
C++Builder
程序开发手册
embarcadero

序

本书『C++Builder 开发手册』的目的是说明如何学习使用 C++Builder 来开发 iOS/Android/Windows App。本书的内容并不是说明 iOS 专业程序设计，而是展示如何使用 C++Builder 整合发展环境来开发 FireMonkey iOS App。

本书分成 2 大部份，第一部份将说明如何使用 C++Builder 开发 iOS 设备上的 App，这部份也将说明如何使用 C++Builder 的 IDE。

本书将讨论如下的内容:

- 如何安装和设定 C++Builder for iOS 整合发展环境
- 使用 C++Builder for iOS 整合发展环境开发您的第一个 iOS App
- 使用 C++Builder for iOS 整合发展环境的功能
- 除错您的 iOS App
- 测试您的 iOS App
- 部署您的 iOS App
- 除错，测试和部署您的 Android App
- 开发一个有趣的 iOS App 吧

➢ 第 2 部份

- 如何安装和设定 C++Builderfor Android 整合发展环境
- 使用 C++Builder 整合发展环境开发您的第一个 Android App

- 除错，测试和部署您的 Android App
- 移植有趣的 iOS App 到 Android 中
- 结合 Android App 和云端功能

在阅读完本书的内容之后您就可以使用 C++Builder 来开发您的 iOS/Android App 了，之后您就可以阅读更深入的 iOS/Android 程序设计书籍并且使用 C++Builder 做为学习和开发的工具。

在本书随后的内容中将以 C++Builderfor iOS 来代表中开发 iOS App 的功能，并以 C++Builder for Android 代表中开发 Android App 的功能。

目录

C++Builder for iOS入门指引手册

本手册的目的是帮助 C++Builder for iOS 的入门使用者快速学习和了解如何使用 C++Builder for iOS 整合发展环境来开发手机 App，本手册将藉由开发一个小型手机 App 来说明 C++Builder for iOS 整合发展环境中经常会被使用的功能，让 C++Builder for iOS 的新使用者能够在阅读本手册之后能够立刻开始使用 C++Builder for iOS 整合发展环境来开发实际的手机 App。

1.安装和设定 C++Builder for iOS

在使用 C++Builder for iOS 开发 iOS App 之前，读者必须正确的安装和设定相关的环境和辅助软件 RAD PAServer，如此一来读者才能够在 C++Builder for iOS 整合发展环境中除错 iOS App 和部署 iOS App。

本书使用的开发环境是在 MacBook Pro 的机器中执行 OSX 10.8.2，XCode 4.6 以及 iOS SDK 6.1。并且使用 VMWare Fusion 5.0.2 执行 Windows 7 64 位的繁体操作系统。

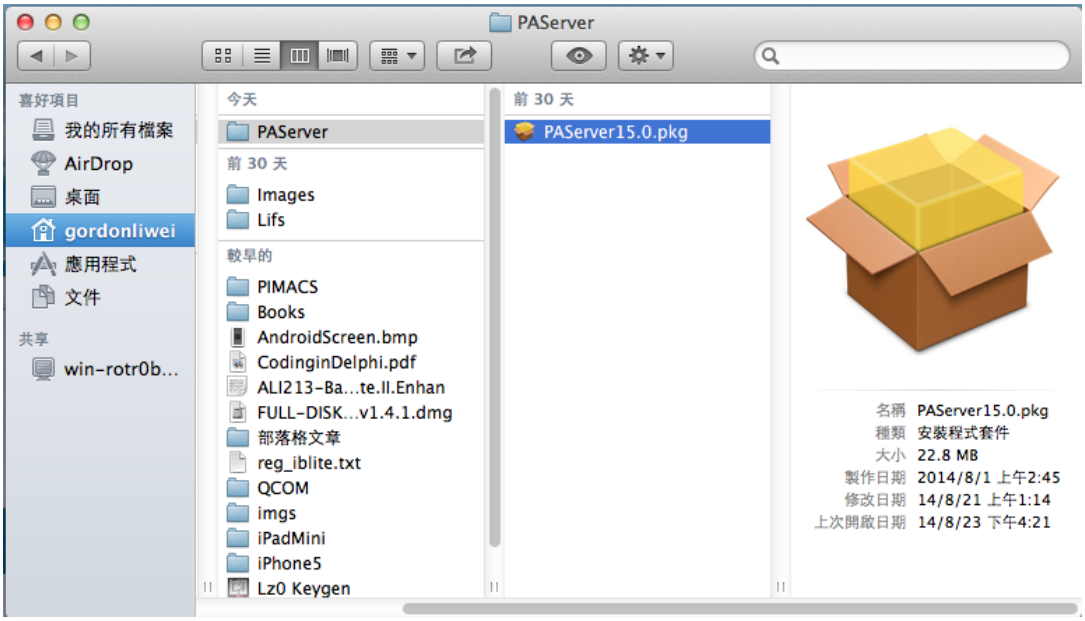
要完成 C++Builder for iOS 的安装，读者需要完成下面的步骤:

1. 安装 C++Builder for iOS 整合发展环境
2. 在 Mac OS 中安装 Platform Assistant Server(RAD PAServer)软件以便除错部署和分发 iOS App

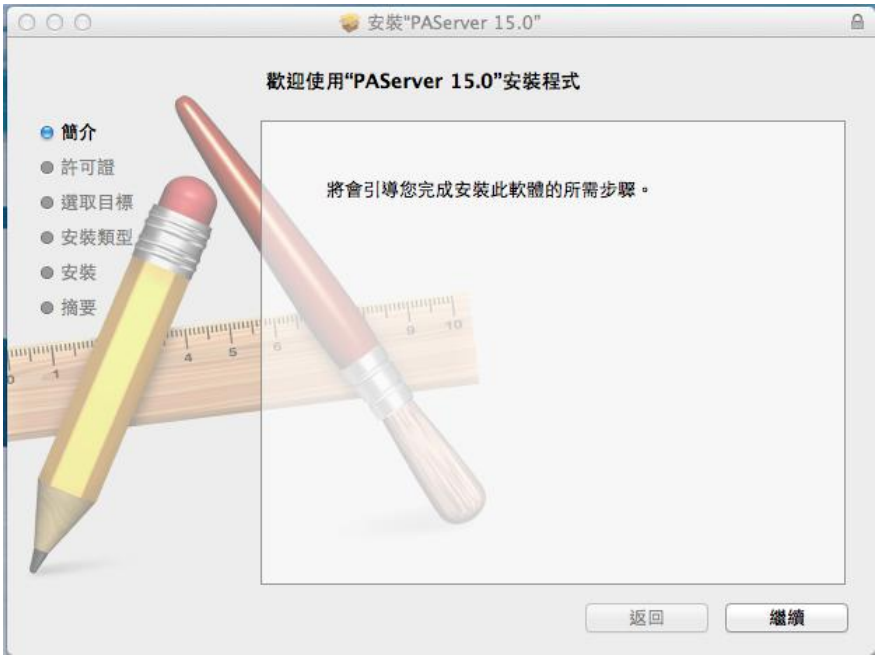
3. 在 C++Builder for iOS 整合发展环境中建立相关的组态，例如建立 iOS 仿真器组态以便除错 iOS App，建立 iOS Device 组态以便部署和分发 iOS App 到 iPhone 或是 iPad/iPad Mini 设备中。

1-1 安装 PAServer

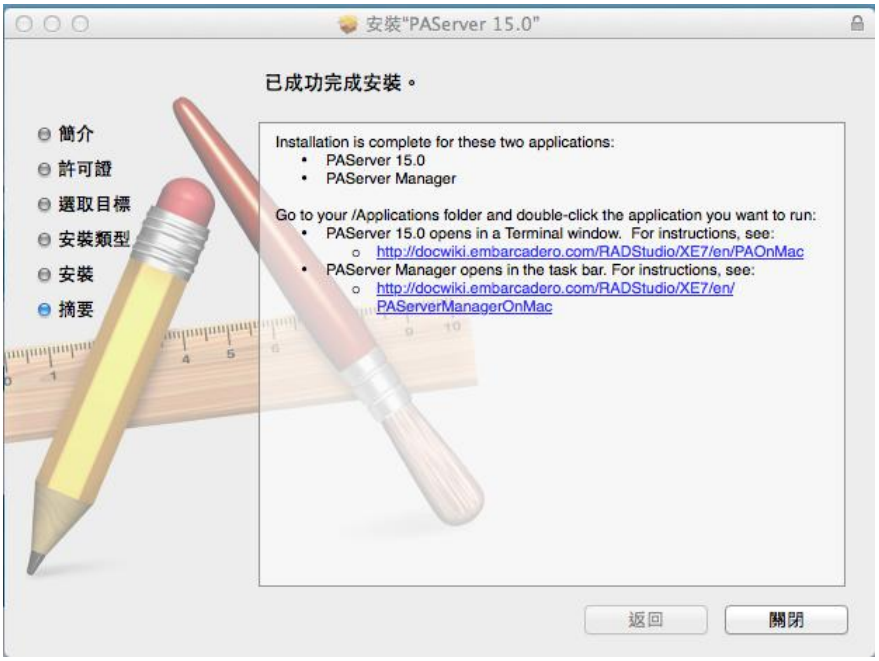
请使用 C++Builder for iOS 的安装 DVD 安装 C++Builder for iOS 整合发展环境，在安装安之后，于 C++Builder for iOS 的安装目录下会有一个『PAServer』子目录，其中的『PAServer15.0.pkg』档案即包含需要安装在 Mac OS 主操作系统的 PAServer 应用程序，请把此档案拷贝到 Mac OS 中。例如下图显示笔者把『PAServer15.0.pkg』拷贝到 Mac 的 SharediOS\PAServer 目录下:



在 Mac 中双击『PAServer15.0.pkg』便会开发安装 PAServer:



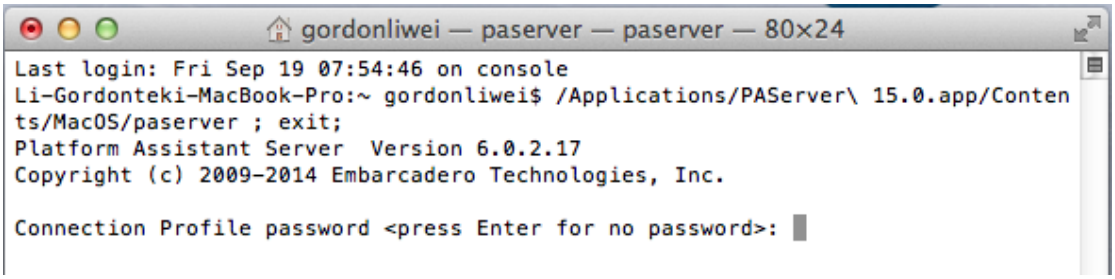
『 PAServer15.0.pkg 』 执 行 完 毕 之 后 会 把 PAServer 安 装 在 『/Applications』目录中。



安装完 PAServer 后让我们执行它以便稍后进行下一个步骤。请到 Mac 的应用程序群组中执行 RAD PAServer ， 如下所示:

或是在 Mac 中点选屏幕下方的 RAD PAServer 图像:

执行 PAServer 后，PAServer 会向读者要求输入一个链接的密码，读者可以自行给予密码或是直接按下 Return 键不使用链接密码，如下所示:



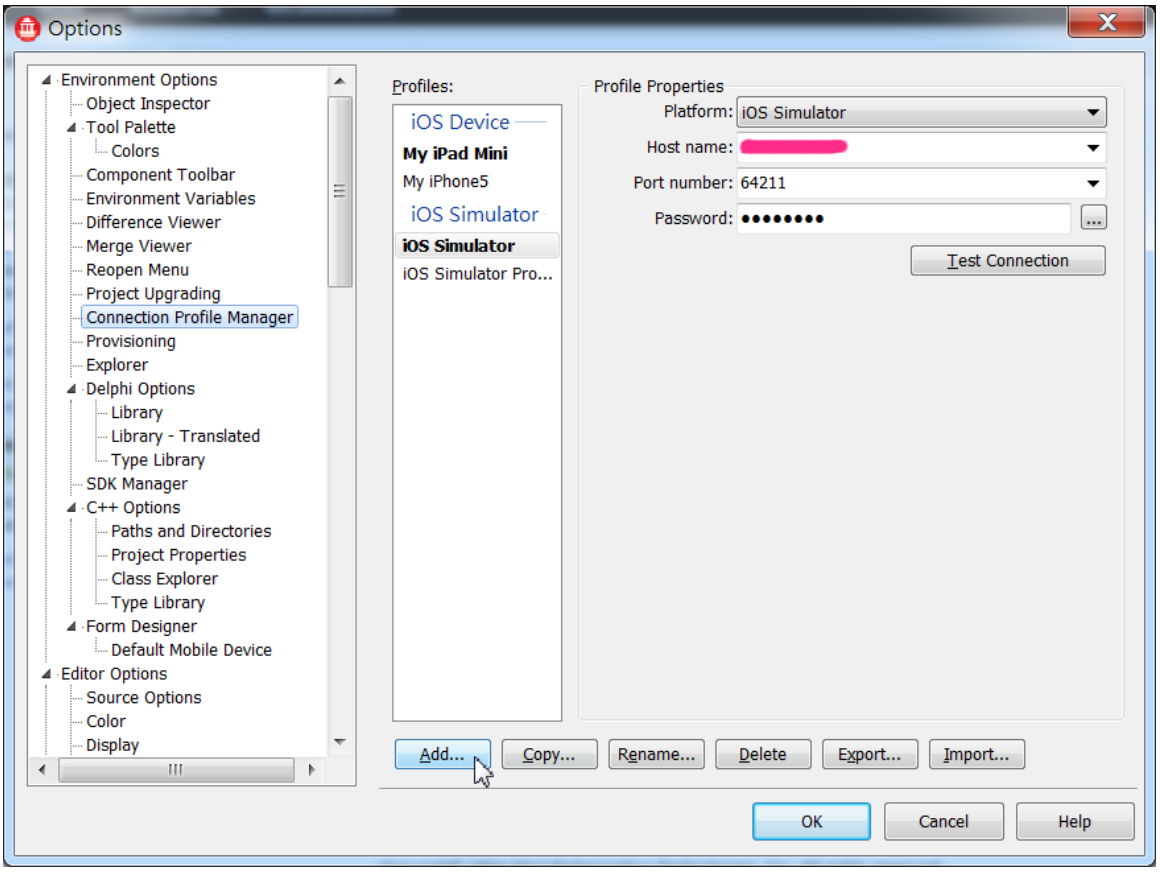
接着 PAServer 就会在执行状态，例如下图就显示了 PAServer 执行在 64211 通信埠，等待连结：

本书在撰写时是使用 C++Builder for 的 Beta 版，因此使用的 PAServer 是早期的版本，读者使用的 PAServer 应该比本书的 PAServer 版本来得更高

1-2 C++Builder for iOS 整合发展环境中建立组态

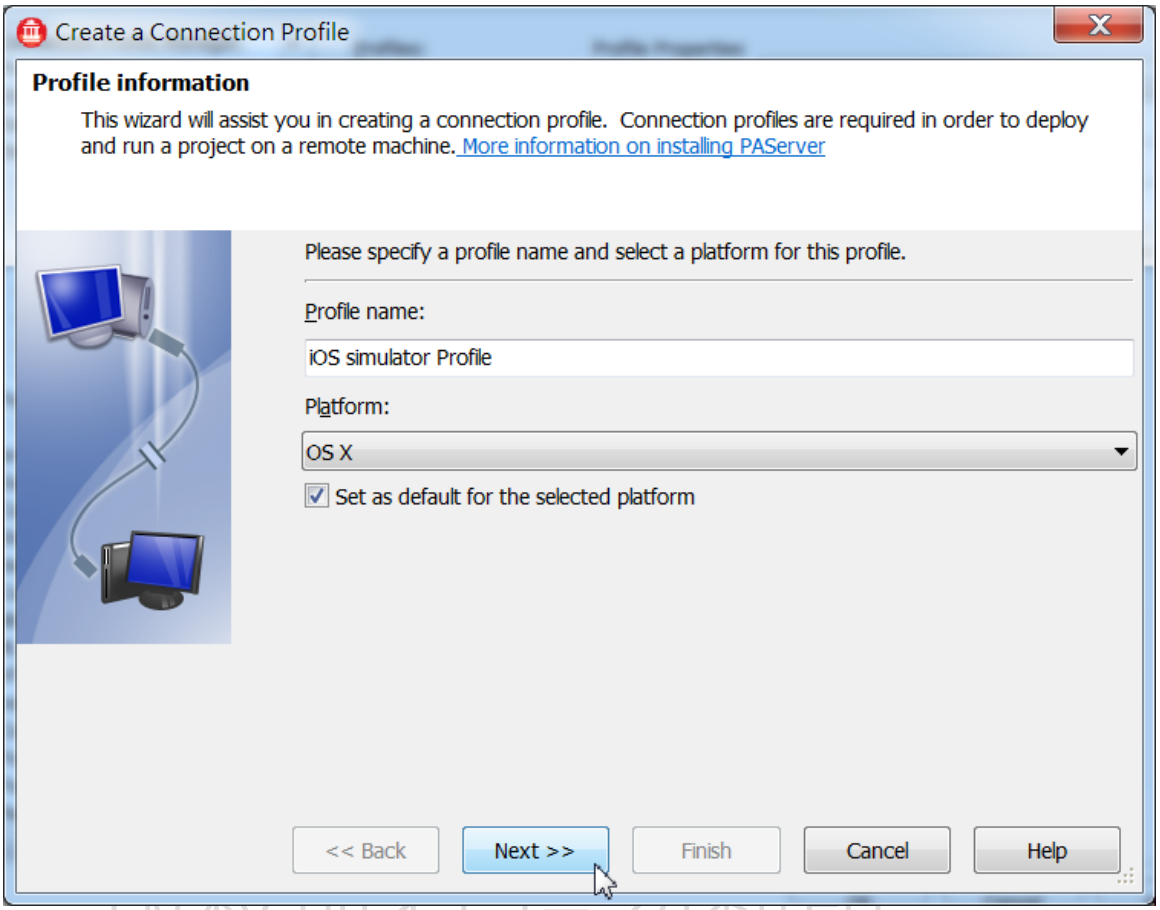
当开发人员在 C++Builder for iOS 中建立 iOS App 项目后，需要指定如何部署和分发此项目的组态。例如如果您需要除错 iOS App 项目，那么 C++Builder for iOS 整合发展环境必须把此 iOS App 分发到 Mac OS 中 XCode 的 iOS 仿真器中执行和除错，因此 C++Builder for iOS 整合发展环境必须知道您的 Mac OS 的相关信息才能够正确的把 iOS App 部署到 iOS 仿真器中，因此这就需要开发人员事先设定好组态信息并且把它指定给 iOS App 项目。

要建立组态信息，请执行 C++Builder for iOS，在整合发展环境中点选上方的 Tools|Option 菜单，在 Options 对话盒中的 Connection Profile Manager 项目中点选左下方的『Add』按钮以新建组态，如下所示:

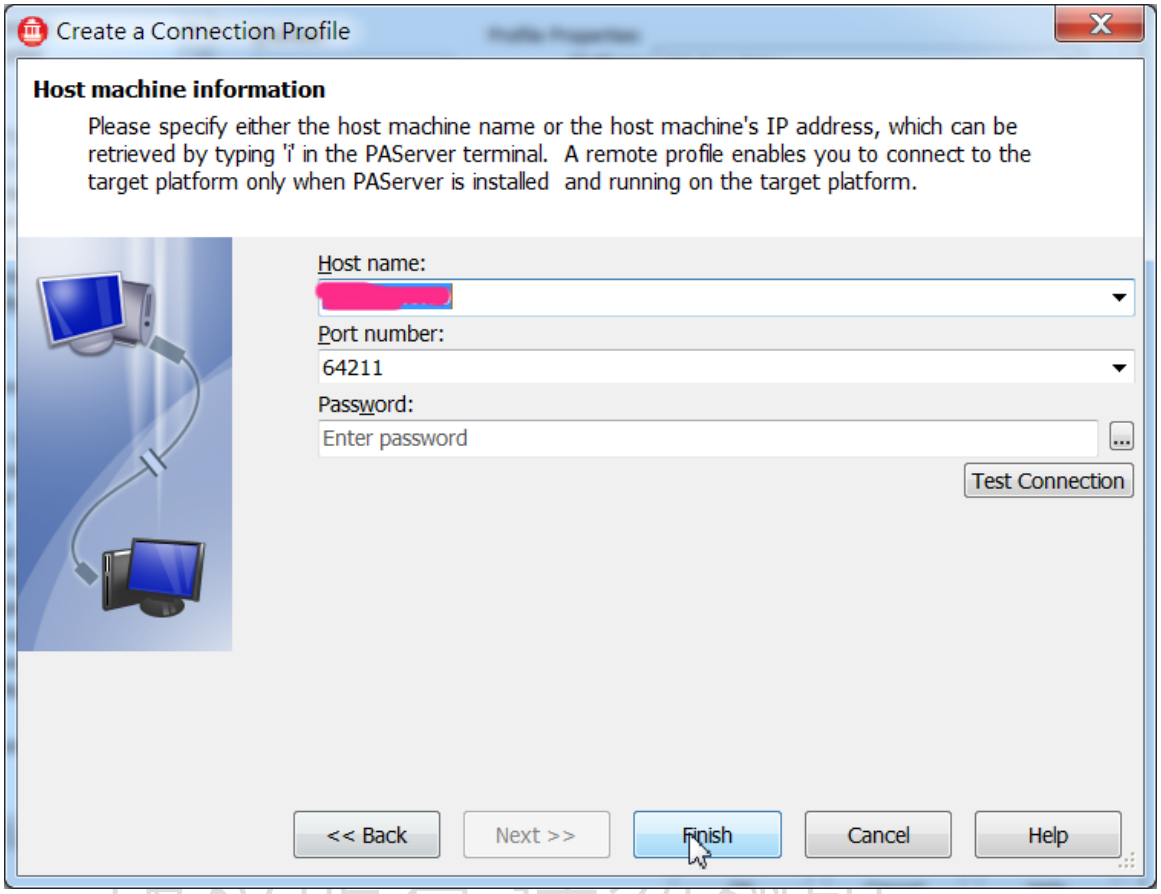


在点选『Add』按钮后 IDE 会显示 Create a Remote Profile 对话盒，您需要在此对话盒中输入您需要在 Profile name 字段中自行输入此组态的名称，接着在 Platform 字段中选择此组态需要部署的平台，例如在下图中笔者为此组态取名为『iOS Simulator Profile』并且选择部署到 iOS 仿真器中以准备开发和测试 iOS App。在 Platform 字段的下拉盒中还有其他的平台，例如 iOS Device 平台，读者可以根据需要选择欲使用的平台。目前的范例中让我们使用『OS X』平台，并且勾选下方的『Set as default for the selected platform』勾选盒以设定它为内定使用的组态。

接着点选『Next』按钮以进入下一步：



接下来您需要设定此组态的 Mac OS 的主机名或是 IP 地址，执行在 Mac OS 中 PAServer 使用的通信埠以及 PAServer 使用的链接密码，如下所示：

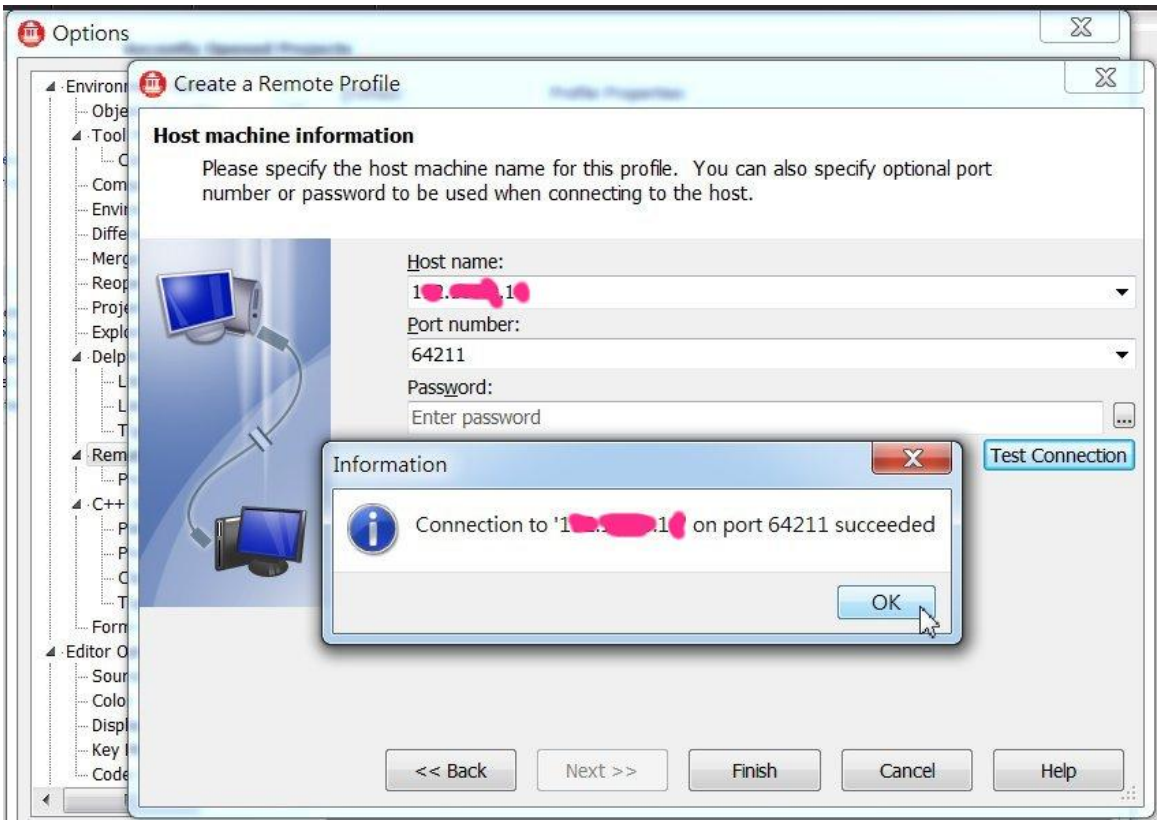


要取得您 Mac OS 的 IP 地址，您可以在您的 Mac OS 中点选屏幕左上方的苹果图像，选择其中的『系统偏好设定』，再点选其中的『网络』图像，接着 Mac OS 就会显示如下的网络对话盒，其中就会显示 IP 地址，例如笔者的 Mac IP 是『1xx.1xx.0.10』，因此在上图的 Host Name 字段中就输入了此 IP。

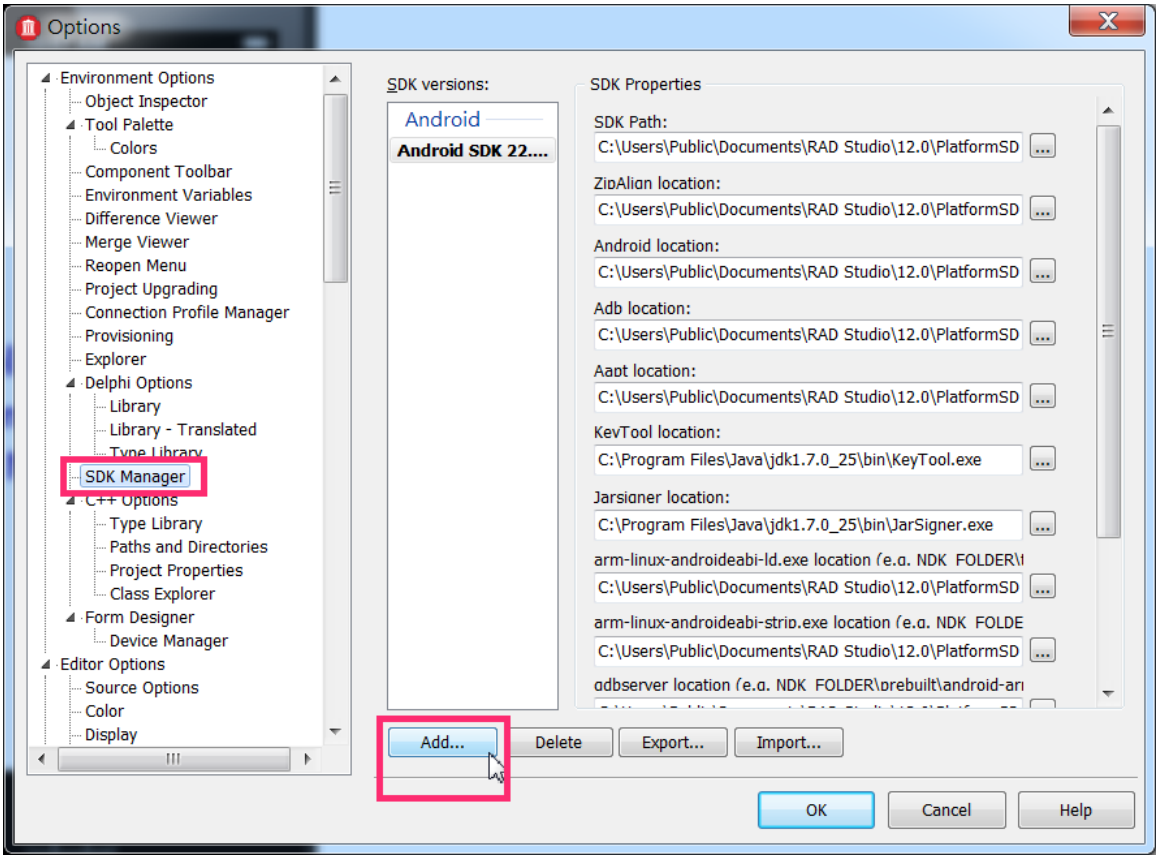
此外由于在 1-1 小节中安装 PAServer 时 PAServer 使用的通信埠是『64211』而且我们没有设定链接密码，因此在上图中 Port number 字段中输入了 64211，Password 字段则没有输入密码。

在输入了这些信息后，请确定 PAServer 已经执行在 Mac OS 中，那么您就可以点选『Create a Remote Profile』对话盒中的『Test Connection』按钮来测试 C++Builder for iOS IDE 是否能够连结到 Mac OS 中的 PAServer。如果读者设定的信息都是正确的话，那么应该会看到类似如下的结果画面，IDE 显示成功的链接您的 Mac OS 中的 PAServer，这代表稍后您就可以正确的部署您的 iOS App 到 iOS 的模拟中测试，除错和执行了。

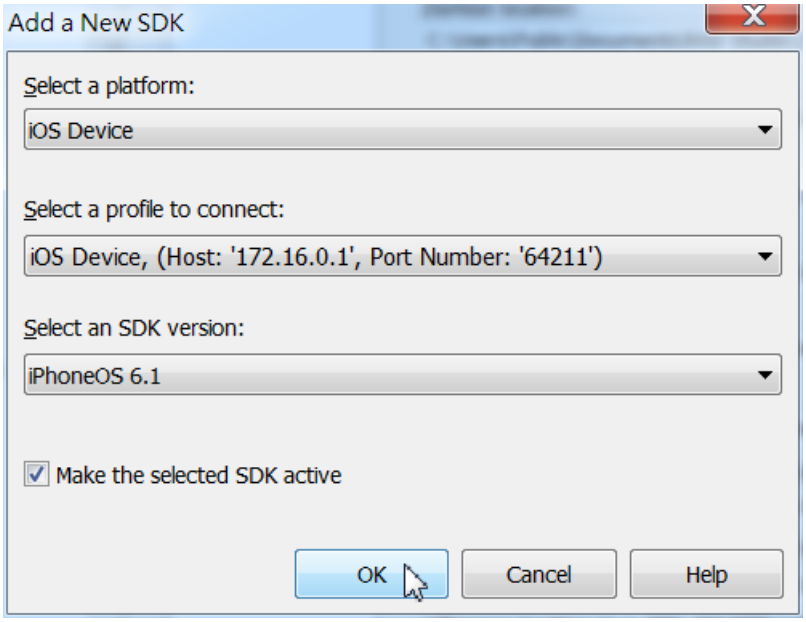
最后请点选『Finish』按钮以完成建立组态的工作。



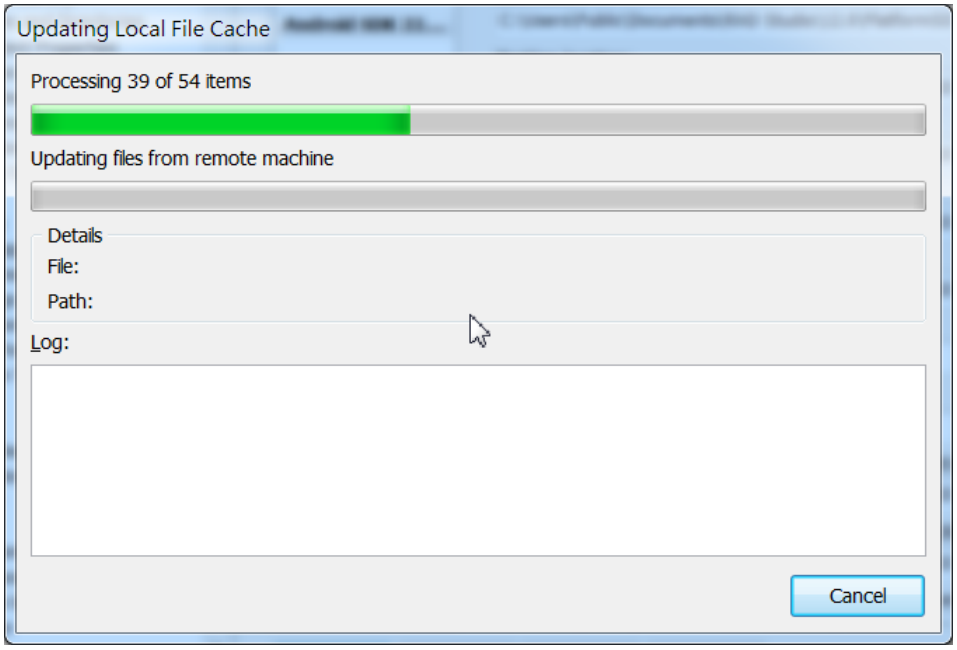
完成建立组态工作之后，最后要进行建立 iOS App 项目需要使用的 iOS SDK 的信息。请如前述一样在在 Options 对话盒中的 SDK Manager 选项中点选 Add 按钮以加入 iOS SDK 信息，如下面图形所示：



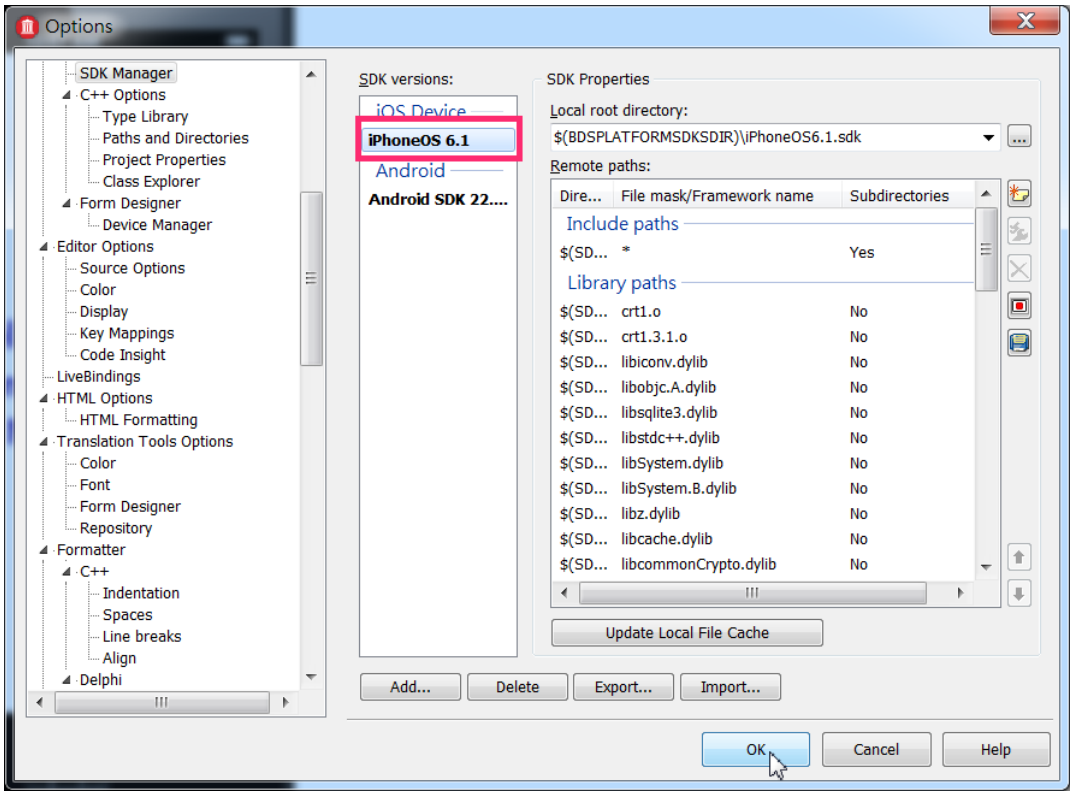
点选了 Add 按钮之后 C++Builder IDE 会显示如下的 Add a New SDK 对话盒，请在 Select a platform 下拉盒中选择 iOS Device，在 Select a profile to connect 下拉盒中选择前面步骤建立的组态名称，最后在 Select an SDK version 下拉盒中会出现你使用的 iOS SDK 版本，请从下拉盒中选择您使用的 SDK 版本，如下面图形所示：



