

# COMX 平台开发手册

第 0.0.5.0 版 2014 年 10 月 28 日



作者：张向奎

大连理工大学汽车工程学院  
School of AutomotiveEngineering,  
Dalian University of Technology

## 变更履历

区分：A=追加 / U=更新 / D=删除

# 目 录

<b>Part-I 基础篇.....</b>	<b>9</b>
<b>第 1 章. 开发环境建立 .....</b>	<b>10</b>
1-1. COMX_SDK 的安装及配置 .....	10
1-1.1. C++开发环境: VS2005 SP1 .....	10
1-1.2. COMX_SDK 开发包安装和配置 .....	10
1-2. 常用命令行开发工具.....	11
<b>第 2 章. KUL&amp;JavaScript.....</b>	<b>12</b>
2-1. 一个简单的 KUL 程序实例 .....	12
2-2. Unit 创建与维护 .....	14
2-3. 布局器 .....	15
2-4. 控件 Binding 机制 .....	16
2-5. 基于 JavaScript 的消息响应函数.....	17
2-6. KUL 中 WIDGET 的属性定义 .....	18
2-7. 系统函数和在线帮助.....	18
2-8. KUL 文件的注册 .....	21
2-9. 文件、变量作用域和全局变量.....	23
2-10. COMX KUL&JS Design 工具.....	24
2-11. COMX UI 的 block 重用机制 .....	24
2-12. Javascript 和 Node.JS 环境下一种典型的模块化方法 .....	27
2-13. Node.JS 简介 .....	27
2-13.1. 什么是 Node.JS? .....	28
2-13.2. NPM 扩展 .....	28
2-14. 模块化和扩展机制 .....	28
2-15. JS.EXT 扩展生成和使用.....	29
2-16. COMX 本地进程调用 API.....	31
2-17. Websocket API .....	32
2-18. 基于 Node.JS 的异步后台进程调用 .....	35
2-18.1. Node.JS 的 child_process 使用举例.....	35
2-18.2. boost 命令行工具 .....	37
2-18.3. jsoncpp 引擎.....	39
2-19. COMX 下的 Websocket 协同架构.....	43
2-20. “沙箱”机制 .....	45
2-20.1. 什么是“沙箱（Sandbox）”? .....	45
2-20.2. COMX 中的“沙箱”的含义 .....	45
2-20.3. COMX 中如何使用“沙箱”? .....	46
2-21. 定时器操作 .....	48
2-22. 组件和接口在 JavaScript 环境下的使用 .....	48
2-23. 基于 KUL&JavaScript 的插件 package 的生成和使用.....	48
2-24. COMX 中的 Distribute 工具 .....	49
2-25. 在那里可以找到 KUL&JS 程序相关实例? .....	50
2-26. 关于图标和图片 .....	50

<b>第 3 章. COMX 组件使用简化教程 .....</b>	<b>51</b>
3-1. COMX 是什么? .....	51
3-2. 组件、接口、出接口.....	51
3-3. 组件的类型和生存周期.....	53
3-4. 类工厂 .....	53
3-5. 生成组件代码框架的工具.....	55
3-6. 组件的协作和拼接.....	57
3-7. 组件的配置脚本 .....	57
3-8. 系统预定义组件集简介.....	59
<b>第 4 章. 开发实例.....</b>	<b>60</b>
4-1. 向导 GUI 程序 .....	60
4-2. Edit Widget 的有效性检查.....	69
4-3. List 控件的插入删除操作 .....	70
4-4. 基于 Bitmap 的 gif 播放和等待进度条 .....	73
4-4.1. 基于图片模式的等待进度条播放 .....	73
4-4.2. 基于 gif 模式的大尺寸动画播放 .....	74
4-5. Scintilla 编辑控件 .....	75
<b>Part-II 进阶篇.....</b>	<b>76</b>
<b>第 5 章. COMX 基础.....</b>	<b>77</b>
5-1. 什么是 COMX? .....	77
5-2. COMX 平台的特点 .....	77
5-3. COMX 平台的体系结构 .....	78
5-4. COMX 中的基本概念 .....	78
5-5. COMX 中的软件工程原则 .....	94
5-6. 一组推荐的命名规则.....	96
<b>第 6 章. 内核和插件 .....</b>	<b>97</b>
6-1. “微内核+插件”机制概述.....	97
6-2. 主程序和内核组件.....	98
6-3. XML 配置脚本和软件包的组织形式.....	101
6-4. 工具条和对话框界面插件的开发.....	102
6-5. “事件”(“出接口”)机制 .....	115
6-6. 一些值得特别关注的基础性组件 .....	116
<b>附录一 代码书写规范 .....</b>	<b>118</b>
前言 .....	120
语法高亮与字体 .....	120
字体 .....	120
语法高亮 .....	121
文件结构 .....	122
文件头注释 .....	122
头文件 .....	122
内联函数定义文件 .....	123

---

实现文件.....	123
<b>命名规则 .....</b>	<b>124</b>
类/结构 .....	124
函数.....	125
变量.....	125
常量.....	127
枚举、联合、typedef .....	127
宏、枚举值.....	127
名空间.....	127
<b>代码风格与版式 .....</b>	<b>128</b>
类/结构 .....	131
函数.....	136
变量、常量.....	141
枚举、联合、typedef .....	144
宏.....	145
名空间.....	145
异常.....	146
<b>代码的添加/更新/删除.....</b>	<b>155</b>
<b>版本控制 .....</b>	<b>156</b>
<b>关于本规范的贯彻实施 .....</b>	<b>156</b>
<b>术语表 .....</b>	<b>157</b>
<b>附录二 <i>example.cpp</i> 文件.....</b>	<b>158</b>

## 表目录

表 1. 1 COMX SDK 常用命令行工具.....	11
表 2. 1 Binding 使用说明 .....	16
表 2. 2 javascript 消息响应函数 .....	17
表 2. 3 客户端的 Websocket API 接口 .....	33
表 2. 4 服务器端的 Websocket API 接口 .....	34
表 2. 5 websocket 中的缓冲区处理函数.....	42
表 5. 1 COMX 内核中的关键字列表.....	78
表 5. 2 IEventSource 接口函数列表 .....	86
表 5. 3 IEventSourceContainer 接口函数列表 .....	86
表 5. 4 用于维护组件出接口动态关联的函数列表.....	88
表 5. 5 COMX 组件所需支持的入口函数列表 .....	89
表 5. 6 命名规则.....	96
表 6. 1 IComxUiPluginServerFrm 的接口函数列表 .....	99
表 6. 2 IUiProgressBarDriver 的接口函数列表.....	99
表 6. 3 IUiToolbarStatusDriver 的接口函数列表 .....	99
表 6. 4 IFactory2 的接口函数列表 .....	99
表 6. 5 IFactory 的接口函数列表 .....	99
表 6. 6 IFactoryGlobalSetting 的接口函数列表 .....	100
表 6. 7 IComxUiPluginServer 的接口函数列表 .....	100
表 6. 8 IComxUiPluginServer2 的接口函数列表 .....	100
表 6. 9 IComxUiPluginServerCueline 的接口函数列表.....	100
表 6. 10 IComxUiPluginServerFrm 的接口函数列表 .....	100
表 6. 11 IProperty 的接口函数列表 .....	101
表 6. 12 IComxUiPluginServerEvent 的接口函数列表.....	101
表 6. 13 IComxUiPluginServerEvent2 的接口函数列表.....	101
表 6. 14 一些值得特别关注的组件 .....	116

## 图目录

图 1. 1 COMX SDK 所需的 C++ 开发环境.....	10
图 1. 2 COMX SDK 命令行开发环境.....	11
图 2. 1 COMX KUL\$JS Design 工具主界面.....	12
图 2. 2 COMX KUL\$JS Design 的入门教程.....	13
图 2. 3 COMX Unit 的入创建.....	14
图 2. 4 KUL 文件格式举例.....	15
图 2. 5 KUL Binding 机制示例.....	16
图 2. 6 在线帮助入口界面.....	18
图 2. 7 Binding Browser 工具 .....	19
图 2. 8 Global Field Browser 工具 .....	20
图 2. 9 Event Browser 工具 .....	20
图 2. 10 Namespace Browser 工具.....	21
图 2. 11 KUL 入口文件示例.....	22
图 2. 12 KUL 文件注册（集成开发环境） .....	22
图 2. 13 KUL 文件注册（控制台环境） .....	23
图 2. 14 创建 block widget.....	25
图 2. 15 form 中插入 block widget .....	26
图 2. 16 创建 JS.EXT 扩展 .....	29
图 2. 17 为 JS.EXT 添加新方法 .....	30
图 2. 18 创建支持 Websocket API 的 Form.....	32
图 2. 19 COMX 协同 Node.JS 服务器控制台窗口 .....	44
图 2. 20 创建 Sandbox 窗体.....	46
图 2. 21 Distribute 工具.....	49
图 2. 22 Distribute 工具的 Additional Options 对话框 .....	49
图 3. 1 组件结构示意图.....	51
图 3. 2 WIN32 GUIDGEN.EXE SDK 工具的使用 .....	54
图 3. 3 VC6 创建 Win32 DLL 工程（Step-1） .....	55
图 3. 4 VC6 创建 Win32 DLL 工程（Step-2） .....	56
图 3. 5 COMX 组件代码框架生成工具.....	56
图 4. 1 Step-1 的 GUI .....	60
图 4. 2 Step-2 的 GUI .....	60
图 4. 3 GUI 界面布局分析(Step-1).....	61
图 4. 4 GUI 界面布局分析（Step-2） .....	61
图 4. 5 Edit 控件的有效性检查 .....	70
图 4. 6 list 控件示例.....	71
图 4. 7 等待进度条示意 .....	74
图 4. 8 打开 KEditor 开发工具自身的源代码 .....	75
图 5. 1 COMX 平台组件体系结构.....	78

图 5. 2 WIN32 GUIDGEN.EXE SDK 工具的使用 .....	84
图 5. 3 支持出接口的 COMX 组件实现结构 .....	87
图 6. 1 COMX 的主程序界面 .....	98
图 6. 2 软件包的组织形式 .....	101
图 6. 3 VC6 中使用的 MFC DLL Wizard .....	103
图 6. 4 VC6 中使用的 ATL/WTL DLL Wizard .....	103
图 6. 5 VC6 中使用的 MFC DLL Wizard (step-2) .....	104
图 6. 6 组件代码框架生成向导 .....	104

## Part-I 基础篇

KUL&JavaScript 基本开发环境

# 第1章. 开发环境建立

## 1-1.COMX\_SDK 的安装及配置

### 1-1.1. C++开发环境：VS2005 SP1

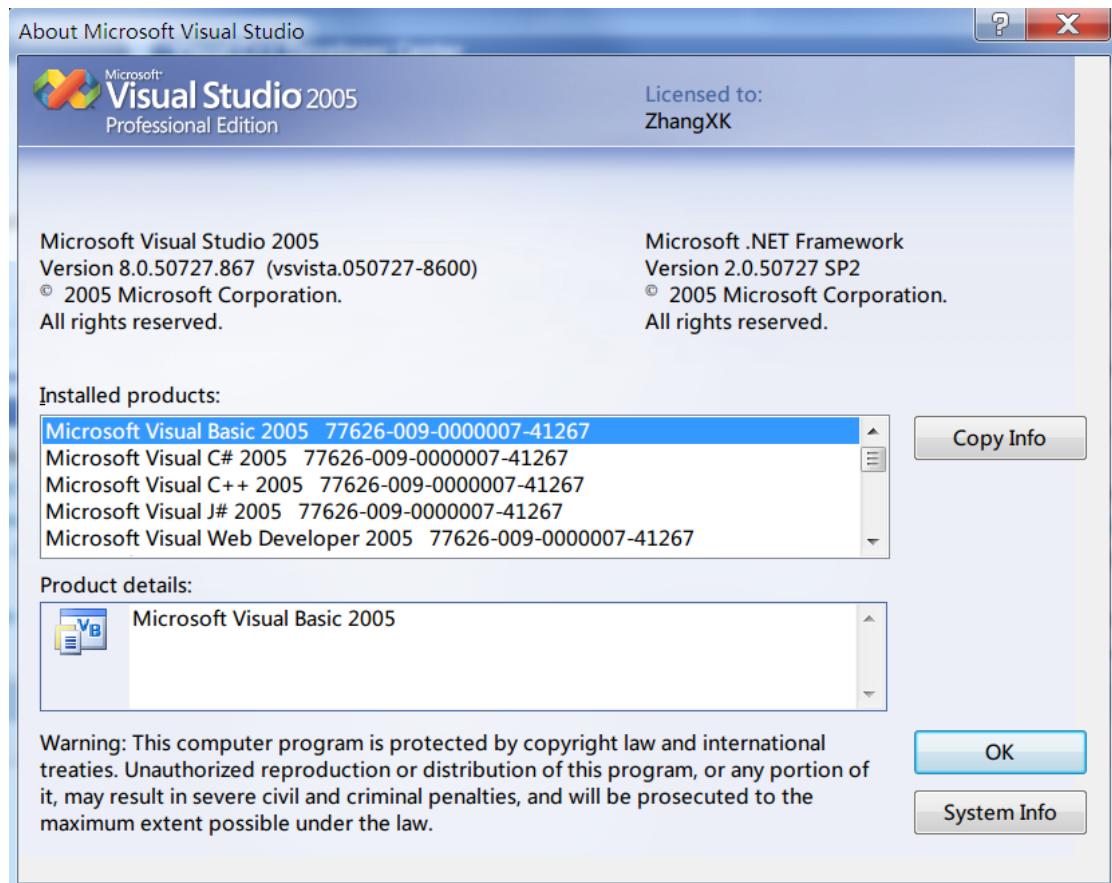


图 1. 1 COMX SDK 所需的 C++开发环境

### 1-1.2. COMX\_SDK 开发包安装和配置

COMX 最新版本开发环境的配置充分利用了 Windows 下的 Cmd 控制台窗口，相关环境变量的设置类似 Siemens NX 中的开发方式，只在当前开发环境 Cmd 窗口中有效，不在污染系统的环境变量空间。

#### 配置步骤：

- ❖ 将 comx\_sdk.zip 解压到一个文件夹下，最好建立一个专门的文件夹管理 COMX SDK 和在其基础上建立的 Unit，比如 D:\COMX.DEV\comx\_sdk，注意：开发包必

须放在名为 comx\_sdk 的子文件夹里。

- ◆ 执行 D:\COMX.DEV\comx\_sdk\run.bat，桌面上生成快捷方式 KEditor，并自动弹出如下图的开发窗口，这个窗口就是 COMX SDK 的基本工作环境：

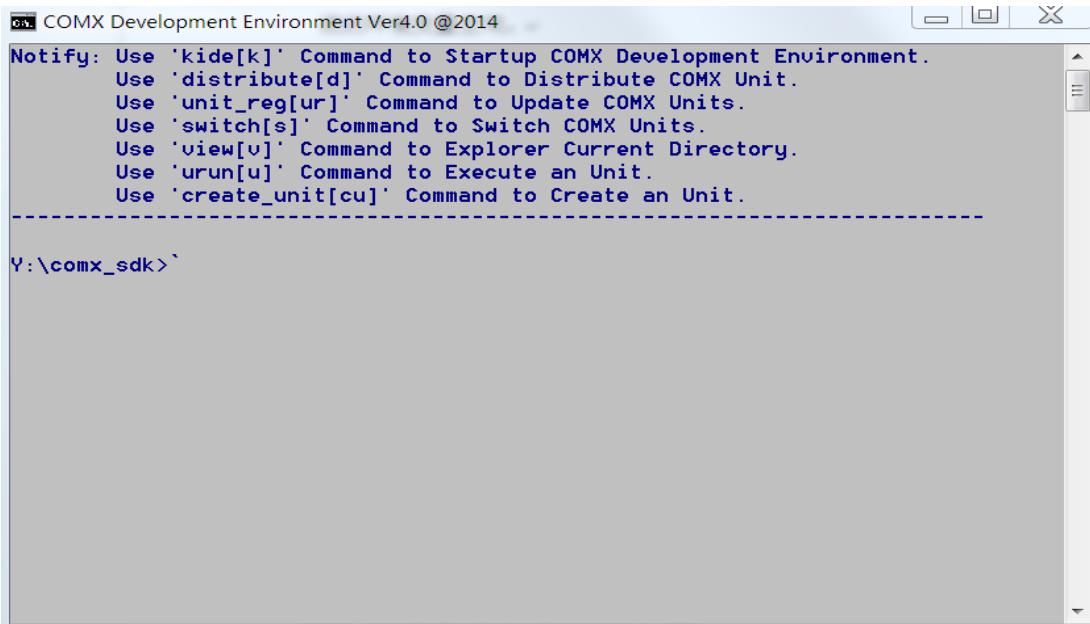


图 1. 2 COMX SDK 命令行开发环境

## 1-2. 常用命令行开发工具

表 1. 1 COMX SDK 常用命令行工具

命令名称	快捷方式	描述
<b>kide</b>	k	启动 COMX SDK 集成开发环境
<b>distribute</b>	d	COMX SDK 和 Unit 打包发布工具
<b>unit_reg</b>	ur	注册一个新的 Unit (一般用于新拷贝的 Unit, 本机建立的 Unit 会自动注册)
<b>switch</b>	s	Unit 之间的切换工具
<b>view</b>	v	打开当前所在目录文件夹窗口
<b>urun</b>	u	运行一个 Unit
<b>create_unit</b>	cu	创建一个新的 Unit
<b>vs2005</b>	-	启动 vs2005

注意：最后注意在 comx\_sdk 的 distribute 文件夹下有一个 vs 工程批量编译工具

build.exe，具体使用方式在命令行下输入：build.exe -h 获取帮助。

## 第2章. KUL&JavaScript

本章内容涉及到 JavaScript 脚本语言，该语言类似于 C/C++比较简单，建议大家找一本相关的书看其基本语法部分即可，HTML 相关部分可以不用理会，有 C/C++语言的开发者应该在 2~3 天内初步掌握该语言的基本语法，这里边需要注意一下几点：

- ✧ JS 是弱类型语言，变量可以不声明使用，并且能够在赋值过程中自动转换类型。
- ✧ JS 是基于函数选型构造类和对象的，也就是说它是基于对象的语言。

注意：KUL&JS 机制和旧版本中的界面机制是完全兼容的，互不影响；也就是说，使用新机制已经做好的旧的代码是无需重写的，二者可以共存。

### 2-1. 一个简单的 KUL 程序实例

- (1) 在图 1.2 所示的命令行窗口中输入 k，然后回车，打开 COMX|KUL&JS Design 工具，如图 2.1 所示，

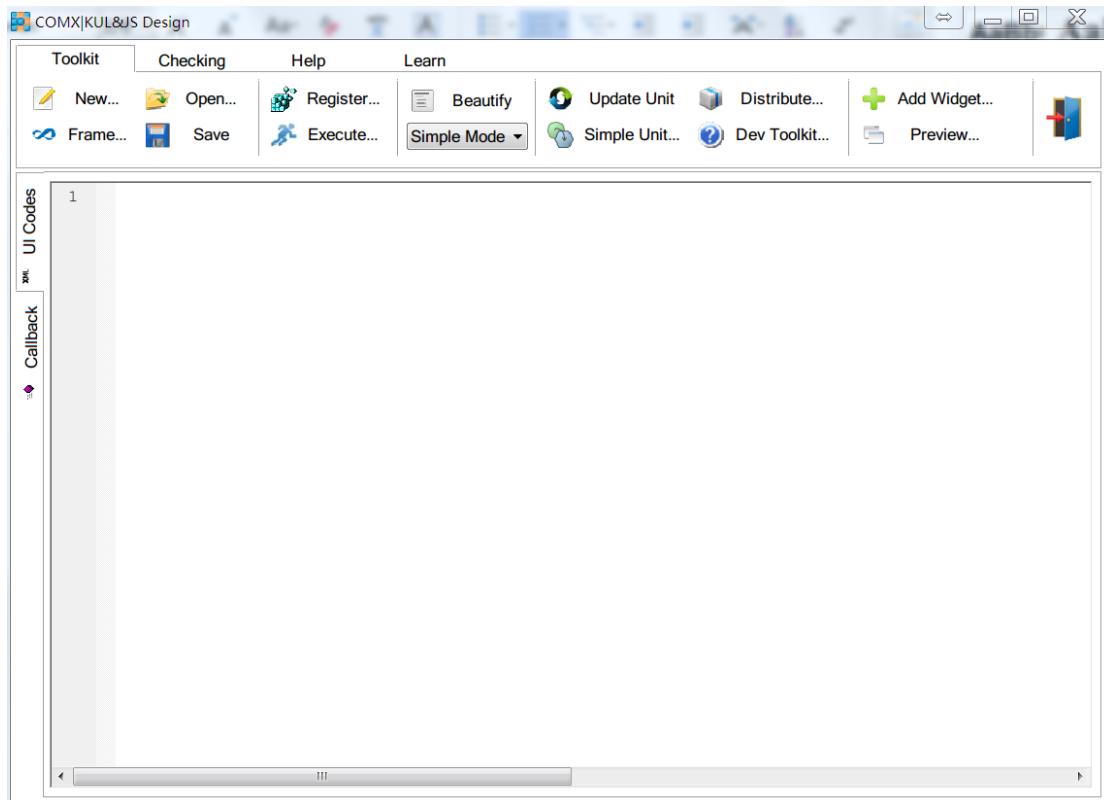


图 2.1 COMX|KUL&JS Design 工具主界面

- (2) 点击图 2.1 所示工具中的 Learn 菜单，打开如图 2.2 的界面，播放教程：[Lesson-1](#)：

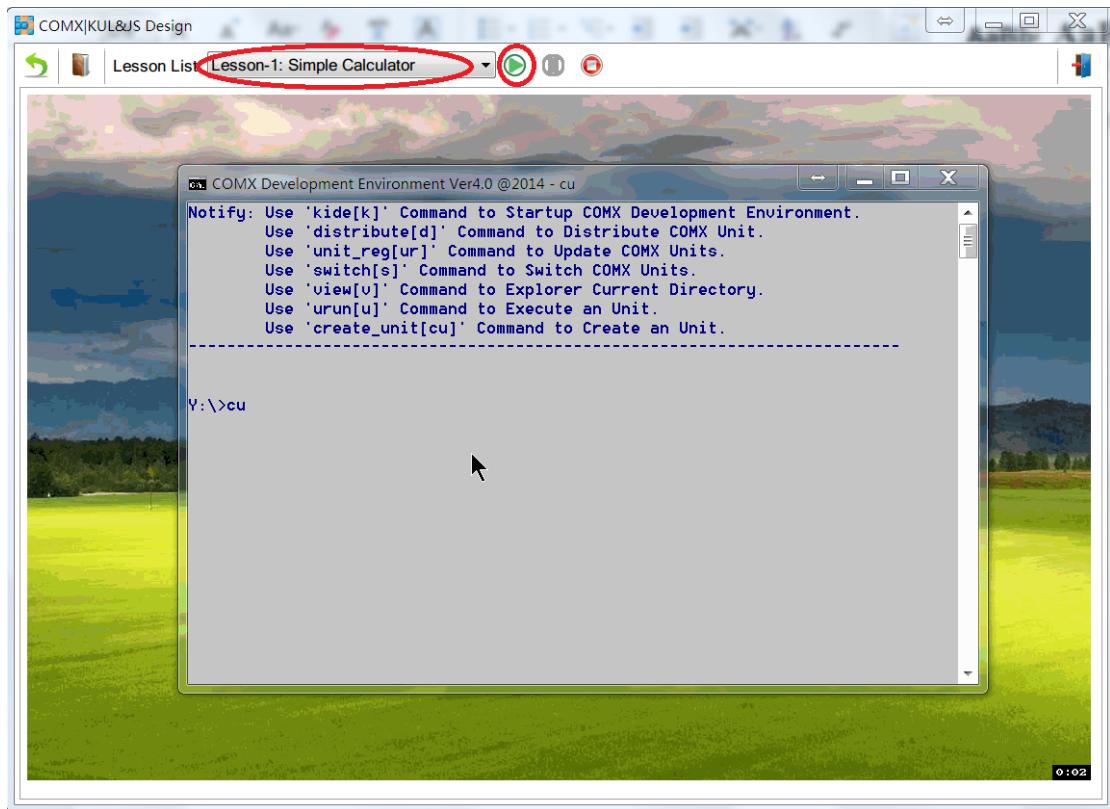
**Simple Calculator**。

图 2. 2 COMX|KUL\$JS Design 的入门教程

(3) 下面的表格给出了上述开发工具的快捷方式列表:

快捷键名称	描述
<b>Ctrl+Shift+G</b>	浏览全局变量
<b>Ctrl+Shift+B</b>	浏览控件 binding 变量
<b>Ctrl+Shift+E</b>	浏览 GUI 事件
<b>Ctrl+Shift+N</b>	浏览扩展函数命名空间
<b>Ctrl+O</b>	打开操作
<b>Ctrl+S</b>	存盘操作
<b>Ctrl+F</b>	查找
<b>Ctrl+R</b>	预览当前 Form
<b>Ctrl+N</b>	新建 Form/Unit/JS.EXT/Method
<b>Ctrl+T</b>	界面重用打包和相应的安装包部署
<b>Ctrl+Q</b>	退出
<b>Ctrl+W</b>	打开一个新的集成开发环境进程
<b>Ctrl+E</b>	运行 Unit

## 2-2. Unit 创建与维护

- 基于 COMX 平台开发的软件包被称为 unit，或者说在 COMX 平台下所有的开发活动都是在某一个 unit 下面完成的，unit 的创建可通过主界面 菜单中的 Unit 选项卡完成，如图 2.3 所示，或者在命令行窗口输入 cu 命令也可以调出该界面（可参考图 2.2 所示的入门教程）：

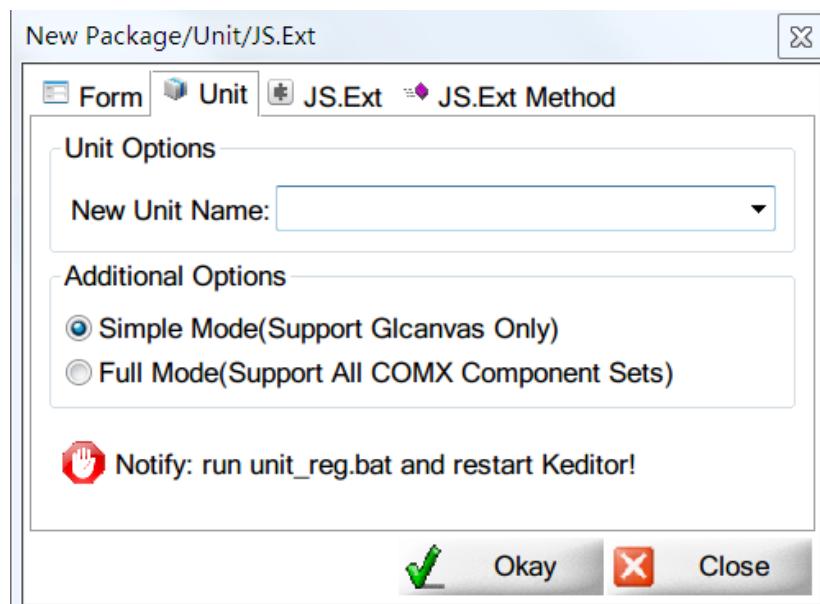


图 2.3 COMX Unit 的创建

- 可通过主界面的 **Update Unit** 菜单更新 Unit 相关头文件和开发环境（只有在使用 COMX 组件机制时才会用到）。
- 可通过主界面的 **Distribute...** 菜单对开发完成的 Unit 打包成 zip 文件，进行绿色版发布。
- 上述工具可以为 COMX 平台下的软件包创建、维护、开发及打包发售提供全生命周期的管理，提升软件部署和开发的效率。

## 2-3. 布局器

(1) KUL 文件格式举例:

```

1  <?xml version="1.0" encoding="gb2312"?>
2  <kul_pkg type="ui" name="prototype">
3      <property name="id">0x61b1e081-0x40f7-0x4df0-0x8f-0xa5-0xde-0xc9-0x97-0xe0-0x59-0x
4          <widget type="form" name="prototype_dialog">
5              <property name="id">0x31252479-0x890a-0x45de-0xbd-0x93-0x29-0x8e-0x19-0xc6-0xc4
6                  <property name="title">KUL|Prototype</property>
7                  <property name="titlebar">true</property>
8                  <property name="maximum_box">true</property>
9                  <property name="resize">true</property>
10                 <property name="center">center</property>
11                 <property name="toolwindow">true</property>
12                 <property name="icon">kmas</property>
13                 <property name="show">max</property>
14                 <property name="style">popup</property>
15                 <property name="app_window">true</property>
16             <widget type="vbox">
17                 <widget type="hbox">
18                     <property name="adjust">horizontal</property>
19                     <widget type="vbox">
20                         <property name="margin">0</property>
21                         <widget type="label">
22                             <property name="id">1025</property>
23                             <property name="text">这是一个KUL演示程序.</property>
24                             <property name="adjust">horizontal</property>
25                         </widget>
26                     </widget>
27                 <widget type="bitmap">

```

图 2.4 KUL 文件格式举例

(2) 从图 2.4 中的 KUL 文件举例我们可以看到 (a) KUL 文件是基于 XML 的。(b) 所有界面元素的 XML tag 之都是<widget>, 类型采用 attribute "type"表达, 而<widget>的属性是通过子 tag <property>来表达的, 格式比较有规律。

(3) KUL 对于界面的描述是基于布局器的, 这里边的容器控件主要有: 【form】、【hbox】、【vbox】、【stack】、【tabctrl】和【subform】。

(4) 容器可以嵌套, 【form】只能有一个子控件, 【subform】是对另一个对话框的整体引用, 其它容器都可以由多个子, 但其排布方式不同 (目前 subform 不建议使用, 仍在改进中)。

(5) 注意: 每个控件都有一个【adjust】 property, 取值为: a)fixed, 水平、垂直方向均固定; b) auto 水平、垂直方向均可自动调整; c) vertical 垂直方向上可调整; d) horizontal 水平方向上可调整。

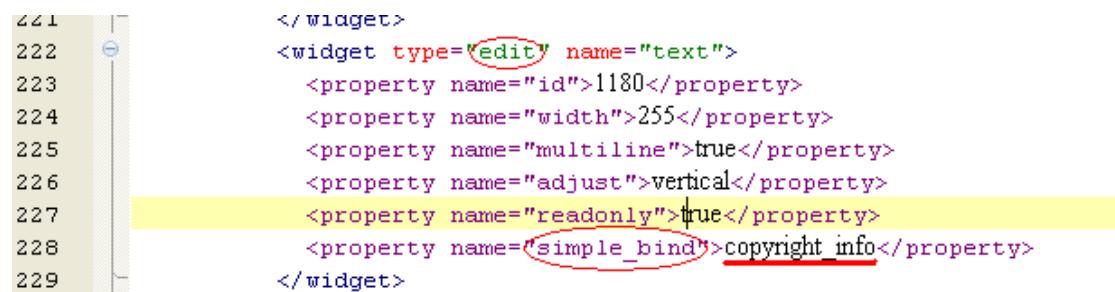
(6) 注意 KUL 中的【space】 widget, 主要用于尺寸的调整 (可理解为弹簧):

- a) 在 hbox 自动为水平的, 在 vbox 中自动为垂直的。

- b) 如果【hbox】或【vbox】中有一个子控件可以吃到空间，该控件自动吃掉空间，如果有多个控件可以吃掉空间，则多余空间在这些控件中平均分配，如果没有可以吃掉空间的控件，则所有子控件自动分配空间。
- (7) 【hbox】中子控件的高度自动取为所有子控件中最大的，【vbox】中子控件的宽度自动取为所有子控件中最大的。
- (8) KUL 文件的位置在当前 unit 文件夹的 kul 子目录。

## 2-4. 控件 Binding 机制

在 COMX 的 KUL&JS 机制中，控件和变量的关联采用 Binding 机制，从图 2.5 的 KUL 文件截图我们可以看到，名称（name）为“simple\_bind”的属性（<property>）将编辑控件（edit）和一个字符串变量捆绑起来，这样在 Javascript 脚本中就可以用变量 binding.copyright\_info 对编辑控件值进行存取和设置。



```

221
222     </widget>
223     <widget type='edit' name="text">
224         <property name="id">1180</property>
225         <property name="width">255</property>
226         <property name="multiline">true</property>
227         <property name="adjust">vertical</property>
228         <property name='simple_bind'>copyright_info</property>
229     </widget>

```

图 2.5 KUL Binding 机制示例

类似的 bind 属性还有：valid\_bind—用于捆绑一个布尔型值；list\_bind—用于捆绑一个一维数组；index1\_bind—用于捆绑一维数组的下标值；rang\_bind—用于捆绑一个范围；table\_bind—用于捆绑一个二维数组；index2\_bind—用于捆绑二维数组的两个下标值；enable\_bind—用于设置控件的 enable 状态；visible\_bind—用于设置控件的 visible 状态；handle\_bind—用于获取控件的句柄（强制转换为 unsigned long 类型，再处理成字符串），表 2.1 给出了每种 bind 类型所适用的控件，及使用示例。

表 2.1 Binding 使用说明

Binding 类型	适用控件	使用举例
simple_bind	Edit, CheckButton, TabCtrl, Scintilla, Bitmap, Combobox, Listbox, Hprogressbar, Vprogressbar, Spin, Hslider, Vslider, List,	Binding.simple_bind_val = 12; Binding.simple_bind_val = "abc"; Binding.simple_bind_val = false; Var sb_val = binding.simple_bind_val;
valid_bind	CheckButton, Edit,	Binding.valid_bind_val = true; Var valid_val = binding.valid_bind_val;

list_bind	Combobox, Listbox	Binding.list_bind_val = "2, abc, def"; Var lbv = binding.list_bind_val;
index1_bind	Stack, TabCtrl, Combobox, Listbox,	Binding.index1_bind = 2; Var idx1_val = binding.index1_bind;
range_bind	Hprogbar, Vprogbar, Hslider, Vslider, Spin	Binding.range_bind.low = 0; Binding.range_bind.high = 100; Var low = binding.range_bind.low; Var high=binding.range_bind.high;
index2_bind	Scintilla, List 注：在 scintilla 中 index1 为行， index2 为列	Binding.index2_bind.index1 = 1; Binding.index2_bind.index2 = 2; Var index1 = binding.index2_bind.index1; Var index2 = binding.index2_bind.index2;
enable_bind	所有可见的窗体子控件	参照 valid_bind 的用法
visible_bind	所有可见的窗体子控件	参照 valid_bind 的用法
handle_bind	所有可见的窗体子控件	单向的只能取值，不能设置，对其赋值不执行任何动作。
table_bind	List	Binding.table_bind = "0, item1, item2, item3, 1, item4, item5, item6"; 注：在 KUL 指定表头有 3 列，这里的 0, 1 表示每一行所使用的图标。图标文件在属性 “image_list” 中定义。

注意：binding 命名空间下的变量做参数的时候是值传递不是地址传递，其模块化的方法参考 12）小结中介绍的方法。

## 2-5. 基于 JavaScript 的消息响应函数

表 2.2 javascript 消息响应函数

类型	描述	举例
onHotKey	适用于 form, 用于处理窗体的快捷键	OnHotKey(id);
onClick	适用于 pushbutton、checkboxbutton、radiobutton、list、toolbar_item	无参数
onChange	适用于 edit、combobox、listbox、hslider、vslider、spin	无参数
onDoubleClick	适用于 listbox、list	无参数
onHeaderClick	适用于 list	无参数
onSwitch	适用于 tabctrl	无参数，当前选中选项卡索引为：comx.gf.tabctrl_index
onLButtonDown	适用于 bitmap、glcanvas	参数 x,y 表示鼠标当前位置

onLButtonUp	适用于 bitmap、glcanvas	参数 x,y 表示鼠标当前位置
onMButtonDown	适用于 bitmap、glcanvas	参数 x,y 表示鼠标当前位置
onMButtonUp	适用于 bitmap、glcanvas	参数 x,y 表示鼠标当前位置
onRButtonDown	适用于 bitmap、glcanvas	参数 x,y 表示鼠标当前位置
onRButtonUp	适用于 bitmap、glcanvas	参数 x,y 表示鼠标当前位置
onMouseMove	适用于 bitmap、glcanvas	参数 x,y 表示鼠标当前位置
onResize	适用于 bitmap、glcanvas	控件尺寸发生改变；参数：cx,cy，表示窗体的宽度和高度

注意：（1）上述事件都是通过属性 KUL 中的<property>来设置的，比如：

```
<property name="onClick">OnOkay();</property>
```

（2）消息响应函数在【form】的“javascript”属性所指定的 js 文件中定义，就是文件的位置是当前工程文件夹的 js 子目录。

## 2-6. KUL 中 WIDGET 的属性定义

KUL 定义中的所有 widget 属性列表，及其默认值定义都在【COMX\_SDK 安装目录】\kul\default.template 文件中定义，该文件也是 xml 格式的。

注意：属性值如果为下面的形式：

```
<property name="position">top/bottom/left/right</property>
```

表示该属性取值范围为{top, bottom, left, right}，默认值为 top。

## 2-7. 系统函数和在线帮助

在 COMX 中提供了一些预定义函数库，这些函数库可以由 COMX 的在线帮助获得，启动任何一个 COMX 下的 GUI 程序，使用【Ctrl】+【Shift】+【d】快捷键可以激活在线帮助入口，如图 2.6 所示，



图 2.6 在线帮助入口界面

(1) 【Binding Browser 工具】：点击上述界面中的按钮 [ Binding Browser ]，或者直

接使用快捷键【Ctrl】+【Shift】+【b】直接进入，界面如图 2.7 所示；该工具主要用于浏览 COMX\_SDK 或者当前正在运行的 unit 中每一个入口界面（对话框，工具条）中为控件定义的捆绑变量，切换 Form ID List 中的值可以浏览不同入口中的控件变量；

**注意：**①所有的空间变量都放置在 binding 命名空间中；②在图 2.7 所示界面中双击一个变量，会自动把该变量的访问方式自动放置到剪切板中，在 js 文件中用【Ctrl】+【V】粘贴就行了。

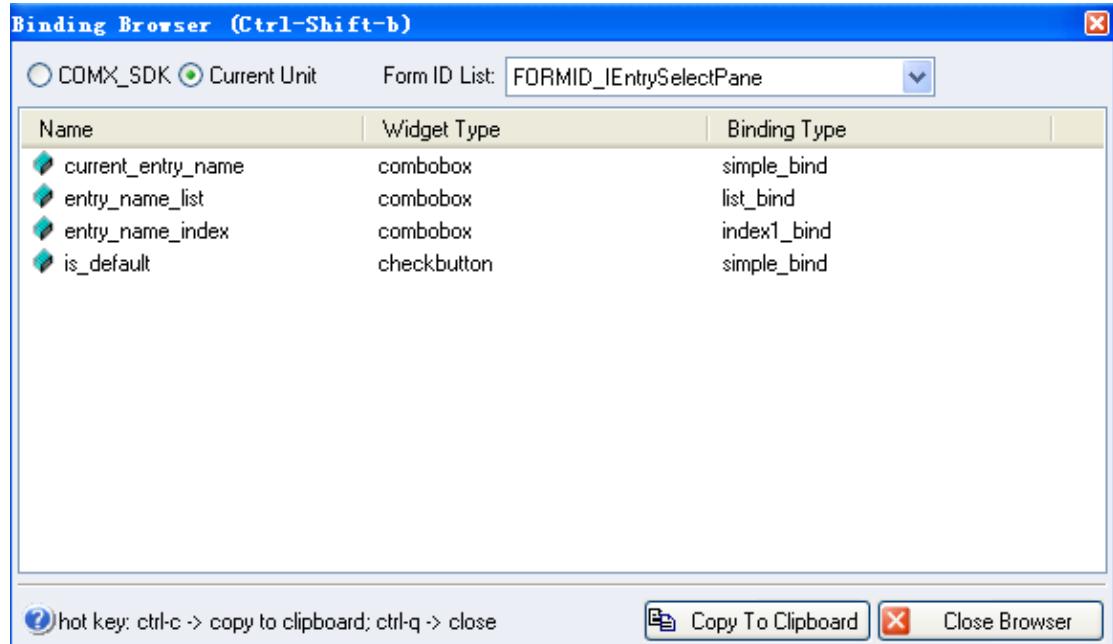


图 2.7 Binding Browser 工具

(2) 【Global Field Browser】工具：点击上述界面中的按钮  **Global Field Browser**，或者直接使用快捷键【Ctrl】+【Shift】+【g】直接进入，界面如图 2.8 所示；该工具主要用于浏览当前时刻 unit 中的全局变量。关于全局变量在本章后面第 2-9 节中会有详细的叙述。

