

# 扩展 Delphi 的 IDE

— Extending the IDE 中译版

原文：Delphi7 官方帮助

翻译：与月共舞 ([master@cnvcl.org](mailto:master@cnvcl.org))

发布：<http://www.cnvcl.org>

版本：V1.0.0.0

创建：2004-06-04

更新：2004-06-12



**CnPack 开发组 [www.cnvcl.org](http://www.cnvcl.org)**

## 目 录

一、	概 述 .....	3
1、	前言 .....	3
2、	术语列表 .....	3
二、	扩展Delphi的IDE .....	4
1、	IDE扩展 .....	4
2、	Tools API概述 .....	4
3、	编写一个专家类 .....	5
4、	实现专家接口 .....	6
5、	安装专家包 .....	6
6、	获得Tools API服务 .....	7
7、	使用本地IDE对象 .....	8
8、	增加图像到图像列表 .....	8
9、	增加Action到Action列表 .....	9
10、	删除工具栏按钮 .....	9
11、	调试专家 .....	10
12、	接口版本号 .....	11
13、	对文件和编辑器的操作 .....	11
14、	使用模块接口 .....	12
15、	使用编辑器接口 .....	12
16、	创建窗体和工程 .....	13
17、	IDE的专家事件通知 .....	16
三、	后 记 .....	20
1、	寻幽访胜靠自己 .....	20

# 一、概 述

## 1、前言

Delphi 的 IDE 扩展是一般程序员很少涉足的领域，不管是网上还是书店里，这方面的资料都是鲜有所见。Delphi7 自带的帮助文件是我们最容易找到的资料，为了方便 CnPack 开发组成员以及对 IDE 扩展感兴趣的朋友对这一领域有更多的认识，我花了点时间把 Delphi7 中 IDE 扩展部分的帮助翻译成中文发布，希望对大家有所帮助。

## 2、术语列表

以下是本文档翻译时使用的术语对照表：

- 插件 (Add-in)，以设计期包或 DLL 形式被设计期的 IDE 调用的扩展工具。
- 专家 (Wizard)，实现了 IOTAWizard 接口的 IDE 插件工具。
- 仓库专家 (Repository Wizard)，用来创建新的单元、窗体或工程的专家。
- 包 (Package)，Delphi 中使用的特殊的动态链接库。
- 设计期包 (Design-time Package)，被编译为允许 IDE 在设计期装载的包。
- 运行期包 (Runtime Package)，被编译为允许 DLL 在运行期调用的包。
- 接口 (Interface)，Delphi 中使用的 COM 风格的接口。
- 通知器 (Notifier)，由用户实现特定接口并由 IDE 在特定事件中调用的用户对象。
- 创建器 (Creator)，由用户实现特定接口的用于创建新的单元、窗体或工程的用户对象。
- 工程 (Project)，Delphi 中的 Project。
- 单元 (Unit)，Delphi 中的 Unit。
- 模块 (Module)，对应着一组在 IDE 中打开的逻辑上关联的文件集，可以是一个单元、包含窗体的单元、工程文件等对象。
- 编辑器 (Editor)，IDE 中用来设计和编辑模块的对象。

其它如 IDE、DLL、Action、Tools API 这样的术语则沿用英文，不作翻译。

## 二、 扩展 Delphi 的 IDE

### 1、 IDE 扩展

通过使用 Open Tools API（通常缩写为 Tools API），你可以用你自己的菜单项、工具栏按钮、动态的窗体创建专家以及更多的东西来扩展和定制 IDE。Tools API 是一套超过 100 个用于关联以及控制 IDE 的接口，包括主菜单、工具栏、主 Action 列表以及图像列表、源代码编辑器内部缓冲区、键盘宏及键盘绑定、窗体设计器中的窗体及其上面的组件、调试器和正在被调试的进程、代码完成、消息视图，以及任务列表。

使用 Tools API 是一件容易的事情，只要写几个实现了特定接口的类，并调用由另一些接口提供的服务即可。你的 Tools API 代码必须编译并作为一个设计期包或 DLL 装载到设计期的 IDE 中。这样，编写一个 Tools API 扩展有些类似编写一个属性或组件编辑器。在阅读这份材料之前，请确信你已经对基本的 [用包来工作](#) 和 [注册组件](#) 比较熟悉了。

下面这些主题描述了怎样使用 Tools API：

- [Tools API 概述](#)
- [编写一个专家类](#)
- [获得 Tools API 服务](#)
- [对文件和编辑器的操作](#)
- [创建窗体和工程](#)
- [IDE 的专家事件通知](#)

### 2、 Tools API 概述

所有的 Tools API 声明都在这一个单元里：ToolsAPI。要使用 Tools API，你通常要引用 designide 这个包，这意味着你需要将你的 Tools API 插件作为设计期包或使用了运行期包的 DLL 来构建。关于包和库的问题，参阅 [安装专家包](#)。

编写一个 Tools API 扩展的主要接口是 *IOTAWizard*，故大部分 IDE 插件都称为专家（Wizards）。在绝大多数情况下，C++Builder 和 Delphi 的专家可以通用。你可以在 Delphi 中编写和编译一个专家，然后在 C++Builder 中使用，反之亦然。这种共用的工作最好在同一个 IDE 版本号之间，但同样也可能编写一个专家并且他们能在两种产品的将来版本里都可使用。要使用 Tools API，你可以编写一个专家类并实现在 ToolsAPI 单元中定义的一个或多个接口。专家可以利用在 Tools API 中提供的服务，每个服务都是一个提供一组相关函数的接口，接口的实现部分被隐藏在 IDE 里面。Tools API 只公布了接口，你可以利用它们来编写你的专家，而不必关心那些接口的实现细节。这些不同的接口提供了对源代码编辑器、窗体设计器、调试器等的访问。怎样在你的专家中使用这些接口包含的服务，参阅 [获得 Tools API 服务](#)。