点选上面对话盒中的 OK 按钮后 C++Builder 便会自动设定 iOS APP 项目需要使用的 iOS SDK 相关信息：



在上面的步骤完成之后您就可以在 SDK Manager 选项中看到出现了 iOS Device 项目，在其中则会出现您的 iOS SDK 的设定信息，如下面图形所示：

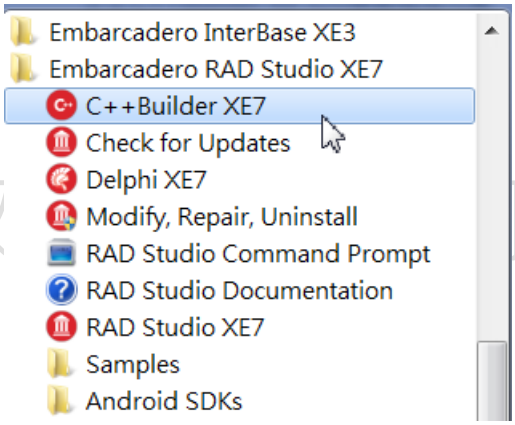


现在我们就可以准备进入 C++Builder for iOS 整合发展环境来开发 iOS App 了。

> C++Builder for iOS 只支持直接在 iOS 设备中开发 App，并不支持 iOS Simulator，因此在前面设定的步骤中是直接设定 iOS 设备的相关组态信息，本书稍后讨论的内容和范例也都是直接在 iOS 设备中测试和执行的，在未来的 C++Builder 版本才会支持 iOS Simulator。

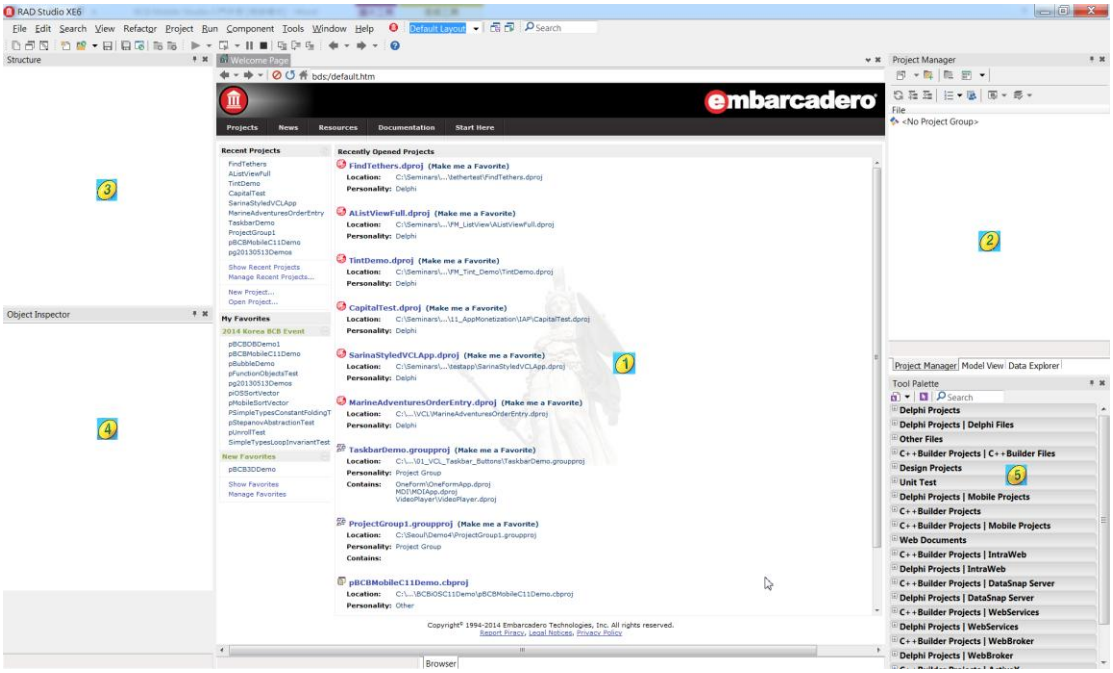
2. 进入 C++Builder for iOS 整合发展环境

当您使用 C++Builder for iOS DVD 安装完之后，您可以藉由点选如下图的 C++Builder for iOS 图像开始执行 C++Builder for iOS 整合发展环境:



点选 Embarcadero RAD Studio 或是 C++Builder 图像执行 C++Builder for iOS

一进入 C++Builder for iOS 整合发展环境，您会看到类似如下的画面，C++Builder for iOS 整合发展环境除了菜单和工具栏之外(稍后说明)，整个整合发展环境分成 5 大区域，在下图中我们以数字来标明这 5 大区域:

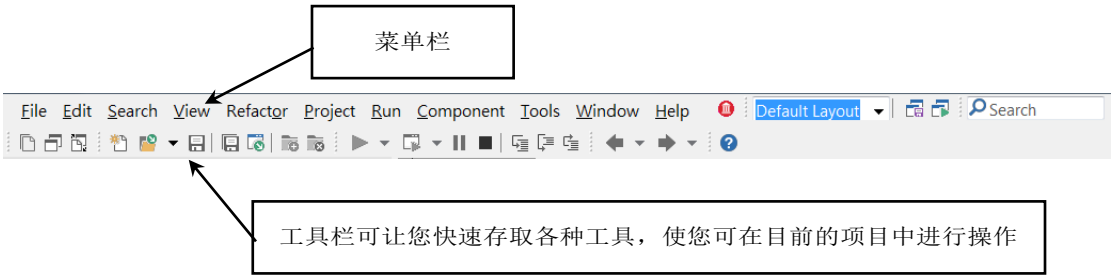


C++Builder for iOS 整合发展环境进入画面

下面的表格说明了这成 5 大区域的功能:

区域编号	说明
1	欢迎画面(Welcome Page)，其中的内容会根据开发人员的设定来显示，在内定上会显示开发人员最近开启和使用的项目。此外开发人员也可以在其中建立『最爱』，并且把项目归类在不同的『最爱』中。
2	项目管理员(Project Manager)，管理项目中所有的档案，由于在一开始没有建立或是开启任何的项目，因此它的内容暂时是空白的
3	项目树状架构(Structure)，可显示目前开启的窗体(Form)中组件的树状架构，由于在一开始没有建立或是开启任何的窗体，因此它的内容暂时是空白的
4	对象查看器(Object Inspector)，在项目设计时期可检视，设定组件的特性值，由于在一开始没有建立或是开启任何的窗体和使用任何的组件，因此它的内容暂时是空白的
5	工具盘(Tool Palette)，列出目前可使用的工具，在一开始它的内容列出了目前在整合发展环境中可建立的项目种类。如果您建立了项目之后，它的内容就会改变，稍后我们会看到如果在项目中设计表格，那么它的内容就会改变为表格可使用的组件

此外，在整合发展环境上方提供了工具栏可帮助您快速完成特定的工作:



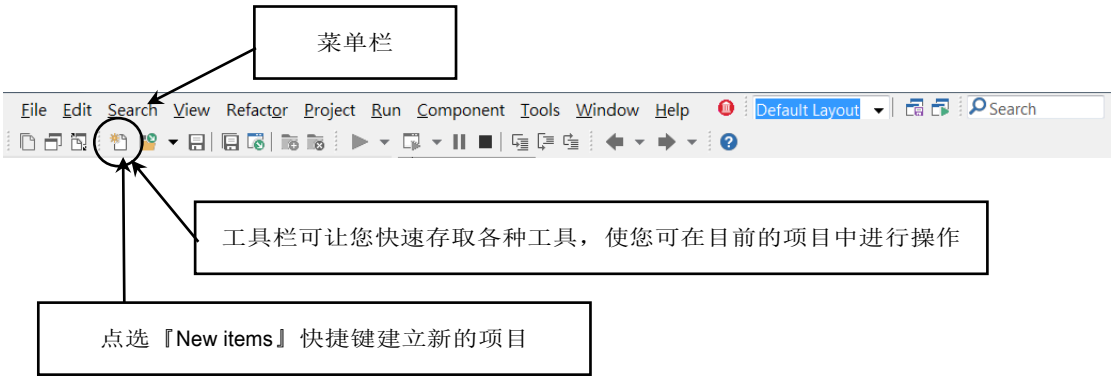
在稍后的教学内容会说明工具栏中不同按钮的功能，现在就让我们开始建立一个新的 C++Builder for iOS 项目。

2. 建立新项目

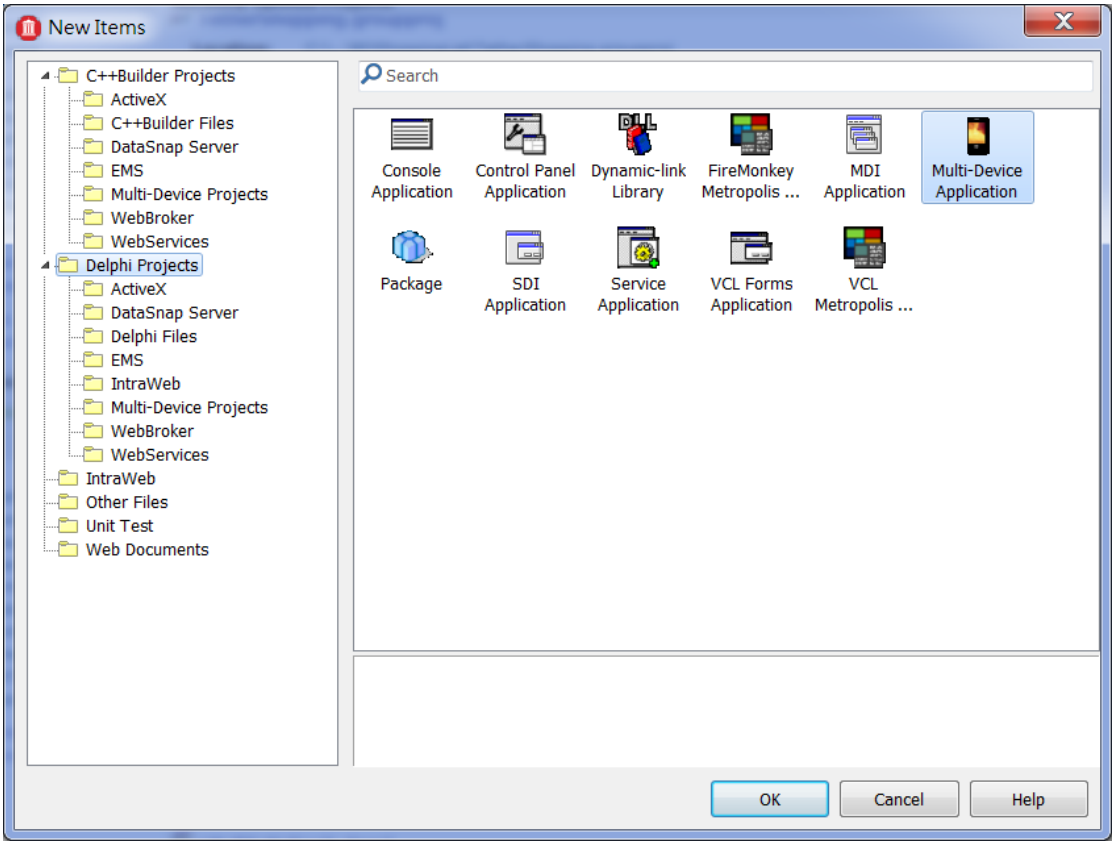
要使用 C++Builder for iOS 开发新的 iOS App，您必须先从建立 iOS App 项目开始:

2-1 如何建立 iOS App 新项目

在 C++Builder for iOS 整合发展环境中有许多方法可以建立新项目，首先您可以点选工具栏中的『New items』快捷键，它位于工具栏从左方开始的第 4 个快捷键:

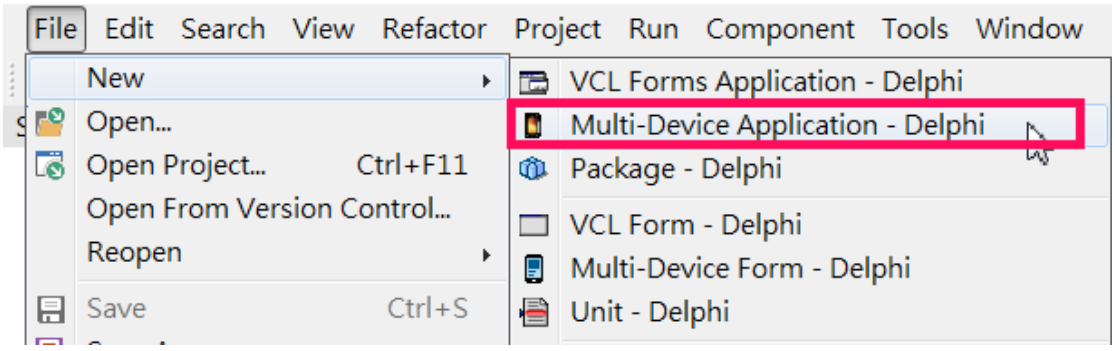


点选了工具栏中的『New items』快捷键之后，整合发展环境会显示如下的 New Items 对话盒，让您从其中选择您想建立的项目型态，请在 C++Builder Projects 项目中选择『Multi-Device Application』图像以建立 iOS App 项目，如下图所示:



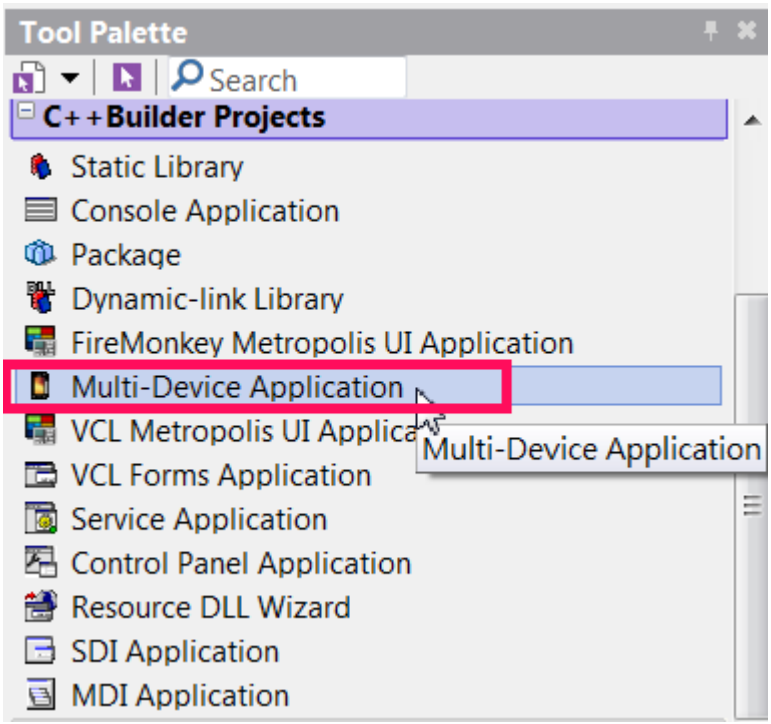
在 New Items 对话盒中选择要建立的项目型态

第二种建立项目的方法是藉由整合发展环境上方的菜单，您可以藉由点选菜单中的 File | New 选项来选择要建立的项目型态，如下所示：



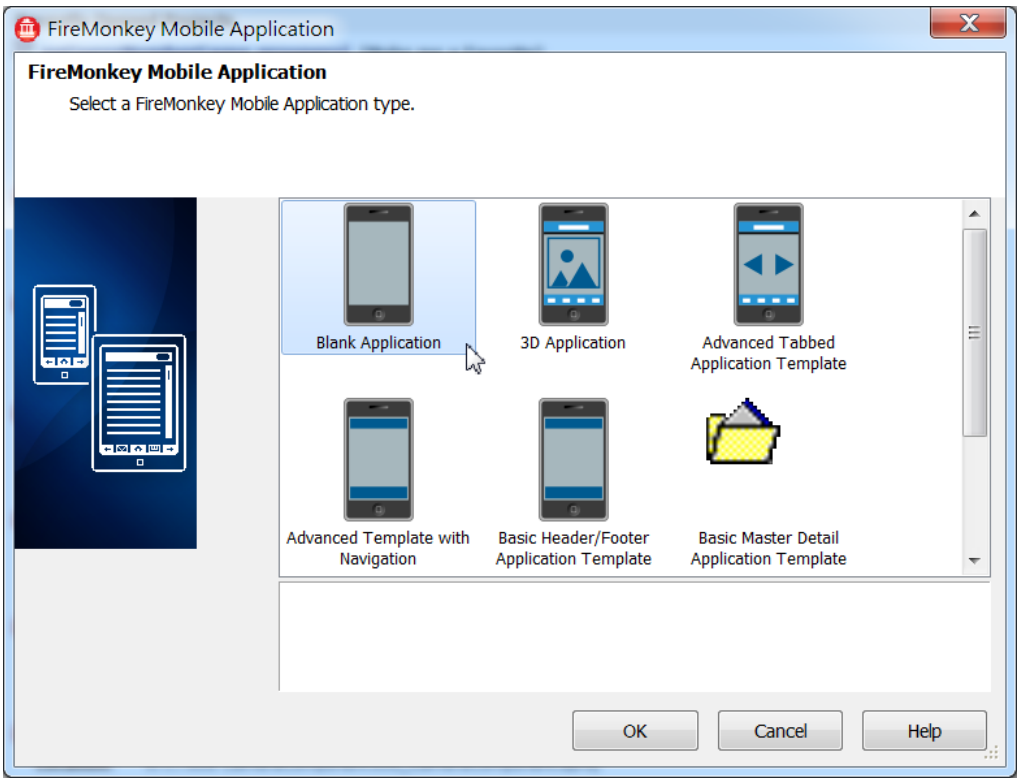
选择菜单中的 File|New 选项并且选择要建立的项目型态

第三种建立项目的方法是点选整合发展环境右下方的工具盘中的项目型态选项来选择要建立的项目型态，如下图所示：



点选整合发展环境中右下方工具盘中的选项来建立项目型态

不管您喜欢使用那一种方法，现在请选择建立『FireMonkey Mobile Application』项目，C++Builder for iOS 整合发展环境便会显示如下的对话盒询问您要建立的项目种类:



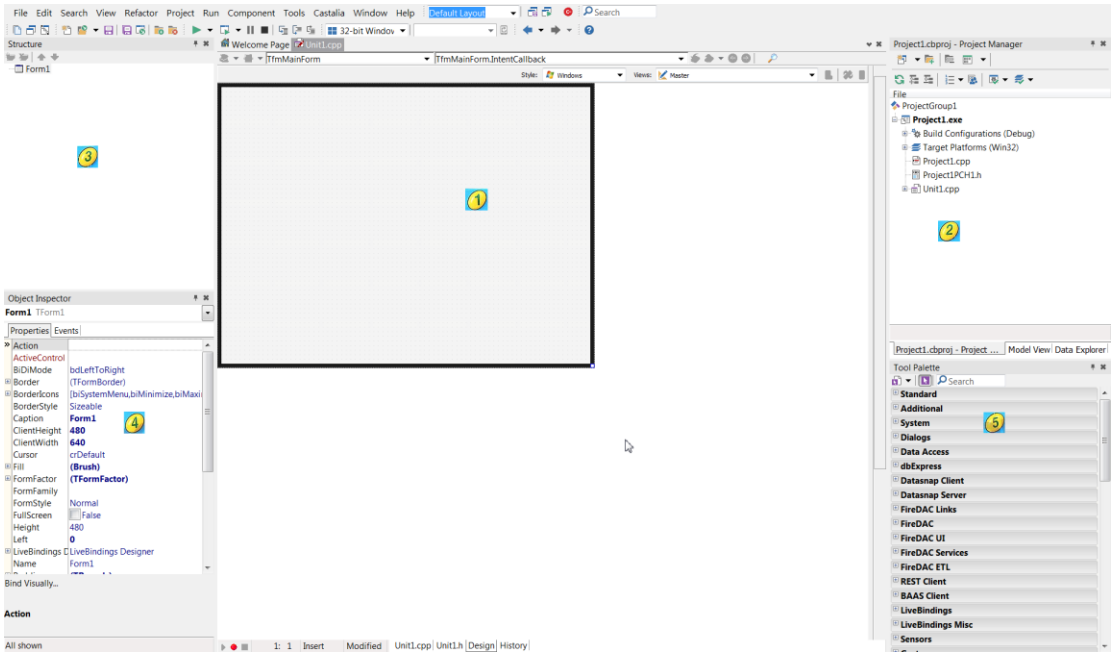
下面的表格说明了这么项目的意义:

专案种类	说明
Blank Application	建立 2 维的空白 iOS App 项目
3D Application	建立 3 维的空白 iOS App 项目
其他种类的 iOS 项目	从内定的样板 iOS App 中选择要建立的项目

在我们的第 1 个 Mobile 范例中将使用 Blank Application 项目，因此请直接点选上图中的 Blank Application 选项建立项目。

2-2 C++Builder for iOS 项目组成元素

现在 C++Builder for iOS 整合发展环境会为您建立 2 维的空白 iOS App 应用程序项目并且自动产生项目的内容档案，此时整合发展环境也会发生一些变化，如下所示:

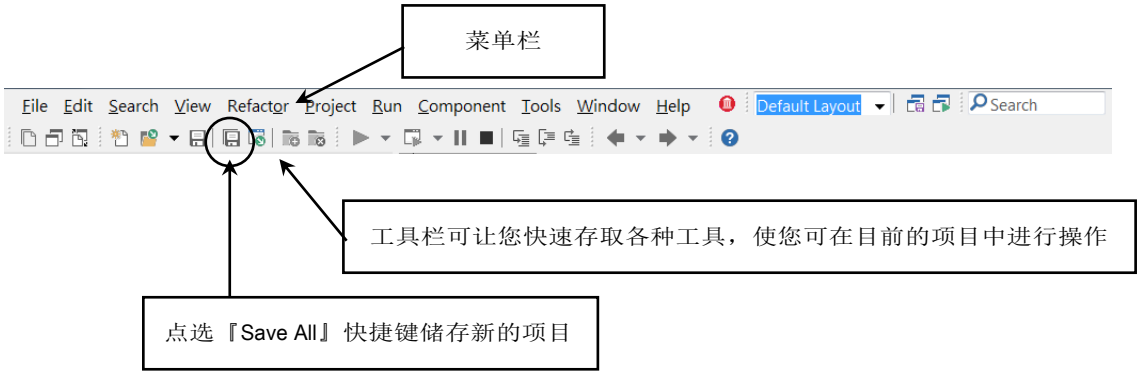


整合发展环境建立 FireMonkey 项目并且开启项目主窗体，准备让您开始设计用户接口

区域编号	说明
1	整合发展环境自动开启项目中的主窗体(Main Form)，准备让您开始设计应用程序的图形用户接口
2	项目管理员显示项目中所有的档案和相关的设定
3	项目树状架构显示主窗体中的组件架构，但由于现在主窗体中没有任何组件，因此项目树状架构中只显示了主窗体(即画面中的 Form1)

	对象查看器显示主窗体所有的特性和它的特性值，现在您就可以在对象查看器中对特性值进行任何的设定或是改变
	工具盘现在显示了所有此项目型态可使用的组件，现在您就可以拖曳这些组件到主窗体中进行应用程序的设计

在继续说明之前，让我们先储存这个范例项目，请点选工具栏中的『Save All』快捷键储存新的项目：

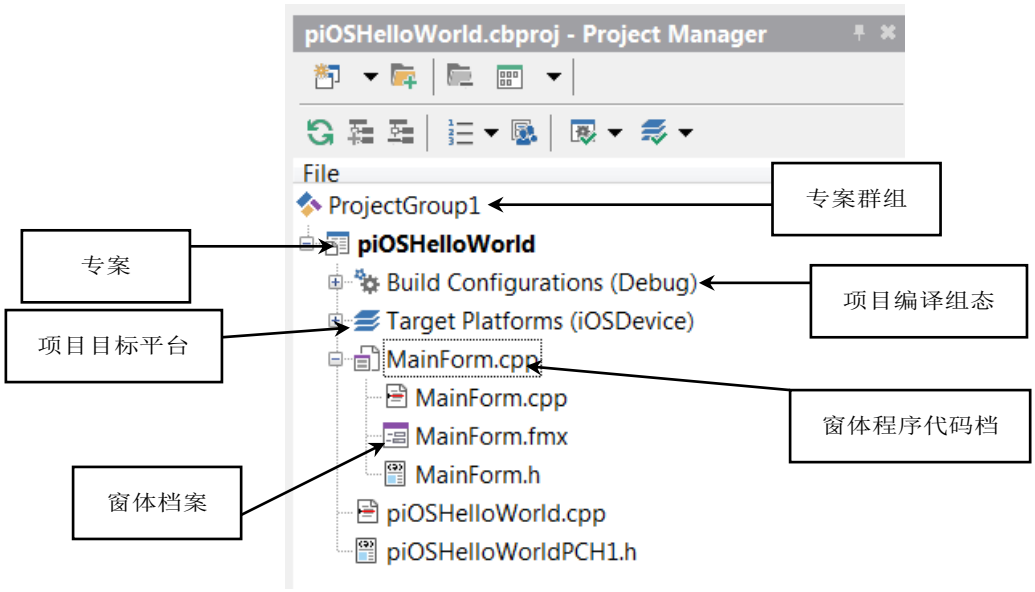


或是同时按下『Shift+Control+S』3 个按键储存项目，当储存项目时整合发展环境会询问以什么名称储存窗体和项目，请以『MainForm』储存主窗体，以『piOSHelloWorld』储存项目。

> 在储存项目时，您可以选择储存在一个特定的目录中，例如 C:\MyDemos，如果您直接储存项目而没有选择特定的目录，那么 C++Builder IDE 会把您的项目储存在 C:\Users\您的名称\Documents\RAD Studio\Projects\的内定目录之下

C++Builder for iOS 的项目是由不同的档案和相关的设定所组成，其中主要的元素就是窗体(Form)和程序代码档案，其中窗体是您设计您的应用程序图形用户接口的元素，而程序代码档案则是您撰写您应用程序逻辑程序代码的元素。在 FireMonkey 型态项目中，窗体是以『.fmx』结尾的档案，而程序代码档案则是以『.pas』结尾的档案，每一个窗体都会拥有一个对应的程序代码档案。

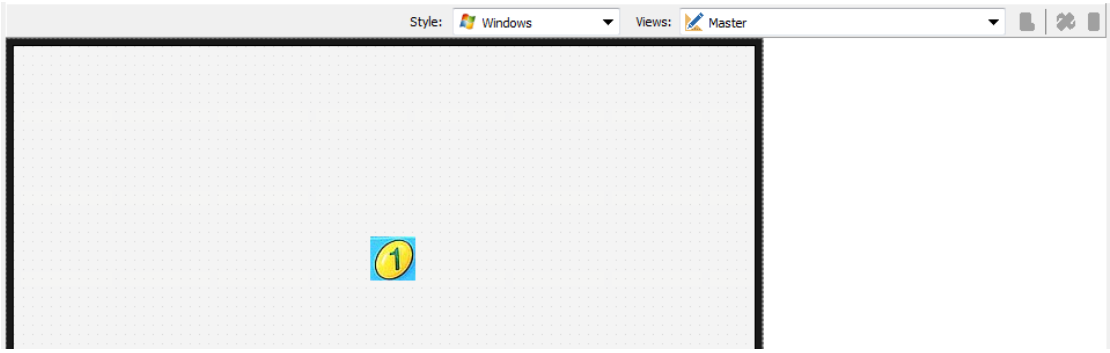
除了窗体和程序代码档案之外，项目中尚有其他的元素，这些元素都会显示在项目管理员中。其中的『项目编译组态』可设定目前项目是使用『除错』模式，『部署』模式或是『客制化』模式。而『项目目标平台』则可设定项目编译的目标平台，FireMonkey 项目可支持 Win32，Win64，Mac OSX 或是 iOS 平台，由于我们是使用 C++Builder for iOS 开发，而且在 1-2 小节中我们设定『iOS Simulator』为内定平台，因此在项目中的 Target Platforms 节点中可以看到 iOS Simulator 被设定为目前项目使用的平台：



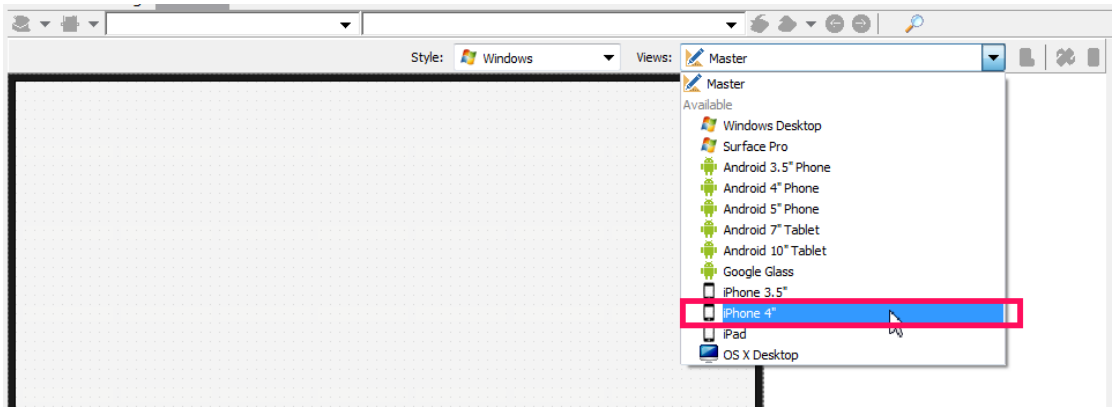
项目管理员

在 IDE 中新增了所谓的 MDD(Multi-Device Designer)的功能，基本上 MDD 是藉由多个 View 来设计不同平台的 UI，当开发人员在 C++Builder IDE 中建立 MDD Application 项目后 IDE 会显示所谓的 Master View，Master View 是所有其他子 View 的父 View，我们马上会建立一个 iPhone 的子 View。

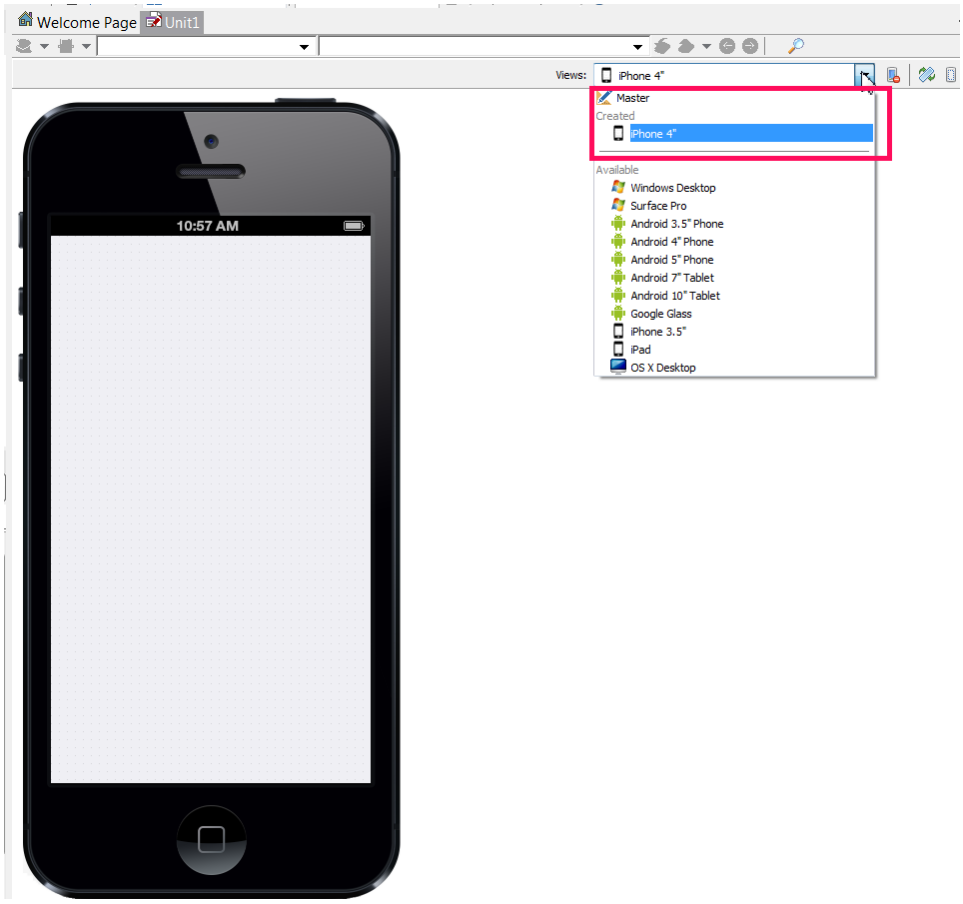
开发人员在 Master View 上加入的任何组件都会自动继承到其他子妁 View 之中。当然开发人员也可以在子 View 中直接进行 UI 设计，



现在就让我们在此项目中建立一个 iPhone 子 View，请点选 IDE 右上方的 Views 下拉盒并选择其中的 iPhone 4，如下所示：



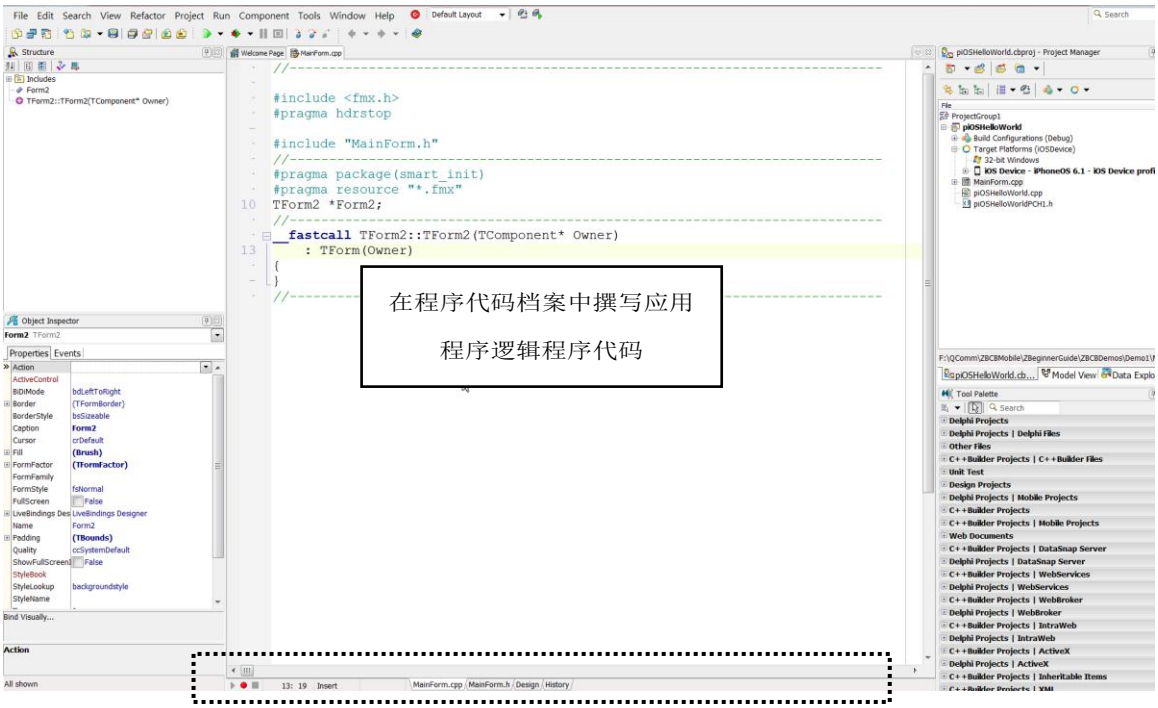
点选了 iPhone 4 子 View 之后 IDE 便会建立一个 iPhone 的设计窗体如下所示并且在 IDE 右上方的 Views 下拉盒中可以看到现在项目有 2 个 View：Master View 和 iPhone 4：



当项目建立之后，整合发展环境会自动开启主窗体，之后您就可以藉由拖曳整合发展环境右下方的组件到窗体中开始进行应用程序设计：

当项目建立之后，整合发展环境会自动开启主窗体，之后您就可以藉由拖曳整合发展环境右下方的组件到窗体中开始进行应用程序设计：

或是切换到下图到窗体的程序代码档案中开始撰写程序代码。要在窗体和它的程序代码档案之间切换，您可以点选整合发展环境中下方的『Code』或是『Design』页次切换到窗体或是程序代码。点选『Code』可切换到程序代码，点选『Design』可切换到窗体。