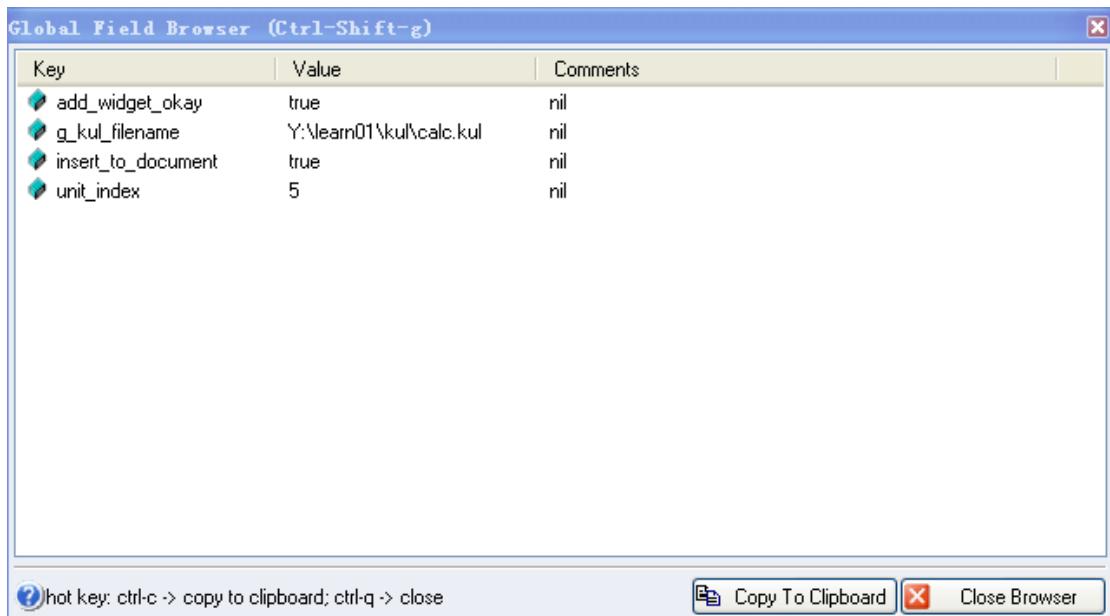


图 2.8 Global Field Browser 工具

(3) 【Event Browser】工具：点击上述界面中的按钮 Event Browser，或者直接使用快捷键【Ctrl】+【Shift】+【e】直接进入，界面如图 2.9 所示；该工具主要用于浏览 COMX\_SDK 或者当前正在运行的 unit 中每一个入口界面（对话框，工具条）中为控件定义的消息响应函数（事件），切换 Form ID List 中的值可以浏览不同入口中的消息响应函数。

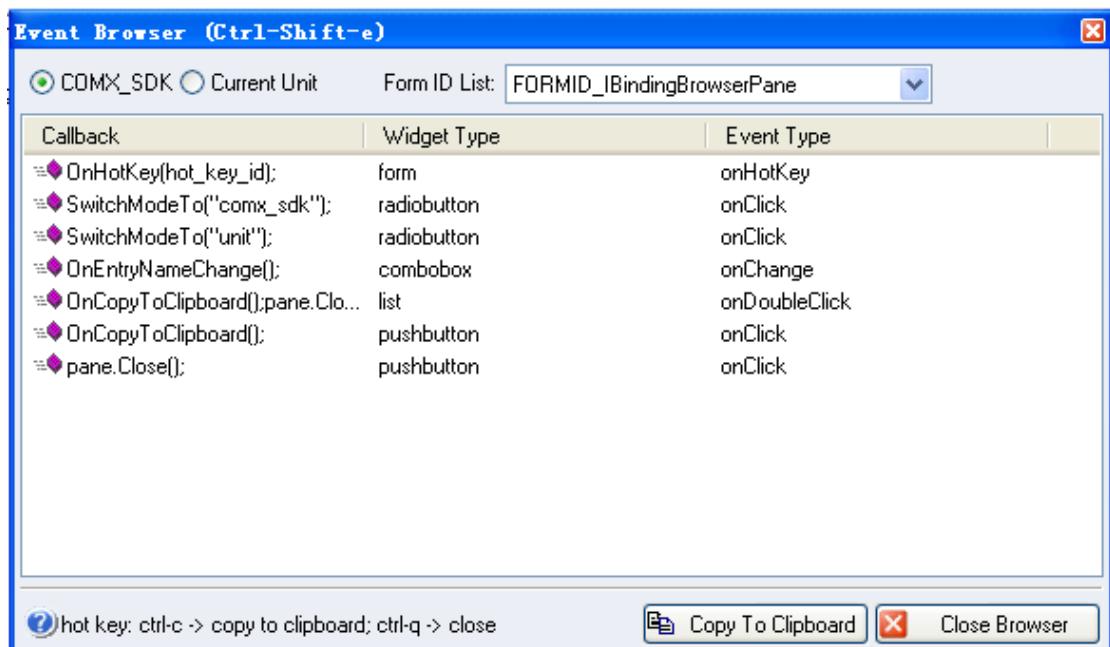


图 2.9 Event Browser 工具

(4) 【Namespace Browser】工具：点击上述界面中的按钮 Namespace Browser，或者直接使用快捷键【Ctrl】+【Shift】+【n】直接进入，界面如图 2.10 所示；该工具主要用于浏览系统预定义的函数、变量以及 JS.Ext 中定义的扩展函数，所有的这些函数都被放置在

采用二级分类的命名空间中。

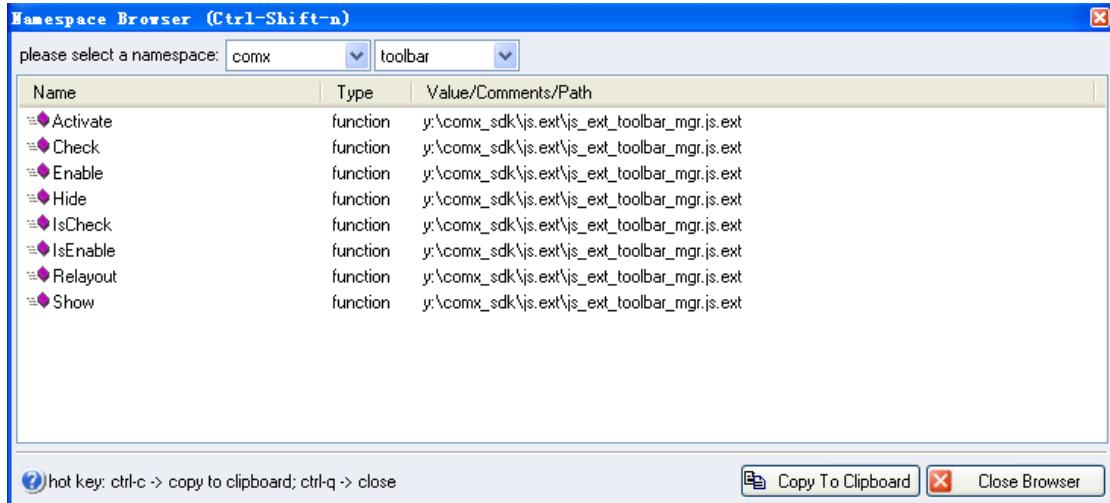


图 2.10 Namespace Browser 工具

系统预定义的命名空间主要有 comx/unit/pane:

- ① 其中 **comx** 中用于存放全局工具函数，其中的子空间主要有：**comx.gf**—用于定义和操作全局变量；**comx.file**—用于定义和访问文件句柄；**comx.entry**—定义 COMX 中预定义的入口元素（对话框和工具条）变量；**comx.ui**—COMX 中预定义的 UI 相关系统函数库；**comx.sys**—COMX 中预定义的系统函数库（比如：文件句柄创建、全局变量管理、程序终止等）；**comx.util**—工具函数库（字符串编码和解码、剪切板操作等）；**comx.dev**—集成开发环境相关函数库。
- ② 其中 **unit** 空间可以访问当前 unit 中入口变量、unit 名字、入口列表、当前执行的入口 ID 值、Dispose 函数。
- ③ 其中 **pane** 空间包含两个成员：**formid**—当前面板（对话框）的 ID 值（KUL 文件中定义的）；**Close ()** 函数—关闭当前面板。

## 2-8. KUL 文件的注册

COMX 中用于定义界面元素（菜单、工具条、对话框等）的 KUL 文件在编辑完成之后需要注册到 unit 中去才可以被调用和执行，这种注册机制可以实现 KUL&JS 机制下的界面可以在不同 unit 之间非常方便的重用；注册是会自动检测是否有同 KUL 文件关联的 JS 文件存在，如果有也会自动拷贝到目标 unit 的 js 子目录中去；

在每个 unit 的文件下面的 **kul** 子目录中会有一个 **kul.entry.xml** 文件用于存放注册在该 unit 的界面元素的入口信息，如下图所示，

```

<?xml version="1.0" encoding="gb2312"?>
<kul_pkg type="entry" name="learn01" xmlns="http://www.kingmesh.com">
    <pkgid name="calc">
        <location>calc.kul</location>
        <value>0xb472686-0x5004-0x479a-0x90-0x26-0x7c-0x86-0x10-0x97-0x5c-0xh8</value>
    </pkgid>
    <formid name="calc_pane">
        <location>calc.kul</location>
        <value>0xc6969347-0x96fa-0x45ad-0x85-0xe2-0xd2-0x43-0xed-0xd2-0xbf-0xd9</value>
    </formid>
</kul_pkg>

```

图 2.11 KUL 入口文件示例

### 如何实现 KUL 文件注册？

- ① 使用【COMX|KUL&JS Design】工具，如下图所示，

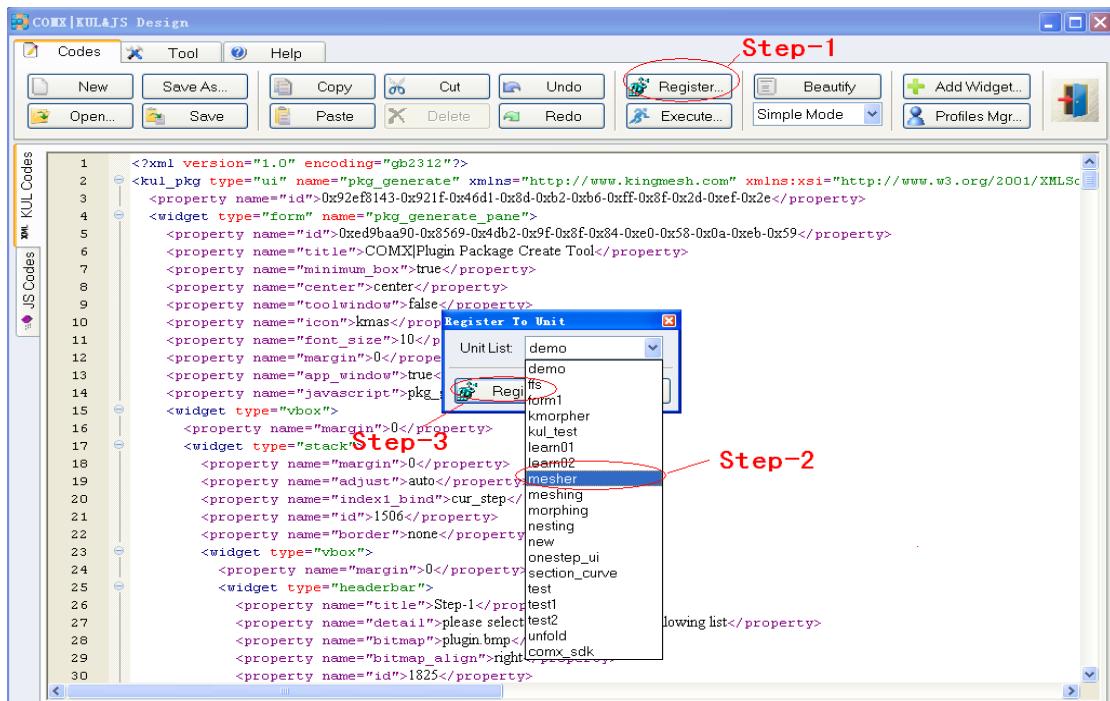


图 2.12 KUL 文件注册（集成开发环境）

- ② 使用 kulreg.exe 命令行工具，该用具位于【COMX\_SDK 文件夹】\bin 目录下，可以在控制台下面执行，在控制台下输入 kulreg -h 或 kulreg -help 可以获得该工具的使用帮助，如下图所示：

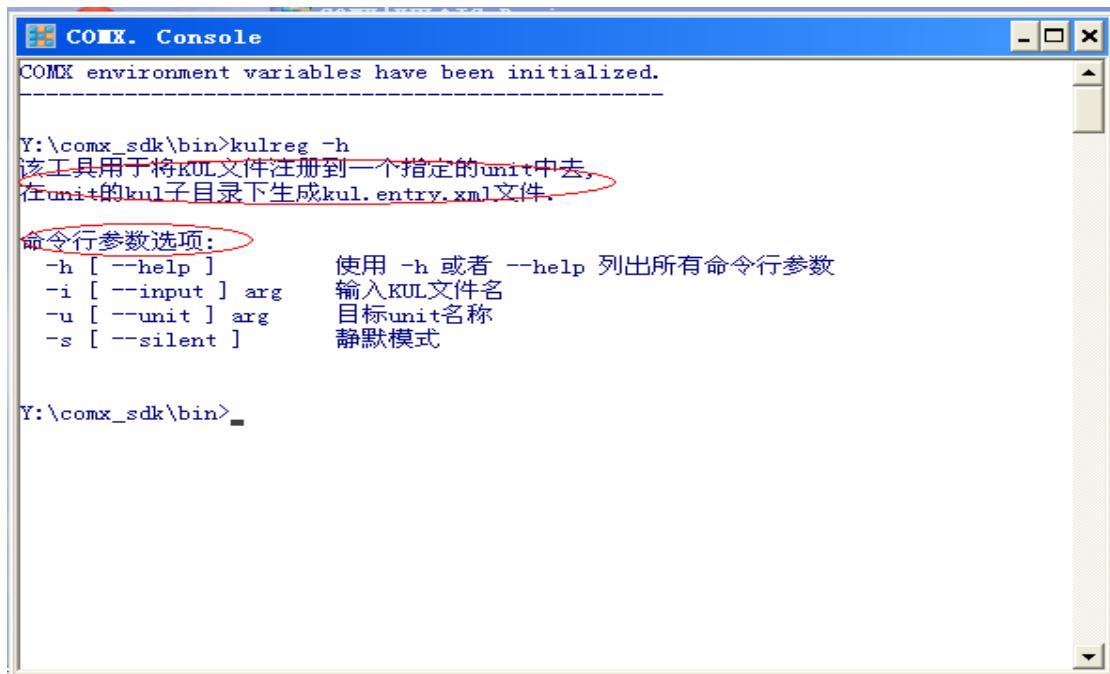


图 2.13 KUL 文件注册（控制台环境）

③ 当前 Unit 生成的 kul 文件会自动注册，只有想在其他 Unit 中重用 kul 文件是才需要注册。

## 2-9. 文件、变量作用域和全局变量

在 COMX 中每一个独立的入口元素（对话框、工具条等）都通过：

```

<property name="javascript">pkg_generate_pane.js</property>
<property name="js_lib">/js/class/widget_wrap.js</property>

```

这样两个属性来定义相关联的 JavaScript Codes，其中 **javascript** 中用于存放消息响应函数等当前入口环境相关的内容，**js\_lib** 则用于引用一些全局定义的通用类和函数库等；前者只能指定一个 js 文件，后者可以用“；”或者“，”分离指定多个 js 文件路径“/js/class/widget\_wrap.js”表示系统（即 COMX\_SDK）下的文件，“~/js/class/widget\_wrap.js”表示当前 unit 下的文件。

**变量作用域：**每一个入口及其 js 文件所构成的 js 运行环境都是独立的，即其中的变量作用域仅仅限于当前入口，包括 binding 命名空间。

可以在 js 文件中定义一个函数 `function OnInitializeData () {...}` 来完成当前 JavaScript 环境中的数据初始化工作，系统会自动检测该函数是否存在，如果有自动执行，没有则忽略。

注意：【Namespace Browser】工具中所能浏览的系统预定义的函数、变量以及 JS.Ext 中

定义的扩展函数等等这些在所有入口中都是可见的。

**全局变量:** 如果不同的入口之间（比如对话框之间）需要传递数据，该怎么办呢？这时候就需要使用全局变量，全局变量在所有入口中都可见，且可以通过本章第 5 节中描述的工具进行浏览。

- ① 全局变量定义在 comx.gf 二级命名空间中。
- ② 所有的全局变量其数据类型都是字符串。
- ③ 字符串数组的表达可以采用类似于 binding 中 list\_bind 的类似方法，把字符串数组转换成字符串，在 JavaScript 中的 String 类和 Array 类提供了非常简单的方法实现二者之间的转换。

定义全局变量的两种方式：

- ① 不需要声明直接赋值，比如：comx.gf.global\_val = “123”;就定义并赋值了一个名为 global\_val 的全局变量。
- ② comx.sys.PushValueToGlobalField (“global\_val”, “123”, “this is some comments”), 这个前面的方式是等价的同样定义了名为 global\_val 的全局变量，区别在于采用这种函数的方式可以为全局变量添加一个注释。

**文件:**JavaScript 本身就是一个解释型的脚本环境，在 COMX 中的定位是用于解决效率要求低的一些问题，比如 GUI 等，因此 COMX 中预定义的文件处理 JS 库函数并未考虑效率，只适用小规模的文本文件。

在 COMX 中可以把文件句柄直接看成字符串进行取值和赋值，分别对应读取文件和文件存盘，相关代码如下所示：

```
comx.sys.CreateFileHandler("js_doc", filename); // 创建文件句柄。  
binding.js_doc = comx.file.js_doc; // 把文件内容放入 scintilla 控件中去  
comx.sys.CloseFileHandler("js_doc"); // 关闭文件句柄。
```

## 2-10. COMX|KUL&JS Design 工具

该工具的使用方式参考图 3 所示教程。

## 2-11. COMX UI 的 block 重用机制

COMX 提供一种 UI 元素的重用方式：Block 块，如图 2.14 所示，可以在 unit 中建立一

个 block，block 要求是一个单根的 widget 树结构，但其中不能出现 form widget。

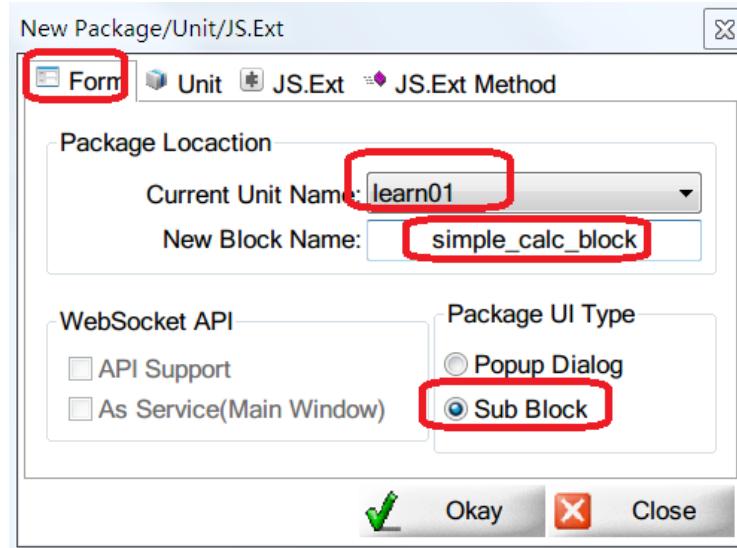


图 2.14 创建 block widget

创建出的 block 例子代码如下所示：

```
<?xml version="1.0" encoding="gb2312"?>
<kul_pkg type="ui_block" name="simple_calc_block">
    <property name="id">0x55f26bf5-0x0385-0x4181-0x94-0xac-0xd6-0x31-0xc3-0x84-0x9f-0xcc</property>
    <widget type="hbox">
        <property name="margin">1</property>
        <property name="adjust">auto</property>
        <widget type="edit" name="text">
            <property name="align">center</property>
            <property name="readonly">false</property>
            <property name="simple_bind">#varLeft</property>
            <property name="onChange">#evtLeftChange</property>
        </widget>
        <widget type="label">
            <property name="align">left</property>
            <property name="text">+</property>
        </widget>
        <widget type="edit" name="text">
            <property name="align">center</property>
            <property name="readonly">false</property>
            <property name="simple_bind">#varRight</property>
            <property name="onChange">#evtRightChange</property>
        </widget>
        <widget type="label">
            <property name="align">left</property>
            <property name="text">=</property>
        </widget>
    </widget>
</kul_pkg>
```

```

<widget type="edit" name="text">
    <property name="align">center</property>
    <property name="readonly">true</property>
    <property name="simple_bind">#varResult</property>
    <property name="onChange"></property>
</widget>
</widget>
</kul_pkg>

```

如上述代码所示 bind 变量和事件处采用#var 和#evt 命名，这样就可以在本 unit 的其它 form 中重用上述 UI 描述，在开发工具主界面点击  Add Widget...，如下图 2.15 所示就可以在 form 中插入 block:

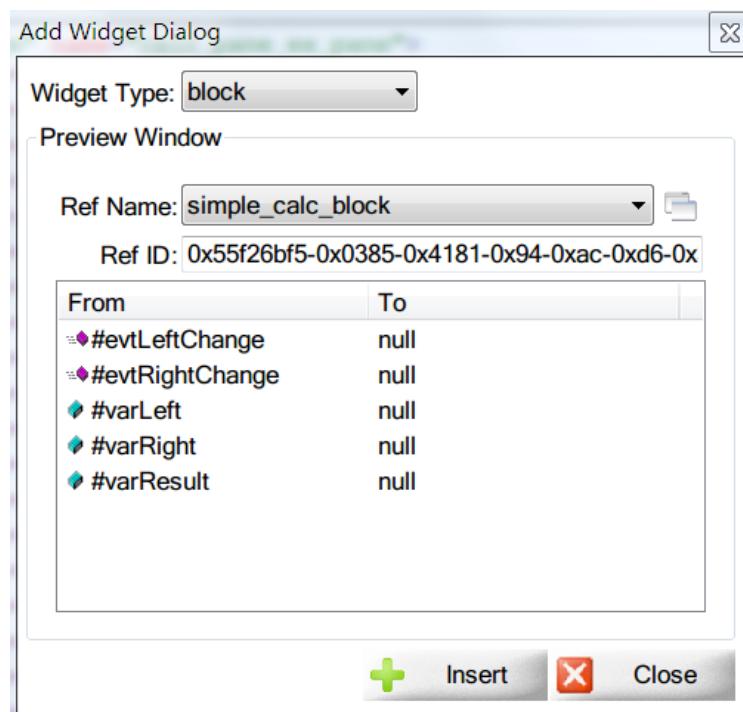


图 2.15 form 中插入 block widget

产生如下的代码，注意 block widget 相当于一个复合 widget 可以在 unit 中多次重用，也可以在一个 form 中多次重用，只要每次指定不同的 binding 变量和回调函数即可：

```

<widget type="block" refname="simple_calc_block" refid="***排版需要 ID 值省略***">
    <event name="#evtLeftChange">OnPlus();</event>
    <event name="#evtRightChange">OnPlus();</event>
    <binding name="#varLeft">left</binding>
    <binding name="#varResult">res</binding>
    <binding name="#varRight">right</binding>
</widget>

```

更详尽的操作流程参考 [【KEditor】→【Learn】→【Lesson-2: Block Widget】](#)。

## 2-12. Javascript 和 Node.JS 环境下一种典型的模块化方法

Javascript 可以将匿名函数作为参数，于是可以有如下的写法：

```
function OnPlus()
{
    //binding.res = parseInt(binding.left) + parseInt(binding.right);

    Plus(binding.left, binding.right, function(res){
        binding.res = res;
    });
}

function Plus(left, right, fn)
{
    var res = parseInt(left) + parseInt(right);

    if(typeof(fn) == 'function')
    {
        fn(res);
    }
}
```

这样就可以实现一种更为灵活的模块化方法：在 Plus 函数中封装通用的代码，具体问题相关代码封装在匿名函数中完成，比传递返回值更为灵活；前文中提到 binding 中的变量不能引用传递，通过上面的方法对 binding 变量的复制就可以封装在匿名函数中完成。

## 2-13. Node.JS 简介

COMX 使用 Node.JS 建立本地后台服务器进行异步调用和远程协同操作，因此本节对 Node.JS 进行简要介绍。

更具体内容请关注如下两个网站：[nodejs.org](http://nodejs.org) 和 [npmjs.org](http://npmjs.org)

### 2-13.1. 什么是 Node.JS?

Node.js 是一个事件驱动 I/O 服务端 JavaScript 环境，基于 Google 的 V8 引擎。目的是为了提供撰写可扩充网络程序，如 Web 服务。第一个版本由 Ryan Dahl 于 2009 年发布，后来，Joyent 雇用了 Dahl，并协助发展 Node.js。其他编程语言的类似开发环境，包含 Twisted 于 Python，Perl Object Environment 于 Perl，libevent 于 C，和 EventMachine 于 Ruby。与一般 JavaScript 不同的地方，Node.js 并不是在 Web 浏览器上运行，而是一种在服务器上运行的 Javascript 服务端 JavaScript。Node.js 实现了部份 CommonJS 规格（Spec）。Node.js 包含了一个交互测试 REPL 环境。

Node.JS 有如下主要特点：

- 1) 采用事件驱动、异步编程。后端支撑的 Javascript 开发环境，Javascript 的匿名函数和闭包特性非常适合事件驱动、异步编程。
- 2) 具有较好的性能。基于 C++ 和 Google V8 引擎实现，以单进程、单线程模式运行。而 COMX 平台的二次开发环境也是基于 V8 引擎实现的 Javascript 脚本环境，可以与 Node.JS 实现良好的协作。
- 3) 提供容易使用的高度可伸缩服务器解决方案。

### 2-13.2. NPM 扩展

Node 包管理器简称 npm。它是一个 Node.js 的包管理器，运行在命令行下，用于管理应用的依赖。按照 Node 官方的定义，npm 是“Node Package Manager”的缩写。从 Node.js 0.6 版本开始，npm 被自动附带在安装包中。

COMX 中 Node.JS 和 NPM 被封装在 Y:\comx\_sdk\nodejs 文件夹里面，如果需要第三方扩展可在控制台窗口切换到该文件夹，然后执行如下指令：

**npm install 【package\_name】**

注：可在 [npmjs.org](http://npmjs.org) 网站查询第三方扩展及其使用方式

## 2-14. 模块化和扩展机制

- 组件对象模型 COMX，参考第 3 章和第 4 章的内容；在最新的架构下面可考虑作为 JS.EXT 的底层，业务逻辑顶层采用 Javascript；

- JS.EXT 扩展机制，将 C++代码封装称 Google V8 Javascript 环境下可以交互的 Javascript API，对于一些对效率要求严苛的场景，可采用回调函数的方式完成模块的链接，Javascript API 仅仅负责模块链接和初始化，这时候可以得到理想的效率；在 JS.EXT 的 C++代码内部可采用 COMX 组件作为底层支撑。
- 通过 comx.shell.run 和 comx.shell.run\_ex 进行简单的进程调用，更为复杂或参数较多的进程调用和交互一般采用 WS API 在 Node.JS 环境下完成。
- 基于 Websocket API 的 Node.JS 异步模块调用，这时候有如下几种模块化方式：
  - ✧ 通过 Node.JS 的 child\_process 进行异步或远程进程调用。
  - ✧ Node.JS 的 C++ Addon
  - ✧ 通过 Node.JS 的其它功能或扩展实现后台 API

## 2-15. JS.EXT 扩展生成和使用

如图 2.16 所示，可以在 KEditor 的 New Pane 中创建一个 JS.EXT 扩展：

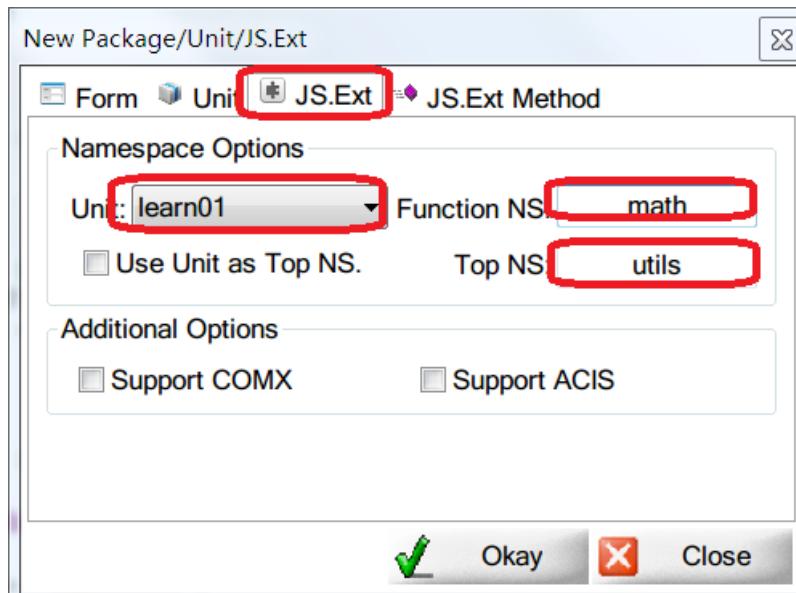


图 2.16 创建 JS.EXT 扩展

一个 JS.EXT 扩展采用两层命名空间来标识，比如上面建立的 JS.EXT 中封装的函数都位于 utils.math 命名空间下，当然仅仅在 learn01 unit 中有效；如果选中了  Use Unit as Top NS。顶层命名空间名称则为当前 unit 名字，这时候该 JS.EXT 中的扩展函数被封装在 learn01.math 命名空间中。

JS.EXT 扩展创建后就可如图 2.17 所示向其中添加新的函数，图中所示的例子将用 C++

实现整形数的加法：

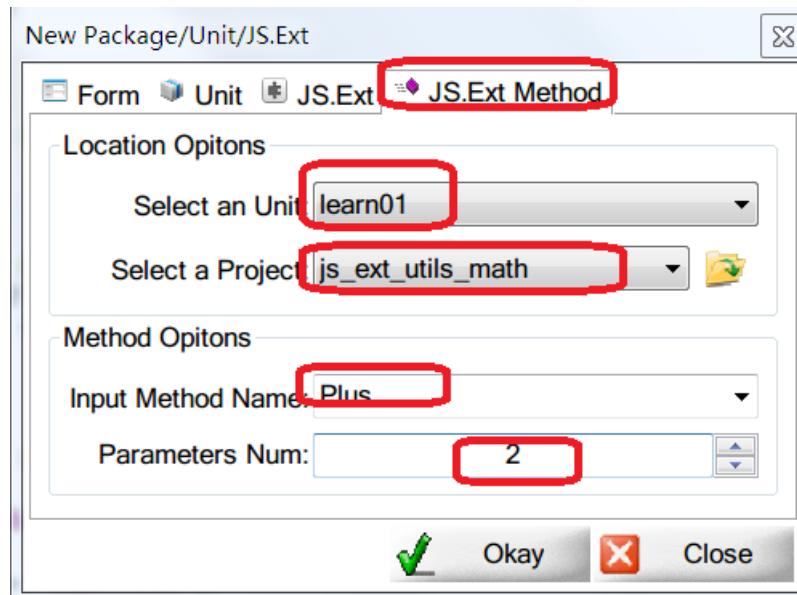


图 2.17 为 JS.EXT 添加新方法

接下来就可以点击上图中的 按钮，打开 vs2005 工程，完成整数加法的 C++ 实现：

```
///////////////////////////// // implement Plus function //////////////////
```

```
#define PLUS_FUNC_USAGE "Plus Usage: var res =  utils.math.Plus(left, right);"
JS_EXT_FUNC_BEGIN(Plus, 2, PLUS_FUNC_USAGE)
{
    int left = JS_EXT PARA(int, 0);
    int right = JS_EXT PARA(int, 1);

    string ret = type cast<string>(left + right);

    string ret_val = type cast<string>(ret);
    JS_EXT FUNC_ASSIGN_RET(ret_val);
}
JS_EXT FUNC_END()
```

注意：在 JS.EXT 中函数的参数和返回值都采用字符串形式传递，JS\_EXT PARA 可以将参数解析成想要的类型，type cast 则能够把返回值转换为字符串形式。

上述代码完成后编译 vs2005 工程，就可以在 unit: learn01 的 js 文件中调用 utils.math.Plus 函数了，相关代码如下：

```
function OnPlus()
{
```

```

//binding.res = parseInt(binding.left) + parseInt(binding.right);

/*
Plus(binding.left, binding.right, function(res){

    binding.res = res;

});

*/
binding.res = utils.math.Plus(binding.left, binding.right);
}

```

JS.Ext 机制使得 COMX 下面集成算法代码更为简单，无需了解复杂的组件机制，只要学生了解 C 语言，知道三种基本程序结构（顺序、分支、循环）就可以按照流程图把算法代码利用 JS.Ext 机制集成进来，极大降低了平台的使用门槛；同时，组件机制还是很重要的，核心的底层数据结构，需要大量重用的代码仍需要通过组件方式完成，这一个层次上的工作要求开发员有 COMX 组件、接口等等的编程功底。这样我们可以实现人力资源分层，让每个人都可以发挥自己的长处，提高项目组整体的工作效率。

注意：Fortran 代码的支持可以在 JS.EXT 的 vs2005 工程中采用 C++/Fortran 混合编程完成。

详细的操作步骤请参考 “**Lesson-3: JS.EXT Sample**” .