这些服务和其它的接口被划分为两个基本的分类，你可以按照用为类型名称的前缀来区分它们：

- NTA（native tools API）本地的 Tools API 允许直接访问实际的 IDE 对象，如 IDE 的 *TMainMenu* 对象。当使用这些接口时，专家必须引用 Borland 的包，这意味着专家将限制于特定的 IDE 版本中。这类专家可以放在一个设计期包或使用了运行期包的 DLL 中。

- OTA (open tools API) 开放的 Tools API 不需要引用包, 只能通过接口访问 IDE。在理论上, 如果你能支持 Delphi 的函数调用约定以及类似于 *AnsiString* 这样的 Delphi 类型, 则你能够使用任何支持 COM 风格接口的语言来编写专家, 但是几乎所有的 Tools API 功能都只能通过 OTA 接口获得。如果一个专家只使用 OTA 接口, 则它有可能写成一个不依赖于特定 IDE 版本的 DLL。

Tools API 有两种类型的接口: 一种是作为程序员的你必须实现的接口, 另一种是 IDE 已经实现了的接口。大部分的接口属于后者的分类: 接口定义了 IDE 的功能而隐藏了真正的实现。你必须实现的接口可分为以下三类: 专家 (Wizards)、通知器 (Notifiers) 以及创建器 (Creators):

- 在前面的主题中提到, 一个专家类要实现 *IOTAWizard* 接口以及可能的派生接口。
- 通知器是 Tools API 中另一种类型的接口。IDE 使用通知器在某些你关注的事情发生时回调你的专家。你可以编写一个类来实现通知器接口, 并使用 Tools API 注册该通知器, 当用户打开一个文件、编译源代码、修改窗体、开始调试会话及其它情况时, IDE 会回调你的通知器对象。通知器的介绍见 [IDE 的专家事件通知](#)。
- 创建器是你必须实现的另一种类型的接口。Tools API 使用创建器来创建新的单元、工程或文件, 或用来打开一个已存在的文件。关于创建器的内容请查看 [创建窗体和工程](#) 部分。

其它的重要接口是模块 (Module) 和编辑器 (Editor)。一个模块接口代表了一个打开的单元, 包含一个或多个文件。一个编辑器接口代表一个打开的文件。不同类型的编辑器接口提供给你对 IDE 中不同方面的访问: 源代码编辑器 (Source Editor) 对应着源代码文件, 窗体设计器 (Form Designer) 对应着窗体文件, 另外还有工程资源 (Project Resource) 对应资源文件。关于模块和编辑器的内容请查看 [对文件和编辑器的操作](#) 部分。

### 3、编写一个专家类

一共有四种类型的专家, 专家的类型依赖于专家类所实现的接口。下面的表格描述了这四种类型的专家:

四种类型的专家	
接口	描述
IOTAFormWizard	用来创建新的单元、窗体或其它文件
IOTAMenuWizard	自动增加到 Help 菜单中
IOTAProjectWizard	用来创建一个新的应用程序工程。
IOTAWizard	不适合放在其它分类中的各种专家

这四种类型的专家区别仅在于用户怎样调用专家:

- 菜单型专家 (Menu Wizard) 将增加到 IDE 的 Help 菜单中。当用户选择该菜单项时, IDE 将调用该专家的 *Execute* 方法。普通的专家表现得更为灵活, 故菜单型专家通常只在原型和调试时使用。
- 窗体和工程专家又叫仓库专家, 因为他们被放在对象仓库 (Object Repository) 中。用户在新建项目对话框中调用这些专家, 用户也能在对象仓库 (通过选择 *Tools|Repository* 菜单项) 中看到这些专家。用户可以为一个窗体专家选中 “New Form” 检查框, 这将通知 IDE 当用户从主菜单中选择 “File|New Form” 时, 将自动调用这个窗体专家。用户同样可以选择 “Main Form” 检查框, 这将通知 IDE 使用这个窗体专家来生成新应用程序默认的主窗体。用户还可以为一个工程专家选择 “New Project” 检查框, 当用户选择 “File|New Application” 时, IDE 将调用选择的工程专家。
- 第四种类型的专家用于不能放到其它分类时的情况。一个普通的专家自身不能做任何事, 取而代

之的是，你必须自己定义专家怎样被调用。

Tools API 并不对专家作任何强制性的约束，比如并不是必须要有个工程专家才能创建工程。你可以很容易地写一个工程专家来创建一个窗体以及写一个窗体专家来创建工程（如果你确实想要这样做的话）。

下面的主题详细说明了怎样实现和安装专家：

- [实现专家接口](#)
- [安装专家包](#)

## 4、实现专家接口

每一个专家类至少必须实现 *IOTAWizard* 接口，同样，也要求实现它的父接口：*IOTANotifier* 和 *IInterface*。窗体和工程专家必须实现他们所有父接口，即：*IOTARepositoryWizard*、*IOTAWizard*、*IOTANotifier* 和 *IInterface*。

你对 *IInterface* 的实现必须遵循 Delphi 接口的一般规则，这同样也是 COM 接口的规则。即，*QueryInterface* 执行类型匹配，*\_AddRef* 和 *\_Release* 管理引用计数。你可能会想使用一个公共的基类来简化专家和通知器类的编写。出于这个考虑，ToolsAPI 单元定义了一个类，*TNotifierObject*，它实现了 *IOTANotifier* 接口并使用了空方法体。

尽管专家继承自 *IOTANotifier*，而且因此必须实现它定义的所有函数，但 IDE 通常并不使用这些函数，所以你的实现可以为空（它们在 *TNotifierObject* 中实现）。因此，当你编写你的专家类时，你只需要声明并实现那些在专家接口中引入的方法就行了，默认使用 *TNotifierObject* 对 *IOTANotifier* 的实现。