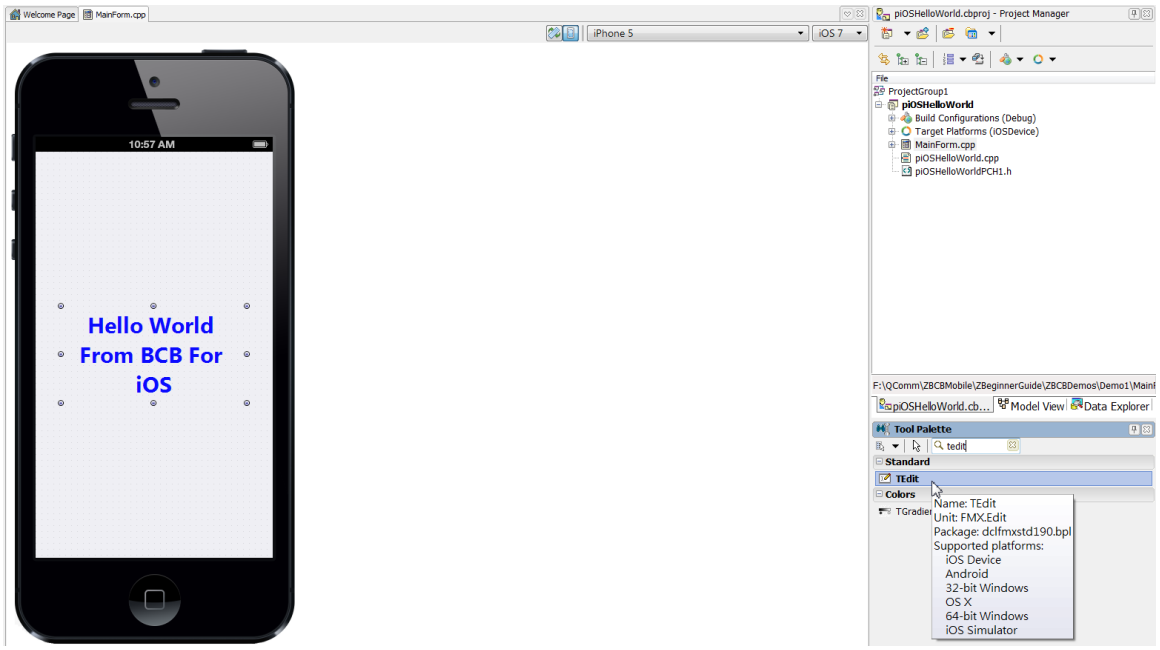
或是按下『F12』键切换窗体和程序代码页次，或是点选工具栏中的『Toggle Form/Unit(F12)』快捷键来切换：

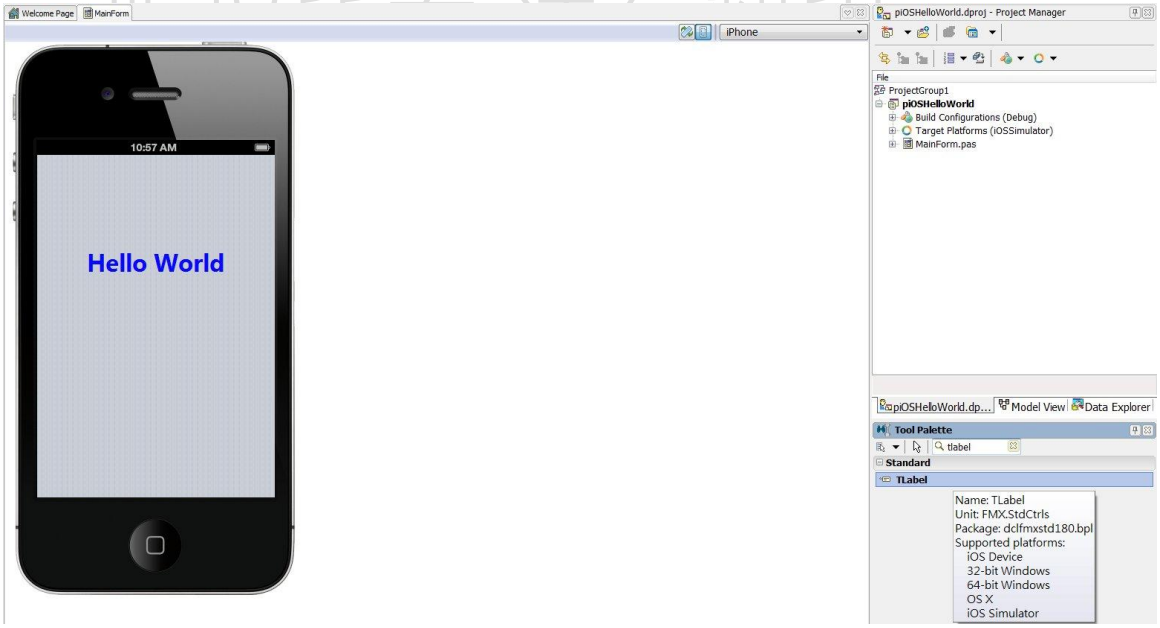
现在我们就可以开始开发您的 iOS 应用程序了。

3 开发您的第 1 个 iOS App

笔者在学习开发 iOS App 时阅读过许多使用 XCode 开发的书籍，许多书籍在介绍如何开发第一个 iOS App 时都是使用 Hello World 这个范例(事实上这几乎成了所有开发书籍的第一个范例了，不是吗?)，但对于习惯使用 C++Builder 的笔者来说，XCode 实在过于繁琐，让我们看看 C++Builder for iOS 是多简单，快速就可以完成更好的 Hello World iOS App。

现在读者继续前面储存的『piOSHelloWorld』项目，在 C++Builder for iOS IDE 加下方的工具盘上方的 Search 字段中输入 tlabel 以搜寻 TLabel 组件，接着拖曳工具盘中的 TLabel 组件到 iPhone 窗体中，如下所示：

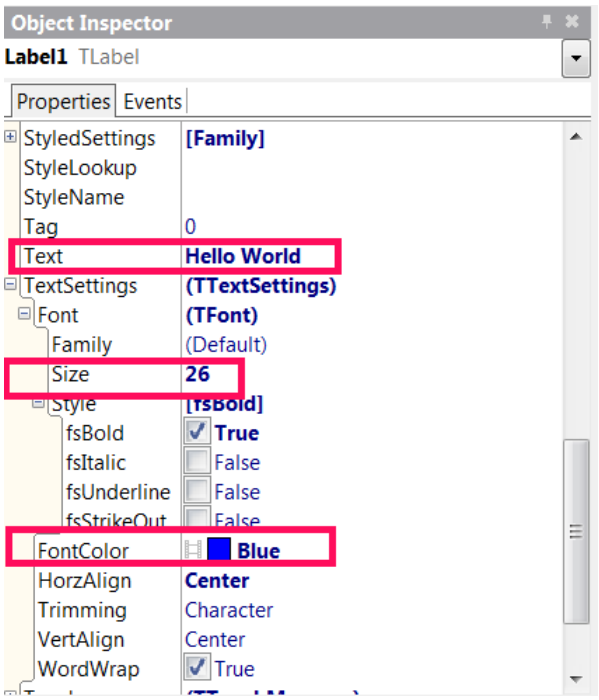




接着到 IDE 左下方的对象查看器中设定 TLabel 如下的特性值：

特性名称	特性值
TestSettings 中的 Font	26
TestSettings 中的 FontColor	Blue
Text	Hello World

对象查看器如下所示:



之后 iPhone 窗体 TLabel 组件中的 Hello World 文字就会变成上图显示的效果。

现在您只需要按下 F9 或是 Shift+Ctrl+F9，C++Builder for iOS 就会编译，并且分发『piOSHelloWorld』项目到您的 iOS 设备中执行了，例如下图就是『piOSHelloWorld』项目在笔者的 iPhone 5 手机中执行的画面(请确定 PAServer 已经执行在 Mac OS 中并且 IDE 的组态已经正确设定):

如何？使用 C++Builder for iOS 开发 iOS App 是不是太方便了呢?

让我们继续改善我们的第一个 iOS App 吧。让我们在工具盘中搜寻 TButton 组件然后拖曳到 iPhone 窗体中，然后在对象查看器中设定它的 Text 特性值为『请点击我』，如下所示：

然后使用鼠标双击 iPhone 窗体中的 TButton 组件，IDE 便会把我们带到程序代码编辑器中，请在其中撰写如下的程序代码：

```
void __fastcall TForm2::Button1Click(TObject *Sender)
{
  ShowMessage("欢迎使用 C++Builder for iOS!");
}
```

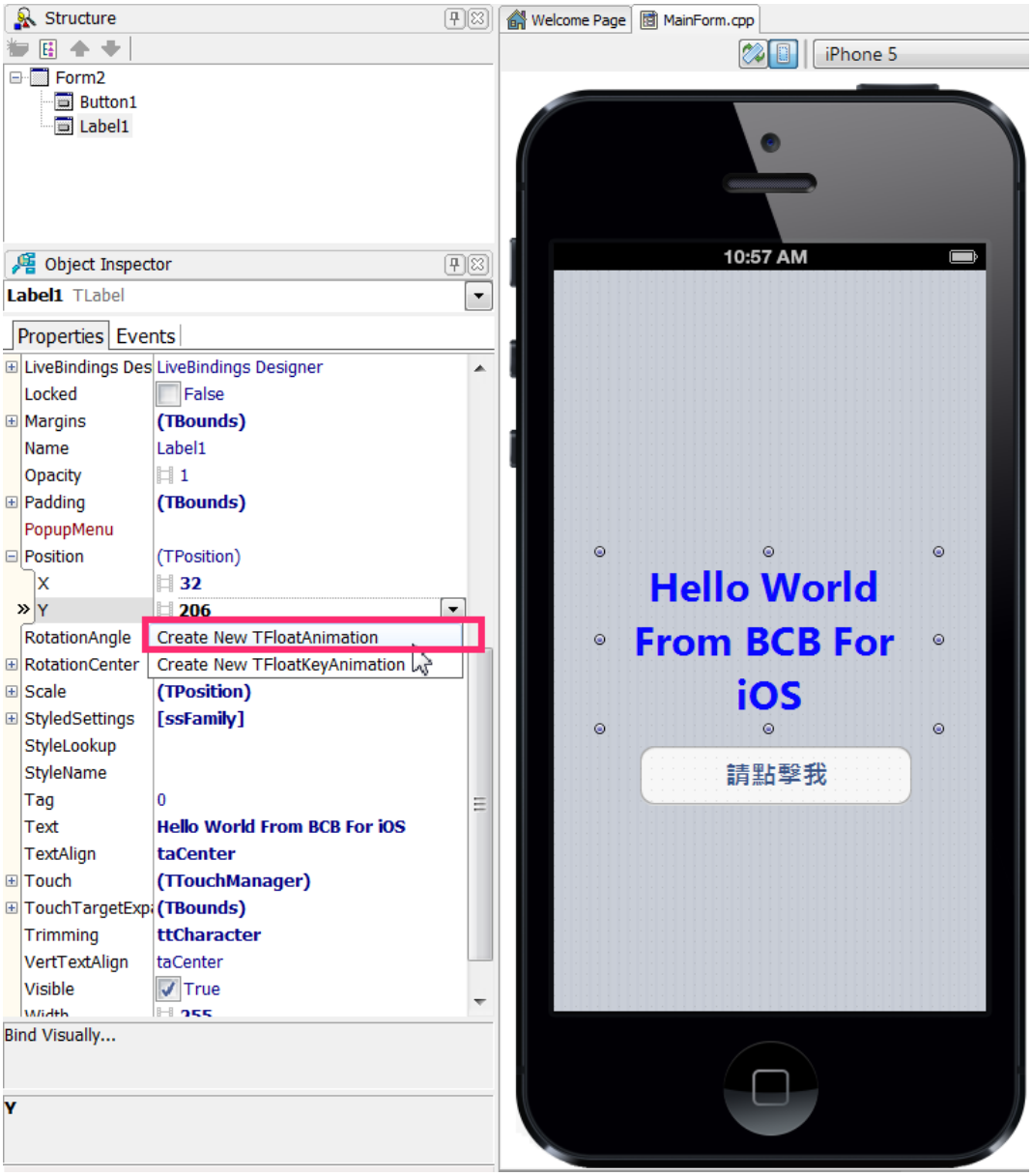
在稍后本书会说明，这段程序代码称为事件处理函式，在其中我们呼叫 ShowMessage 函式显示讯息，ShowMessage 函式是 C++Builder for iOS 的执行时期函式馆中的函式。

现在请再次按下 F9 执行『piOSHelloWorld』项目，我们就可以在 iOS 模拟中看到修改过的 piOSHelloWorld App，如果读者使用鼠标点选窗体中的『请点选我』按钮，就可以看到 piOSHelloWorld App 使用 iPhone 的原生讯息盒显示讯息，如下所示：

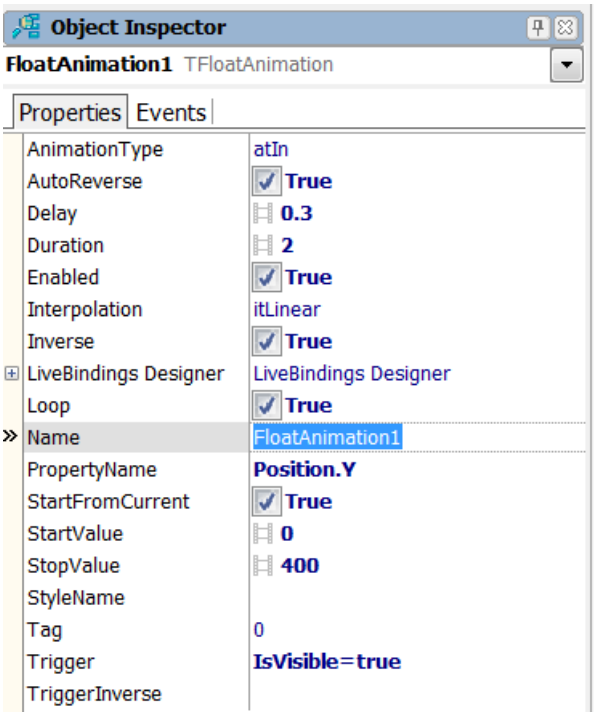
读者可以看到使用 C++Builder for iOS 撰写 iOS 程序代码并且链接可视化组件是多么的容易。在我们离开第一个 C++Builder 开发的 iOS App 之前，最后再让我们为『Hello World From BCB For iOS』这个文字加入动画的功能。

让我们在执行 piOSHelloWorld App 把『』』这个文字从目前的位置动态的往下移动，再往上移动回原位置，如此反复不停的移动。要达到这个可视化效果非常的简单，因为这基本上就是把 iPhone 窗体中的 TLabel 组件在 Y 轴移动和变化。

因此请回到 C++Builder for iOS IDE，点选 iPhone 窗体中的 TLabel 组件，然后在对象查看器中打开它的 Position 特性，我们可以在其中看到它的 X 和 Y 两个子特性，再使用鼠标点选 Y 特性，就会如下图看到对象查看器显示一个下拉菜单，请点选其中的『Create New TFloatAnimation』以便为 TLabel 组件的 Y 轴建立动画效果对象。



此时对象查看器便会显示此新建立的 TFloatAnimation 对象的特性，如下所示：



请在对象查看器中设定 TFloatAnimation 对象如下的特性值:

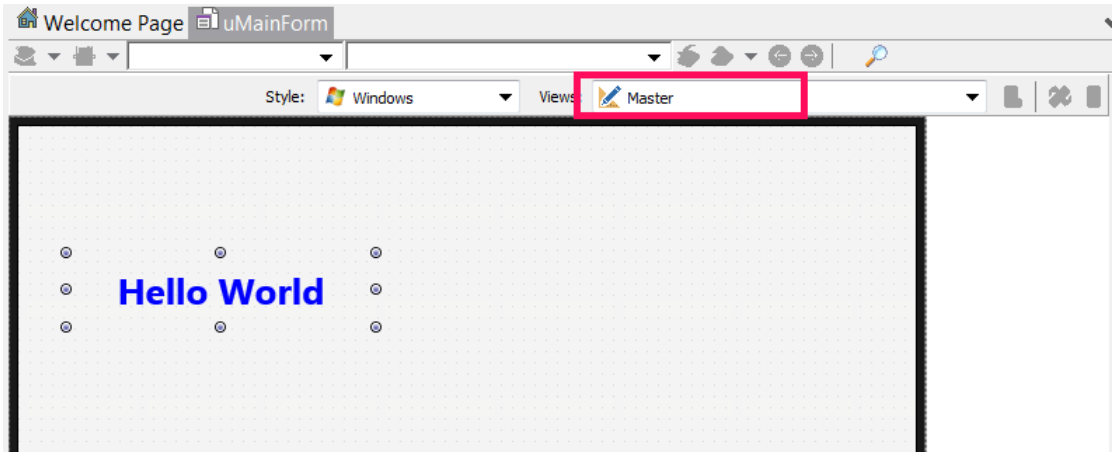
特性	特性值	说明
AutoReverse	True	当动画完成时自动以反方相再执行一次
Delay	0.3	每一动画动作之间延迟 0.3 秒
Duration	2	此动画效果一共执行 2 秒
Enabled	True	启动动画功能
Loop	True	不断重复执行此动画功能
StartFromCurrent	True	从目前 TLabel 组件的 Y 轴位置开始动画功能
StopValue	400	动画功能到达 TLabel 组件的 Y 轴位置 400 的地方时就完成
Trigger	IsVisible=true	当 TLabel 组件显示时就触发执行动画功能

现在请再次按下 F9 执行『piOSHelloWorld』项目我们，就可以在 iOS 模拟中看到修改过的 piOSHelloWorld App，此时 iPhone 窗体中的 TLabel 组件就会不停的自动上下移动，如下所示：

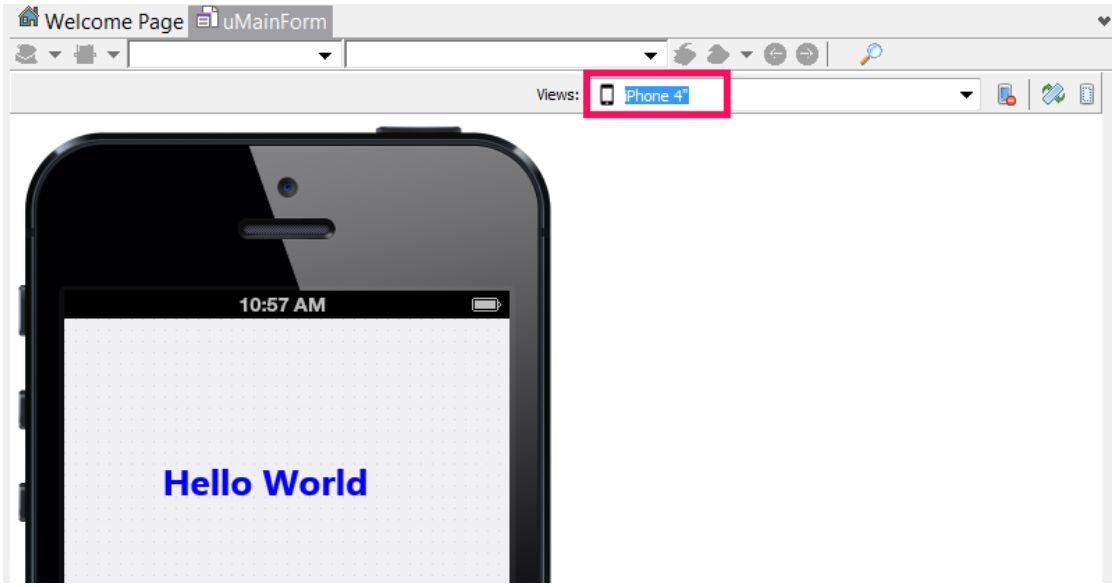
OK，现在我们完成了第一个 iOS App，这个具备动态可视化效果的 Hello World iOS App 使用 C++Builder for iOS 开发起来不但比其他 iOS 开发工具更简单，提供的功能也远远超过了只是静态的显示 Hello World 文字。从这个简单的范例就可以证明 C++Builder for iOS 比其他 iOS 开发工具更强大，更易于使用，也具体更高的生产力。

3-1 使用 MDD 设定

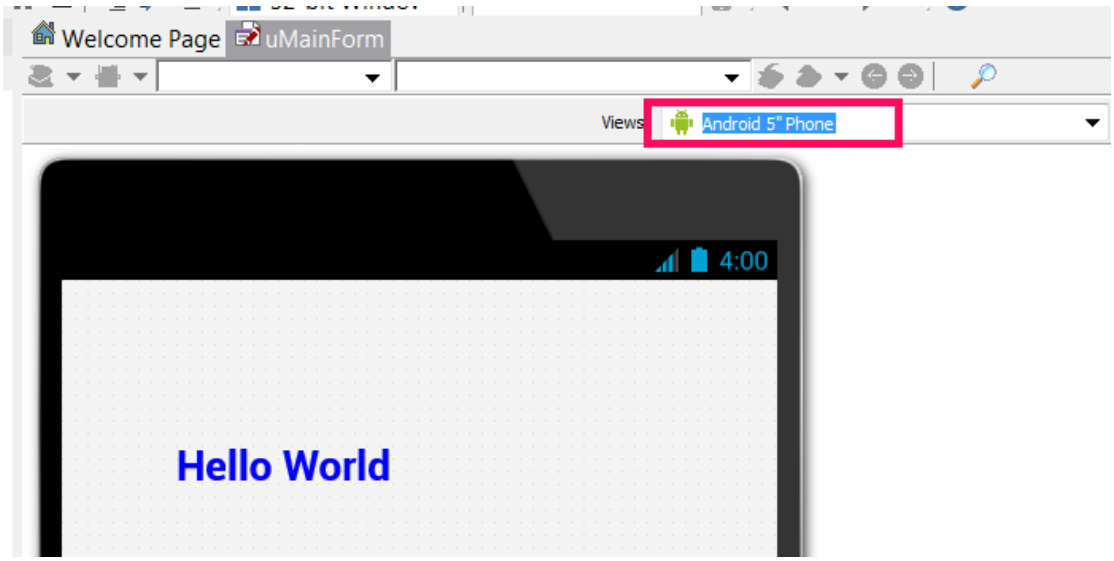
在前面的说明中我们是直接在 iPhone 的子 View 中进行 UI 设计，当然您也可以在 Master View 中进行共享设计再到特定的子 View 中进行调整，例如我们可以先在 Master View 中加入 Hello World 的 TLabel 组件：



再切换到 iPhone View 就可以看到 iPhone View 继承了 Master View 中的组件以及设定：



如果此时我们再加入开发 Android View 也可以看到 Android View 也继承了 Master View 中的组件和设定：

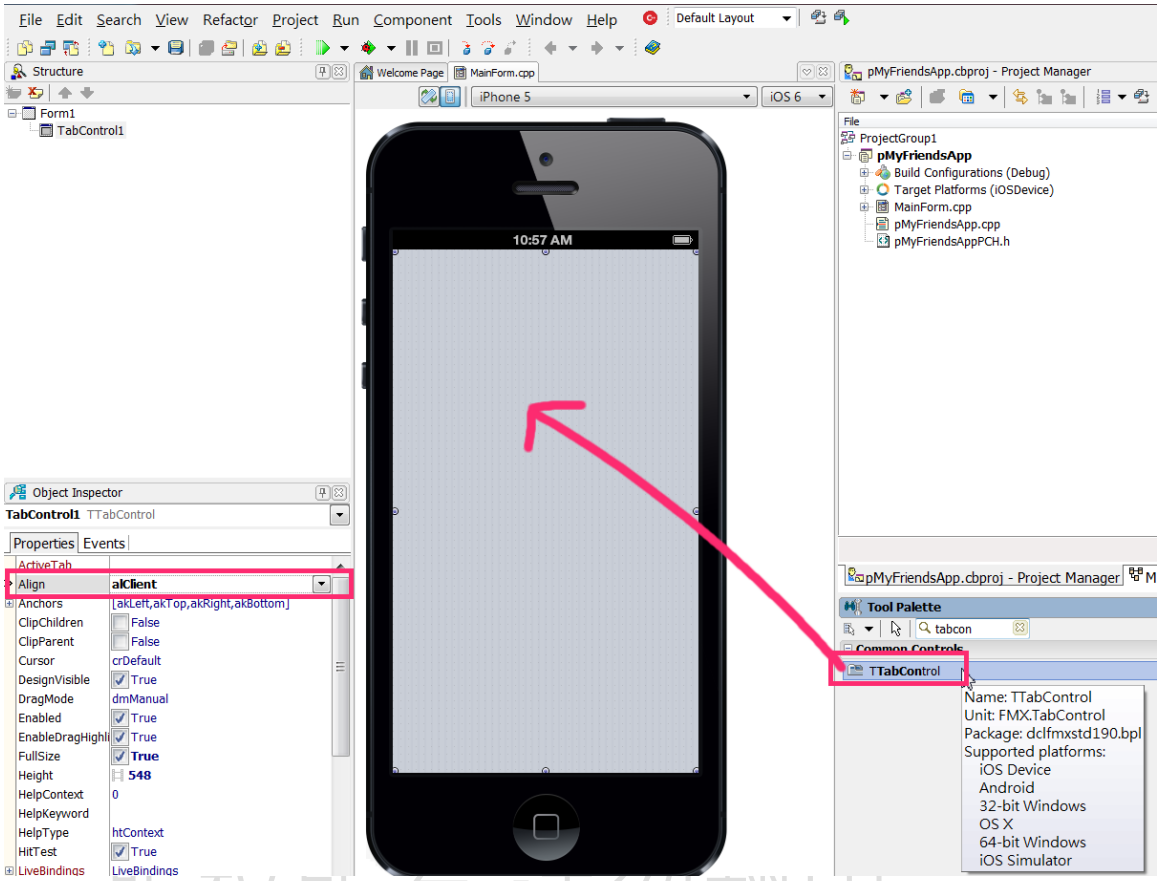


接下来读者将开始学习如何有效的使用 C++Builder for iOS IDE 来开发 iOS App，我们将使用 FireMonkey For Mobile 框架开发个人通讯簿 App，在开发的过程中读者将学习许多 C++Builder for iOS 的使用技巧以及 FireMonkey For Mobile 框架的组件。

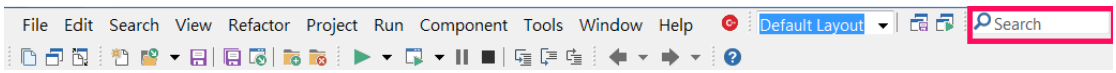
4. 使用 C++Builder for iOS 整合发展环境

请执行 C++Builder for iOS IDE，建立一个『HD FireMonkey Mobile Application』项目，并且以『pMyFriendsApp』为名称储存此 iOS App。

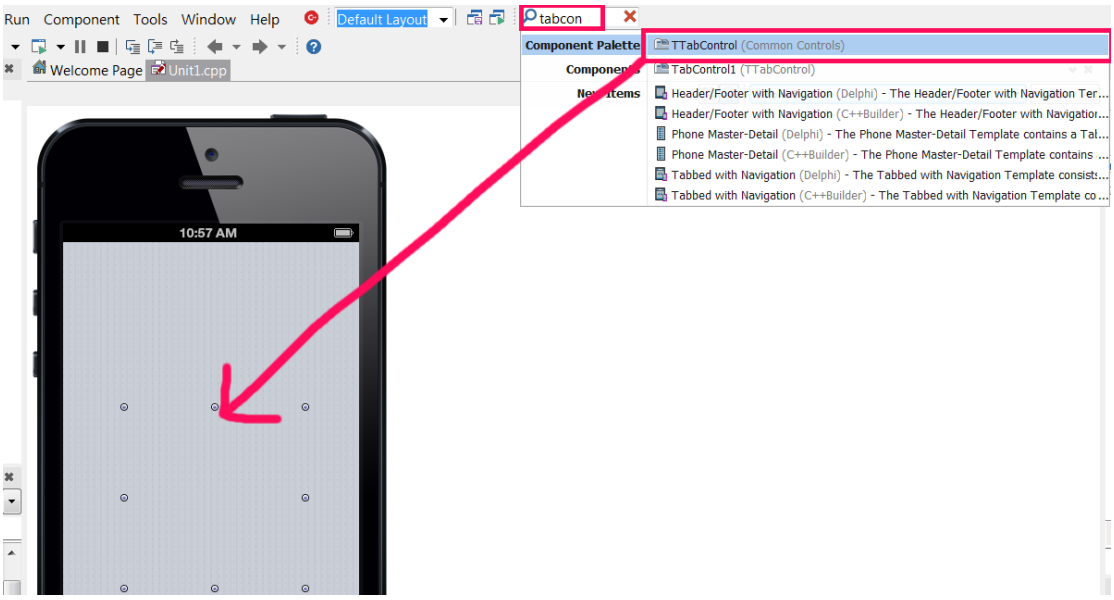
在 IDE 左下方的工具盘中搜寻 TTabControl 组件，拖曳到 iPhone 主窗体中，在对象查看器中设定 TTabControl 组件的 Align 特性值为『alClient』以便让 TTabControl 组件占据整个用户显示区域，如下所示：



由于 FireMonkey For Mobile 框架提供的组件非常的多，因此您在 IDE 右上方的 Search 控件中搜寻任何 C++Builder 相关的对象，如下所示：

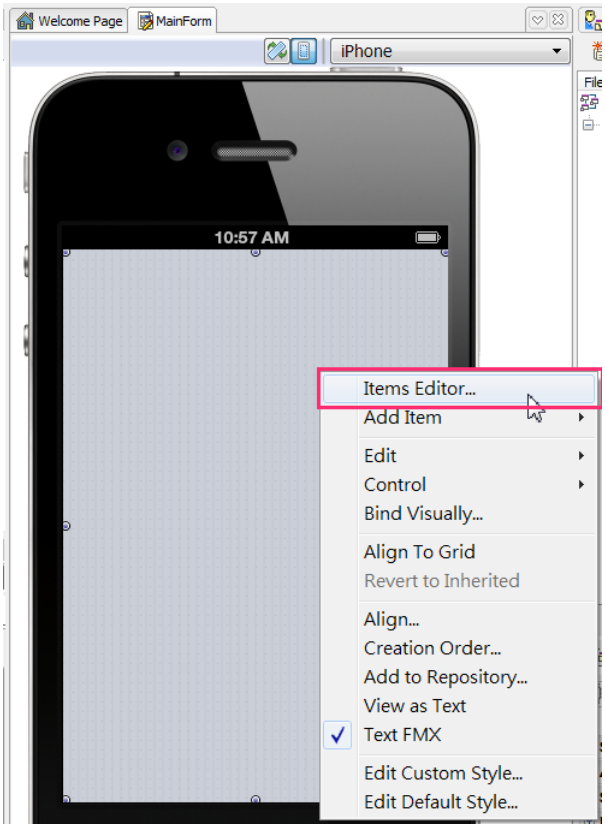


您可以在 IDE 右上方的 Search 控件中输入 TTabControl 即可找到 TTabControl 组件，接着使用鼠标双击 TTabControl 即可自动把 TTabControl 加入窗体中。或是您要搜寻的组件的部份名称，例如『tabcont』，『bcontrol』等都可以找到符合输入名称的组件。

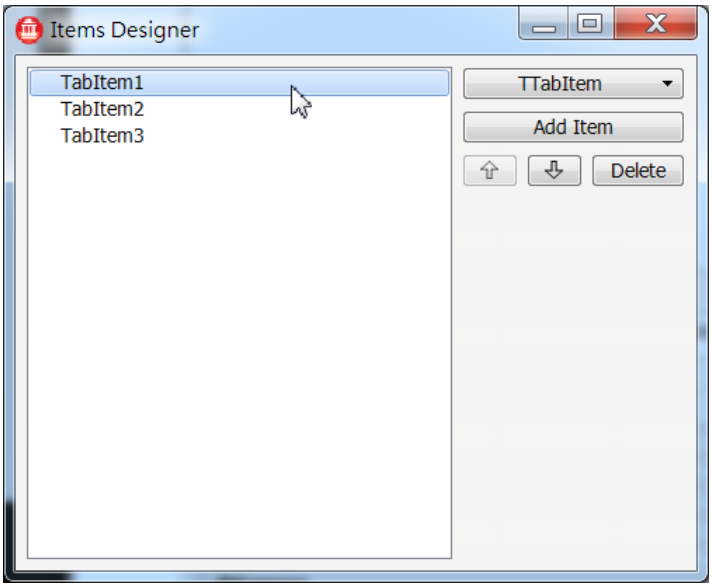


请注意『IDE Insight』会根据您现在使用的模式来搜寻内容，例如如果您是在程序代码编辑器中于 IDE 右上方的 Search 控件中搜寻 TTabControl，那么便搜寻不到，因为 TTabControl 组件无法加入到程序代码之中。

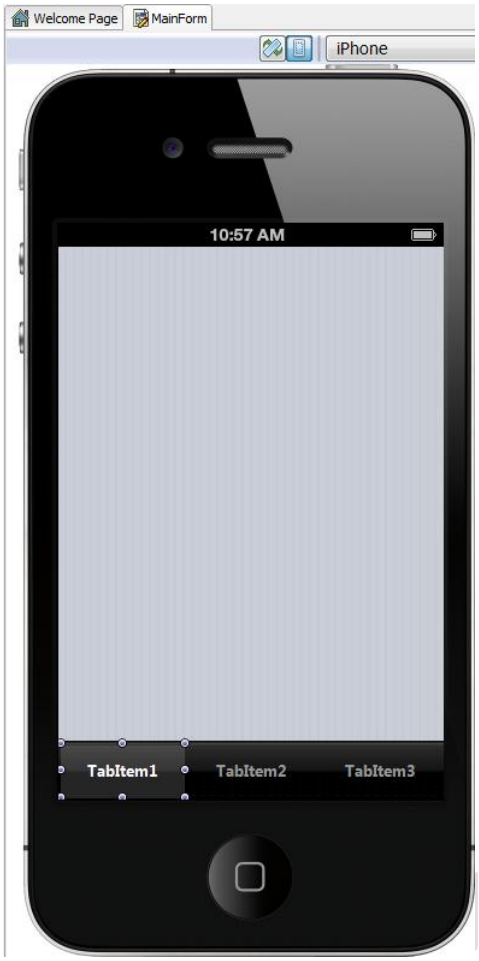
现在让我们在这个 TTabControl 组件中加入数个页面，请点选窗体设计家中的 TTabControl 组件并且点选鼠标右键，此时 IDE 便会显示一个快捷菜单，请选择其中的『Items Editor…』选项以启动选项设计者对话盒：



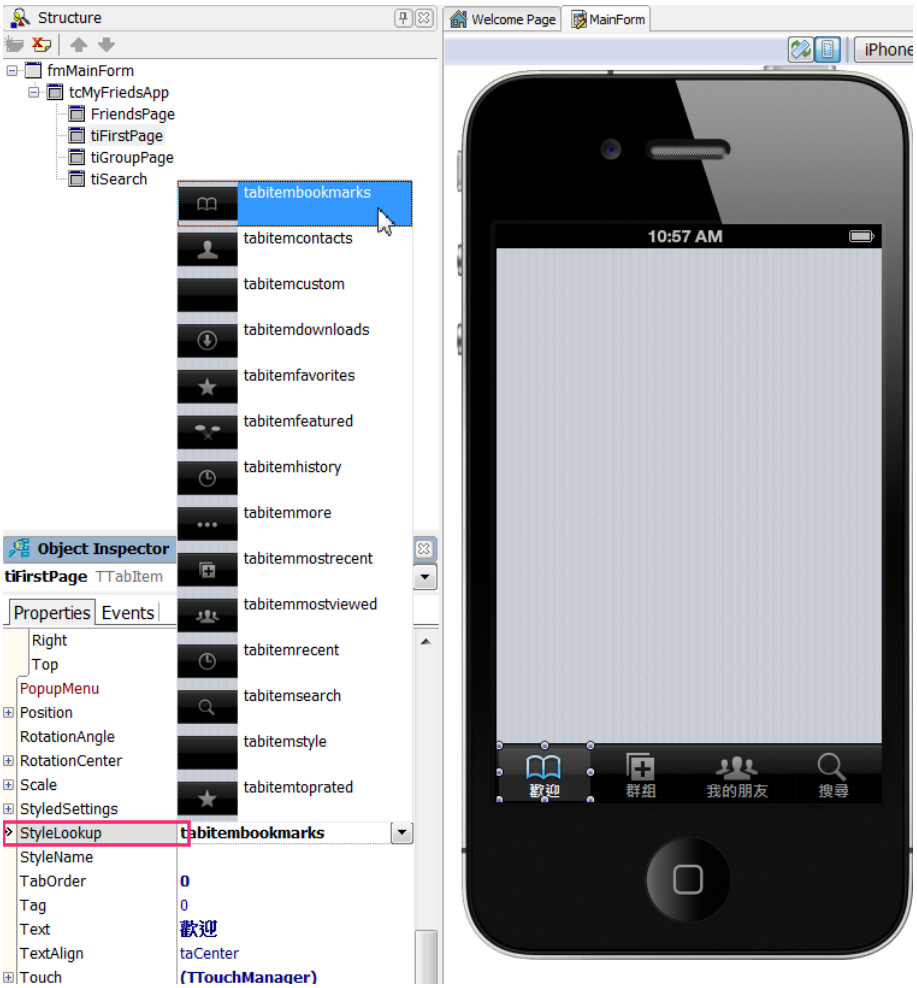
接着点选对话盒中的『Add Item』按钮在 TTabControl 组件中加入四个 TTabItem 子组件，如下所示：



