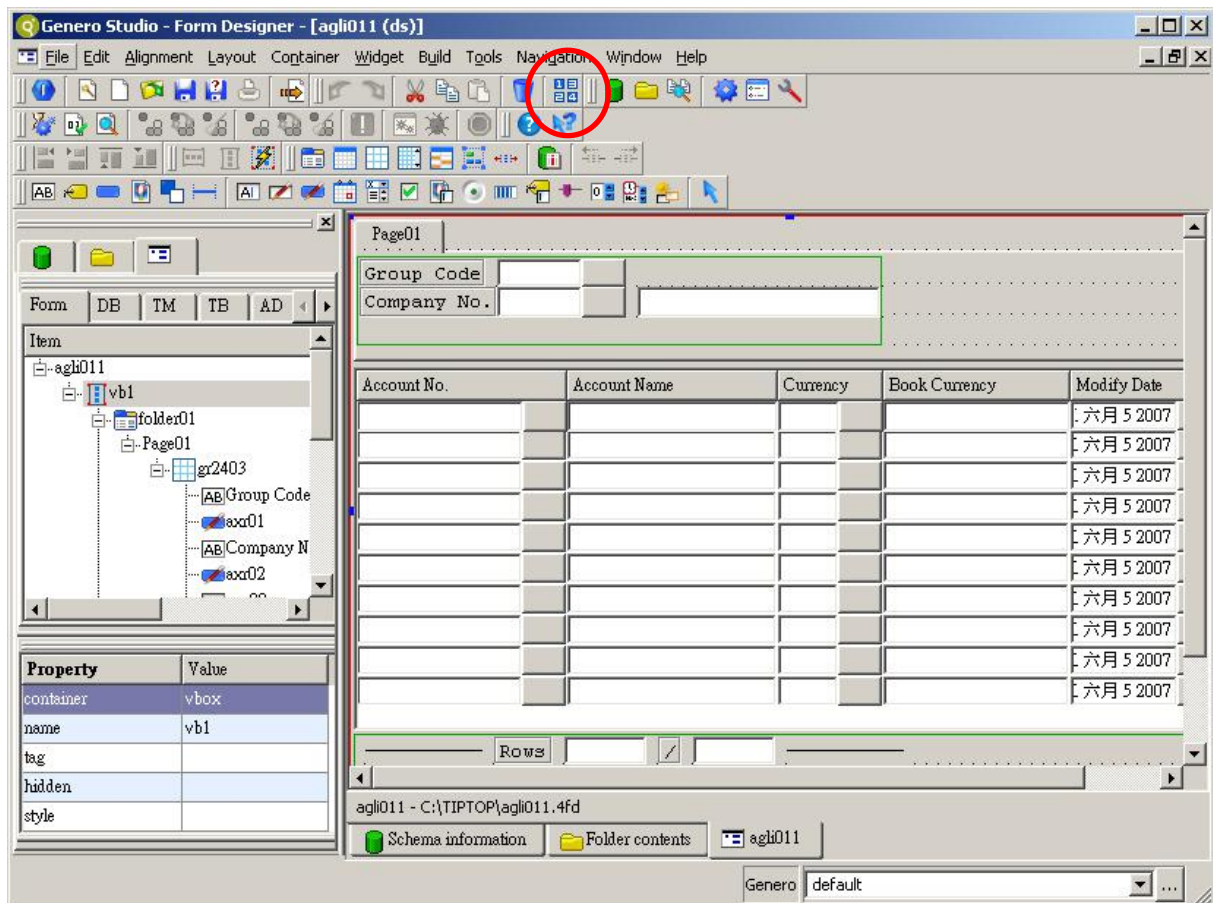
在 Form 的 **Style file** 属性中点选『.4st』档，元件的 style 属性就会出现这个样式档中所定义的所有样式的下拉式选项。

选取后，可以在预览画面看到结果。



Tab Index（元件排列顺序）

定义画面上元件的输入顺序（按 Tab 键可以跳到下一个元件）。可以在 Edit Menu 中的 Tab Index 或是如下图中的小图示上按鼠标左键执行。




更改顺序方式：

- 双击鼠标左键将顺序数字重设为 1。
- 在任一数字上按鼠标左键将从目前的数字顺序往下排。
- 结束目前的设定按 ESC 键。
- 再按一次 ESC 键离开 Tab index 设定（数字会消失）。

更改顺序也可以透过直接更改元件的 tabindex 属性的数值，但必须在离开 Tab Index 设定顺序的状态下。

Alignment（对齐）

在图示列  或是在选单中 Alignment 的功能。

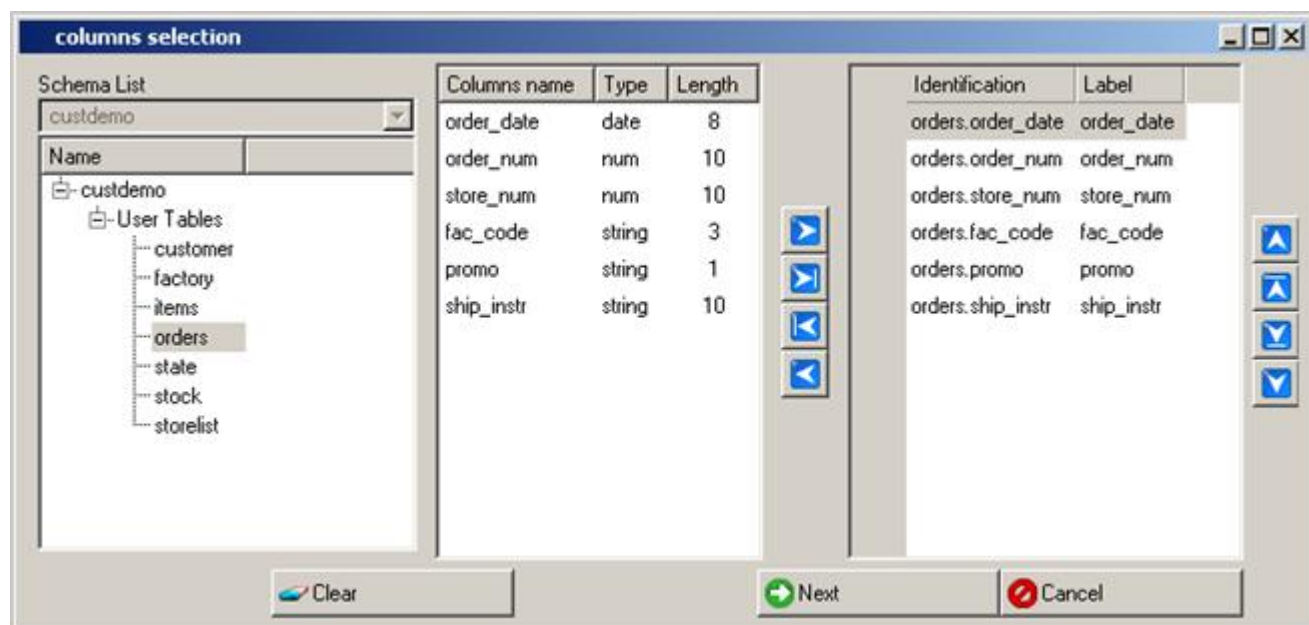
只有在同一个容器（**container**）内的元件（**widgets**）可以进行对齐功能。

也可以同时选取多个元件修改它们的 PosX/PosY 属性以达到对齐的功能。这些属性对上层容器物件来说是相对位置，而不是绝对位置。

DataControl





选择栏位





由 Container→DataControl 进入。



1. 若还未有 Schema 资料，则先利用 Database Browser 加入 Schema。
2. 接着再执行 DataControl，选择一个 schema 资料。Schema List 内会列出所有可以选择的资料表（左边面板）及栏位（中间面板）。
3. 利用中间面板与右边面板中的箭头图示按钮选择栏位放入 DataControl 中（右边面板）。
4. 利用最右边的箭头图示按钮调整 DataControl 栏位的排列顺序。在右边面板的标签（Label）栏位双击鼠标左键可以编辑标签的显示文字。

按钮说明：

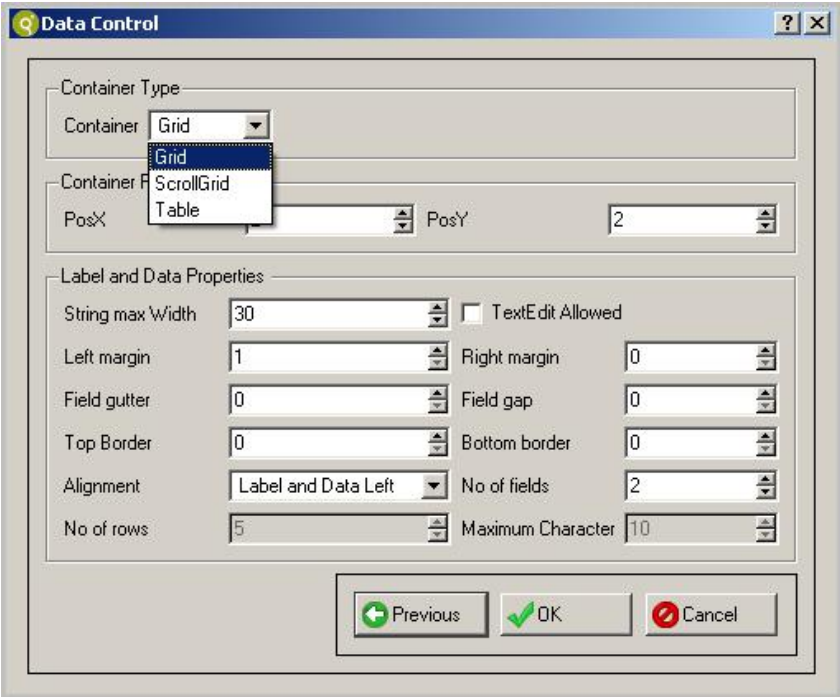
| | |
|---|----------------------------|
|  | 将中间面板选定的栏位新增到右边面板。 |
|  | 将中间面板所有栏位新增到右边面板。 |
|  | 移除右边面板（DataControl）中的所有栏位。 |
|  | 移除右边面板（DataControl）所选定的栏位。 |

| | |
|---|----------------|
|  | 将选定的栏位移到前一个位置。 |
|  | 将选定的栏位移到顶点位置。 |
|  | 将选定的栏位移到最后的位置。 |
|  | 将选定的栏位移到后一个位置。 |

| | |
|---|---------------------------------------|
|  | 移除右边面板（DataControl）中的所有栏位。 |
|  | 验证目前的选择并执行下一步。 |
|  | 离开 DataControl 回到 Form Designer 设计画面。 |

画面配置

选择完栏位后，按下“Next”，设定栏位在画面上的排列方式。



在 DataControl 的这些属性的预设值都可以在 **Preferences** 中设定。

以下为上图设定区块说明：

1. **Container Type:** 所产生的容器物件的型态，有 Grid、ScrollGrid、Table 三种。

2. **Container Properties:** （容器属性）

- PosX: 容器物件左上角的 X 坐标（水平）。
- PosY: 容器物件左上角的 Y 坐标（垂直）。

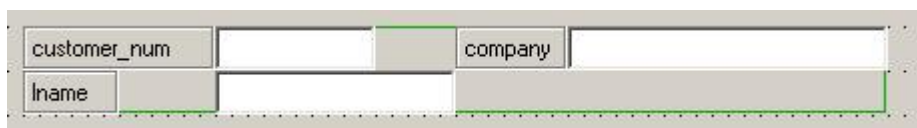
3. **Label and Data Properties:** （标签与资料属性）

- 通用属性：
 - TextEdit Allowed: 勾选时，资料栏位型态为 TEXT 的栏位会产生成 TextEdit 元件，不勾选时，则 TEXT 型态的栏位产生成 Edit 元件。
- Grid / ScrollGrid 使用的属性：
 - String max Width: 若资料栏位的长度超过这个数值，则产生的 EDIT 元件的 gridWidth 就会等于这个数值。
 - Left and Right Margin: 产生的资料栏位组(一个标签与一个资料栏位)跟容器物件的左右边界所保留的空格符数。
 - Field gutter: 宽度最宽的标签与资料栏位之间保留的空格符数。
 - Field gap: 宽度最宽的资料栏位组与同行下一个资料栏位组之间保留的空格符数。
 - Top and Bottom border: 产生的资料栏位组(一个标签与一个资料栏位)跟容器物件的上下边界所保留的空格符数。
 - Alignment: 资料栏位组对齐的设定。
 - No. of fields: 每行的资料栏位组数目。
- Tables 使用的属性：
 - No. of rows: Table 的行数。
 - Maximum Characters: 每个栏位的最大字元数(宽度)。

设定完成后，将产生新的画面。

画面范例：

Grid:



ScrollGrid:

A screenshot of a ScrollGrid control. It contains a form with four fields: 'order_num' with the value '0', 'price' (empty), 'quantity' (empty), and 'stock_num' with the value '0'. Each field has a small blue arrow icon to its right, indicating it is a scrollable numeric field. The form is enclosed in a green border.