## 5、安装专家包

类似于其它的设计期包，一个专家包（Wizard Package）也必须实现一个 *Register* 函数。（关于 *Register* 函数的详细说明见 [注册组件](#)。）在 *Register* 函数中，通过调用 *RegisterPackageWizard*，你可以注册任意多的专家，并传递一个专家对象作为唯一的参数，如下所示：

```
procedure Register;  
begin  
    RegisterPackageWizard(MyWizard.Create);  
    RegisterPackageWizard(MyOtherWizard.Create);  
end;
```

同样，你也可以注册属性编辑器、组件等等，作为同一个包的一部分。

请记住，设计期包是 Delphi 主程序的一部分，这意味着所有窗体的名称在整个应用程序和所有其它的设计期包中都应该都是唯一的。这是使用包方式主要的一个缺点：你并不知道其它人会怎样命名他们的窗体。

在开发中，安装包专家类似于其它的设计期包：在包管理器中点击 **Install** 按钮。IDE 将编译和连接该包并尝试装载它。如果装载包成功，IDE 会显示一个对话框通知你。

（译注：如果是 DLL 类型的专家，需要在注册表中注册，例如在 Delphi7 中注册一个名称 MyWizard 的专家，可以在注册表 HKEY\_CURRENT\_USER\Software\Borland\Delphi\7.0\Experts 中增加一个字符串项：MyWizard，值为 DLL 的完整路径文件名）

## 6、获得 Tools API 服务

为了做一些有用的工作，专家需要访问 IDE：它的编辑器、窗体、菜单等等，这些是服务接口的任务。Tools API 包括很多的服务，例如用 Action 服务执行文件 Action 操作，用编辑器服务访问源代码编辑器，用调试器服务访问调试器，等等。下面的表格总结了所有的服务接口：

Tools API 服务接口	
接口	描述
INTAServices	提供对本地 IDE 对象的访问：主菜单、Action 列表、图像列表和工具栏。
IOTAActionServices	实现基本的文件操作：打开、关闭、保存和重装载文件。
IOTACodeCompletionServices	提供对代码完成的访问，允许专家安装自定义的代码完成管理器。
IOTADebuggerServices	提供对调试器的访问。
IOTAEditorServices	提供对源代码编辑器及其内部缓冲区的访问。
IOTAKeyBindingServices	允许专家注册自定义的键盘绑定。
IOTAKeyboardServices	提供对键盘宏和绑定的访问。
IOTAKeyboardDiagnostics	切换按键调试。
IOTAMessageServices	提供对消息视图（Message View）的访问。
IOTAModuleServices	提供对打开的文件的访问。
IOTAPackageServices	查询已安装的包及他们的组件的名称。
IOTAServices	其它的服务。
IOTAToDoServices	提供对 To-Do 列表的访问，允许专家安装自己的 To-Do 列表管理器。
IOTAToolsFilter	注册工具过滤通知器（Tools Filter Notifiers）。
IOTAWizardServices	注册及删除专家。

要使用服务接口，使用在 SysUtils 单元中定义的全局的 *Supports* 函数将 *BorlandIDEServices* 变量转换为目标服务接口。例如：

```
procedure set_keystroke_debugging(debugging: Boolean);
var
  diag: IOTAKeyboardDiagnostics;
begin
  if Supports(BorlandIDEServices, IOTAKeyboardDiagnostics, diag) then
    diag.KeyTracing := debugging;
end;
```

如果你的专家频繁地使用一个特定的服务，你可以将这个服务的指针作为一个数据成员保存在你的专家类中。

下面的主题讨论了使用 Tools API 服务接口来工作时一些特定的事项：

- [使用本地 IDE 对象](#)
- [调试专家](#)
- [接口版本号](#)

## 7、使用本地 IDE 对象

专家可以完全地访问 IDE 的主菜单、工具栏、Action 列表和图像列表。（注：IDE 的很多弹出菜单不能直接通过 Tools API 来访问。）

对 IDE 本地对象的操作以 *INTAServices* 接口为起点，你可以使用这个接口来增加图像到图像列表，增加 Action 到 Action 列表，添加菜单项到主菜单，以及在工具栏上添加按钮。你也可以关联 Action 到菜单项和工具栏按钮。当专家释放的时候，它必须清除那些由它自己创建的对象，但是不能删除它增加到图像列表中的图像，因为删除图像可能会打乱所有在该专家之后增加的其它图像的索引号。

下面的主题阐述了如何执行这些操作：

- [增加图像到图像列表](#)
- [增加Action到Action列表](#)
- [删除工具栏按钮](#)

专家操作的是 IDE 中真实的 *TMainMenu*、*TActionList*、*TImageList* 和 *TToolBar* 对象，故你可以象在编写其它应用程序那样写代码。这同样意味着你有很大的机会让 IDE 崩溃或者禁用掉一些重要的功能，例如删除文件菜单。[调试专家](#) 讨论了当你发现类似这样的问题时，怎样调试你的专家的方法。

## 8、增加图像到图像列表

假如你打算增加一个菜单项来调用你的专家，你同样也会允许用户增加一个工具栏按钮来调用这个专家。第一个步骤是增加一个图像到 IDE 的图像列表，然后你增加的图像的索引号就可以在 Action 中使用，随后也就可在菜单项和工具栏中使用。使用图像编辑器（Image Editor）创建一个包含 16X16 位图资源的资源文件，然后在你的专家构造器中加上下面的代码：

```
constructor MyWizard.Create;  
  
var  
    Services: INTAServices;  
    Bmp: TBitmap;  
    ImageIndex: Integer;  
  
begin  
    inherited;  
    Supports(BorlandIDEServices, INTAServices, Services);  
    { Add an image to the image list. }  
    Bmp := TBitmap.Create;  
    Bmp.LoadFromResourceName(HInstance, 'Bitmap1');  
    ImageIndex := Services.AddMasked(Bmp, Bmp.TransparentColor,  
                                     'Tempest Software.intro wizard image');  
  
    Bmp.Free;  
  
end;
```