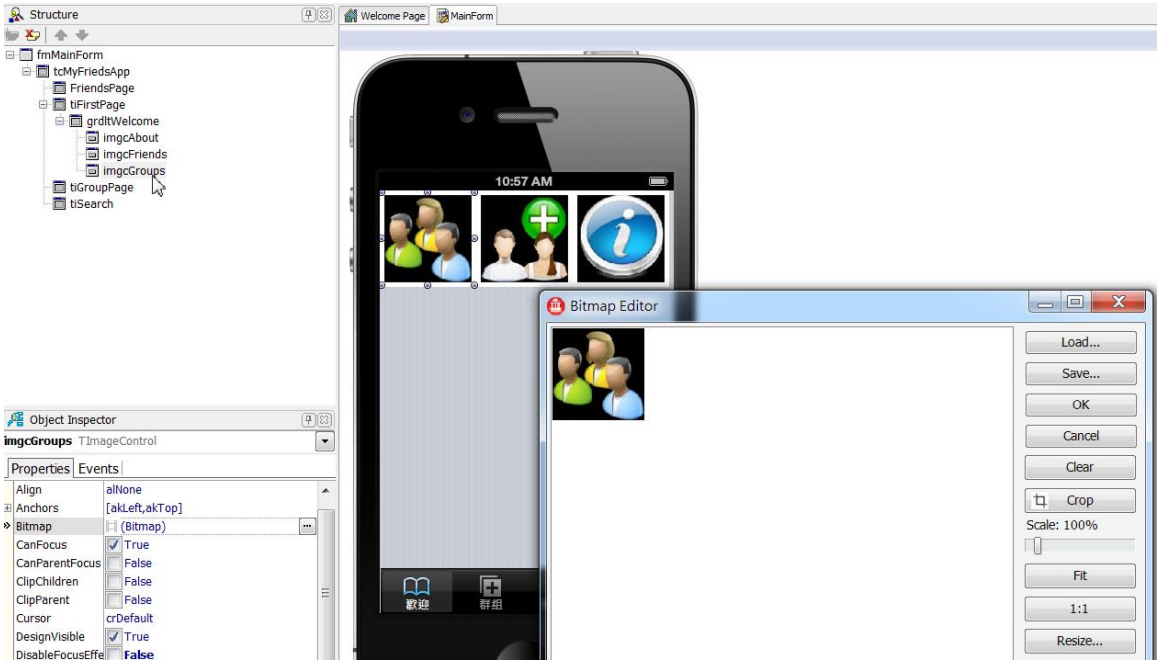
接着在对象查看器中设定 TTabControl 组件的 TabPosition 特性值为『tpBottom』，此时窗体设计者便类似如下所示：



现在在 TTabControl 组件的下方便会出现页面的按钮，让我们改变这些按钮的外观，让这个 FireMonkey 应用程序更像典型的 iOS App，请点选 TTabControl 组件下方的按钮，在对象查看器中点选 StyleLookup 特性，您就可以从其中为按钮选择不同的外观风格。例如下图就是为 4 个页面按钮选择并且设定不同外观风格的结果:



接着让我们在第一个页面加入三个 TImageControl 组件以加载和显示 3 个图像。要在 TImageControl 组件中加载图像，我们可以点选 TImageControl 组件，然后在对象查看器中双击它的 Bitmap 特性，IDE 便会显示 Bitmap 特性的特性值编辑器，在这个『Bitmap Editor』对话盒中我们就可以点选『Load…』按钮以加载需要的图像，如下所示：

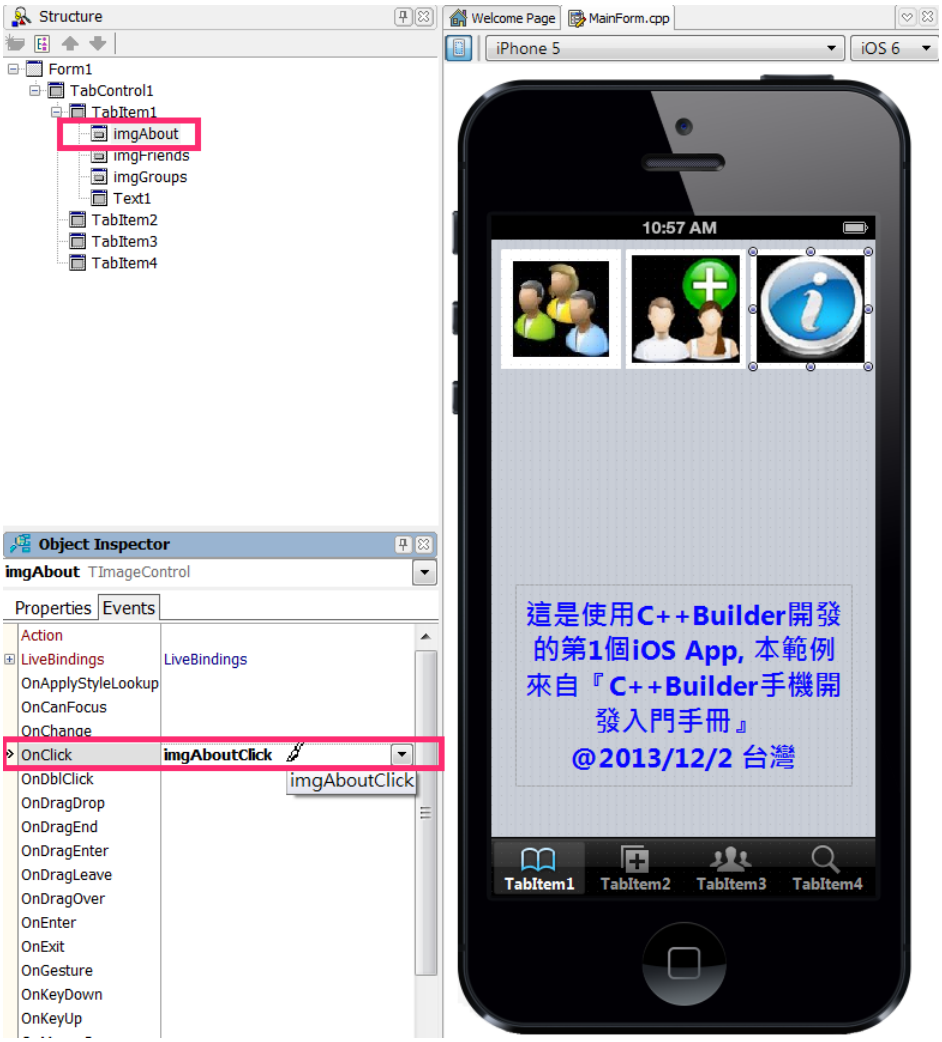


现在让我们为这个 FireMonkey App 加入一些有趣的功能，这个功能就是当这个 FireMonkey App 执行时，如果使用者点选上图中最右方的图像时就动态显示一个此 App 的信息文字，这个动态文字会从画面下方往上出现，接着在一定的时间之后这个信息文字就会消失。

这种动态显示文字的功能可以使用 FireMonkey 框架中的动画功能(Animation)轻易的完成。因此首先请在工具盘中找到 TText 组件，拖曳到 TTabControl 的第一个页面的下方，并且请在 TText 组件的 Text 特性中输入一些信息文字，如下所示:

我们希望点选主窗体中最右方的惊叹号图像时动态显示 TText 组件的内容，因此请点选惊叹号图像，在对象查看器的 Events 页次中双击它的 OnClick 项目以自动产生 OnClick 事件处理函式，如下所示:

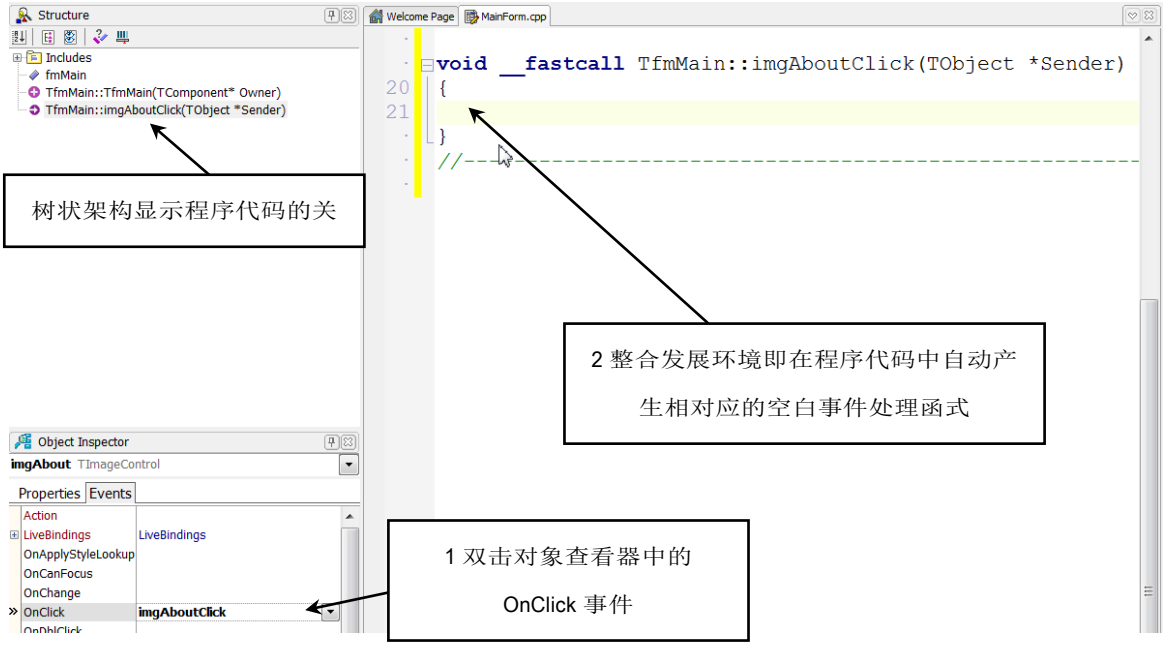
现在我们开始需要撰写一些程序代码了，请点选 TTabControl 的第一个页面中最右方的图像组件，接着点选对象查看器中的『Events』页次，并且双击其中的 OnClick 事件，如下所示:



点选窗体，在对象查看器的『Events』页次双击事件以自动产生事件处理函式

这个动作会让整合发展环境自动在程序代码页次中产生空白的事件处理函式，接着您就可以在程序代码页次中开始撰写程序代码了，这个产生事件处理函式的动作是您之后使用整合发展环境最常使用的功能之一。下图即显示了执行这个动作之后整合发展环境会自动切换到程序代码页次并且让光标自动停驻在空白的事件处理函式程序区块中，准备让您撰写程序代码。

请注意现在整合发展环境左上方的树状架构窗口也从显示组件关系切换为显示程序代码的关系：



双击事件整合发展环境即可自动建立相对应的事件处理函式

现在请您在 OnClick 事件处理函式中撰写如下的程序代码(请勿输入每行程序代码之前的行数号，例如 001, 002 等，行数号只为说明程序代码的意义使用):

```
001 ShowWelcomeText;
```

OnClick 事件处理函数调用了 ShowWelcomeText()方法来显示前面加入的 TText 组件，但在说明 ShowWelcomeText 方法之前，我们需要再撰写一些初始化程序代码。请点选整合发展环境左上方的树状架构窗口，选择主窗体对象『fmMainForm』，再于『Events』页次双击 OnActivate 以自动产生主窗体的 OnActivate 事件处理函式，如下所示:

主窗体的 OnActivate 事件处理函式在 003 行设定主窗体中的 TTabControl 组件显示第一个页面，接着 004 行呼叫 SetupWelcomeText()方法对于 TText 组件进行初始化设定。

SetupWelcomeText()方法于 010 行设定 TText 组件(txtWelcome)的 Visible 特性值为 false 以隐藏此组件，并且于 010 行改变它的 Y 轴位置，

```
txtWelcome->Position->Y = this->Height + 10;
```

的功能就是把此 Text 组件移动到主窗体可显示的区域之外，让它无法显示在主窗体的区域中。

```
001    void __fastcall TfmMain::FormActivate(TObject *Sender)
002    {
003    tcMyFriedsApp->ActiveTab = tiFirstPage;
004    SetupWelcomeText();
005    }
006
//---------------------------------------------------------------------------
007    void TfmMain::SetupWelcomeText()
008    {
```

```
009    txtWelcome->Visible = false;
010    txtWelcome->Position->Y = this->Height + 10;
011    }
```

OK，接下来就简单了。由一开始我们就把 TText 组件隐藏而且移动到显示区域之外，因此在 ShowWelcomeText()方法中我们只需需要把它移回显示区域并且显示它即可。但只是简单的如此做没什么意思，让我们使用 FireMonkey 框架的动态显示功能来显示此 Text 组件。

在 ShowWelcomeText()方法的 003 行先设定 txtWelcome 的 Visible 特性值为 true 以显示它，004 行呼叫 txtWelcome 的 AnimateFloat()方法动态的显示出来。AnimateFloat()方法的宣告原型如下:

```
    void __fastcall AnimateFloat(const System::UnicodeString
APropertyName, const float NewValue, float Duration = 2.000000E-01f,
TAnimationType AType = (TAnimationType)(0x0), TInterpolationType
AInterpolation = (TInterpolationType)(0x0));
```

AnimateFloat()的第一个参数是需要动态效果的特性名称，第二个参数是此特性动态效果之后的新数值，第三个参数是动态效果持续的时间，最后的 2 个参数是动态效果的种类，由于最后 2 个参数都拥有内定值，因此我们只需要传入前 3 个参数即可。

因此 004 行传入 AnimateFloat()方法的第一个参数是"Position.Y"，代表要对 TText 组件的垂直位置进行动态效果，第二个传入的参数值是

```
((TControl *) (txtWelcome->Parent))->Height - txtWelcome->Height + 10
```

它代表 TText 组件的新垂直位置，也就是 TTabControl 组件第一个页面的高度减去 TText 组件的高度，再加上 10 个像素的高度。

最后的参数值 2 代表整个动态效果的时间会维持 2 秒钟。

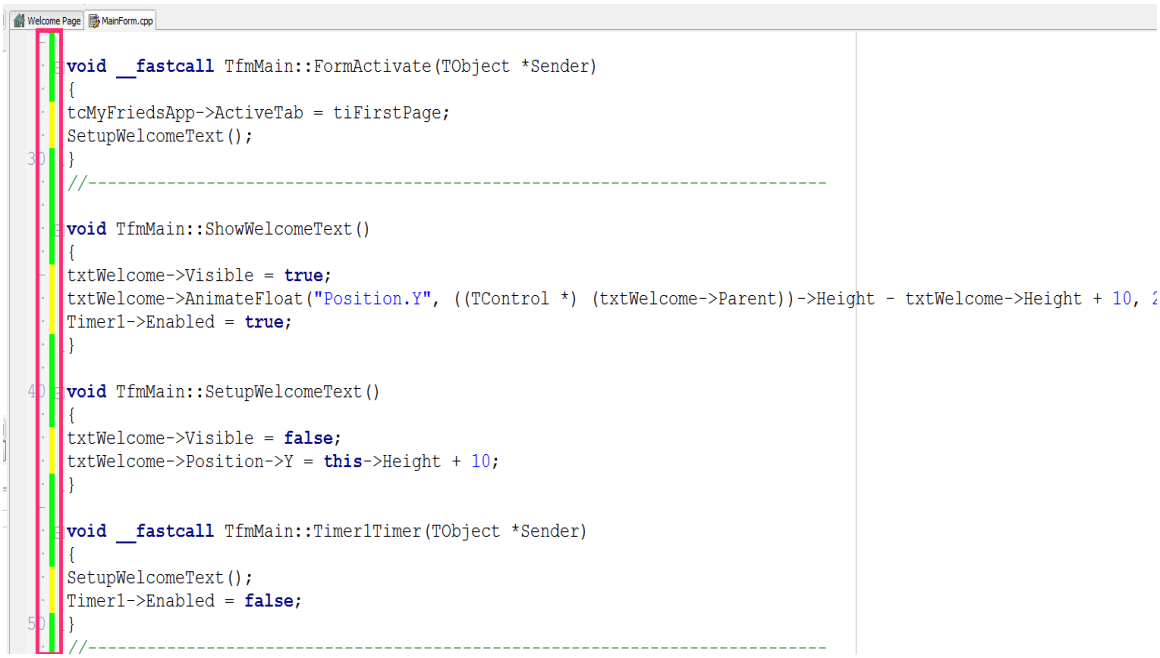
```
001    void TfmMain::ShowWelcomeText()
002    {
003    txtWelcome->Visible = true;
004    txtWelcome->AnimateFloat("Position.Y", ((TControl *)
(txtWelcome->Parent))->Height - txtWelcome->Height + 10, 2);
005    Timer1->Enabled = true;
006    }
```

最后在主窗体中加入一个 TTimer 组件，设定它的 Interval 特性值为 5000，在它的 OnTimer 事件处理函式中撰写如下的程序代码:

```
001    void __fastcall TfmMain::Timer1Timer(TObject *Sender)
002    {
003    txtWelcome->AnimateFloat("Position.Y", this->Height + 10, 2);
004    Timer1->Enabled = false;
005    }
```

OnTimer 事件处理函式的工作很简单，就是在 TText 组件显示了 5 秒之后就在 003 行再次呼叫 AnimateFloat()方法把 TText 组件回复成初始化的位置，004 行再停止 TTimer 组件的运作。

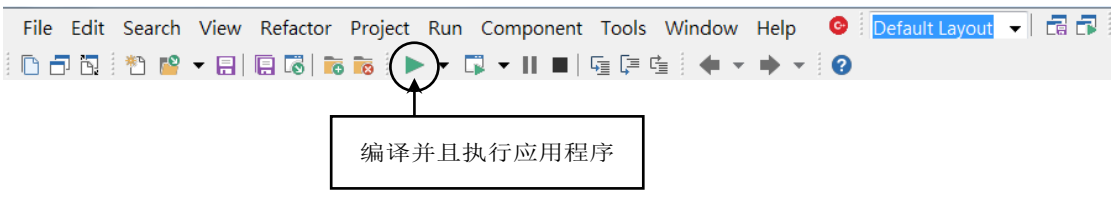
现在您的程序代码页次应该看起来类似如下所示:



整合发展环境编辑器的变更长条

请注意在程序代码页次的左方有一条绿色和黄色的线条，这个线条称为『变更长条』，绿色代表从上次储存这个程序代码页次之后没有被改变的程序代码，而黄色则代表已经被改变的程序代码，由于刚才我们加入了 3 行程序代码，因此新加入的程序代码之前的『变更长条』都是黄色的线条。如果您此时储存此项目，那么这 3 行程序代码就会变成绿色，您可以试试看。

现在您就可以试着执行此范例应用程序了，您可以点选工具栏中的『Run Without Debugging』快捷键编译并且执行此应用程序，如下所示:



或是同时按下『Shift+Ctrl+F9』键。

如果您没有打错程序代码的话，那么您就可以在 iOS 的设备中看到下面的 2 个执行画面，当您点选最右边的惊叹号图像时就可以看到文字动态的慢慢的从下往上出现非常的意思，读者使用 FireMonkey 框架可以轻易的在 iPhone/iPad 中实作出精彩的动态效果，您使用 C++Builder For iOS 整合发展环境开发的第一个应用程序已经正确的执行了，使用 C++Builder For iOS 开发 iOS App 真的又简单生产力又高，不是吗?

4-1 移动程序代码区块

再看看刚才输入的程序代码，它们都靠在编辑器的最左边，这其实是不太好的程序代码风格，让我们看看如何使用按键来移动程序代码区块。首先把光标移动到下图 55 行左边第一个位置，按下 Shift 键，再按下向下键『↓』把 2 行程序代码选择，此时编辑器会以反白显示被选择的程序区块，或是直接使用鼠标选

择这 2 行程序区块，接着同时按下『Ctrl+Shift+I』，您就可以看到被选择的程序区块整个往右方移动，如下图所示:

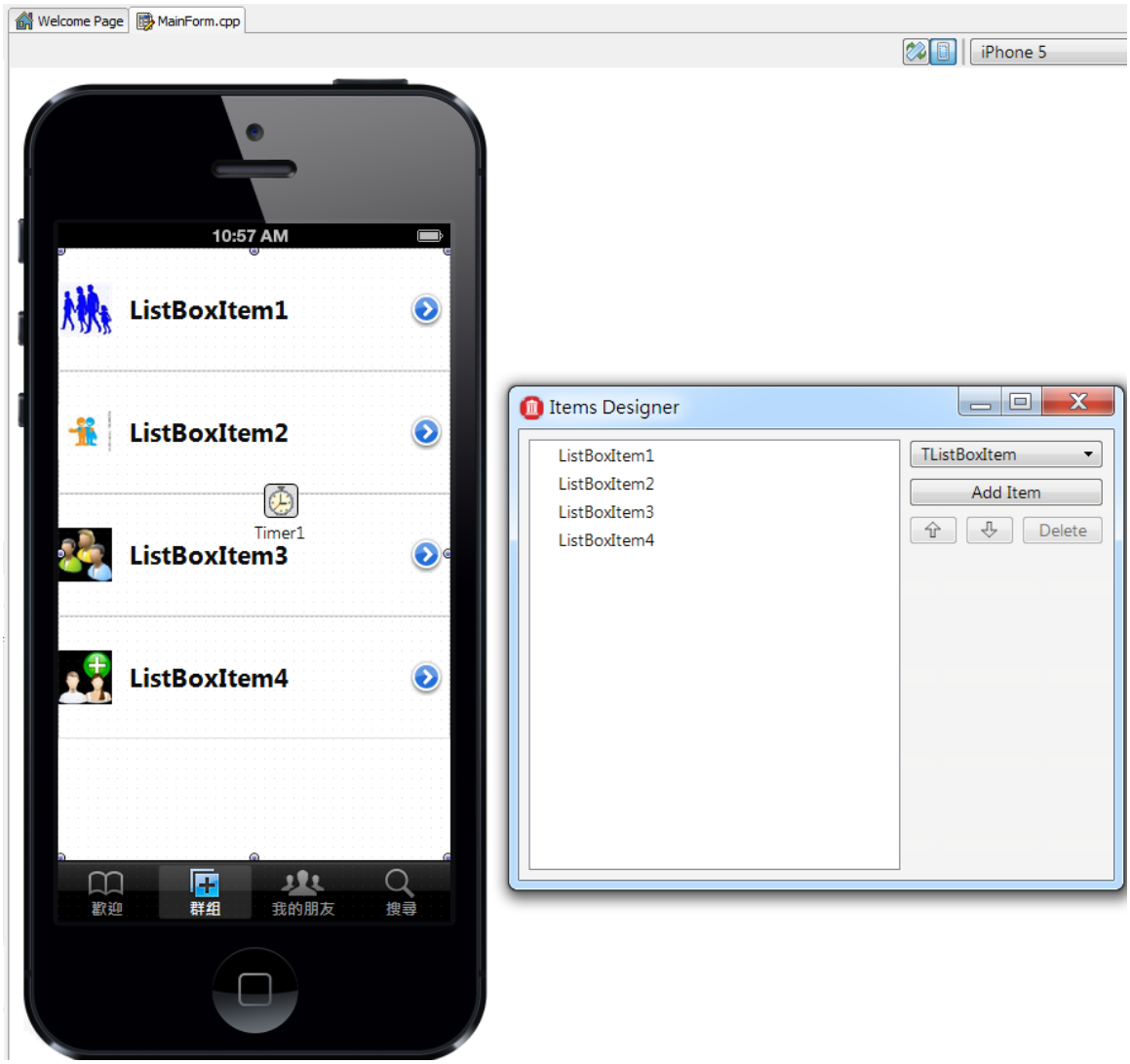
当然您也可以把程序区块往左移动，您只要按下『Ctrl+Shift+U』就可以把整个程序区块往左方移动。

同时按下 Ctrl+Shift+I 按键把程序区块往右移动

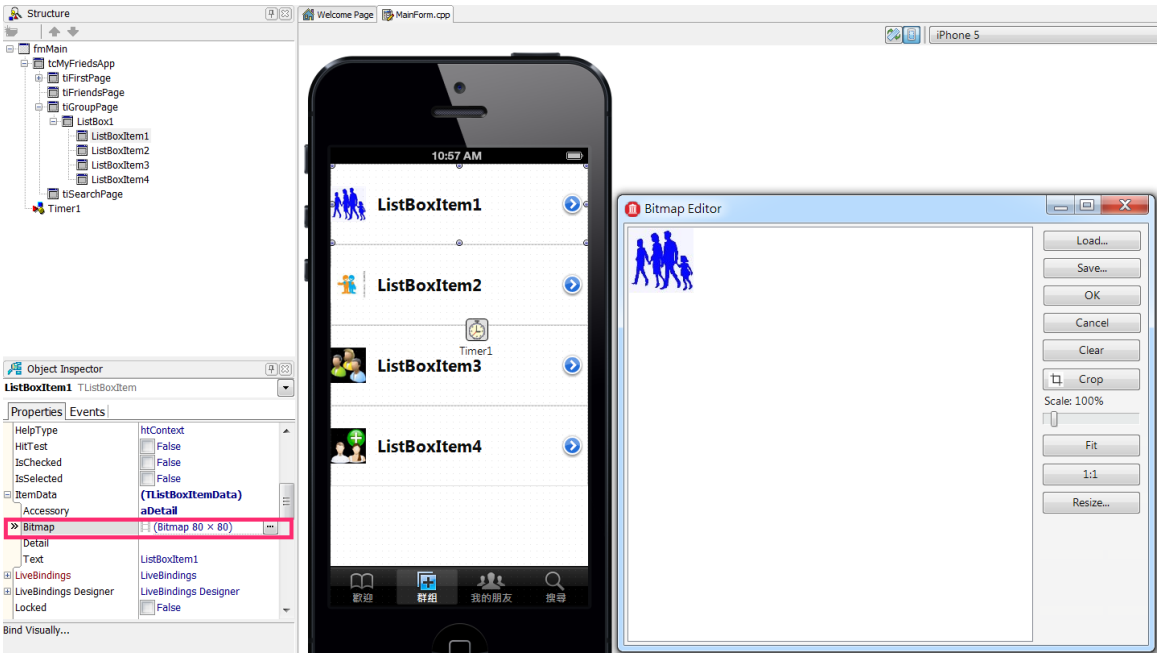
现在让我们继续开发这个范例应用程序，让它更有趣一点。

4-2 储存/切换桌面设定

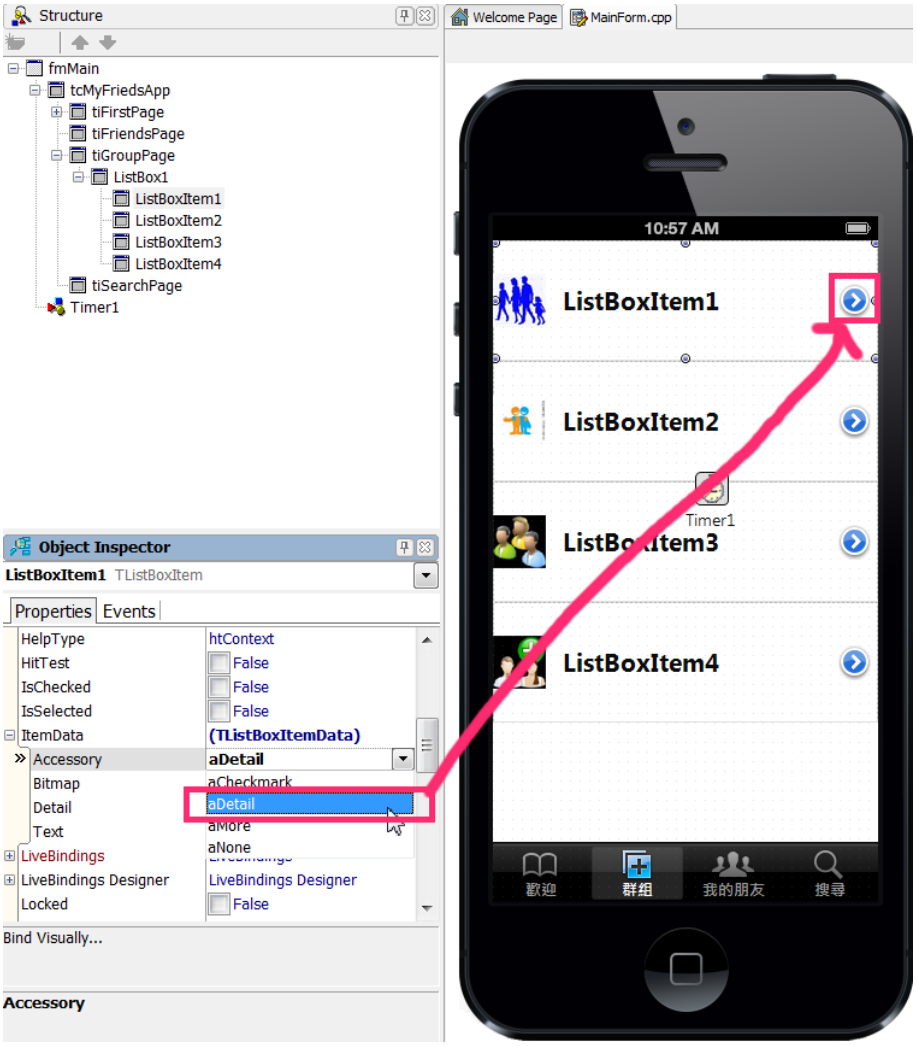
回到范例 HD Mobile FireMonkey 应用程序，点选主窗体中的『群组』按钮以开启 TTabControl 组件的第 2 个页面，从工具盘中拖曳 TListBox 组件到其中，设定 TListBox 的 Align 特性值为 alClient，接着点选鼠标右键开启 TListBox 的快捷菜单，从菜单中选择『Items Editor…』选项以开启 TListBox 的项目设计家，从右上方的下拉按钮中选择『TMetropolisUIListBoxItem』项目，再点选 4 次『Add Item』按钮以便在 TListBox 组件中加入 4 个 TMetropolisUIListBoxItem 对象，如下所示:



接着关闭项目设计家对话盒，然后在主窗体中选择 TListBox 中的 TListBoxItem 对象，然后在对象查看器点选它的 ItemData 下的 Bitmap 子特性，从下拉菜单中选择 Edit…选项以开启它的 Bitmap Editor 对话盒，点选其中的 Load 按钮以加载图像，再点选 OK 之后此图像就会显示在 TListBoxItem 对象中，如下所示，读者可以如法炮制为 TListBox 中的每一个 TListBoxItem 对象都加载图像。



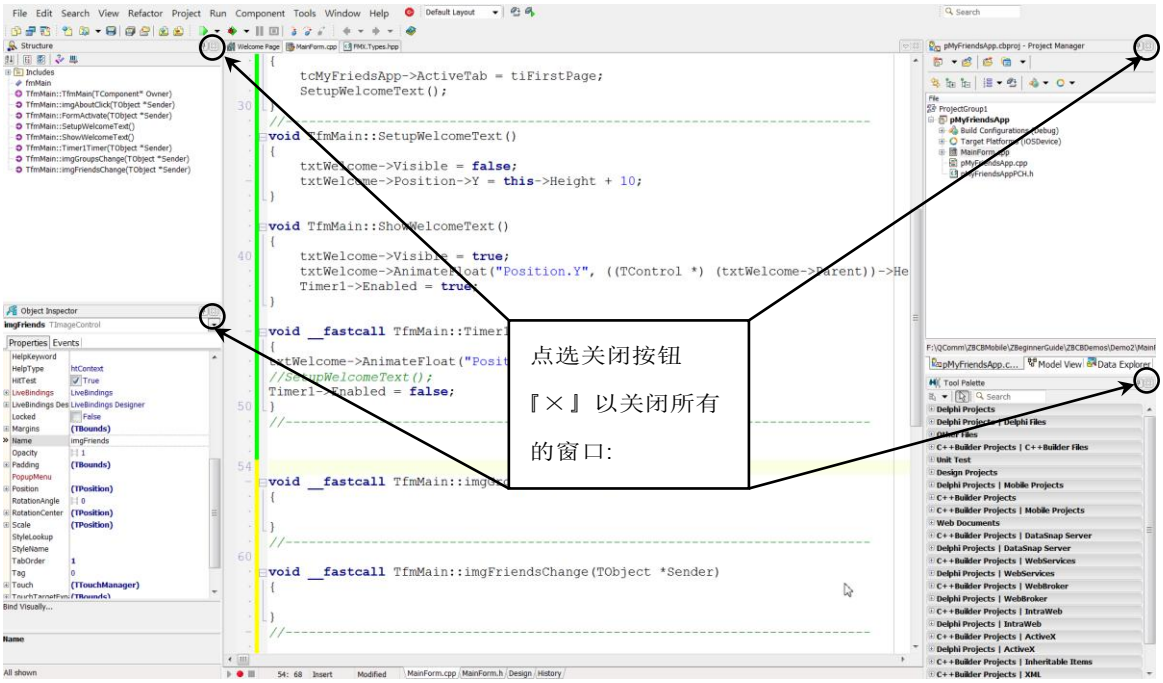
再接着让我们为每一个 TListBoxItem 对象再加入一个指示符号以代表点选当执行此 App 时如果点选 TListBoxItem 对象的话，就可以显示更为详细的信息。因此请点选 TListBoxItem 对象，在对象查看器中点选它的 ItemData 特性左方的『+』号以展开 ItemData 的子特性，在其中有一个 Accessory 子特性，请点选它，从下拉盒中再选择『aDetail』特性值，如下所示。如此一来我们就可以在 TListBoxItem 对象最右方加入一个『  』符号:



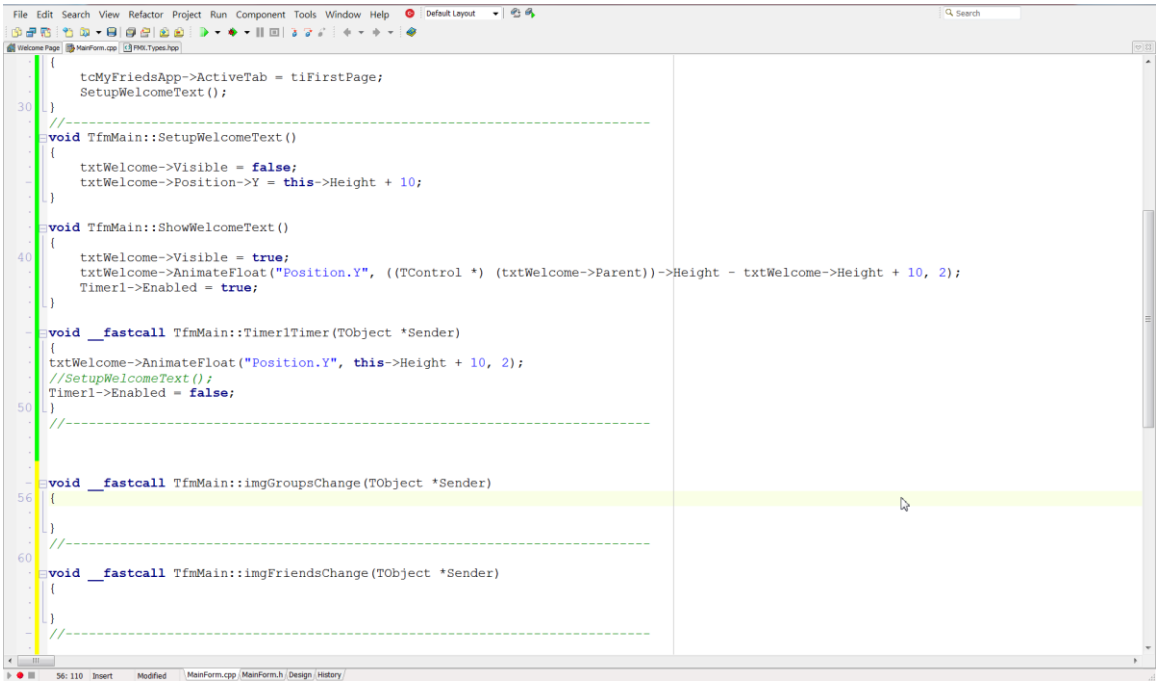
回到 TTabControl 组件的第一个页面，为最左方和中间的图像在『Events』页次分别建立它们的 OnClick 事件处理函式:

```
void __fastcall TfmMain::imgGroupsChange(TObject *Sender)
{


}
//---------------------------------------------------------------------------
void __fastcall TfmMain::imgFriendsChange(TObject *Sender)
{


}
```

好了我们终于可以开始撰写一些程序代码了，现在我们要集中焦点在程序代码而不是可视化设计家，因此请按下『F12』切换回编辑器画面，接着如下图所示单击『树状架构』，『对象查看器』，『项目管理员』和『工具盘』窗口右上方的关闭按钮『×』以关闭所有的窗口：

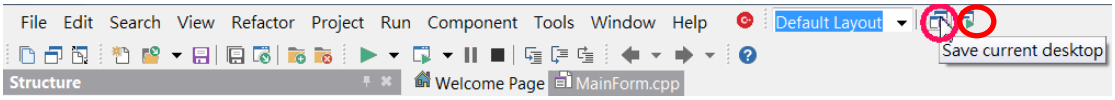


现在应该如下图只剩下编辑器窗口:



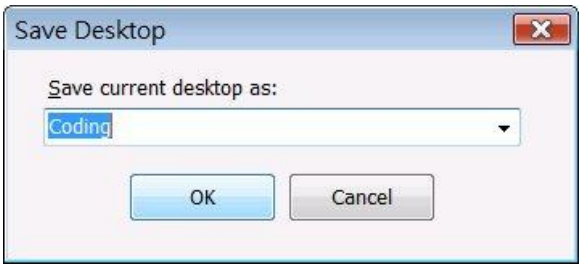
当您在专心撰写程序代码时，可能不希望看到其他不相关的窗口，因此可以使用上面说明的方法只使用编辑器来撰写程序代码，但每次都需要关闭 4 个窗口非常的麻烦，因此您可以把现在只使用编辑器来撰写程序代码的桌面组态储存起来，那么当下次您只需要编辑器时，就只要选择这个组态就可以把整合发展环境回复成现在的组态。

请点选整合发展环境右上方的『Save current desktop』按钮，如下图所示：



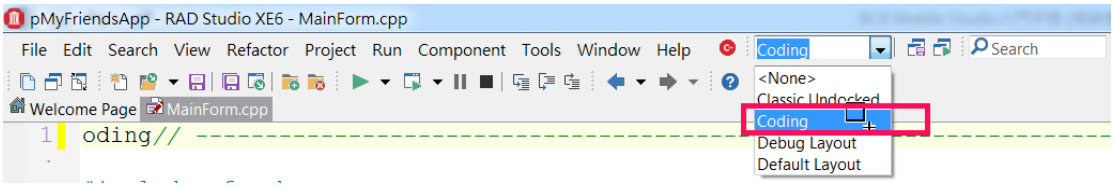
储存目前桌面的设定组态

接着 C++Builder for iOS 整合发展环境会显示如下的对话盒，询问您以什么名称储存目前整合发展环境的组态，请输入『Coding』以代表这个桌面组态是专门使用于撰写程序代码时使用的：



设定储存的桌面组态名称为 Coding

点选上图中的『OK』按钮之后，现在您如果再点选整合发展环境右上方『Help』右边的下拉盒，就可以看到如下的结果，刚才您储存的『Coding』组态已经存在于其中：



现在桌面组态下拉盒中就存在了您储存的桌面组态名称

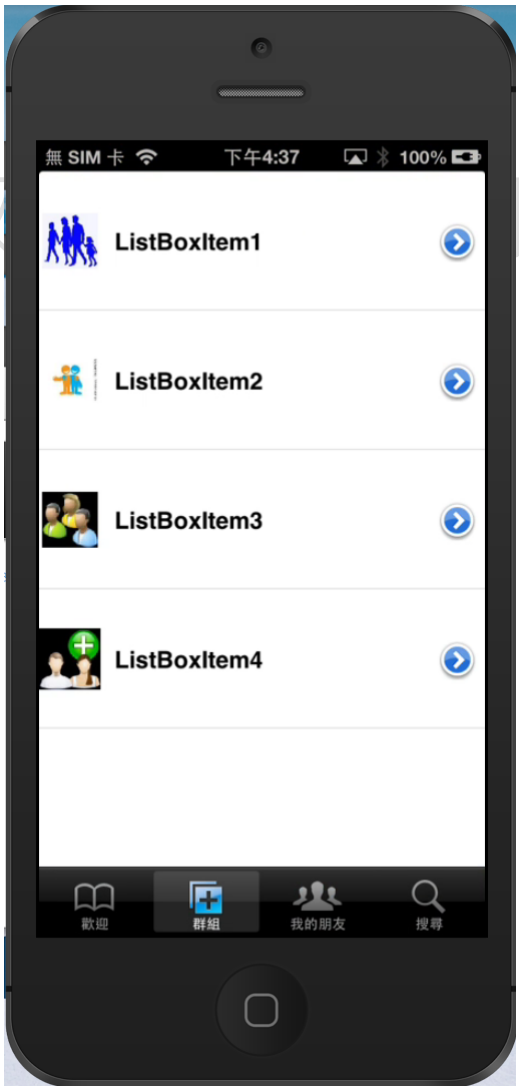
请试着在此下拉盒中选择不同的组态，例如先选择『Default Layout』，您会发现此时整合发展环境回复到一开始 5 个不同的窗口都显示的组态，再选择您储存的『Coding』组态，那么整合发展环境又会回复到只开启编辑器的组态了。下面的表格说明了下拉盒中不同组态的意义：

组态	说明
<None>	不使用任何设定的组态，维持目前的整合发展环境
<Classic Undocked>	使用 C++Builder 5~7 经典的桌面组态
<Debug Layout>	使用除错桌面组态，稍后本书会说明如何使用除错桌面组态
< Default Layout>	使用整合发展环境内定的桌面组态，同时开启 5 个窗口

在在程序代码编辑器中为前面建立的两个 OnClick 事件处理函式实作如下的程序代码，在这两个 OnClick 事件处理函式中我们只是切换目前显示的页面:

```
void __fastcall TfmMain::imgFriendsClick(TObject *Sender)
{
  tcMyFriedsApp->ActiveTab = tiFriendsPage;
}

void __fastcall TfmMain::imgGroupsClick(TObject *Sender)
{
  tcMyFriedsApp->ActiveTab = tiGroupPage;
}
```

在 IDE 中执行此范例 App，它就会执行在 iOS 设备中，如果我们点选第一个页面中不同的图像就可以看到范例 App 可以显示不同的内容，类似如下所示:

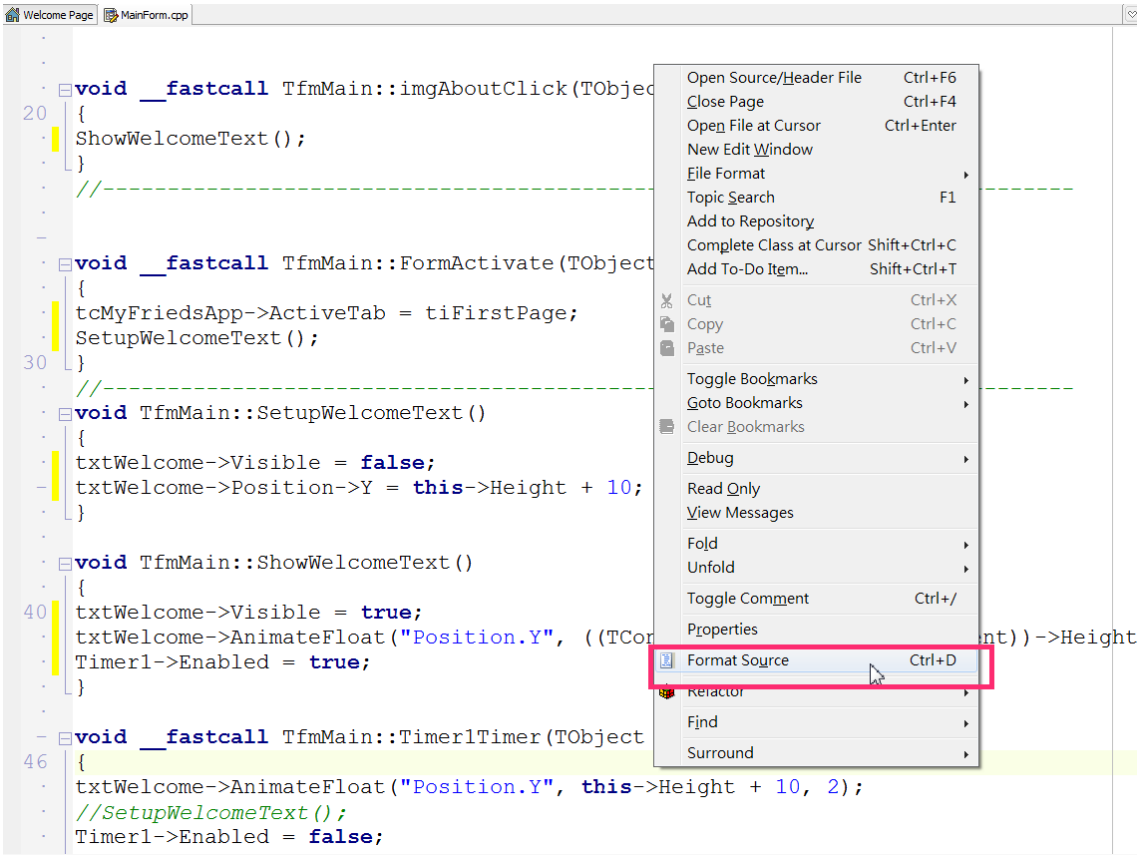


4-3 原始码格式化

请回头看看撰写的程序代码，又是所有的程序代码都在编辑器的最左边，当然您可以使用前面学习到的技巧使用『Ctrl+Shift+I』把程序代码逐一的右移，但当程序代码很多的时候这可能需要花上一些时间，在这种情形中，您可以直接使用整合发展环境的『格式化程序代码』的功能来自动格式化您的程序代码。

```
void __fastcall TfmMain::imgAboutClick(TObject *Sender)
{
ShowWelcomeText();
}
//---------------------------------------------------------------------------

void __fastcall TfmMain::FormActivate(TObject *Sender)
{
tcMyFriedsApp->ActiveTab = tiFirstPage;
SetupWelcomeText();
}
//---------------------------------------------------------------------------
void TfmMain::SetupWelcomeText()
{
txtWelcome->Visible = false;
txtWelcome->Position->Y = this->Height + 10;
}

void TfmMain::ShowWelcomeText()
{
txtWelcome->Visible = true;
txtWelcome->AnimateFloat("Position.Y", ((TControl *) (txtWelcome->Parent))->Height
Timer1->Enabled = true;
}
```

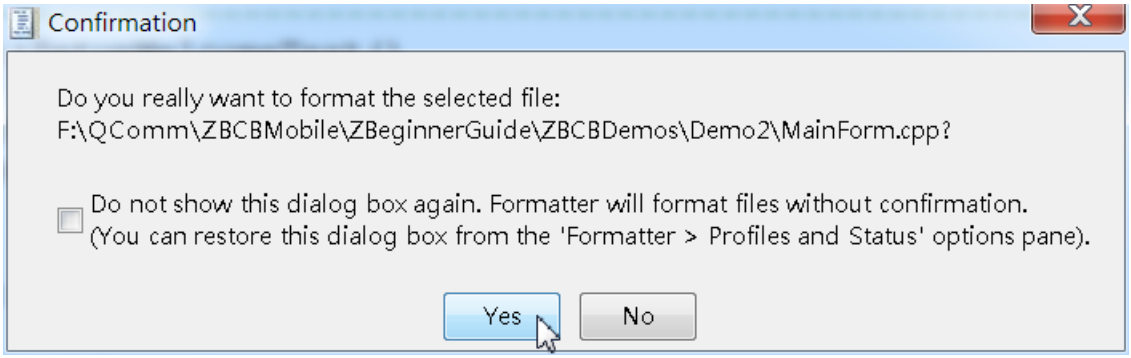
请回到编辑器中刚才实作 imgcAboutClick 的程序中，接着点选鼠标右键，IDE 便会显示一个快捷菜单，请在其中选择『Format Source』选项，如下图所示，或是直接在编辑器中同时按下『Ctrl+D』键:




```
void __fastcall TfmMain::imgAboutClick(TObjec
{
ShowWelcomeText();
}
//---------------------------------------------

void __fastcall TfmMain::FormActivate(TObject
{
tcMyFriedsApp->ActiveTab = tiFirstPage;
SetupWelcomeText();
}
//---------------------------------------------
void TfmMain::SetupWelcomeText()
{
txtWelcome->Visible = false;
txtWelcome->Position->Y = this->Height + 10;
}

void TfmMain::ShowWelcomeText()
{
txtWelcome->Visible = true;
txtWelcome->AnimateFloat("Position.Y", ((TCo                nt))->Height
Timer1->Enabled = true;
}

void __fastcall TfmMain::Timer1Timer(TObject
{
txtWelcome->AnimateFloat("Position.Y", this->Height + 10, 2);
//SetupWelcomeText();
Timer1->Enabled = false;
```

在编辑器的快捷菜单中选择『Format Source』选项

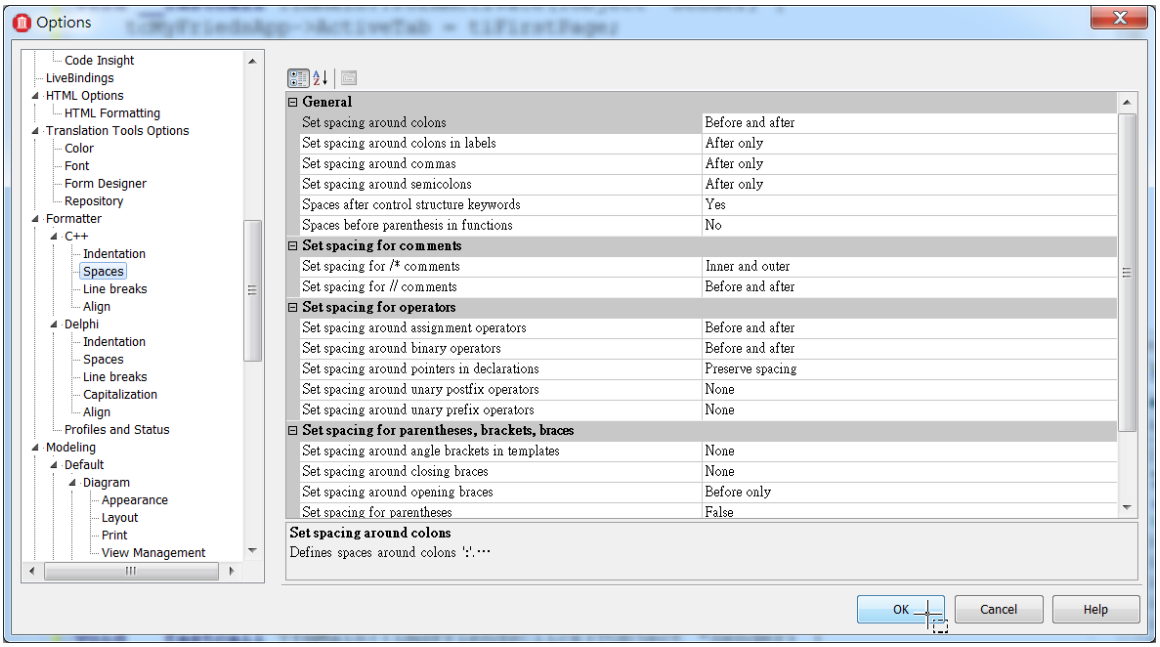
接着 IDE 会显示下面的对话盒询问您是否确定要格式化源代码，请点选『Yes』按钮，或是顺便勾选对话盒中的勾选盒，避免每次要格式化源代码是 IDE 都会询问您。



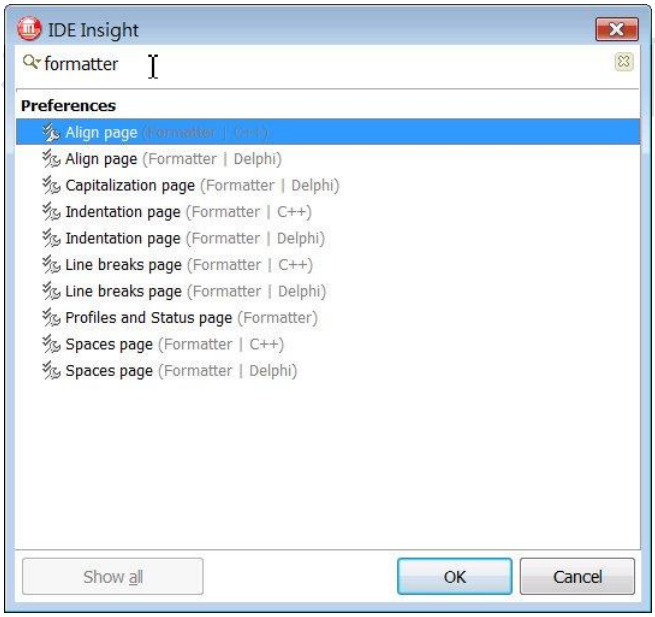
点选『Yes』按钮之后，IDE 便会开始格式化您的源代码，下图就是格式化之后的结果，您可以看到程序代码风格好多了，也更容易明了，当然您也可以养成良好的习惯，在撰写程序代码时就使用类似的风格。

```
void __fastcall TfmMain::imgAboutClick(TObject *Sender) {
    ShowWelcomeText();
}
// ---------------------------------------------------------------------------

void __fastcall TfmMain::FormActivate(TObject *Sender) {
    tcMyFriedsApp->ActiveTab = tiFirstPage;
    SetupWelcomeText();
}

// ---------------------------------------------------------------------------
void TfmMain::SetupWelcomeText() {
    txtWelcome->Visible = false;
    txtWelcome->Position->Y = this->Height + 10;
}

void TfmMain::ShowWelcomeText() {
    txtWelcome->Visible = true;
    txtWelcome->AnimateFloat("Position.Y",
        ((TControl *)(txtWelcome->Parent))->Height - txtWelcome->Height +
        10, 2);
    Timer1->Enabled = true;
}

void __fastcall TfmMain::Timer1Timer(TObject *Sender) {
    txtWelcome->AnimateFloat("Position.Y", this->Height + 10, 2);
    // SetupWelcomeText();
    Timer1->Enabled = false;
}
// ---------------------------------------------------------------------------

void __fastcall TfmMain::imgFriendsClick(TObject *Sender) {
    tcMyFriedsApp->ActiveTab = tiFriendsPage;
```


格式化源代码功能是根据 IDE 中如何格式程序代码的设定而执行的结果，您当然也可以控制如何格式化您的程序代码，或是使用什么风格来格式化您的程序代码。您可以点选 IDE 上方菜单中的 Tools|Options...选项启动 Options 对话盒，然后在其中找寻 Formatter|C++Builder 选项，在这个选项之中有数 10 个如何格式化源代码的设定，您可以使用这些设定来定义您自己喜欢使用的风格，如下所示:



或是在 IDE 中按下 F6，在 IDE Insight 对话盒中直接输入 formatter 就可以搜寻到格式化源代码的设定选项，如下所示:



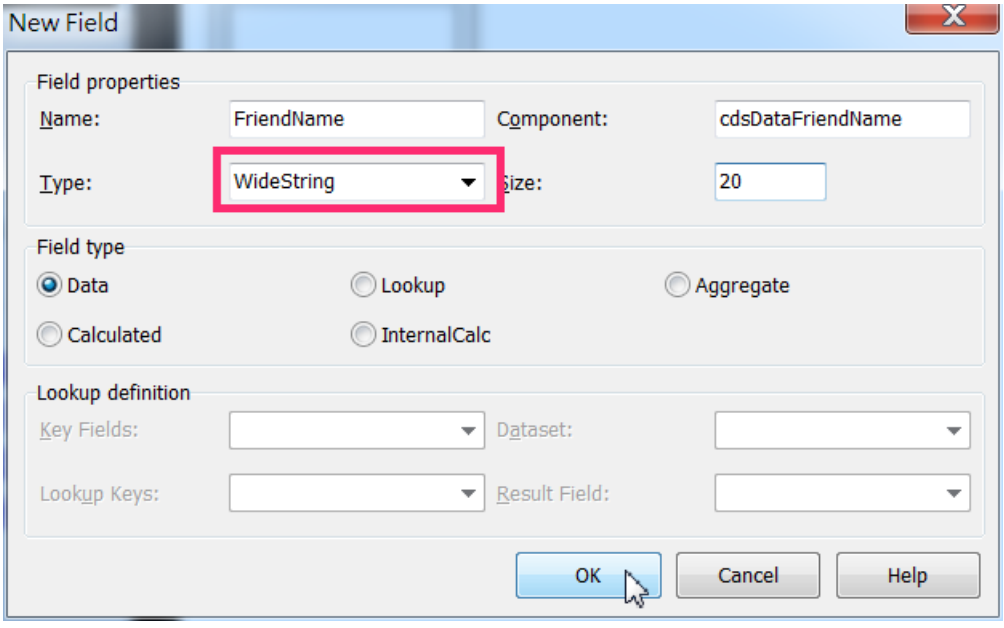
现在您就可以试着改变一些设定然后再次重新格式化您的源代码，看看改变这些设定之后有什么效果了。

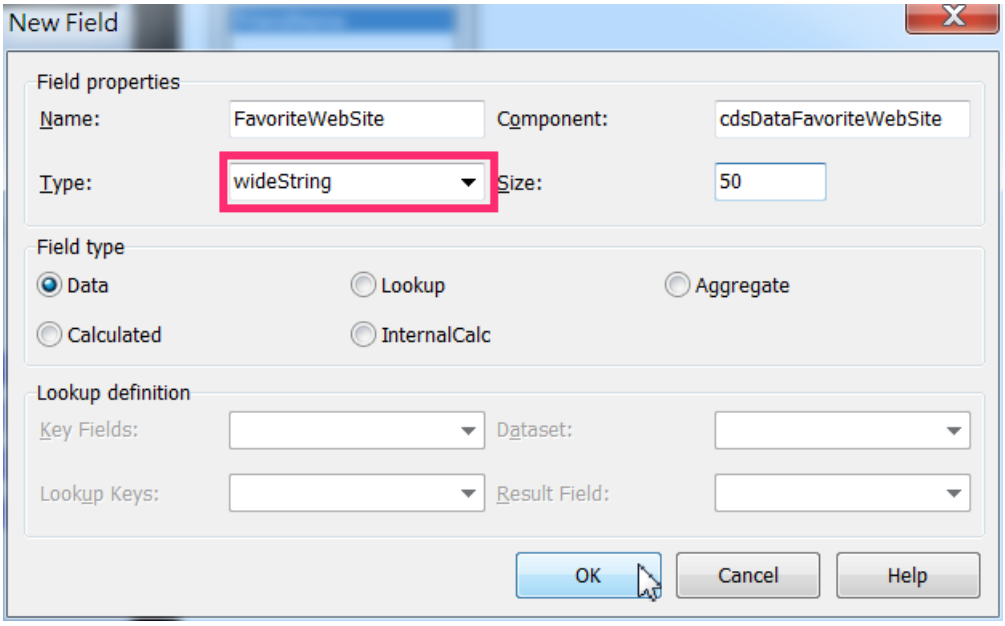
让我们为这个范例 iOS 加入一些更为复杂和有趣的功能，那就是开始加入存取数据的能力。如果您是使用其他 iOS 开发工具而需要为 iOS App 加入数据处理的功能的话，您需要花费许多的时间去撰写大量的程序代码，但使用 C++Builder for iOS 却非常的简单。

首先请在主窗体中加入 TClientDataSet 设定它的 Name 特性值为 cdsData，加入 TDataSource 组件，设定它的 Name 特性值为 dsData。接着点选主窗体中的 cdsData，点选鼠标右键从快捷菜单中选择『Fields Editor…』选项以启动字段编辑器，我们将使用它为 cdsData 加入两个字段。现在主窗体应该看起来类似如下：

在 cdsData 的字段编辑器启动之后，再于其中点选鼠标右键，从快捷菜单中选择『New field…』选项以加入字段对象，如下所示：

接着使用 New Field 对话盒加入如下的两个字段对象:





请注意，在 iOS 平台中如果要使用中交数据的话，那么必须使用 WideString 字段型态。

现在 cdsData 组件就拥有 FriendName 和 FavoriteWebSite 这两个字符串字段对象了，接着切换到编辑器，在程序代码中加入的两个方法 CreateGroupDataSet()和 FillGroupDataSet():


```
void TfmMainForm::CreateGroupDataSet()
{
  TIndexDef *pIndex;

  pIndex = cdsMyData->IndexDefs->AddIndexDef();
  pIndex->Fields = "FriendName";
  pIndex->Name = "idxFriendName";

  cdsMyData->CreateDataSet();
  cdsMyData->Active = true;
}
void TdmDemoApp::FillGroupDataSet()
{
  cdsMyData->Insert();
  cdsMyData->FieldByName("FriendName")->Value = “Jackson Wang”;
  cdsMyData->FieldByName("FavoriteWebSite")->Value =
“www.youtube.com”;
  cdsMyData->Post();
```

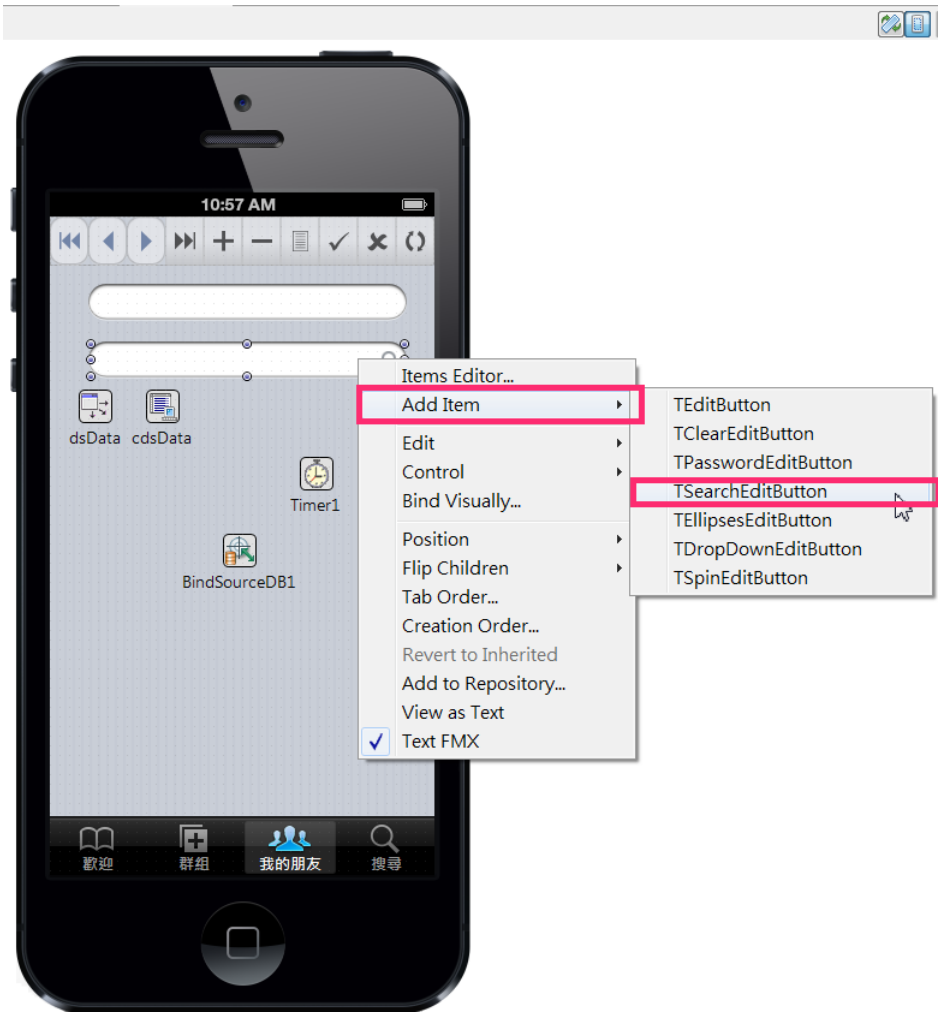
```
  cdsMyData->Insert();
  cdsMyData->FieldByName("FriendName")->Value = "Hua Lee";
  cdsMyData->FieldByName("FavoriteWebSite")->Value =
"www.embarcadero.com";
  cdsMyData->Post();

  cdsMyData->Insert();
  cdsMyData->FieldByName("FriendName")->Value = "DongHseng Cheng";
  cdsMyData->FieldByName("FavoriteWebSite")->Value = "www.msn.com.tw";
  cdsMyData->Post();

  cdsMyData->Insert();
  cdsMyData->FieldByName("FriendName")->Value = "YuHei Chu";
  cdsMyData->FieldByName("FavoriteWebSite")->Value =
"C++Builder.ktop.com.tw";
  cdsMyData->Post();

  cdsMyData->Insert();
  cdsMyData->FieldByName("FriendName")->Value = "老夫子";
  cdsMyData->FieldByName("FavoriteWebSite")->Value =
"www.google.com.tw";
  cdsMyData->Post();
}
```

CreateGroupDataSet()在 cdsData 中加入一个索引对象，再呼叫 CreateDataSet()方法建立 cdsData 中的数据集。而 FillGroupDataSet()则是在 cdsData 组件中新增一些数据。

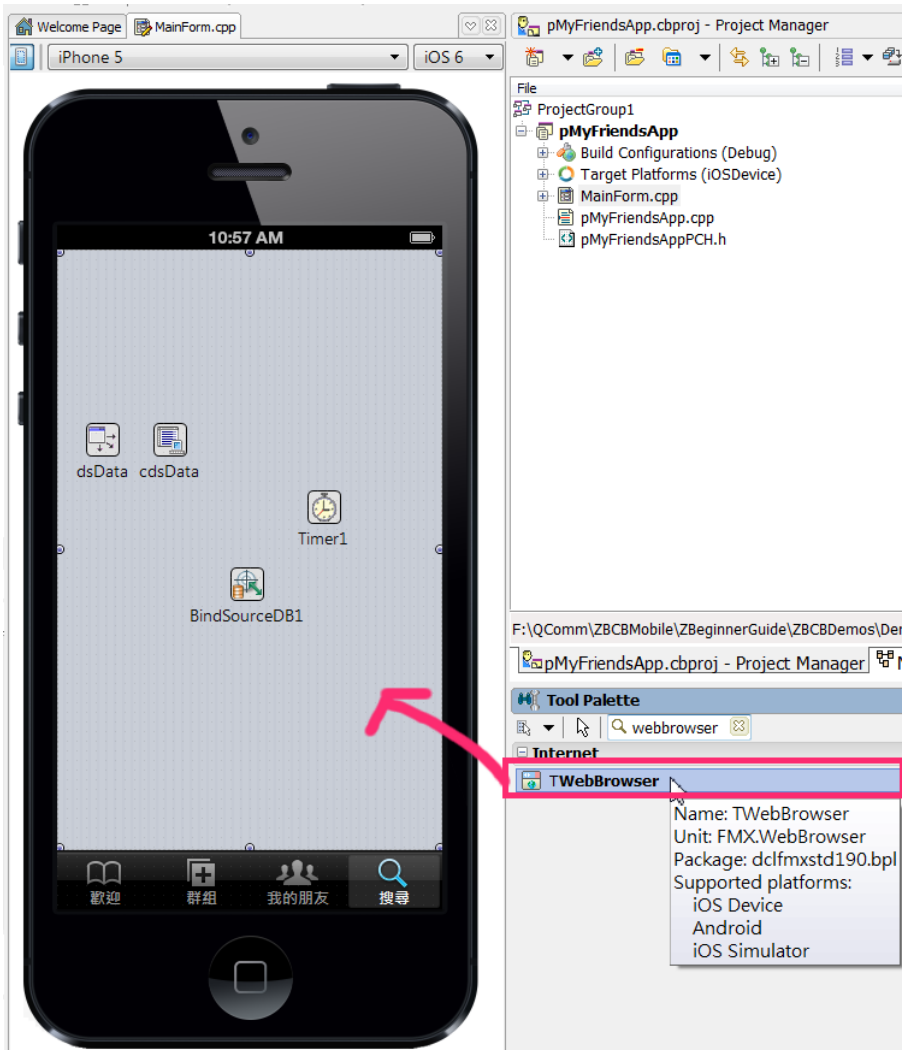
```
void __fastcall TfmMain::FormCreate(TObject *Sender)
{
  CreateGroupDataSet();
  FillGroupDataSet();
}
```

接着切换到主窗体中 TTabControl 组件的第 3 个页次，在其中加入 TBindNavigator 组件，设定它的 DataSource 特性值为 cdsData，再加入两个 TEdit 组件，如下所示:

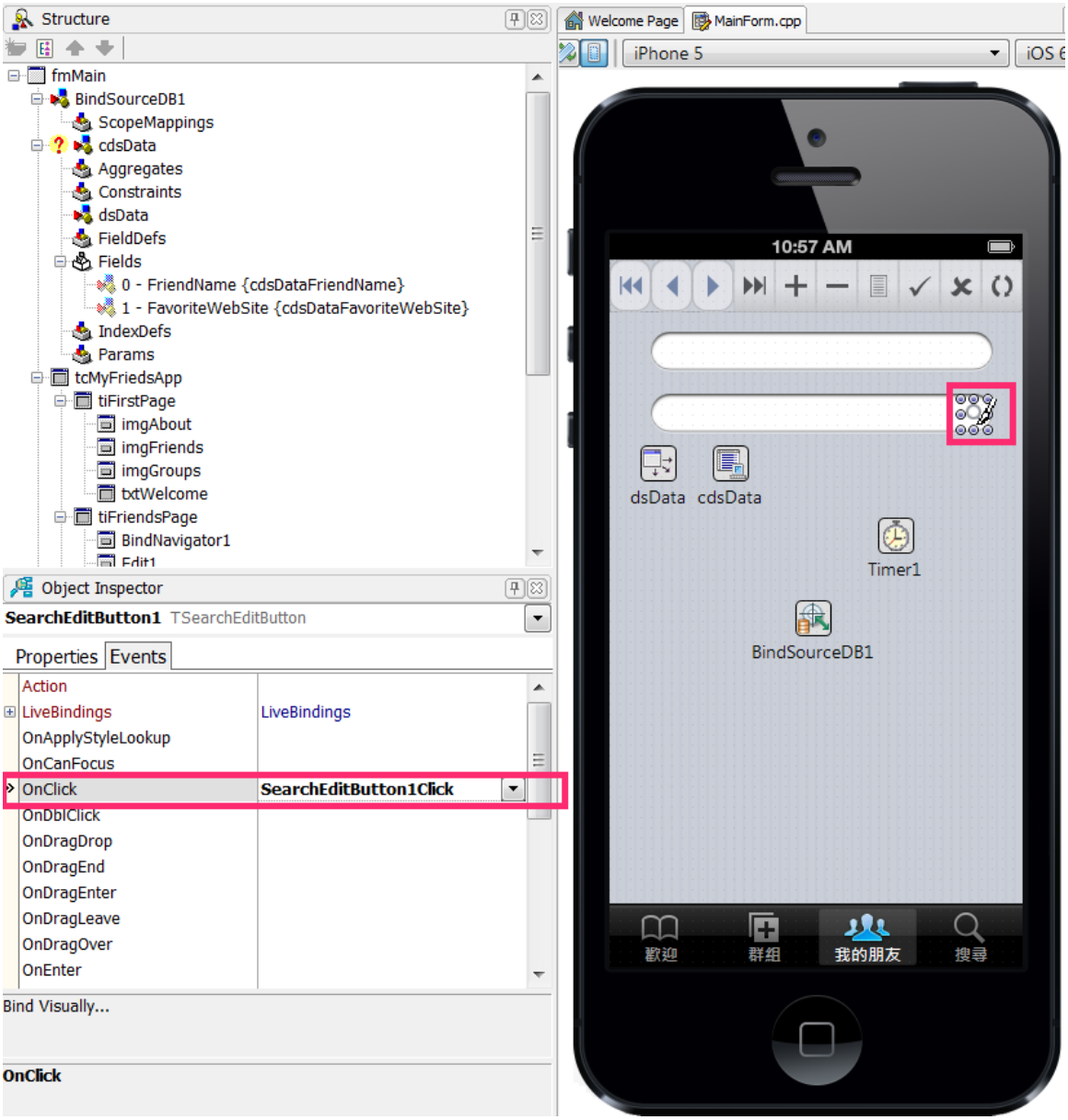
再让我们为刚才加入的第二个 TEdit 组件加入一个搜寻按钮，请点选第二个 TEdit 组件，再点选鼠标右键，从快捷菜单中选择『Add Item』选项，再从其中选择『TSearchEditButton』，如下所示:



切换到 TTabControl 组件的第 4 个页次，在工具盘中搜寻 TWebBrowser 组件再拖曳到第 4 个页次中，设定 TWebBrowser 组件的 Align 特性值为 alClient，如下所示:



最后再回到 TTabControl 组件的第 3 个页次为 TSearchEditButton 组件建立 OnClick 事件处理函式，如下所示：




```
void __fastcall TfmMain::SearchEditButton1Click(TObject *Sender)
{
  //
}
```

接下来我们就可以开始试着系结 cdsData 中的数据和主窗体中的可视化在一起了，但在这之前让我们再学习数个整合发展环境的技巧。

4-4 SyncEdit

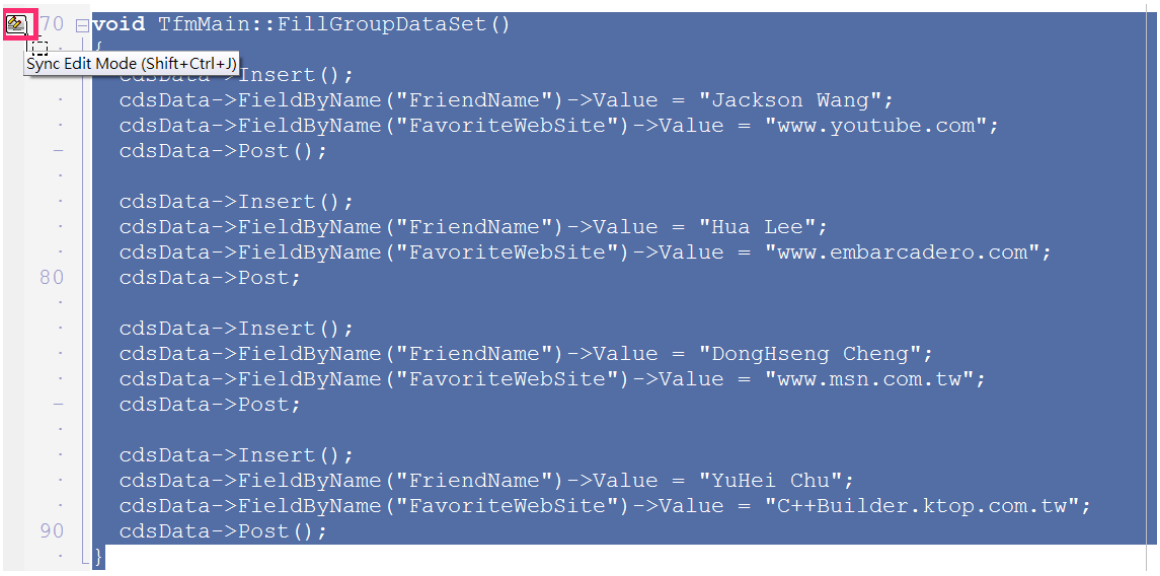
在继续改善程序代码之前，您需要学习一个非常有用的编辑器技巧，那就是所谓的 SyncEdit。