## 2-16. COMX 本地进程调用 API

对于一些简单的子进程调用可以采用 V8 本地引擎中的两个扩展函数：

- 1) comx.shell.run(cmd, is\_hide);
- 2) comx.shell.run\_ex(cmd, is\_hide, is\_block);

而对于一些参数较多或需要多个子进程组合调用的需求一般通过 Websocket API 将指令委托给后台 Node.JS 服务器完成；Node.JS 的子进程调用可以实现异步调用和多任务。

下面一行简单的例子代码实现了 KEditor→Learn→Book 中的回掉函数，用于打开 word 文档格式的开发手册：

```
comx.shell.run_ex("y:\comx_sdk\data\help.bat", true, false);
```

help.bat 中的代码如下：

```
@echo off
@start y:\comx_sdk\manual\comx.manual.doc
```

## 2-17. Websocket API

COMX 的 Unit 所建立 Form 程序可以在后台自动启动一个 Node.JS 服务器并通过 Websocket 协议（socket.io）版本与之通讯，这样就可以把一些耗时较长的计算任务、需要异步处理的后台任务、文件处理、异步子进程调用放在 Node.JS 服务器上进行，同时 Node.JS 服务器本身也可以作为 socket.io 客户端实现远程通讯，这一点也是 2-19 节协同机制的基础。

如图 2.18 所示，就可以实现 Websocket 功能的支持；注意：如果所创建的 Form 是主窗口就需要选中  As Service(Main Window)，如果是弹出窗口则不要选中。

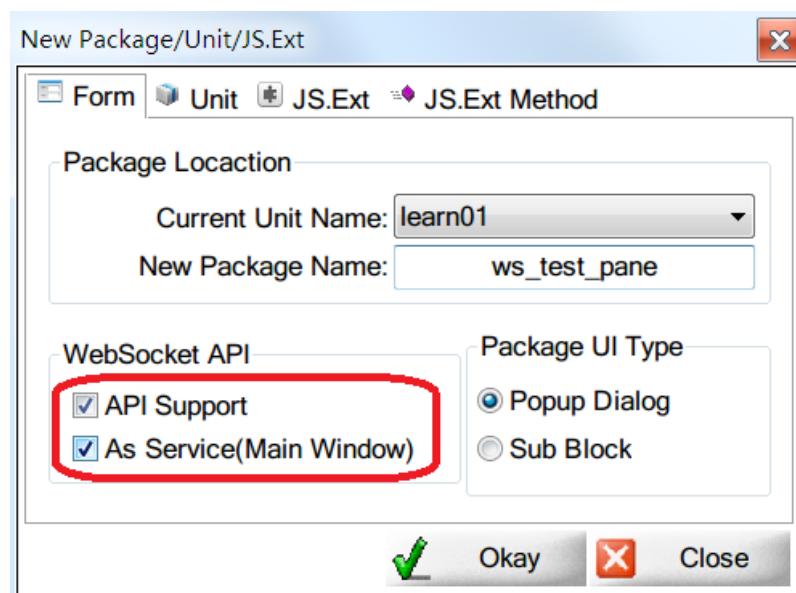


图 2.18 创建支持 Websocket API 的 Form

这样我们就可以在向导生成的 JS 脚本文件中看到如下的代码：

```
var websocket = require('socket.io/comx.websocket.js');
```

对象 websocket 中封装了对 WS API 的支持，下面两个函数 OnInitializeData 和 OnCloseForm 分别启动和关闭 WS 服务，注意 websocket.Startup 中的参数如果为 true 则会打开 Node.JS 控制台窗口调试模式：

```
function OnInitializeData()
{
```

```

    websocket.Startup(true/*debug_mode*/);

}

function OnCloseForm()
{
    websocket.Stop();
}

```

表 2.3、表 2.4 分别给出了客户端和服务端的 Websocket API 接口，值得注意的是客户端和服务端在事件和消息传递方面是对等的，即都可以通过 `Invoke` 和 `on` 函数向对方发送和响应对方发送的事件，异步事件响应和数据发送都是通过 `send` 函数完成。

表 2.3 客户端的 Websocket API 接口

函数名	描述
<b>Startup</b>	用于启动 Websocket 服务器，一般在主窗体的 <code>OnInitializeData</code> 中调用，一个参数： <code>flag: true</code> -Node.JS 控制台调试模式； <code>false</code> -关闭调试模式
<b>Stop</b>	终止 Websocket 服务器，一般在主窗体 <code>OnCloseForm</code> 中调用，无参数。
<b>ConnectTo</b>	和 <code>Connect</code> 功能基本相同，有一个参数： <code>port:</code> 在连接的时候可指定 WS 服务的端口，该函数一般用于本地“沙箱”机制，这时候沙箱中的程序的主窗体要调用该函数， <code>OnCloseForm</code> 中调用 <code>Dispose</code> 函数
<b>ConnectToEx</b>	有两个参数： <code>ip_address</code> 和 <code>port</code> 分别制定 WS 服务器的 ip 地址和端口，用于实现同远程 WS 服务器的链接通讯，此时在 <code>OnCloseForm</code> 中也调用 <code>Dispose</code> 函数。
<b>Connect</b>	连接到主窗体所创建的 Websocket 服务器，一般在弹出窗口的 <code>OnInitializeData</code> 中调用，无参数。
<b>Dispose</b>	断开和主窗体创建的 WS 服务器的连接，一般在弹出窗口的 <code>OnCloseForm</code> 中调用
<b>send</b>	向服务器发送数据，一个参数：可以是任意可通过 <code>JSON.stringify</code> 转换成字符串的 JS 类型，一般用于在 <code>on</code> 函数

	中向服务器发送异步事件响应消息和数据。
<b>Invoke</b>	<p>向 WS 服务器发送异步事件，有三个参数：</p> <p>method_name: 字符串类型，事件名称</p> <p>parameters: 对象类型，参数</p> <p>callback: 通过该函数异步响应 WS 服务器反馈信息，服务器端一般是由 send 函数激发的。</p>
<b>on</b>	响应服务器端的 Invoke 事件，反馈信息通过 send 函数完成。
<b>IsSyncLock</b>	判别协同服务通道目前是否锁定，用于避免协同时候的“死锁”
<b>CmdSync</b>	发送协同指令，类似于 Invoke
<b>onCmdSync</b>	响应协同指令，类似于 on
<b>ConnectToCmdSrv</b>	连接到协同服务器，两个参数： ip_address 和 port
<b>DisconnectFromCmdSrv</b>	断开和协同服务器的连接。

表 2.4 服务器端的 Websocket API 接口

函数名	描述
<b>Invoke</b>	<p>向客户端发送异步事件，有三个参数：</p> <p>method_name: 字符串类型，事件名称</p> <p>parameters: 对象类型，参数</p> <p>callback: 通过该函数异步响应客户端反馈信息，客户端一般是由 send 函数激发的。</p>
<b>InvokeEx</b>	和 Invoke 功能基本相同，参数完全一致；所不同在于该函数会同时在 Websocket 层面同时发送广播消息，主要用于如下场合：“沙箱”机制，存在多个客户端连接
<b>on</b>	响应客户端的 Invoke 事件，反馈信息通过 send 函数完成。
<b>send</b>	<p>向客户端发送数据，两个参数：</p> <p>data: 可以是任意可通过 JSON.stringify 转换成字符串的 JS 类型，一般用于在 on 函数中向服务器发送异步事件响应消息和数据。</p> <p>is_broadcast: 是否发送广播消息，和 InvokeEx 配合使用，主</p>

要用于沙箱机制。

详细的操作步骤请参考 “**Lesson-4: Websocket API Sample**”

## 2-18. 基于 Node.JS 的异步后台进程调用

在 2-17 节中提到，Unit 中的 Form 程序可以开启一个 Node.JS 本地后台服务器，并通过 Websocket API 与之通讯，在 Node.JS 服务器中我们可以调用其子模块及各种 npm 扩展来实现各种异步调用，Node.JS 中能够实现对 C++ 代码调用有两种方式：Addon C++ 扩展和 child\_process 子进程调用，在 COMX 中 C++ 扩展已通过 2-15 节详述的 JS.EXT 扩展机制实现，所不同的是不能直接实现“异步”调用，而 2-20 所述的“沙箱机制”可以很好的解决该问题，所以 Node.JS 中的 Addon C++ 插件被排除在 COMX 整体解决方案之外，只保留其通过 child\_process 子模块实现的异步子进程调用，这样我们的 C++ 模块就可以封装成子进程以这种方式调用。

### 2-18.1. Node.JS 的 child\_process 使用举例

依旧以前面提到的简易加法计算器为例，在 Lesson-4 中我们利用 Node.JS 中的 Javascript 语法实现了加法，此处我们把加法操作封装在一个子进程中；首先假定子进程已完成，在命令行下的调用方式如下：plus.exe --left 12 --right 34(简写为：plus.exe -l 12 -r 34)，执行结束后再控制台上输出 46；相应的后台 Websocket 服务 API 如下，这样就可以在 GUI 界面上调用后台子进程中的 C++ 代码：

```

var g_plus_result = false;

websocket.on('Plus', function(data){
    var left = data.parameters.left;
    var right = data.parameters.right;
    if(!left || !right)
    {
        return;
    }
    var cmd = home_dir + 'bin/plus.exe';
    var parameters = [];

```

```

parameters.push('-l');

parameters.push(left);

parameters.push('-r');

parameters.push(right);

var cp = require('child_process').spawn(cmd, parameters, []);

cp.stdout.on('data', function(result){

    g_plus_result = (" + result).split('\r')[0].split('\n')[0];

});

cp.on('exit', function(code){

    if(" + code == '0'

    {

        data.parameters = {'result' : g_plus_result};

        websocket.send(data);

    }

});

//data.parameters = {'result' : parseInt(left) + parseInt(right)};

//websocket.send(data);

});

```

这里面需要说明如下两点：

1. cp 的 stdout.on 和'exit'消息响应中要通过如下方式： (" + result)和" + code 把返回值转换成字符串。
2. 在 stdout.on 消息响应函数中的代码：

```
(" + result).split('\r')[0].split('\n')[0];
```

其目的在于去掉输出行末尾的回车符（在 windows 上的回车符一般为\r\n，也就是 C++ 代码中的 cout << endl; split 是 Javascript 字符串中预定义的函数，起作用是以参数中的字符串（此处是\r 和\n）为分隔符把字符串分解成字符串数组。

## 2-18.2. boost 命令行工具

在 2-18.1 节的介绍中, 假定把加法的 C++ 编码封装成一个命令行工具 plus.exe -l 12 -r 34, 同时该工具输入 -h ( --help ) 参数可以显示其使用帮助, 这是命令行工具的一种标准写法, 可以通过 boost 库中的命令行工具来完成, 代码如下:

```
#include <boost/program_options.hpp>
using namespace boost;

#include <iostream>
#include <string>
using namespace std;

#include <base/type_cast.hxx>
using namespace KMAS::type;

bool parse_cmd_line(int argc, char **argv, string &left, string &right);

int _tmain(int argc, _TCHAR* argv[])
{
    string left = "", right = "";
    if(parse_cmd_line(argc, argv, left, right))
    {
        int iLeft = type_cast<int>(left);
        int iRight = type_cast<int>(right);
        cout << iLeft + iRight << endl;
        return 0;
    }
    else
    {
        return -1;
    }
}

#define DESCRIPTION "这是一个简易的加法计算器"

bool parse_cmd_line(int argc, char **argv, string &left, string &right)
{
    boost::program_options::options_description options("命令行参数选项");

    try
    {
```

```
options.add_options() ("help,h", "使用-h 或者--help 列出所有命令行参数")
    ("left,l", boost::program_options::value<string>(), "左操作数")
    ("right,r", boost::program_options::value<string>(), "右操作数");

boost::program_options::variables_map vmap;

boost::program_options::store(
    boost::program_options::parse_command_line(argc, argv, options), vmap);
boost::program_options::notify(vmap);

if (argc == 1)
{
    cout << "使用-h 或者--help 列出所有命令行参数" << endl;
    return false;
}

if (vmap.count("help"))
{
    cout << DESCRIPTION << endl << endl;
    cout << options << endl;
    return false;
}

if (vmap.count("left"))
{
    left = vmap["left"].as<string>();

    if (left == "")
    {
        cout << "left 参数不能为空;可使用-h 或者--help 列出所有命令行参数" << endl;
        return false;
    }
    else
    {
        cout << "left 参数不能为空;可使用-h 或者--help 列出所有命令行参数" << endl;
        return false;
    }
}

if (vmap.count("right"))
{
    right = vmap["right"].as<string>();
```

```

        if (right == "")
        {
            cout << "right 参数不能为空;可使用-h 或者--help 列出所有命令行参数" << endl;
            return false;
        }
    }
    else
    {
        cout << "right 参数不能为空;可使用-h 或者--help 列出所有命令行参数" << endl;
        return false;
    }
}

catch (...)
{
    cout << "使用-h 或者--help 列出所有命令行参数" << endl;
    return false;
}

return true;
}

```

上述代码基本上可以作为模版使用,注意**红色**字体部分要替换成与当前工具有关的特定代码,更详细的说明请参考 boost.org 上的帮助手册。

2-18.1 和 2-18.2 节中的内容细节请参考“**Lesson-5: Node.JS Child Process**”。

### 2-18.3. jsoncpp 引擎

在 2-18.1 和 2-18.2 所述的例子中,同子进程的通讯是采用命令行参数完成的,但命令行参数的缓冲区尺寸是有限的,这对于数据量比较大或参数较多、较复杂的情形无法胜任,这时候可以采用缓冲文件的方式进行数据传递, cp.stdout.on('data', ...)用于出发数据传递事件, cp.on('exit', ...)事件用于发送子进程终止事件。在 COMX 的 Javascript 环境下缓冲文件一般采用 json 格式。

#### 1. 什么是 json?

- a) JSON 指的是 JavaScript 对象表示法 (JavaScript Object Notation)
- b) JSON 是存储和交换文本信息的语法。类似 XML。
- c) JSON 是轻量级的文本数据交换格式,比 XML 更小、更快,更易解析。
- d) JSON 使用 Javascript 语法来描述数据对象,但是 JSON 仍然独立于语言和平台。JSON 解析器和 JSON 库支持许多不同的编程语言。目前非常多的动态

(PHP, JSP, .NET) 编程语言都支持 JSON。

## 2. JSON 的语法

**JSON 语法是 JavaScript 语法的子集。**

- a) 数据在名称/值对中
- b) 数据由逗号分隔
- c) 花括号保存对象
- d) 方括号保存数组

**JSON 的名称/值对**

JSON 数据的书写格式是：名称/值对。

名称/值对包括字段名称（在双引号中），后面写一个冒号，然后是值：

```
"firstName" : "John"
```

这很容易理解，等价于这条 JavaScript 语句：

```
firstName = "John"
```

**JSON 值**

- a) 数字（整数或浮点数）
- b) 字符串（在双引号中）
- c) 逻辑值（true 或 false）
- d) 数组（在方括号中）
- e) 对象（在花括号中）
- f) Null

**JSON 举例**

```
var employees = [  
    { "firstName": "John" , "lastName": "Doe" },  
    { "firstName": "Anna" , "lastName": "Smith" },  
    { "firstName": "Peter" , "lastName": "Jones" }  
];
```

在上面的例子中，对象 "employees" 是包含三个对象的数组。每个对象代表一条关于某人（有姓和名）的记录。

**JSON 文件**

- a) JSON 文件的文件类型是 ".json"

b) JSON 文本的 MIME 类型是 "application/json"

### JSON 解析器

注意: `eval()` 函数可编译并执行任何 JavaScript 代码。这隐藏了一个潜在的安全问题。

使用 JSON 解析器将 JSON 转换为 JavaScript 对象是更安全的做法。

JSON 解析器只能识别 JSON 文本，而不会编译脚本。在浏览器中，这提供了原生的 JSON 支持，而且 JSON 解析器的速度更快。较新的浏览器和最新的 ECMAScript (JavaScript) 标准中均包含了原生的对 JSON 的支持。

Web 浏览器支持	Web 软件支持
<ul style="list-style-type: none"> <li>▪ Firefox (Mozilla) 3.5</li> <li>▪ Internet Explorer 8</li> <li>▪ Chrome</li> <li>▪ Opera 10</li> <li>▪ Safari 4</li> </ul>	<ul style="list-style-type: none"> <li>▪ jQuery</li> <li>▪ Yahoo UI</li> <li>▪ Prototype</li> <li>▪ Dojo</li> <li>▪ ECMAScript 1.5</li> </ul>

由于 COMX 和 Node.JS 的 Javascript 引擎都是基于 Google V8 的，因此二者支持同样的内置 JSON 解析器语法，举例如下：

```
var employee_str = JSON.stringify(employees); //转换为字符串
var employee = JSON.parse(employee_str); //转换为对象
```

### 3. jsoncpp 引擎及其使用举例

仍然以前面的简易加法计算器为例，我们可以在 COMX Javascript 脚本中利用 2-9 节中所述的文本文件读写引擎把参数存储在一个 json 文件中（注意用 `JSON.stringify` 转换为字符串），然后在 Websocket API 中传递文件名就行了，然后在 Node.JS 中通过如下方式把 json 文件传送给 plus.exe： `plus.exe -f y:\learn01\data\cache\plus.parameters.json`，这就需要在 plus 子进程的 C++ 代码中能够从 json 文件中读取参数，并把结果写到该文件中去，这个工作一般是通过 jsoncpp 开源引擎完成的，jsoncpp 引擎的源码在 `y:\comx_sdk\json` 文件夹下面。

相关细节请参考“Lesson-6: jsoncpp Sample”，代码模版请参考 learn01 unit 中的 `src\plus` 工程，learn01 unit 可通过 `y:\comx_sdk\manual\learn01.zip` 文件解压并在 COMX 环境下配置得到。

客户端关键代码片段如下：

```

function OnPlus()
{
    if(binding.left == "" || binding.right == "")
    {
        return;
    }

    var cache = websocket.CacheJSON({'left':binding.left, 'right':binding.right})

    websocket.Invoke('PlusEx', {'cache' : cache}, function(data){
        binding.result = websocket.ParseCache(data.parameters.cache).result;
        websocket.DetachCache(data.parameters.cache);
    });
}

```

上述代码中使用了在 websocket 中封装的三个函数，如表 2.5 所示

表 2.5 websocket 中的缓冲区处理函数

函数名	描述
<b>websocket.CacheJSON</b>	把一个 Javascript 对象封装到 json 文件中去，文件路径为该函数的返回值。
<b>websocket.ParseCache</b>	从 json 缓冲文件中解析处 Javascript 对象。
<b>websocket.DetachCache</b>	删除缓冲区中的 json 文件。

C++子进程中的关键代码如下：

```

Json::Value cache_root;
Json::Reader reader;

string cache_str = "";
ifstream ifs(cache_fname.c_str());
if(ifs)
{

```

```

cache_str += string(istreambuf_iterator<char>(ifs.rdbuf()),
                     istreambuf_iterator<char>());
}

if(!reader.parse(cache_str, cache_root))
{
    return -1;
}

string left = cache_root["left"].asString();
string right = cache_root["right"].asString();

int iLeft = type_cast<int>(left);
int iRight = type_cast<int>(right);

string result = type_cast<string>(iLeft + iRight);

cache_root["result"] = result;

ifs.close();

Json::FastWriter fast_writer;
ofstream ofs(cache_fname.c_str());

if(ofs)
{
    ofs << fast_writer.write(cache_root);
}

```

上述红色代码片段分别为 json 文件的读和写操作。

## 2-19. COMX 下的 Websocket 协同架构

在表 2. 3 中给出了一组协同相关的 Websocket API 接口，通过这些 API 可以在 COMX 框架下实现软件协同操作。

首先协同功能需要一个服务器，在 COMX 开发环境下可以在控制台窗口执行 service.bat 工具启动服务器（默认端口为 1976），在 unit 发行包中会有一个 service 子文件夹，执行其中的 start.exe 即可启动服务器（默认端口也为 1976），该指令执行后会弹出如图 2. 19 所示的 Node.JS 控制台窗口：

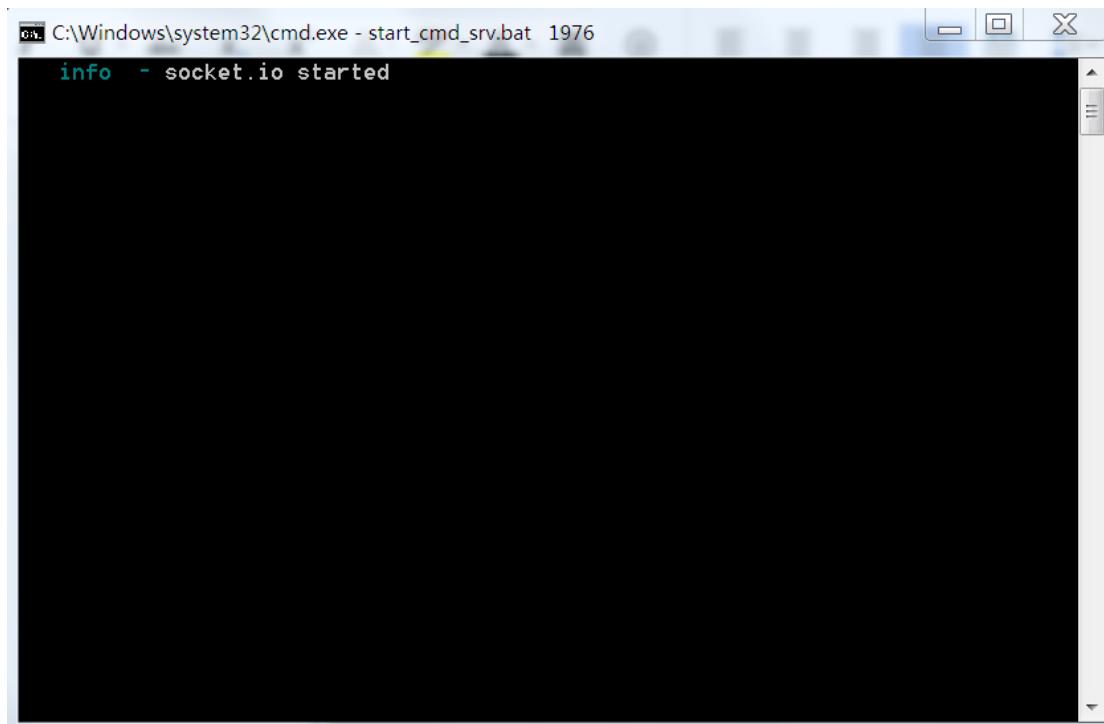


图 2.19 COMX 协同 Node.JS 服务器控制台窗口

接下来就可以在 COMX 的 Form 客户端中实现协同功能（此处仍以 ws\_calc\_pan1.kul 中的加法计算器为例）；

- 1) 首先在 Javascript 脚本的 OnInitializeData 函数中添加协同服务器初始化代码：

```
websocket.ConnectToCmdSrv('localhost', '1976');
```

- 2) 然后在 OnCloseForm 中添加协同服务器退出代码：

```
websocket.DisconnectFromCmdSrv();
```

- 3) 原有的 OnPlus 代码被重构为如下形式（去掉红色代码，打开 OnPlus 中的注释就是重构前的代码）：

```
function OnPlus()  
{  
    if(!websocket.IsSyncLock())  
    {  
        websocket.CmdSync('PlusSync', {'left':binding.left, 'right':binding.right});  
        Plus();  
    }  
}
```

```
//Plus();//重构前的代码  
}
```

```
websocket.onCmdSync('PlusSync', function(data){  
    binding.left = data.parameters.left;
```

```

binding.right = data.parameters.right;

Plus();
});

function Plus()
{
    if(binding.left == "" || binding.right == "")
    {
        return;
    }

    var cache = websocket.CacheJSON({'left':binding.left, 'right':binding.right})

    websocket.Invoke('PlusEx', {'cache' : cache}, function(data){
        binding.result = websocket.ParseCache(data.parameters.cache).result;
        websocket.DetachCache(data.parameters.cache);
    });
}

```

这里需要特别注意的是`websocket.IsSyncLock`函数的使用，主要用于避免编辑事件激发同步指令，同步指令再激发编辑事件造成的“死锁”现象。

## 2-20. “沙箱” 机制

### 2-20.1. 什么是“沙箱（Sandbox）”？

沙箱是一种按照安全策略限制程序行为的执行环境。早期主要用于测试可疑软件等，比如黑客们为了试用某种病毒或者不安全产品，往往可以将它们在“沙箱”环境中运行。

经典的沙箱系统的实现途径一般是通过拦截系统调用，监视程序行为，然后依据用户定义的策略来控制和限制程序对计算机资源的使用，比如改写注册表，读写磁盘等。

### 2-20.2. COMX 中的“沙箱”的含义

COMX中沙箱的概念与上述定义略有不同，从前面章节知道COMX开发环境提供如下不同“粒度”上的模块化机制：①COMX组件②JS.EXT扩展③本地子进程调用（comx.shell.run）④Node.JS异步子进程调用⑤Node.JS C++ Addon扩展等不同的方式，其中①②两种方式子模块中的C++代码和COMX Unit建立的主窗体处于一个进程空间中，考虑到COMX平台的可扩

拓展所带来的第三方插件的不可预知性，需要有一个机制避免第三方插件崩溃或出错导致主程序崩溃。

于是，我们基于Websocket API和Node.JS子进程调用机制把需要隔离的①②两种方式的扩展隔离在独立的进程空间内（称为“沙箱”进程），“沙箱”进程和主窗体之间可以通过Websocket API通讯和交互。

简言之，COMX中的“沙箱”就是把Unit中的一个Form封装在一个子进程中完成；如果需要执行的第三方扩展没有GUI界面，可以把它放在沙箱Form的OnInitializeData中进行，计算结果通过WS API返回，然后调用pane.Close();马上关闭沙箱Form，并且该Form以隐藏方式执行（可在kul文件中通过“show”属性取值为“minimize”、“app\_window”属性取值为“true”设定）。

### 2-20.3. COMX 中如何使用 “沙箱”？

如图2. 20所示，可以创建一个Sandbox窗体：

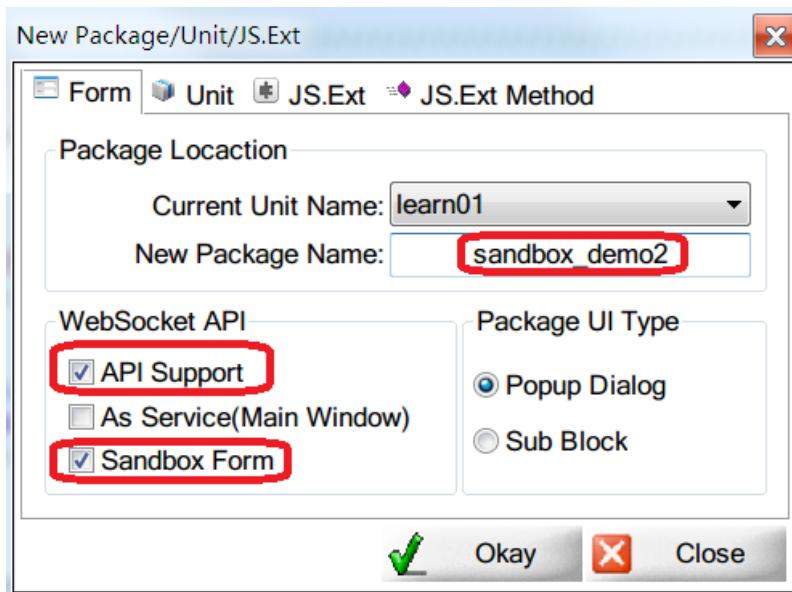


图 2. 20 创建 Sandbox 窗体

点击Okay按钮之后会自动生成下述代码：

```
function OnInitializeData(sandbox)
{
    if(!sandbox)
    {
        alert('sandbox form can\'t be executed alone!');
```

```

    pane.Close();

    return;

}

websocket.BeginSandbox(sandbox.port, sandbox.id, function(){

    //some init codes here.

    util.math.Error();

    pane.Close();

});

}

```

如果是一个自动执行的隐藏沙箱进程，则需在 `//some init codes here.` 代码行下面添加被沙箱保护的代码调用，然后调用 `pane.Close();` 关闭进程，被保护代码调用期间可以通过 WS API 和主窗体通讯；同时要注意在 `kul` 文件中对顶层 Form Widget 通过“`show`”属性取值为“`minimize`”、“`app_window`”属性取值为“`true`”让沙箱隐藏执行，并且 `kul` 文件中至少要有一个 `widget`，不能为空，否则沙箱将异常终止。

接下来，就可以在主窗体里调用沙箱进程了，代码如下所示：

```

function OnError()

{
    //utils.math.Error();

    websocket.InvokeSandbox(unit.FORMID_ISandboxDemo1Pane, {'data':'hello,sandbox!'},

        function(data){

            if(data.parameters.code != 0)

            {

                alert('沙箱进程异常终止');

            }

        });

}

```

详细操作细节请参考“[Lesson-7: Sandbox Sample](#)”。

## 2-21. 定时器操作

COMX 还支持定时器的操作，代码示例如下：

```
SetTimeout (50, function (nID) {
    KillTimeout (nID);
    //需要在定时器中执行的代码
});
```

**SetTimeout** 函数用于创建定时器，其中的参数 50 表示触发定时器的时间（单位：毫秒），回调函数中的 nID 参数用于标识定时器，**SetTimeout** 也可通过返回值获取该标识，和回掉函数中的参数 nID 值完全相同。

**KillTimeout** 用于销毁 nID 参数制定的定时器，此处代码在定时器第一次触发即销毁定时器，那么定时器只触发一次，否则定时器一直触发，直到 **KillTimeout** 函数调用为止，当然 **KillTimeout** 函数也可以在回调函数外的任何位置调用只要把 **SetTimeout** 函数返回值传递给它就行了。

## 2-22. 组件和接口在 JavaScript 环境下的使用

目前组件和接口的使用主要是通过 JS.Ext 中的 C++ 代码对组件和接口进行访问，后面的版本中可能会提供 JavaScript 级别上对组件的直接访问机制，不过目前来看在 JS.Ext 中实现对组件的访问已经足够用了。

## 2-23. 基于 KUL&JavaScript 的插件 package 的生成和使用

在 COMX 新的机制中提供三种 UI 重用的方式：①KUL 文件直接注册到另一个工程中去；②subform widget 带来的重用机制，目前这种重用方式不推荐使用（它会带来一定的复杂性，考虑是否还有必要存在，将来有可能被废弃掉）③插件 package 的生成和安装，可以把某一个 unit 中自称体系（或者叫相对独立）的 KUL 文件、JS 文件、二者所需的附件 DLL 文件、图片文件、数据文件、脚本文件等打包到一个 pkg 文件中去，然后可以随意安装到 COMX 下任何一个 unit 下面。

在之前 COMX 版本中插件机制一直停留在一个概念阶段，本节中的 pkg 文件可以理解为插件的实体化表达方式。

Package 机制的一个典型应用描述，比如：在 KMAS|Onestep 有一个材料库及材料参数导入、选取的功能，其中包括一个工具条，几个对话框和若干通过 JS.Ext 或组件方式集成访问的 C++代码，我们可以把上述内容采用视频 learn02 中描述的打包工具打包成一个 pkg 文件，然后安装到 KMAS|Increment 中，这样我们就可以非常方便的实现材料库机制的重用了。

## 2-24. COMX 中的 Distribute 工具

当一个 unit 开发完成之后，可以在控制台工具窗口输入 distribute 指令（简写为 d）将其打包成可独立发型的 zip 文件，如图 2. 21 所示：

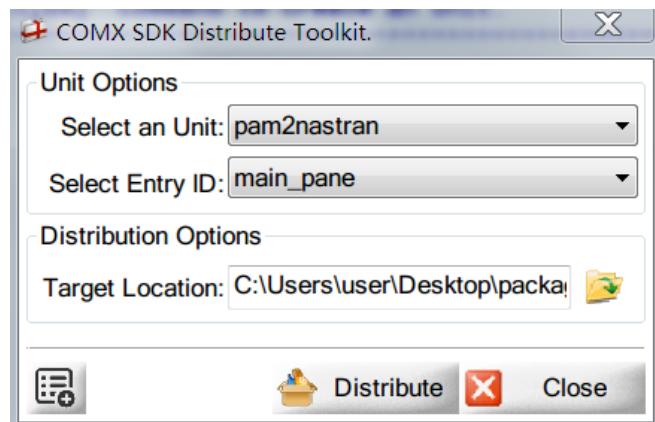


图 2. 21 Distribute 工具

点击 按钮将自动在 Target Location 文件夹中自动生成 unit 的 zip 打包文件。如果有一些需要手动添加的额外打包文件，点击图 2. 21 中的 按钮弹出如图 2. 22 所示的界面，可以添加附加文件、文件夹，并决定是否生成桌面快捷方式：

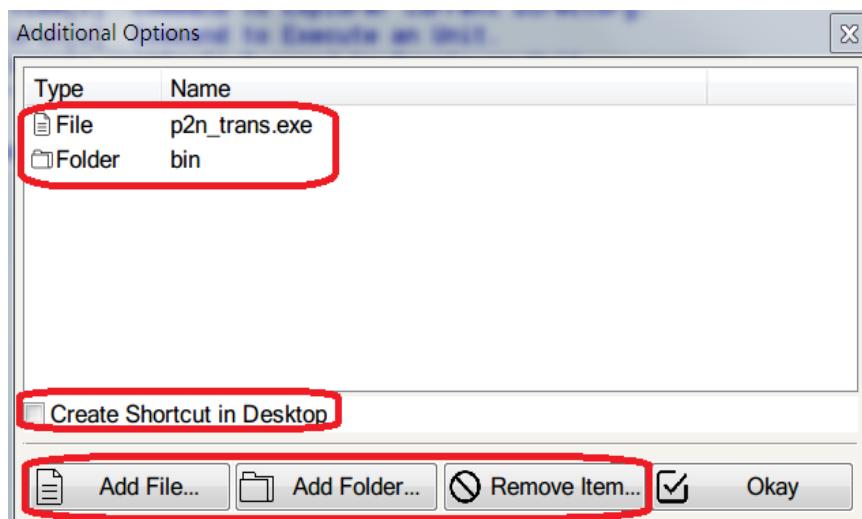


图 2. 22 Distribute 工具的 Additional Options 对话框

## 2-25. 在那里可以找到 KUL&JS 程序相关实例？

在 COMX\_SDK 安装文件夹的 kul 和 js 子文件夹下存放了大量的例子代码，这些代码您都可以用【COMX|KUL&JS Design】工具浏览和进行参考，他们就是我们前面在视频教程中用到的相关工具的代码，其中也包括【COMX|KUL&JS Design】工具本身（kul\_editor.kul）。

## 2-26. 关于图标和图片

在 KUL 文件中有一些图片相关的属性比如：“icon” 需要读取.ICO 文件；“bitmap”、“picture” 等需要读取.BMP 文件，这些文件的指定不要给出全路径只需给出文件名即可，系统首先在当前 unit 的 picture 子文件夹中搜索，如果没找到在到 COMX\_SDK 的 picture 子文件夹下搜索，如果还未找到则使用 COMX\_SDK 的 picture 子文件夹下的 default.bmp 文件，它是一个笑脸的图案。

在 KUL 中图标和图片文件的扩展名可以省略，COMX 解析 KUL 文件是会自动添加(.ico 和.bmp)。

# 第3章. COMX 组件使用简化教程

本章的内容从使用者的角度讲述 COMX 组件机制如何创建和使用，平台的使用者基本上掌握本章的内容即可，如果您需要了解 COMX 组件的原理等更到细节内容可以参考第 4 章；

第 5 章的内容讲述旧的 COMX 开发包中如何利用 MFC 开发界面相关的组件和插件，在新的机制下面完全可以被 KUL&JS 机制取代，如果您不需要维护基于第 5 章技术的旧版本软件代码，该章内容也可以忽略；即便如此，采用第 5 中的技术创建出来的界面组件是被新机制兼容的，可以同 KUL&JS 机制共存。

## 3-1. COMX 是什么？

为了设计一个通用 CAE 基础软件开发平台我们选取微软 COM 技术中最核心的部分，设计和实施了一个面向多种操作系统（WINDOWS/LINUX/UNIX 等）的针对 CAE（CAD）软件开发的简化版本的 COM 内核，并在此基础上构建了基于“微内核+插件”的 CAE 软件开发框架；上述精简版本的跨平台 COM 内核被称为：A simplified Component Object Model for X Platforms（其中，X 代表 WINDOWS/LINUX/UNIX 等操作系统），简称为 COMX；并且，“微内核+插件”体系结构中的“内核”和“插件”都是以 COMX 组件形式进行构造的，因此将本文所涉及的基于“微内核+插件”的 CAE 基础软件开发平台称为 COMX 开发平台（简称为 *COMX 平台*）。

## 3-2. 组件、接口、出接口

COMX 中的组件可以简单理解成二进制级别上可重用的软件积木块，如下图所示：



图 3.1 组件结构示意图

上图中以左下方的黄色积木块为例，积木块提供给别的积木块使用的突出的部分我们称为组件的接口，即组件可以提供的被其它组件使用的一个功能集合；积木块需要其它积木块填充的凹陷部分被称为出接口，及组件为了完整运行需要其它组件提供的功能集合。

在 COMX 中，理论上所有组件对其他组件的调用都通过出接口完成，这样一个组件的接口和出接口一旦定义出来就是自成体系的；接口和出接口可以理解成不同组件积木块能够匹配需要满足的标准；使用出接口的另外一个好处就是组件在程序运行时刻可以实现动态拼接，拥有更好的模块化特性。

**下面是在 COMX 中定义一个接口的语法如下：**

```
// {0E962D73-B070-418b-ADCF-7774A398585E}

const MUID IID_IFemRender =
{ 0xe962d73, 0xb070, 0x418b, { 0xad, 0xcf, 0x77, 0x74, 0xa3, 0x98, 0x58, 0x5e } };

interface IFemRender : public IRoot {
    virtual TStatus STDCALL SetDisplayMode(const int &mode) = 0;
    virtual TStatus STDCALL GetDisplayMode(int &mode) = 0;
    virtual TStatus STDCALL SetVisualMode(const int &mode) = 0;
    virtual TStatus STDCALL GetVisualMode(int &mode) = 0;
    .....
};
```

这里需特别指出如下几点：a) 接口中所有的成员函数都必须为**纯虚函数**。b) 所有接口函数的返回值都必须为**TStatus** 类型，并且用**STDCALL** 关键字修饰。c) 接口函数中的参数类型必须为**C++语言中的原生数据类型或简单类型**（即 int, double, float, char, 结构体类型 struct），当然也包含这些类型的指针和引用类型，有时根据设计上的需要，接口类型也可出现在其它接口的参数列表中，但一般并不推荐这种用法。d)**所有的接口都必须直接或间接地派生于根接口 IRoot。**

某一个接口在组件 A 中实现，在组件 B 中被引用，那末相对于 A 来说它就是接口，相对于组件 B 来说它就是出接口，出接口的定义不需要特别的语法。

### 3-3. 组件的类型和生存周期

组件一般可以分成如下几类：服务型的、单实例的、普通组件。

**服务型组件：**在 COMX 主程序启动时自动创建，在主程序退出时自动销毁，类似于 Windows 系统中的后台服务程序。

**单实例组件：**一旦该组件的实例被创建出来，且未被销毁，那末无论该组件在后边创建多少实例，都会返回内存中已有的未被销毁的组件实例，即在任意时刻单实例组件的实例只存在一个。

**普通组件：**既非服务型组件也非单实例组件的组件就是普通组件，类比于 C/C++ 中动态分配的局部变量。

### 3-4. 类工厂

COMX 中的组件的创建和管理都是通过类工厂来完成的，类工厂在 C++ 程序上对应于一个类 TFactory，所在的头文件是 #include<base/factory.hxx>；

可以调用类工厂的 CreateInstance 创建普通组件或者单实例组件（视组件创建是指定的类型而定），调用 QueryService 方法查询服务型组件。

代码示例如下：

```
TFactory factory;
IFemCore *p_fem_core = NULL;
Factory.CreateInstance(CLSID_IFemCore, IID_IFemCore, (void**)&p_fem_core);
If(p_fem_core != NULL)
{
    //通过 p_fem_core 调用组件中相关方法。
    P_fem_core->Release();
}
```

前三行代码也可以用一个宏简化，实例如下：

```
CREATE_COMX_OBJ(factory, CLSID_IFemCore, IFemCore, p_fem_core);
```

接下来是服务型组件查询的实例代码：

```
TFactory factory;
```

```

IGlContent *p_gl_content = Null;

Factory.QueryService(CLSID_IGlContent, IID_IGlContent, (void**)&p_gl_content);

If(p_gl_content != NULL)

{

    //通过 p_gl_content 调用组件中相关方法。

    p_gl_content ->Release();

}

```

前二行代码也可用一个宏简化，实例如下：

```
QUERY_SERVICE(factory, CLSID_IGlContent, IGlContent, p_gl_content);
```

**关于上述代码我们需要注意以下几点：**

- ① 所有组件的访问都是通过其接口指针的方式来完成的，该指针使用完毕之后必须调用 Release() 方法，实现组件实例的释放，服务型组件也是如此。
- ② 每一组件通过一个 CLSID 值来标识，每一个接口都通过 IID 值来标识，全局性唯一标识符 UUID 是一个独有的、128 位的、并且具有非常高可靠率的数值。它把一个独有的网络地址(48 位)和一个非常精细的时间印鉴(100 纳秒)结合在一起。COMX 对 UUID 的实现被称为全局特有标识符 MUID(在 COM 中相应的名字为 GUID，COMX 中为了名称上的区别，而不至于在某些 WIN32 编程环境中产生语法歧义，将其命名为 MUID)，它在结构上和 UUID 非常类似。COMX 中用 MUID 来识别组件的类 (CLSID)、接口 (IID) 和出接口 (EID) 等基本元素。在实际组件开发过程中，使用 WIN32 上的 GUIDGEN.EXE 工具来生成 MUID，如图 3.2 所示，一般情况下组件代码框架的工具会自动产生 CLSID 和 IID，上述工具在手工编码时使用：

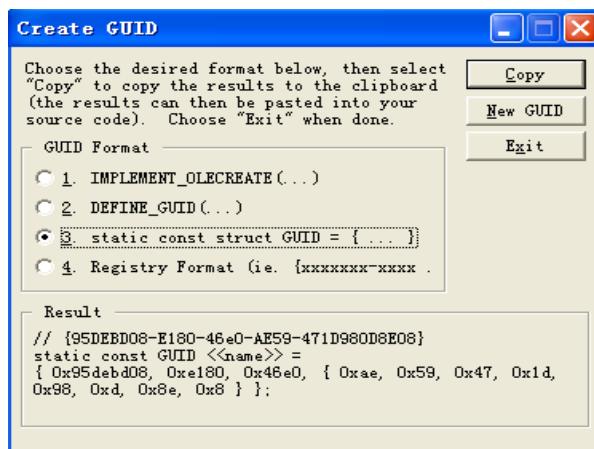


图 3.2 WIN32 GUIDGEN.EXE SDK 工具的使用

- ③ 每一个接口指针都可以调用 QueryInterface 查询出该接口指针所在组件所支持的其

它接口，如果欲查询接口类型不存在返回 NULL 值，代码示例如下：

```
IGlContent *p_gl_content = NULL;
//一些获取接口的代码
IFemCore *p_fem_core = NULL;
P_fem_core->QueryInterface(IID_IFemCore, (void**)&p_fem_core);
If(p_fem_core != NULL)
{
    //利用指针 p_fem_core 调用相关方法
    P_fem_core->Release();
}
```

上述代码的前两行可简化为：

```
QUERY_INTERFACE(p_gl_content, IFemCore, p_fem_core);
```

- ④ 如果一个接口名称为 IFemCore 那么它的 IID 常量名称必须为 IID\_IFemCore。

### 3-5. 生成组件代码框架的工具

COMX 中组件代码框架可以采用工具自动生成，步骤如下：

- ① 在 VC6 中创建一个工程，错误!未找到引用源。、错误!未找到引用源。所示，

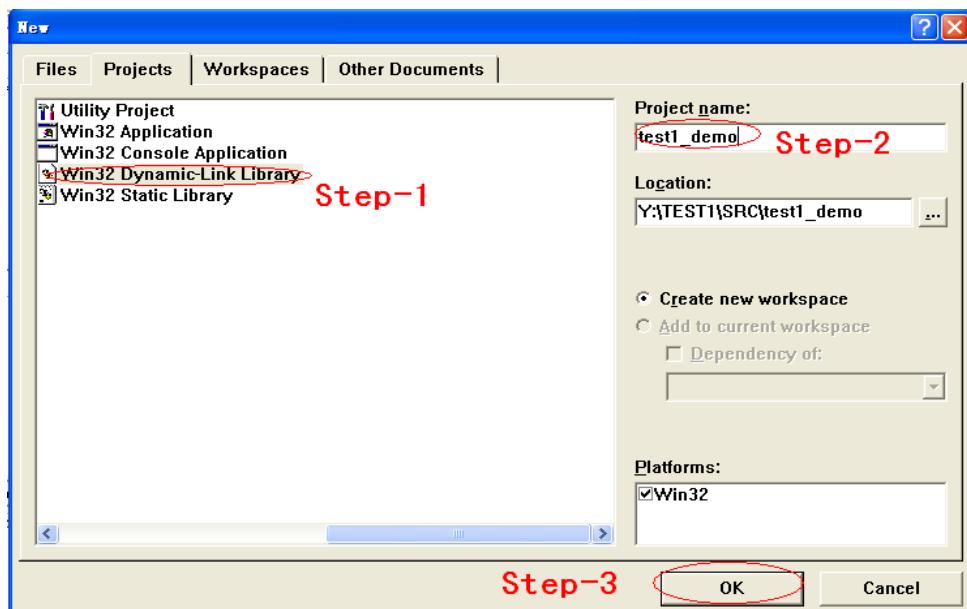


图 3.3 VC6 创建 Win32 DLL 工程 (Step-1)