请确认使用你在资源文件中指定的名称或 ID 来装载位图。你必须选择一个颜色来作为图像的背景颜色。如果你不想要背景颜色，可以选择一个在位图中不存在的颜色。

## 9、增加 Action 到 Action 列表

在 [增加图像到图像列表](#) 中获得的图像索引号可以用来创建 Action，如下所示。专家使用 *OnExecute* 和 *OnUpdate* 事件。在专家中常用的方法是在 *OnUpdate* 事件中启用或禁用 Action，请确认 *OnUpdate* 事件能很快的返回，否则用户将会发现在装载你的专家后，IDE 变得慢如蜗牛。Action 的 *OnExecute* 事件类似于专家的 *Execute* 方法。如果你使用菜单项来调用一个窗体或工程专家，你甚至可能希望让 *OnExecute* 直接调用 *Execute* 方法。

```
NewAction := TAction.Create(nil);
NewAction.ActionList := Services.ActionList;
NewAction.Caption := GetMenuText();
NewAction.Hint := 'Display a silly dialog box';
NewAction.ImageIndex := ImageIndex;
NewAction.OnUpdate := action_update;
NewAction.OnExecute := action_execute;
```

菜单项可以设置它的 *Action* 属性为新创建的 Action。创建一个新的菜单项时比较复杂的部分是要知道它应该被插入到哪儿。下面的例子查找 View 菜单，并且创建一个新的菜单项作为第一项插入到 View 菜单下。（通常，依赖于绝对位置不是个好主意：你无法知道还有哪些其它的专家会将自己插入到菜单中。另外，将来版本的 Delphi 也可能会调整菜单项的顺序。一个更好的方法是使用特定的名称查找指定的菜单项。下面的简单例子仅用来说明概念。）

```
for I := 0 to Services.MainMenu.Items.Count - 1 do
begin
  with Services.MainMenu.Items[I] do
  begin
    if CompareText(Name, 'ViewsMenu') = 0 then
    begin
     NewItem := TMenuItem.Create(nil);
     NewItem.Action := NewAction;
      Insert(0,NewItem);
    end;
  end;
end;
```

增加 Action 到 IDE 的 Action 列表中后，用户就能在定制工具栏时看到这个 Action 了。用户可以选择 Action 并把它作为按钮增加到工具栏上。这在你的专家被卸载时会带来一些问题：所有这些指向已经不存在的 Action 的工具栏按钮和它们的 OnClick 事件句柄都将被悬空。为了防止访问违规错误，你的专家必须查找所有引用了它的 Action 的工具栏按钮，并且删除它们。

## 10、删除工具栏按钮

此处没有直接的函数来从工具栏中删除按钮，你必须自己发送 CM\_CONTROLCHANGE 消息。消息的第一个参数是要修改的控件，第二个参数为零表示从工具栏中删除（非零是添加）。删除工具栏按钮后，专家析构器删除 Action 和菜单项，删除这些对象将自动把它们从 IDE 的 ActionList 和 MainMenu 中移除。

```
procedure remove_action (Action: TAction; ToolBar: TToolBar);
var
  I: Integer;
  Btn: TToolButton;
begin
  for I := ToolBar.ButtonCount - 1 downto 0 do
  begin
    Btn := ToolBar.Buttons[I];
    if Btn.Action = Action then
    begin
      { Remove "Btn" from "ToolBar" }
      ToolBar.Perform(CM_CONTROLCHANGE, WPARAM(Btn), 0);
      Btn.Free;
    end;
  end;
end;

destructor MyWizard.Destroy;
var
  Services: INTAServices;
  Btn: TToolButton;
begin
  Supports(BorlandIDEServices, INTAServices, Services);
  { Check all the toolbars, and remove any buttons that use this action. }
  remove_action(NewAction, Services.ToolBar[sCustomToolBar]);
  remove_action(NewAction, Services.ToolBar[sDesktopToolBar]);
  remove_action(NewAction, Services.ToolBar[sStandardToolBar]);
  remove_action(NewAction, Services.ToolBar[sDebugToolBar]);
  remove_action(NewAction, Services.ToolBar[sViewToolBar]);
  remove_action(NewAction, Services.ToolBar[sInternetToolBar]);
  NewItem.Free;
  NewAction.Free;
end;
```

## 11、 调试专家

Tools API 为你的专家与 IDE 之间的交互提供了极大的灵活性。然而，这种灵活性也带来了风险，很容易因为空指针和访问违规导致错误。

当使用本地 Tools API 编写专家时，你编写代码的可能会导致 IDE 崩溃掉，还有可能当你编写完专家安装后却发现它不能象你预期那样工作，这就要求对设计期代码进行调试。不过这个问题很容易解决，因为专家被安装在 Delphi 自身中，你只需要简单的在“Run|Parameters...”菜单项中设置专家包的宿主应用程序为 Delphi 可执行文件（delphi32.exe）即可。

当你想要或需要调试一个包时，先不要安装它。取而代之的是，从菜单栏中选择“Run|Run”，这将会重新启动一个新的 Delphi 的实例。在新的实例中，从菜单栏中选择“Components|Install Package...”