首先请回到主窗体把原先的 TClientDataSet 的名称 Name 特性值从 cdsData 改变成 cdsMyData。由于我们改变了组件名称因此程序代码中使用的 cdsData 对象变量名称也必须修改。但如果我们到 FillGroupDataSet 方法会发现其中使用了多次 cdsData，因此我们必须进行多次的修改，这实在非常的麻烦而且容易出错:

```
void TfmMain::FillGroupDataSet()
{
  cdsData->Insert();
  cdsData->FieldByName("FriendName")->Value = "Jackson Wang";
  cdsData->FieldByName("FavoriteWebSite")->Value = "www.youtube.com";
  cdsData->Post();
  ...
}
```

这个时候就是使用 SyncEdit 的好时机，现在就让我们使用 SyncEdit 来修改 cdsData 为 cdsMyData。

首先请如下图使用鼠标或是键盘选择整个使用 cdsData 对象变量名称的程序代码部份(使用鼠标或是使用 Shift+↓键)，请注意此时在编辑器最左方会出现一个类似双铅笔的图像，如下所示:



选择程序代码并且启动 SyncEdit 功能

现在请使用鼠标双击编辑器中图像，此时编辑器会立刻把整个使用 cdsData 对象变量名称的程序代码部份中所有的变量标示出来，由于 cdsData

是第 1 个变量，因此它是以方框标示，此时所有的 cdsData 变量中的第 1 个 cdsData 更以方框反白标示，如下图所示:

```
70  void TfmMain::FillGroupDataSet()
    {
72    cdsData->Insert();
      cdsData->FieldByName("FriendName")->Value = "Jackson Wang";
      cdsData->FieldByName("FavoriteWebSite")->Value = "www.youtube.com";
      cdsData->Post();

      cdsData->Insert();
      cdsData->FieldByName("FriendName")->Value = "Hua Lee";
      cdsData->FieldByName("FavoriteWebSite")->Value = "www.embarcadero.com";
80    cdsData->Post;

      cdsData->Insert();
      cdsData->FieldByName("FriendName")->Value = "DongHseng Cheng";
      cdsData->FieldByName("FavoriteWebSite")->Value = "www.msn.com.tw";
      cdsData->Post;

      cdsData->Insert();
      cdsData->FieldByName("FriendName")->Value = "YuHei Chu";
      cdsData->FieldByName("FavoriteWebSite")->Value = "C++Builder.ktop.com.tw";
90    cdsData->Post();
    }
```

現在请您直接在编辑器中输入 cdsMyData，请注意这时编辑器中所有变量 cdsData 都立刻修改为 cdsMyData，如下所示:

```
98    cdsMyData.CreateDataSet;
      cdsMyData.Active := True;
100 end;

    procedure TfmMainForm.FillGroupDataSet;
    begin
      cdsMyData.Insert;
      cdsMyData.FieldByName('FriendName').AsString := 'Jackson Wang';
      cdsMyData.FieldByName('FavoriteWebSite').AsString := 'www.youtube.com';
      cdsMyData.Post;

      cdsMyData.Insert;
110   cdsMyData.FieldByName('FriendName').AsString := 'Hua Lee';
      cdsMyData.FieldByName('FavoriteWebSite').AsString := 'www.embarcadero.com';
      cdsMyData.Post;

      cdsMyData.Insert;
      cdsMyData.FieldByName('FriendName').AsString := 'DongHseng Cheng';
      cdsMyData.FieldByName('FavoriteWebSite').AsString := 'www.msn.com.tw';
      cdsMyData.Post;

      cdsMyData.Insert;
120   cdsMyData.FieldByName('FriendName').AsString := 'YuHei Chu';
      cdsMyData.FieldByName('FavoriteWebSite').AsString := 'Delphi.ktop.com.tw';
      cdsMyData.Post;
    end;
```

如果此时您不断的按下『Tab』键，整个使用 cdsData 对象变量名称的程序代码部份中会逐一的以方框反白标示每一个可修改的变量，类别名称或是方法名称，例如您第 1 次按下『Tab』键方框反白标示就会停驻在变量 Insert 上，再按下一次就会停驻在类别 FieldByName 上，您就可以像刚才修改 cdsData 为 cdsMyData 一样改变 Insert 或是 FieldByName 的名称。

最后要关闭 SyncEdit 功能，您只需要再次使用鼠标点选图像  ，或是直接按下 ESC 键即可。

4-5 待办清单(To-Do List)

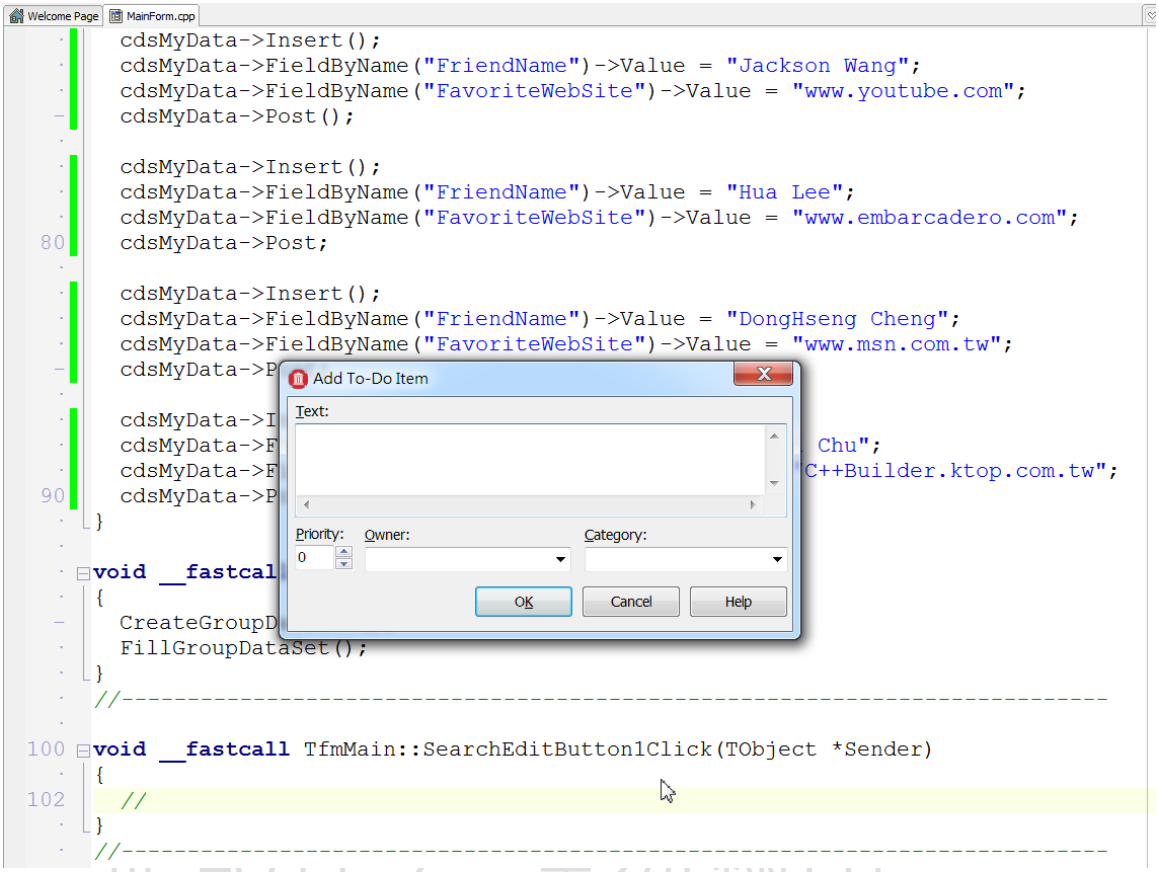
在撰写程序代码时，您可以先在程序代码中列出待办清单，接着一一的撰写程序代码实现这些待办清单，您也可以追踪，管理程序代码中的待办清单以便了解您是否完成了所有的待办清单，以免遗漏应该完成的功能，这在撰写大量的功能和程序代码时是非常实用的功能。

例如现在 TSearchEditButton 组件的 OnClick 事件处理函式还没有实作，但我们可以在编辑器加入这些待办清单，并且追踪这些待办清单以便我们最终能够完成这些功能。

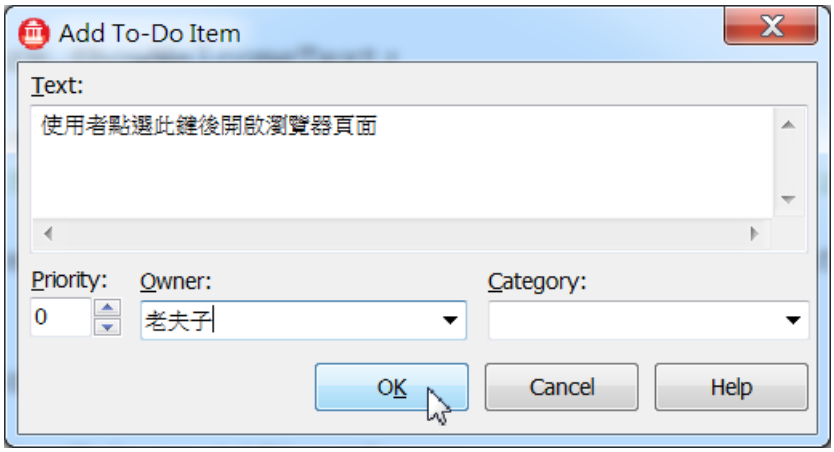
要在编辑器中加入待办清单，您可以在程序代码中适当的地方同时按下 Shift+Ctrl+T，或是在编辑器中点选鼠标右键，在快捷菜单中选择『Add To-Do item...』选项。例如 TSearchEditButton 组件建立 OnClick 事件处理函式尚未实作，因此让我们移动鼠标到 SearchEditButton1Click 程序的第 1 行程序代码之上，同时按下 Shift+Ctrl+T 准备加入待办清单，此时 IDE 就会显示一个 Add To-Do Item 对话盒如下，请您输入待办列表的详细信息，对话盒中字段的意义说明如下:

字段	说明
Text	待办列表说明文字
Priority	待办清单优先次序
Owner	待办清单负责人
Category	待办清单归纳分类

例如下图就是在 SearchEditButton1Click 程序中启动待办列表功能:



接着让我们在 Add To-Do Item 对话盒中输入如下的信息:

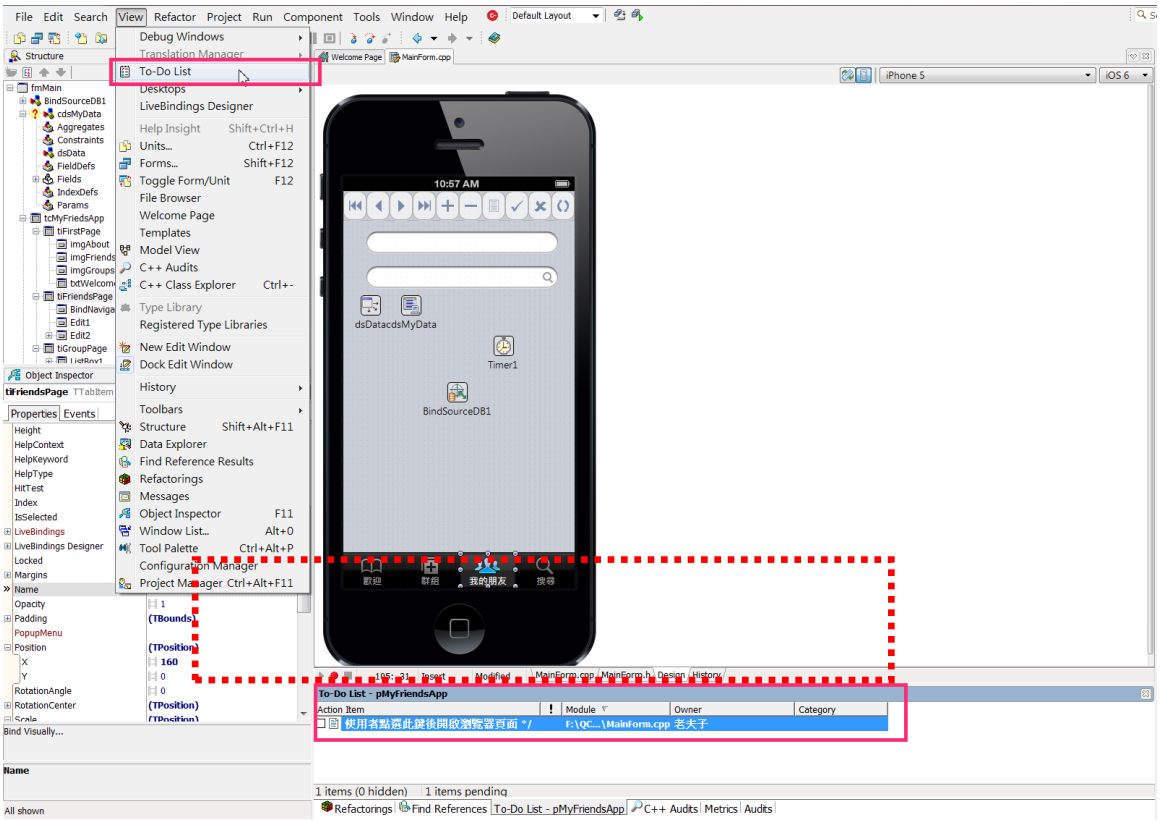


当您点选 Add To-Do Item 对话盒中的 OK 按钮之后，您就可以在程序代码中看到 IDE 在您的程序代码中加入了如下的待办清单注释:

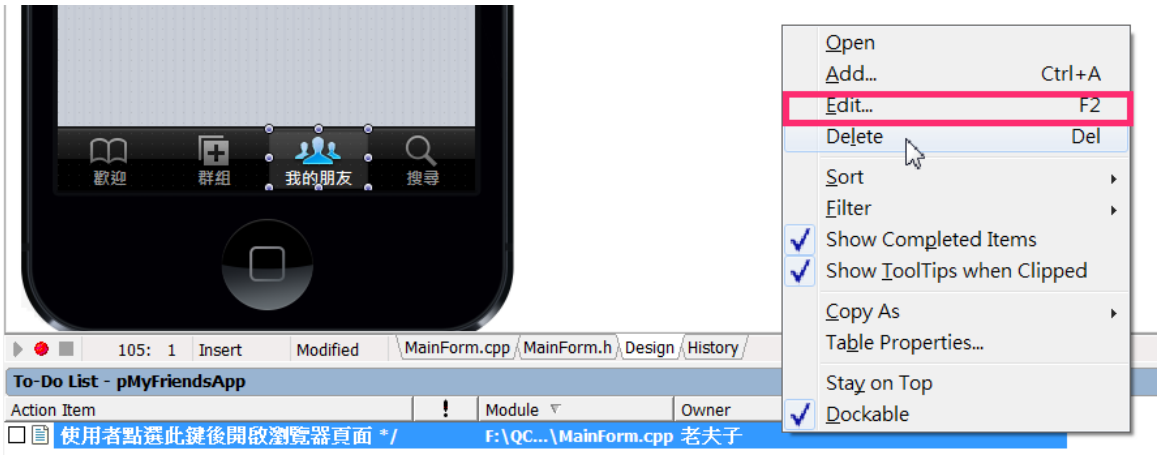
```
void __fastcall TfmMain::SearchEditButton1Click(TObject *Sender)
{
  ///* TODO -o老夫子 : 使用者点选此键后开启浏览器页面 */
```

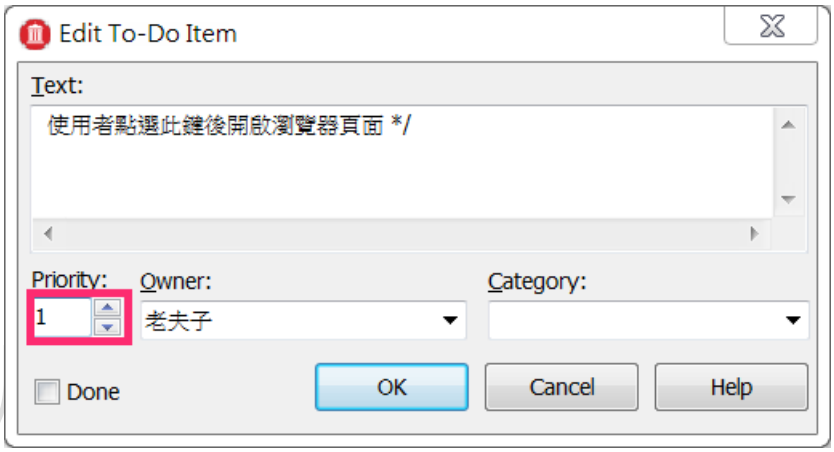
```
}
```

完成了之后您可以点选 IDE 上方的 View|To-Do List 菜单，您就可以看到类似如下的画面，IDE 可以显示所有您的待办事项，当您双击其中任何的待办事项时，编辑器就会移动到您的程序代码到对应的待办事项和程序代码处，如下所示：



当然您也可以修改待办事项，现在请使用鼠标点选 IDE 下方的『**使用者点选此键后开启浏览器页面**』待办事项，点选鼠标右键，从快捷菜单中选择『Edit...』选项，或是直接按下 F2 键，那么 Edit To-Do Item 对话盒就会显示，您就可以进行修改，例如我们增加了『待办清单优先次序』和『待办列表归纳分类』信息，如下图所示。

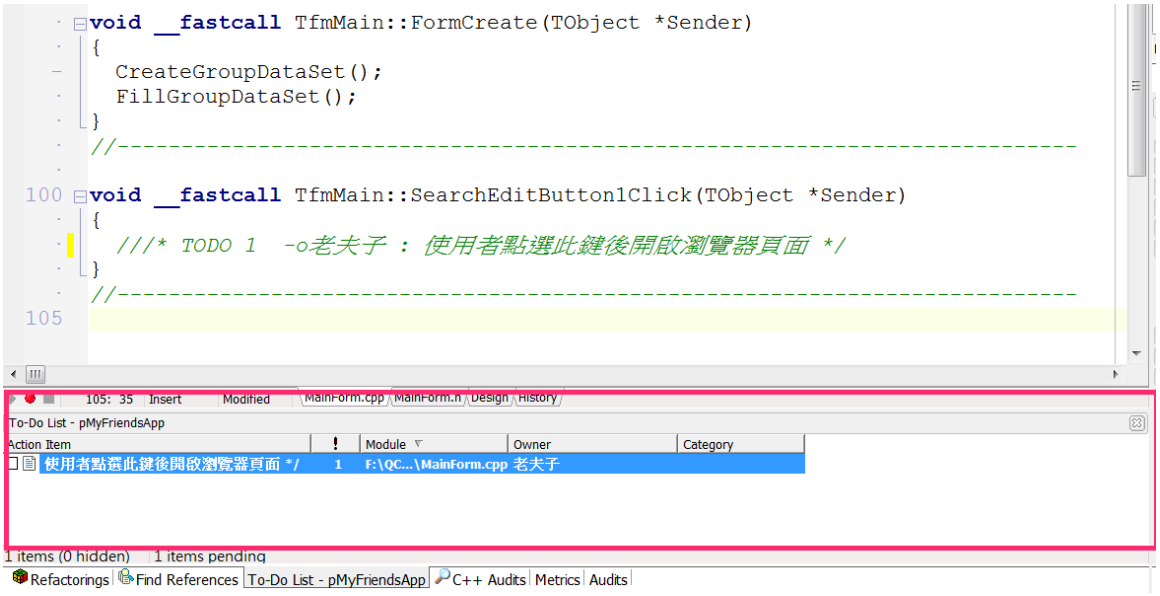




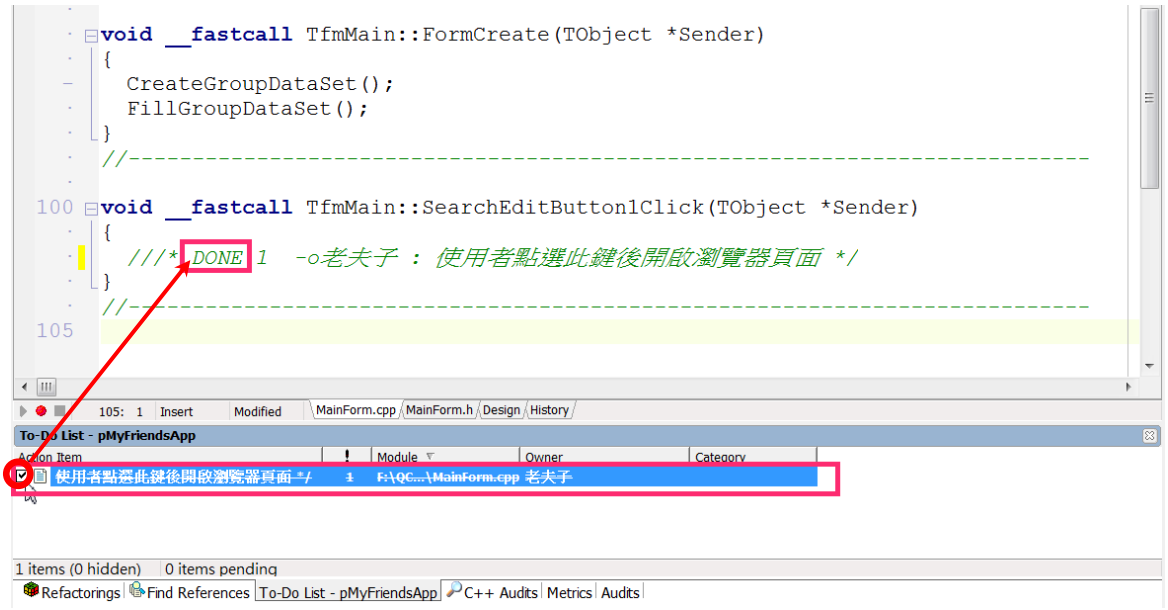
点选 OK 按钮之后程序代码就会被修改如下：

```
void __fastcall TfmMain::SearchEditButton1Click(TObject *Sender)
{
  ///* TODO 1  -o 老夫子 : 使用者点选此键后开启浏览器页面 */
}
```

如果您点选 View|To-Do List 菜单就会在整合发展环境中看到 To-Do List 窗口，其中会显示所有程序代码中的待办事项列表：



当然，如果我们完成了某项待办清单，那么我们可以更正待办事项为『完成』的状态，我们只需要勾选待办事项最左方的勾选盒，那么 IDE 就会修改程序代码中的待办事项为 DONE，如下图所示:



4-6 程序区块批注

在 C++Builder for iOS 程序语言中，您可以使用『//』批注单行程序代码的意义，或是使用『/*...*/』来批注多行程序代码的意义。在 IDE 中如果您想批注多行的程序代码，您可以使用鼠标选择想批注的程序代码，然后同时按下『Ctrl+/』，那么 IDE 就会自动帮您批注这些程序代码，如下所示:

```
void TfmMain::InsertData(const String sName, const String sWebSite)
{
//  cdsMyData->Insert();
//  cdsMyData->FieldByName("FriendName")->Value = sName;
//  cdsMyData->FieldByName("FavoriteWebSite")->Value = sWebSite;
//  cdsMyData->Post();
}
```

如果您想取消批注的程序代码，那么同样的您只要选择已经批注程序代码，再同时按下『Ctrl+/』就可以取消选择的程序代码的批注了。

4-7 程序代码浏览

当程序代码中的事件处理函式和方法愈来愈多时，您可能需要快速在不同的方法程序代码中浏览或是移动，您可以双击 IDE 左上方的项目树状架构中的方法节点，那么编辑器会立刻移动到此方法，如下所示:

或是您可以使用下面的快捷键组合在不同的方法中快速移动:

键盘快捷键	效果
按 CTRL+ALT+向上键	移至目前方法的顶端
按 CTRL+ALT+向下键	移至下一个方法
按 CTRL+ALT+HOME	移至档案中的第一个方法
按 CTRL+ALT+END	移至档案中的最后一个方法
按 CTRL+ALT	并卷动鼠标滚轮，那么编辑器就会以方法为单位移动

现在您就可以在开启范例项目的程序代码编辑器中试着使用这些快捷键。

4-8 设定和使用书签

当您在撰写大量的程序代码时，您可能需要先暂停目前的程序代码而移动到另外的程序代码处撰写其他的程序代码，之后再回到目前的程序代码继续撰写。在这种使用需求下您可以利用 IDE 的书签功能。

IDE 提供了最多 10 个书签可让您在程序代码中标注，一旦您在程序代码中设定了书签，就可以藉由快捷键立刻在不同的书签程序代码标注处移动。要在 IDE 中设定书签，您可以使用 Ctrl+Shift+0，Ctrl+Shift+1 一直到 Ctrl+Shift+9 最多设定 10 个书签。

> 在繁体中文 OS 的环境下，您可能无法使用 Ctrl+Shift+0，Ctrl+Shift+1 来设定书签。

> 您也可以使用 Ctrl+k+0 到 Ctrl+k+9 来设定书签，使用 Ctrl+k+0 和 Ctrl+k+1 就可以避免无法使用 Ctrl+Shift+0，Ctrl+Shift+1 设定的问题。

现在请您移动光标到 FillGroupDataSet()方法的第 1 行程序代码处，接着同时按下 Ctrl+Shift+2，就可以如下图看到 IDE 在编辑器最左边出现了一个『2』的标志，这就代表您在这行程序代码设定了一个书签：

```
void TfmMain::FillGroupDataSet()
{
  InsertData("Jackson Wang", "www.youtube.com");
  InsertData("Hua Lee", "www.embarcadero.com");
  InsertData("DongHseng Cheng", "www.msn.com.tw");
  InsertData("YuHei Chu", "Delphi.ktop.com.tw");
  InsertData("老夫子", "www.google.com.tw");
  SearchPerson(PersonName);
}
```

接着再请您移动光标到 InsertData()方法的第 1 行程序代码处，接着同时按下 Ctrl+Shift+3，就可以如下图看到 IDE 在编辑器最左边出现了一个『3』的标志，这就代表您在这行程序代码设定了一个书签：

```
70  void TfmMain::InsertData(const String sName, const String sWebSite)
    {
72    cdsMyData->Insert();
      cdsMyData->FieldByName("FriendName")->Value = sName;
      cdsMyData->FieldByName("FavoriteWebSite")->Value = sWebSite;
      cdsMyData->Post();
    }
```

现在您可以藉由同时按下 Ctrl+2 立刻把光标移动回 FillGroupDataSet()方法的第 1 行程序代码处，如果您同时按下 Ctrl+3，那么又可以立刻把光标移动回 InsertData()方法的第 1 行程序代码处。

藉由使用 IDE 的书签功能，可以方便的让您在不同的工作程序代码处快速的移动和撰写。

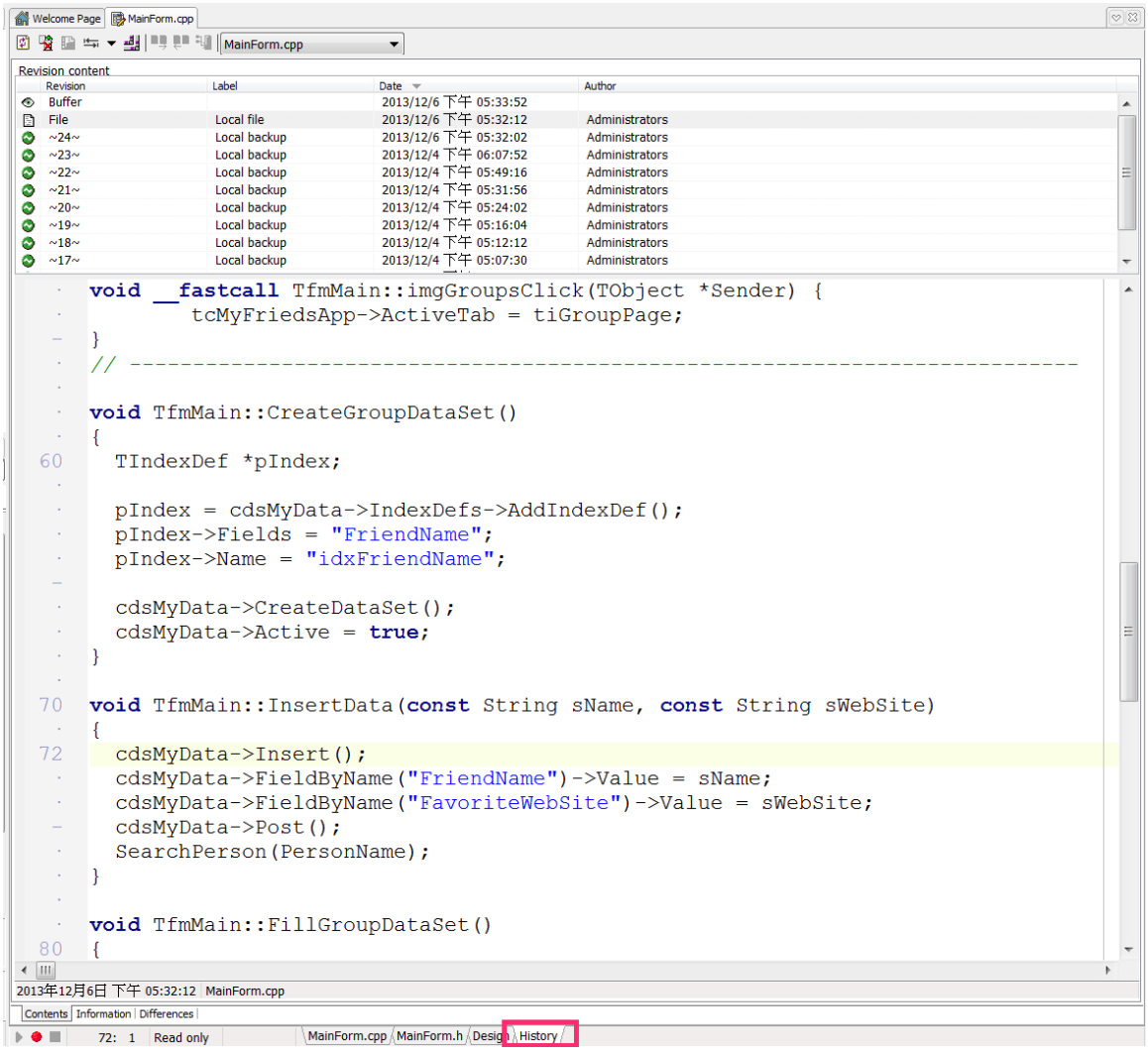
4-9 历程管理员(History)

请您仔细看看上图项目目录中所有的档案，您会看到其中有一个 History 子目录，其中的文件名比较特别，都是以『XXX.XXX.~数字~』样例为档案，如果您再仔细的观察会发现其中的 XXX.XXX 都是您项目中的档案。

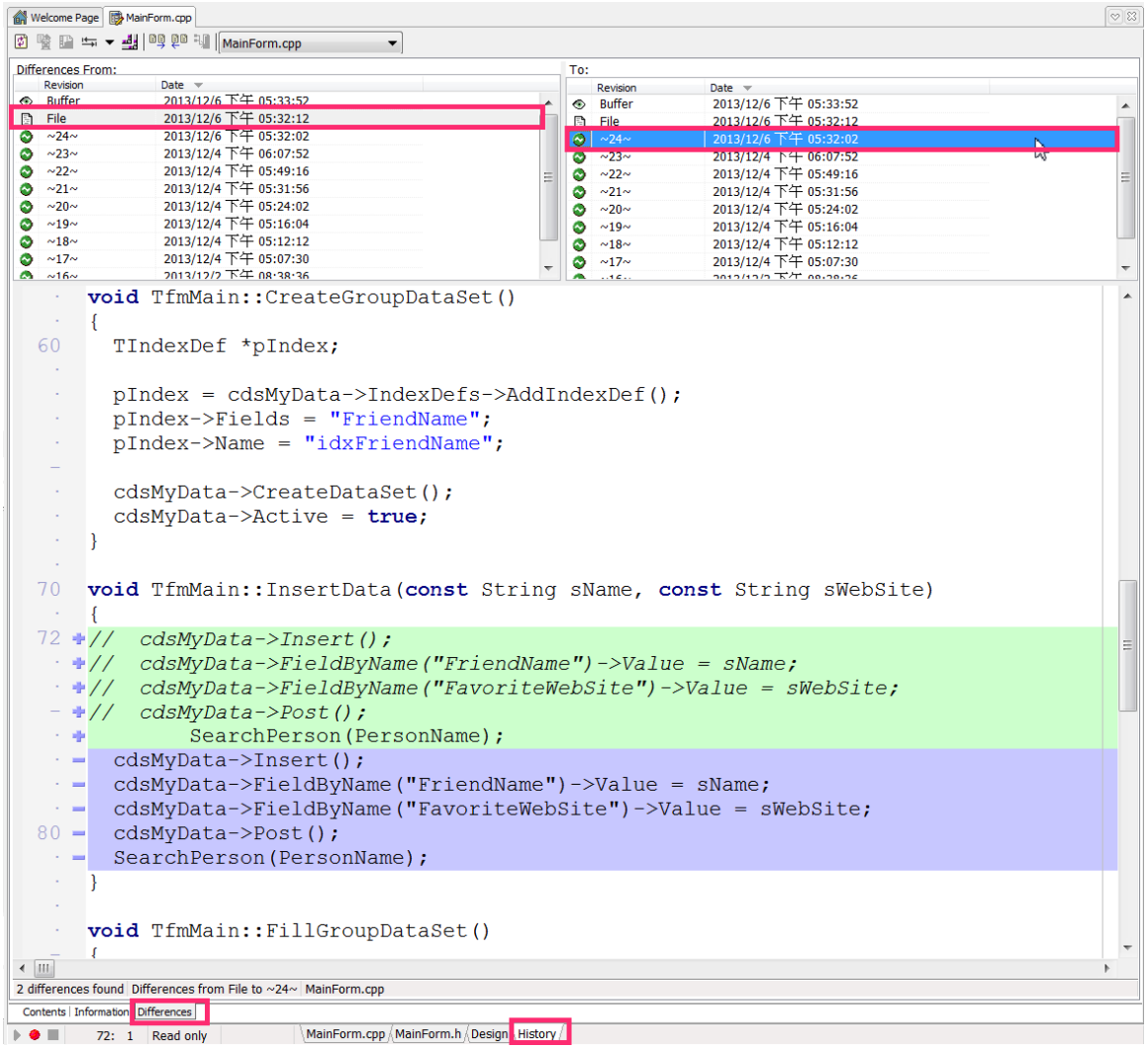
事实上当您在 IDE 中对项目中的档案进行异动时并且储存之后，IDE 便会在这个 History 子目录中储存一份版本，因此这个 History 子目录中所有的档案便是您对项目修改的历程，也就是说 IDE 会在这个 History 子目录提供一些类似版本控制软件提供的功能。

> 如果您使用 C++Builder 开发真正的项目时，强烈的建议您应该使用真正的版本控制软件来管理您的项目和原始程序档案。

IDE 的这个功能称为『历程管理员』，您可以在 IDE 中启动『历程管理员』的管理和检视接口，现在请您回到项目主窗体的程序代码页次，在编辑器下方您会看到一个『History』页次，请双击这个页次就会看到类似如下的画面，每一个画面上方的档案就是您储存(或是 IDE 自动储存)的版本档案，下方的窗口则是您点选上方版本的档案内容:



您也可以比较不同版本之间的差异，请点选『History』页次左下方的『Differences』子页次，就可以看到类似如下的画面:



在上图中您可以先选择左上方窗口中的版本档案，再点选右上方中要比较差异的版本，例如上图就是比较目前编辑器中的版本(Buffer)以及上一个已经储存版本(~24~)的差异，那么在下方的窗口中就可以看到 IDE 显示两个版本的差异，其中程序代码最左方如果出现『+』符号就代表是新增的程序代码，而『-』符号则代表是被删除的程序代码。