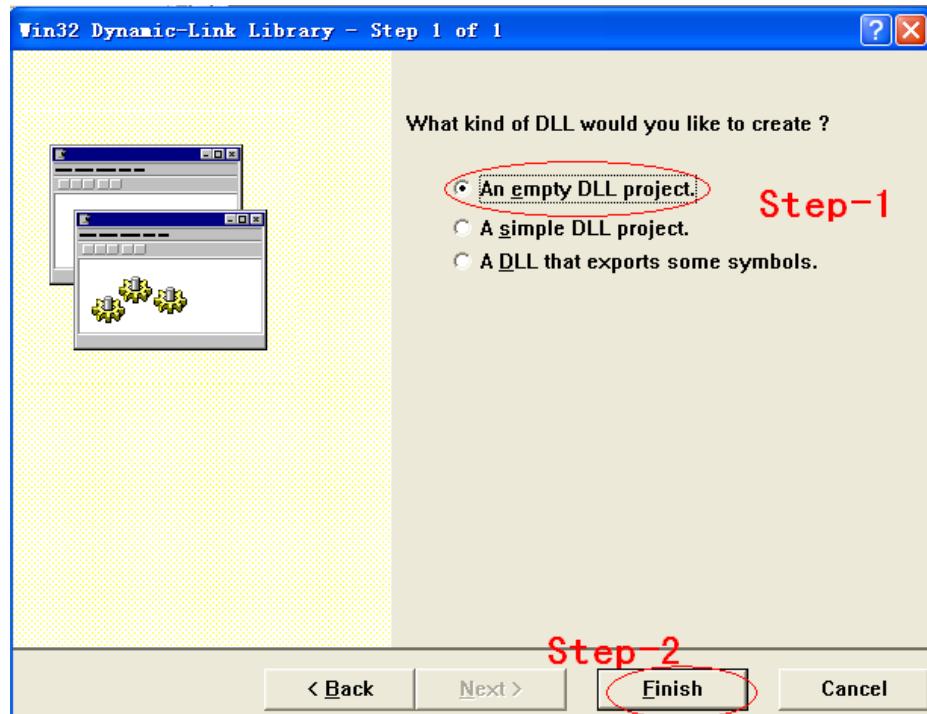


图 3.4 VC6 创建 Win32 DLL 工程 (Step-2)

- ② 点击 VC6 工具条上的 按钮 (如果找不到参考第 1 章), 打开图 3.5 所示的工具生成组件代码框架,

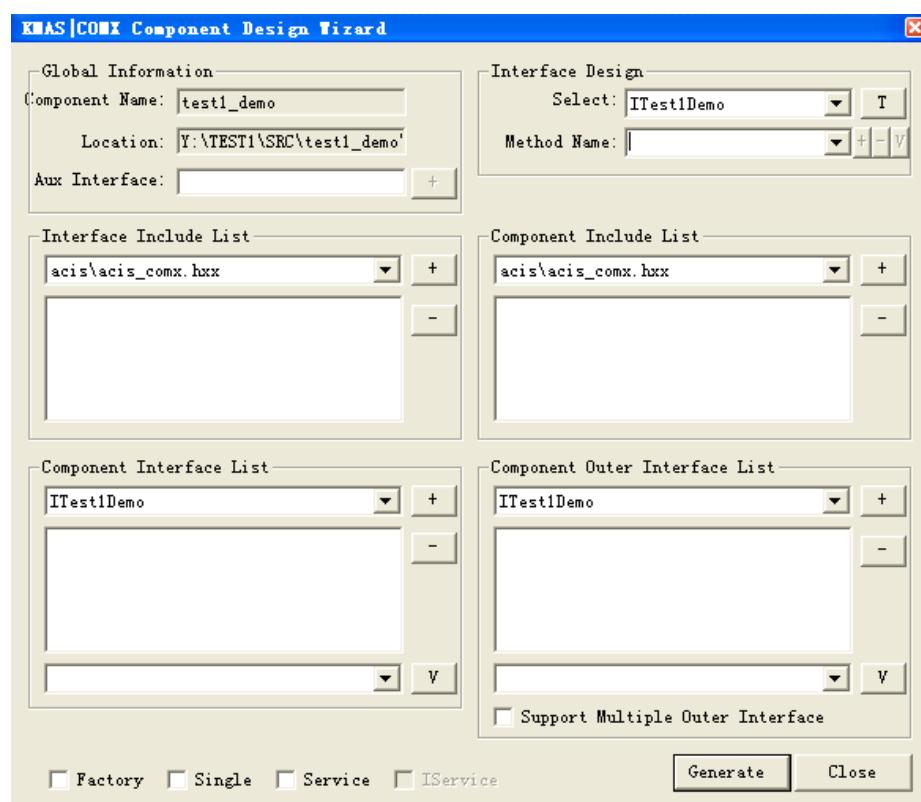


图 3.5 COMX 组件代码框架生成工具

- ③ 步骤②中的接口和出接口定义和指定完毕，点击  按钮之后就再 VC6 工程中自动产生了组件的 C++ 代码框架，一般只需实现\*\*\*\_impl.hxx 文件中定义好的函数体即可。
- ④ 如果组件代码框架生成后需要添加新接口、新出接口、新方法或者是接口升级则需要手工加入代码，具体细节参考下一章中的相关内容。

## 3-6. 组件的协作和拼接

组件可以在运行时刻采用 connect 和 disconnect 进行动态拼接和拆开，具体细节参看下一章，COMX 中也封装了类 TConnectManager 可以简化前述的函数调用。

## 3-7. 组件的配置脚本

每个 COMX 下的 unit 都有两个配置文件（均为 XML 格式）：**【unit\_name】.kproduct** 和 script 子目录下的 package.xml 文件。

**Kproduct 文件**主要用于设置 COMX 软件包的脚本搜索路径、数据文件搜索路径和组件文件的搜索路径，同时该文件也负责 COMX 软件包的打包文件解析。该文件中的一些主要配置代码片段列举如下：

```

<config>
  <path>
    <home_path>"y:/onestep_ui/"</home_path>
    <root_path>"y:/comx_sdk/"</root_path>
  <script_paths>
    <item>"~/script/"</item>
  </script_paths>
  <data_paths>
    <item>"~/data/"</item>
  </data_paths>
  <module_path>
    <item>"~/module/"</item>
  </module_path>
</config>
```

```

    </module_path>
    </path>
    <package>"package.xml"</package>
    </config>

```

其中 <home\_path> 和 <root\_path> 用于简化配置文件中的文件路径，比如：

"y:/comx\_sdk/lib/base/" 可简化为 "/lib/base/"，"y:/onestep\_-ui/module/" 可简化为 "~/module/"；<script\_paths> 用于指定脚本文件的搜索路径；<data\_paths> 用于指定软件包中的数据文件的搜索路径；<module\_path> 用于指定 kpi 文件的搜索路径。<package> 用于指定软件包的主脚本文件，该文件从 <script\_paths> 条目中所指定的脚本文件路径列表中搜索得到。

#### *package.xml* 文件中的一些主要配置代码片段列举如下：

```

<addProperties>
    <addPropertyItem name = "Data" clsid = "CLSID_IFemCore" iid = "IID_IFemCore" hide = "no"/>
</addProperties>

```

该代码片段用于在脚本状态下动态创建 COMX 组件对象。

```

<reduceProperties>
    <reducePropertyItem name="fem:render:content" from="fem:render" iid="IID_IG1ContentRender" hide="yes" />
</reduceProperties>

```

该代码片段用于在脚本状态下动态接口查询。

```

<addServices>
    <addServiceItem name="gl:content:service" clsid="CLSID_IG1Content" iid="IID_IG1ContentConnector" />
</addServices>

```

该代码片段用于在脚本状态下动态查询服务型组件接口。

```

<addConnects>
    <addConnectItem src = "Origin:Render" target = "Origin:Data"/>
</addConnects>

```

该代码片段用于在脚本状态下动态建立 COMX 组件间的出接口连接。

**注意：**

- ① name = "Data" 属性所表示的组件对象指针在 C++ 代码中可以用如下方式访问：

```

IFemCore *p_fem_core = NULL;
COMX_GetPackageProp("fem:data", (void**)&p_fem_core);

p_fem_core->Clear();
p_fem_core->Clean();

p_fem_core->Release();

```

- ② 在 package.xml 中预定义一个有限元网格的数据结构 “fem:data” 和一个基于 ACIS 的 CAD 数据结构组件 “acis:data” ,并在其中为它们已定义好了相关的连接（出接口设置），只要打开指定格式的文件，就可以在主程序的 OpenGL 显示区中显示，这些数据结构可以通过①中的方式在 C++下进行访问。
- ③ 如果您想要在脚本中预定义您自己的组件实例，并做好出接口的配置连接，可以模仿 package.xml 预定义的数据结构类比者进行。

### 3-8. 系统预定义组件集简介

COMX 预定义了一系列 CAD/CAE 相关的组件，如下表所示，这些组件的使用手册将在本手册的下一个版本中的到补充：

子文件夹名称	描述
acis	ACIS 相关的 CAD 组件集
base	类工厂等核心组件集
dlut_mesher	关振群教授的网格生成器组件化封装
fem	有限元相关组件集
gl	OpenGL 相关组件集
math	数学组件集
mesh	网格生成器组件集
post_ui	后处理组件集
ui	GUI 相关组件集
utils	工具组件集
xml	XML 处理相关组件集

## 第4章. 开发实例

### 4-1. 向导 GUI 程序

**目标：**完成如图 4.1 和图 4.2 所示的向导方式的 GUI 程序。

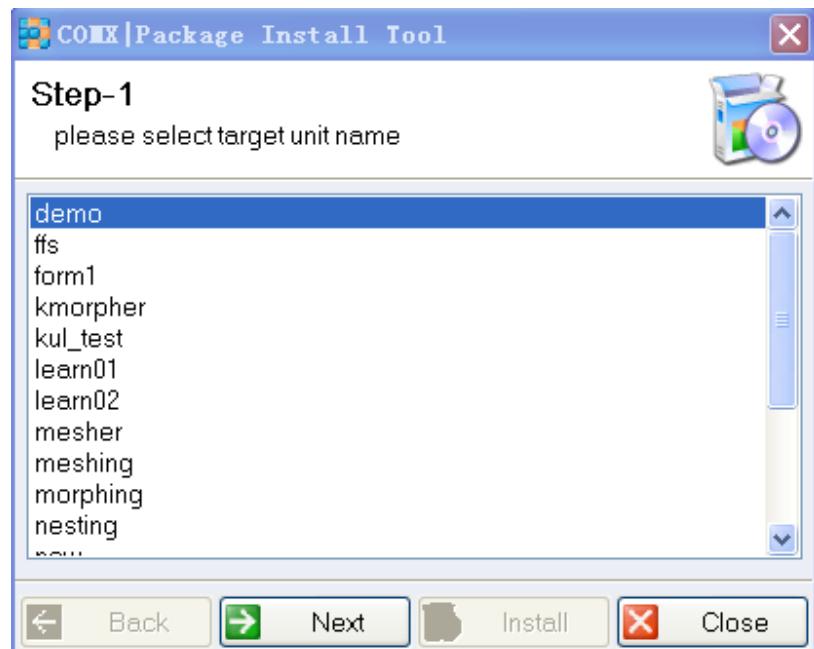


图 4.1 Step-1 的 GUI

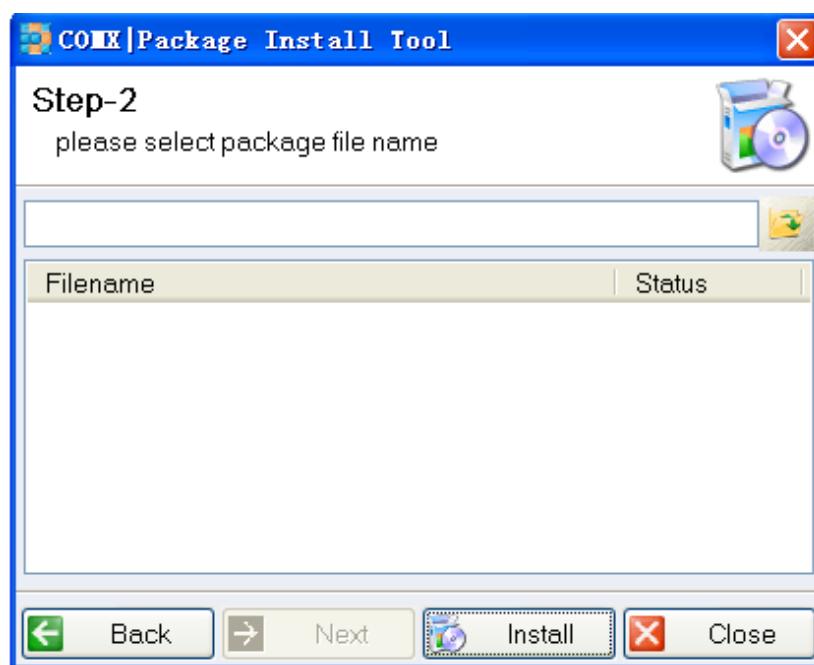


图 4.2 Step-2 的 GUI

界面布局分析：这一步可以在纸上绘制草图描述 KUL 文件的布局，如图 4.3、图 4.4 所示

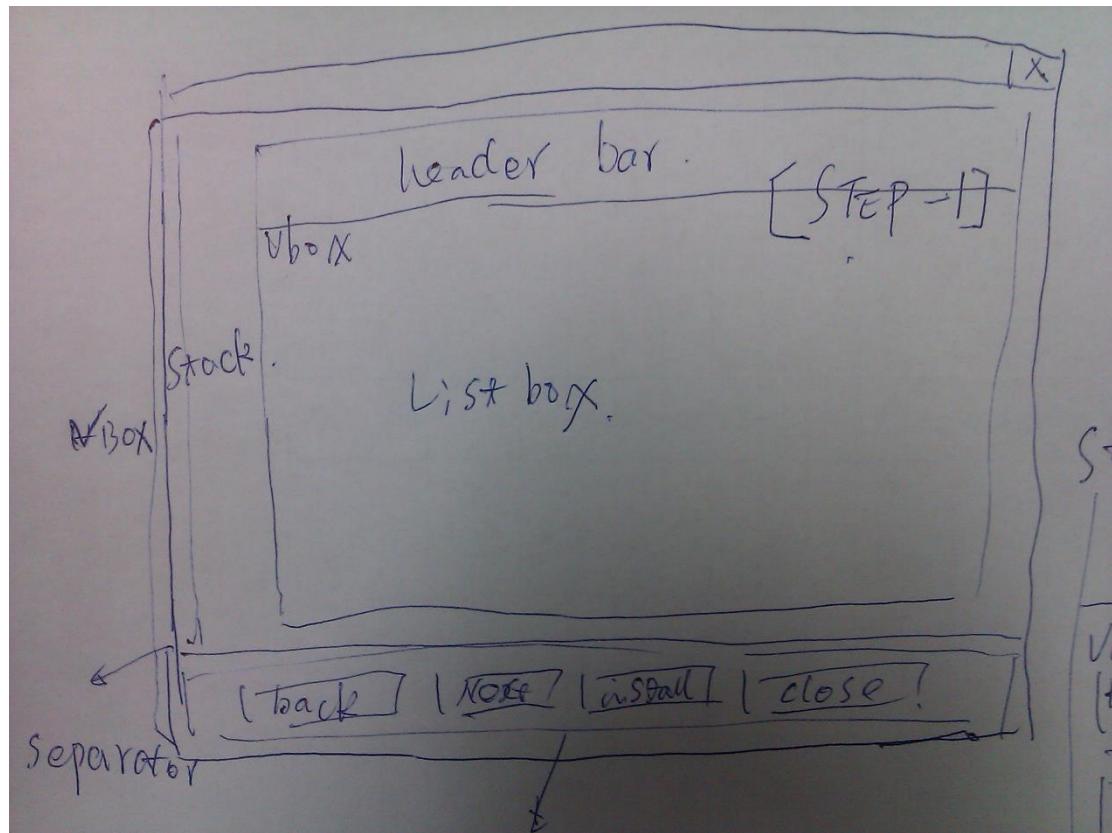


图 4.3 GUI 界面布局分析(Step-1)

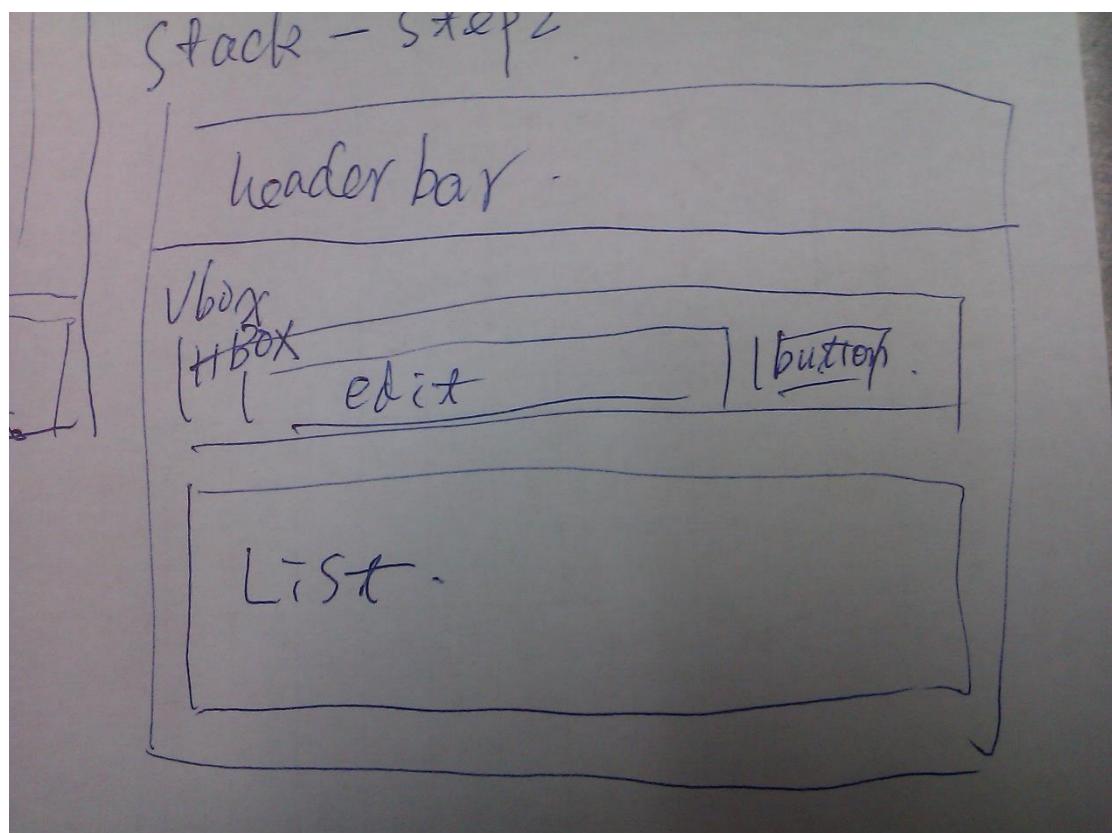


图 4.4 GUI 界面布局分析 (Step-2)

**代码实现:** 接下来需要在【COMX|KUL\$JS Design】工具中实现相关 KUL 和 JS 文件的编辑，编辑好的代码如下：

**Kul 文件** (这些代码可以通过“Add Widget”，“Simple Mode”和“Full Mode”自动添加、调整和修改，不需要大量的手工输入)：[Y:/comx\\_sdk/kul/package\\_generate.kul 的一部分](Y:/comx_sdk/kul/package_generate.kul)

```
<?xml version="1.0" encoding="gb2312"?>

<kul_pkg type="ui" name="pkg_generate" xmlns="http://www.kingmesh.com"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.kingmesh.com
  kul.xsd">

  <property name="id">0x92ef8143-0x921f-0x46d1-0x8d-0xb2-0xb6-0xff-0x8f-0x2d-0xef-0x2e</property>

  <widget type="form" name="pkg_install_toolkit">

    <property
      name="id">0xd2c848e3-0xb906-0x425e-0x9a-0xa9-0x50-0x94-0xda-0x11-0x26-0x35</property>

    <property name="title">COMX|Package Install Tool</property>

    <property name="center">center</property>

    <property name="toolwindow">false</property>

    <property name="icon">kmas</property>

    <property name="font_size">10</property>

    <property name="margin">0</property>

    <property name="app_window">true</property>

    <property name="javascript">pkg_install_toolkit.js</property>

    <widget type="vbox">

      <property name="margin">0</property>

      <widget type="stack">

        <property name="margin">0</property>

        <property name="adjust">horizontal</property>

        <property name="index1_bind">stack_index</property>

        <property name="id">2010</property>

        <property name="border">none</property>

      <widget type="vbox">
```

```
<property name="margin">0</property>

<widget type="headerbar">

    <property name="title">Step-1</property>

    <property name="detail">please select target unit name</property>

    <property name="bitmap">setup.bmp</property>

    <property name="bitmap_align">right</property>

    <property name="id">2036</property>

</widget>

<widget type="separator">

    <property name="id">2037</property>

    <property name="margin">0</property>

</widget>

<widget type="vbox">

    <property name="margin">3</property>

    <widget type="listbox">

        <property name="id">2038</property>

        <property name="height">70</property>

        <property name="adjust">auto</property>

        <property name="simple_bind">unit_name</property>

        <property name="list_bind">unit_list</property>

    </widget>

</widget>

<widget type="vbox">

    <property name="margin">0</property>

    <widget type="headerbar">

        <property name="title">Step-2</property>

        <property name="detail">please select package file name</property>

        <property name="bitmap">setup.bmp</property>

        <property name="bitmap_align">right</property>
```

```
<property name="id">2087</property>

</widget>

<widget type="separator">

<property name="id">2088</property>

<property name="margin">0</property>

</widget>

<widget type="vbox">

<property name="margin">2</property>

<widget type="hbox">

<property name="margin">0</property>

<widget type="edit" name="text">

<property name="id">2089</property>

<property name="adjust">horizontal</property>

<property name="readonly">true</property>

<property name="simple_bind">pkg_fpath</property>

</widget>

<widget type="pushbutton">

<property name="label" />

<property name="bitmap">open.bmp</property>

<property name="id">2090</property>

<property name="width">14</property>

<property name="shadow">true</property>

<property name="shadow_type">metal</property>

<property name="onClick">OnOpenPkg();</property>

</widget>

</widget>

<widget type="list">

<property name="id">2091</property>

<property name="height">80</property>

<property name="adjust">auto</property>
```

```
<property name="type">report</property>

<property name="table_bind">conflict_table</property>

<property name="data">

    <header>

        <item label="Filename" width="300" />

        <item label="Status" width="95" />

    </header>

    <image_list>pass,block</image_list>

    <content>0,abc.kul,okay,1,test.kul,conflict</content>

</property>

</widget>

</widget>

</widget>

<widget type="separator">

    <property name="id">2013</property>

</widget>

<widget type="hbox">

    <widget type="pushbutton">

        <property name="label">Back</property>

        <property name="bitmap">back.bmp</property>

        <property name="id">2014</property>

        <property name="onClick">OnBack();</property>

        <property name="enable_bind">back_btn_enable</property>

    </widget>

    <widget type="pushbutton">

        <property name="label">Next</property>

        <property name="bitmap">next.bmp</property>

        <property name="id">2015</property>

        <property name="onClick">OnNext();</property>

    </widget>
```

```

<property name="enable_bind">next_btn_enable</property>

</widget>

<widget type="pushbutton">

<property name="label">Install</property>

<property name="bitmap">setup.bmp</property>

<property name="id">2016</property>

<property name="onClick">OnInstall();</property>

<property name="enable_bind">install_btn_enable</property>

</widget>

<widget type="pushbutton">

<property name="label">Close</property>

<property name="bitmap">close.bmp</property>

<property name="id">2017</property>

<property name="onClick">pane.Close();</property>

</widget>

</widget>

</kul_pkg>

```

**注意：为了保证 headerbar 能够和对话框边界有效融合，要把一些“margin”的属性值调整为 0，见实例代码中被涂成红色的部分**

**绿色代码是 list 控件的初始化实例（比如表头的设置，图标的设置，默认数据设置等）**

**JS 文件（手工输入）：** [Y:/comx\\_sdk/js/pkg\\_install\\_toolkit.js](Y:/comx_sdk/js/pkg_install_toolkit.js)

```

function OnInitializeData()

{
    binding.back_btn_enable = false;

    binding.install_btn_enable = false;

    binding.unit_list = comx.sys.GetUnitNameList(false); //初始化 step-1 中的 unit 列表控件
}

```

```
function OnNext()
{
    binding.stack_index += 1; //Stack Widget 的 “index1_bind” 类型控件变量的加、减可以实现其中子页面
    //的切换，直观而简单

    binding.back_btn_enable = true; //处理按钮的可操作状态，类似代码不再重复描述
    binding.install_btn_enable = true;

    binding.next_btn_enable = false;

    binding.conflict_table = "";

    if(binding.pkg_fpath != "")
        CheckConflict();
}

function OnBack()
{
    binding.stack_index -= 1; //Stack Widget 的 “index1_bind” 类型控件变量的加、减可以实现其中子页面
    //的切换，直观而简单

    binding.back_btn_enable = false;
    binding.install_btn_enable = false;

    binding.next_btn_enable = true;
}

function OnOpenPkg() //Package 文件打开操作
{
```

```

var filter = new Array();
filter[0] = "Package Files (*.pkg)";
filter[1] = "All Files (*.*)";

var filename = comx.ui.CallOpenFileDialog(filter);

if(filename != "")
    binding.pkg_fpath = filename;//类似这样 binding.pkg_fpath 在 binding 命名空间下的变量都是在 KUL 中预定义好的。

CheckConflict();

}

function CheckConflict()
{
    var check_list = comx.package.ConfilictCheck(binding.unit_name, binding.pkg_fpath);// comx.package 是一个用于进行插件打包和安装的 JS.Ext 扩展，后面不再重复说明注释。

    binding.conflict_table = check_list;
}

function IsConflictBetweenPkgAndUnit()
{
    var arr = binding.conflict_table.split(',');
    for(loop = 0; loop < arr.length; ++loop)
    {
        if(loop%3 == 0 && arr[loop] == "1")
            return true;
    }
}

```

```
return false;  
}  
  
function OnInstall()  
{  
    if(binding.pkg_fpath == "")  
    {  
        comx.ui.InformationBox("Package file path can't be empty!");  
        return;  
    }  
  
    var ret = true;  
  
    var is_conflict = IsConflictBetweenPkgAndUnit();  
  
    if(is_conflict)  
    {  
        ret = comx.ui.CallYesNoDialog("current package conflict with unit - \\" + binding.unit_name +  
        "\\",\\ndo you want to continue?", "COMX|Notify");  
  
        if(ret)  
        {  
            comx.package.InstallPackage(binding.unit_name, binding.pkg_fpath);  
            CheckConflict();  
        }  
    }  
}
```

## 4-2. Edit Widget 的有效性检查

在 COMX 的 Edit 控件中，可以通过下述三个属性的设置，完成输入数据的有效性检查，

```

<property name="valid_check">true</property>
<property name="valid_okay">.*</property>
<property name="valid_warning">.*</property>

```

**Valid\_check** 如果呗设置为 true，表示开启 Edit 控件的有效性检查机制，下面两个属性：**valid\_okay** 和 **valid\_warning** 生效，它们的属性值均为正则表达式；为方便起见，我们称 **valid\_okay** 的属性值为**条件1**，**valid\_warning** 的属性值为**条件2**，如果条件 1 和条件 2 均不满足 Edit 控件的背景为红色，表示数据错误；如果条件 2 满足但条件 1 不满足，Edit 控件背景为黄色，表示数据存在潜在错误，这是一个警告状态；如果两个条件均满足，Edit 控件背景色为绿色，表示其中的数据是合法的。

**Valid\_check** 如果呗设置为 true，**<property name="valid\_bind">【valid\_val】</property>** 则会起作用，红色、黄色状态下其值为“false”，绿色状态下其值为“true”。

以 learn01 视频教程中的计算器为例，我们把第一个 Edit 的上述三个属性改为如下形式：

```

<property name="valid_check">true</property>
<property name="valid_okay">^[1-9]{2}$</property>
<property name="valid_warning">^[0]{1}[1-9]{1}$</property>

```

这样 Edit 控件只接受两位整数，如果为两位整数但首字符为“0”则为警告状态，效果如下面的图所示：



图 4.5 Edit 控件的有效性检查

正则表达的简单实用其实并不难，看一参考【COMX\_SDK 安装目录】\manual 下的《正则表达式 30 分钟入门教程.pdf》文档。

### 4-3. List 控件的插入删除操作

在 COMX 平台中 List 控件插入删除行操作主要涉及到 Javascript 中的字符串操作，考虑到相关操作比较简单没有在框架中封装相关操作，此处作简要说明，如图 4.6 所示的 list 控件总共有 3 列，通过 2-4 节中的 binding 机制可以通过'table\_bind'属性把 list 控件中的内容

和 binding.list\_table 变量关联起来，这时候 list 控件中对于行的插入删除操作其实就是 Javascript 中的一些字符串操作技巧：

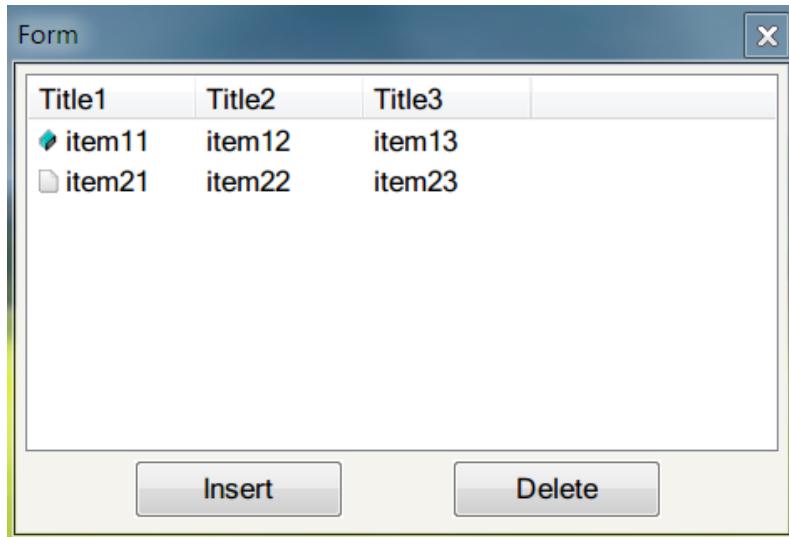


图 4.6 list 控件示例

上述 list 控件的字符串内容为：“0,item11,item12,item13,1,item21,item22,item23”，注意这里的 0、1 两个数字表示每一行的图标，相关图表在 kul 文件中通过 list 控件的“<image\_list>var,new,kmas</image\_list>”属性指定。

上述控件的插入删除代码如下（在末尾插入行、删除当前行）：

```

function OnInsert()
{
    InsertRow(binding.list_table, 0, "-,-,-", function(table){
        binding.list_table = table;
    });
}

function InsertRow(table, img_index, row, callback)
{
    if(table)
    {
        table += ',';
    }

    table += (img_index + ',' + row);
    if(callback)
    {
        callback(table);
    }
}

return table;

```

```

}

function OnDelete()
{
    DeleteRow(binding.list_table, 3, binding.list_index.index1, function(table){
        binding.list_table = table;
    });
}

function DeleteRow(table, column_num, row_index, callback)
{
    var row_size = column_num + 1;
    var ret = [];

    table = table.split(',');
    for(var loop = 0; loop < table.length / row_size; ++loop)
    {
        if(loop == row_index)
        {
            continue;
        }

        for(var loop_sub = 0; loop_sub < row_size; ++loop_sub)
        {
            ret.push(table[loop * row_size + loop_sub]);
        }
    }

    ret = ret.join(',');
}

if(callback)
{
    callback(ret);
}

return ret;
}

```

注意：上述代码返回值得方式有回掉函数和返回值两种，一般推荐使用回掉函数方式，正如 2-12 节中所讲述的那样这几乎是成熟的 Javascript 模块化的标准写法。

相关代码可以在 [【learn01 unit】中的 list\\_demo 窗体](#) 中找到。

## 4-4. 基于 Bitmap 的 gif 播放和等待进度条

在 COMX 中设计了两种 gif 播放机制，都是通过 Bitmap Widget 来完成的：

### 4-4.1. 基于图片模式的等待进度条播放

这种方式事实上是利用了 bitmap 控件的 simple\_bind 属性，比如该属性值为 gif\_prog，那么就可以通过如下方式切换控件中显示的图片：

```
binding.gif_prog = "y:/a.bmp";
```

这时候 bitmap 控件中建议采用如表 4.1 所示的属性设置：

表 4.1 bitmap 控件采用图片方式播放 gif 的推荐属性设置

属性名称	取值
fit_origin_size	<b>false</b>
transparent_background	<b>true</b>
bkcolor	和 Form 的 bkcolor 取值相同
picture	<b>nil.bmp</b>

接下来就可使用下述代码在 bitmap 中播放 gif 动画，gif 会被按比例缩放到 bitmap 控件中：

```
var player = require ('class/gif_animate.js');
...
var nID = player.Play ('progress.gif', 50, function (fname) {
    binding.prog_bmp = fname;
});
...
player.Stop (nID);
...
```

一般这种方式用于显示如图 4.7 所示的等待进度条，其特点在于播放速度可通过 Play 函数总的第二个参数设置时间间隔来完成，所播放的 gif 文件在 comx\_sdk 或 unit 的 picture 文件夹下，播放时会在该文件夹下的 cache 子目录解压 gif 文件。

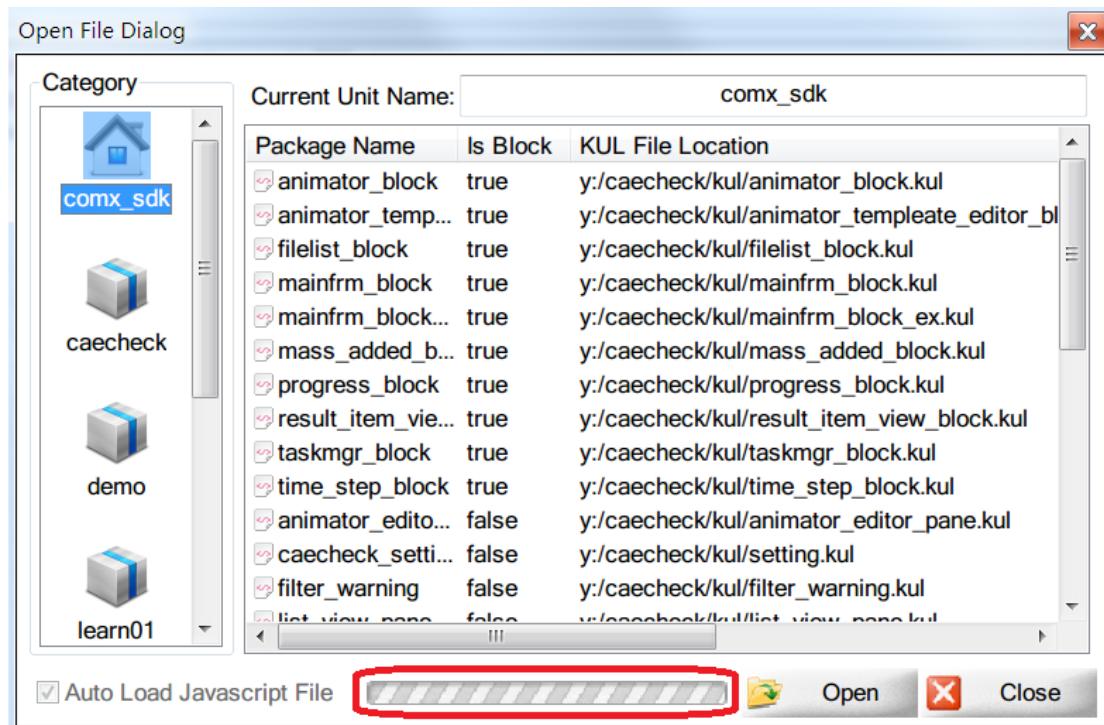


图 4.7 等待进度条示意

#### 4-4.2. 基于 gif 模式的大尺寸动画播放

另外，在 bitmap widget 还提供对于 gif 的原生支持，即不通过 4-4.1 节中的 gif 解码播放器切换图片也能播放 gif 图片，所不同的是这种原生方式不能调整动画帧的切换时间； bitmap 控件中和 gif 有关的属性如表 4.2 所示：

表 4.2 widget 控件和 gif 有关属性

属性名称	取值
<b>picture</b>	如果该属性指定成一个有效的 gif 文件，则控件自动切换为 gif 模式
<b>autoPlay</b>	布尔类型，指定是否自动播放 gif
<b>playMode</b>	copy：按原有大小播放 gif，不缩放； zoom：缩放铺满整个屏幕； scale：保持长宽比例缩放；
<b>simple_bind</b>	智能作为左值，可动态切换 gif；如果赋值为“play”开始播放，“stop”停止播放，“”暂停

一般大尺寸且播放时间较长的屏幕录制视频采用这种方式播放，如果等待进度条不需要控制播放速度也可以采用这种方式播放。相关示例可参考 KEditor Learn 面板相关代码。

## 4-5. Scintilla 编辑控件

COMX 还封装了 scintilla 控件作为语法高亮编辑器，并且支持自动完成等特征，KEditor 开发工具本身就是该控件的最好示例，可以在该工具中打开其自身的 KUL 和 Javascript 文档进行仔细研读，如图 4.8 所示：

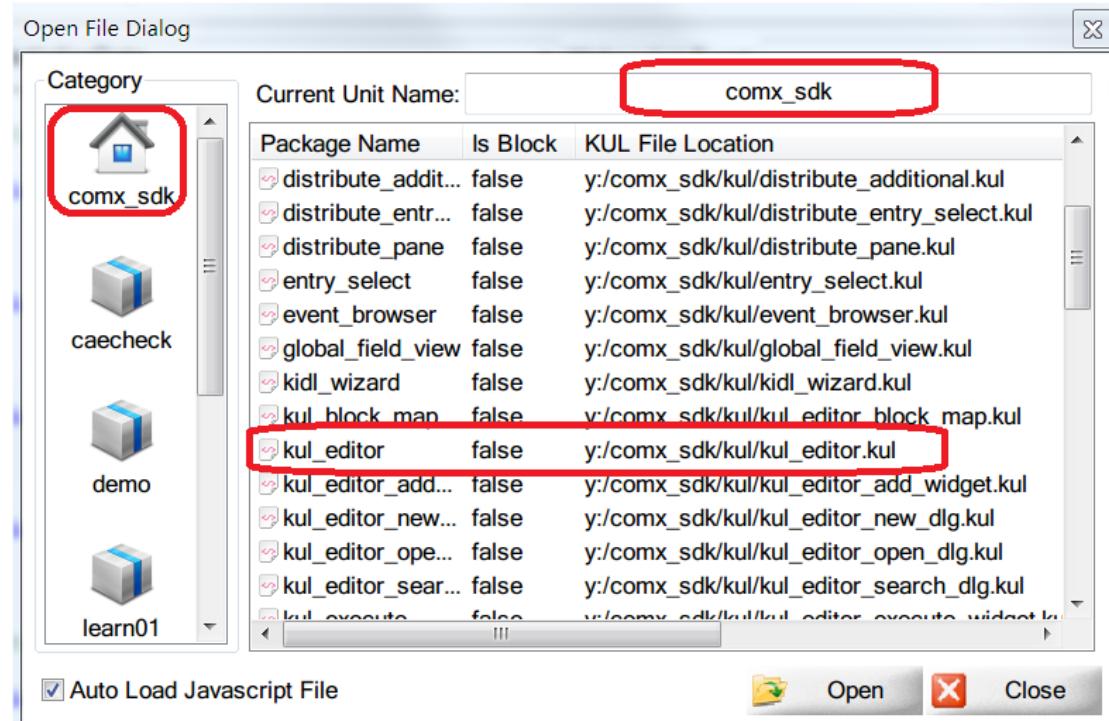


图 4.8 打开 KEditor 开发工具自身的源代码

# Part-II 进阶篇

## COMX 底层组件开发环境

# 第5章. COMX 基础

## 5-1. 什么是 COMX?

为了设计一个通用 CAE 基础软件开发平台我们选取微软 COM 技术中最核心的部分，设计和实施了一个面向多种操作系统（WINDOWS/LINUX/UNIX 等）的针对 CAE（CAD）软件开发的简化版本的 COM 内核，并在此基础上构建了基于“微内核+插件”的 CAE 软件开发框架；上述精简版本的跨平台 COM 内核被称为：A simplified Component Object Model for X Platforms（其中，X 代表 WINDOWS/LINUX/UNIX 等操作系统），简称为 COMX；并且，“微内核+插件”体系结构中的“内核”和“插件”都是以 COMX 组件形式进行构造的，因此将本文所涉及的基于“微内核+插件”的 CAE 基础软件开发平台称为 COMX 开发平台（简称为 *COMX 平台*）。