并选中已经编译好的包。回到原来的 Delphi 进程实例中，现在你将看到在专家的源代码中，那些小蓝点意味着你可以在源代码中设置断点了。（如果没有，打开你的编译设置确认设置了允许调试；确认安装了正确的包；另外还可以打开进程模块来确保你装载的 .bpl 文件确实是你想要的。）

这种方式下，你不能调试进入 VCL、CLX 或 RTL 的代码，但是你可以全面地调试你自己的专家，对于查找是哪里出错来说，已经足够了。

## 12、 接口版本号

如果你细心地查看过某些接口，例如 *IOTAMessageServices*，你会发现他们从另一个名字很接近的接口继承而来，例如 *IOTAMessageServices50*，而后者又继承自 *IOTAMessageServices40*。使用这些版本号可以帮助你的代码隔离不同 Delphi 发布版本之间的变化。

Tools API 遵循 COM 的基本规则，即接口和它的 GUID 永远不会修改。如果一次新的发布为某个接口增加了一些功能，Tools API 将声明一个新的接口并继承自旧的接口。保留旧的 GUID 附加在原来未变化的接口上，而新的接口则使用一个新的 GUID。使用旧的 GUID 的专家还可以继续工作。

Tools API 还会修改接口的名称以保持源代码的兼容性。要想知道他们是怎样工作的，很重要的一点是要区别 Tools API 中两种不同类型的接口：由 Borland 实现的接口以及由用户实现的。

如果是由 IDE 实现的接口，名称会保留给最新版本的接口。因为新功能并不会影响到原来的代码，而旧的接口则在后面加上旧的版本号。

对一个由用户实现的接口来说，接口中新的成员需要在你的代码中用新的函数来实现，因此名称被保留给旧接口使用，而新的接口在其后面加上版本号。

以消息服务来举例，Delphi 6 引入了一个新的功能：消息分组。因此，基本的消息服务接口需要一些新的成员函数。这些函数将在一个新的接口类中声明，新接口将继续使用 *IOTAMessageServices* 的名称，而旧的消息服务接口改名为 *IOTAMessageServices50*（表示版本号 5）。旧的 *IOTAMessageServices* 的 GUID 与新的 *IOTAMessageServices50* 的 GUID 相同，因为它们的成员函数是一致的。

*IOTAIDENotifier* 接口是一个由用户实现接口的例子。Delphi 5 增加了新的 overload 函数：*AfterCompile* 和 *BeforeCompile*。已经使用 *IOTAIDENotifier* 编写的代码不需要修改，但是如果使用新功能，新的代码则需要修改，改为覆盖从 *IOTAIDENotifier50* 接口中继承来的新函数。Delphi 6 中并没有为这个接口增加新的功能，所以当前版本使用的是 *IOTAIDENotifier50*。

一个建议的规则是当编写新代码时使用最后面的派生类。而如果你只是想在新的 Delphi 版本中重新编译的话，保留原来旧的代码不变就行了。

## 13、 对文件和编辑器的操作

理解 Tools API 怎样操作文件是很重要的。这里主要的接口是 *IOTAModule*，每一个模块对应着一组逻辑上关联的打开的文件。例如，一个简单的模块可能对应着一个单独的单元文件（Unit）。一个模块可以有一个或多个编辑器，每个编辑器对应着一个文件，比如源代码（.pas）或窗体（.dfm 或 .xfm）文件。编辑器接口反映了 IDE 编辑器状态，故专家也可以看到用户所见到的修改过的代码和窗体，甚至那些用户还没有保存过的修改。

下面的主题提供了关于模块和编辑器接口的内容：

- [使用模块接口](#)
- [使用编辑器接口](#)

## 14、 使用模块接口

要获得模块接口，先要从模块服务 (*IOTAModuleServices*) 开始。你可以查询模块服务来获得所有打开的模块，通过一个文件或窗体的名字来检索模块，或者打开一个文件以获得它的模块接口。

针对不同类型的文件，有不同类型的模块，例如工程、资源以及类型库。可以尝试将一个模块接口转换成特定类型的模块接口以判断该模块是否属于那种类型。例如，以下是一种获得当前工程组接口的方法：

```
{ Return the current project group, or nil if there is no project group. }  
function CurrentProjectGroup: IOTAProjectGroup;  
var  
    I: Integer;  
    Svc: IOTAModuleServices;  
    Module: IOTAModule;  
begin  
    Supports(BorlandIDEServices, IOTAModuleServices, Svc);  
    for I := 0 to Svc.ModuleCount - 1 do  
    begin  
        Module := Svc.Modules[I];  
        if Supports(Module, IOTAProjectGroup, Result) then  
            Exit;  
    end;  
    Result := nil;  
end;
```

## 15、 使用编辑器接口

每一个模块都有至少一个编辑器接口。有些模块拥有好几个编辑器，比如一个源代码 (.pas) 文件和一个窗体描述 (.dfm) 文件。所有的编辑器都实现了 *IOTAEditor* 接口，可以尝试将一个编辑器转换成另一个类型的编辑器接口以判断它是否属于那种类型。例如，要想获得一个单元的窗体编辑器接口，你可以这样做：

```
{ Return the form editor for a module, or nil if the unit has no form. }  
function GetFormEditor(Module: IOTAModule): IOTAFormEditor;  
var  
    I: Integer;  
    Editor: IOTAEditor;  
begin  
    for I := 0 to Module.ModuleFileCount - 1 do  
    begin  
        Editor := Module.ModuleFileEditors[I];  
        if Supports(Editor, IOTAFormEditor, Result) then  
            Exit;  
    end;  
    Result := nil;
```

```
end;
```

编辑器接口给予你对编辑器内部状态的访问。你可以查看用户正在编辑的源代码或窗体上的组件，修改源代码、组件、属性、修改当前源代码及窗体编辑器中选择的内容，以及完成差不多所有最终用户能执行的操作。

使用窗体编辑器接口，专家能访问到窗体上所有的组件。每一个组件（包括根窗体或数据模块）都有一个关联的 *IOTAComponent* 接口。专家可以查看或修改组件的大多数属性。如果你需要完全控制组件，你可以将 *IOTAComponent* 接口转换为 *INTAComponent*。本地组件接口允许你的专家直接访问 *TComponent* 指针。这一点在当你需要读取或修改一个 Class 类型的属性时是非常有用的，比如 *TFont* 的属性，只有在通过 NTA 风格的接口时才能访问。