此外在上图左上方，右上方窗口中的档案前都有一个特定的图像，不同的图像代表不同的意义，下面的表格说明了不同图像的意义：

图像	说明
	最近一次储存的版本
	备份的档案版本
	目前存在于缓冲存储器的版本，包含了尚未储存的异动程序代码
	储存在版本控制系统中的版本

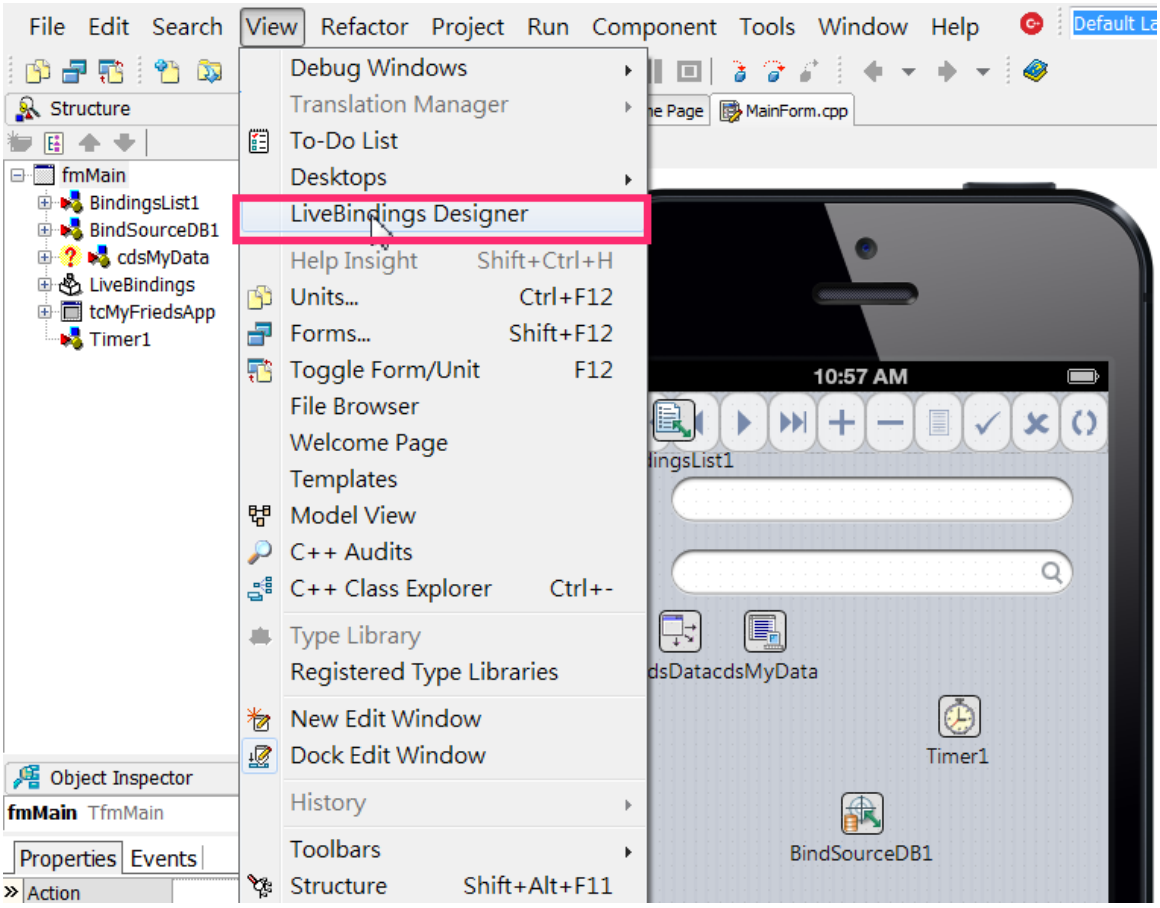
	从控制系统中签出(Check Out)的版本

现在我们就可以继续开发这个范例 App 了，首先让我们使用 LiveBinding 功能系结数据，最后再实作 SearchEditButton1Click 事件处理函式。

4-10 使用 LiveBinding 系结数据和可视化组件

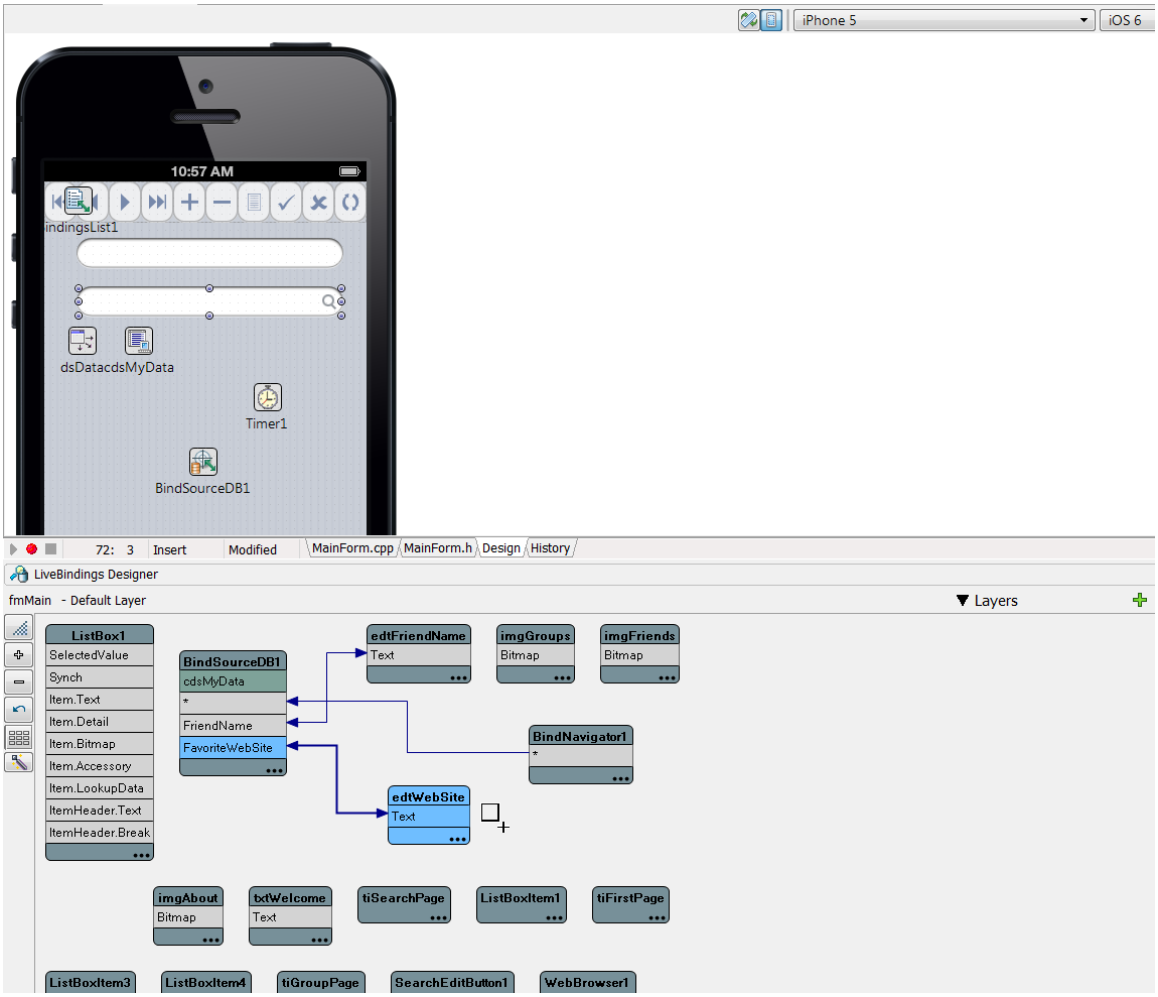
C++Builder for iOS 提供了非常可视化，简单又强大的 LiveBinding 功能让 iOS 开发人员能够轻易的存取数据并且把数据显示(系结)在可视化组件中。

在前面的 FillGroupDataSet() 方法中我们已经建立了一个 TClientDataSet 并且在其中新增了几笔数据，现在让我们在主窗体的 TTabControl 的第 3 个页次显示这些数据。点选 TTabControl 的第 3 个页次，再点选整合发展环境主菜单的 View | LiveBindings Designer 菜单以开启可视化实时数据系结设计家，如下所示:



可视化实时数据系结设计家中您可以使用拖曳拉线的方式系结数据和组件。例如现在我们想把 TClientDataSet 中的 FriendName 字段显示在 TTabControl 的第 3 个页次中的第一个 TEdit 组件，把 FavoriteWebSite 字段

显示在第二个 TEdit 组件中。因此请在可视化实时数据系结设计家中先点选 BindSourceDB1 中的 FriendName 字段再持续按着鼠标左键拖曳到 edtFriendName 的 Text 特性上再放开鼠标左键，此时在这 2 者之间就会出现一个双向箭头的线条，这就代表现在我们已经系结了 FriendName 字段的数值和 edtFriendName->Text，也就是说 FriendName 字段的数值会自动显示在 edtFriendName->Text 中。同样的，先点选 BindSourceDB1 中的 FavoriteWebSite 字段再持续按着鼠标左键拖曳到 edtWebSite 的 Text 特性上再放开鼠标左键，此时 2 者之间也会出现一个双向箭头的线条，如下所示:



现在如果我们编译并且分发此时的范例 iOS App 到 iOS 的仿真器中的话，点选第 3 个页次就可以看到类似如下的执行结果，数据果然自动显示在 2 个 TEdit 组件中，如果点选上方的 Navigator 组件就可以在不同的数据中浏览和移动了，开发能够存取数据的 iOS App 就是这么简单，太酷了。

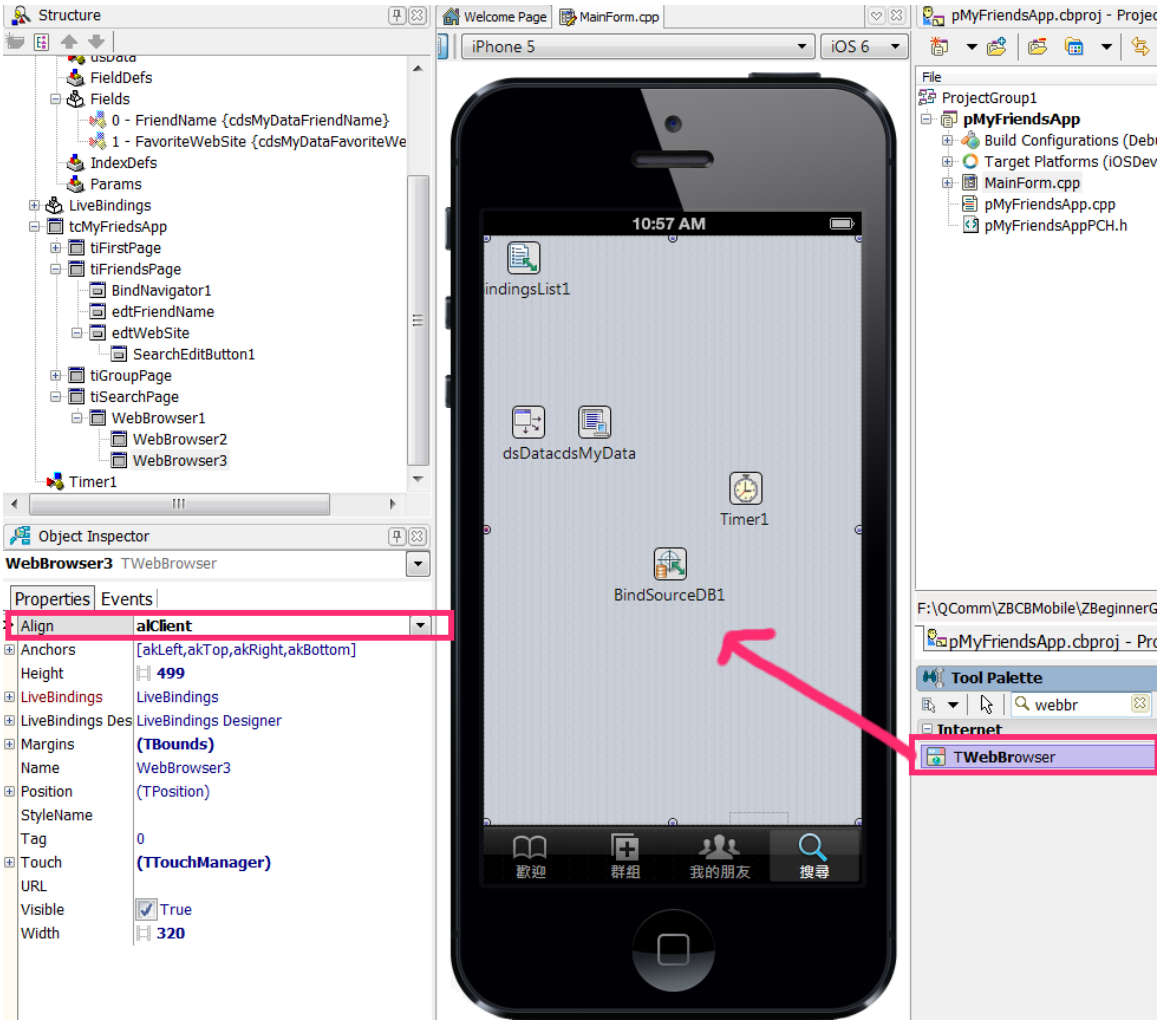
注意，由于现在的范例 iOS App 使用了 C++Builder for iOS 的 DataSnap 功能，因此您无法只编译它就执行，您需要分发 DataSnap 相关的档案和分享函式库。在稍后的章节中会说明如何使用整合发展环境分发需要额外功能和档案的 iOS App。

4-11 实作 SearchEditButton1Click 事件处理函式

现在我们可以完成这个范例 iOS App 的最后一个功能了，那就是当使用者点选了第 3 个页次中位于 FavoriteWebSite 旁的搜寻按钮时就开启第 4 个页次并且使用浏览器带领使用者到 FavoriteWebSite 字段数值指定的网站。

实作这个功能非常的简单，C++Builder for iOS 提供了 TWebBrowser 组件，只要我们使用 TWebBrowser 组件并且设定它的 URL 特性值，再呼叫它的 Navigate 方法即可。因此点选主窗体中的『搜寻』页次并且如下图般把

TWebBrowser 组件拖曳到『搜寻』页次中并且在对象查看器中设定 TWebBrowser 组件的 Align 特性值为 alClient:



接着请在 SearchEditButton1 的事件处理函式中撰写如下的程序代码:

```
void __fastcall TfmMain::SearchEditButton1Click(TObject *Sender)
{
  ///* DONE 1  -o 老夫子 : 使用者点选此键后开启浏览器页面 */
  WebBrowser1->URL = edtWebSite->Text;
  tcMyFriedsApp->ActiveTab = tiSearchPage;
  WebBrowser1->Navigate();
}
```

再次编译并且执行范例 iOS App 就可以在 iOS 仿真器中点选第 3 个页次中位于 FavoriteWebSite 旁的搜寻按钮时看到类似如下的结果，范例 iOS App 果然可以浏览到 FavoriteWebSite 字段指定的网站，Cool。

4-12 开启多执行程序编译功能

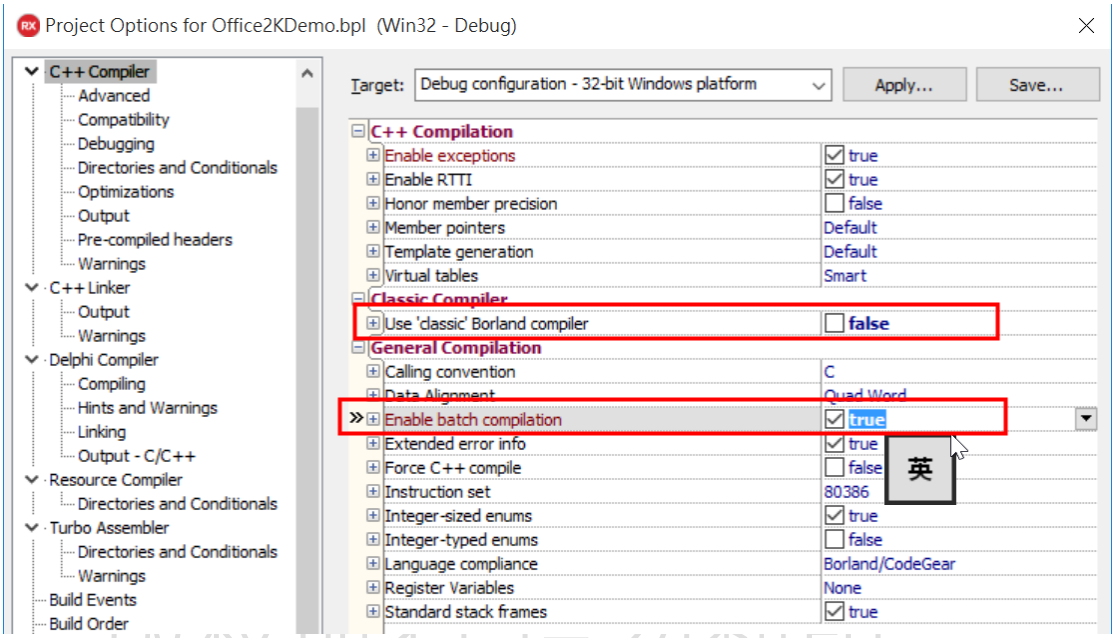
最重要的功能之一就是为 Win32 开发了基于 LLVM 的全新 Clang 编译程序，新的 Clang Win32 编译程序除了如同 Win64 编译程序支持 C++11 标准外，C++Builder 的 IDE 也为 Clang 编译程序开启了多执行程序编译功能，如此一来可让 C/C++程序员在使用多核心的机器时能够同时使用每一个闲置核心来编译项目中每一份不同的 C++程序单元，这样可以大幅加快项目的编译速度。

要开启 C/C++的多线程编译功能，程序员必须：

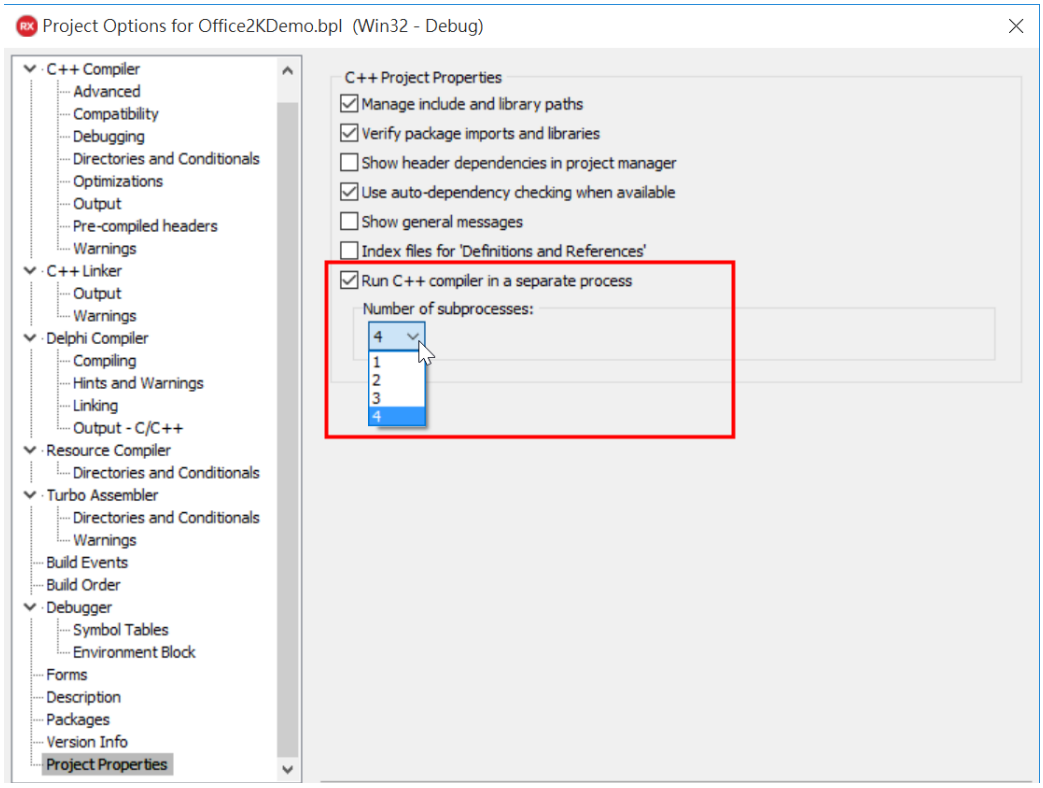
1. 使用 Clang 编译程序而不是使用旧的 Borland 编译程序
2. 开启批次编译功能
3. 开启多执行程序编译功能

第一步是使用 Clang 编译程序，请在项目管理员中使用鼠标右击项目，点选突显示选单中的 Options 选项。在 C++ Compiler 选项中取消”'Use 'classsic' Borland compiler” 选项。

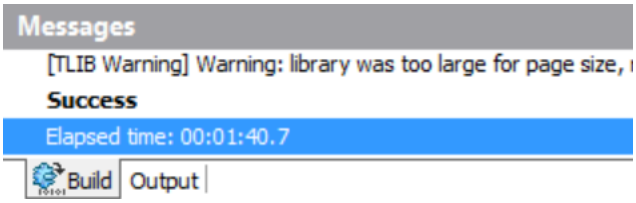
第二步是勾选其中的”Enable batch compilation” 选项：



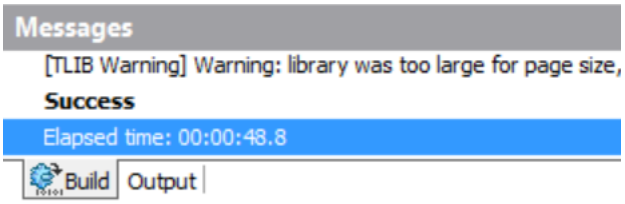
最后一步则是在对话盒左方最下面的 Project Properties 选项中勾选其中的”Run c++ compiler in a separate process” 选项，此时其下方的”Number of subprocesses”下拉盒就会启动，您就可以根据您机器可以的 CPU 核心数来设定要启动多少个程序一起编译项目，例如笔者的虚拟机分配了 4 个核心，因此笔者选择 4，如下所示：



一旦启动了多执行程序编译功能，当您编译您的项目时就会查觉到项目编译的速度快了许多，例下面是笔者一个包含 8 个 C++窗体的项目，在正常情形下需要 1 分 40 秒左右在笔者的虚拟机中完全编译：



但在开启多执行程序编译功能后，整个项目完全编译的时间下降到只需要 48 秒左右：



4-13 Visual Assist 功能

C++Builder 在 12 的版本终于加入期待以久的 Visual Assist(VA)功能。VA 功能其实来自多年前 Idera 并购的 Whole Tomato， Whole Tomato 是 Visual C++非常受欢迎的插件软件，其功能是可大幅加强程序员撰写程序代码的生产力，因此 C++Builder 的 VA 功能也就是强化程序员撰写程序代码效率的功能。

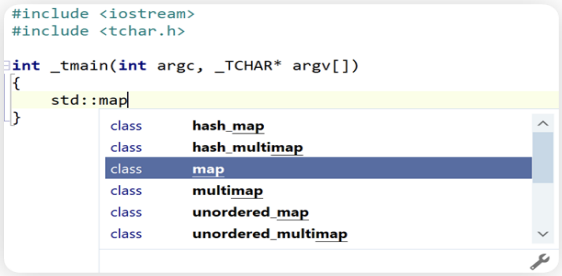
12 版的 VA 主要包含 3 个主要功能:

- Code Insight
- Refatoring
- Navigation

Code Insight

长久以来 C++Builder 的 Code Insight 功能一直无法跟上 Delphi Code Insight 的功能，其中最主要的原因就是因为编译程序的不同。C++Builder 和 Delphi 的 Code Insight 都是基于编译程序提供必要的显示信息，但由于 C++是 3-Pass 编译程序，而 Delphi 则是 1-Pass 编译程序，因此 C++Builder 的 Code Insight 在显示速度，信息更新都差很多。从以前的 Borland 传统 C/C++编译程序一直到现在的 Clang 编译程序都有一样的问题，即使 C++Builder 的 LSP 功能的确为 Code Insight 带来了改善，但是 C++Builder 的 Code Insight 在使用经验上仍然无法提供像 Delphi 的 Code Insight 一样的速度和正确性。

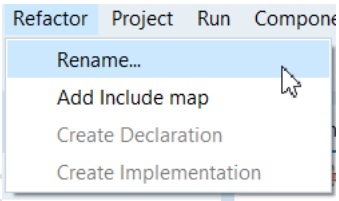
但是 12 版的 VA 的 Code Insight 功能带来了改变，VA 的 Code Insight 并不是基于编译程序，VA 使用了自己的模糊逻辑运算法则来提供 Code Insight 的信息，因此 12 版的 Code Insight 速度已经能够和 Delphi 一样快了，而且由于模糊逻辑运算法则，因此 VA 甚至能根据程序员撰写的程序代码来预测 Code Insight 的信息，并且能够根据程序员撰写的程序代码来建议要加入的表头档(.h/.hpp)。例如在下面的图形中显示出 VA 的 Code Insight 在尚未加入必须的表头档之前就能够提供 Code Insight 显示的信息：



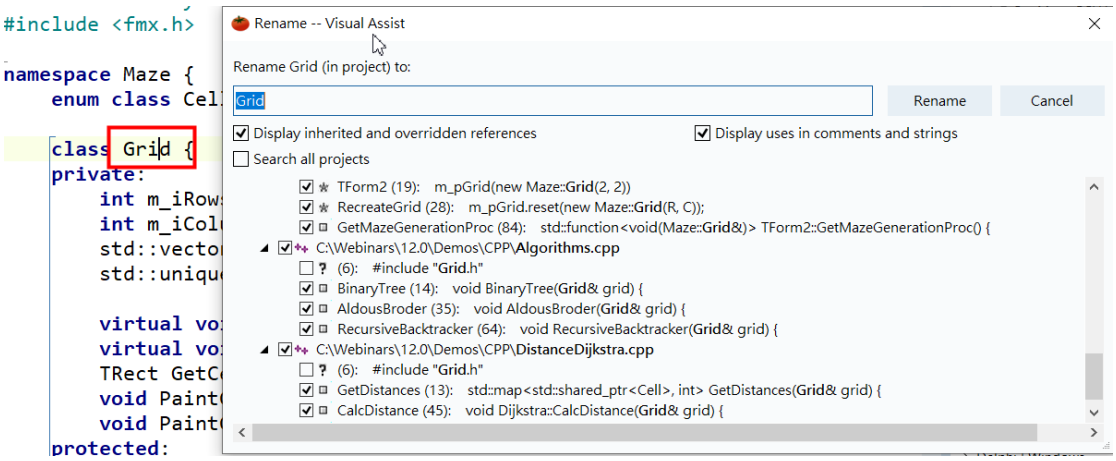
这个能力是以前的 Code Insight 无法提供的。

Refactoring

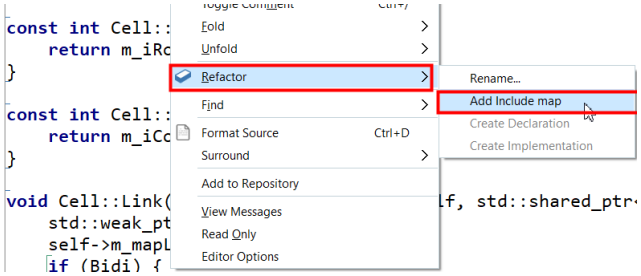
VA 的 Refactoring 提供了 3 个功能：可以为程序代码中的程序安全的重新命名，自动为程序加入表头文件以及自动产生函式的宣告或是实作内容。这 3 项功能都是以前 C++Builder 无法正确提供的



要安全的重新命名任何变量/函式，程序员可以把光标放在变量，函式中然后点选 Refactor|Rename，或是在编译程序中点选鼠标右键再点选 Refactor|Rename，接着 VA 会显示 Rename 对话盒列出所有定义和使用此变量/函式的程序代码给程序员检查，接着程序员只需要输入变量/函式的新名称，VA 就会安全的把所有相关的程序员改动完成。



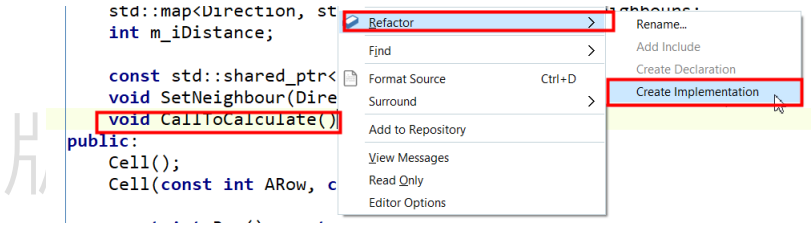
Refactoring 也可以自动帮程序员加入相关函式/API 的表头档，例如在前面 Code Insight 的范例中使用了 std::map 变量，那么程序员可以点选鼠标右键再点选 Refactor|Add Include map:



VA 就会自动在程序代码中加入相关的表头档

```
#include <map>
```

最后当程序员在宣告部份定义了函式之后就可以点选鼠标右键再点选 Refactor|Create Implementation：



VA 就会在 Cpp 档的实作中自动产生函式实作程序代码：

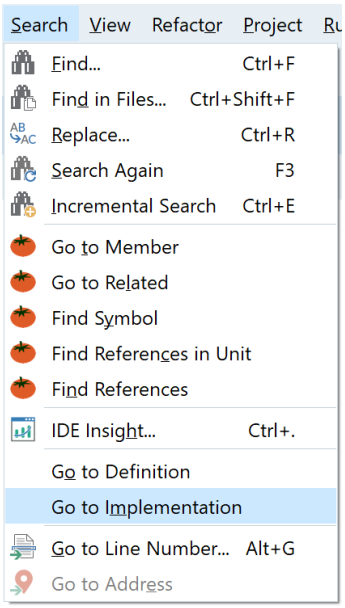
```
void Cell::CallToCalculate()
{


}
```

当然程序员也可以在 Cpp 档的实作中先撰写函式实作程序代码再要求 VA 自动在表头档中产生函数声明。

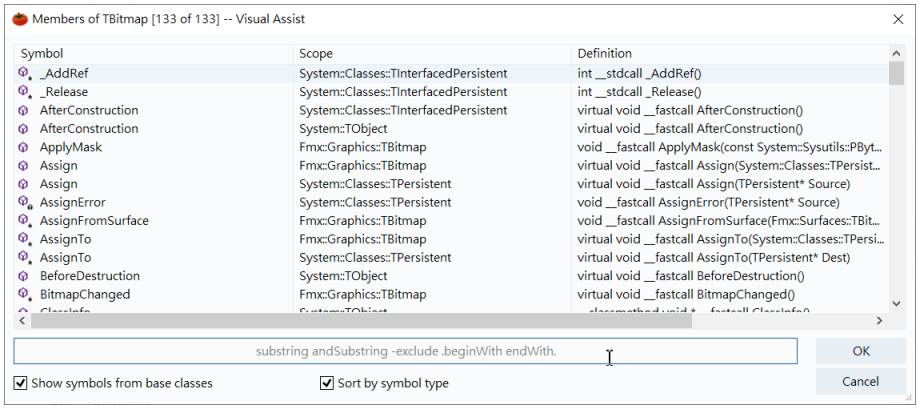
Navigation

VA 的 Navigation 几乎可以带领程序员寻找和观看变量/函式/类别所有的相关信息，这包含了宣告/实作/继承信息。

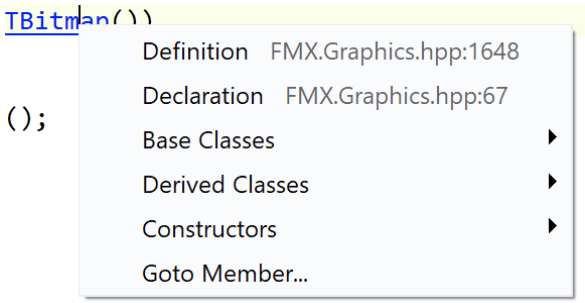
在 12 的 Search 选单中加入了 VA Navigation 所有的功能：



Go to Member 可以带领程序员到类别/变量定义和实作程序代码，让程序员可以快速在相关程序代码中来回。

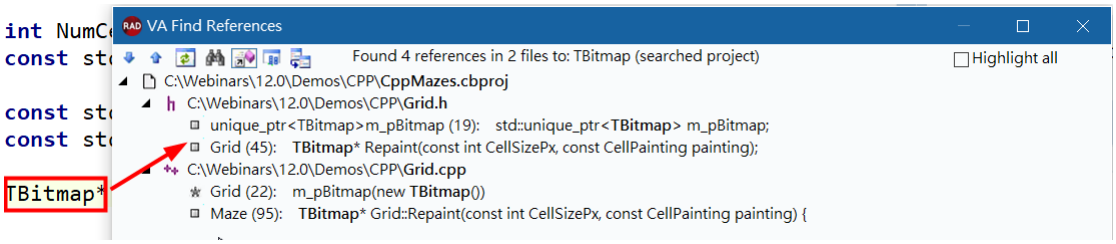


Go to Related 则可以让程序员查阅类别/变量的相关信息，例如点选类别并点选 Go to Related 后就可以查看类别的定义，成员，父代类别以及衍生类别：

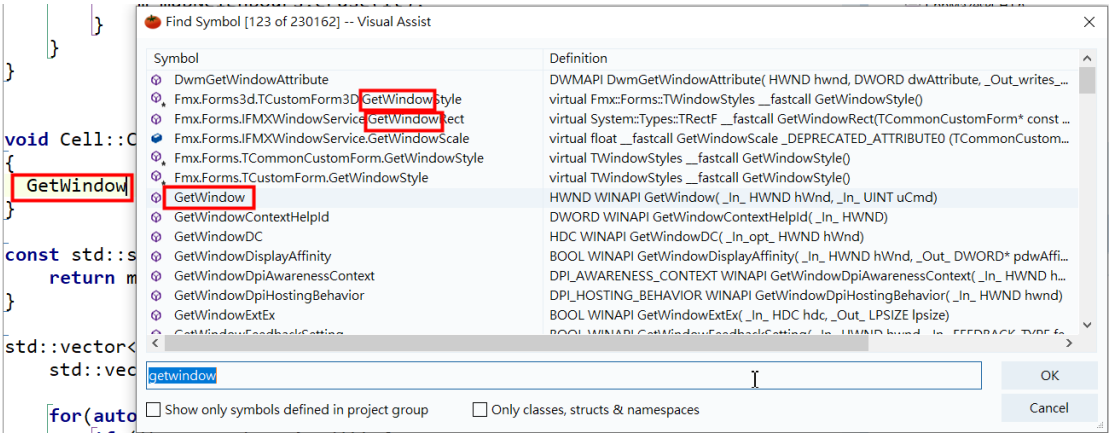


Find References in Unit 和 Find References 可以在本程序单元或是在项目中搜寻所有使用特定类别/变量的相关程序代码，例如在下面显示使用 Find

References 功能在项目中搜寻所有使用或是参考 TBitmap 类别的相关程序代码：



Find Symbol 则可以在项目或是**系统**中搜寻特定字符串/标记的信息，例如下面是在程序代码中呼叫 GetWindow 这个 API，但是程序员不确定 GetWindow 的定义以及定义的档案，此时程序员可以使用 Find Symbol 功能来搜寻，VA 就会显示所有和 GetWindow 有关的程序代码：



程序员只需要点选想要的信息，VA 就会自动开启 GetWindow 定义的 winuser.h 档案并且带领程序员到达 GetWindow 的定义程序代码：

```
WINUSERAPI
HWND
WINAPI
GetWindow(
    _In_ HWND hWnd,
    _In_ UINT uCmd);
```