Table:


| customer_num | company | lname |
|--------------|---------|-------|
| | | |
| | | |
| | | |
| | | |
| | | |

验证、预览、编译画面档

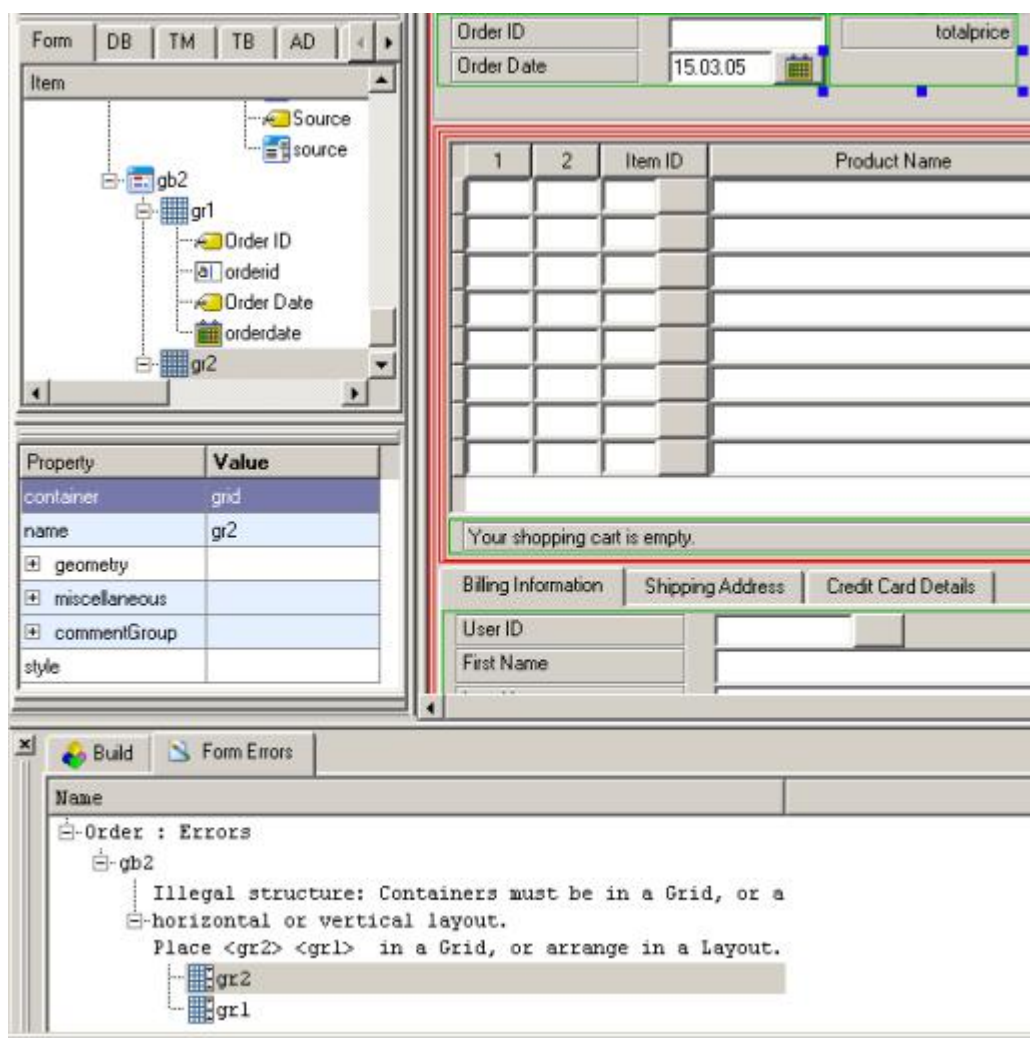
储存档案

在 File Menu 中选取『Save』或『Save as』将设计的画面存成副档名为『.4fd』的档案格式。

验证画面档

按图示  或是选单 Build 中的 Validate 可进行画面档的验证。

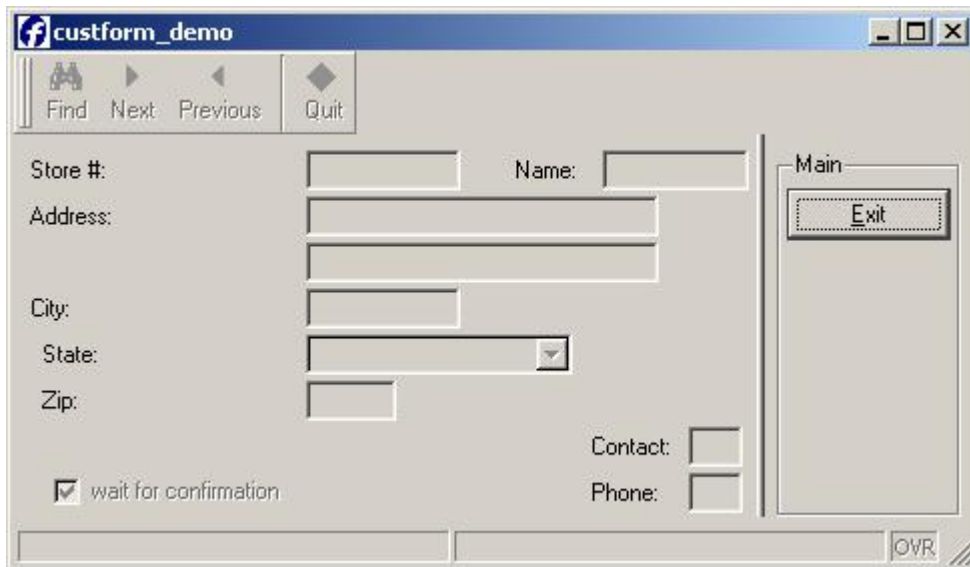
不正确的 Form 结构（例：元件没有放置在 Grid 中）会导致编译失败，错误讯息会显示在输出面板（Output Panel）的 Form Errors 页签上。



在输出面板上的『错误项』按鼠标左键，可以移到指定的物件，按照提示的讯息修正它并重试一次。

预览画面

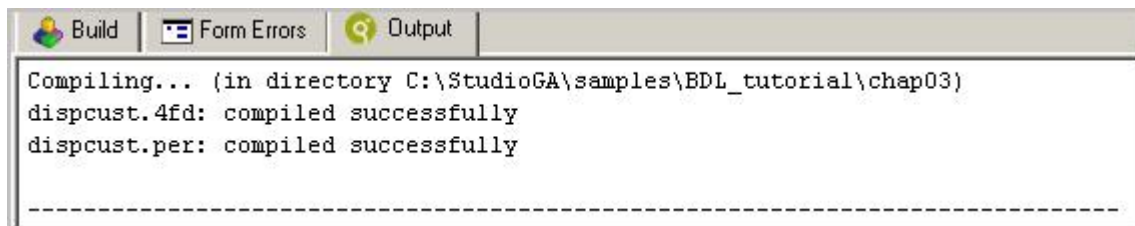
Genero Studio 所提供的 preview 功能因为需要安装 Genero 的 Develop License，若无法安装时则无法使用。（注）



编译画面档

可从图示列或是选单 Build 中的 Compile 选项执行。

编译动作会将画面定义转换成副档名为“.42f”的 XML 档案。这个 XML 档案被用在客户端环境中显示应用程序的画面。编译结果会显示在输出面板。



注: TIPTOP GP 另行提供 r.gf 指令替代 Genero Studio 的预览功能, 此指令在 TIPTOP GP 手册中会详细说明。

Chapter 3

变量及运算

变量的定义（DEFINE Variable）

设定变量的资料型态。

在程序中若要使用变量（Variables），此变量必须先经定义（DEFINED），宣告此变量的名称及资料型态，且须注意的是变量的定义必须在 MAIN，FUNCTION 或 GLOBALS 或 REPORT FUNCTION 的叙述之后。

直接定义

```
DEFINE employee_no CHAR(10)
```

此范例中，直接定义 employee_no(员工编号)这个变量型能是 10 个字符。

对应数据库栏位

```
DATABASE database_id  
  
DEFINE p_employee_no LIKE employee_file.employee_no
```

此范例中，定义 p_employee_no 与数据库中的 employee_file 这个 table 的 employee_no 栏位有相同的资料型态。

说明：

1. 使用对应数据库栏位的设定时，需在设定前以『DATABASE database_name』指定设定要参照的数据库名称。
2. 程序撰写时，尽量以对应数据库栏位方式设定变量型态，若后续有对栏位形态、栏宽设定进行异动时，只需重新编译程序即可达到型态更新的目的。
若采用直接设定的方式，则遇有栏位型态、宽度要变更时，可能就必需同时进行大量的程序码修正，较为不便。

变量的型态

| 型态名称 | 说明 | 预设值 |
|---------------|--|------------|
| 字符资料型态 | | |
| CHAR | 固定字符 | Null |
| VARCHAR | 动态字符 | Null |
| STRING（注） | 动态定义大小的字符串储存空间 | Null |
| 日期资料型态 | | |
| DATE | 日期 | 1899-12-31 |
| DATETIME | 日期时间 | Null |
| 数值资料型态 | | |
| INTEGER | 4 BYTE 整数 | Zero |
| SMALLINT | 2 BYTE 整数 | Zero |
| FLOAT | 8 BYTE 单精度浮点数 | Zero |
| SMALLFLOAT | 4 BYTE 双精度浮点数 | Zero |
| DECIMAL(p,s) | 高精度数值-precision 为所有数字个数(不含小数点), scale 为小数点的位数, 不可大于 precision 的位数 | Null |
| MONEY | 同 DECIMAL, 但会在数值前面加上货币符号 | Null |
| 大量资料型态 | | |
| BYTE | 大量二进制资料(图片) | Null |
| TEXT | 大量字符资料 | Null |

注：Genero BDL 对于型别转换的要求较其他语言比较不算严谨，只要合理可转换的，系统均会自行判断进行转换。

注：STRING 为一特别的资料型态，具有基本的物件属性，常用的属性在本章中均会以附注方式说明用法。

说明：STRING 除截取资料有特别的物件函式（以下称 METHOD）外，也提供其他功能的 METHOD，整理功能如下：

| METHOD 名称 | 功能说明 |
|--|----------------------------|
| append(str STRING) RETURNING STRING | 将传入字符串加到原来的 STRING 后。 |
| equals(src STRING) RETURNING INTEGER | 判断原字符串与传入字符串是否相等。 |
| equalsIgnoreCase(src STRING) RETURNING INTEGER | 判断原字符串与传入字符串是否相等（忽略大小写差异）。 |
| getCharAt(pos INTEGER) RETURNING STRING | 抓取指定位置的字符。 |
| getIndexOf(str STRING, spos INT) RETURNING INTEGER | 从 INT 处开始寻找，判断是否含有传入字符串。 |
| getLength() RETURNING INTEGER | 计算此字符串总长度。 |
| subString(spos INT, epos INT) RETURNING STRING | 切截出指定起点至终点的子字符串。 |
| toLowerCase() RETURNING STRING | 将字符串转换为小写字。 |
| toUpperCase() RETURNING STRING | 将字符串转换为大写字。 |
| trim() RETURNING STRING | 切掉字符串头尾两侧的空格符。 |
| trimLeft() RETURNING STRING | 切掉字符串头端的空格符。 |
| trimRight() RETURNING STRING | 切掉字符串尾端的空格符。 |

变量的集合（Records）

不同资料型态变的组合。

直接定义

```
MAIN
  DEFINE rec RECORD
        id    INTEGER,
        name  VARCHAR(100),
        birth  DATE
  END RECORD
END MAIN
```

此范例中，直接定义 rec 这个 Records 中的各个变量型态。

对应数据库栏位

```
DATABASE example_database
MAIN
  DEFINE cust01 RECORD LIKE customer.*
  DEFINE cust02 RECORD
        id    LIKE customer.id,
        name  LIKE customer.name,
        birth  LIKE customer.birth,
        sales LIKE salesman.name
  END RECORD
END MAIN
```

此范例中，定义 cust01 这个 Record 的变量与数据库中的 customer 这个 table 的栏位有相同的名称及资料型态。若在这组变量中有穿插非同一 TABLE 栏位，或是需要做顺序上的调动，则必需参照 cust02 的作法，将变量组中的每一个项目独立写出。

变量的设定

定义完变量后，可利用 LET 指令指定变量值。

语法：LET variable = expression

范例：

```
DATABASE ds
MAIN
  DEFINE c1, c2  CHAR(10)
  LET c1 = "Genero"
  LET c2 = c1
END MAIN
```

注：若变量形态为 CHAR 和变量形态为 VARCHAR 时，指定给予的值有差异。

```
DATABASE ds
MAIN
  DEFINE c1, c2  CHAR(10)
  DEFINE c3      VARCHAR(10)
  LET c1 = "Genero"
  LET c2 = c1
  LET c3 = c1
END MAIN
```

```
-- c2 = 『Genero   』（含空格）
-- c3 = 『Genero』（无空格）
```

初始化一组变量的值

若要初始化一组 RECORD 变量为 NULL，或者是初始化为数据库 Table 的预设值，可利用 INITIALIZE。

语法：INITIALIZE 变量串列 { LIKE 栏位串列 | TO NULL }

范例：

```
DATABASE ds
MAIN
  DEFINE cr RECORD LIKE customer.*
  INITIALIZE cr.cust_name TO NULL
  INITIALIZE cr.* LIKE customer.*
END MAIN
```


预定义完成变量（PRE-DEFINED Variable）

以下说明一些系统的预先定义变量及用途：

| 变量名称 | 说明 |
|----------|--|
| INT_FLAG | 当设定 DEFER INTERRUPT 时，系统即会在使用者每次按下『中断键』时，将此变量设定为『TRUE』，须要程序恢复回原值（FALSE）。 |
| STATUS | 储存每次 SQL 的执行状态。 |

常数的定义（CONSTANT）

设定常数的资料（型态）。

语法： `CONSTANT constant_id [data_type] = value`

常数可以直接指定其值，可以不需指定其型态，若欲指定型态但又指定错误，系统会自行修正为正确型态。

范例：

| | |
|------------------------------|-------------------|
| CONSTANT c1 = "Drink" | -- 自行宣告为 STRING |
| CONSTANT c2 = 4711 | -- 自行宣告为 INTEGER |
| CONSTANT c3 SMALLINT = 12000 | -- 自行修正为 INTEGER |
| CONSTANT c4 CHAR(10) = "abc" | -- 遵照设定为 CHAR(10) |

预定义完成常数（PRE-DEFINED Constant）

以下说明一些系统的预先定义常数及用途：

| 常数名称 | 说明 |
|----------|--|
| NULL | 即表示 NULL。 |
| TRUE | 表示布林逻辑中的『非零』值，预设为『1』，但不可视为『1』运算。 |
| FALSE | 表示布林逻辑中的『零』值，预设为『0』，但不可视为『0』运算。 |
| NOTFOUND | 表示 SQL 讯息中的『找不到 notfound』，预设为『100』，但不可视为『100』运算。 |

表达式

比较表达式

| 表达式 | 意义 | 范例 |
|---------|---------------|------------------------------|
| IS NULL | 空白 | IF a IS NULL THEN... |
| == 或 = | 等于 | IF a == 10 THEN... |
| != 或 <> | 不等于 | IF a != 10 THEN... |
| < | 小于 | IF a < 10 THEN... |
| <= | 小于等于 | IF a <= 10 THEN |
| > | 大于 | IF a > 10 THEN... |
| >= | 大于等于 | IF a >= 10 THEN... |
| := | 指定值（用于 LET 内） | LET a = b := 10 |

数值表达式

| 表达式 | 意义 | 范例 |
|-----|-----|------------------------|
| + | 加法 | LET a = a + b |
| - | 减法 | LET a = a - b |
| * | 乘法 | LET a = a x b |
| / | 除法 | LET a = a / b |
| ** | 次方 | LET a = a ** b |
| Mod | 取余数 | LET a = a MOD b |

范例：

```
MAIN
  DEFINE i,j SMALLINT
  LET i = 9
  LET j = 2
  DISPLAY i + j
  DISPLAY i - j
  DISPLAY i * j
  DISPLAY i / j
  DISPLAY j ** i
  DISPLAY i mod j
END MAIN
```

```
-- Displays 11
-- Displays 7
-- Displays 18
-- Displays 4.5
-- Displays 512
-- Displays 1
```