## 16、 创建窗体和工程

Delphi 已经提供了非常多的窗体和工程专家，当然你也可以编写你自己的。对象仓库（Object Repository）允许你创建可以在工程中使用的静态模板，而专家提供了更强大的功能，因为它是动态的。专家可以提示用户并根据用户不同的选择以创建不同类型的文件。

窗体或工程专家可以创建一个或多个新文件。通常，最好是创建未命名、未保存的模块，以代替真实的文件，这样当用户保存它们时，IDE 会提示用户选择文件名。专家使用创建器（Creator）对象来创建这些模块。

创建器类实现一个继承自 *IOTACreator* 的创建器接口。专家传递一个创建器对象给模块服务的 *CreateModule* 方法，IDE 会根据创建模块时需要的参数来回调创建器对象。

例如，一个创建新窗体的窗体专家通常会实现 *GetExisting* 并返回 *False*，以及实现 *GetUnnamed* 并返回 *True*。此时将创建一个没有名字的模块（这样用户在保存文件时必须选择一个文件名）并且没有已存在的文件与之对应（这样用户甚至在没有做任何改动时也需要保存文件）。创建器的其它方法将告诉 IDE 要创建的文件是什么类型的（例如工程、单元或窗体）、提供文件的内容或返回窗体的名称、父类的名称以及其它一些重要信息。附加的回调方法使得专家可以增加新的模块到新创建的工程，或增加组件到新创建的窗体中。

要创建新的文件，在窗体或工程专家中通常需要实现为新文件提供内容。要做到这点，可以写一个实现了 *IOTAFile* 接口的新类。如果你的专家只需要使用默认文件内容的话，你也可以在需要返回 *IOTAFile* 接口的函数中返回 *nil*。

例如，假定你的组织要求在每个单元的最上面有一个标准的注释块，你可以在对象仓库中使用静态模板，但是你需要手动修改该注释块来反映其作者和创建日期。代替的方法是，你可以使用一个创建器来在文件创建时动态地填写这些内容。

编写专家的第一个步骤是创建一个新的单元和窗体。大多数的创建器函数返回零、空字符串或其它的缺省值，以通知 Tools API 使用它自己默认的实现来创建新的单元或窗体。覆盖 *GetCreatorType* 方法以通知 Tools API 将创建哪一种类型的模块：单元或是窗体。要创建单元，返回 *sUnit*，要创建窗体，返回 *sForm*。为了简化代码，这里使用一个简单的类将构造器参数作为创建器的类型，通过保存创建器类型在一个数据成员中，这样 *GetCreatorType* 返回这个值就可以了。然后，实现 *NewImplSource* 以及 *NewIntfSource* 来返回需要生成的文件的内容。

```
TCreator = class(TInterfacedObject, IOTAModuleCreator)
public
    constructor Create(const CreatorType: string);
    { IOTAModuleCreator }
    function GetAncestorName: string;
```

```
function GetImplFileName: string;
function GetIntfFileName: string;
function GetFormName: string;
function GetMainForm: Boolean;
function GetShowForm: Boolean;
function GetShowSource: Boolean;
function NewFormFile(const FormIdent, AncestorIdent: string): IOTAFile;
function NewImplSource(const ModuleIdent, FormIdent, AncestorIdent: string): IOTAFile;
function NewIntfSource(const ModuleIdent, FormIdent, AncestorIdent: string): IOTAFile;
procedure FormCreated(const FormEditor: IOTAFormEditor);
{ IOTACreator }
function GetCreatorType: string;
function GetExisting: Boolean;
function GetFileSystem: string;
function GetOwner: IOTAModule;
function GetUnnamed: Boolean;
private
  FCreatorType: string;
end;
```

*TCreator* 的大部分方法返回 0、**nil** 或空字符串。**Boolean** 类型的方法返回 *True*, 除了 *GetExisting*, 它要返回 *False*。最有趣的方法是 *GetOwner*, 它返回一个指向当前工程模块的接口指针, 如果没有工程则返回 **nil**。此处没有简单的方法来获得当前工程或当前工程组。代替的是, *GetOwner* 必须遍历所有打开的模块, 如果找到一个工程组, 它将是唯一被打开的工程组, 此时 *GetOwner* 就可以返回它的当前工程了。否则, 该方法返回它找到的第一个工程模块, 如果没有打开的工程则返回 **nil**。

```
function TCreator.GetOwner: IOTAModule;
var
  I: Integer;
  Svc: IOTAModuleServices;
  Module: IOTAModule;
  Project: IOTAProject;
  Group: IOTAProjectGroup;
begin
  { Return the current project. }
  Supports(BorlandIDEServices, IOTAModuleServices, Svc);
  Result := nil;
  for I := 0 to Svc.ModuleCount - 1 do
  begin
    Module := Svc.Modules[I];
    if Supports(Module, IOTAProject, Project) then
    begin
      { Remember the first project module }
      if Result = nil then
        Result := Project;
    end
  end
end;
```

```
else if Supports(Module, IOTAProjectGroup, Group) then
begin
  { Found the project group, so return its active project }
  Result := Group.ActiveProject;
  Exit;
end;
end;
end;
```

创建器在 *NewFormSource* 中返回 **nil**，以产生一个默认的窗体文件。比较有趣的方法是 *NewImplSource* 和 *NewIntfSource*，它们会创建一个 *IOTAFile* 接口的实现来返回文件的内容。