## 5-2. COMX 平台的特点

- (1) 从语言的角度上讲可以将 COMX 理解成一个更好的 C++。
- (2) COMX 是跨平台的。拥有基于 C++语言的可移植性—在不同平台上要单独编译。
- (3) 二进制级别上的可重用性（软件的模块化的到强制保证）。

相较于 C++而言这是一个巨大的进步，首先它避免了源码级别上重用的编译问题（我们在使用类库中经常见到，也是最头疼的问题之一）；软件的模块性通过组件得到强制保证，而 C++类库做不到这一点（取决于程序员的水平以及他是否自觉）；接口和出接口的使用保证了模块之间的关联尽可能少，尽最大程度保证“高内聚、低耦合”。

- (4) 可扩展性强（比如，组件接口的平滑升级特性）。

在 C++虽然也可勉强实现，但得不到语义级别上的强调和强制保证。

- (5) 采用“微内核 + 插件”的体系结构。

进一步强化了 COMX 平台的可扩展性，以及开放性。KMAS 自身软件开发方式，和提供给用户使用的二次开发方式是一致的，这种开发性更便于开发平台的普及。

- (6) 基于 XML 配置文件的软件打包机制。

软件是动态的、可配置的，提高了在 COMX 平台下开发软件时的灵活性。

- (7) 基于 KUL 的 GUI 组件化、脚本化机制。

这是实现 COMX 平台跨平台的保证，用户界面的跨平台问题也是实现 COMX 开发平台跨平台的过程中遇到的最大难题；同时，KUL 机制也极大简化了 COMX 平台下 GUI 界面的开发，使我们可以快速建立用户界面原型、快速进行界面调整，使开发者的经历更集中于算法。GUI 界面的开发只需写一个 KUL 脚本无需动用 MFC/WTL/GTK 等开发工具，大大降低 COMX 平台的使用门槛，这对 COMX 平台将来形成标准，并加以普及的重要保证。

### 5-3. COMX 平台的体系结构

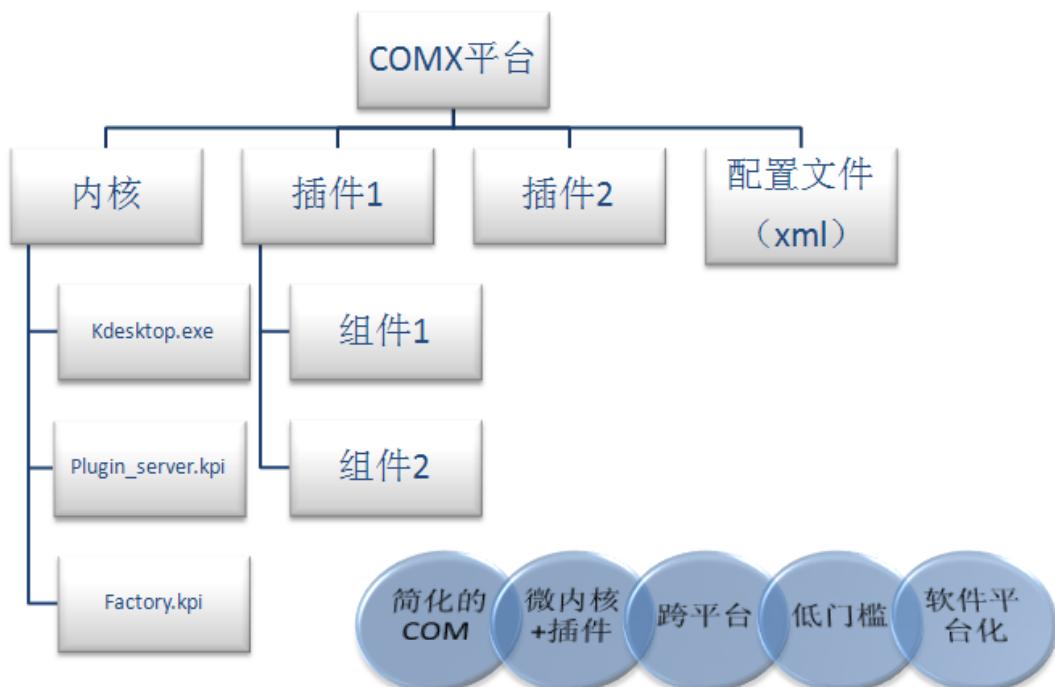


图 5.1 COMX 平台组件体系结构

### 5-4. COMX 中的基本概念

表 5.1 COMX 内核中的关键字列表

序号	关键字	简要描述	C++定义(VC++版本为例)
1	interface	在 COMX 内核中定义接口类型的关键字	#ifndef interface #define interface struct #endif /*interface*/
2	TStatus	定义组件接口中成员函数的返回值类型	#ifndef TStatus #define TStatus int #endif /*TStatus*/

3	STDCALL	接口成员函数的修饰符，用于指示参数出入栈的顺序和方法	<pre>#ifndef STDCALL #define STDCALL __stdcall #endif /* STDCALL */</pre>
4	MUID/MIID/ MREFIID	用于定义 COMX 中的接口、出接口和组件对象的全局唯一标识符类型	<pre>#ifndef MUID_DEFINED #define MUID_DEFINED  typedef struct _MUID {     unsigned long Data1;     unsigned short Data2;     unsigned short Data3;     unsigned char Data4[8]; } MUID;  typedef MUID MIID; typedef MUID MTYPEID; #define MREFEID const MUID &amp; #endif /* MUID_DEFINED */</pre>
5	KPI	COMX 组件动态链接库的扩展名(全称为：KMAS Program Interface)	—
6	DLL_API	KPI 文件的导出入口函数标识符	<pre>#ifdef DLL_EXPORTS #define DLL_API __declspec(dllexport) #else #define DLL_API __declspec(dllimport) #endif</pre>
7	IID	接口的全局性唯一标识符，其类型为 MUID	—
8	CLSID	组件的全局性唯一标识符，其类型为 MUID	—
9	IRoot	COMX 中所有其它接口的根接口	<pre>//{97159EFA-DD23-41a0-B7F2-1BC893380313} const MIID IID_IRoot = { 0x97159efa, 0xdd23, 0x41a0, { 0xb7, 0xf2, 0x1b, 0xc8, 0x93, 0x38, 0x3, 0x13 } };  interface IRoot{     virtual TStatus STDCALL QueryInterface (MREFIID riid, void **ppv) = 0;     virtual TStatus STDCALL AddRef() = 0;     virtual TStatus STDCALL Release() = 0; };</pre>
10	IEventSource	用于支持 COMX 中	<pre>interface IEventSource : public IRoot</pre>

		出接口机制的基本接口	<pre>{     virtual TStatus STDMETHODCALLTYPE GetEventSourceInterface(MIID *pIID) = 0;     virtual TStatus STDMETHODCALLTYPE GetEventSourceContainer(IEventSourceContainer **ppESC) = 0;     virtual TStatus STDMETHODCALLTYPE Advise(IRoot *pRoot, DWORD *pwdCookie) = 0;     virtual TStatus STDMETHODCALLTYPE Unadvise(DWORD dwCookie) = 0; };</pre>
11	connect、 disconnect	用于实现 COMX 中出接口机制的动态连接工具函数	内容太多，此处从略。本节后面的内容中将进行详述。
12	IAggregation	用于支持 COMX 组件聚合机制的基本接口	<pre>// {C9540745-9C35-4b24-970D-D211B34962E8} const MUID IID_IAggregation = { 0xc9540745, 0x9c35, 0x4b24, { 0x97, 0xd, 0xd2, 0x11, 0xb3, 0x49, 0x62, 0xe8 } };  interface IAggregation : public IRoot {     virtual TStatus STDMETHODCALLTYPE QueryAggregation(MREFIID riid, void **ppv) = 0; };</pre>

### (1) COMX 中的接口概念。

把接口和实现分离的动机，是把对象内部的细节（对组件的客户而言）都隐藏起来。这条基本原则提供了一层间接性，允许实现类内部的数据成员的数量和顺序都可以发生变化，但客户程序无需重新编译。这条原则也允许客户可以在运行时询问对象以便发现对象的扩展功能。最重要的一点是，接口和实现分离的原则也为二进制级别的组件复用提供了一个普遍的基础。由此可以看出，接口是 COMX 内核中最重要、最为基础性的概念。

首先，什么是接口呢？接口（Interface）是 COMX 技术中最精髓的概念，它主要用来定义一种程序的协定。实现接口的类或者结构要与接口的定义严格一致。有了这个协定，就可以抛开编程语言的限制（理论上）。接口可以从多个基接口继承，而类或结构可以实现多个接口。接口可以包含方法、属性、事件和索引器。接口本身不提供它所定义的成员的实现。接口只指定实现该接口的类或接口必须提供的成员。接口好比一种模版，这种模版定义了对象必须实现的方法，其目的就是让这些方法可以作为接口实例被引用。接口不能被实例化。类可以实现多个接口并且通过这些实现的接口被索引。接口变量只能索引实现该接口的

类的实例。

其次，**接口和组件有什么关系呢？** 接口描述了组件对外提供的服务。在组件和组件之间、组件和客户之间都通过接口进行交互。组件一旦发布，它只能通过预先定义的接口来提供合理的、一致的服务，这种接口定义之间的稳定性使客户应用开发者能够构造出坚固的应用。

一个组件可以实现多个组件接口，而一个特定的组件接口也可以被多个组件来实现。组件接口必须是能够自我描述的，这意味着组件接口应该不依赖于具体的实现，将实现和接口分离彻底消除了接口的使用者和接口的实现者之间的耦合关系，增强了信息的封装程度。

当一个组件需要提供新的服务时，可以通过增加新的接口来实现。不会影响原接口已存在的客户。而新的客户可以重新选择新的接口来获得服务。

**第三，使用接口有什么好处呢？** 接口概念在 COMX 技术中的使用赋予了软件模块（即本文中的 COMX 组件）下列一些属性：语言无关性（或语言环境的独立性，language independence）、版本升级的鲁棒性（即稳健型，robust versioning）、位置的独立性（location transparency）和面向对象特性（object orientation）。

**第四，怎样定义接口，COMX 中的接口在 C++ 语法层面上的本质是什么？** 在 C++ 语法层面上接口本质上就是一个抽象基类，当然其中所有的方法都是公开的（public）。我们首先定义了一个关键字 interface 用于对接口的标识，其定义形式如下：

```
#ifndef interface
#define interface struct
#endif /*interface*/
```

为什么将 interface 定义成 struct 呢？在 C++ 语法中 struct 和 class 是等价的，所不同的是，struct 中的所有成员函数默认都是 public 的。而接口中所有的方法也都必须是公开的（public），所以此处将 COMX 中的 interface 关键字定义成 struct。

另外，每一个接口都有一个全局性的唯一标识符（关于全局性唯一标识符的概念，在后文中将有更详尽的讨论），其命名规则为“IID\_接口名称”，比如对于接口 IFemCore 而言，它的唯一性全局标识符名字必须为 IID\_IFemCore。下面我们看一个 COMX 中接口定义的例子：

```
// {0E962D73-B070-418b-ADCF-7774A398585E}
```

```
const MUID IID_IFemRender =
```

```
{ 0xe962d73, 0xb070, 0x418b, { 0xad, 0xcf, 0x77, 0x74, 0xa3, 0x98, 0x58, 0x5e } };
```

```
interface IFemRender : public IRoot{
    virtual TStatus STDCALL SetDisplayMode(const int &mode) = 0;
    virtual TStatus STDCALL GetDisplayMode(int &mode) = 0;
    virtual TStatus STDCALL SetVisualMode(const int &mode) = 0;
    virtual TStatus STDCALL GetVisualMode(int &mode) = 0;
    .....
};
```

这里需特别指出如下几点：a) 接口中所有的成员函数都必须为纯虚函数。b) 所有接口函数的返回值都必须为 `TStatus` 类型，并且用 `STDCALL` 关键字修饰。c) 接口函数中的参数类型必须为 C++ 语言中的原生数据类型或简单类型（即 `int`, `double`, `float`, `char`, 结构体类型 `struct`），当然也包含这些类型的指针和引用类型，有时根据设计上的需要，接口类型也可出现在其它接口的参数列表中，但一般并不推荐这种用法。d) 所有的接口都必须直接或间接地派生于根接口 `IRoot`。

COMX 中有一个最重要的接口，那就是 `IRoot`（根接口），该接口相当于 COM 技术中的 `IUnknown` 接口，COMX 中的所有其他接口都必须从 `IRoot` 接口中直接或间接派生出来。`IRoot` 接口提供了两个非常重要的特性：`生存期控制` 和 `接口查询`。客户程序（可执行程序或其它组件）只能通过接口与 COMX 对象进行通信，虽然客户程序可以不管对象内部的实现细节，但它要控制对象的存在与否。如果客户还要继续对对象进行操作，则它必须保证对象能一直存在于内存中；如果客户对对象的操作已经完成，以后也不再需要该对象了，则它必须及时的把对象释放掉，以提高资源的利用率。`IRoot` 引入了“引用记数”方法可以有效的控制对象的生存周期。另一方面，如果一个 COM 对象实现了多个接口，在初始时刻，客户程序不太可能得到该对象的所有接口指针（即使在程序上能做到这一点，也会使代码变得非常啰嗦），它只会拥有一个接口指针。如果客户程序需要其它指针，那么它如何获取这些接口的指针呢？`IRoot` 使用了“接口查询”方法完成接口之间的跳转。另外，“接口查询”也是接口平滑升级的一个重要支撑元素。

首先我们来看一下 `IRoot` 的定义：

```

// {97159EFA-DD23-41a0-B7F2-1BC893380313}

const IID IID_IRoot =
{ 0x97159efa, 0xdd23, 0x41a0, { 0xb7, 0xf2, 0x1b, 0xc8, 0x93, 0x38, 0x3, 0x13 } };

interface IRoot
{
    virtual TStatus STDMETHODCALLTYPE QueryInterface(MREFIID riid, void **ppv) = 0;
    virtual TStatus STDMETHODCALLTYPE AddRef(void) = 0;
    virtual TStatus STDMETHODCALLTYPE Release(void) = 0;
};

```

IRoot 包含了三种成员函数:QueryInterface、AddRef 和 Release。函数 QueryInterface 用于查询 COM 对象的其他接口指针，函数 AddRef 和 Release 用于对引用记数进行操作。

具体在组件的实现方面，我们为 IRoot 定义了一组宏，用于在实现 COMX 组件时封装 IRoot 接口的实现代码，代码举例如下：

```

BEGIN_IMPLEMENT_ROOT() //指示开始IRoot接口实现编码

IMPLEMENT_INTERFACE(IFemCore) //将IFemCore接口添加到组件的接口查询列表中

IMPLEMENT_INTERFACE_ROOT(IFemCore) //将IRoot接口添加到组件的接口查询列表中

END_IMPLEMENT_ROOT() //指示结束IRoot接口实现编码

```

## (2) COMX 中的全局性唯一标识符。

在分布式对象 (distributed-object) 和基于组件 (component-based) 的环境里，组件和它们的接口的独有标识符是非常重要的。在微软的 COM 技术中，对远程进程调用 (Remote Procedure Calls, RPC) 使用了一种特别的技术，该技术在分布式计算环境 (Distributed Computing Environment, DCE) 标准中进行了描述。该标准说明了一种被称为通用独有标识符 (Universally Unique Identifier) 的技术。在 COMX 的设计和实施中也使用了全局性唯一标识符的概念。

全局性唯一标识符 UUID 是一个独有的、128 位的、并且具有非常高可靠率的数值。它

把一个独有的网络地址（48 位）和一个非常精细的时间印鉴（100 纳秒）结合在一起。COMX 对 UUID 的实现被称为全局特有标识符 MUID（在 COM 中相应的名字为 GUID，COMX 中为了名称上的区别，而不至于在某些 WIN32 编程环境中产生语法歧义，将其命名为 MUID），它在结构上和 UUID 非常类似。COMX 中用 MUID 来识别组件的类（CLSID）、接口（IID）和出接口（EID）等基本元素。在实际组件开发过程中，使用 WIN32 上的 GUIDGEN.EXE 工具来生成 MUID，如图 5. 2 所示：

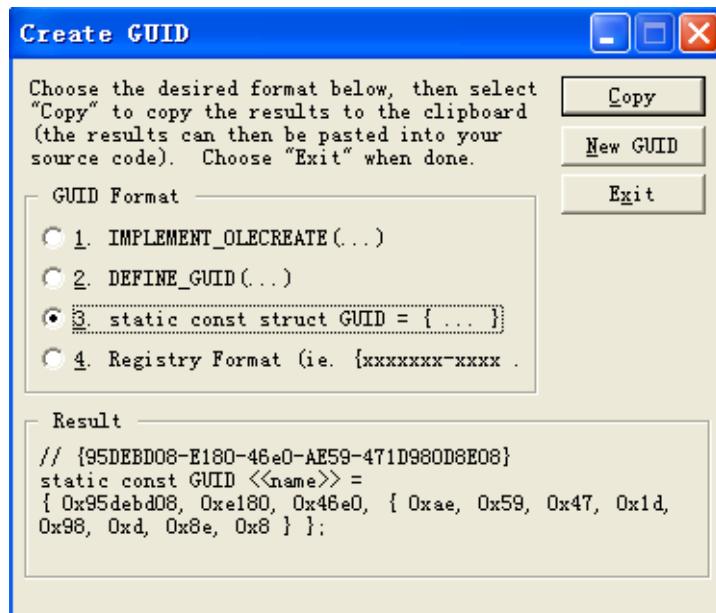


图 5. 2 WIN32 GUIDGEN.EXE SDK 工具的使用

### (3) COMX 中的出接口 (outgoing interface) 概念。

谈到 COMX 中出接口的概念，首先让我们来看一个程序上的例子：

```
TPoint TFemRender::GetCenter()
{
    TFactory factory;
    CREATE_COMX_OBJ(factory, CLSID_IG1WCS, IGLWCS, p_gl_wcs);

    TPoint pt;
    .....
    p_gl_wcs->Release();
```

```

    return pt;
}

```

这是某组件中的 GetCenter 接口函数的实现代码,从代码中我们可以看出,在 GetCenter 接口函数的实现过程中创建了另外一个组件对象 p\_gl\_wcs。也就是说,该接口函数乃至它所在的组件要依赖于一个全局唯一性标识符为 CLSID\_IG1WCS 的组件及其中的 IG1WCS 接口而存在,显然这为 GetCenter 接口函数所在组件的重用带来了障碍。如何降低这种依赖关系,提高组件的可重用性呢?我们来看一段改进后的代码:

```

TPoint TFemRender::GetCenter(IG1WCS *&p_gl_wcs)
{
    p_gl_wcs->AddRef();

    TPoint pt;

    .....

    p_gl_wcs->Release();

    return pt;
}

```

从改进后的代码我们可以看出,接口函数 GetCenter 的参数发生了改变,它使用了 IG1WCS 接口的指针作为它的参数,这样就消除了该接口函数所在组件和 CLSID\_IG1WCS 组件之间的依赖性,而接口之间的依赖性仍然存在,这对于可重用性的设计指标而言仍然是一个障碍。如何进一步降低接口的依赖性,采用一种规范化的方式管理组件之间的依赖关系呢?在 COMX 中我们使用了 COM 技术中的引出接口 (outgoing interface, 简称为出接口) 技术。

引出接口的概念是相对于引入接口 (incoming interface) 而言的。在前面的讨论中,术语接口 (interface) 并未全面真正描述接口的真正含义。在 COMX 中存在两种不同类型的接口,即引入接口 (incoming interface, COMX 中也简称为接口) 和引出接口 (outgoing

interface, COMX 中也简称为出接口)。一个引入接口是指由组件实现的接口；一个引出接口是指在组件的依赖说明文件中所描述的接口，一般而言它是由组件的客户程序实现的，描述了组件依赖于哪些接口而存在。客户程序必须获得组件对它的出接口的描述（在 COMX 中是通过使用一个 XML 格式的描述文件来完成的），并选择某种机制来实现它。

特别值得注意的一点：就一般情况而言，从一个特定的出接口角度上看，COMX 组件和它的客户程序（或组件）是一对一的关系。

在 COMX 中组件的出接口主要是通过对下面两个接口的支持而实现的，即 IEventSource 和 IEventSourceContainer，表 5. 2、表 5. 3 分别列出了上述接口的功能及含义。

表 5. 2 IEventSource 接口函数列表

接口函数名称	接口函数功能描述
GetEventSourceInterface	指示可以连接何种类型的出接口
GetEventSourceContainer	该出接口的宿主对象指针
Advise	拥有并使用 pRoot 参数指定的出接口对象，并返回其唯一性的连接标识 dwCookie
Unadvise	停止拥有和使用与 dwCookie 相关联的出接口对象指针

表 5. 3 IEventSourceContainer 接口函数列表

接口函数名称	接口函数功能描述
FindEventSource	获取 riid 参数指定的 IEventSource 实现

现在用于支持出接口的基本接口定义完成了，那在 COMX 组件实现时如何支持出接口呢？图 5. 3 描述了支持出接口的 COMX 组件的实现结构：

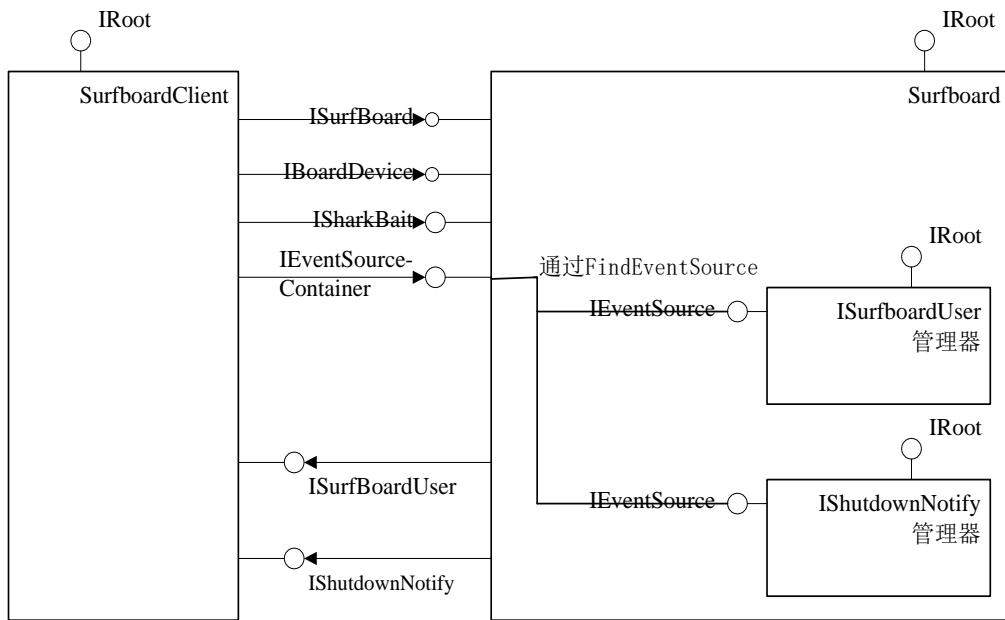


图 5.3 支持出接口的 COMX 组件实现结构

图 5.3 中所示的出接口组件结构在编码实现上非常复杂，并且很容易出错，但不同组件的用于支持出接口的代码在结构上非常类似，为了简化 COMX 组件的出接口编码工作，本文中定义了一组宏用于实现对 COMX 组件中出接口的支持，宏代码举例如下（此处以 fem\_render 组件中的出接口支持为例）：

```

BEGIN_IMPLEMENT_EVENT_SOURCE_CONTAINER() //指示开始IEventSourceContainer接口的实现
    IMPLEMENT_EVENT_SOURCE_CONTAINER(IFemCore) //将IFemCore加入出接口支持列表中
    IMPLEMENT_EVENT_SOURCE_CONTAINER(IFemRenderColor)
    IMPLEMENT_EVENT_SOURCE_CONTAINER(IGlMaterialConnector)
END_IMPLEMENT_EVENT_SOURCE_CONTAINER() //指示结束IEventSourceContainer接口的实现

IMPLEMENT_EVENT_SOURCE(TFemRender, IFemCore) //将IFemCore加入出接口支持列表中
IMPLEMENT_EVENT_SOURCE(TFemRender, IFemRenderColor)
IMPLEMENT_EVENT_SOURCE(TFemRender, IGlMaterialConnector)

```

出接口除了用于规范化地封装组件之间相互调用的依赖关系，还可用于组件之间的事件/消息触发机制，这一点很好理解，一旦某个组件触发（执行）了它的某一个出接口，就相当于它调用了通过该出接口与组件连接的客户程序（可执行程序或其它组件）代码，也可以理解为组件向它的客户程序发送了一个消息（事件）。既然我们采用出接口机制来实现组件之间的事件/消息触发机制，就不得不面对这样一个问题：有些时候，某个组件所触发的某个事件需要被多个客户端程序（或组件）响应，这就要求我们提供一种机制支持组件和它的客户程序（或组件）在一个特定的出接口上维护一对多的关系。在 COMX 中这种一对多的关

系通过 multiple 类型的出接口得到支持，用于相关实现的宏代码举例如下：

```
BEGIN_IMPLEMENT_EVENT_SOURCE_CONTAINER () //指示开始IEventSourceContainer接口的实现
    IMPLEMENT_EVENT_SOURCE_CONTAINER (IFemCore) //将IFemCore加入出接口支持列表中
    IMPLEMENT_EVENT_SOURCE_CONTAINER (IFemRenderColor)
    IMPLEMENT_EVENT_SOURCE_CONTAINER (IG1MaterialConnector)
END_IMPLEMENT_EVENT_SOURCE_CONTAINER () //指示结束IEventSourceContainer接口的实现

IMPLEMENT_EVENT_SOURCE_MULTI (TFemRender, IFemCore) //将IFemCore加入出接口支持列表中
IMPLEMENT_EVENT_SOURCE (TFemRender, IFemRenderColor)
IMPLEMENT_EVENT_SOURCE (TFemRender, IG1MaterialConnector)
```

这段代码使 `fem_render` 组件的 `IFemCore` 出接口成为 `multiple` 类型的，即一个 `fem_render` 组件对象可连接多个不同实例的 `IFemCore` 出接口。相关的代码实现也很简单，只是把宏 `IMPLEMENT_EVENT_SOURCE` 替换成 `IMPLEMENT_EVENT_SOURCE_MULTI` 即可。

那么组件如何从客户程序中获取出接口，也就是说，怎么把一个组件和它的客户程序关联在一起呢？在 COMX 中定义了两个基本的工具函数 `connect` 和 `disconnect` 来动态维护组件和它的客户程序（或组件）之间的关联，表 5. 4 对这两个函数进行了描述。

表 5. 4 用于维护组件出接口动态关联的函数列表

函数名称	描 述
connect	<p>函数原型：</p> <pre>template&lt;typename I&gt; void connect(IRoot *pRoot, I *pInterface, const MIID iid, DWORD &amp;rdwCookie)</pre> <p>功能描述：用于在组件及其客户端程序（或组件）之间建立关联，从而实例化组件的某个出接口。</p> <p>参数描述：pRoot是指向组件的根接口指针；pInterface表示连接到该组件的由客户程序（或组件）实现的出接口指针；iid是该种类型出接口的接口IID值；dwCookie是一个返回值，是该函数所建立的出接口连接的唯一性标识。</p>
disconnect	<p>函数原型：</p> <pre>void disconnect(IRoot *pRoot, const MIID &amp;iid, DWORD dwCookie);</pre> <p>功能描述：用于断开由参数dwCookie所标识的出接口连接。</p> <p>参数描述：pRoot是指向组件的根接口指针；iid是该种类型出接口的接口IID值；dwCookie就是connect的返回值，是disconnect函数所要断开的出接口连接的唯一性标识。</p>

(4) COMX 组件文件的访问入口。

前面的章节中我们曾经介绍过，COMX 的组件文件（.kpi）实质上就是动态链接库（WINDOWS 平台上）或共享库（LINUX/UNIX 平台上）。这就不可避免地要提到一个问题：动态库的访问入口函数是什么？在微软的 COM 开发包中定义一个进程内的组件需要支持 4 个入口函数用于组件的注册、访问，以实现 COM 组件的位置无关性。

在 COMX 组件入口机制的设计过程中，考虑到跨平台特性和尽可能降低 COMX 组件开发门槛，采用了如下的设计策略：采用全局性的类工厂机制，即所有的组件都从一个统一的类工厂组件中创建出来，而不像微软的设计那样，每个 COM 组件都包含一个类工厂对象；组件的位置信息存储在 xml 配置文件里，这就避免了注册表的使用（在作者看来，有时候 WINDOWS 中的注册表简直就是软件开发者和使用者的一场噩梦）。关于 COMX 的类工厂机制和 xml 配置文件在后面章节中会深入阐述，此处不做详细介绍。基于上面的设计策略，COMX 组件的入口函数机制非常简洁，如表 5.5 所示。

表 5.5 COMX 组件所需支持的入口函数列表

入口函数名称	描述
CreateInstance	<p>函数原型：</p> <pre>extern "C" {     DLL_API TStatus CreateInstance(MREFTYPEID rclsid, MREFIID         riid, void **ppv); }</pre> <p>函数描述：</p> <p>COMX 组件的入口函数，用于创建指定类型的 COMX 组件对象。</p> <p>参数描述： rclsid 表示目标组件的 class id； riid 表示目标接口的 interface id； ppv 是该函数的返回值，用于返回所创建的 COMX 组件对象的接口指针。</p>
ServiceEntry	<p>函数原型：</p> <pre>extern "C" {     DLL_API const MUID &amp;ServiceEntry(); }</pre> <p>函数描述：</p> <p>该函数是一个可选函数，主要用于对服务型 COMX 组件的支持。</p> <p>参数描述： 参数为空</p>

在 COMX 组件的编码实现过程中，入口函数的支持也是通过一组宏来完成的，代码举例如下（gl\_content 组件中的入口函数宏编码）：

```
BEGIN_IMPLEMENT_CREATE_INSTANCE()
IMPLEMENT_CREATE_INSTANCE(TGLContent, CLSID_IGlContent)
```

```
END_IMPLEMENT_CREATE_INSTANCE()
```

在表 2-6 中提到了入口函数 ServiceEntry，该函数主要用于对服务型 COMX 组件的支持，本文在此就对 COMX 中的服务型组件进行一些介绍。

COMX 中服务型组件的概念有些类似于操作系统中的后台服务。这种类型的组件在 COMX 平台主程序启动时被自动加载，并且可以在任意一个组件内部查询使用。在 CAE 软件系统的开发中有很多组件模块都有这方面特性的需求，比如，用于对 OpenGL 图形上下文进行包装的组件 gl\_content，用于加载所有其它 GUI 界面插件的服务型组件 plugin\_server 等。服务型组件的支持也是通过宏来实现的，若我们想把上述的 gl\_content 组件变成服务型组件，只需在上面的宏代码之后添加一行新代码，举例如下：

```
BEGIN_IMPLEMENT_CREATE_INSTANCE()
    IMPLEMENT_CREATE_INSTANCE(TGLContent, CLSID_IGlContent)
END_IMPLEMENT_CREATE_INSTANCE()

SUPPORT_SERVICE_COMPONENT(CLSID_IGlContent)
```

### (5) COMX 中的组件包容和聚合机制。

COMX 中的包容(Containment)，也被称为委派(Delegation)，与 C++技术中类组合(class composition) 的概念是等同的。在 C++技术环境中，你可能从未听说过这个概念（也被称为嵌入：embedding），但是在创建 C++类时你可能经常使用它。

关于 COMX 聚合 (Aggregation)，除了内部组件接口直接向外公开以外，它与组件包容非常相似。聚合对象在它的内部不需要或根本不使用被包含对象的功能，而是直接向外公开内部对象的接口，就像这些接口属于它自己一样。

默认情况下，COMX 对它们的包容对象提供支持。在对它们包容对象进行处理时不需要做任何附加的事情。然而，当一个组件在聚合中充当内部组件时，它必须提供确切的支持。

由于聚合在编码实现上的复杂性，COMX 内核中也定义了一组宏来简化组件聚合机制的实现，下面将给出相关宏代码的使用步骤。

首先在聚合容器组件中添加下述的宏代码（以 fem\_core 组件为例）：

```
BEGIN_IMPLEMENT_AGGRAGATION_CONTAINER()
    CREATE_AGGRAGATION_OBJ(CLSID_IFemFrame)
END_IMPLEMENT_AGGRAGATION_CONTAINER()
```

```
IMPLEMENT_AGGREGATION_CONTAINER(TFemCore)
```

然后在被聚合组件中添加下述宏代码（以 fem\_frame 组件为例）：

```
BEGIN_IMPLEMENT_AGGREGATION_ROOT()
    IMPLEMENT_INTERFACE(IFemFrame)
    IMPLEMENT_INTERFACE_AGGREGATION_ROOT(IFemFrame)
    IMPLEMENT_EVENT_SOURCE_INTERFACE_EX()
END_IMPLEMENT_AGGREGATION_ROOT()
```

```
IMPLEMENT_AGGREGATION_FEATURE(TFemFrameComponent)
```

并将被聚合组件的入口函数支持宏代码修改成下面的样子：

```
BEGIN_IMPLEMENT_CREATE_AGGREGATION_INSTANCE()
    IMPLEMENT_CREATE_AGGREGATION_INSTANCE(TFemFrameComponent, CLSID_IFemFrame)
END_IMPLEMENT_CREATE_AGGREGATION_INSTANCE()
```

这样 fem\_frame 组件就被顺利的聚合在 fem\_core 组件中了。从接口级别上来看，fem\_frame 组件中的所有接口看起来都像是在 fem\_core 组件内部实现的一样，相关的封装和访问机制从接口的角度上看对组件使用者是透明的。

## (6) 组件

组件是 COMX 中的最基本概念，也是最基本的开发单位，其对外表现形式为：一组接口和出接口的集合，每个组件都有一个特有的 CLSID。

组件按其生存周期主要可分为：一般性组件、服务型组件和单实例组件。

下面是关于组件基本结构的简单例子。

```
// glBackground.cpp : Defines the entry point for the DLL application.
//

#include "stdafx.h"

#include <gl/gl.hxx>
#include <base/base.hxx>

using namespace KMAS::Die_maker::comx;
```

```

#include "glBackgroundcontext.h"

class TGIBackground : public IGIContentRender ,
    public IGIBackground,
    public IService,
    public TGIBackgroundContext,
    public TFactory
{
public:
    TGIBackground() :  INITIALIZE_REF_COUNT()
    {
        //nothing;
    }
    ~TGIBackground(){}
public:
    BEGIN_IMPLEMENT_ROOT()
        IMPLEMENT_INTERFACE(IGIContentRender)
        IMPLEMENT_INTERFACE_ROOT(IGIContentRender)
        IMPLEMENT_INTERFACE(IService)
        IMPLEMENT_INTERFACE(IGIBackground)
    END_IMPLEMENT_ROOT()
public:
    virtual TStatus STDCALL EnableBackground(bool flag = true);
    virtual TStatus STDCALL GetBackgroundState(bool &flag);

    virtual TStatus STDCALL PaintGL();
    virtual TStatus STDCALL MouseGL(TMouseEvent event){return M_OK;}
    virtual TStatus STDCALL InitializeGL(){return M_OK;}
public:
    virtual TStatus STDCALL InitializeService();
    virtual TStatus STDCALL TerminalService();
private:
private:
    TConnectManager tm;
};

BEGIN_IMPLEMENT_CREATE_INSTANCE()
IMPLEMENT_CREATE_INSTANCE(TGIBackground,CLSID_IGIBackground)
END_IMPLEMENT_CREATE_INSTANCE()

SUPPORT_SERVICE_COMPONENT(CLSID_IGIBackground)

```

```

TStatus STDCALL TGIBackground::InitializeService()
{
    TStatus status = M_OK;
    IGICContentConnector *p_content_connector = NULL;
    TFactory::QueryService(CLSID_IGICContent,IID_IGICContentConnector,(void**)&p_content_connector);
    tm.AddConnect(p_content_connector, IID_IGICContentRender, (IGICContentRender*)this);
    p_content_connector->Release();

    return status;
}

TStatus STDCALL TGIBackground::TerminalService()
{
    tm.ReleaseConnects();

    return M_OK;
}

TStatus STDCALL TGIBackground::PaintGL()
{
    TGIBackgroundContext::SetBackground();

    return M_OK;
}

TStatus STDCALL TGIBackground::EnableBackground(bool flag)
{
    TGIBackgroundContext::enableBackground(flag);

    return M_OK;
}

TStatus STDCALL TGIBackground::GetBackgroundState(bool &flag)
{
    TGIBackgroundContext::getBackgroundState(flag);

    return M_OK;
}

BOOL APIENTRY DllMain( HANDLE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved
                      )

```

```
{
    return TRUE;
}
```

**几个有用的宏:** CREATE COMX OBJ 创建组件对象  
QUERY SERVICE 查询服务型组件  
QUERY INTERFACE. 接口查询

## 5-5. COMX 中的软件工程原则

(1) **自顶向下的组件设计原则。**自顶向下法要求开发者首先制定系统的总体规划，然后逐步分离出高度结构化的子系统，从上至下实现整个系统。运用这类方法可以为企业或机构系统的中期或长期发展规划奠定基础，同时支持系统的整体性，为系统的总体规划、子系统的协调和通信提供保证。实践表明软件中稳定度从高到低依次为数据、功能、对象、接口，由此可以知道接口与实现分离，实现软件的高内聚、低耦合是适应软件演化特性，提高软件复用度的指导原则。我们知道，接口和实现分离是 COMX 组件技术基本特点之一。一般而言，接口设计先于组件功能模块的实现，这和自顶向下的开发方法是一致的。因此，作者推荐采用自顶向下的开发方法作为在 COMX 平台下开发软件组件的基本设计原则。

(2) **组件的正交性和领域分解原则。**正交(orthogonal)本来是一个几何概念，表示的是垂直相交，后来又被扩展到线性代数中，表示内积为零的两个向量之间的关系。在计算机科学领域中，正交这个词通常表示基本元素之间的互相独立与良好隔离，并且它们具备组合出一个完整“空间”的能力，而缺少其中任何一个都会丧失这种能力。比如 UNIX/Linux 系统提供了许多基本工具，其中每个工具都只专注于完成一种基本任务，并且任务间不互相重叠，所以这些工具的接口之间是正交关系，用户能够通过管道等机制组合使用这些工具以完成各种复杂的任务。软件的复杂性是影响系统开发和维护的主要因素，从上面的论述和例子中可以看到，正交分解是应付复杂性的好办法。然而，并不是所有情况下都能把软件系统设计成彼此正交的组件集合，这一点在目前的软件开发理论中也并未得到证明，没有一种规范化的方法保证所设计的组件之间的正交性。因此，在实际的软件组件开发实践中，我们可以把“正交性”原则作为终极设计目标，并尽可能保证它，而不必绝对地去强求。另外，在软件开发实践中我们发现，不同领域中的概念往往是正交的（或近似正交的）。比如，文件读写子模块和三维模型显示子模块分别属于文件 I/O 和图形显示等不同领域范畴，显见它们是正交的。因此，在绝大多数情况下我们都可以以不同领域范畴内的概念作为组件“正交