字符串表达式

| 表达式 | 意义 | 范例 |
|-------------|----------------------------------|------------------------|
| , | 字符串连结 (append) | LET a = a, b |
| | 字符串连结, 但连结值有一个是 NULL, 结果就会是 NULL | LET a = a b |
| [start,end] | 从字符串中取出子字符串【只能用在 CHAR、VARCHAR】 | LET a = a[1,10] |
| USING | 针对数值或日期设定其打印的格式 | LET a = a USING "####" |
| CLIPPED | 消除尾部空白 | LET a = a CLIPPED |
| SPACES | 输出空白字符串 | LET a = 10 SPACES |
| ASCII | 指定以 ASCII 码方式输入文字 | LET a = ASCII 37 |

范例:

| | |
|--|--|
| <pre> MAIN DEFINE i,j,k CHAR(20) LET i = "TIPTOP GP" LET j = "Genero BDL" DISPLAY i , j CLIPPED DISPLAY i j CLIPPED DISPLAY i j k DISPLAY i[4,6] DISPLAY i CLIPPED , j CLIPPED DISPLAY 5 SPACES,i END MAIN </pre> | <pre> --Displays "TIPTOP GP Genero BDL" --Displays "TIPTOP GP Genero BDL" --Displays "" --Displays "TOP" --Displays "TIPTOP GPGenero BDL" --Displays 1" TIPTOP GP" </pre> |
|--|--|

说明:

表达式[start,end]表示从字符串中取出子字符串, 此表示方式仅能用在 CHAR 或 VARCHAR 上, 若变量型态为 STRING, 则参照如下范例:

| | |
|--|--------------------------------------|
| <pre> MAIN DEFINE i,j STRING LET i = "TIPTOP GP" LET j = i.subString(1,6) DISPLAY j END MAIN </pre> | <pre> --Displays j "TIPTOP" </pre> |
|--|--------------------------------------|

逻辑表达式

| 表达式 | 意义 | 范例 |
|-----|----|----------------------------|
| NOT | 反向 | IF NOT a = b THEN... |
| AND | 而且 | IF a = b AND c = d THEN... |
| OR | 或者 | IF a = b OR c = d THEN... |

日期表达式

| 表达式 | 意义 | 范例 |
|---------|-----------------|------------------------------|
| TODAY | 取出今天的日期 | LET a = TODAY() |
| CURRENT | 回传现在的时间和日期 | LET a = CURRENT |
| DATE | 转换为日期型态 | LET a = DATE("07/31/2005") |
| MDY | 组成日期形态 | LET a= MDY(07, 31, 2005) |
| TIME | 取出时间值 | LET a = TIME (CURRENT) |
| YEAR | 取出年 | LET a = YEAR(CURRENT) |
| MONTH | 取出月 | LET a = MONTH(CURRENT) |
| DAY | 取出日 | LET a = DAY(CURRENT) |
| WEEKDAY | 回传今天是第几个工作天（星期） | LET a = WEEKDAY(CURRENT) |

栏位表达式

| 表达式 | 意义 | 范例 |
|---------|-------------|---------------------------|
| INFIELD | 判断是否在某画面栏位中 | IF INFIELD(aaa01) THEN... |

全域变量指定（GLOBALS）

此指令用在辨视哪些变量要列入系统的全域变量部份，以使列入的变量在一个完整的作业中均能随意的被取用变更。GLOBALS 有两种写法，语法一为直接定义，语法二则可引入外部的设定档，在系统程序数量较大时，建议采『语法二』的作法。

语法一：直接写定 GLOBALS 区块

GLOBALS

declaration-statement

[,...]

END GLOBALS

语法二：读入已写好的共同设定档（外部档案）。

GLOBALS "filename"

变量的生命周期

Local 变量(Local Variables)

定义位置:定义在 Module 中的函式里 (MAIN、FUNCTION 等)。

生命周期:只属于该定义的函式使用，离开此函式即不能再使用。

Module 变量(Module Variables)

定义位置: Module 中，但不被任何的函式包围。

生命周期:为该 Module 中的共享变量。

Global 变量(Global Variables)

定义位置:由 GLOBALS 及 END GLOBALS 所包围的变量。

生命周期:为所 link 使用的所有 MODULE 的共享变量。

范例:

```
DATABASE ds
GLOBALS
    DEFINE g_employee CHAR(10)
END GLOBALS
    DEFINE g_tty CHAR(32)
MAIN
    DEFINE answer CHAR(1)
END MAIN
FUNCTION ins_employee()
    DEFINE flag    CHAR(1),
           change  SMALLINT
END FUNCTION
```

以上范例属于 GLOBAL 变量的有 g_employee

属于 MODULE 变量的有 g_tty

属于 LOCAL 变量的有 answer、flag、change

USING

针对数值或日期设定其打印或显示于画面上的格式，若需设定时，须注意溢位（overflow）的问题。

数值格式标志

| | |
|----|---------------------------------|
| * | 空白的地方以*置换 |
| & | 空白的地方以 0 置换 |
| # | 不会对输出的数字作任何影响，通常用于限制字符串输出时的最大长度 |
| < | 将数字改为向左靠 |
| , | 指定逗号出现的位置 |
| . | 指定小数点出现的位置 |
| - | 当输出的数字小于零时，加上一个负号 |
| + | 当输出的数字大于零，加上一个正号 |
| | 当输出的数字小于零，加上一个负号 |
| \$ | 数值出现一个钱字号 |
| (| 当输出的数字小于零时，加上一个左括号 |
|) | 当输出的数字小于零时，加上一个右括号 |

日期格式标志

| | |
|------|-------------|
| Dd | 以二位数字表示日期 |
| Ddd | 以三位英文简写表示星期 |
| Mm | 以二位数字表示月份 |
| Mmm | 以三位英文简写表示月份 |
| Yy | 以二位数字表示年度 |
| Yyyy | 以四位数字表示年度 |

范例一：

| | |
|---|--|
| <pre> MAIN DEFINE i,j SMALLINT LET i = 12345 LET j = -12345 DISPLAY i DISPLAY j DISPLAY i USING "*****" DISPLAY j USING "*****" DISPLAY i USING "&&&&&&" DISPLAY j USING "&&&&&&" DISPLAY i USING "#####" DISPLAY j USING "#####" DISPLAY i USING "<<<<<<" DISPLAY j USING "<<<<<<" DISPLAY i USING "-----" DISPLAY j USING "-----" DISPLAY i USING "++++++" DISPLAY j USING "++++++" DISPLAY i USING "\$\$\$\$\$\$" DISPLAY j USING "\$\$\$\$\$\$" DISPLAY i USING "(#####)" DISPLAY j USING "(#####)" DISPLAY i USING "###,###.&&" DISPLAY j USING "###,###.&&" END MAIN </pre> | <pre> -- Displays: 12345 -- Displays: -12345 -- Displays:**12345 -- Displays:**12345 -- Displays:0012345 -- Displays:0012345 -- Displays: 12345 -- Displays: 12345 -- Displays:12345 -- Displays:12345 -- Displays: 12345 -- Displays: -12345 -- Displays: +12345 -- Displays: -12345 -- Displays: \$12345 -- Displays: \$12345 -- Displays: 12345 -- Displays:(12345) -- Displays: 12,345.00 -- Displays: 12,345.00 </pre> |
|---|--|

范例二：

| | |
|---|---|
| <pre> MAIN DISPLAY TODAY USING "yyyy-mm-dd" DISPLAY TODAY USING "yy-mm-dd" DISPLAY TODAY USING "yy-mmm-ddd" END MAIN </pre> | <pre> -- Displays 2004-06-15 -- Displays 04-06-15 -- Displays 04-Jun-Tue </pre> |
|---|---|

A decorative crosshair consisting of a vertical line and a horizontal line intersecting in the center of the page.

Chapter 4

程控流程

Module 程序架构

在第一章中已有讨论：『一个完整的作业，应为程序（4GL）与画面档（PER）的组成』，在第二张中也已讨论过画面档结构。本章则讨论程序部份。程序部份可区分为 MAIN() 函式、一般 FUNCTION() 函式及报表结构定义 REPORT() 函式等。

MAIN() 函式

MAIN() 函式是程序执行的入口，一个完整可执行的程序必定仅含一个 MAIN() 函式。

范例：

```
MAIN
  DISPLAY "Hello, world!"
END MAIN
```

说明：如以上的范例，仅含有一个 MAIN 函式，可构成完整的程序功能，因此已可视为一个完整的作业。

MAIN() 函式可简单撰写如上范例所示，也可增加一些设定区块（Control Block），以下列出特别会定义在 MAIN() 函式中的控制设定区块。下列控制区块除非特别说明，否则均为一经设定，在此作业中全都有效。

DEFER 设定

此设定可定义程序是否要拦截『当使用者按下中断（interrupt）或离开（quit）键』时所送出的系统讯号。

语法：DEFER { INTERRUPT | QUIT }

说明：

- 当设定『DEFER INTERRUPT』时，当系统侦测到使用者按下中断键（一般为 Control-C 键或是 Delete 键），会将讯息传送到变量 INT_FLAG 中，令其为 TRUE。若程序中存在『ON ACTION interrupt』，则系统也会触发执行。
- 当设定『DEFER QUIT』时，当系统侦测到使用者按下离开键，会将讯息传送到变量 QUIT_FLAG 中，令其为 TRUE。并中断程序。

OPTIONS 设定

此段设定可变更系统预设的选项。

| 语法 | 意义 | 备注 |
|-----------------|--------------------|-----------------|
| INPUT [NO] WRAP | 在离开最后一个输入格后，是否重新开始 | NO WRAP 表示不重新开始 |
| FORM LINE | 画面开始显示的位置 | 仅于 Console 模式有效 |
| MESSAGE LINE | 讯息显示的位置 | 仅于 Console 模式有效 |
| PROMPT LINE | 提示说明显示的位置 | 仅于 Console 模式有效 |

Exceptions 设定

定义当遇到 SQL 错误时，系统要采取何种方式因应。此设定不限 MAIN() 函式可设定，此设定在程序中想改变因应方式时可再次指定，即可以新设定方式处理。

语法：

WHenever [ANY] ERROR { CONTINUE | STOP | CALL function | GOTO label }

范例：

```
MAIN
  OPTIONS                                #改变一些系统预设值
    FORM LINE      FIRST + 2,           #画面开始的位置
    MESSAGE LINE   LAST,                #讯息显示的位置
    PROMPT LINE    LAST,                #提示讯息的位置
    INPUT NO WRAP                                     #输入的方式：不打转
  DEFER INTERRUPT

  Whenever ERROR STOP

  DISPLAY "Change Exception!"

  Whenever ERROR CALL chk_err           #此处的 CALL 是没有括号的

END MAIN

FUNCTION chk_err( )
  DISPLAY "Error Happened!"
END FUNCTION
```

一般 FUNCTION() 函式

执行某个特定功能的子函式，程序中可将某些功能独立编写为一个个的子函式，以供互相呼叫之用，如此即可趋近模块化的目标。同一支完整作业内的 Function 名称不能相同。

范例：

```
MAIN
  CALL sayIt()
END MAIN

FUNCTION sayIt()
  DISPLAY "Hello, world!"
END FUNCTION
```

报表结构 REPORT() 函式

为函式的一种，专门用来设定报表打印格式，后续章节有详细的介绍。

REPORT 结构范例：

```
REPORT test_rep(sr)
. . .
  FORMAT
    PAGE HEADER
    BEFORE GROUP
    ON EVERY ROW
    AFTER GROUP
. . .
END REPORT
```

4GL 注解符号

- {} 可以将某个范围做备注。
- # 将某行做备注。
- -- 将某行做备注。

CALL

执行指定的函数（Function），若有回传值，以 RETURNING 接回。

语法：

```
CALL function ( [ parameter [...] ] ) [ RETURNING variable [...] ]
```

RETURN

传回原呼叫函数所需的变量值，并停止此子函数的执行。

语法：

```
RETURN [ value [...] ]
```

范例一：回传单一值

```
MAIN
  DEFINE var1 CHAR(10)
  DEFINE var2 CHAR(2)
  LET var1 = foo()
  DISPLAY "var1 = " || var1
  CALL foo() RETURNING var2
  DISPLAY "var2 = " || var2
END MAIN

FUNCTION foo()
  RETURN "Hello"
END FUNCTION
```

范例二：回传单一值（布林值）

```
MAIN
  IF foo() THEN
    DISPLAY "Choice is OK!"
  END IF
END MAIN

FUNCTION foo()
  RETURN TRUE
END FUNCTION
```

范例 3：回传多值

```
MAIN
  DEFINE var1 CHAR(15)
  DEFINE var2 CHAR(15)
  CALL foo() RETURNING var1, var2
  DISPLAY var1, var2
END MAIN

FUNCTION foo()
  DEFINE r1 CHAR(15)
  DEFINE r2 CHAR(15)
  LET r1 = "return value 1"
  LET r2 = "return value 2"
  RETURN r1, r2
END FUNCTION
```

范例 4-1: (test1.4gl)

```
MAIN
  DISPLAY "MAIN FUNCTION"
  CALL a1()
  CALL a2()
  RUN "fglrun test3"
END MAIN

FUNCTION a1()
  DISPLAY "SUB FUNCTION a1()"
END FUNCTION
```

以上程序段会呼叫另外两个 Function，并且利用 RUN 指令执行一道 unix 指令。

范例 4-2: (test2.4gl)

```
FUNCTION a2()
  # a2 function
  DISPLAY "SUB FUNCTION a2()"
END FUNCTION
```

范例 4-3: (test3.4gl)

```
MAIN
  DISPLAY "This is test3.4gl"
END MAIN
```

IF

依条件执行程序

语法:

```
IF condition THEN
    statement
    [...]
[ ELSE
    statement
    [...]
]
END IF
```

范例:

```
MAIN
  DEFINE name CHAR(20)
  LET name = "John Smith"
  IF name MATCHES "John*" THEN
    DISPLAY "The first name is too common to be displayed."
    IF name MATCHES "*Smith" THEN
      DISPLAY "Even the last name is too common to be displayed."
    END IF
  ELSE
    DISPLAY "The name is " , name , "."
  END IF
END MAIN
```

CASE

执行符合条件的特定程序段

语法一：

```
CASE expression-1
  WHEN expression-2
    { statement | EXIT CASE }
    [...]
  [ OTHERWISE
    { statement | EXIT CASE }
    [...]
  ]
END CASE
```