*TFile* 类实现了 *IOTAFile* 接口，它返回 -1 作为文件的日期（这将意味着文件不存在），并以字符串返回文件的内容。为了简化 *TFile* 类，这里由创建器来生成字符串，*TFile* 简单地把它传进去。

```
TFile = class(TInterfacedObject, IOTAFile)
public
  constructor Create(const Source: string);
  function GetSource: string;
  function GetAge: TDateTime;
private
  FSource: string;
end;
constructor TFile.Create(const Source: string);
begin
  FSource := Source;
end;
function TFile.GetSource: string;
begin
  Result := FSource;
end;
function TFile.GetAge: TDateTime;
begin
  Result := TDateTime(-1);
end;
```

为了让文件内容更容易修改，你可以把它存储为资源，不过出于简化的目的，这个例子在专家中用硬编码的方法来实现。下面的例子生成了源代码，并假定有一个窗体。你也可以很容易地增加对普通单元的判断：检查 *FormIdent*，如果它为空，创建一个普通的单元，否则创建一个窗体单元。这个例子中生成的代码框架跟 IDE 默认的基本一致（当然在最顶上增加了注释块），但是你可以按照你的意愿任意扩充它。

```
function TCreator.NewImplSource(
  const ModuleIdent, FormIdent, AncestorIdent: string): IOTAFile;
var
  FormSource: string;
begin
  FormSource :=
    '{ ----- ' + #13#10 +
    '%s - description'+ #13#10 +
```

```

'Copyright &#169; %y Your company, inc.'+ #13#10 +
'Created on %d'+ #13#10 +
'By %u'+ #13#10 +
' ----- }' + #13#10 +
#13#10;
return TFile.Create(Format(FormSource, ModuleIdent, FormIdent,
AncestorIdent));
}
    
```

最后一步是创建两个窗体专家：一个使用 `sUnit` 创建器类型，另一个使用 `sForm`。考虑更好的用户化需要，你还可以使用 `INTAServices` 来增加菜单项到 `File|New` 菜单中来调用这些专家。菜单项的 `OnClick` 事件句柄直接调用专家的 `Execute` 函数。

一些专家需要允许或禁用菜单项，这依赖于 IDE 中发生的事件。例如，一个用于源代码控制系统集成的专家，需要在 IDE 中没有打开文件时禁用掉它的 `Check In` 菜单项，此时你可以通过在你的专家中 [使用通知器](#) 来实现。

## 17、 IDE 的专家事件通知

编写一个表现良好的专家的重要标志是专家是否具有对 IDE 事件的响应能力。特别是，那些保留了模块接口的专家必须知道用户在什么时候关闭了模块，以释放模块接口。要做到这点，专家需要一个通知器，这意味着你必须编写一个通知器类。

所有的通知器类实现一个或多个通知器接口。通知器接口定义回调方法，专家使用 `Tools API` 注册通知器对象，当某些重要事件发生时，IDE 回调通知器。

每一个通知器接口都派生自 `IOTANotifier`，尽管对一个特定通知器而言，并非 `IOTANotifier` 中所有的方法都会用到。下表列出了所有的通知器接口，并为每个接口给出了一个概要的描述。

通知器接口	
接口	描述
<code>IOTANotifier</code>	所有通知器的抽象基类
<code>IOTABreakpointNotifier</code>	在调试器中触发或切换一个断点
<code>IOTADebuggerNotifier</code>	在调试器中运行一个程序，或增加/删除断点
<code>IOTAEditLineNotifier</code>	跟踪源代码编辑器中代码行的变更
<code>IOTAEditorNotifier</code>	修改或保存源代码文件，或在编辑器中选择文件
<code>IOTAFormNotifier</code>	保存窗体，或修改窗体或窗体上的组件（或数据模块）
<code>IOTAIDENotifier</code>	装载工程，安装包，以及其它的全局 IDE 事件
<code>IOTAMessageNotifier</code>	在消息视图中增加或删除一个标签（消息组）
<code>IOTAModuleNotifier</code>	切换、保存或重命名模块
<code>IOTAProcessModNotifier</code>	在调试器中装载一个进程
<code>IOTAProcessNotifier</code>	在调试器中创建或销毁线程和进程
<code>IOTAThreadNotifier</code>	在调试器中切换线程的状态
<code>IOTAToolsFilterNotifier</code>	调用工具过滤器

要了解怎样使用通知器，先查看一下 [创建窗体和工程](#) 中的例子。该例子示范了模块创建器，通过创建一个专家来增加注释块到每个源代码文件中。注释块中包含了单元的初始名称，但是用户几乎总是会使用不同的文件名来保存文件。在这个例子中，如果专家能自动更新注释以匹配真实的文件名的话，会让用户感觉更为友好。

要做到这一点，你需要一个模块通知器。专家保存 *CreateModule* 返回的模块接口，并使用它来注册模块通知器。当用户修改或保存文件时，模块通知器将接收到通知，不过这些事件在这个专家中意义不大，所以并没有被实现，*AfterSave* 及相关的函数只是空方法体。重要的函数是 *ModuleRenamed*，当用户使用一个新名称来保存文件时 IDE 将会调用它。模块通知器类的声明如下：

```
TModuleIdentifier = class(TNotifierObject, IOTAModuleNotifier)
public
    constructor Create(const Module: IOTAModule);
    destructor Destroy; override;
    function CheckOverwrite: Boolean;
    procedure ModuleRenamed(const NewName: string);
    procedure Destroyed;
private
    FModule: IOTAModule;
    FName: string;
    FIndex: Integer;
end;
```