性”分解的基本依据。

(3) “窄接口”原则。在这里我们首先解释一下接口的概念，什么是接口呢？在 COMX 软件开发平台中，从广义上讲，接口就是一个组件对外所提供的功能相似或相关的一组方法的集合；从狭义上讲，接口就是关键字 `interface` 所代表的 COMX 平台下一种特有的概念。如果您有一些软件工程知识，那么下面的这些你一定不会陌生：一个软件实体应尽可能做到“高内聚，低耦合”！而做到低耦合的一个方法就是向外界提供一个“窄接口”。因此在设计基于 COMX 软件开发平台的组件接口时，我们也应尽可能遵循“窄接口”的接口分解原则。

(4) “用况驱动”即需求贯穿始终的原则。软件系统是为了服务于它的用户而出现的。因此，为了构造一个成功的软件系统，必须了解其预期用户所希望和需要的是什么。此处用户（user）这个术语所指的不仅仅是人，也可以是其他系统。从这个意义上讲，术语用户代表了与所开发的系统进行交互的某个人或某件事（例如，被提及的系统之外的另一个系统）。这种交互就是一个“用况”。用况是能够向用户提供有价值结果的系统中的一种功能。用况所获取的是功能需求。所有的用况合在一起就构成“用况模型”，它描述了系统的全部需求。该模型代替了传统的系统功能说明。现代软件工程理论证明，用况不只是一种确定系统需求的工具，它们还能驱动系统的设计、实现和测试的执行，也就是说，用况可以驱动开发过程，可以使开发者的注意力始终集中于被开发系统的需求上面，从而更为准确地实现系统最初的设计目标。

(5) “微内核+插件”架构下的增量式、迭代式的软件开发流程。开发一个具有一定规模的软件产品是一项非常艰巨的任务，可能会持续几个月甚至一年以上。将这项工作划分为较小的部分或“袖珍项目”（mini-project）是切实可行的。每个“袖珍项目”都是一次能够产生一个增量的迭代过程。迭代是指工作流中的步骤，而增量是指产品中增加的部分。为了获得最佳的效果，迭代过程必须是受控的（controlled）；也就是说，它们必须按照计划好的步骤有选择地执行。在每次迭代过程中，开发人员标识并详细描述有关的用况，以选定的架构为向导来创建设计，用组件来实现设计，并验证这些组件是否满足用况。如果一次迭代达到了目标（通常如此），开发工作便可进入下一次迭代。如果一次迭代未能达到预期目标，开发人员必须重新审查前面的方案，并使用一种新的方法。用况驱动、以架构为中心以及迭代和增量开发的概念是同等重要的。架构提供了一种结构来指导迭代过程中的工作，而用况则确定了目标并驱动每次迭代的工作。三者缺一不可，就像一张三条腿的凳子，缺少任何一条腿凳子都会翻倒。

## 5-6. 一组推荐的命名规则

表 5.6 命名规则

项目	举例
类名称	TShape
结构名	shape_t
类私有成员变量	_data
类私有函数	_set_data
类保护函数	setData
类公有函数	SetData
全局变量	G_data
静态全局变量	S_data
函数指针	pfnSetData
全局函数	G_fnSetData
静态全局函数	S_fnSetData

注：关于编码规范的细节参见李晓东老师的《C++编码规约》（见附录二、附录三）

# 第6章. 内核和插件

## 6-1. “微内核+插件” 机制概述

插件机制是软件的可扩展行得到支持的一种有效手段，基于插件的系统一般首先编写系统框架，并预先定义好系统的扩展接口。插件是根据系统预定的接口编写的扩展功能，实际上就是系统的扩展功能模块。一般而言，插件都是以独立文件的形式出现的。对于系统来说并不知道插件的具体功能，仅仅是为插件留下预定的接口，系统启动的时候根据插件的配置寻找插件，根据预定的接口把插件挂接到系统中。基于插件机制的软件系统一般分为“微内核+插件”和“据内核+插件”两种，微内核与巨内核之争已经有很长历史了，这在操作系统的设计中尤为突出，网上对于微内核与巨内核的讨论也同样适用于插件系统。微内核的好处在于系统的可扩展性强，如果你愿意，甚至可以将整个开发环境都替换掉。巨内核的好处在于插件非常简单，只需要将业务部分用统一的接口公布出来就可以，在开发具体模块的时候可以不用考虑开发的是不是插件。结合 CAE 软件的自身特点及其在可扩展性方面的需求，我们选择“微内核+插件”作为 CAE 基础软件开发平台的基本架构形式。

那么我们应该怎样给插件下一个定义呢？插件的定义可谓是指件架构体系在使用时会碰到的最大问题，何谓插件，不同人的看法都会有所不同，由于对于何谓插件的看法不同，必然就导致了在整个系统的设计中采用不同的设计方式，主要是在插件的粒度控制上会有不同的设计方式。在基于插件架构体系进行系统构建的时候插件的粒度定义是最难把握的，甚至难度上会超过以前的组件设计，需要的是更为规范的模块化设计方式，这在中小型软件企业中或 CAE 仿真算法研究机构中（这个领域中的研究机构一般都把注意力集中于关键的仿真算法上，比如如何建立新的本构、如何找到更高效的方程组解法等，在软件设计和软件工程方面往往很少关注，有时候甚至还赶不上一些小型的软件企业）往往是很难做到的，主要仍然是架构设计时的控制，在设计时重要的因素在于保持系统的松散结构，至于概要设计则完全可按照架构设计的约束去进行，详细设计则完全和平常的做法一样。为了尽可能降低插件机制的使用门槛，提升 COMX 平台下的软件开发效率，在本文中对 COMX 插件采取了一种非常简单的定义方式：

COMX 中的插件是一个或若干个功能相关的组件的集合，它在 COMX 内核程序上 (KDESKTOP.EXE + FACTORY.KPI + PLUGIN\_SERVER.KPI) 完成相对独立的软件功能。每个插件都包括相对自成体系的输入输出界面逻辑、底层算法及数据结构组件，甚至可以脱离其它插件独立形成一个 COMX 软件包而存在。这里面需特别注意的一点是：不同的插件之间可以共享相同的底层算法及数据结构组件。

在第一章的叙述中我们注意到组件机制也是保障软件模块拥有“可扩展性”的有效手段，那为什么还要在 COMX 平台中引入插件的设计机制呢，插件和组件又有什么样的区别与联系呢？在 COMX 中，插件是基于组件的，而又不能等同于组件。

首先，相较于组件而言插件拥有更大的粒度，更侧重于在功能上的自成体系（“高内聚、低耦合”），从而将软件系统有效的分解成相对独立的子系统，降低目标系统的规模和复杂性，极大提高软件开发效率。而组件一般比插件有更细的粒度，更侧重于组件自身的可重用性、

可扩展性等性能指标，一般并不考虑如何在功能上自成体系，往往作为插件或其它软件子系统的底层而存在。

其次，组件机制和插件机制的区别在于它们是否拥有一个功能性的、可扩展的可执行主框架平台（主程序）。组件设计级别上一般不存在这样的平台，组件设计机制的使用者必须独立设计并实施其目标系统主模块（主程序）的架构，而这种设计往往带有更多的随意性，并且对开发者的软件设计水平有较高的要求。而插件机制由于规范化地定义了软件系统的主模块（主程序），可以使开发者的精力集中于功能性的插件开发上，从而极大降低 COMX 开发平台的使用门槛，提升其开发效率。这也是 COMX 开发平台能否迅速得到普及取得成功的关键设计元素。

## 6-2. 主程序和内核组件

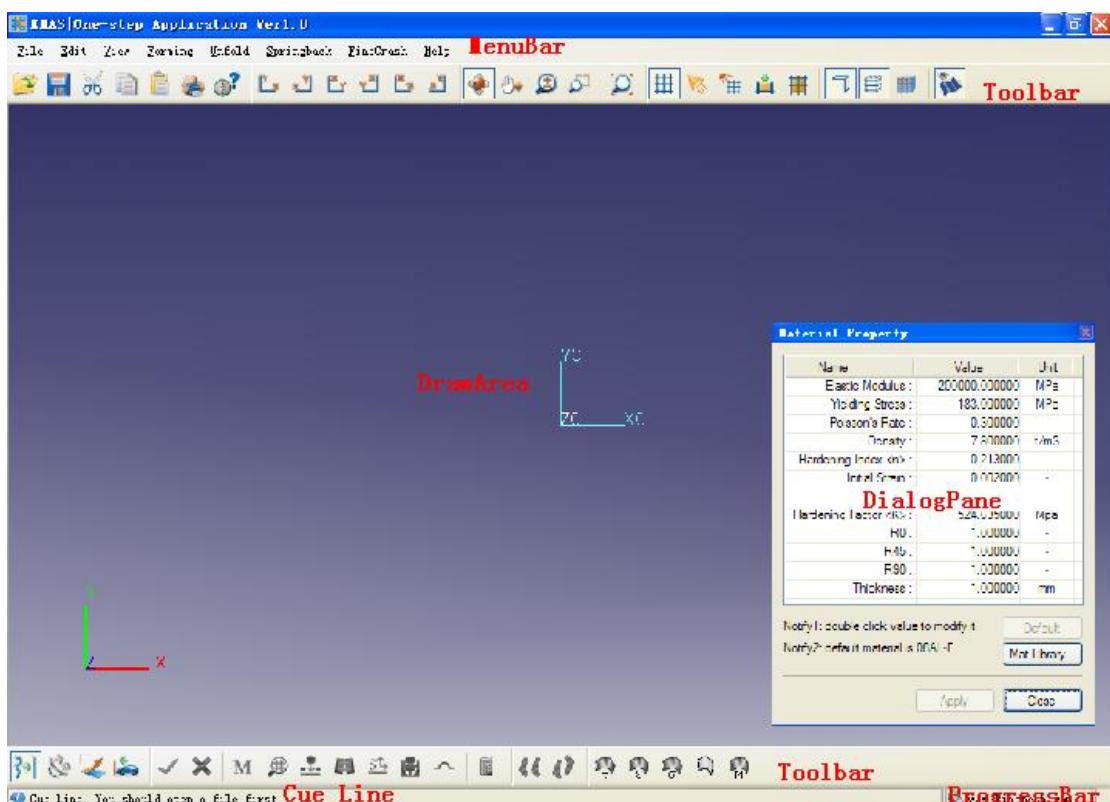


图 6.1 COMX 的主程序界面

从图 6.1 中可以看到在 COMX 主程序的界面上有如下的可扩展点：MenuBar、Toolbars、DrawArea、DialogPane、CueLine、ProgressBar。在 COMX 中可实现上述可扩展功能的插件支撑环境（即“内核”）主要是由如下几个组件共同完成的，即：KDESKTOP.EXE（这是 GUI 插件的主程序，在其中也实现了一些界面相关的接口，因此也可以将其视为一个组件）、factory.kpi 组件和 plugin\_server.kpi 组件。下面分别对这些组件在 GUI 插件支撑环境中所起的作用分述如下：

KDESKTOP.EXE 是 COMX 的 GUI 插件机制的主程序，它的主要功能有：读取 xml 配置文件

中的信息，设定主界面的标题、是否最大化和是否锁定工具条等属性；加载 factory.kpi 组件；通过 plugin\_server.kpi 组件获取各种界面插件（组件）中的界面元素描述信息，然后创建这些界面元素（比如菜单、工具条和对话框面板等）；处理鼠标、键盘等各种用户输入，并将其转发到相应的界面插件中；系统进度栏、cue line 提示栏的驱动；实现主窗口和 gl\_content.kpi 组件的连接等。在 KDESKTOP.EXE 中实现了下列一些接口：IComxUiPluginServerFrm、IUiProgressBarDriver 和 IUiToolbarStatusDriver。

表 6.1 IComxUiPluginServerFrm 的接口函数列表

接口函数名称	描    述
Relayout	刷新 KDESKTOP.EXE 的主窗口的界面元素布局
GetMainWnd	获取 KDESKTOP.EXE 的主窗口句柄
GetPackageCLSID	获取当前 COMX 软件包的 Class ID
GetClientWnd	获取图形显示区窗口的句柄

表 6.2 IUiProgressBarDriver 的接口函数列表

接口函数名称	描    述
UpdateUI	刷新系统公共进度条提示栏
Startup	初始化系统公共进度条提示栏
Closure	销毁系统公共进度条提示栏

表 6.3 IUiToolbarStatusDriver 的接口函数列表

接口函数名称	描    述
UpdateUIDriver	刷新系统 Cue Line 提示栏

FACTORY.KPI 组件的主要作用有：用于加载和解析当前 COMX 软件包的总体配置文件和打包文件（二者均为 xml 文件格式）；用于自动加载包括 GUI 插件组件在内的组件动态链接库文件（.kpi）；用于创建和管理包括 GUI 插件组件对象在内的各种组件对象；用于维护服务型组件（当然也包括服务型的 GUI 插件组件）对象的生存周期等。

表 6.4 IFactory2 的接口函数列表

接口函数名称	描    述
CreateInstanceEx	创建一个 COMX 组件对象，该函数和 CreateInstance 函数的区别在于：它支持被聚合组件对象的创建，而后者不支持被聚合组件对象的创建

表 6.5 IFactory 的接口函数列表

接口函数名称	描    述
CreateInstance	创建一个 COMX 组件对象
Register	注册一个 COMX 组件文件（.kpi）到当前 COMX 软件包中
Initialize	初始化当前 COMX 软件包中的所有服务型组件
Dispose	销毁当前 COMX 软件包中的所有服务型组件
QueryService	查询一个服务型组件的接口
LoadScript	加载并解析 COMX 软件包的打包文件（xml 格式）

表 6. 6 IFactoryGlobalSetting 的接口函数列表

接口函数名称	描    述
GetHomePath	获取 home 路径, 一般指向 COMX 软件包所在文件夹
SetHomePath	设置 home 路径, 一般指向 COMX 软件包所在文件夹
GetRootPath	获取 root 路径, 一般指向 COMX 内核所在文件夹
SetRootPath	设置 root 路径, 一般指向 COMX 内核所在文件夹
AppendScriptPath	添加一个脚本路径到当前脚本搜索路径列表中
GetScriptPathNum	获取当前脚本路径列表中的路径数目
GetScriptPath	在脚本路径列表中搜索指定脚本文件, 返回其绝对路径
AppendDataPath	添加一个数据文件路径到当前数据搜索路径列表中
AppendModulePath	添加一个组件文件路径到当前组件搜索路径列表中
SearchDataFilePath	在数据文件路径列表中搜索指定数据文件, 返回其绝对路径
SearchModuleFilePath	在组件文件路径列表中搜索指定组件文件, 返回其绝对路径
LoadProductInfo	加载并解析 COMX 软件包的总体配置文件 (xml 格式)
GetPackageFileName	获取 COMX 软件包的打包文件名 (xml 格式)。

PLUGIN\_SERVER.KPI 组件的作用主要有: 遍历维护所有 GUI 界面插件; 转发 KDESKTOP.EXE 的用户界面事件到相关 GUI 插件中; 包装和管理 KDESKTOP.EXE 中的设置 Cue Line 信息、界面布局刷新、获取相关窗口句柄等功能; 维护 GUI 插件的创建、关闭等全局性事件; 提供当前 COMX 软件包在打包文件 (xml 文件) 中动态创建的组件对象的遍历和访问功能。该组件实现了 IComxUiPluginServer、IComxUiPluginServer2、IComxUiPluginServerCueline、IComxUiPluginServerFrm、IProperty、IComxUiPluginServerEvent 和 IComxUiPluginServerEvent2 等接口。

表 6. 7 IComxUiPluginServer 的接口函数列表

接口函数名称	描    述
GenerateCmdID	为 GUI 插件中的界面元素自动生成 Command ID
GetPluginCount	获取系统中的 GUI 组件总数
GetPlugin	根据指定的索引值获取相关的 GUI 组件句柄

表 6. 8 IComxUiPluginServer2 的接口函数列表

接口函数名称	描    述
ResetToolbar	复位 KDESKTOP.EXE 主窗口中的工具条

表 6. 9 IComxUiPluginServerCueline 的接口函数列表

接口函数名称	描    述
GetCueline	设置主窗口中的 Cue Line 信息
SetCueline	获取主窗口中的当前 Cue Line 信息
ResetCueline	将主窗口中的 Cue Line 信息复位成默认值

表 6. 10 IComxUiPluginServerFrm 的接口函数列表

接口函数名称	描    述
Relayout	刷新 KDESKTOP.EXE 的主窗口的界面元素布局
GetMainWnd	获取 KDESKTOP.EXE 的主窗口句柄
GetPackageCLSID	获取当前 COMX 软件包的 Class ID

GetClientWnd	获取图形显示区窗口的句柄
--------------	--------------

表 6. 11 IProperty 的接口函数列表

接口函数名称	描    述
GetProperty	根据组件对象的脚本名字获取该组件对象的指针

表 6. 12 IComxUiPluginServerEvent 的接口函数列表

接口函数名称	描    述
CloseAllPane	激活插件关闭事件
Initialize	激活插件初始化事件

表 6. 13 IComxUiPluginServerEvent2 的接口函数列表

接口函数名称	描    述
GetContentMenu	获取系统上下文菜单 (Content Menu) 的描述信息

### 6-3. XML 配置脚本和软件包的组织形式

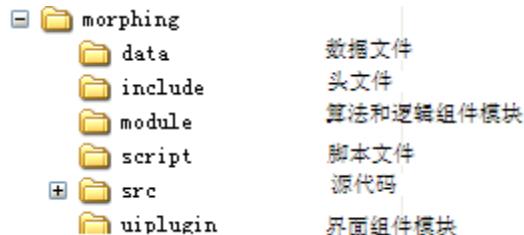


图 6. 2 软件包的组织形式

图 6. 2 给出了 COMX 中软件包的结构，这里设计到若干个组件的加载、连接和配置，如果这些操作都用 C++ 代码去实现，将会是一件非常繁琐、容易出错的工作，同时也失去了组件之间可动态连接的灵活性，在 COMX 中这些工作主要是由 xml 配置脚本来完成。

注意前述表格中的 LoadProductInfo 和 LoadScript 接口函数，它们是 COMX 软件包的 xml 配置和打包机制的基础。

接口函数 LoadProductInfo 用于加载 COMX 软件包的总体配置文件，该文件是一个 xml 格式的配置文件，其扩展名规定必须为 kproduct，比如 onestep\_ui.kproduct。该文件主要用于设置 COMX 软件包的脚本搜索路径、数据文件搜索路径和组件文件的搜索路径，同时该文件也负责 COMX 软件包的打包文件解析。该文件中的一些主要配置代码片段列举如下：

```

<config>
  <path>
    <home_path>y:/onestep_ui/</home_path>
    <root_path>y:/comx_sdk/</root_path>
    <script_paths>
      <item>~/script/</item>
    </script_paths>
    <data_paths>
      <item>~/data/</item>
    </data_paths>
    <module_path>
      <item>~/module/</item>
    
```

```

</module_path>
</path>
<package>"package.xml"</package>
</config>

```

其中 <home\_path> 和 <root\_path> 用于简化配置文件中的文件路径，比如：“y:/comx\_sdk/lib/base/”可简化为“/lib/base/”，“y:/onestep\_-ui/module/”可简化为“~/module/”；<script\_paths> 用于指定脚本文件的搜索路径；<data\_paths> 用于指定软件包中的数据文件的搜索路径；<module\_path> 用于指定 kpi 文件的搜索路径。<package> 用于指定软件包的主脚本文件，该文件从<script\_paths>条目中所指定的脚本文件路径列表中搜索得到。

接口函数 LoadScript 用于加载并解析 COMX 软件包的打包文件，该文件一般放在 COMX 软件的 script 子目录下面，文件名一般都指定成 package.xml。打包文件主要用于设置 COMX 的主框架程序 KDESKTOP.EXE 的一些基本属性（比如标题、是否最大化、是否锁定工具条等）、设置 COMX 界面插件中工具条的属性（停靠位置、是否隐藏以及是否新建一行等属性）、通过脚本创建 COMX 组件对象以及通过脚本创建组件对象出接口连接等功能。package.xml 文件中的一些主要配置代码片段列举如下：

```

<uiconfig title = "KMAS|One-step Application Ver1.0" max = "yes" lockbar = "yes">
    <toolbar name = "Common" dock = "top" hide = "no" newline = "yes"></toolbar>
</uiconfig>

```

这段代码中的<uiconfig>用于设置 COMX 的主框架程序 KDESKTOP.EXE 的基本属性，<toolbar>用于设置 COMX 界面插件中的工具条属性。

```

<addProperty>
    <addPropertyItem name = "Data" clsid = "CLSID_IFemCore" iid = "IID_IFemCore" hide = "no"/>
</addProperty>

```

该代码片段用于在脚本状态下动态创建 COMX 组件对象。

```

<addConnects>
    <addConnectItem src = "Origin:Render" target = "Origin:Data"/>
</addConnects>

```

该代码片段用于在脚本状态下动态建立 COMX 组件间的出接口连接。

## 6-4. 工具条和对话框界面插件的开发

总的来说界面插件和其它的组件大致上差不多，本质上都是动态链接库。所不同的是在 Windows 平台上一般组件是一个 Win32 DLL，而界面组件是一个 MFC DLL 或者 ATL DLL（界面元素分别有 MFC 和 wtl 支持），MFC DLL 一般使用得更多一些。图 6.3 和图 6.4 分别给出了 VC6 中建立相应的工程所使用的向导类型。

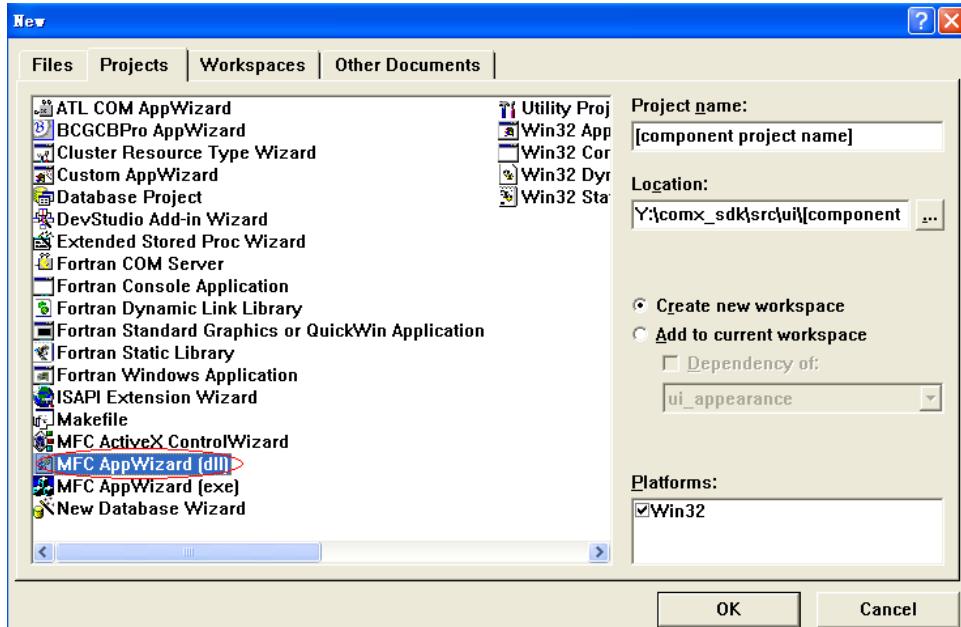


图 6.3 VC6 中使用的 MFC DLL Wizard

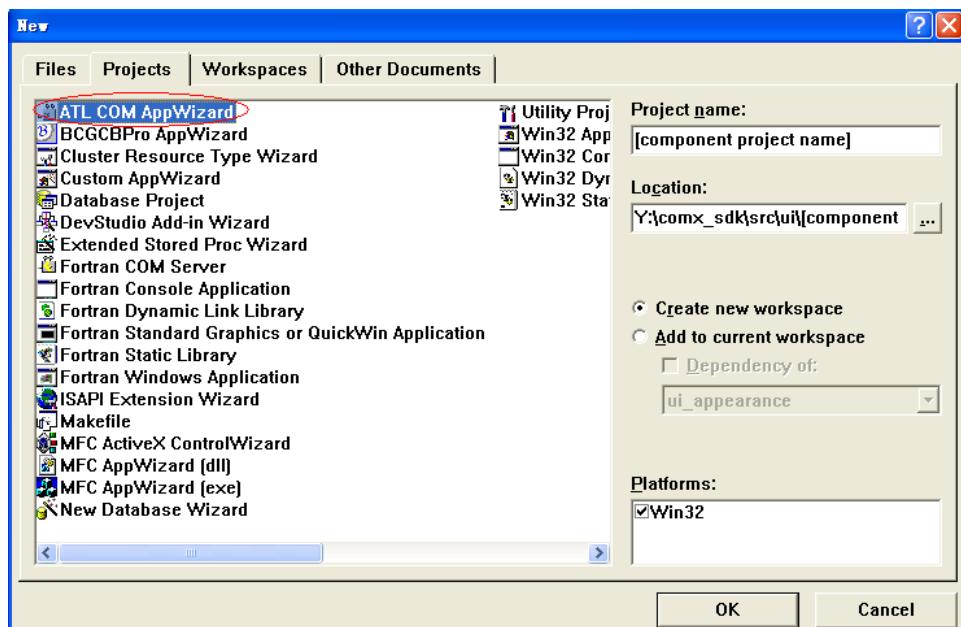


图 6.4 VC6 中使用的 ATL/WTL DLL Wizard

### (1) 工具条

我们以 training\_plugin 为例讲解 COMX 中的工具条组件。

首先使用图 6.3 中的向导建立一个 MFC DLL 工程, 选择“Regular DLL using shared MFC DLL”, 如图 6.5 所示。

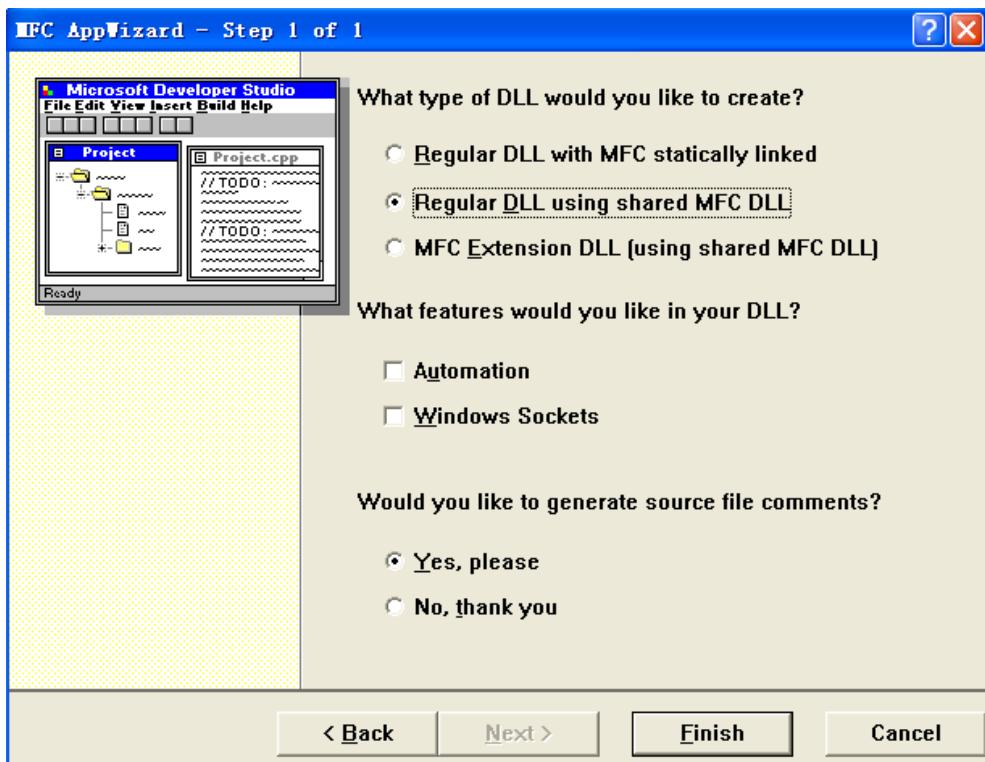


图 6.5 VC6 中使用的 MFC DLL Wizard (step-2)

点击 COMX Wizard 工具条上的  按钮激活组件代码框架生成向导，如图 6.6 所示。

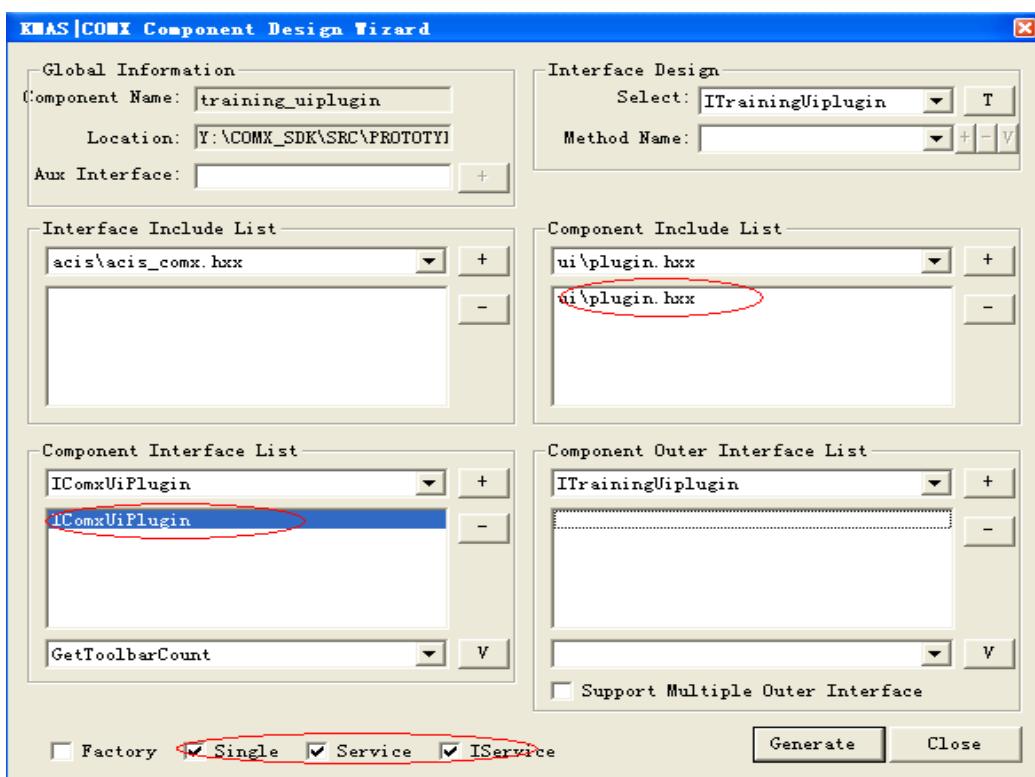


图 6.6 组件代码框架生成向导

在 Component include list 中加入 ui\plugin.hxx，然后 Components interface list 中加入 IComxUiPlugin 接口，该接口主要用于支持工具条，之后再选中 Single、Service 和 IService 三个 Toggle Button，最后点击 Generated 按钮就生成了组件代码框架。

代码框架自动生成了 training\_plugin.hxx、training\_plugin\_frame.cxx 和 training\_plugin\_impl.hxx 三个源代码文件。

training\_plugin.hxx 用于定义组件接口和 CLSID 等信息。

training\_plugin\_frame.cxx 中定义了组件代码框架。在该文件中除了// Append your codes here.注释所指定的位置外一般不允许添加代码，须添加的代码一般如下所示：

**代码片段 1:**

```
/*- Other Declarations Begin -*/
// Append your codes here.
private:
    DWORD dwCookie;

/*- Other Declaration End -*/
};
```

**代码片段 2:**

```
TStatus STDCALL TTrainingUipluginComponent::InitializeService()
{
    TStatus status = M_FAIL;

    /*- InitializeService Begin -*/

    // Append your codes here.
    IComxUiPluginServer *p_plugin_server = NULL;
    TFactory factory;
    factory.QueryService(
        CLSID_IPlugServer,
        IID_IComxUiPluginServer,
        (void**)&p_plugin_server);
    assert(p_plugin_server);

    dwCookie = 0;
    KMAS::Die_maker::comx::connect(
        p_plugin_server,
        (IComxUiPlugin*)this,
        IID_IComxUiPlugin,
        dwCookie);

    p_plugin_server->Release();

    /*- InitializeService End -*/

    return status;
}
```

## 代码片段 3:

```

TStatus STDCALL TTrainingUipluginComponent::TerminalService()
{
    TStatus status = M_FAIL;

    /*- TerminalService Begin -*/

    // Append your codes here.
    IComxUiPluginServer *p_plugin_server = NULL;
    TFactory factory;
    factory.QueryService(
        CLSID_IPlugServer,
        IID_IComxUiPluginServer,
        (void**)&p_plugin_server);
    assert(p_plugin_server);

    KMAS::Die_maker::comx::disconnect(
        p_plugin_server,
        IID_IComxUiPlugin,
        dwCookie);

    p_plugin_server->Release();

    /*- TerminalService End -*/

    return status;
}

```

## 代码片段 4:

```

#define UI_EXTEND

#include "stdafx.h"
#include "training_uiplugin.hxx"

#include <base\service.hxx>
#include <base\factory.hxx>

#include <ui\plugin.hxx>

using namespace KMAS::Die_maker::comx;

#include "resource.h"
#include "rex.hxx"
#include <ui\plugin_util.hxx>
#include "training_uiplugin_impl.hxx"

```

training\_plugin\_impl.hxx 中应加入组件功能实现的代码。在该文件中应手工加入一些代码，如下所示：

## 代码片段 1:

```

protected:
    TTrainingUipluginImpl()
    {
        init_toolbar();
    }

```

## 代码片段 2:

```

template<typename TOwner>
bool TTrainingUipluginImpl<TOwner>::implGetInstance(HINSTANCE& hInstance)
{
    TOwner *pOwner = static_cast<TOwner*>(this);
    bool ret = false;

    /*- implGetInstance Begin -*/

    // Append your codes here.
    IMPL_GET_INSTANCE(hInstance, ret);

    /*- implGetInstance End -*/

    return ret;
}

```

代码片段 3:

```

template<typename TOwner>
bool TTrainingUipluginImpl<TOwner>::implGetToolbarCount(int& cnt)
{
    TOwner *pOwner = static_cast<TOwner*>(this);
    bool ret = false;

    /*- implGetToolbarCount Begin -*/

    // Append your codes here.
    IMPL_GET_TOOLBAR_COUNT(cnt, ret);

    /*- implGetToolbarCount End -*/

    return ret;
}

```

代码片段 4:

```

template<typename TOwner>
bool TTrainingUipluginImpl<TOwner>::implGetToolbar(const int& i
{
    TOwner *pOwner = static_cast<TOwner*>(this);
    bool ret = false;

    /*- implGetToolbar Begin -*/

    // Append your codes here.
    IMPL_GET_TOOLBAR(index, tb, ret);

    /*- implGetToolbar End -*/

    return ret;
}

```

代码片段 5:

```

template<typename TOwner>
bool TTrainingUipluginImpl<TOwner>::impl_DispatchCmd(const unsigned int& nID)
{
    TOwner *pOwner = static_cast<TOwner*>(this);
    bool ret = false;

    /* impl_DispatchCmd Begin */
    // Append your codes here.
    IMPL_DISPATCH_CMD(nID, dispatch_cmd, ret);
    /* impl_DispatchCmd End */

    return ret;
}

```

代码片段 6:

```

template<typename TOwner>
bool TTrainingUipluginImpl<TOwner>::impl_GetMask(const unsigned int& nID,
{
    TOwner *pOwner = static_cast<TOwner*>(this);
    bool ret = false;

    /* impl_GetMask Begin */
    // Append your codes here.
    IMPL_GET_MASK(nID, mask, ret);
    /* impl_GetMask End */

    return ret;
}

```

代码片段 7:

```

template<typename TOwner>
void TTrainingUipluginImpl<TOwner>::dispatch_cmd(
    const int &tb_index,
    const int &item_index)
{
    flags_mqr.Activate(tb_index, item_index);
    if (tb_index == 0)
    {
        switch(item_index)
        {
            case 1:
                //dispatch msg.
                break;
            case 2:
                //dispatch msg.
                break;
            default:
                //dispatch msg.
        }
    }
}

```

代码片段 8:

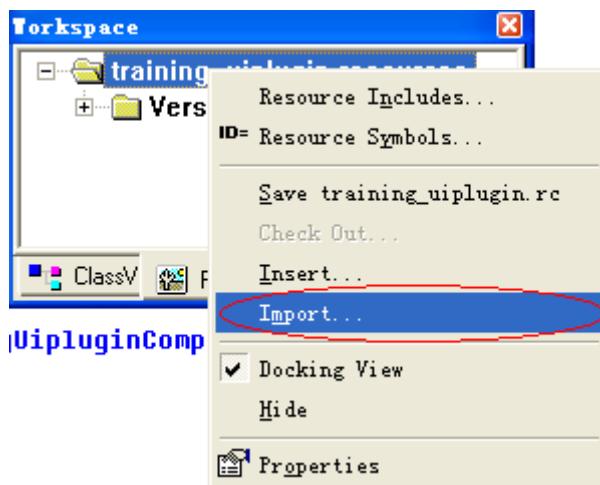
```

// Append your codes here.
private:
    TCOMXPluginFlagManager flags mgr;

/*- Other Declaration End -/
};

```

利用图像处理软件为工具条制作两幅图片：pic.bmp 和 pic\_gray.bmp , 第二幅图片为第一幅图片的灰度格式。将上述图片文件加入 VC 工程中。首先将 Workspace 切换到 ResourceView，然后在 ResourceView 根上单击右键，如下图所示，



在 Import Resource 对话框中选择 pic.bmp 文件将其导入。同样地，在导入 pic\_gray.bmp 文件。两个 bitmap 分别命名为：IDB\_BITMAP\_FLOAT 和 IDB\_BITMAP\_GRAY，如下图所示，



在工程中手工加入一个 res.hxx 文件其中内容如下：

```

#ifndef _COMX_RES_H
#define _COMX_RES_H

BEGIN_TOOLBAR_GROUP()
BEGIN_TOOLBAR()
    TOOLBAR_NAME("Common")
    TOOLBAR_MENUPATH("&File\n0\n&Common\n0")
    TOOLBAR_BMP_ID(IDB_BITMAP_FLOAT)
    TOOLBAR_BMP_FLOAT_ID(IDB_BITMAP_FLOAT)
    TOOLBAR_BMP_GRAY_ID(IDB_BITMAP_GRAY)

```

```
TOOLBAR_DOCK(PLUGIN_TOP)
TOOLBAR_NEWLINE(true)
TOOLBAR_SHOW(true)
END_TOOLBAR()
END_TOOLBAR_GROUP()

BEGIN_TOOLBAR_IMPL(0)
    BEGIN_TOOLBAR_ITEM()
        TOOLBAR_ITEM_ID(0)
        TOOLBAR_ITEM_NAME("&Open...|tCtrl+O")
        TOOLBAR_ITEM_FLAG(PLUGIN_ENABLE)
        TOOLBAR_ITEM_MASK(PLUGIN_TOOLBAR | PLUGIN_MENU)
        TOOLBAR_ITEM_TIP("Open")
        TOOLBAR_ITEM_HELP("Open an existing document")
        TOOLBAR_ITEM_MENU_PATH("&File\n0")
        TOOLBAR_ITEM_MENU_INDEX(-1)
    END_TOOLBAR_ITEM()
    BEGIN_TOOLBAR_ITEM()
        TOOLBAR_ITEM_ID(0)
        TOOLBAR_ITEM_NAME("&Save...|tCtrl+S")
        TOOLBAR_ITEM_FLAG(PLUGIN_ENABLE)
        TOOLBAR_ITEM_MASK(PLUGIN_TOOLBAR | PLUGIN_MENU)
        TOOLBAR_ITEM_TIP("Save")
        TOOLBAR_ITEM_HELP("Save the active document")
        TOOLBAR_ITEM_MENU_PATH("&File\n0")
        TOOLBAR_ITEM_MENU_INDEX(-1)
    END_TOOLBAR_ITEM()
    BEGIN_TOOLBAR_ITEM()
        TOOLBAR_ITEM_ID(0)
        TOOLBAR_ITEM_NAME("Cu&t|tCtrl+X")
        TOOLBAR_ITEM_FLAG(PLUGIN_ENABLE)
        TOOLBAR_ITEM_MASK(PLUGIN_TOOLBAR | PLUGIN_MENU)
        TOOLBAR_ITEM_TIP("Cut")
        TOOLBAR_ITEM_HELP("Cut the selection and put it on the Clipboard")
        TOOLBAR_ITEM_MENU_PATH("&Edit\n1")
        TOOLBAR_ITEM_MENU_INDEX(-1)
    END_TOOLBAR_ITEM()
    BEGIN_TOOLBAR_ITEM()
        TOOLBAR_ITEM_ID(0)
        TOOLBAR_ITEM_NAME("&Copy|tCtrl+C")
        TOOLBAR_ITEM_FLAG(PLUGIN_ENABLE)
        TOOLBAR_ITEM_MASK(PLUGIN_TOOLBAR | PLUGIN_MENU)
        TOOLBAR_ITEM_TIP("Copy")
        TOOLBAR_ITEM_HELP("Copy the selection and put it on the Clipboard")
```