说明：若判别式很单纯，例如：只需判断一个数字的数值，因该值不同而有不同的选项时，可采用此写法（如下列范例）。若判断式较复杂，或可能同时出现两种以上的条件（成立）时，请改用语法二。

范例：

```
MAIN
  DEFINE v CHAR(10)
  LET v = "C1"
  CASE v
    WHEN "C1"
      DISPLAY "Value is C1"
    WHEN "C2"
      DISPLAY "Value is C2"
    WHEN "C3"
      DISPLAY "Value is C3"
    OTHERWISE
      DISPLAY "Unexpected value"
  END CASE
END MAIN
```


语法二：

```
CASE
  WHEN boolean-expression
    { statement | EXIT CASE }
    [...]
  [ OTHERWISE
    { statement | EXIT CASE }
    [...]
  ]
END CASE
```

范例：

```
MAIN
  DEFINE v CHAR(10)
  LET v = "C1"
  CASE
    WHEN ( v="C1" OR v="C2" )
      DISPLAY "Value is either C1 or C2"
    WHEN ( v="C3" OR v="C4" )
      DISPLAY "Value is either C3 or C4"
    OTHERWISE
      DISPLAY "Unexpected value"
  END CASE
END MAIN
```

注：当采用此种语法时需注意：需避免同时有两种情况成立的情形发生。若在未注意的状况下发生，则系统仅会走第一条符合条件路径。

```
MAIN
  DEFINE a,b  INT
  LET a = b := 10

  CASE
    WHEN a=10    DISPLAY " a is ok "
    WHEN b=20    DISPLAY " b is ok "
    OTHERWISE    DISPLAY " nothing is ok "
  END CASE
END MAIN
```

画面仅会显示『 a is ok 』

FOR

依指定的次数执行程序

语法:

```
FOR counter = start TO finish [ STEP value ]  
    statement  
    [...]  
END FOR
```

范例:

```
MAIN  
  DEFINE i, i_min, i_max INTEGER  
  LET i_min = 1  
  LET i_max = 10  
  DISPLAY "Look how well I can count from " , i_min , " to " , i_max  
  DISPLAY "I can count forwards..."  
  FOR i = i_min TO i_max  
    DISPLAY i  
  END FOR  
  DISPLAY "... and backwards!"  
  FOR i = i_max TO i_min STEP -1  
    DISPLAY i  
  END FOR  
END MAIN
```

WHILE

执行程序直到条件式不成立为止

语法:

```
WHILE b-expression  
    statement  
    [...]  
END WHILE
```

范

例:

```
MAIN  
  DEFINE a,b INTEGER  
  LET a=20  
  LET b=1  
  WHILE a > b  
    DISPLAY a , "> " , b  
    LET b = b + 1  
  END WHILE  
END MAIN
```

CONTINUE

重新执行循环

语法:

```
CONTINUE { FOR | FOREACH | MENU | CONSTRUCT | INPUT |  
          WHILE }
```

范例:

```
MAIN  
  DEFINE i INTEGER  
  LET i = 0  
  WHILE i < 5  
    LET i = i + 1  
    DISPLAY "i=" || i  
    CONTINUE WHILE  
    DISPLAY "This will never be displayed !"  
  END WHILE  
END MAIN
```

EXIT

离开控制段

语法：

```
EXIT { CASE | FOR | MENU | CONSTRUCT | FOREACH | REPORT |  
      DISPLAY | INPUT | WHILE | PROGRAM (注) }
```

范例：

```
MAIN  
  DEFINE i INTEGER  
  LET i = 0  
  WHILE TRUE  
    DISPLAY "This is an infinite loop. How would you get out of here ?"  
    LET i = i + 1  
    IF i = 100 THEN  
      EXIT WHILE  
    END IF  
  END WHILE  
  DISPLAY "Well done."  
END MAIN
```

注：若程序执行到 EXIT PROGRAM，则表示『正常离开作业』（非异常中止），所以不会有任何 stderr 码传出。

但此指令后可跟随 exit_code，exit_code 可为 8bit 的数值，系统传出时会转换为正整数，以供原呼叫此作业之程序判别状态。

SLEEP

程序依指定秒数暂停

语法: **SLEEP** *seconds*

注意: 如果 *seconds* < 0 或 *seconds* IS NULL, 则程序不会停止。

范例:

```
MAIN
  DISPLAY "Please wait 5 seconds..."
  SLEEP 5
  DISPLAY "Thank you."
END MAIN
```

LABEL

宣告程序注标。

语法: **LABEL** *label-id*:

注意: 结尾处有一冒号 (:)。

GOTO

移到指定的程序段继续执行 (为了程序的易读性与结构性, 建议勿使用此指令)。

语法: **GOTO:** *label-id*

范例:

```
MAIN
  DISPLAY "Before GOTO"
  GOTO: label_id1
  DISPLAY "Never Been Displayed"
  LABEL label_id1:
  DISPLAY "After GOTO"
END MAIN
```



Chapter 5

WINDOWS 与 FORM

WINDOWS 与 FORM

在上一章节介绍如何编辑画面，但 WINDOWS 与 FORM 本身是无法执行的，它必须透过 Program 去启动它，以下介绍在程序中启动及显示 WINDOWS & FORM 的指令。

OPEN WINDOW

语法：

```
OPEN WINDOW window-id [AT line, column ]
      WITH [ FORM form-file | height ROWS, width COLUMNS ]
      [ ATTRIBUTES ( window-attributes ) ]
```

说明：

- ***window-id***: 定义这个 window name 名称。
- **AT *line, column***: 表示让画面开启的起始坐标，仅限于 Console 上执行有效。
- ***form-file***: 经过编译后的画面档文件名称(不含附档名)，之前可以指定放置路径。
- ***height* ROWS, *width* COLUMNS**: 实际画面档不存在时，可先指定画面占用行数及栏数，其他部份待 4GL 执行时再行动态设定。
- **ATTRIBUTES (*window-attributes*)**: 可以加上属性设定。

ATTRIBUTE 属性

通用

| Attribute | 系统 Default | 说明 |
|----------------|------------|----------------------|
| TEXT = string | NULL | 将 string 显示在视窗标题列 |
| STYLE = string | NULL | 读取 string 的画面设定属性【注】 |

主机执行用（仅限于 Console 模式）

| Attribute | 系统 Default | 说明 |
|-----------|------------|--------------|
| BORDER | No border | WINDOWS 外加边框 |
| REVERSE | No reverse | |

【注】：Genero BDL 语言有一画面设定档格式，附档名为『.4st』，TIPTOP GP 将此设定档置于『\$TOPCONFIG/4st』目录下并命名为『tiptop.4st』，使用者若有兴趣自行设定，可参照系统在线说明文件进行设定。

CLEAR

在目前所显示的画面，清除指定的栏位变量内容

语法: CLEAR *field-list*

范例:

```
FOR i=1 TO 10
    CLEAR g_employee[i].*
END FOR
```

CLEAR FORM

此指令是清除目前所执行视窗（含 WINDOW 及 FORM）内所有变量内容。但它不影响其他画面栏位显示状态。

语法: CLEAR FORM *window-id*

范例:

```
MAIN
    OPEN WINDOW w1 AT 1, 1 WITH FORM "employee"
    CLEAR FORM
    CLOSE WINDOW w1
END MAIN
```

CLOSE WINDOW

画面用完后，可透过 CLOSE WINDOW 指令将视窗关闭。

语法：CLOSE WINDOW *window-id*

范例：

```
MAIN
  DEFINE  Is_flow_pic  STRING
  OPEN WINDOW act_w AT 3, 10 WITH FORM "/u1/users/top/topmenu"
  ATTRIBUTE(BORDER, CYAN)
  MENU ""
    ON ACTION add
      CALL act_a()
      :
    ON ACTION quit
      EXIT Program
  END MENU
  CLOSE WINDOW act_w

END MAIN
```

实做样式：在画面开出后，按下离开（exit），就会关闭视窗。

CURRENT WINDOW

程序部份需要开很多视窗，且需切换不同视窗，则可以利用 CURRENT WINDOW。

语法：

CURRENT WINDOW IS *window-id*

范例：

```
MAIN
  OPEN WINDOW w1 WITH FORM "edit"
  OPEN WINDOW w2 WITH FORM "topmenu"
  MENU "Change Windows"
    ON ACTION edit
      CURRENT WINDOW IS w1
      CALL act1_a()

    ON ACTION topmenu
      CURRENT WINDOW IS w2
  ON ACTION exit
    EXIT MENU
  END MENU
  CLOSE WINDOW w1
  CLOSE WINDOW w2
END MAIN
```

实作样式：

当点选『edit』，会切换到画面『edit』执行程序，

当点选『topmenu』，会切换到画面『topmenu』执行程序，

OPEN FORM

语法:

OPEN FORM form-id FROM “file-name”

说明:

- form-id: 定义画面的代码, 为一全域变量。
- file-name: 画面档经过编译后的文件名称 (不含附档名), 可以指定放置的相对或绝对路径。

注: FORM 在 Genero BDL 中的定义是简单的 WINDOW, 它被定义为『不需长时间出现在画面上, 用在与使用者交谈些简单的问题上』, 它与 WINDOW 功能相比, 少了切换视窗的功能等。

TIPTOP GP 多使用 WINDOW。

DISPLAY FORM 指令

语法:

DISPLAY FORM form-id

说明:

form-name 与 OPEN FORM 的 form-name 要一致。

事先要做 OPEN FORM 将画面做开启的动作, 透过 DISPLAY FORM 将画面显示在荧幕上。

CLOSE FORM 指令

语法:

CLOSE FORM form-id

说明:

FORM 使用后, 透过 CLOSE FORM 指令可将 form 关闭。

范例:

```
MAIN
  OPEN WINDOW w1 WITH 11 ROWS, 63 COLUMNS
  OPEN FORM f1 FROM "edit"      #开 form f1
  DISPLAY FORM f1
  CALL act1_a()
  CLOSE FORM f1

  OPEN FORM f2 FROM "combobox" #开 form f2
  DISPLAY FORM f2
  CALL act1_a()
  CLOSE FORM f2
  CLOSE WINDOW w1
END MAIN

FUNCTION act1_a()
  DEFINE a VARCHAR(50)
  INPUT a from formonly.a
END FUNCTION
```

实作样式:

程序执行时先开启 edit form 执行，执行完流程则关掉 edit form，后续开启 combobox form 执行程序，执行程序完再结束掉。



Chapter 6

功能表

建立功能表

功能表 MENU

Genero BDL 所提供的 MENU 命令，是一种环状显示的功能表（Ring Menu）。以下介绍如何操作并设定想要的功能。

MENU 命令

语法：

```
MENU [title]  ATTRIBUTE ( control-attributes )
```

BEFORE MENU

COMMAND option [comment]
[HELP help-number]

COMMAND KEY (key-name)
option [comment]
[HELP help-number]

ON ACTION action-name

ON IDLE idle-seconds

END MENU

CONTINUE MENU
EXIT MENU
NEXT OPTION option
SHOW OPTION {ALL|option [, ...]}
HIDE OPTION {ALL|option [, ...]}

说明：

- 1. 在一个 MENU 中，可以定义 menu 的属性， STYLE 属性预设'default'。

属性包括

| 属性 | 描述 |
|------------------|---|
| STYLE = string | 定义 menu 型态，值可以为'default', 'dialog' or 'popup' |
| COMMENT = string | 定义 menu show 出来的讯息 |
| IMAGE = string | 定义选单中的图示 |

2. BEFORE MENU 可以不写，若有一些特别的控管如隐藏某些 menu 功能或参数控管是否进入某 menu 功能时，可在 BEFORE MENU 来下一些 menu-statement 来控管 menu 部份。
3. 功能表仅为让使用者选择要执行的功能用，其他功能则不在此指令支援的范围中，例如：支援完整的 TABLE 或 SCROLLGRID 显示（使用时，若存在超过荧幕可一次显示的资料量，则只能显示前几笔，卷轴是无效的）等。

范例：

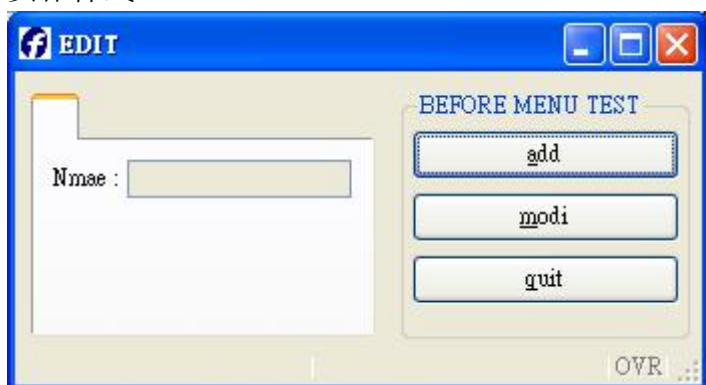
```

MAIN
  DEFINE ls_flow_pic STRING
  OPEN WINDOW act_w WITH FORM "/u1/tiptop/edit"
    ATTRIBUTE(STYLE = "tiptop.4st" )

  MENU "BEFORE MENU TEST"

    BEFORE MENU
      HIDE OPTION "del"      #隐藏 del 功能
      #SHOW OPTION "del"    #显示 del 功能
      :
      ON ACTION add
        CALL act_a()
      ON ACTION del
      :
    END MENU
  CLOSE FORM a1
  CLOSE WINDOW act_w
END MAIN
  
```

实作样式：



MENU-OPTION

包含 **COMMAND option /COMMAND key /ON ACTION /ON IDLE**

1. **COMMAND option**: 定义 MENU 功能名称及定义进入某功能快速键，透过 **COMMAND** 指令清楚掌握 **USER ACTION**。
2. **COMMAND key**: 定义 MENU 功能快速键，且不须 show 在 menu 上

范例一：

```

MAIN
:
MENU ""
  COMMAND "A.add"      #menu 功能名称"A.新增", a 为快速键
    CALL bdl1_a()
  COMMAND "U.modi"
    CALL bdl1_u()
  COMMAND "Q.qry"
    CALL bdl1_q()
  COMMAND KEY (CONTROL-A) # CONTROL-A 为快速键且
    CALL bdl1_a()          #此功能不 SHOW MENU 上
  COMMAND "exit"
    exit program
END MENU
END MAIN

```

实作样式：



3. ON ACTION: 跟 COMMAND 指令功能一样, 不同的是 COMMAND 可以重复定义 COMMAND KEY, 执行时以最后定义为主, 但是 ON ACTION 不能重复定义, compiler 会产生错误, 两者定义名称时都不分大小写

范例二:

```
MAIN
:
MENU ""
  ON ACTION add
    CALL act_a()
  ON ACTION modi
    :
  ON ACTION quit
    EXIT Program

END MENU
END MAIN
```