编写通知器的一个方法是在它的构造器中自动注册它自身，并在析构器中反注册通知器。在模块通知器的例子中，当用户关闭文件时，IDE 会调用 *Destroyed* 方法。此时，通知器必须反注册它自身并释放它保存的模块接口。IDE 释放它保存的通知器接口，将导致它的引用计数变为零并释放掉通知器对象。因此，你编写析构器时必须考虑到：通知器有可能已经被反注册过了。

```
constructor TModuleNotifier.Create(const Module: IOTAModule);
begin
    FIndex := -1;
    FModule := Module;
    { Register this notifier. }
    FIndex := Module.AddNotifier(self);
    { Remember the module's old name. }
    FName := ChangeFileExt(ExtractFileName(Module.FileName), '');
end;

destructor TModuleNotifier.Destroy;
begin
    { Unregister the notifier if that hasn't happened already. }
    if FIndex >= 0 then
        FModule.RemoveNotifier(FIndex);
end;

procedure TModuleNotifier.Destroyed;
begin
    { The module interface is being destroyed, so clean up the notifier. }
    if FIndex >= 0 then
        begin
            { Unregister the notifier. }
            FModule.RemoveNotifier(FIndex);
            FIndex := -1;
        end;
end;
```

```
FModule := nil;  
end;
```

当用户重命名文件时，IDE 会回调通知器的 *ModuleRenamed* 函数。这个函数传入新的文件名作为参数，专家可以使用它来更新注释块。要编辑源代码缓冲区，专家需要使用编辑位置接口。专家先定位到正确的位置，双重检查确定找到的是否正确的文本，并用新文件名替换掉该文本。

```
procedure TModuleNotifier.ModuleRenamed(const NewName: string);  
var  
    ModuleName: string;  
    I: Integer;  
    Editor: IOTAEditor;  
    Buffer: IOTAEditBuffer;  
    Pos: IOTAEditPosition;  
    Check: string;  
begin  
    { Get the module name from the new file name. }  
    ModuleName := ChangeFileExt(ExtractFileName(NewName), '');  
    for I := 0 to FModule.GetModuleFileCount - 1 do  
        begin  
            { Update every source editor buffer. }  
            Editor := FModule.GetModuleFileEditor(I);  
            if Supports(Editor, IOTAEditBuffer, Buffer) then  
                begin  
                    Pos := Buffer.GetEditPosition;  
                    { The module name is on line 2 of the comment. }  
                    Skip leading white space and copy the old module name,  
                    to double check we have the right spot. }  
                    Pos.Move(2, 1);  
                    Pos.MoveCursor(mmSkipWhite or mmSkipRight);  
                    Check := Pos.RipText('', rfIncludeNumericChars or rfIncludeAlphaChars);  
                    if Check = FName then  
                        begin  
                            Pos.Delete(Length(Check)); // Delete the old name.  
                            Pos.InsertText(ModuleName); // Insert the new name.  
                            FName := ModuleName; // Remember the new name.  
                        end;  
                end;  
        end;  
    end;  
end;
```

如果用户在模块名之前插入了附加的注释会怎样呢？如果是这样的话，在模块名确定后，你需要使用一个编辑行通知器来保持对行号的跟踪。要做到这点，可以使用 *IOTAEditLineNotifier* 和 *IOTAEditLineTracker* 接口。

在编写通知器时你需要非常谨慎，你必须确保没有通知器的生存期能超过它所在的专家。例如，如果用户原来使用专家创建了一个新的单元，然后卸载了专家，此时还会有一个通知器附加在该单元上面。结果将是不可预料的，不过大多数情况下，IDE 将会崩溃。因此，专家需要跟踪它创建的所有通知器，并且

在专家释放前必须反注册每一个通知器。另一方面，如果用户首先关闭了文件，模块通知器接收到了一个 *Destroyed* 通知，这意味着通知器必须反注册它自己并且释放所有对模块的引用，此时通知器也必须将自身从主专家的通知器列表中删除掉

以下是专家 *Execute* 函数的最终版本。它创建了一个新的模块，使用模块接口来创建模块通知器，然后保存模块通知器到一个接口列表中 (*TInterfaceList*)。

```
procedure DocWizard.Execute;
var
  Svc: IOTAModuleServices;
  Module: IOTAModule;
  Notifier: IOTAModuleNotifier;
begin
  { Return the current project. }
  Supports(BorlandIDEServices, IOTAModuleServices, Svc);
  Module := Svc.CreateModule(TCreator.Create(creator_type));
  Notifier := TModuleNotifier.Create(Module);
  list.Add(Notifier);
end
```

专家的析构器遍历接口列表并反注册列表中的每一个通知器。简单地让接口列表释放掉通知器接口是不够的，因为 IDE 还保留了同样的接口。你必须告诉 IDE 释放这些通知器接口来释放通知器对象。在这个例子中，析构器欺骗通知器让它们以为它们的模块被释放掉了。在更复杂的场合中，你会发现最好是为通知器类单元写一个反注册函数。

```
destructor DocWizard.Destroy; override;
var
  Notifier: IOTAModuleNotifier;
  I: Integer;
begin
  { Unregister all the notifiers in the list. }
  for I := list.Count - 1 downto 0 do
  begin
    Supports(list.Items[I], IOTANotifier, Notifier);
    { Pretend the associated object has been destroyed.
      That convinces the notifier to clean itself up. }
    Notifier.Destroyed;
    list.Delete(I);
  end;
  list.Free;
  FItem.Free;
end;
```

专家的其他地方处理注册专家、安装菜单以及诸如此类的其它事务。

## 三、后 记

### 1、寻幽访胜靠自己

因为开发 CnWizards 专家包,近二年来,CnPack 开发组对 IDE 扩展做了大量的探索和实践。Delphi 的 IDE 及 Tools API 接口向我们展示了一种开放式的应用程序体系设计思想,开发组从中汲取到了很多知识和经验。

本文档所翻译的只是 Tools API 的基础知识,IDE 中还有非常多有趣的东西等着用户自己去发掘。如果你对编写 IDE 扩展工具感兴趣,欢迎加入到 CnPack 开发组中或与我们一起交流!

CnPack开发组网站: <http://www.cnvcl.org>

管理员信箱: [master@cnvcl.org](mailto:master@cnvcl.org)

如果对本文档有什么建议或疑问,欢迎与我们联系!