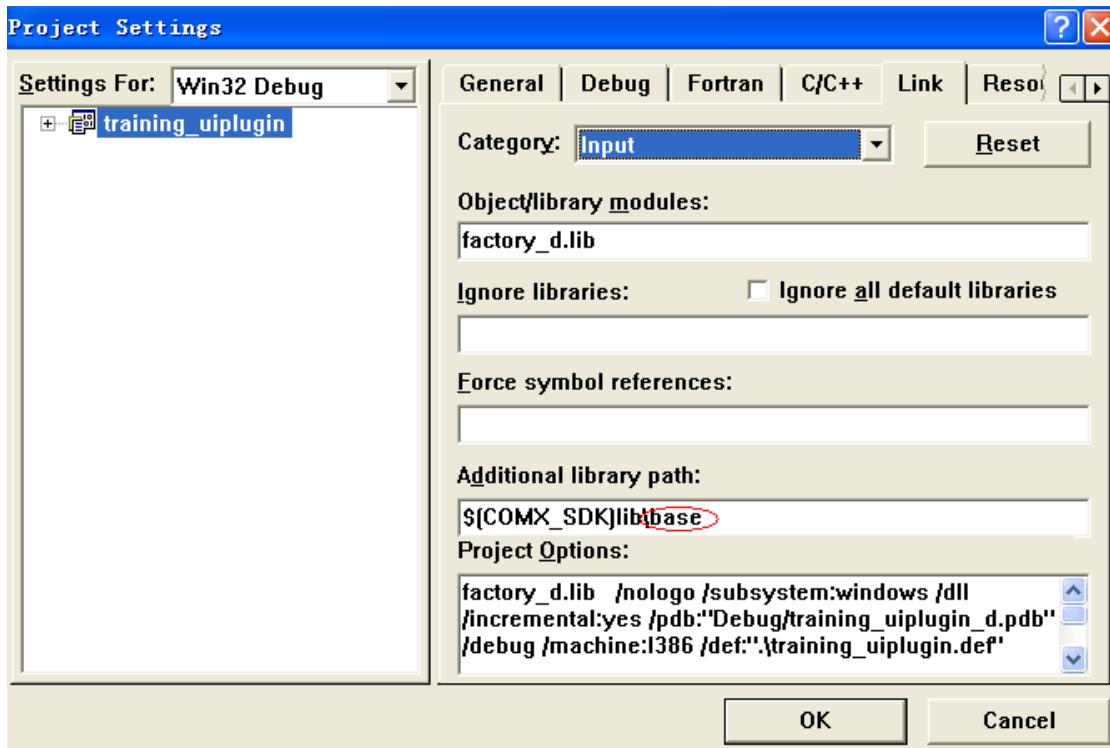
```

    TOOLBAR_ITEM_MENU_PATH("&Edit\n1")
    TOOLBAR_ITEM_MENU_INDEX(-1)
END_TOOLBAR_ITEM()
BEGIN_TOOLBAR_ITEM()
    TOOLBAR_ITEM_ID(0)
    TOOLBAR_ITEM_NAME("&Paste\tCtrl+V")
    TOOLBAR_ITEM_FLAG(PLUGIN_ENABLE)
    TOOLBAR_ITEM_MASK(PLUGIN_TOOLBAR | PLUGIN_MENU)
    TOOLBAR_ITEM_TIP("Paste")
    TOOLBAR_ITEM_HELP("Insert Clipboard contents")
    TOOLBAR_ITEM_MENU_PATH("&Edit\n1")
    TOOLBAR_ITEM_MENU_INDEX(-1)
END_TOOLBAR_ITEM()
BEGIN_TOOLBAR_ITEM()
    TOOLBAR_ITEM_ID(0)
    TOOLBAR_ITEM_NAME("&Print...\tCtrl+P")
    TOOLBAR_ITEM_FLAG(PLUGIN_ENABLE)
    TOOLBAR_ITEM_MASK(PLUGIN_TOOLBAR | PLUGIN_MENU)
    TOOLBAR_ITEM_TIP("Print")
    TOOLBAR_ITEM_HELP("Print the active document")
    TOOLBAR_ITEM_MENU_PATH("&File\n0")
    TOOLBAR_ITEM_MENU_INDEX(-1)
END_TOOLBAR_ITEM()
BEGIN_TOOLBAR_ITEM()
    TOOLBAR_ITEM_ID(0)
    TOOLBAR_ITEM_NAME("&About...")
    TOOLBAR_ITEM_FLAG(PLUGIN_ENABLE)
    TOOLBAR_ITEM_MASK(PLUGIN_TOOLBAR | PLUGIN_MENU)
    TOOLBAR_ITEM_TIP("About")
    TOOLBAR_ITEM_HELP("Display program information, version number and
copyright")
    TOOLBAR_ITEM_MENU_PATH("&Help\n6")
    TOOLBAR_ITEM_MENU_INDEX(-1)
END_TOOLBAR_ITEM()
END_TOOLBAR_IMPL(0)

BEGIN_GENERATE_INIT_TOOLBAR_FUN()
    INIT_TOOLBAR(0)
END_GENERATE_INIT_TOOLBAR_FUN()

#endif
注意：在向导版本较低生成的代码需要做下述调整：
1)修改链接选项

```

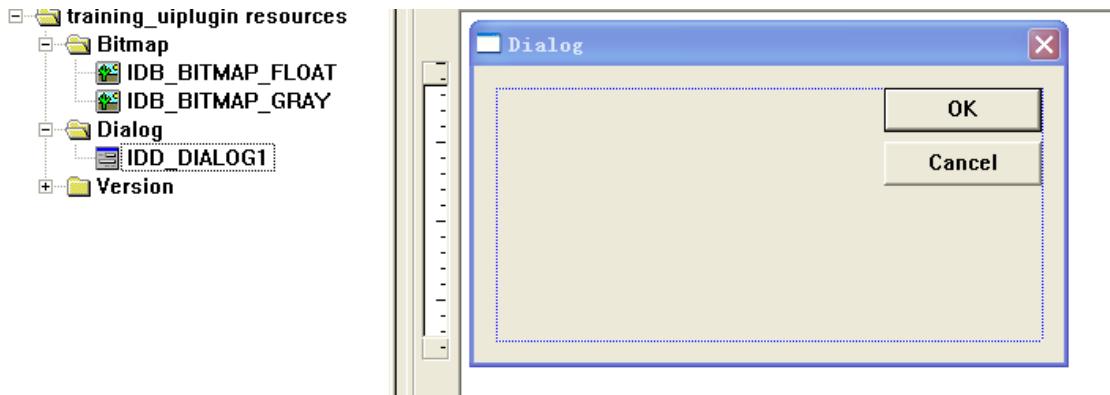


2) 编译时若出现.h 头文件找不到，在.h 后加 xx 即改为.hxx.

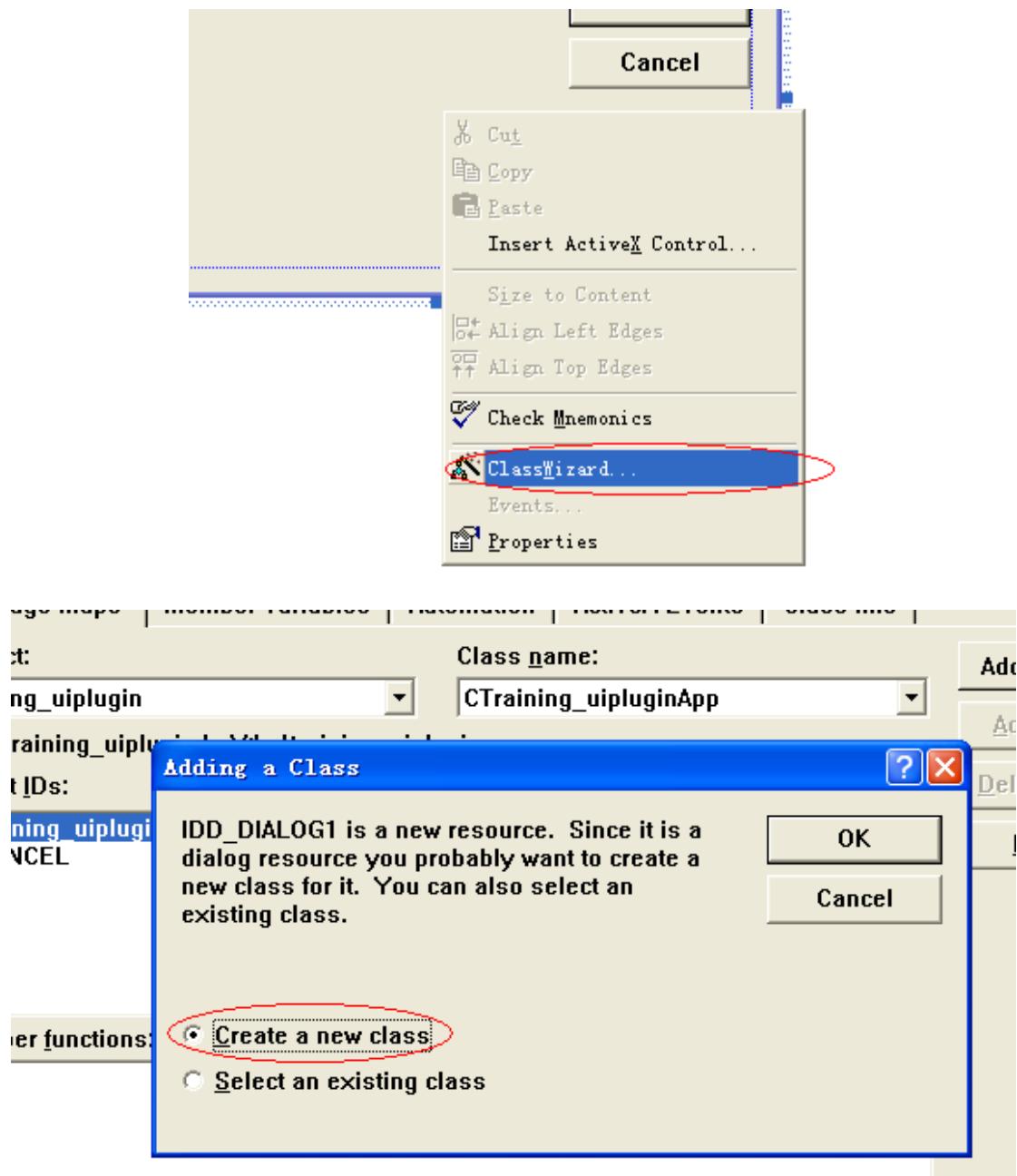
**编译成功之后，工具条组件就完成了。**

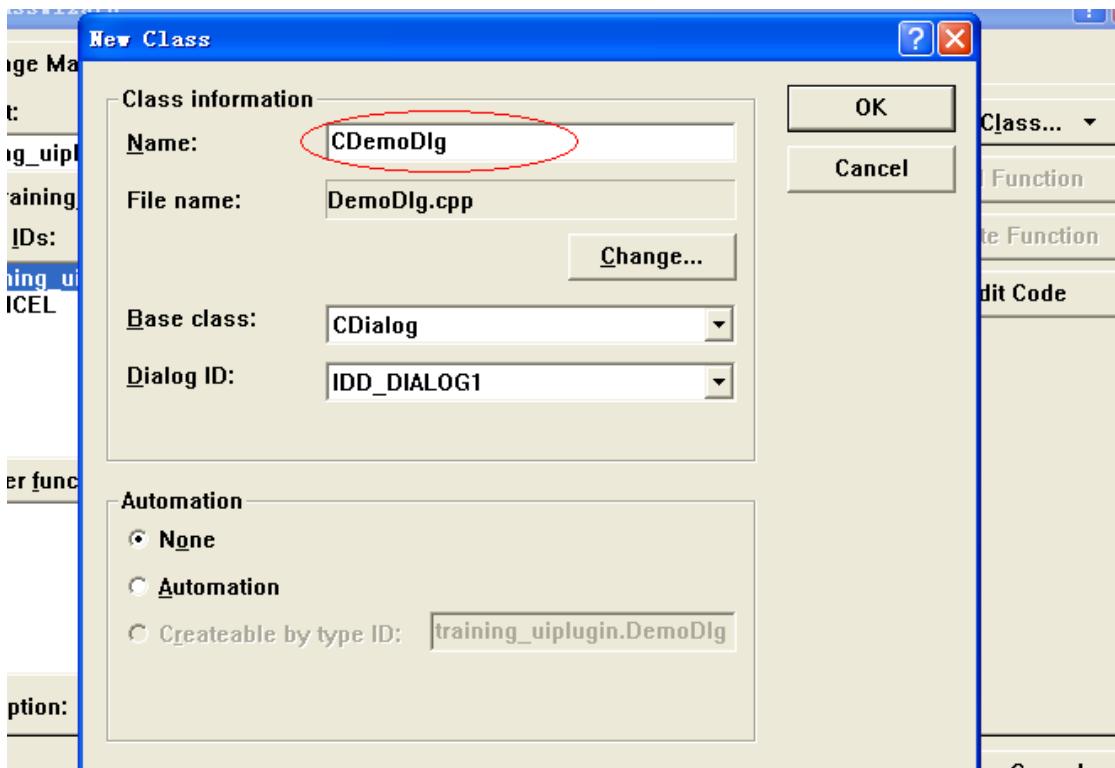
## (2) 对话框

首先在 ResourcesView 中插入一个对话框，如下图：



采用类向导生成对话框类，如下图：





在 training\_uiplugin\_frame.hxx 中加入下述代码，

```
#define UI_EXTEND

#include "stdafx.h"
#include "training_uiplugin.hxx"

#include <base\service.hxx>
#include <base\factory.hxx>

#include <ui\plugin.hxx>

using namespace KMAS::Die_maker::comx;

#include "resource.h"
#include "rex.hxx"
#include <ui\plugin_util.hxx>

#include "DemoDlg.h" // This line is circled in red

#include "training_uiplugin_impl.hxx"
```

在 training\_uiplugin\_impl.hxx 中加入下述代码，

```

template<typename TOwner>
void TTrainingUipluginImpl<TOwner>::dispatch_cmd(
    const int &tb_index,
    const int &item_index)
{
    flags_mgr.Activate(tb_index, item_index);
    if (tb_index == 0)
    {
        switch(item_index)
        {
            case 0:
            {
                AFX_MANAGE_STATE(AfxGetStaticModuleState());
                CDemoDlg dlg;
                dlg.DoModal();
            }
            //dispatch msg.
            break;
            case 1:
            //dispatch msg.
            break;
            default:
            //dispatch msg.
        }
    }
}

```

注意：AFX\_MANAGE\_STATE(AfxGetStaticModuleState());的使用

## 6-5. “事件”（“出接口”）机制

前面介绍过“出接口”的概念，“出接口”也可以被理解成消息响应的“事件”，即：引用“出接口”的组件可以看成是“事件的发送者”，实现“出接口”的组件可以看成是“事件的接受者”。在插件机制中，内核与插件、插件与插件之间的通讯和联系必须要使用到“事件”机制，所以“出接口”（“事件”）的机制非常重要，在此进行进一步的强调。

下面我们以对话框面板切换机制为例讲解“事件”机制的重要性。设想有一组对话框插件，它们完成一组顺序性的工作：即当某一个面板打开的时候，其它面板应当被关闭。为了完成上述功能，我们设想一个最简单的解决方案：在每个面板被打开前都调用其它面板的关闭方法。这个解决方案有如下问题：每增加一个新的面板插件，都要在所有其它相关面板插件中调用该面板的关闭方法，显然这导致了面板之间的互相依赖（接口依赖），增加新的面板也非常麻烦，对于插件的“可扩展性”、“模块化”都带来很大影响。运用事件机制可以有效解决上述问题：首先我们在 plugin\_server 组件中定义一个出接口 IComxUiPluginEvent (Multi)、定义一个接口 IComxUiPluginServerEvent，前者用于事件的定义，后者用于发送事件。这样只需要在每个面板插件中都实现 IComxUiPluginEvent，然后连接到 plugin\_server 组件，再在每个面板打开前调用 IComxUiPluginServerEvent 接口发送事件就可以了。

## 6-6. 一些值得特别关注的基础性组件

表 6.14 一些值得特别关注的组件

组件名称	描述
tag_pool	本身的类型为 unsigned long，其作用在于将任何类型的指针转化为一个 unsigned long 类型，比如 ACIS 中的 EDGE*, FACE*, BODY* 等，可避免接口定义对于特定复杂类型的依赖。
gl_content	建立 opengl 环境和 DrawArea 之间的连接、相应绘图和鼠标消息，通过出接口 IG1ContentRender 把绘图事件和鼠标事件转发到其它组件，是 opengl 最核心的组件，其它 opengl 相关组件都直接或间接依赖于该组件。
gl_wcs	用于保存当前模型空间的包围盒，将图形放缩到一个合适的范围，以便于观察。
gl_data_base	基于显示列表，管理所有 opengl 可绘制实体，完成 opengl 显示和选取的底层功能。
gl_drag	实现 opengl 点拖拽功能
gl_light	封装 opengl 光照模型。
gl_material	封装 opengl 材质模型
gl_pickup	封装 opengl 公用选取机制
gl_sensor	Opengl 传感器，实现旋转、平移、缩放等功能。
gl_toolbox	Opengl 工具箱
gl_window	实现一个 opengl 窗口，提供透明机制
gl_window_manager	Opengl 窗口管理器
gl_plot	基于 opengl 的 2D 绘图纸
ui_progress_bar	驱动主界面右下角的进度条
ui_toolbar_status	驱动主界面下方的 cueline 状态条
ui_acis_selection_toolkit	Acis 实体选择工具箱
ui_meshes_selection_toolkit	网格选择工具箱

comx_storage	跨平台的目录和文件夹机制
xml_stream	公共 xml 文件读写组件。

# 附录一 代码书写规范

## C++编码规约

第 0.0.1 版 2009 年 03 月 20 日



作者：李晓东

大连理工大学汽车工程学院  
School of AutomotiveEngineering,  
Dalian University of Technology

## 变更履历

区分：A=追加 / U=更新 / D=删除

## 前言

对于任何工程项目来说，统一的施工标准都是保证工程质量的重要因素。堪称当今人类最抽象、最复杂的工程——软件工程，自然更加不能例外。

高品质、易维护的软件开发离不开清晰严格的编码规范。本文档详细描述 C++ 软件开发过程中的编码规范。本规范也适用于所有在文档中出现的源码。

除了“语法高亮”部分，本文档中的编码规范都以：

规则（或建议）	解释
---------	----

的格式给出，其中强制性规则使用黑色，建议性规则使用灰色。

## 语法高亮与字体

### 字体

字体规范如下：

使用等宽字体	由于非等宽字体在对其等方面问题多多，任何情况下，源码都必须使用等宽字体编辑和显示。
每个制表符（TAB）的宽度为 4 个半角字符	不一致的缩进宽度会导致行与行之间的参差不齐，进而严重影响代码的可读性。使用 TAB 或者 4 个半角空格均可，只要保证一致就行。否则，在不同的编辑器里可能会出现格式错乱的情况。
优先使用 <b>Fixedsys</b>	<p>在 Windows 平台中，应该优先使用字体：<b>Fixedsys</b>，这也是操作系统 UI（所有的菜单、按钮、标题栏、对话框等等）默认使用的字体。该字体的好处很多：</p> <ul style="list-style-type: none"> <li>• 兼容性好：所有 Windows 平台都支持该字体</li> <li>• 显示清晰：该字体为点阵字体，相对于矢量字体来说在显示器中呈现的影像更为清晰。矢量字体虽然可以自由缩放，但这个功能对于纯文本格式的程序源码来说没有任何实际作用。</li> </ul> <p>而且当显示字号较小（12pt 以下）时，矢量字体还有一些明显的缺陷：</p> <ul style="list-style-type: none"> <li>◦ 文字的边缘会有严重的凹凸感。</li> <li>◦ 一些笔画的比例也会失调。</li> <li>◦ 开启了柔化字体边缘后，还会使文字显得模糊不清。</li> </ul>

	<p>说句题外话，这也是 Gnome 和 KDE 等其它 GUI 环境不如 Windows 的一个重要方面。</p> <ul style="list-style-type: none"> <li>支持多语言：Fixedsys 是 UNICODE 字体，支持世界上几乎所有的文字符号。这对编写中文注释是很方便的。</li> </ul>
--	--

## 语法高亮

所有在文档中出现的代码段均必须严格符合下表定义的语法高亮规范。在编辑源码时，应该根据编辑器支持的自定义选项最大限度地满足下表定义的高亮规范。

类型颜色举例：

类型	颜色	举例
注释	R0;G128;B0 (深绿)	// 注释例子
关键字	R0;G0;B255 (蓝)	typedef, int, dynamic_cast class ...
类、结构、联合、枚举等 其它自定义类型	R0;G0;B255 (蓝)	class CMyClass, enum ERRTYPE, typedef int CODE ...
名空间	R0;G0;B255 (蓝)	namespace BaiY
数字	R255;G0;B0 (红)	012 119u 0xff ...
字符、字符串	R0;G128;B128 (深蓝绿)	"string", 'c ...
宏定义、枚举值	R255;G128;B0 (橙黄)	#define UNICODE, enum { RED, GREEN, BLUE };
操作符	R136;G0;B0 (棕色)	<>, = + - * / ; { } () [ ] ...
方法/函数	R136;G0;B0 (棕色)	MyFunc()
变量	R128;G128;B128 (中灰色)	int nMyVar;
背景	R255;G255;B255 (白色)	

其它	<b>R0;G0;B0</b> (黑色)	other things (通常是一个错误)
----	----------------------	------------------------

## 文件结构

### 文件头注释

所有 C++ 的源文件均必须包含一个规范的文件头，文件头包含了该文件的名称、功能概述、作者、版权和版本历史信息等内容。标准文件头的格式为：

格式所限，参照“example.cpp”。

每行注释的长度都不应该超过 80 个半角字符。还要注意缩进和齐，以利阅读。

### 头文件

头文件通常由以下几部分组成：

文件头注释	每个头文件，无论是内部的还是外部的，都应该由一个规范的文件头注释作为开始。
预处理块	为了防止头文件被重复引用，应当用 #ifndef/#define/#endif 结构产生预处理块。 参照“example.cpp”。
函数和类/结构的声明等	声明模块的接口
需要包含的内联函数定义文件（如果有的话）	如果类中的内联函数较多，或者一个头文件中包含多个类的定义（不推荐），可以将所有内联函数定义放入一个单独的内联函数定义文件中，并在类声明之后用 “#include” 指令把它包含进来。

头文件的编码规则：

引用文件的格式	用 #include < <i>filename.h</i> > 格式来引用标准库和系统库的头文件（编译器将从标准库目录开始搜索）。  用 #include "filename.h" 格式来引用当前工程中的头文件（编译器将从该文件所在目录开始搜索）。
分割多组接口（如果有的话）	如果在一个头件中定义了多个类或者多组接口（不推荐），为了便于浏览，应该在每个类/每组接口间使用分割带把它们相互分开。  // ##### // #####CXXX 类开始#####  // ##### CXXX 类结束#####

	// #####
--	----------

## 内联函数定义文件

如上所述，在内联函数较多的情况下，为了避免头文件过长、版面混乱，可以将所有的内联函数定义移到一个单独的文件中去，然后再用#include 指令将它包含到类声明的后面。这样的文件称为一个内联函数定义文件。

按照惯例，应该将这个文件命名为“filename.inl”，其中“filename”与相应的头文件和实现文件相同。

内联函数定义文件由以下几部分组成：

文件头注释	每内联函数定义文件都应该由一个规范的文件头注释作为开始
内联函数定义	内联函数的实现体

内联函数定义文件的编码规则：

分割多组接口（如果有的话）	如果在一个内联函数定义文件中定义了多个类或者多组接口的内联函数（不推荐），必须在每个类/每组接口间使用分割带把它们相互分开。
文件组成中为什么没有预处理块？	与头文件不同，内联函数定义文件通常不需要定义预处理块，这是因为它通常被包含在与其相应的头文件预处理块内。

## 实现文件

实现文件包含所有数据和代码的实现体。实现文件的格式为：

文件头注释	每个实现文件都应该由一个规范的文件头注释作为开始
对配套头文件的引用	引用声明了此文件实现的类、函数及数据的头文件
对一些仅用于实现的头文件的引用（如果有的话）	将仅与实现相关的接口包含在实现文件里(而不是头文件中)是一个非常好的编程习惯。这样可以有效地屏蔽不应该暴露的实现细节，将实现改变对其他模块的影响降低到最少。
程序的实现体	数据和函数的定义

实现文件的编码规则：

分割每个部分	在本地（静态）定义和外部定义间，以及不同接口或不同类的实现之间，应使用分割带相互分开。
--------	---

## 命名规则

如果想要有效的管理一个稍微复杂一点的体系，针对其中事物的一套统一、带层次结构、清晰明了的命名准则就是必不可少而且非常好用的工具。

在软件开发这一高度抽象而且十分复杂的活动中，命名规则的重要性更显得尤为突出。一套定义良好并且完整的、在整个项目中统一使用的命名规范将大大提升源代码的可读性和软件的可维护性。

在引入细节之前，先说明一下命名规范的整体原则：

同一性	在编写一个子模块或派生类的时候，要遵循其基类或整体模块的命名风格，保持命名风格在整个模块中的同一性。
标识符组成	标识符采用英文单词或其组合，应当直观且可以拼读，可望文知意，用词应当准确。
最小化长度 && 最大化信息量原则	在保持一个标识符意思明确的同时，应当尽量缩短其长度。
避免过于相似	不要出现仅靠大小写区分的相似的标识符，例如“i”与“I”，“function”与“Function”等等。
避免在不同级别的作用域中重名	程序中不要出现名字完全相同的局部变量和全局变量，尽管两者的作用域不同而不会发生语法错误，但容易使人误解。
正确命名具有互斥意义的标识符	用正确的反义词组命名具有互斥意义的标识符，如：“nMinValue” 和 “nMaxValue”，“GetName()” 和 “SetName()” ....
避免名字中出现数字编号	尽量避免名字中出现数字编号，如 Value1, Value2 等，除非逻辑上的确需要编号。这是为了防止程序员偷懒，不肯为命名动脑筋而导致产生无意义的名字（因为用数字编号最省事）。

## 类/结构

除了异常类等个别情况（不希望用户把该类看作一个普通的、正常的类之情况）外，C++类/结构的命名应该遵循以下准则：

C++类/结构的命名	类的名称都要以大写字母“C”开头或者“T”，后跟一个或多个单词。为便于界定，每个单词的首字母要大写。
推荐的组成形式	类的命名推荐用“名词”或“形容词+名词”的形式，例如：“CAnalyzer”，“CFastVector” ....

不同于 C++类的概念，传统的 C 结构体只是一种将一组数据捆绑在一起的方式。传统 C 结构体的命名规则为：

传统 C 结构体的命名	传统 C 结构体的名称全部由大写字母组成，单词间使用下划线界定，例如：“SERVICE_STATUS”，“DRIVER_INFO”……
-------------	---

## 函数

函数的命名	函数的名称由一个或多个单词组成。为便于界定，每个单词的首字母要大写。
推荐的组成形式	函数名应当使用“动词”或者“动词+名词”（动宾词组）的形式。例如：“GetName()”，“SetValue()”，“Erase()”，“Reserve()”……
保护成员函数	保护成员函数的开头应当加上一个下划线“_”以示区别，例如：“_SetState()”……
私有成员函数	类似地，私有成员函数的开头应当加上两个下划线“__”，例如：“__DestroyImp()”……
虚函数	虚函数习惯以“Do”开头，如：“DoRefresh()”，“_DoEncryption()”……
回调和事件处理函数	回调和事件处理函数习惯以单词“On”开头。例如：“_OnTimer()”，“OnExit()”……

## 变量

变量应该是程序中使用最多的标识符了，变量的命名规范可能是一套 C++命名准则中最重要的部分：

变量的命名	变量名由作用域前缀+类型前缀+一个或多个单词组成。为便于界定，每个单词的首字母要大写。							
	对于某些用途简单明了的局部变量，也可以使用简化的方 式，如：i, j, k, x, y, z……							
作用域前缀	作用域前缀标明一个变量的可见范围。作用域可以有如下几种：							
	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="padding: 2px;">前缀</th> <th style="padding: 2px;">说明</th> </tr> </thead> <tbody> <tr> <td style="padding: 2px;">无</td> <td style="padding: 2px;">局部变量</td> </tr> <tr> <td style="padding: 2px;">m_</td> <td style="padding: 2px;">类的成员变量（member）</td> </tr> </tbody> </table>	前缀	说明	无	局部变量	m_	类的成员变量（member）	
前缀	说明							
无	局部变量							
m_	类的成员变量（member）							

	<table border="1"> <tr> <td>sm_</td><td>类的静态成员变量 (static member)</td></tr> <tr> <td>s_</td><td>静态变量 (static)</td></tr> <tr> <td>g_</td><td>外部全局变量 (global)</td></tr> <tr> <td>sg_</td><td>静态全局变量 (static global)</td></tr> <tr> <td>gg_</td><td>进程间共享的共享数据段全局变量 (global global)</td></tr> </table> <p>除非不得已，否则应该尽可能少使用全局变量。</p>	sm_	类的静态成员变量 (static member)	s_	静态变量 (static)	g_	外部全局变量 (global)	sg_	静态全局变量 (static global)	gg_	进程间共享的共享数据段全局变量 (global global)												
sm_	类的静态成员变量 (static member)																						
s_	静态变量 (static)																						
g_	外部全局变量 (global)																						
sg_	静态全局变量 (static global)																						
gg_	进程间共享的共享数据段全局变量 (global global)																						
类型前缀	<p>类型前缀标明一个变量的类型，可以有如下几种：</p> <table border="1"> <thead> <tr> <th>前缀</th><th>说明</th></tr> </thead> <tbody> <tr> <td>n</td><td>整型和位域变量 (number)</td></tr> <tr> <td>e</td><td>枚举型变量 (enumeration)</td></tr> <tr> <td>c</td><td>字符型变量 (char)</td></tr> <tr> <td>b</td><td>布尔型变量 (bool)</td></tr> <tr> <td>f</td><td>浮点型变量 (float)</td></tr> <tr> <td>p</td><td>指针型变量和迭代子 (pointer)</td></tr> <tr> <td>pfn</td><td>特别针对指向函数的指针变量和函数对象指针 (pointer of function)</td></tr> <tr> <td>g</td><td>数组 (grid)</td></tr> <tr> <td>i</td><td>类的实例 (instance)</td></tr> <tr> <td></td><td>对于经常用到的类，也可以定义一些专门的前缀，如： std::string 和 std::wstring 类的前缀可以定义为"st"， std::vector 类的前缀可以定义为"v"或"vec"等等。</td></tr> </tbody> </table> <p>类型前缀可以组合使用，例如"gc"表示字符数组，"ppn"表示指向整型的指针等等。</p>	前缀	说明	n	整型和位域变量 (number)	e	枚举型变量 (enumeration)	c	字符型变量 (char)	b	布尔型变量 (bool)	f	浮点型变量 (float)	p	指针型变量和迭代子 (pointer)	pfn	特别针对指向函数的指针变量和函数对象指针 (pointer of function)	g	数组 (grid)	i	类的实例 (instance)		对于经常用到的类，也可以定义一些专门的前缀，如： std::string 和 std::wstring 类的前缀可以定义为"st"， std::vector 类的前缀可以定义为"v"或"vec"等等。
前缀	说明																						
n	整型和位域变量 (number)																						
e	枚举型变量 (enumeration)																						
c	字符型变量 (char)																						
b	布尔型变量 (bool)																						
f	浮点型变量 (float)																						
p	指针型变量和迭代子 (pointer)																						
pfn	特别针对指向函数的指针变量和函数对象指针 (pointer of function)																						
g	数组 (grid)																						
i	类的实例 (instance)																						
	对于经常用到的类，也可以定义一些专门的前缀，如： std::string 和 std::wstring 类的前缀可以定义为"st"， std::vector 类的前缀可以定义为"v"或"vec"等等。																						
推荐的组成形式	变量的名字应当使用“名词”或者“形容词+名词”。例如：“nCode”， “m_nState”， “nMaxWidth” ....																						

## 常量

C++中引入了对常量的支持，常量的命名规则如下：

常量的命名	常量名由 <b>类型前缀十全大写字母</b> 组成，单词间通过下划线来界定，如： <b>cDELIMITER, nMAX_BUFFER ...</b>  类型前缀的定义与变量命名规则中的相同。
-------	--

## 枚举、联合、typedef

枚举、联合及 typedef 语句都是定义新类型的简单手段，它们的命名规则为：

枚举、联合、typedef 的命名	枚举、联合、typedef 语句生成的类型名由全大写字母组成，单词间通过下划线来界定，如： <b>FAR_PROC, ERROR_TYPE ...</b>
-------------------	---

## 宏、枚举值

宏、枚举值的命名	宏和枚举值由全大写字母组成，单词间通过下划线来界定，如： <b>ERROR_UNKNOWN, OP_STOP ...</b>
----------	--

## 名空间

C++名空间是“类”概念的一种退化（相当于只包含静态成员且不能实例化的类）。它的引入为标识符名称提供了更好的层次结构，使标识符看起来更加直观简捷，同时大大降低了名字冲突的可能性。

名空间的命名规则包括：

名空间的命名	<p>名空间的名称不应该过长，通常都使用缩写的形式来命名。</p> <p>例如，一个图形库可以将其所有外部接口存放在名空间“<b>GLIB</b>”中，但是将其换成“<b>GRAPHIC_LIBRARY</b>”就不大合适。</p> <p>如果碰到较长的名空间，为了简化程序书写，可以使用：</p> <pre style="background-color: #e0e0e0; padding: 5px;"><code>namespace new_name = old_long_name;</code></pre> <p>语句为其定义一个较短的别名。</p>
--------	--

## 代码风格与版式

代码风格的重要性怎么强调都不过分。一段稍长一点的无格式代码基本上是不可读的。先来看一下这方面的整体原则：

空行的使用	<p>空行起着分隔程序段落的作用。空行得体（不过多也不过少）将使程序的布局更加清晰。空行不会浪费内存，虽然打印含有空行的程序是会多消耗一些纸张，但是值得。所以不要舍不得用空行。</p> <ul style="list-style-type: none"> <li>在每个类声明之后、每个函数定义结束之后都要加 2 行空行。</li> <li>在一个函数体内，逻辑上密切相关的语句之间不加空行，其它地方应加空行分隔。</li> </ul>
语句与代码行	<ul style="list-style-type: none"> <li>一行代码只做一件事情，如只定义一个变量，或只写一条语句。这样的代码容易阅读，并且方便于写注释。</li> <li>“if”、“for”、“while”、“do”、“try”、“catch” 等语句自占一行，执行语句不得紧跟其后。不论执行语句有多少都要加 “{ }”。这样可以防止书写失误。</li> </ul>
缩进和对齐	<ul style="list-style-type: none"> <li>程序的分界符 “{” 和 “}” 应独占一行并且位于同一列，同时与引用它们的语句左对齐。</li> <li>“{ }” 之内的代码块在 “{” 右边一个制表符（4 个半空格符或 TAB）处左对齐。如果出现嵌套的 “{ }”，则使用缩进对齐。</li> <li><b>如果一条语句会对其后的多条语句产生影响的话，应该只对该语句做半缩进（2 个半角空格符），以突出该语句。</b></li> </ul>

例如：

```

void
Function(int x)
{
    CSessionLock iLock(*m_psemLock);

    for (初始化; 终止条件; 更新)
    {
        // ...
    }

    try
    {
        // ...
    }

    catch (const exception& err)
}

```

	<pre>{     // ... }  catch (...)  {     // ... }  // ... }</pre>
最大长度	代码行最大长度宜控制在 70 至 80 个字符以内。代码行不要过长，否则眼睛看不过来，也不便于打印。
长行拆分	<p>长表达式要在低优先级操作符处拆分成新行，操作符放在新行之首（以便突出操作符）。拆分出的新行要进行适当的缩进，使排版整齐，语句可读。</p> <p>例如：</p> <pre>if ((very_longer_variable1 &gt;= very_longer_variable2)     &amp;&amp; (very_longer_variable3 &lt;= very_longer_variable4)     &amp;&amp; (very_longer_variable5 &lt;= very_longer_variable6)) {     dosomething(); }</pre>
空格的使用	<ul style="list-style-type: none"> <li>关键字之后要留空格。象 “const”、“virtual”、“inline”、“case” 等关键字之后至少要留一个空格，否则无法辨析关键字。象 “if”、“for”、“while”、“catch” 等关键字之后应留一个空格再跟左括号 “(”，以突出关键字。</li> <li>函数名之后不要留空格，紧跟左括号 “(”，以与关键字区别。</li> <li>“(” 向后紧跟。而 “)”、“,”、“;” 向前紧跟，紧跟处不留空格。</li> <li>“,” 之后要留空格，如 Function(x, y, z)。如果 “;” 不是一行的结束符号，其后要留空格，如 for (initialization; condition; update)。</li> <li>赋值操作符、比较操作符、算术操作符、逻辑操作符、位域操作符，如 “=”、“+=”、“&gt;=”、“&lt;=”、“+”、“*”、“%”、“&amp;&amp;”、“  ”、“&lt;&lt;”、“^” 等二元操作符的前后应当加空格。</li> <li>一元操作符如 “!”、“~”、“++”、“--”、“&amp;”（地址运算符）等前后不加空格。</li> <li>象 “[”、“.”、“-&gt;”这类操作符前后不加空格。</li> <li>对于表达式比较长的 for、do、while、switch 语句和 if 语句，为了紧凑起见可以适当地去掉一些空格，如 for (i=0; i&lt;10;</li> </ul>

	<p>i++ 和 if ((a&lt;=b) &amp;&amp; (c&lt;=d))</p> <p>例如：</p> <pre> void Func1(int x, int y, int z); // 良好的风格 void Func1 (int x,int y,int z); // 不良的风格  // ===== if (year &gt;= 2000) // 良好的风格 if(year&gt;=2000) // 不良的风格 if ((a&gt;=b) &amp;&amp; (c&lt;=d)) // 良好的风格 if(a&gt;=b&amp;&amp;c&lt;=d) // 不良的风格  // ===== for (i=0; i&lt;10; i++) // 良好的风格 for(i=0;i&lt;10;i++) // 不良的风格 for (i = 0; I &lt; 10; i++) // 过多的空格  // ===== x = a &lt; b ? a : b; // 良好的风格 x=a&lt;b?a:b; // 不好的风格  // ===== int* x = &amp;y; // 良好的风格 int * x = &amp; y; // 不良的风格  // ===== array[5] = 0; // 不要写成 array [ 5 ] = 0; a.Function(); // 不要写成 a . Function(); b-&gt;Function(); // 不要写成 b -&gt; Function();</pre>
修饰符的位置	<ul style="list-style-type: none"> <li>为便于理解，应当将修饰符 “*” 和 “&amp;” 紧靠数据类型。</li> </ul> <p>例如：</p> <pre> char* name;  int* x, y; // 为避免 y 被误解为指针，这里必须分行写。  int* Function(void* p);</pre>