4. ON IDLE: 在 menu 中主要来控管停留在 menu 段的时间是否闲置太久, 透过 ON IDLE 指令时间一到, 可执行系统指定的 IDLE 处理程序 (如执行强制中断程序或显示违规警示等), 以减少可执行权 (license) 被占用或系统资料被锁定等相关问题。

范例三:

```
MAIN
:
MENU ""
  ON ACTION add
    CALL act_a()
  :
  ON IDLE 3          #闲置时间超过 3 秒处理流程
    PROMPT "IDLE 3 SECOND, Exit Menu?(Y/N) " FOR I_chr
    IF I_chr='Y' OR I_chr='y' THEN
      EXIT PROGRAM
    ELSE
      CONTINUE MENU
    END IF
  :
END MENU
END MAIN
```

實作樣式:

在等待時間到後 (ON IDLE 3 即表示等 3 秒), 系統會啟動 PROMPT。



Chapter 7

DISPLAY 与 INPUT

DISPLAY 与 INPUT

当使用者在做新增、查询、修改、删除资料时，可以透过程序中最基本的两个指令：『INPUT 和 DISPLAY』，以供使用者和程序之间的交谈。

DISPLAY 指令

此叙述主要是用来显示程序变量的值到画面的指定栏位上（注）。

语法 1:

```
DISPLAY expression [...] TO field-list [...]  
[ ATTRIBUTES ( display-attribute [...]) ]
```

语法 2:

```
DISPLAY BY NAME { variable | record.* } [...]  
[ ATTRIBUTES ( display-attribute ,... ) ]
```

说明：两者之间的差别是 DISPLAY BY NAME 变量须对应真实栏位名称，而 DISPLAY ..TO..只需对应 per 档所定义的名称即可。

ATTRIBUTES 列表

| Attribute | 描 述 |
|--|-----------|
| BLACK、BLUE、CYAN、GREEN、MAGENTA、RED、WHITE、YELLOW | 显示资料的颜色型态 |
| BOLD、DIM、NORMAL | 显示资料的字型型态 |
| REVERSE、BLINK、UNDERLINE | 显示资料的影像型态 |

注：若仅写出 DISPLAY 而未有 TO 栏位名称，则系统会将讯息显示在背景视窗中，若使用 Netterm 或 VTCP 类 telnet 软件先登入主机再执行，则此 telnet 视窗即为背景视窗；若由 Web 登入，则无背景视窗（讯息则弃置）。
上述指令通常搭配 SLEEP 指令做程序开发或除错阶段的讯息追踪。

范例一：

```
DEFINE p_employee RECORD LIKE employee_file.*  
SELECT * INTO p_employee.* FROM employee_file  
WHERE no = 'Mary'  
  
DISPLAY p_employee.* TO no, name, tel, address, salary
```

以上 DISPLAY 指令部份，因为其变量与栏位名称一致，可改用以下范例方式。

范例二：

```
DISPLAY BY NAME p_employee.*
```

INPUT 指令

在使用 INPUT 指令前必须先开启画面格式档，将画面显示在荧幕上。

语法 1: Implicit field-to-variable mapping

```
INPUT BY NAME { variable | record.* } [...]  
    [ WITHOUT DEFAULTS ]  
    [ ATTRIBUTE ( { display-attribute | control-attribute } [...]) ]  
    [ HELP help-number ]  
    [ dialog-control-block  
    [...]  
END INPUT ]
```

语法 2: Explicit field-to-variable mapping

```
INPUT { variable | record.* } [...]  
    [ WITHOUT DEFAULTS ]  
    FROM field-list  
    [ ATTRIBUTE ( { display-attribute | control-attribute }  
    [...]) ]  
    [ HELP help-number ]  
    [ dialog-control-block  
    [...]  
END INPUT ]
```

说明:

1. 当程序执行到 INPUT 指令时，会将控制权交给使用者，让使用者输入资料。
2. 使用者输入完栏位的资料，会将资料回传给程序中的变量接收。
3. 只要执行到 INPUT 的指令，程序会将每个栏位 Default 为 NULL。

INPUT 的语法有两种，如上述，而两者的差别在于若使用 INPUT BY NAME 则其变量名称必须要与栏位名称一致。

范例一：

```
INPUT p_employee.no FROM no
```

上述范例其变量与栏位名称一样，因此可改为范例二（注）。

范例二：

```
INPUT BY NAME p_employee.no
```

注：当使用 INPUT BY NAME 时，栏位的输入顺序即为 DEFINE 此组变量的元素顺序，例如：

```
DEFINE g_zx RCODRD
      zx01 LIKE zx_file.zx01,
      zx02 LIKE zx_file.zx02
END RECORD
```

则此时 INPUT BY NAME 时的输入顺序即为 zx01、zx02，设计时须注意 PER 档中输入栏位的位置不可差异太远，否则会增加使用者于资料输入时的复杂度。

INPUT 架构

INPUT *variable-list* FROM *field-list*

BEFORE INPUT

AFTER INPUT

BEFORE FIELD *field-list*

AFTER FIELD *field-list*

ON CHANGE *field-list*

ON ACTION *action-name*

ON IDLE *idle-seconds*

ACCEPT INPUT

CONTINUE INPUT

EXIT INPUT

NEXT FIELD { CURRENT | NEXT | PREVIOUS
| *field-name* }

END INPUT

控制区间执行顺序表

| 使用者动作 | 控制区间执行顺序 |
|-----------------------|---|
| 输入 filed | 1. BEFORE INPUT 2. BEFORE FIELD (first field) |
| 移动 field A to field B | 1. [ON CHANGE] (for field A, if value has changed) 2. AFTER FIELD (for field A) 3. BEFORE FIELD (for field B) |
| 在指定的 filed 改变值 | 1. ON CHANGE |
| 栏位输入完后 | 1. [ON CHANGE] (for field A, if value has changed) 2. AFTER FIELD 3. AFTER INPUT |
| 结束所有栏位输入 | 1. AFTER INPUT |

INPUT 控制点

BEFORE INPUT

当进入 INPUT 动作之前，计算机会处理 BEFORE INPUT 下的程序段。

AFTER INPUT

当使用者按了中断键或结束键时，计算机会处理 AFTER INPUT 下的程序段。

BEFORE FIELD

在进入指定的栏位动作之前，计算机会先处理 BEFORE FIELD 下的程序段。

AFTER FIELD

在输入完指定的栏位动作完毕后，计算机会处理 AFTER FIELD 下的程序段。

ON CHANGE

当栏位修改时会跑到此段，通常比较不常用到此段。

ON IDLE *idle-seconds*

设定闲置时间。

ON ACTION *action-name*

这个指令在设定功能。

NEXT FIELD *field_name*

将光标移到指定的 *field_name* 上。

EXIT INPUT

跳出 INPUT 的叙述。

END INPUT

上述任何的叙述被使用时，即需要加上 END INPUT，以表示完整的 INPUT 结束。

WITHOUT DEFAULTS

在 INPUT 指令中，我们有提到，当程序执行到 INPUT 时，计算机会预设栏位为 NULL，也就是栏位的资料值会清空。因此，若原先变量里已经放入数据库的值了，当跑到 INPUT 状态时，这些变量就变成 NULL 的状态，所以要保留原先的资料，INPUT 指令必须配合 WITHOUT DEFAULTS。

范例：

```
SELECT * INTO p_employee.* FROM employee_file
  WHERE no = 'David'
DISPLAY BY NAME p_employee.*
INPUT BY NAME p_employee.* WITHOUT DEFAULTS
```

PROMPT

若只有简单的问题要询问使用者，可考虑使用 **PROMPT** 指令，此指令只需宣告并告知所需要询问的问题，就可开出一个视窗，对使用者询问并取回答案。

语法： **PROMPT** question FOR [CHAR[ACTER]] variable
 [**ATTRIBUTE** (input-attributes)]

ON ACTION *action-name*

ON IDLE *idle-seconds*

} 仅有二控制区块，与 **INPUT** 功能相同。

END PROMPT

说明：

- **FOR CHAR**：若设定 **FOR CHAR**，则系统只抓取使用者打入系统的第一个字符。

范例：

```
MAIN
  DEFINE a CHAR(10)

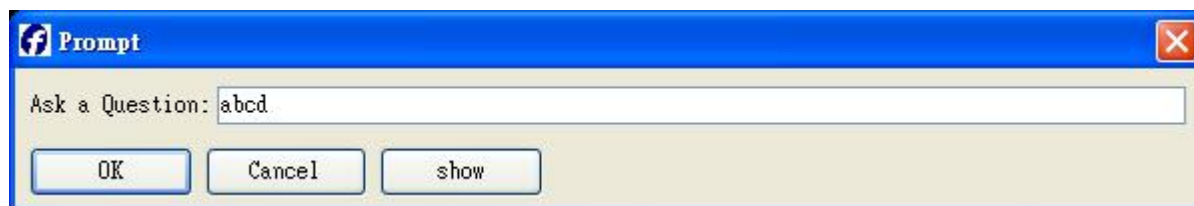
  PROMPT "Ask a Question:" FOR CHAR a

  ON ACTION show
    DISPLAY "Pressed Show!"

  ON IDLE 30
    DISPLAY "IDLE too Long!"

  END PROMPT
END MAIN
```

范例编译执行样式：





Chapter 8

CURSOR 的应用

资料的查询、修改

在 Genero BDL 语言中，当下达『DATABASE dstabase_id』指令后，即与数据库执行连结，也就是可以开始进行资料的查询（SELECT）、或更改（UPDATE）、新增（INSERT）等动作。

在处理资料时，若只有单笔资料的选取，则可用单纯的 SQL 指令即可（本书设定各位读者对 SQL 指令均已熟悉，故不再说明 SQL 指令的写法，请参阅其他 SQL 语言介绍专书）。

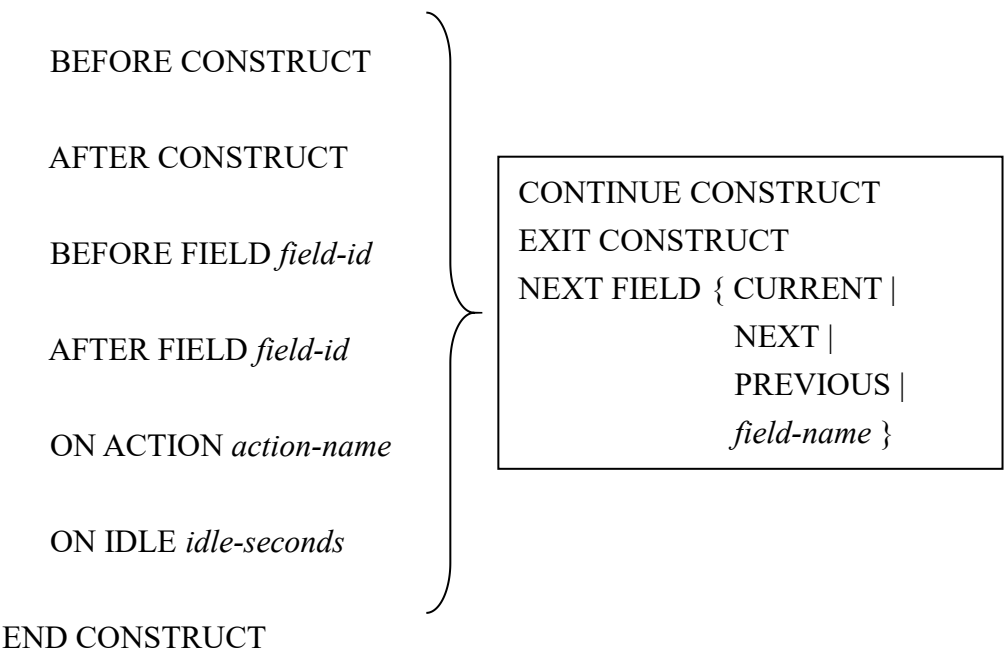
若是要抓取多笔资料处理时，就会因抓取的特性不同（例如：一次只抓一笔资料处理，完成后再抓次笔，如『个人资料表』；或一次全部抓取，一起编辑，如『个人门禁进出记录表』等），而须使用不同的指标（CURSOR）。

在介绍指标（CURSOR）前，先介绍其他将应用到的相关指令。

CONSTRUCT

此指令可让使用者在画面上输入查询条件（通称 Query By Example; QBE），以取得使用者的查询范围资料。使用者的查询资料会组成一串 WHERE 指令（参下页注解），并置入设定好的变量中。若使用者未输入任何条件，即按下『确定』离开 CONSTRUCT，系统也会自动于此变量中补入『1=1』。

语法一： CONSTRUCT *char_variable* ON *column_list* FROM *field_list*



注：INPUT 的资料输入后，其值是分散于个变量中，若要以 INPUT 指令来接取使用者所输入的查询条件，则必需以一连串复杂的字符串组合指令来组出查询指令所需要的 WHERE 条件。而 CONSTRUCT 的优点在：系统会自动判别使用者输入条件并组合，后续只要将之与 SELECT 等指令组合即可，可降低设计的复杂度。

语法二：若栏位名称和变量名称相同时，也可改采下列写法：

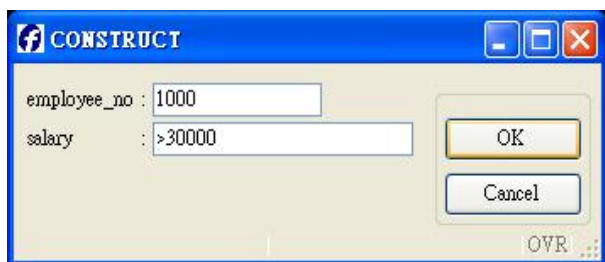
CONSTRUCT BY NAME *char_variable* ON *column_list*

说明：

- *char_variable* 为接取使用者输入资料的字符串变量（建议以 STRING 格式变量接取）
- *column_list* 为对应到表格（TABLE）的栏位名称清单（逗号隔开）
- *field_list* 为画面（WINDOW 或是 FORM）上的栏位代码清单（逗号隔开）
- 若有增加控制区段（CONTROL BLOCK，如 ON ACTION 等），则就要加上『END CONSTRUCT』

范例：

```
CONSTRUCT BY NAME l_str ON employee , salary  
  
ON IDLE 10  
EXIT PROGRAM  
  
END CONSTRUCT
```



若要查询一员工，其员工代号为『1000』，且薪水『在 30000 元以上』，则画面的员工代号栏输入『1000』，薪水栏位输入『>30000』，则 CONSTRUCT 结束后会将组合完成的 WHERE 条件置入『l_str』后回传到程序中。

l_str = 『employee='1000' AND salary>'30000'』

后续在组 SQL 指令时，即可透过此变量组出符合条件的 SQL 查询指令。

PREPARE

若已经得到一个完整的 WHERE 条件后,接下来即可将此条件,组合成 SQL 字符串,再转换为一个完整且可以抓取符合条件的 SQL 指令。

执行完 CONSTRUCT 后,系统只能得到一个『SQL 字符串(如上页范例的 l_str)』,并非为『可执行的指令』,因此必须透过 PREPARE 指令,将此 SQL 字符串转换成『可执行的 SQL 指令』PREPARE 会将字符串传入数据库查核语法的正确性,再回传予一个 prepared-id 以供后续呼叫之用。

语法:

```
PREPARE statement-name FROM char_variable
```

说明:

- *statement-name* 是为一个 PREPARE 完成后的替代代码 (prepared-id)
- 执行 PREPARE 指令前须先组好 SELECT 叙述。

FREE

释放 PREPARE 的记录。

语法:

```
FREE statement-name
```

范例:

```
LET l_str = "employee='1000' AND salary>'30000' "  
LET l_sql = " SELECT * FROM employee_file WHERE ",l_str  
PREPARE emp_pre FROM l_sql  
...  
FREE emp_pre
```

资料的查询

Genero BDL 中有两种查询用的指标（CURSOR）可以运用在资料的查询：

SCROLLING CURSOR

通常运用在单档控制或**查询类**的程序，如『个人资料表』般的作业，可以**随机**抓取资料，一次一笔，再处理完后可以选择往前一比、往后一笔或往这个查询序列中的任何一笔资料移动的指标（CURSOR）。

语法：

```
DECLARE cursor_id SCROLL CURSOR [WITH HOLD] FOR sql statement
OPEN cursor_id [USING value]
FETCH [first|last|previous|next| cursor_id INTO variable
CLOSE cursor_id
```

Non-SCROLLING CURSOR

通常运用在双档控制程序或**报表程序**，如『个人出缺勤统计表』般的作业，抓取资料是**依序（sequential）**的方式，一次可以将合条件要求的资料一笔接着一笔的抓出，直到资料全数抓完（或被强制终止）为止。

语法：

```
DECLARE cursor_id CURSOR [WITH HOLD] FOR sql statement
FOREACH [first|last|previous|next| cursor_id INTO variable
...
END FOREACH
```

SCROLLING CURSOR

在执行『个人基本资料』的查询时，大多都会希望调用出来的资料，一次只有一个人的资料，这样比较容易聚焦，待查询或编修等工作完成后，再选择上一笔（或下一笔、任何一笔）资料来继续处理。如这类的『一次仅调用单笔资料』，即为 SCROLLING CURSOR 的作业范围。组成 SCROLLING CURSOR 需以下列指令组成：

DECLARE 叙述（注）

语法（1）：使用已知的 SQL 叙述进行 CURSOR 宣告

DECLARE *cursor_id* **SCROLL CURSOR FOR** *select-statement*

语法（2）：使用已宣告的 PREPARED ID 来进行 CURSOR 宣告

DECLARE *cursor_id* **SCROLL CURSOR FOR** *prepared_id*

语法（3）：使用已知的 STRING 叙述进行 CURSOR 宣告

DECLARE *cursor_id* **SCROLL CURSOR FROM** *string-expression*

OPEN 叙述

语法：

OPEN *cursor_id*

说明：

- 本指令可用 STATUS 来检视是否执行成功。
- 在后续要使用 *cursor* 时，之前要先将 *cursor* OPEN 起来。
- 此叙述仅决定符合的资料，并不是真正从数据库中撷取资料。

注：在 DECLARE 前可用 PREPARE 指令执行 SQL 字符串的转换及检查：

- SQL 查询指令中含有问号（Question Mark，参考本章中『USING 的用法』）
- SQL 查询指令中，拥有多组的资料可能来源（*tables*），如

CASE condition

WHEN “A” LET l_sql = “SELECT count(*) FROM abc_file “

WHEN “B” LET l_sql = “SELECT count(*) FROM def_file “

END CASE

如上述情况，就必需使用 PREPARE 指令。

FETCH 叙述

语法：
FETCH *cursor_id* INTO *program_variable*

说明：当宣告为 SCROLLING CURSOR 时，可以配合以下移动 Cursor 的指令：

| 移动式 Cursor | 说明 |
|----------------|------------------------------|
| FETCH FIRST | 将 cursor 指到符合条件资料的第一笔 |
| FETCH PREVIOUS | 将 cursor 指到 current row 的前一笔 |
| FETCH NEXT | 将 cursor 指到 current row 的下一笔 |
| FETCH LAST | 将 cursor 指到符合条件资料的最后一笔 |
| FETCH ABSOLUTE | 将 cursor 移动到指定的 row |
| FETCH RELATIVE | 将 cursor 移动到目前资料的相对位置 |

CLOSE 叙述

语法：
CLOSE *cursor_id*

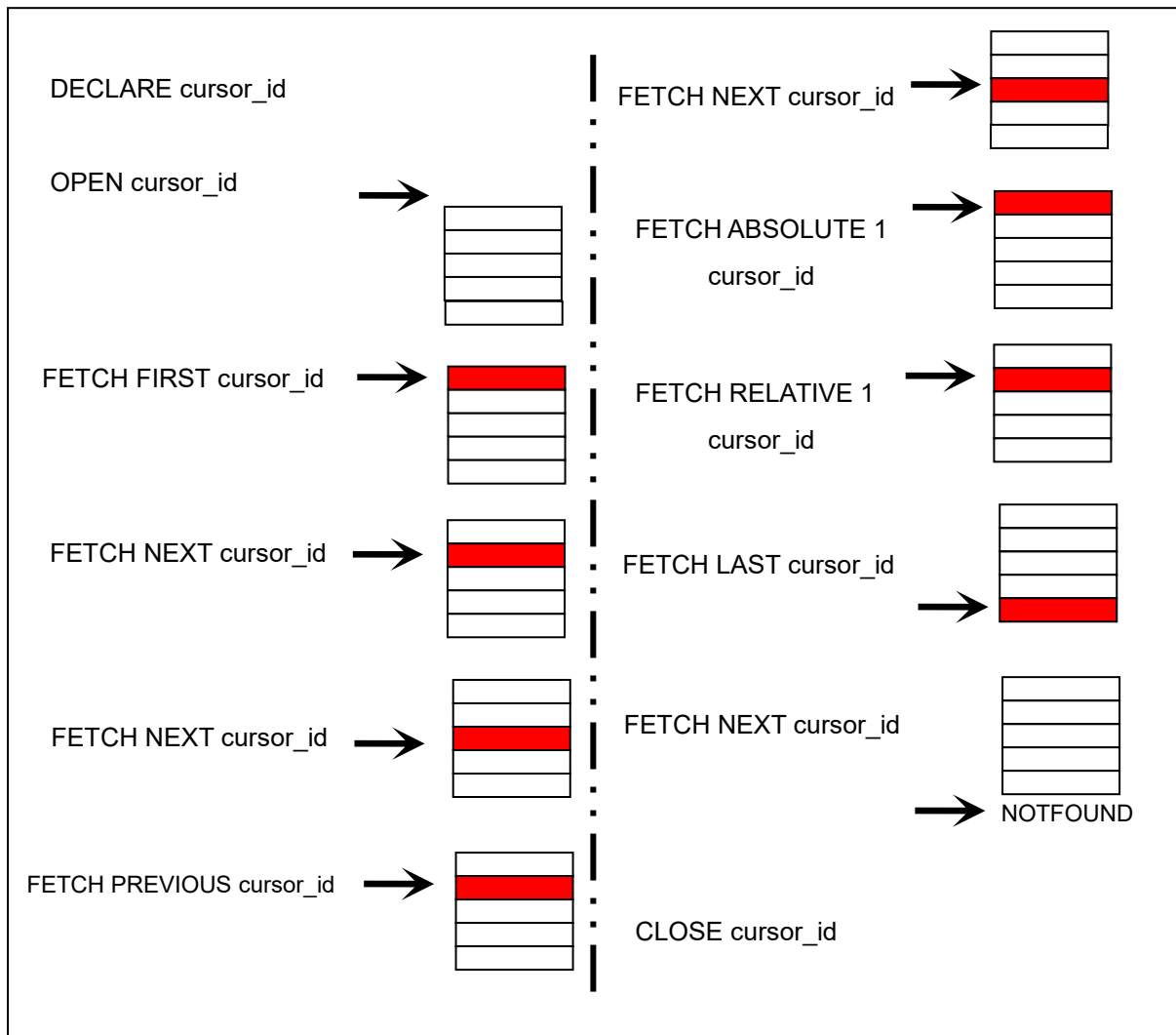
说明：关闭并释放指标（CURSOR）的储存空间。

范例：

```
DATABASE ds
MAIN
  DEFINE a  STRING
  DEFINE b,c  CHAR(10)

  DECLARE test01 SCROLL CURSOR FOR
    SELECT zz01 FROM zz_file WHERE zz01 = "axmt410"
  OPEN test01
  FETCH FIRST test01 INTO b
  DISPLAY b

  LET c = "axmt410"
  LET a = "SELECT zz01 FROM zz_file WHERE zz01=",c CLIPPED," "
  DECLARE test02 SCROLL CURSOR FROM a
  OPEN test02
  FETCH LAST test02 INTO b
  DISPLAY b
END MAIN
```



不同 FETCH 指令下 CURSOR 动作示意图

Non-SCROLLING CURSOR

如查询『个人出勤统计表』（或打印）时，就会希望系统将指定人的资料全数列示出来（因为不希望看一笔、按键、再看一笔...），如果要达到这个动作，就必须改用 Non-SCROLLING CURSOR。以下分别列出 Non-Scrolling 所需的控制指令：

DECLARE 叙述

语法（1）：使用已知的 SQL 叙述进行 CURSOR 宣告

```
DECLARE cursor_id CURSOR FOR select-statement
```

语法（2）：使用已宣告的 PREPARED ID 来进行 CURSOR 宣告

```
DECLARE cursor_id CURSOR FOR prepared_id
```

语法（3）：使用已知的 STRING 叙述进行 CURSOR 宣告

```
DECLARE cursor_id CURSOR FROM string-expression
```

说明：因为是『Non-SCROLLING CURSOR』，所以直接写『CURSOR』即可。

FOREACH（LOOP）叙述

语法：

```
FOREACH cursor_id INTO program_variable
```

```
    Statement
```

```
        :
```

```
        :
```

```
END FOREACH
```

```
    } [CONTINUE FOREACH]
    } [EXIT FOREACH]
```

说明：宣告完 CURSOR 后即可用 FOREACH 进行将所有符合条件的资料一笔一笔的抓取出来处理。

以下为 FOREACH 与 FETCH 的不同：

- FOREACH 具有循环处理的架构，而 FETCH 叙述则必须配合 WHILE 循环一起用。
- FOREACH 只能循序处理，而 FETCH 可做随机跳跃的选取。
- 执行 FETCH 指令之前必须先执行 OPEN 指令，结束时必须有 CLOSE 关闭并释放 CURSOR，但 FOREACH 指令可自动开启与关闭 CURSOR。

范例：

```
MAIN
  DEFINE clist ARRAY[200] OF RECORD
    cnum INTEGER,
    cname CHAR(50)
  END RECORD
  DEFINE i INTEGER
  DATABASE stores
  DECLARE c1 CURSOR FOR SELECT customer_num, cust_name FROM customer
  LET i=0
  FOREACH c1 INTO clist[i+1].*
    LET i=i+1
    DISPLAY clist[i].*
  END FOREACH
  DISPLAY "Number of rows found: ", i
END MAIN
```

资料的锁定

当要进行资料的修改（UPDATE）时，最令人担心的就是有多人同时修到同一笔资料，因此，如何在开始编修前进行资料锁定（LOCK），以确保同时间只有一人能取得修改权，就成了在撰写程序时应注意的事。Genero BDL 中延续 INFORMIX 4GL 的作法，采用 LOCKING CURSOR 对资料进行锁定。

LOCKING CURSOR

又称『FOR UPDATE CURSOR』。通常运用在资料更新（UPDATE）程序段，将资料进行一个上锁的动作，以避免两组以上的连线同时再更新同一 TABLE 下的同一笔资料（RECORD）。如果未作 LOCK 的动作，可能再抓取资料的同时，有其他人正在进行资料的异动。此 CURSOR 不属于资料查询的 CURSOR，而需列为更新的 CURSOR。

语法：

```
DECLARE cursor_name CURSOR FOR sql statement FOR UPDATE [NOWAIT]
OPEN cursor_id [USING value]
FETCH cursor_id INTO variable
CLOSE cursor_id
```

DECLARE 叙述

语法:

```
DECLARE cursor_name CURSOR FOR select_statement  FOR UPDATE [NOWAIT]
```

说明:

- 此处与 SCROLL CURSOR 或 Non-SCROLL CURSOR 最大的差异。在于 SQL 查询指令的最后须加上『**FOR UPDATE** (ORACLE 数据库需再加上 **NOWAIT**)』, 以标明此 CURSOR 为 LOCKING CURSOR。
- 此处亦可使用 FROM *char_variable* 方式来定义 SQL 查询指令。

OPEN 叙述

语法:

```
OPEN cursor_id
```

FETCH 叙述

语法:

```
FETCH cursor_id INTO program_variable
```

说明:

此叙述除从数据库中取得资料外, 在 LOCKING CURSOR 的状态下, 还会将所抓取到的资料锁住 (LOCK), 直到程序执行 CLOSE *cursor_id* 的指令才会释放。

CLOSE 叙述

语法:

```
CLOSE cursor_id
```

说明: 关闭并释放 CURSOR, 待释放完成后, 系统才会将被锁定的资料释放。

范例：

```
DATABASE ds
MAIN
  DEFINE g_gav01 LIKE gav_file.gav01
  DEFINE g_gav08 LIKE gav_file.gav08

  LET g_forupd_sql = "SELECT * from gav_file WHERE gav01=? AND gav08=? ",
    " FOR UPDATE "

  DECLARE p_per_lock_u CURSOR FROM g_forupd_sql
  OPEN p_per_lock_u USING g_gav01,g_gav08
  OPEN p_per_lock_u USING g_gav01,g_gav08
  IF STATUS THEN
    CLOSE p_per_lock_u
    RETURN
  END IF
  FETCH p_per_lock_u INTO g_gav_lock.*
  IF SQLCA.sqlcode THEN
    CLOSE p_per_lock_u
    RETURN
  END IF
  CLOSE p_per_lock_u
END MAIN
```