	<p>参见：变量、常量的风格与版式 → 指针或引用类型的定义和声明</p>										
注释	<ul style="list-style-type: none"> <li>• 注释的位置应与被描述的代码相邻，可以放在代码的上方或右方，不可放在下方。</li> <li>• 边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性。不再有用的注释要删除。</li> <li>• 注释应当准确、易懂，防止注释有二义性。错误的注释不但无益反而有害。</li> <li>• 当代码比较长，特别是有多重嵌套时，应当在一些段落的结束处加注释，便于阅读。</li> </ul>										
与常量的比较	<p>在与宏、常量进行 “==”， “!=”， “&gt;=”， “&lt;=” 等比较运算时，应当将常量写在运算符左边，而变量写在运算符右边。这样可以避免因为偶然写错把比较运算变成了赋值运算的问题。</p> <p>例如：</p> <pre>if (NULL == p) // 如果把 “==” 错打成 “=”，编译器就会报错 {     // ... }</pre>										
为增强代码的可读性而定义的宏	<p>以下预定义宏对程序的编译没有任何影响，只为了增加代码的可读性：</p> <table border="1"> <thead> <tr> <th>宏</th> <th>说明</th> </tr> </thead> <tbody> <tr> <td>NOTE</td> <td>需要注意的代码</td> </tr> <tr> <td>TODO</td> <td>尚未实现的接口、类、算法等</td> </tr> <tr> <td>FOR_DBG</td> <td>标记为调试方便而临时增加的代码</td> </tr> <tr> <td>OK</td> <td>仅用于调试的标记</td> </tr> </tbody> </table> <p>例如：</p> <pre>TODO class CMyClass; TODO void Function(void);  FOR_DBG cout &lt;&lt; "...";</pre>	宏	说明	NOTE	需要注意的代码	TODO	尚未实现的接口、类、算法等	FOR_DBG	标记为调试方便而临时增加的代码	OK	仅用于调试的标记
宏	说明										
NOTE	需要注意的代码										
TODO	尚未实现的接口、类、算法等										
FOR_DBG	标记为调试方便而临时增加的代码										
OK	仅用于调试的标记										

## 类/结构

类是 C++中最重要也是使用频率最高的新特性之一。类的版式好坏将极大地影响代码品质。

质。

注释头 与类声 明	<p>与文件一样，每个类应当有一个注释头用来说明该类的各个方面。</p> <p>类声明换行紧跟在注释头后面，“class”关键字由行首开始书写，后跟类名称。界定符 “{” 和 “}；” 应独占一行，并与 “class” 关键字左对其。</p> <pre style="background-color: #f0f0f0; padding: 10px;"> /*! @class ***** * &lt;PRE&gt; 类名称 :CXXX 功能 :&lt;简要说明该类所完成的功能&gt; 异常类 :&lt;属于该类的异常类（如果有的话）&gt;  ----- 备注 :&lt;使用该类时需要注意的问题（如果有的话）&gt; 典型用法 :&lt;如果该类的使用方法较复杂或特殊，给出典型的代码例子&gt;  ----- 作者 :&lt;xxx&gt; &lt;/PRE&gt; *****</pre> <p><code>class CXXX</code></p> <p>{     // ... };</p>
继承	<p>基类直接跟在类名称之后，不换行，访问说明符(public, private, 或 protected)不可省略。如：</p> <pre style="background-color: #f0f0f0; padding: 10px;"> class CXXX : public CAAA, private CBBB {     // ... };</pre>
以行为 为中心	<p>没人喜欢上来就看到一大堆私有数据，大多数用户关心的是类的接口与其提供的服务，而不是其实现。</p>

	所以应当将公有的定义和成员放在类声明的最前面，保护的放在中间，而私有的摆在最后。
访问说明符	访问说明符（public, private, 或 protected）应该独占一行，并与类声明中的‘class’关键字左对齐。
类成员的声明版式	<p>对于比较复杂（成员多于 20 个）的类，其成员必须分类声明。</p> <p>每类成员的声明由访问说明符（public, private, 或 protected）+ 全行注释开始。注释不满全行（80 个半角字符）的，由 “/” 字符补齐，最后一个 “/” 字符与注释间要留一个半角空格符。</p> <p>如果一类声明中有很多组功能不同的成员，还应该用分组注释将其分组。分组注释也要与 “class” 关键字对齐。</p> <p>每个成员的声明都应该由 “class” 关键字开始向右缩进一个制表符（4 个半角空格符），成员之间左对齐。</p> <p>例如：</p> <pre> class CXXX { public:     //////////////////////////////// 类型定义     typedef vector&lt;string&gt; VSTR;  public:     //////////////////////////////// 构造、析构、初始化     CXXX();     ~CXXX();  public:     //////////////////////////////// 公用方法      // [[ 功能组 1     void Function1(void) const;     long Function2(IN int n);     // ]] 功能组 1      // [[ 功能组 2     void Function3(void) const;     bool Function4(OUT int&amp; n);     // ]] 功能组 2 </pre>

	<pre> <b>private:</b> /// 属性 // ...  <b>private:</b> /// 禁用的方法 // 禁止复制 <b>CXXX(IN const CXXX&amp; rhs);</b> <b>CXXX&amp; operator=(IN const CXXX&amp; rhs);</b> }; </pre>
正确地使用 <code>const</code> 和 <code>mutable</code>	<p>把不改变对象逻辑状态的成员都标记为 <code>const</code> 成员不仅有利于用户对成员的理解，更可以最大化对象使用方式的灵活性及合理性（比如通过 <code>const</code> 指针或 <code>const</code> 引用的形式传递一个对象）。</p> <p>如果某个属性的改变并不影响该对象逻辑上的状态，而且这个属性需要在 <code>const</code> 方法中被改变，则该属性应该标记为 “<code>mutable</code>”。</p> <p>例如：</p> <pre> class CString { public:     //! 查找一个子串, find()不会改变字符串的值所以为 const 函数     int find(<b>IN const CString&amp;</b> str) <b>const</b>;     // ...  <b>private:</b>     // 最后一次错误值, 改动这个值不会影响对象的逻辑状态,     // 像 find()这样的 const 函数也可能修改这个值     <b>mutable</b> int m_nLastError;     // ... }; </pre>
嵌套的类声明	<p>在相应的逻辑关系确实存在时，类声明可以嵌套。嵌套类可以使用简单的单行注释头：</p> <pre> class CXXX {     //! 嵌套类说明     class CYYY     {         // ...     }; }; </pre>

初始化列表	<p>应当尽可能通过构造函数的初始化列表来初始化成员和基类。初始化列表至少独占一行，并且与构造函数的定义保持一个制表符（4个半角空格）的缩进。</p>
<p>例如：</p> <pre>CXXX::CXXX(IN int nA, IN bool bB)     : m_nA(nA), m_bB(bB) {     // ... };</pre>	<p>初始化列表的书写顺序应当与对象的构造顺序一致，即：先按照声明顺序写基类初始化，再按照声明顺序写成员初始化。</p>
	<p>如果一个成员 “a” 需要使用另一个成员 “b” 来初始化，则 “b” 必须在 “a” 之前声明，否则将会产生运行时错误（有些编译器会给出警告）。</p> <p>例如：</p> <pre>// ...  class CXXX : public CAA, public CBB {     // ...     CYY m_iA;     CZZ m_iB; // m_iA 必须在 m_iB 之前声明 };</pre> <pre>CXXX::CXXX(IN int nA, IN int nB, IN bool bC)     : CAA(nA), CBB(nB), m_iA(bC), m_iB(m_iA) // 先基类，后成员,   // 分别按照声明顺序书写 {     // ... };</pre>

关于类声明的模板，请参照“example.cpp”。

## 函数

函数是程序执行的最小单位，任何一个有效的 C/C++ 程序都少不了函数。

函数原型	<p>函数原型的格式为：</p> <pre>[存储类] 返回值类型 [名空间或类]::函数名(参数列表) [const 说明符] [异常过滤器]</pre> <p>例如：</p> <pre>static inline void Function1(void)  int CSem::Function2(<b>IN</b> const char* pcName) const throw(<b>Exp</b>)</pre> <p>其中：</p> <ul style="list-style-type: none"> <li>• 以 “[ ]” 括住的为可选项目。</li> <li>• 除了构造/析构函数外，“返回值类型” 和 “参数列表” 项不可省略（可以为 “void”）。</li> <li>• “const 说明符” 仅用于成员函数中</li> <li>• “存储类”，“参数列表” 和 “异常过滤器”的说明见下文</li> </ul>
函数声明	<p>函数声明的格式为：</p> <pre><b>//! 函数功能简单说明 (可选)</b> <b>函数原型;</b></pre> <p>例如：</p> <pre><b>//! 执行某某操作</b> <b>static void Function(void);</b></pre> <p>函数声明和其它代码间要有空行分割。</p> <p>声明成员函数时，为了紧凑，返回值类型和函数名之间不用换行，也可以适当减少声明间的空行。</p>
函数定义	<p>函数定义使用如下格式：参照 “example.cpp”</p> <pre>*****  // FullName: get_entity_length  // Note:      注明函数的作用</pre>

```

// Access:    public/protected/provite
// Parameter: Type      Value      I/O  Note
//           char *     filename   I  读入文件名
//           ENTITY *   ent       I/O  实体
// Returns:   Type  Value  Note
//           bool   TRUE   处理成功
//           FALSE  处理失败
// Qualifier:
// Create: 20090318 Lixiaodong KMAS_PART_1
// Update: 20090319 Lixiaodong KMAS_PART_1 加入数据取得操作
// Update:
//*****
函数原型
bool get_entity_length(char* filename, ENTITY* ent)
{
    //
}

```

对于返回值、参数意义都很明确简单函数（代码不超过 20 行），也可以使用单行函数头：

```

//! 函数实现功能
函数原型
{
    // ...
}

```

函数定义和其它代码之间至少分开 2 行空行。

#### 参数描述宏

以下预定义宏对程序的编译没有任何影响，只为了增强对参数的理解：

宏	说明
IN	输入参数
OUT	输出参数
OPTIONAL	可选参数—通常指可以为 NULL 的指针参数，带默认值的参数不需要这样标明
RESERVED	这个参数当前未被支持，留待以后扩展
OWNER	获得参数的所有权，调用者不再负责销毁参数指定的对象
UNUSED	标明这个参数在此版本中已不再使用
CHANGED	参数类型发出变化

	<table border="1"> <tr> <td>ADDED</td><td>新增的参数</td></tr> <tr> <td>NOTE</td><td>需要注意的参数—参数意义发生变化</td></tr> </table>	ADDED	新增的参数	NOTE	需要注意的参数—参数意义发生变化
ADDED	新增的参数				
NOTE	需要注意的参数—参数意义发生变化				
其中：					
<ul style="list-style-type: none"> <li>• 除了空参数 “void” 以外，每个参数左侧都必须有 “IN” 和/或 “OUT” 修饰</li> <li>• 既输入又输出的参数应记为：“IN OUT”，而不是 “OUT IN”</li> <li>• IN/OUT 的左侧还可以根据需要加入一个或多个上表中列出的其它宏</li> </ul>					
<p>参数描述宏的使用思想是：只要一个宏可以用在指定参数上（即：对这个参数来说，用这个描述宏修饰它是贴切的），那么就应当使用它。</p> <p>也就是说，应该把能用的描述宏都用上，以期尽量具体地描述一个参数。</p>					
参数列表	<p>参数列表的格式为：</p> <pre>参数描述宏 1 参数类型 1 参数 1, 参数描述宏 2 参数类型 2 参数 2, ...</pre> <p>例如：</p> <pre>IN const int nCode, OUT string&amp; nName OWNER IN CDatabase* pDB, OPTIONAL IN OUT int* pRecordCount = NULL IN OUT string&amp; stRuleList, RESERVED IN int nOperate = 0 ...</pre>				
其中：					
<ul style="list-style-type: none"> <li>• “参数描述宏” 见上文</li> <li>• 参数命名规范与变量的命名规范相同</li> </ul>					
存储类	“extern”， “static”， “inline” 等函数存储类说明应该在声明和定义中一致并且显式地使用。不允许隐式地使用一个类型声明，也不允许一个类型声明仅存在于函数的声明或定义中。				
成员函数的存储类	由于 C++ 语言的限制，成员函数的 “static”， “virtual”， “explicit” 等存储类说明不允许出现在函数定义中。  但是为了明确起见，这些存储类应以注释的形式在定义中给出。				

	<p>例如：</p> <pre><code>/*virtual*/ CThread::EXITCODE CSrvCtl::CWrkTrd::Entry(void) {     // ... }  /*static*/ inline void stringEx::regex_free(IN OUT void*&amp; pRegEx) {     // ... }</code></pre>
	<p>特别地，为缩短声明的长度，“inline”关键字可以在成员函数声明中省略。</p>
默认参数	<p>类似地，参数的默认值只能出现在函数声明中，但是为了明确起见，这些默认值应以注释的形式在定义中给出。</p> <p>例如：</p> <pre><code>bool stringEx::regex_find(OUT VREGEXRESULT&amp; vResult,     IN stringEx stRegEx,     IN size_t nIndex /*= 0*/,     IN size_t nStartPos /*= 0*/,     IN bool bNoCase /*= false*/,     IN bool bNewLine /*= true*/,     IN bool bExtended /*= true*/,     IN bool bNotBOL /*= false*/,     IN bool bNotEOL /*= false*/,     IN bool bUsePerlStyle /*= false*/) const {     // ... }</code></pre>
异常过滤器	<p>对于任何可能抛出异常的函数，必须在其声明和定义中显式地指定异常过滤器，并在过滤器中列举该函数可能抛出的异常。</p> <p>例如：</p> <pre><code>int Function(IN const char* pcName) throw(byExp, exception);</code></pre>
代码段注释	<p>如果函数体中的代码较长，应该根据功能不同将其分段。代码段间以空行分离，并且每段代码都以代码段分割注释作为开始。</p>

例如：

```

Void CXXX::Function(IN void* pmodAddr)
{
    if (NULL == pmodAddr)
        return;

    { CSessionLock iLock(*sm_hSELock);

        // =====
        // = 判断指定模块是不是刚刚被装入，由于在 NT 系列平台中，“A”系列函数
        // 都是
        // = 由“W”系列函数实现的。所以可能会有一次 LoadLibrary 产生多次本函
        // 数调
        // = 用的情况。为了增加效率，特设此静态变量判断上次调用是否与本次相
        // 同。
        static PVOID pLastLoadedModule = NULL;
        if (pLastLoadedModule == pmodAddr)
        {
            return; // 相同，忽略这次调用
        }
        pLastLoadedModule = pmodAddr;

        // =====
        // = 检查这个模块是否在旁路模块表中
        stringEx stModName;
        if (!BaiY_IMP::GetModuleNameByAddress(pmodAddr, stModName))
        {
            return;
        }

        if (CHookProc::sm_sstByPassModTbl.find(stModName)
            != CHookProc::sm_sstByPassModTbl.end())
        {
            return;
        }

        // =====
        // = 在这个模块中 HOOK 所有存在于 HOOK 函数表中的函数
        PROCTBL::iterator p;
        for (p=sm_iProcTbl.begin(); p!=sm_iProcTbl.end

```

	<pre>         }     } // SessionLock } </pre>
	<p>明显地，如果需要反复用到一段代码的话，这段代码就应当作为一个函数实现。</p> <p>当一个函数过长时（超过 100 行），为了便于阅读和理解，也应当将其中的一些代码段实现为单独的函数。</p>
调用系统 API	<p>所有系统 API 调用前都要加上全局名称解析符 “::”。</p> <p>例如：</p> <pre> ::MessageBoxA(NULL, gcErrorMsg, "!FATAL ERROR!", MB_ICONSTOP MB_OK);  if (0 == ::GetTempFileName(m_basedir.c_str(), byT("bai"), 0, stR.ref())) {     // ... } </pre>

## 变量、常量

声明格式	<p>变量、常量的声明格式如下：</p> <pre>[存储类] 类型 变量名;</pre> <p>其中：</p> <ul style="list-style-type: none"> <li>• 以 “[ ]” 括住的为可选项目。</li> <li>• “存储类”的说明见下文</li> </ul>
定义格式	<p>变量、常量的定义格式如下：</p> <pre>[存储类] 类型 变量名 = 初始值;</pre> <p>其中：</p> <ul style="list-style-type: none"> <li>• 以 “[ ]” 括住的为可选项目。</li> <li>• “存储类”的说明见下文</li> </ul>

存储类	除 "auto" 类型以外，诸如 "extern", "static", "register", "volatile" 等存储类均不可省略，且必须在声明和定义中一致地使用（即：不允许仅在声明或定义中使用）。
成员变量的存储类	<p>由于 C++ 语言的限制，成员变量的 "static" 等存储类说明不允许出现在变量定义中。</p> <p>但是为了明确起见，这些存储类应以注释的形式在定义中给出。</p> <p>例如：</p> <pre data-bbox="462 631 890 669">/*static*/ int CThread::sm_nPID = 0;</pre>
指针或引用类型的定义和声明	<p>在声明和定义多个指针或引用变量/常量时，每个变量至少占一行。例如：</p> <pre data-bbox="462 878 679 990">int* pn1,     *pn2 = NULL,     *pn3;</pre> <pre data-bbox="462 1046 589 1158">char* pc1; char* pc2; char* pc3;</pre> <p>// 错误的写法：</p> <pre data-bbox="462 1248 759 1282">int* pn11, *pn12, *pn13;</pre>
常指针和指针常量	<p>声明/定义一个常指针（指向常量的指针）时，“const”关键字一律放在类型说明的左侧。</p> <p>声明/定义一个指针常量（指针本身不能改变）时，“const”关键字一律放在变量左侧、类型右侧。</p> <p>例如：</p> <pre data-bbox="462 1720 890 1832">const char* pc1; // 常指针 char* const pc2; // 指针常量 const char* const pc3; // 常指针常量</pre> <p>// 错误的写法：</p> <pre data-bbox="462 1922 1276 1956">char const* pc1; // 与 const char* pc1 含义相同，但不允许这样写</pre>
全局变量、常量	全局变量、常量的注释独占一行，并用 “//!” 开头。

的注释	<p>例如：</p> <pre>//! 当前进程的 ID static int sg_nPID = 0;  //! 分割符 static const char* pcDTR = "\V";</pre>								
类型转换	<p>禁止使用 C 风格的 “(类型)” 格式转换，应当优先使用 C++ 的 “xxx_cast” 风格的类型转换。C++ 风格的类型转换可以提供丰富的含义和功能，以及更好的类型检查机制，这对代码的阅读、修改、除错和移植有很大的帮助。</p> <p>其中：</p> <table border="1" data-bbox="462 765 1314 1500"> <tbody> <tr> <td data-bbox="462 765 779 900">static_cast</td><td data-bbox="779 765 1314 900">static_cast 用于编译器认可的，安全的静态转换，比如将 “char” 转为 “int” 等等。该操作在编译时完成</td></tr> <tr> <td data-bbox="462 900 779 1091">reinterpret_cast</td><td data-bbox="779 900 1314 1091">reinterpret_cast 用于编译器不认可的，不安全的静态转换，比如将 “int*” 转为 “int” 等等。这种转换有可能产生移植性方面的问题，该操作在编译时完成</td></tr> <tr> <td data-bbox="462 1091 779 1271">const_cast</td><td data-bbox="779 1091 1314 1271">const_cast 用于将一个常量转化为相应类型的变量，比如将 “const char*” 转换成 “char*” 等等。这种转换通常伴随潜在的错误。该操作在编译时完成</td></tr> <tr> <td data-bbox="462 1271 779 1500">dynamic_cast</td><td data-bbox="779 1271 1314 1500">dynamic_cast 是 C++RTTI 机制的重要体现，用于在类层次结构中漫游。dynamic_cast 可以对指针和引用进行自由度很高的向上、向下和交叉转换。被正确使用的 dynamic_cast 操作将在运行时完成</td></tr> </tbody> </table> <p>此外，对于定义了 <code>单参构造函数</code> 或 <code>类型转换操作</code> 的类来说，应当优先使用构造函数风格的类型转换，如：<code>“string(“test”)”</code> 等等。</p> <p>通常来说，“xxx_cast” 格式的转换与构造函数风格的类型转换之间，最大的区别在于：构造函数风格的转换通常会生成新的临时对象，可能伴随相当的时间和空间开销。</p> <p>而 “xxx_cast” 格式的转换只是告诉编译器，将指定内存中的数据当作另一种类型的数据看待，这些操作一般在编译时完成，不会对程序的运行产生额外开销。当然，“dynamic_cast” 则是一个例外。</p>	static_cast	static_cast 用于编译器认可的，安全的静态转换，比如将 “char” 转为 “int” 等等。该操作在编译时完成	reinterpret_cast	reinterpret_cast 用于编译器不认可的，不安全的静态转换，比如将 “int*” 转为 “int” 等等。这种转换有可能产生移植性方面的问题，该操作在编译时完成	const_cast	const_cast 用于将一个常量转化为相应类型的变量，比如将 “const char*” 转换成 “char*” 等等。这种转换通常伴随潜在的错误。该操作在编译时完成	dynamic_cast	dynamic_cast 是 C++RTTI 机制的重要体现，用于在类层次结构中漫游。dynamic_cast 可以对指针和引用进行自由度很高的向上、向下和交叉转换。被正确使用的 dynamic_cast 操作将在运行时完成
static_cast	static_cast 用于编译器认可的，安全的静态转换，比如将 “char” 转为 “int” 等等。该操作在编译时完成								
reinterpret_cast	reinterpret_cast 用于编译器不认可的，不安全的静态转换，比如将 “int*” 转为 “int” 等等。这种转换有可能产生移植性方面的问题，该操作在编译时完成								
const_cast	const_cast 用于将一个常量转化为相应类型的变量，比如将 “const char*” 转换成 “char*” 等等。这种转换通常伴随潜在的错误。该操作在编译时完成								
dynamic_cast	dynamic_cast 是 C++RTTI 机制的重要体现，用于在类层次结构中漫游。dynamic_cast 可以对指针和引用进行自由度很高的向上、向下和交叉转换。被正确使用的 dynamic_cast 操作将在运行时完成								

## 枚举、联合、typedef

枚举、联合的定义格式

枚举、联合的定义格式为:

```
//! 说明（可选）
enum|union 名称
{
    内容
};
```

例如:

```
//! 服务的状态
enum SRVSTATE
{
    SRV_INVALID = 0,
    SRV_STARTING = 1,
    SRV_STARTED,
    SRV_PAUSING,
    SRV_PAUSED,
    SRV_STOPPING,
    SRV_STOPPED
};

//! 32 位整数
union INT32
{
    unsigned char cByte[4];
    unsigned short nShort[2];
    unsigned long nFull;
};
```

typedef 的定义格式

typedef 的定义格式为:

```
//! 说明（可选）
typedef 原类型 新类型;
```

例如:

```
//! 返回值类型
typedef int EXITCODE;
```

```
//! 字符串数组类型
typedef vector<string> VSTR;
```

## 宏

何时使用宏	应当尽量减少宏的使用，在所有可能的地方都使用常量和内联函数来代替宏。
边界效应	<p>使用宏的时候应当注意边界效应，例如，以下代码将会得出错误的结果：</p> <pre>#define PLUS(x,y) x+y  cout &lt;&lt; PLUS(1,1) * 2;</pre> <p>以上程序的执行结果将会是 "3"，而不是 "4"，因为 "PLUS(1, 1) * 2" 表达式将被展开为： "1 + 1 * 2"。在定义宏的时候，只要允许，就应该为它的替换内容括上 "( )" 或 "{ }"。例如：</p> <pre>#define PLUS(x,y) (x+y)  #define SAFEDELETE(x) {delete x; x=0}</pre>

## 名空间

名空间的使用	名空间可以避免名字冲突、分组不同的接口以及简化命名规则。应当尽可能地将所有接口都放入适当的名字空间中。
将实现和界面分离	<p>提供给用户的界面和用于实现的细节应当分别放入不同的名空间中。</p> <p>例如：如果将一个软件模块的所有接口都放在名空间 "MODULE" 中，那么这个模块的所有实现细节就可以放入名空间 "MODULE_IMP" 中。</p>

## 异常

异常使 C++ 的错误处理更为结构化；错误传递和故障恢复更为安全简便；也使错误处理代码和其它代码间有效的分离开来。

何时使用异常	<p>异常机制只用在发生错误的时候，仅在发生错误时才应当抛出异常。这样做有助于错误处理和程序动作两者间的分离，增强程序的结构化，还保证了程序的执行效率。</p> <p>确定某一状况是否算作错误有时会很困难。比如：未搜索到某个字符串、等待一个信号量超时等等状态，在某些情况下可能并不算作一个错误，而在另一些情况下可能就是一个致命错误。</p> <p>有鉴于此，仅当某状况必为一个错误时（比如：分配存储失败、创建信号量失败等），才应该抛出一个异常。而对另外一些模棱两可的情况，就应当使用返回值等其它手段报告。</p>
用异常代替 goto 等其它错误处理手段	<p>曾经被广泛使用的传统错误处理手段有 goto 风格和 do...while 风格等，以下是一个 goto 风格的例子：</p> <pre>//! 使用 goto 进行错误处理的例子 bool Function(void) {     int nCode, i;     bool r = false;      // ...      if (!Operation1(nCode))     {         goto onerr;     }      try     {         Operation2(i);     }     catch (...)     {         r = true;         goto onerr;     } }  onerr: // 处理错误逻辑 </pre>

```

    }

r = true;

onerr:
    // ... 清理代码
    return r;
}

```

由上例可见， goto 风格的错误处理至少存在问题如下：

- 错误处理代码和其它代码混杂在一起，使程序不够清晰易读
- 变量必须在 “goto” 之前声明，违反就近原则
- 多处跳转的使用破坏程序的结构化，影响程序的可读性，使程序容易出错
- 对每个会抛出异常的操作都需要用额外的 try...catch 块检测和处理
- 稍微复杂一点的分类错误处理要使用多个标号和不同的 goto 跳转（如：“onOp1Err”，“onOp2Err” ...）。这将使程序变得无法理解和错误百出。

再来看看 do...while 风格的错误处理：

```

//! 使用 do...while 进行错误处理的例子
bool
Function(void)
{
    int nCode, i;
    bool r = false;

    // ...

    do
    {
        if (!Operation1(nCode))
        {
            break;
        }

        do
        {
            try

```

```

    {
        Operation2(i);
    }
    catch (...)
    {
        r = true;
        break;
    }
} while (Operation3())

r = true;

} while (false);

// ... 清理代码
return r;
}

```

与 goto 风格的错误处理相似：

- 错误处理代码和其它代码严重混杂，使程序非常难以理解
- 无法进行分类错误处理
- 对每个会抛出异常的操作都需要用额外的 try...catch 块检测和处理

此外，还有一种更为糟糕的错误处理风格——直接在出错的位置完成错误处理：

```

//! 直接进行错误处理的例子
bool
Function(void)
{
    int nCode, i;

    // ...

    if (!Operation1(nCode))
    {
        // ... 清理代码
        return false;
    }
}

```

```

try
{
    Operation2(i);
}
catch (...)
{
    // ... 清理代码
    return true;
}

// ...

// ... 清理代码
return true;
}

```

这种错误处理方式所带来的隐患可以说是无穷无尽，这里不再列举。

与传统的错误处理方法不同，C++的异常机制很好地解决了以上问题。使用异常做出错处理时，可以将大部分动作都包含在一个 try 块中，并以不同的 catch 块捕获和处理不同的错误：

```

//! 使用异常进行错误处理的例子
bool
Function(void)
{
    int nCode, i;
    bool r = false;

    try
    {
        if (!Operation1(nCode))
        {
            throw false;
        }

        Operation2(i);
    }
    catch (bool err)
    {
        // ...
        r = err;
    }
}

```

```

}
catch (const exception& err)
{
    // ... exception 类错误处理
}
catch (...)
{
    // ... 处理其它错误
}

// ... 清理代码
return r;
}

```

以上代码示例中，错误处理和动作代码完全分离，错误分类清晰明了，好处不言而喻。

#### 构造函数中的异常

在构造函数中抛出异常将中止对象的构造，这将产生一个没有被完整构造的对象。

对于 C++来说，这种不完整的对象将被视为并未创建而不被认可，也意味着其析构函数永远不会被调用。这个行为本身无可非议，就好像公安局不会为一个流产的婴儿发户口一样。但是这有时也会产生一些问题，例如：

```

class CSample
{
    // ...

    char* m_pc;
};

CSample::CSample()
{
    m_pc = new char[256];
    // ...
    throw -1; // m_pc 将永远不会被释放
}

CSample::~CSample() // 析构函数不会被调用
{
    delete m_pc;
}

```

解决这个问题的方法是在抛出异常以前释放任何已被申请的资源。一种更好的方法是使用“资源申请即初始化”的类型（如：句柄类、灵巧指针类等等）来代替一般的指针类型，如：

```
template <class T>
struct CAutoPtr
{
    CAutoPtr(T* p = NULL) : m_p(p) {};
    ~CAutoPtr() {delete m_p;}
    T* operator=(T* rhs)
    {
        if (rhs == m_p)
            return m_p;
        delete m_p;
        m_p = rhs;
        return m_p;
    }
    // ...

    T* m_p;
};

class CSample
{
    // ...

    CAutoPtr<char> m_hc;
};

CSample::CSample()
{
    m_hc = new char[256];
    // ...
    throw -1; // 由于 m_hc 已经成功构造,
    m_hc ~CAutoPtr() 将会
                // 被调用, 所以申请的内存将被释放
}
```

注意：上述 CAutoPtr 类仅用于示范，对于所有权语义的通用自动指针，应该使用 C++ 标准库中的 “auto\_ptr” 模板类。对于带引用计数和自定义销毁策略的通用句柄类，可以使用白杨工具库中的 “CHandle” 模板类。

## 析构函数中的异常

析构函数中的异常可能在 2 种情况下被抛出：

1. 对象被正常析构时
2. 在一个异常被抛出后的退栈过程中——异常处理机制退出一个作用域，其中所有对象的析构函数都将被调用。

由于 C++ 不支持异常的异常，上述第二种情况将导致一个致命错误，并使程序中止执行。例如：

```
class CSample
{
    ~CSample();
    // ...
};

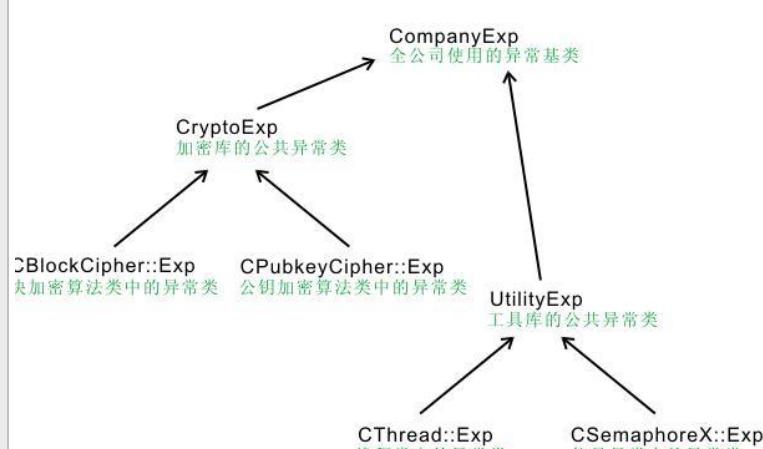
CSample::~CSample()
{
    // ...
    throw -1; // 在 "throw false" 的过程中再次抛出异常
}

void
Function(void)
{
    CSample iTest;
    throw false; // 错误，iTest.~CSample() 中也会抛出异常
}
```

如果必须要在析构函数中抛出异常，则应该在异常抛出前用 “`std::uncaught_exception()`” 事先判断当前是否存在已被抛出但尚未捕获的异常。例如：

```
class CSample
{
    ~CSample();
    // ...
};

CSample::~CSample()
{
    // ...
    if (!std::uncaught_exception()) // 没有尚未捕获的异常
}
```

	<pre> {     throw -1; // 抛出异常 } }  void Function(void) {     CSample iTest;     throw false; // 可以, iTest.&lt;CSample()不会抛出异常 } </pre>
异常的组织	<p>异常类型应该以继承的方式组织成一个层次结构，这将使以不同粒度分类处理错误成为可能。</p> <p>通常，某个软件生产组织的所有异常都从一个公共的基类派生出来。而每个类的异常则从该类所属模块的公共异常基类中派生。例如：</p>  <pre> graph TD     CompanyExp[CompanyExp 全公司使用的异常基类] --&gt; CryptoExp[CryptoExp 加密库的公共异常类]     CompanyExp --&gt; UtilityExp[UtilityExp 工具库的公共异常类]     CryptoExp --&gt; CBlockCipherExp[CBlockCipher::Exp 块加密算法类中的异常类]     CryptoExp --&gt; CPubkeyCipherExp[CPubkeyCipher::Exp 公钥加密算法类中的异常类]     UtilityExp --&gt; CThreadExp[CThread::Exp 线程类中的异常类]     UtilityExp --&gt; CSemaphoreXExp[CSemaphoreX::Exp 信号量类中的异常类] </pre>
异常捕获和重新抛出	<ul style="list-style-type: none"> <li>异常捕获器的书写顺序应当由特殊到一般（先子类后基类），最后才是处理所有异常的捕获器（“catch(...)"）。否则将使某些异常捕获器永远不会被执行。</li> <li>为避免捕获到的异常被截断，异常捕获器中的参数类型应当为常引用型或指针型。</li> <li>在某级异常捕获器中无法被彻底处理的错误可以被重</li> </ul>

新抛出。重新抛出采用一个不带运算对象的“throw”语句。重新抛出的对象就是刚刚被抛出的那个异常，而不是处理器捕获到的（有可能被截断的）异常。

例如：

```
try
{
    // ...
}

// 公钥加密错误
catch (const CPubKeyCipher::Exp& err)
{
    if (可以恢复)
    {
        // 恢复错误
    }
    else
    {
        // 完成能做到的事情
        throw; // 重新抛出
    }
}

// 处理其它加密库错误
catch (const CryptoExp& err)
{
    // ...
}

// 处理其它本公司模块抛出的错误
catch (const CompanyExp& err)
{
    // ...
}

// 处理 dynamic_cast 错误
catch (const bad_cast& err)
{
    // ...
}

// 处理其它标准库错误
catch (const exception& err)
{
    // ...
}

// 处理所有其它错误
```

	<pre><code>catch (...) {     throw; // 重新抛出 }</code></pre>
异常和效率	<p>对于几乎所有现代编译器来说，在不抛出异常的情况下，异常处理的实现在运行时不会有额外开销，也就是说：正常情况下，异常机制比传统的通过返回值判断错误的开销还来得小。</p> <p>相对于函数返回和调用的开销来讲，异常抛出和捕获的开销通常会来得大一些。不过错误处理代码通常不会频繁调用，所以错误处理时开销稍大一点基本上不是什么问题。这也是我们提倡仅将异常用于错误处理的原因之一。</p> <p>更多关于效率的讨论，参见：RTTI、虚函数和虚基类的开销分析和使用指导</p>

## 代码的添加/更新/删除

在代码的开发当中，修改最初的 source 成为经常的工作，这在修正后期 bug 的时候会更突出。如何将各时期对代码的修改有效地记录下来，更好的取得影响范围，作者根据在外企的工作经验，推荐用以下两种方法。（在项目开始时确定用哪种，开发阶段各开发者要严格执行，保持风格不变）

- 方法 1：使用条件编译（#ifdef、#else、#endif）和注释

首先，必须定义编译开关，如

```
#define KMAS_PART_1
```

Add source	格式为“条件编译关键字+编译开关+注释” <pre>#ifdef KMAS_PART_1 /* KMAS_PART_1 Lixiaodong 20090318 */     Added source; #endif /* KMAS_PART_1 Lixiaodong 20090318 */</pre>
Modify source	<pre>#ifdef KMAS_PART_1 /* KMAS_PART_1 Lixiaodong 20090318 */     New source; #else     // Old source; #endif /* KMAS_PART_1 Lixiaodong 20090318 */</pre>
Delete source	<pre>#ifdef KMAS_PART_1 /* KMAS_PART_1 Lixiaodong 20090318 */ #else</pre>

```
// deleted source;
#endif /* KMAS_PART_1 Lixiaodong 20090318 */
```

- 方法 2：直接使用注释

Add source	直接使用注释，只是包含开发项目名、编码人姓名、添加时间、add start (add end)。  /* KMAS_PART_1 Lixiaodong 20090318 add start */ Added source; /* KMAS_PART_1 Lixiaodong 20090318 add end */
Modify source	/* KMAS_PART_1 Lixiaodong 20090318 update start */ New source; // Old source; /* KMAS_PART_1 Lixiaodong 20090318 update end */
Delete source	/* KMAS_PART_1 Lixiaodong 20090318 delete start */ // deleted source; /* KMAS_PART_1 Lixiaodong 20090318 delete end */

## 版本控制

代码使用 SVN 进行，因此版本控制参照 SVN 的使用帮助及手册。

## 关于本规范的贯彻实施

像这样一套完整、详细、繁琐又重要的规范，最难的恐怕就是贯彻实施的环节了。有鉴于此，特提供几点意见：

设立代码审查小组	对于大型开发团队，可以设立专门的代码审查小组，对项目的所有源码进行有计划的审查和评估。
设立交叉审查制度	在程序员间设立源码的交叉审查制度，鼓励大家互相督促。在外企中，这步工作是每一个开发阶段结束之后一定要做的关键工作，叫做 review。
提供专门的检查工具	以 IDE 插件和独立应用程序的形式提供专门的自动化批量代码检查工具，提高审查与自我检查的效率和精度。

## 术语表

术语	解释
API	应用程序编程接口
UI	用户界面
GUI	图形用户界面
IDE	集成开发环境

## 附录二 example.cpp 文件

文件头注释

```
/*-----
```

Copyright 2009 DUT SAE

All rights reserved

File description:

-- Please append file description informations here --

Module Name : <文件所属的模块名称>

File Name : <文件名>

Function : <描述该文件实现的主要功能>

Date	Name	Keyword	Description of Change
2009/03/19	Lixiaodong	KMAS_PART_1	Create.
2009/03/19	Lixiaodong	KMAS_PART_1	Update the Function Max().

\$HISTORY\$

\*/

预处理块

```
#ifndef EXAMPLE_CPP // 防止被重复引用（注意：包括文件的最后一行）
#define EXAMPLE_CPP
```

引用文件的格式

```
#include <stdlib.h> // 引用标准库的头文件
```

```
// ...
```

```
#include "../common.h" // 引用当前工程中的头文件
```

```
// ...
```

### 类声明的模版

```
/*! @class
*****
<PRE>
类名称    :
功能      :
异常类    :
-----
备注      :
典型用法  :
-----
作者      :<xxx>
</PRE>
*****
class CXXX
{
public:
////////// 类型定义

public:
////////// 构造、析构、初始化

public:
////////// 虚函数

public:
////////// 公用方法

public:
////////// 静态方法

protected:
////////// 内部方法

private:
////////// 私有类型定义
```

```

private:
////////////////////////////////////////////////////////////////// 私有方法

private:
////////////////////////////////////////////////////////////////// 属性

private:
////////////////////////////////////////////////////////////////// 静态属性

private:
////////////////////////////////////////////////////////////////// 禁用的方法

};


```

### 函数注释

```

*****
FullName: _get_entity_length
Note:      注明函数的作用
Access:    public/protected/provite
Parameter: Type      Value      I/O   Note
           char *    filename    I    读入文件名
           ENTITY *  ent        I/O  实体
Returns:   Type  Value  Note
           bool  TRUE   处理成功
           FALSE  处理失败
Qualifier:
Create: 20090318 Lixiaodong KMAS_PART_1
Update: 20090319 Lixiaodong KMAS_PART_1 加入数据取得操作
Update:
*****
```

```

bool T::_get_entity_length(char* filename, ENTITY* ent)
{
}


```

### 分割带

```

// #####
// ##### XXXXXXXX 开始#####

```

```
// ##### XXXXXX 结束#####  
// #####
```

```
#endif //EXAMPLE_CPP
```

## 附录三 预定义组件集参考手册

待添加