USING 的使用时机

此处的『USING』和先前谈变量格式输出的 USING 有不同的意义。

有时，在组合 SQL 查询指令时，可能尚未取得相关的 KEY 值供选取资料用，又不可以随意填不相关的值进入系统。

如上方范例：在组查询的 SQL 指令时，根本无法确知将要锁定的资料为何，因此『可以在 SQL 查询指令中使用问号（?），待后续要使用此 CURSOR 时，再将已知值用 USING 传入』。

语法：

```
LET sql statement = "SELECT * FROM table_id WHERE key_value = ? "
DECLARE cursor_name CURSOR FOR sql statement FOR UPDATE [NOWAIT]
OPEN cursor_id USING value
```

说明：

- 问号（?）可以有多个。
- USING 必需跟在 OPEN 后方，若有多个问号，则此处需依序对映，并以逗号隔开。

TRANSACTION

当程序中执行『UPDATE』、『INSERT』、『DELETE』等指令时，都直接对数据库的资料进行异动，若有时资料需要多表格的同时连动时，会期望『所有的动作一起写入或一起毁弃』时，可采用『TRANSACTION』作法。

在一个 TRANSACTION 开始后(以『BEGIN WORK』指令开始)，所有的『UPDATE』、『INSERT』、『DELETE』指令均不会对实体数据库做动作，一直到结束后，依照不同的条件进行写入实体数据库(以『COMMIT WORK』指令做整批写入)、或执行变更毁弃(以『ROLLBACK WORK』指令进行)的动作。

语法范例：

BEGIN WORK

[UPDATE *statement*]

[INSERT *statement*]

[DELETE *statement*]

IF (*condition*) THEN

COMMIT WORK

ELSE

ROLLBACK WORK

END IF

说明：

- BEGIN WORK 开始后，一定要有 COMMIT WORK 或 ROLLBACK WORK 做资料是否写入的判断。
- TRANSACTION 区中的程序尽量不要太长，以免影响需要调用同笔资料的其他使用者（但使用者本身不受影响，未 COMMIT WORK 前仍可调用已变更资料）。
- 有些 DDL 指令（如 CREATE TABLE）会有自动 COMMIT WORK 功能。
- 当 COMMIT WORK（或 ROLLBACK WORK）时，会自动关闭无宣告『WITH HOLD』的 CURSOR。

大量执行同一 SQL 指令（EXECUTE）

当系统要执行大量的同一 SQL 指令（例如：要在工作资料中连续 INSERT 十年的工作日资料），系统执行效能就会变成一个令人关心的议题。如塞入工作日资料，当然可以写做：

```
INSERT INTO work_date VALUES ( "2005/07/01" , "Friday", "Weekday")
```

然后再以一个 FOR LOOP 包起，这个作法一样可以执行，但效能还有可以改进的空间。程序的撰写若有强烈的效能考量，则可以考虑改用『EXECUTE』作法。

语法：

```
PREPARE prepared_id FROM sql_statement  
EXECUTE prepared_id [USING variable_list] [ INTO fvar [...] ]  
FREE prepared_id
```

说明：

- EXECUTE 可以应用在『SELECT』、『UPDATE』、『INSERT』、『DELETE』等处。
- 使用 EXECUTE 时，尽量将 KEY 值或其他需要变异的值，保留至 EXECUTE 指令时再以 USING 方式传入。

范例：

```
FUNCTION update_customer_name( key, name )  
  DEFINE key INTEGER  
  DEFINE name CHAR(10)  
  PREPARE s1 FROM "UPDATE customer SET name=? WHERE customer_num=?"  
  EXECUTE s1 USING name, key  
  FREE s1  
END FUNCTION
```

大量执行新增指令（PUT..FLUSH）

以上页塞入工作日资料范例来谈，除改用 EXECUTE 外还可以有更快速的作法，就是用『INSERT CURSOR』。Genero BDL 提供一 INSERT CURSOR: 『PUT...FLUSH』，范例如下所示：

语法：

```
PREPARE prepared_id FROM INSERT_sql_statement
DECLARE cursor_id [WITH HOLD] FOR prepared_id
OPEN cursor_id
PUT cursor_id FROM variable_list
FLUSH cursor_id
CLOSE cursor_id
FREE cursor_id
FREE prepared_id
```

范例：

```
MAIN
  DEFINE i INTEGER
  DEFINE rec RECORD
    key INTEGER,
    name CHAR(30)
  END RECORD
  DATABASE stock
  PREPARE is FROM "INSERT INTO item VALUES (?,?)"
  DECLARE ic CURSOR FOR is
  BEGIN WORK
    OPEN ic
    FOR i=1 TO 100
      LET rec.key = i
      LET rec.name = "Item #" || i
      PUT ic FROM rec.*
      IF i MOD 50 = 0 THEN
        FLUSH ic
      END IF
    END FOR
    CLOSE ic
  COMMIT WORK
  FREE ic
  FREE is
END MAIN
```




Chapter 9

ARRAY 的应用

ARRAY（阵列）

阵列可以依一维，二维或三维阵列方式来储存资料

固定阵列（Static Array）定义

ARRAY [*intconst* [, *intconst* [, *intconst*]]] OF *datatype*

动态阵列（Dynamic Array）定义

DYNAMIC ARRAY [WITH DIMENSION *rank*] OF *datatype*

物件

| 物件名称 | 说明 |
|-----------------------------------|---|
| getLength() RETURNING INTEGER | 回传单层阵列的长度 |
| clear() | 将动态阵列（Dynamic Array）中，所有记录移除。 将固定阵列（Static Array）中，所有纪录值清为 NULL。 |
| appendElement() | 在动态阵列（Dynamic Array）后面加上一笔新的记录。 这个物件在固定阵列（Static Array）中无效。 |
| insertElement(INTEGER) | 在指定位置新增记录，并将指定位置后之资料往下移。 动态阵列（Dynamic Array）的笔数加 1。 |
| deleteElement(INTEGER) | 移除指定位置记录，并将指定位置后之资料往上移。 动态阵列（Dynamic Array）的笔数减 1。 |

范例一：定义固定及动态阵列

```
MAIN
  DEFINE a1 ARRAY[100] OF INTEGER
  DEFINE a2 DYNAMIC ARRAY OF INTEGER
  DEFINE i INTEGER
  LET i = 12
  LET a1[50] = 12456
  LET a2[5000] = 12456
  LET a2[500+i] = 12456
END MAIN
```

范例二：getLength()

```
MAIN
  DEFINE a DYNAMIC ARRAY OF INTEGER
  LET a[5000] = 12456
  DISPLAY a.getLength()
END MAIN
```

-- Displays 5000

范例三：clear()

```
MAIN
  DEFINE a DYNAMIC ARRAY OF INTEGER
  LET a[10] = 11
  DISPLAY a.getLength()
  CALL a.clear()
  DISPLAY a.getLength()
END MAIN
```

-- Displays 10
-- Displays 0

范例四：appendElement()

```
MAIN
  DEFINE a DYNAMIC ARRAY OF INTEGER
  LET a[10] = 10
  CALL a.appendElement()
  LET a[a.getLength()] = a.getLength()
  DISPLAY a.getLength()
  DISPLAY a[10]
  DISPLAY a[11]
END MAIN
```

-- Displays 11
-- Displays 10
-- Displays 11

范例五: insertElement()

```
MAIN
  DEFINE a DYNAMIC ARRAY OF INTEGER
  LET a[10] = 11
  CALL a.insertElement(10)
  LET a[10] = 10
  DISPLAY a.getLength()
  DISPLAY a[10]
  DISPLAY a[11]
END MAIN
```

-- Displays 11
-- Displays 10
-- Displays 11

范例六: deleteElement()

```
MAIN
  DEFINE a DYNAMIC ARRAY OF INTEGER
  LET a[10] = 9
  DISPLAY a.getLength()
  CALL a.deleteElement(5)
  DISPLAY a.getLength()
  DISPLAY a[9]
END MAIN
```

-- Displays 10
-- Displays 9
-- Displays 9

范例七: 二、三维阵列

```
MAIN
  DEFINE a2 DYNAMIC ARRAY WITH DIMENSION 2 OF INTEGER
  DEFINE a3 DYNAMIC ARRAY WITH DIMENSION 3 OF INTEGER
  LET a2[50,100] = 12456
  LET a2[51,1000] = 12456
  DISPLAY a2.getLength()
  DISPLAY a2[50].getLength()
  DISPLAY a2[51].getLength()
  LET a3[50,100,100] = 12456
  LET a3[51,101,1000] = 12456
  DISPLAY a3.getLength()
  DISPLAY a3[50].getLength()
  DISPLAY a3[51].getLength()
  DISPLAY a3[50,100].getLength()
  DISPLAY a3[51,101].getLength()
  CALL a3[50].insertElement(10)
  CALL a3[50,10].insertElement(1)
END MAIN
```

-- Displays 51
-- Displays 100
-- Displays 1000
-- Displays 51
-- Displays 100
-- Displays 101
-- Displays 100
-- Displays 1000
-- Inserts at 50,10
-- Inserts at 50,10,1


INPUT ARRAY

Input Array 可以让使用者透过 Screen Record 输入资料

语法:

```
INPUT ARRAY array [ WITHOUT DEFAULTS ] FROM screen-array.*  
[ HELP help-number ]  
[ ATTRIBUTE ( { display-attribute | control-attribute }  
[...]) ]
```

```
BEFORE INPUT  
AFTER INPUT  
AFTER DELETE  
BEFORE ROW  
AFTER ROW  
BEFORE FIELD field-list  
AFTER FIELD field-list  
ON ROW CHANGE  
ON CHANGE field-list  
ON IDLE idle-seconds  
ON ACTION action-name  
BEFORE INSERT  
    CANCEL INSERT  
AFTER INSERT  
    CANCEL INSERT  
BEFORE DELETE  
    CANCEL DELETE  
END INPUT
```



```
ACCEPT INPUT  
CONTINUE INPUT  
EXIT INPUT  
NEXT FIELD { CURRENT |  
             NEXT |  
             PREVIOUS |  
             field-name }
```

显示属性

| 属性名称 | 说明 |
|--|-----------|
| BLACK、BLUE、CYAN、GREEN、MAGENTA、RED、WHITE、YELLOW | 显示资料的颜色型态 |
| BOLD、DIM、INVISIBLE、NORMAL | 显示资料的字型型态 |
| REVERSE、BLINK、UNDERLINE | 显示资料的影像型态 |

控制属性

| 属性名称 | 说明 |
|------------------------------------|------------------|
| APPEND ROW [= <i>bool</i>] | 定义使用者可否在最后一列新增资料 |
| COUNT = <i>row-count</i> | 定义固定阵列的总列数 |
| DELETE ROW [= <i>bool</i>] | 定义使用者可否删除资料 |
| INSERT ROW [= <i>bool</i>] | 定义使用者可否新增资料 |
| MAXCOUNT = <i>row-count</i> | 定义阵列最大资料列数 |
| UNBUFFERED [= <i>bool</i>] | 定义阵列及时显示资料 |
| WITHOUT DEFAULTS [= <i>bool</i>] | 保留原先资料，不清为 NULL |
| CANCEL = <i>bool</i> | 定义”CANCEL”功能可否使用 |
| ACCEPT = <i>bool</i> | 定义”ACCEPT”功能可否使用 |

控制段执行顺序

| 使用者动作 | 控制段执行顺序 |
|------------------|--|
| 进入阵列 | BEFORE INPUT BEFORE ROW BEFORE FIELD |
| 移动至其他列 | AFTER FIELD (for field A in the current row) [AFTER INSERT] (if a new row was inserted or new row was appended and modified) [ON ROW CHANGE] (if values have changed in current row) AFTER ROW (for the current) BEFORE ROW (the new row) BEFORE FIELD (for field B in the new row) |
| 移动至同列中的不同栏位 | [ON CHANGE] (for field A, if value has changed) AFTER FIELD (for field A) BEFORE FIELD (for field B) |
| 删除一行 | BEFORE DELETE (the row to be deleted) AFTER DELETE (the deleted row) AFTER ROW (the deleted row) BEFORE ROW (the new current row) BEFORE FIELD |
| 新增一行 (插在两列中) | AFTER FIELD [AFTER INSERT] (if a new row was created) [ON ROW CHANGE] (if values have changed) AFTER ROW (the previous row) BEFORE INSERT BEFORE FIELD |
| 新增一行 (进入最后一列) | AFTER FIELD [AFTER INSERT] (if a new row was created) [ON ROW CHANGE] (if values have changed) AFTER ROW (the previous row) BEFORE ROW (the new current row) BEFORE INSERT BEFORE FIELD |
| 按下确定 | [ON CHANGE] AFTER FIELD [AFTER INSERT] (if a new row was created) [ON ROW CHANGE] (if values have changed) AFTER ROW AFTER INPUT |
| 按下取消 | AFTER ROW AFTER INPUT |

范例一：Form

```

DATABASE ds

LAYOUT
  TABLE
    {
      id      Name      department
      [gen01  ][gen02    ][gen03      ]
      [gen01  ][gen02    ][gen03      ]
      [gen01  ][gen02    ][gen03      ]
      [gen01  ][gen02    ][gen03      ]
      [gen01  ][gen02    ][gen03      ]
      [gen01  ][gen02    ][gen03      ]
    }
  END
END

TABLES gen_file

ATTRIBUTES
  gen01 = gen01;
  gen02 = gen02;
  gen03 = gen03;
END

INSTRUCTIONS
  SCREEN RECORD sr_cust[6](gen01,gen02,gne03);
END

```

范例二：INPUT ARRAY

MAIN

```
DEFINE custarr ARRAY[500] OF RECORD
```

```
    gen01 CHAR(8),
```

```
    gen02 CHAR(8),
```

```
    gen03 CHAR(6)
```

```
END RECORD
```

```
DEFINE counter INTEGER
```

```
DEFER INTERRUPT
```

```
DEFER QUIT
```

```
OPEN WINDOW w1 WITH FORM "FormFile"
```

```
LET INT_FLAG = FALSE
```

```
INPUT ARRAY custarr WITHOUT DEFAULTS FROM sr_cust.*
```

```
IF INT_FLAG = FALSE THEN
```

```
    FOR counter = 1 TO arr_count()
```

```
        DISPLAY custarr[counter].*
```

```
    END FOR
```

```
END IF
```

```
CLOSE WINDOW w1
```

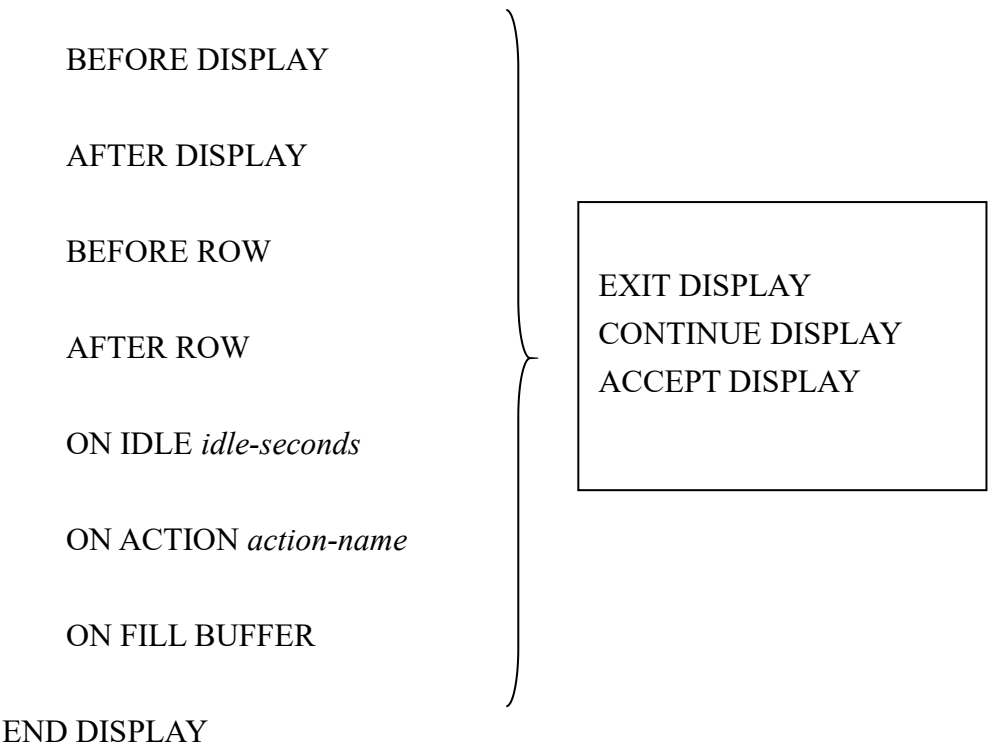
```
END MAIN
```

DISPLAY ARRAY

将程序阵列的值显示在画面上

语法:

```
DISPLAY ARRAY record-array TO screen-array.* [ HELP help-number ]  
[ ATTRIBUTE ( { display-attribute | control-attribute } [,...] ) ]
```



控制属性

| 属性名称 | 说明 |
|-----------------------------------|-------------------|
| COUNT = <i>row-count</i> | 定义固定阵列的总列数 |
| UNBUFFERED [= <i>bool</i>] | 定义阵列及时显示资料 |
| WITHOUT DEFAULTS [= <i>bool</i>] | 保留原先资料，不清为 NULL |
| CANCEL = <i>bool</i> | 定义”CANCEL”功能可否使用 |
| ACCEPT = <i>bool</i> | 定义” ACCEPT”功能可否使用 |

控制段执行顺序

| 使用者动作 | 控制段执行顺序 |
|--------|---|
| 进入阵列 | BEFORE DISPLAY BEFORE ROW |
| 移动至不同列 | AFTER ROW (the current row) BEFORE ROW (the new row) |
| 执行确定功能 | AFTER ROW AFTER DISPLAY |
| 执行取消功能 | AFTER ROW AFTER INPUT |

范例一：Form

```

DATABASE ds

LAYOUT
  TABLE
  {
    id      Name      department
    [gen01  ][gen02    ][gen03      ]
    [gen01  ][gen02    ][gen03      ]
    [gen01  ][gen02    ][gen03      ]
    [gen01  ][gen02    ][gen03      ]
    [gen01  ][gen02    ][gen03      ]
    [gen01  ][gen02    ][gen03      ]
  }
  END
END

TABLES gen_file

ATTRIBUTES
  gen01 = gen01;
  gen02 = gen02;
  gen03 = gen03;
END

INSTRUCTIONS
  SCREEN RECORD sr_cust[6](gen01,gen02,gen03);
END

```

范例二：DISPLAY ARRAY

```
DATABASE ds
MAIN
  DEFINE cnt INTEGER
  DEFINE arr DYNAMIC ARRAY OF RECORD
    gen01 CHAR(8),
    gen02 CHAR(8),
    gen03 CHAR(6)
  END RECORD

  OPEN WINDOW f1_w AT 3,3 WITH FORM "custlist"
  DECLARE c1 CURSOR FOR
    SELECT gen01,gen02,gen03 FROM gen_file
  LET cnt = 1
  FOREACH c1 INTO arr[cnt].*
    LET cnt = cnt + 1
  END FOREACH

  CALL arr.deleteElement(cnt)
  LET cnt = cnt - 1

  DISPLAY ARRAY arr TO srec.* ATTRIBUTES(COUNT=cnt)
  ON ACTION print
    DISPLAY "Print a report"
  END DISPLAY
END MAIN
```




Chapter 1 0

REPORT 撰写

REPORT 撰写

BDL 要印制出一份报表，程序分成二个部份：

1. 主程序中先驱动 REPORT DRIVER，并且撷取报表所需报表的资料。
2. 取得资料后进入 REPORT FUNCTION，做报表格式的设定，例如报表的版面设定、以及表头、表身、表尾如何安排。

FUNCTION 中启动 REPORT

START REPORT

语法：START REPORT *rep_name* [TO { SCREEN | FILE *filename* | PRINTER }]

说明：

1. 这个指令是在驱动 REPORT DRIVER。
2. TO SCREEN：为系统 DEFAULT 可不写。
3. TO FILE *filename*：将 REPORT 的结果送到一个档案中(档名自订)。
4. TO PRINTER：将 REPORT 结果送到打印机(系统打印机)。
5. 若(2)、(3)都不写，则系统会将结果自动送到荧幕输出。

OUTPUT TO REPORT 叙述

语法：OUTPUT TO REPORT *rep_name*(*expr_list*)

说明：该功能类似 CALL function，并传递参数，将资料送到 REPORT FUNCTION。

FINISH REPORT 叙述

语法：

FINISH REPORT *rep_name*

说明：此指令为结束报表印制的指令。

4GL 报表范例:

```
FUNCTION sel_employee( )
  DEFINE p_employee RECORD LIKE employee_file.*

  DECLARE emp_cs CURSOR FOR
    SELECT * FROM employee_file
      :
  START REPORT emp_rep
      :
  FOREACH emp_cs INTO p_employee.*
    OUTPUT TO REPORT emp_rep(p_employee.*)
  END FOREACH
      :
  FINISH REPORT emp_rep

END FUNCTION

REPORT emp_rep(s_employee)
  DEFINE s_employee RECORD LIKE employee_file.*
  FORMAT
      :
      :
END REPORT
```

REPORT FUNCTION 的组成

```
REPORT report_name(expr_list)  
  [DEFINE define_statement]  
  
  [OUTPUT output_statement]  
  
  [ORDER BY sort_list]  
  
  FORMAT  
    control_block  
    statement  
    :  
    :  
END REPORT
```

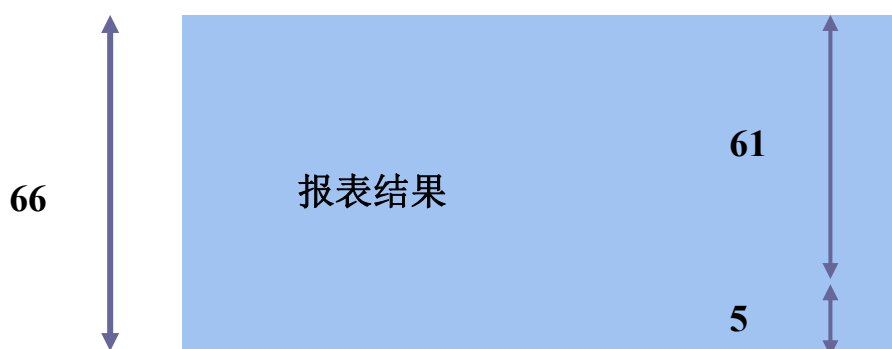
REPORT *report_name* (*expr_list*) 叙述

report_name 即是 REPORT 的名称，与 START REPORT *rep_name* 的 *rep_name* 是一致的。*expr_list* 为先前主程序所传递的参数。

OUTPUT Section

此叙述为定义报表的边界，报表长度。以下介绍设定报表边界的指令，及若没有 OUPUT 叙述时，系统的预设值。

| 指令 | 预设值 |
|------------------------|----------|
| LEFT MARGIN 0 | 5 spaces |
| TOP MARGIN 0 | 3 lines |
| BOTTOM MARGIN 5 | 3 lines |
| PAGE LENGTH 66 | 66 lines |



ORDER BY Section

语法:

ORDER [**EXTERNAL**] **BY** *field_id_1* [**ASC**|**DESC**] , *field_id_2* [**ASC**|**DESC**] , ...

说明:

1. **ORDER BY** 区间，主要是作排序栏位用，以逗号分开，摆在最前面的栏位为主键，同时只能用所接收的参数内的栏位来排序。
2. **ORDER BY** 会先将资料排序过并存在暂存档中，配合 **BEFORE GROUP OF** 或 **AFTER GROUP OF** 区段排序后再由暂存档印出。
3. 加上 **EXTERNAL** 即表示传入的资料已经经过外部（**START REPORT** 时）的排序，因此此处仅为注标，系统不会重复的执行工作。
4. 若后续于 **FORMAT** 中有使用 **GROUP** 的分群输出功能，则一定要有 **ORDER BY** 宣告，必要时可搭配上述 **EXTRRNAL** 功能。

FORMAT Section

这个部份为 REPORT FUNCTION 的重心，所以报表的排版均由此部份控制。以下说明此 SECTION 的每个 CONTROL BLOCKS 用途。

FIRST PAGE HEADER : 报表第一页的表头控制段。

PAGE HEADER : 报表每一页的表头控制段。

BEFORE GROUP OF : 设定在一组资料的开始之前所必须执行的叙述。

ON EVERY ROW : 指定每一笔记录的输出格式。

AFTER GROUP OF : 在控制区段设定一组资料之后必须执行的叙述。

PAGE TRAILER : 每一页报表页尾的控制段。

ON LAST ROW : 所有资料印完后要做的动作，例如”总计”。

FORMAT 内可用指令

- **PRINT**

1. 打印讯息或资料。
2. 可用逗号【,】分隔要打印的栏位。
3. 一个 PRINT 指令即占用一行的打印空间。
4. 若以分号【;】结尾，表示下一个 PRINT 指令亦接续本行打印。

范例：

PRINT “员工编号：”, employee_no

- **SKIP**

1. 打印时要跳几行或几页。
2. 若在打印报表中有使用 IF、CASE 等流程判断指令，需确保每个流程的每一个情境下，所用的报表打印空间（行数）是一致的。

范例：

SKIP 2 LINES →跳两行

SKIP TO TOP OF PAGE→跳页

- NEED

1. NEED *integer_value* LINES
2. 在打印前，先确定是否有足够的剩余行数，可供接下来打印，如果不够将先跳页后，再继续往下印。

打印时可运用的表达式或函数

CLIPPED

将字符串后面的空白清掉。

USING

针对数值定其打印的格式。

COLUMN

指定输出的行位置。

LINENO

取得目前打印行的列号值。

PAGENO

取得目前的页数。

SPACES

传回空白。

TIME

传回系统时间，格式:”hh:mm:ss”

TODAY

传回系统今天的日期。

LENGTH(*expr*)

expr 为一字符串变量，其会传回 *expr* 的长度。

群组函数

在群组区块中，提供以下函数，方便在撰写报表时的数值运算。

GROUP SUM(*expression*) :GROUP 区块中，数值的加总。

GROUP COUNT(*) :GROUP 区块中，资料的总数。

GROUP MIN(*expression*) :GROUP 区块中，取得数值的最小值。

GROUP MAX(*expression*) :GROUP 区块中，取得数值的最大值。

GROUP AVG(*expression*) :GROUP 区块中，取得数值的平均值。

GROUP PERCENT(*) :GROUP 区块中，资料的百分比。



Chapter 1 1

Debugger 的应用

Debugger 的使用

以除错模式执行程序

一般执行程序的命令为 `fglrun 程序名称`
以除错模式执行程序的命令为 `fglrun -d 程序名称`

除错模式的在线说明

除错命令一览表 `help`
单一除错命令完整说明 `help command`

除错模式的常用指令

| 指令 | 说明 | 范例 | 缩写 |
|----------|----------------|-----------------------------|-----------------|
| Break | 设定断点 | <code>break test01_a</code> | <code>b</code> |
| Run | 执行程序 | <code>run</code> | <code>ru</code> |
| Step | 逐步执行程序 | <code>step 10</code> | <code>s</code> |
| Continue | 执行到下一个断点 | <code>continue</code> | <code>co</code> |
| List | 列出程序码 line | <code>list</code> | <code>l</code> |
| Print | 显示变量值 | <code>print g_sql</code> | <code>p</code> |
| Quit | 结束 Debugger 环境 | <code>quit</code> | <code>q</code> |
| Delete | 删除断点 | <code>delete 1</code> | <code>d</code> |
| Help | 查看指令内容 | <code>help list</code> | <code>h</code> |

Break 的用法介绍

1. 依行数中断
ex: break 315
2. 依函式中
ex: break test01_a
3. 依状态中断
ex: break if status < 0

Step 的用法介绍

1. 指定步数执行
ex: step 500
2. 进入函式
ex: step into

List 的用法介绍

1. 列出程序第 1 行之后程序码
ex: list 1
2. 列出程序 function 段程序码
ex: list test01_a

Print 的用法介绍

1. 直接显示
ex: p g_cnt

Help 的用法介绍

1. 直接显示所有 debugger 指令
ex: help
2. 直接显示所指定 debugger 指令内容
ex: help step



附录 C

相关参考信息

参考资料

目前在市面上流通的 Genero BDL 资料可以说是付之阙如,因此 Genero BDL 语法的最大参考来源即是 FourJS 公司的官方网页『[http:// www.4js.com](http://www.4js.com)』,或是『Genero BDL 光盘片』中的 ON-Line Help 文件。

除了 Genero BDL 文件外,也可参阅相关的 INFORMIX 文件,但是在阅读时就要小心比较其差异性。目前流通的繁体中文版 INFORMIX 文件有:

INFORMIX 4GL 学习手册(附 CD); 陶淑瑗着: 儒林;
2004 年 10 月出版; 繁体中文



附录 D

上课地点位置图



台北总公司位置路线图



台北总公司外围停车信息图



台中分公司位置路线图