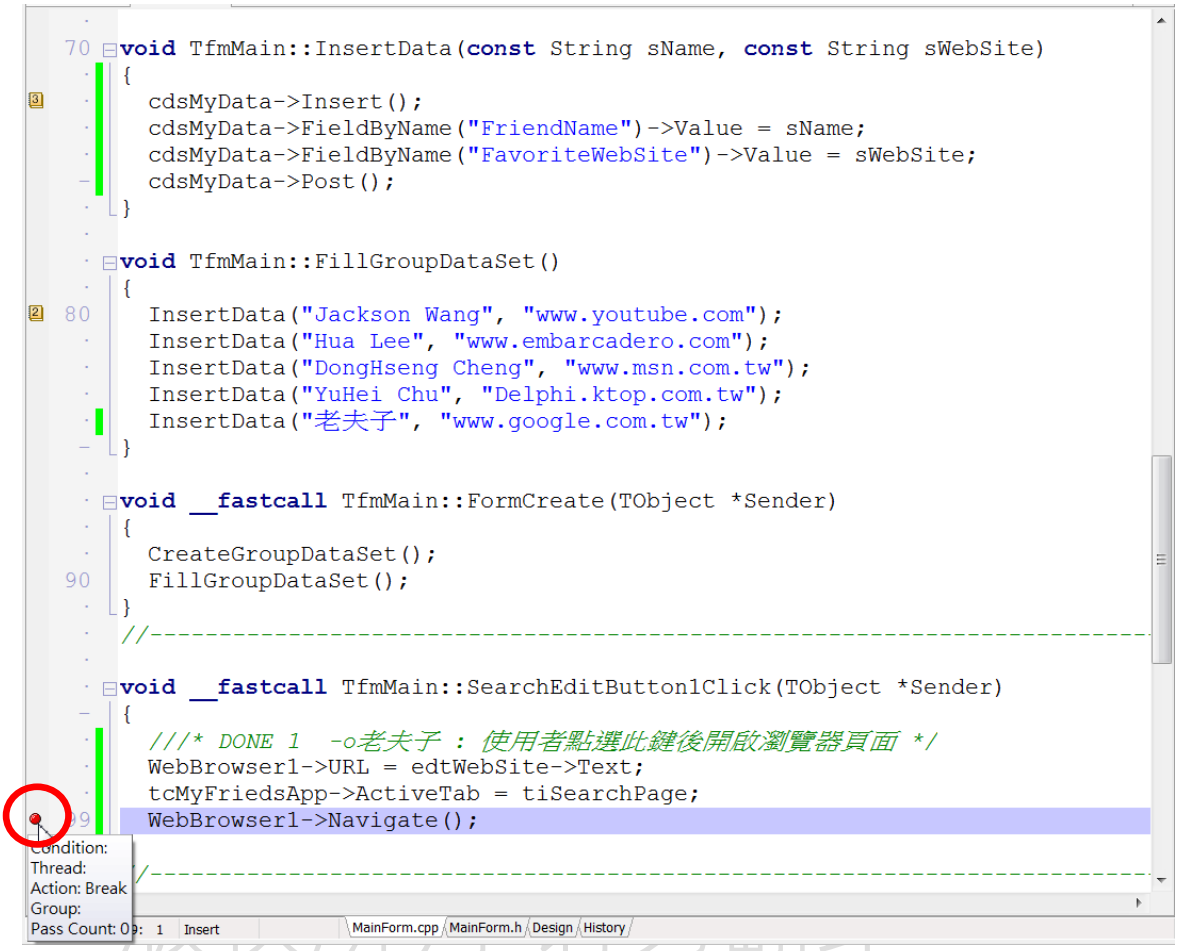
12 版的 VA 功能的确为 C++Builder 程序员带来了更高效率的程序代码生产力，特别是当您的项目愈来愈庞大时就能更体会到 VA 的好处，未来 C++Builder 会持续加入更多的 VA 功能。

5 除错您的 iOS App

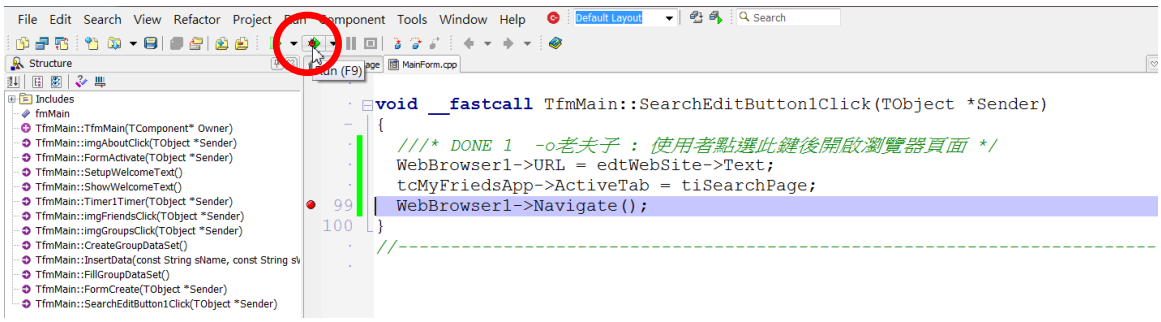
当您使用 C++Builder for iOS 开发 iOS App 时一定会需要除错您的应用程序，C++Builder for iOS 提供了非常方便又强大的除错功能能帮助您，例如本书的范例应用程序在实作了上一小节的程序代码后可能发生了一些错误，因此让我们学习如何除错应用程序以便修正范例应用程序中可能的错误。

C++Builder for iOS 能够让您在 iOS 仿真器中除错或是在 iOS 设备中除错，甚至提供了一个 Windows 仿真接口提供您除错。在本书中让我们展示如何在 iOS 仿真器中除错。首先让我们在范例应用程序中设定『断点』，『断点』是指当 iOS App 在 iOS 仿真器中或是在 iOS 设备中执行到此地时便会中断，以便让开发人员可以检查相关的变量，对象或是内存，CPU 等重要的信息，来决定程序代码是否发生了错误。

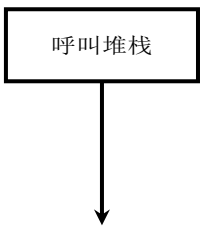
现在让我们在前面刚实作的 SearchEditButton1Click 程序中设定一个断点，请切换到程序代码页次，使用鼠标在 SearchEditButton1Click 程序的最后一行程序代码处的最左边单击鼠标左键，此时在 WebBrowser1.Navigate 最左边就会出现一个红色的圆点『』，这就代表在此设定了一个『断点』，稍后当范例 iOS App 在 iOS 设备中执行到此地时就会中断执行并且把执行权从应用程序切换回 C++Builder for iOS 的 IDE，如下所示：

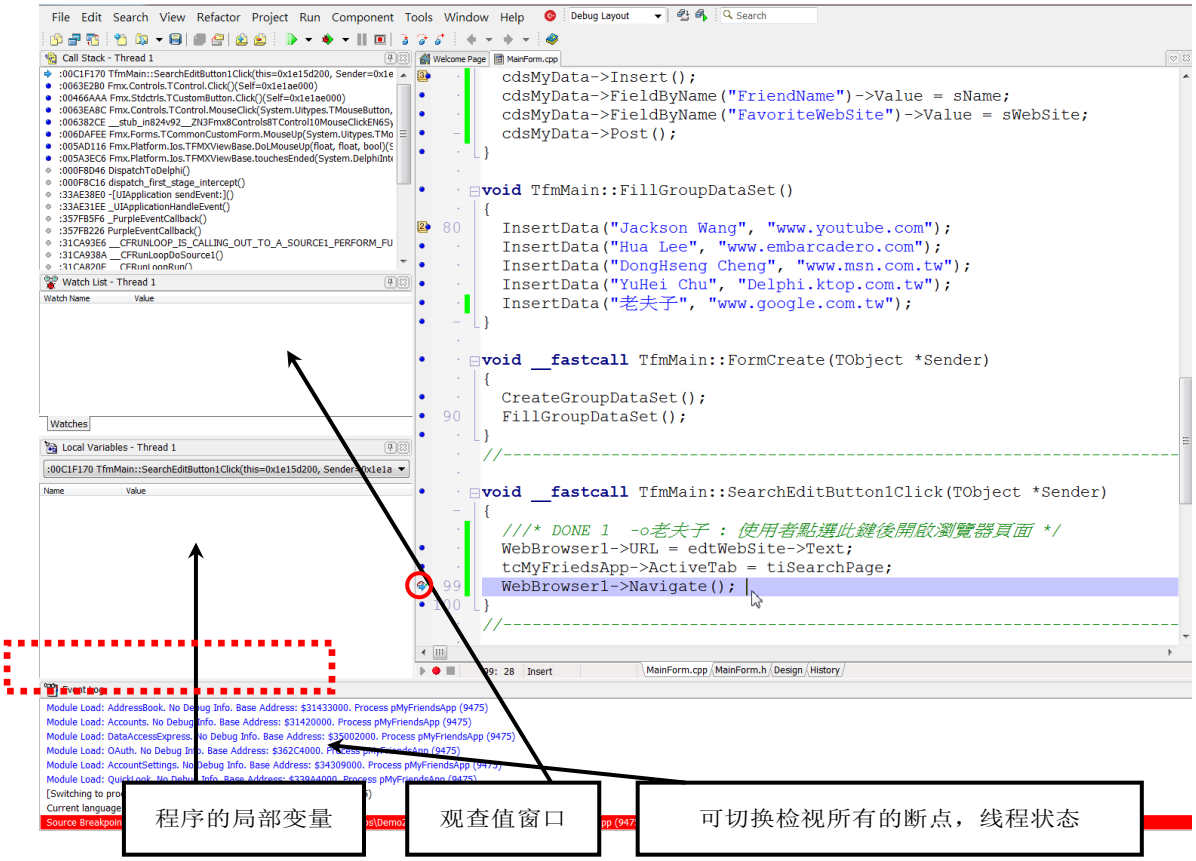


设定好此『断点』之后，您就可以点选 IDE 上方的『Run』按钮，或是按下『F9』，IDE 就会启动除错器开始执行您的应用程序，如下所示:



当范例 iOS App 执行后，请先点选主窗体 TTabControl 第 3 个页次，再点选 3 个页次中位于 FavoriteWebSite 旁的搜寻按钮，就可以看到范例 iOS App 被暂停执行，并且切换回 C++Builder for iOS 的 IDE，此时您会看到 IDE 暂停在刚才设定的『断点』上，而且原先『断点』的符号现在变成『』符号，如下所示。



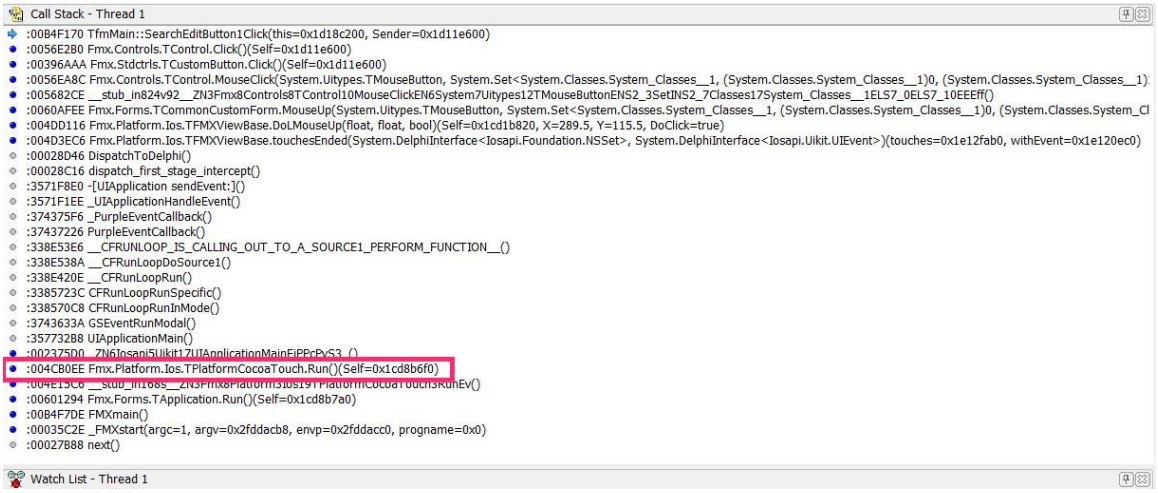


而且请您注意 IDE 右上方，此时 IDE 也被设定成在除错的桌面组态设定:

除错的组态设定会自动显示呼叫堆栈窗口，程序的局部变量窗口和观查值窗口。下面的表格说明了这些窗口的意义:

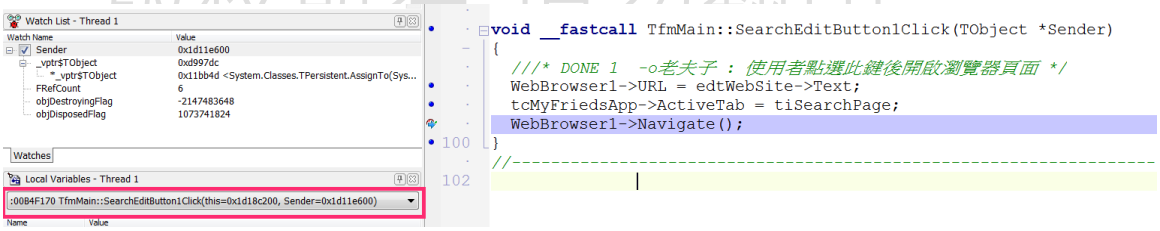
窗口	说明
堆栈窗口	应用程序的呼叫堆栈次序，您可以在这个窗口中看到断点被呼叫的执行次序
程序的局部变量窗口	此窗口自动显示目前程序中所有的局部变量的数值
观查值窗口	您可以在此窗口中加入检视任何的全局变量，数据结构或是对象的数值

现在请您先观察 IDE 左上方的『堆栈窗口』，您可以看到类似如下的内容:



『堆栈窗口』显示了您的 iOS App 的执行路径，例如在上图中可以看到这个范例 iOS App 的进入点是 FMX.Platform.iOS.TPlatformCocoaTouch.Run 程序，接着范例 iOS App 响应刚才在主窗体中鼠标的点选事件，从 TControl.Click 方法呼叫主窗体中的 TfmMainForm.SearchEditButton1Click 程序。

接着再观察『局部变量窗口』，如果您仔细比较『局部变量窗口』的内容和 SearchEditButton1Click 程序，如下所示:



您可以发现『局部变量窗口』中显示的内容正是 SearchEditButton1Click 程序中所有的局部变量和参数，如此一来当您除错 SearchEditButton1Click 程序时就可以对所有的局部变量值以及参数值一目了然。

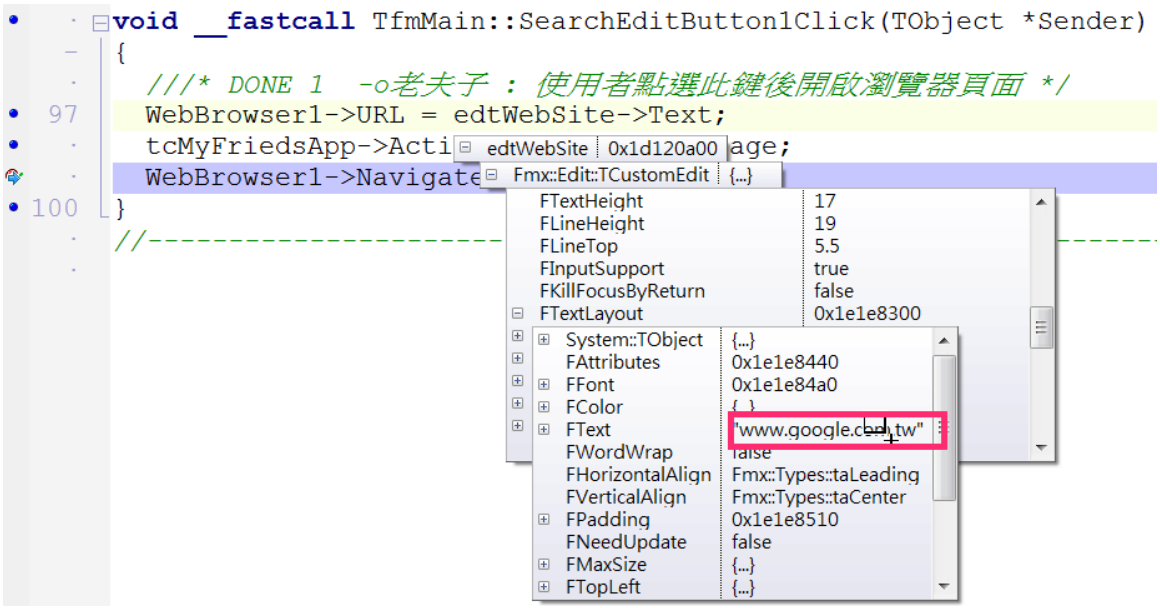
当应用程序执行权暂停在断点时，您也可以使用鼠标来检视程序代码中的变量，数据结构或是对象的数值。例如现在 SearchEditButton1Click 程序中使用了:

```
WebBrowser1->URL = edtWebSite->Text;
```

上面的 edtWebSite.Text 是 3 个页次中位于 FavoriteWebSite 字段的数值，此时您可能想知道它的数值是什么。

有数种方法可以让您观察 iOS App 程序代码中数据结构中的数据，第 1 种方法是使用鼠标选择您想观察的数据结构数值，然后暂停鼠标在此程序代码之上

一下子，IDE 就会直接显示这个数据结构中包含的数值。例如下图就是使用鼠标选择了程序代码中的 edtWebSite->Text 之后，您就可以看到 IDE 在光标下方显示了目前 edtWebSite->Text 中的数值：

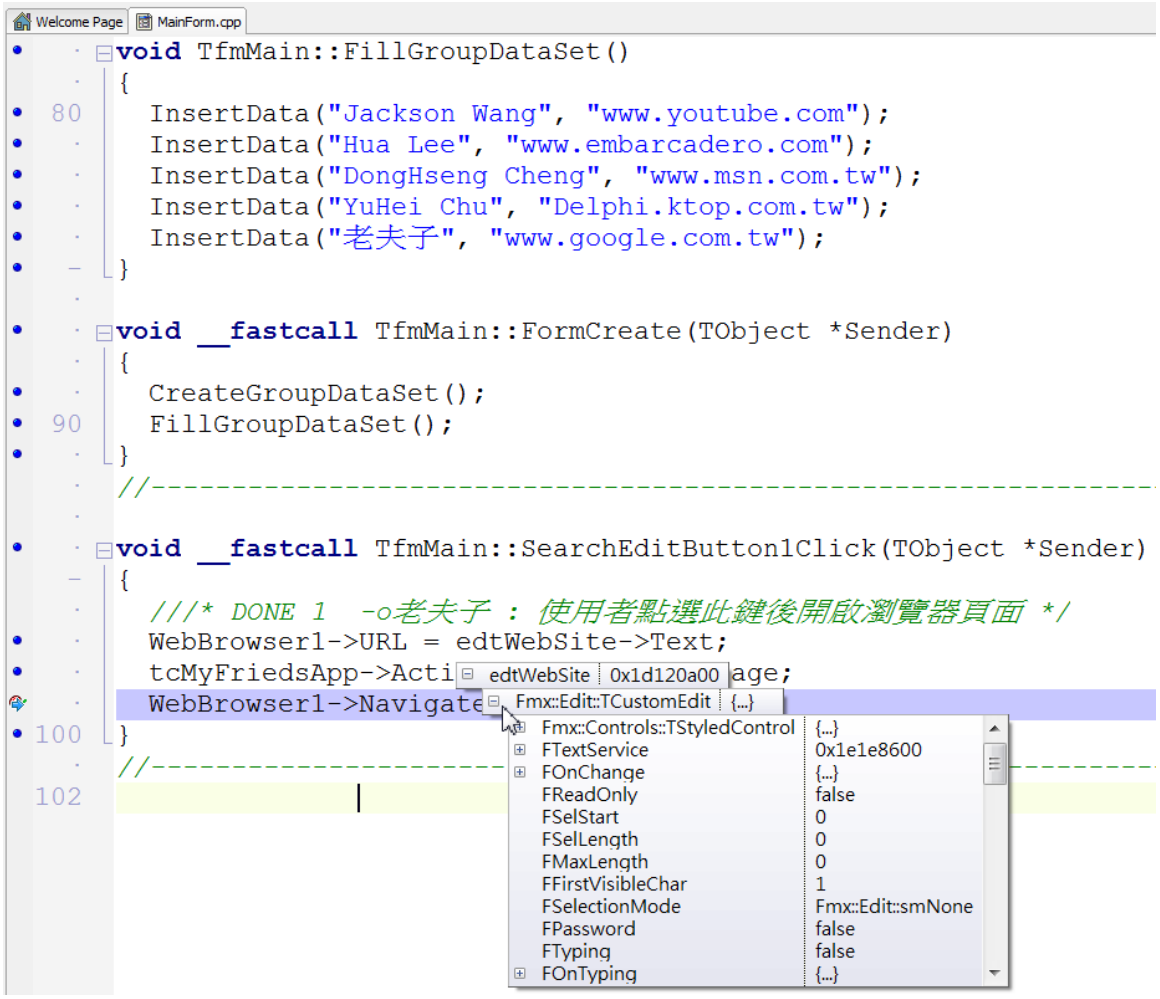


第 2 种方法是把这个程序代码拖曳到『观查值窗口』中，例如下图就是使用鼠标选择了 edtWebSite->Text 之后，把它拖曳到『观查值窗口』中。

一旦数据结构被拖曳到『观查值窗口』之后，它就会停驻在『观查值窗口』中，而且当数据结构中的数值改变时，『观查值窗口』也会立刻显示最新的数值。

如果您想观察整个数据结构或是对象中所有的数据，那么您可以使用鼠标把光标放在此数据结构或是对象之上，那么除错器就会显示其中所有的数值。例如下图就是把鼠标光标放在程序代码中的 edtWebSite 之上，除错器就立刻显示 edtWebSite 之中所有的数据：

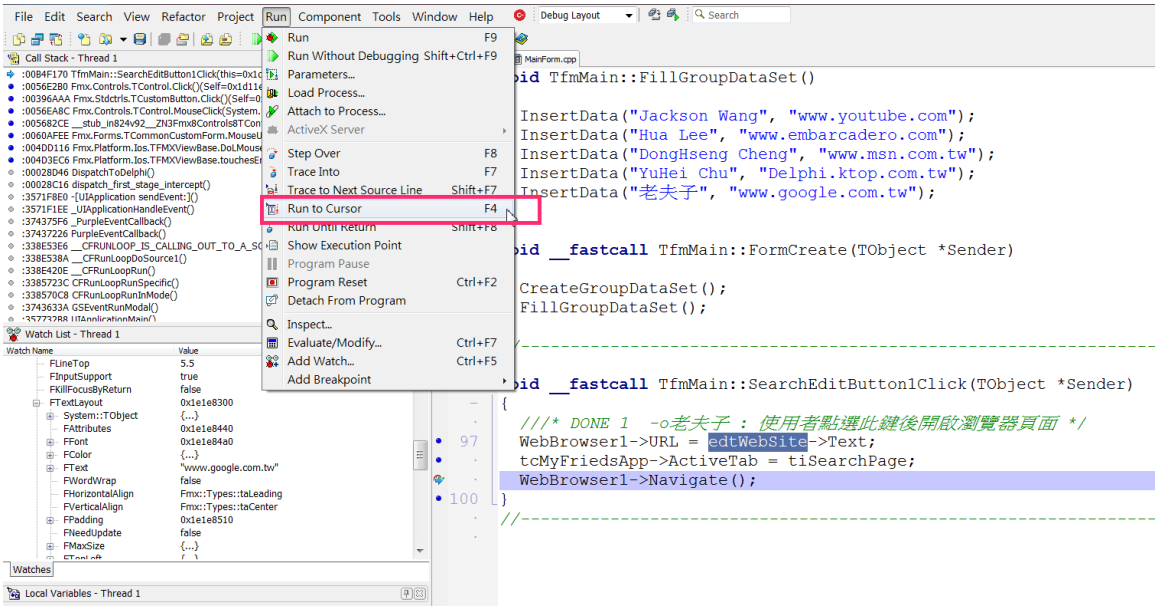
请注意上图中 edtWebSite 左方有一个『+』符号，这代表您可以使用鼠标展开其中的内容。例如在下图使用鼠标单击 edtWebSite 左方的『+』符号，就可以看到除错器展开 edtWebSite 中的内容，除错器详细的列出了 edtWebSite 中每一个元素的数值，这个功能对于观察复杂的数据结构或是对象的内容是非常有用的。



现在请您再次单击程序代码中的断点以取消原本的断点，然后按下 F9 执行范例应用程序，您就可以看到范例 iOS App 会继续执行下去了：

现在范例 iOS App 果然根据不同的数据中 FavoriteWebSite 字段的数值带领使用者到不同的网站了。

当您在除错应用程序时，您也可以按下 F7 键一行一行的除错程序代码，或是按下 F8 键一次执行一个方法。如果您的程序代码中拥有循环而您不想一直在循环中除错，您可以在循环之后的程序代码处设定断点，再按下 F4 键一次执行到循环之后的断点，下图就是您在 IDE 中除错时可以使用的功能键:



6 开发和分发 iOS App 到 iOS 设备中

在 C++Builder For iOS 中要分发 App 到实际的 iOS 设备中，您需要执行下面的流程:

- 先确定安装了 XCode 的命令行工具
- 在 C++Builder For iOS 中建立 iOS 设备的远程组态
- 在 C++Builder For iOS 开发和除错您的 iOS App
- 取得 Apple 开发/分发认证
- 使用 C++Builder For iOS 的部署管理员分发您的 iOS App

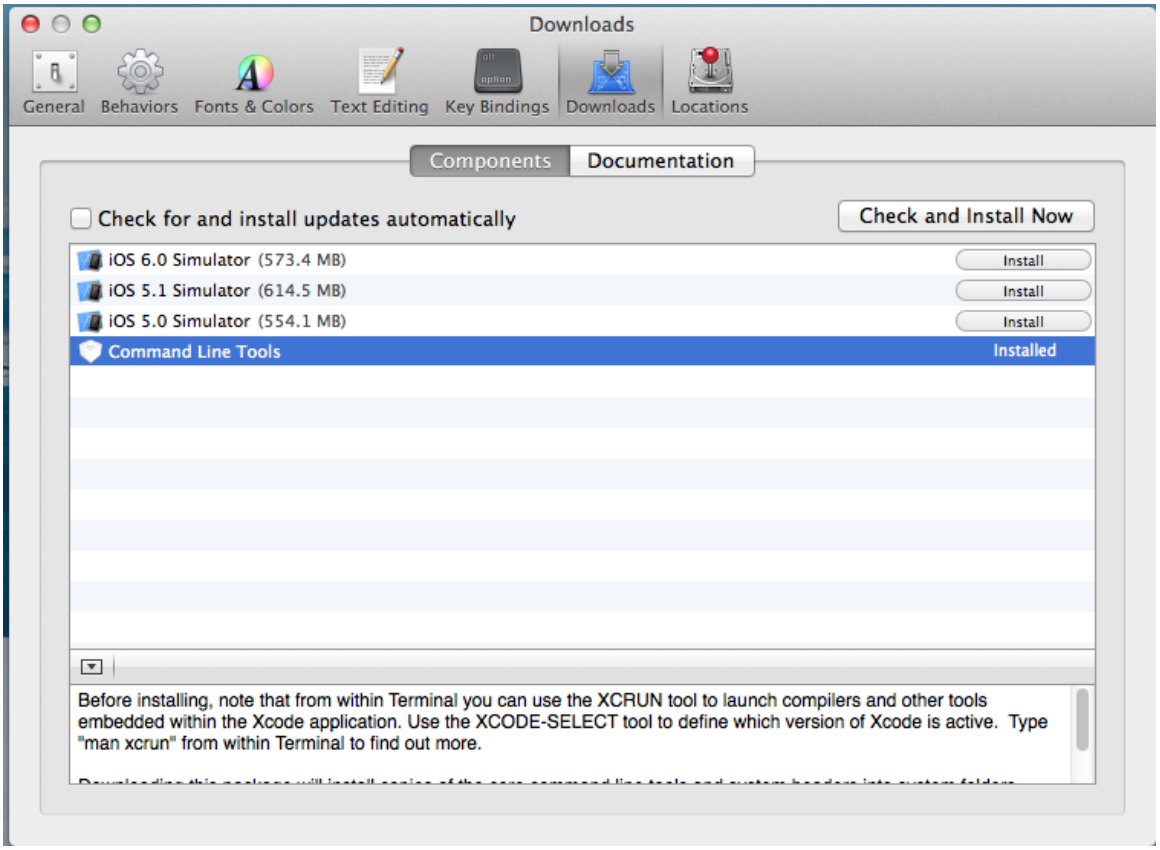
在下面的小节将使用一个实际的范例来说明上述的流程。

确定安装了 XCode 的命令行工具

在实际能够分发 iOS App 之前，请确定您的 XCode 已经安装命令行工具，因为 C++Builder For iOS 需要这些命令行工具来数字签名分发的 App 到 iOS 设备中。要安装 XCode 命令行工具，请执行 Mac 中的 XCode，然后点选:

```
[Xcode]|[Preferences...]|[Donwloads]|[Components]|[Command Line Tools]|[Install]
```

安装命令行工具，在正确安装完之后，您应该可以看到类似如下的结果:



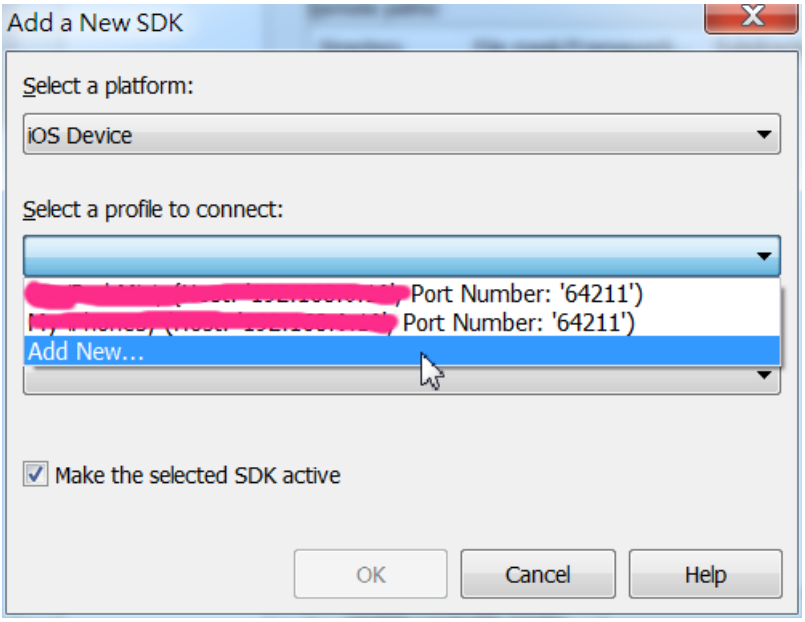
现在我们就可以进行下一步了。

建立 iOS 设备的远程组态

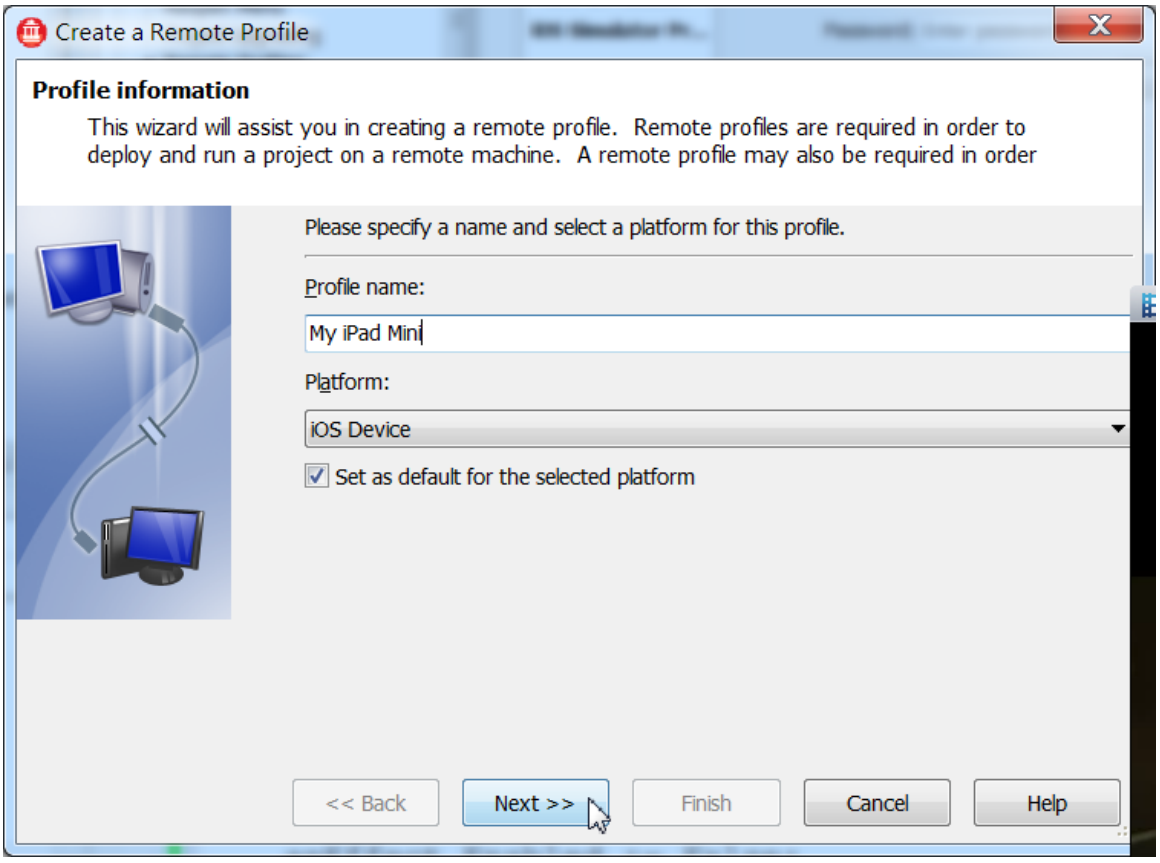
在前面我们说明了如何在 C++Builder For iOS 整合发展环境中建立 iOS 仿真器的组态以便我们开发和测试 iOS App。如果我们需要在 C++Builder For iOS 整合发展环境中实际分发 iOS App，我们也需要建立 iOS 设备的远程组态，才能藉由 C++Builder for iOS 和 XCode 的命令行工具把 App 自动分发到 iOS 设备中。

> 在建立 iOS 设备远程组态之后请先确定 Mac 平台中的 PSServer 已经在执行状态中。

请在 C++Builder for iOS 整合发展环境中点选 Tools | Options 菜单，在 SDK Manager 中点选『Add』按钮建立远程组态，此时会出现一个『Add a New SDK』对话盒，请在 Select a platform 字段中选择 iOS Device，再于 Select a profile to connect 字段中选择『Add New...』选项，如下所示:

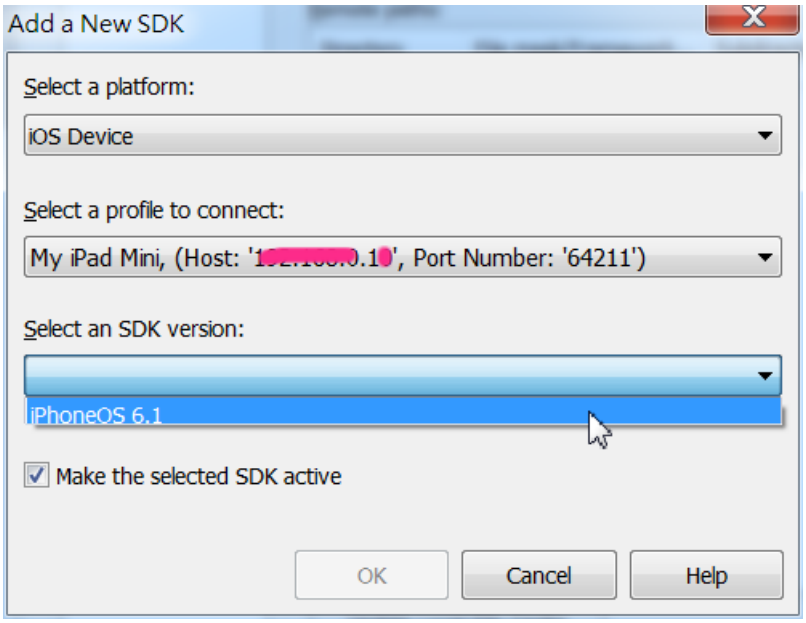


然后在 Create a Remote Profile 对话盒中为您的 iOS 设备取一个名称并且在 Platform 字段中选择建立『iOS Device』平台组态。例如笔者使用的 iOS Device 是 iPad Mini，因此在下面的对话盒中取名为 My iPad Mini 设备名称:



接着点选 Next 按钮到下一个页面输入 Mac 主机的名称或 IP 位置之后就会回到『Add a New SDK』对话盒，再点选 Select an SDK version 后『Add a

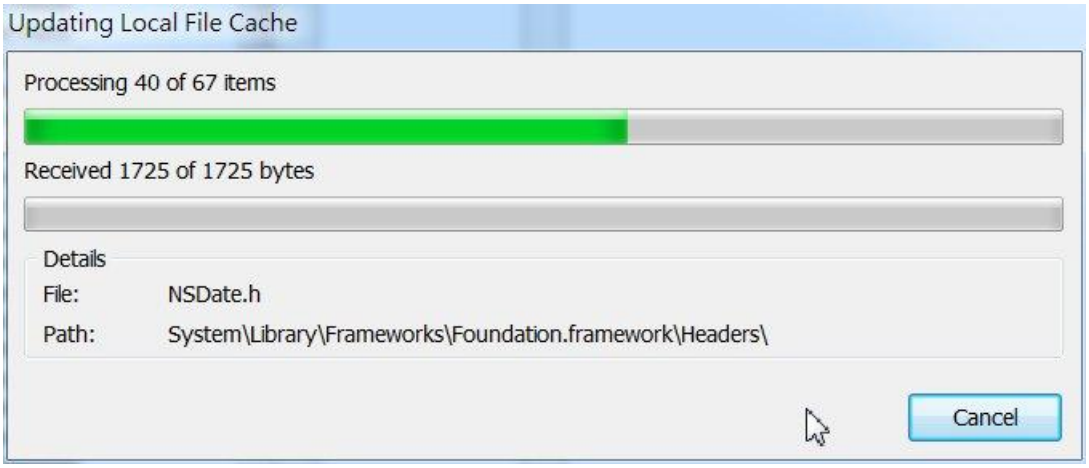
New SDK』对话盒就会立刻和 Mac 通讯并未找出此 iOS 设备使用的 SDK 版本，例如下图就显示『Add a New SDK』对话盒找到笔者使用的 iPad Mini 是使用 iOS 6.1 的版本，请选择显示的版本：



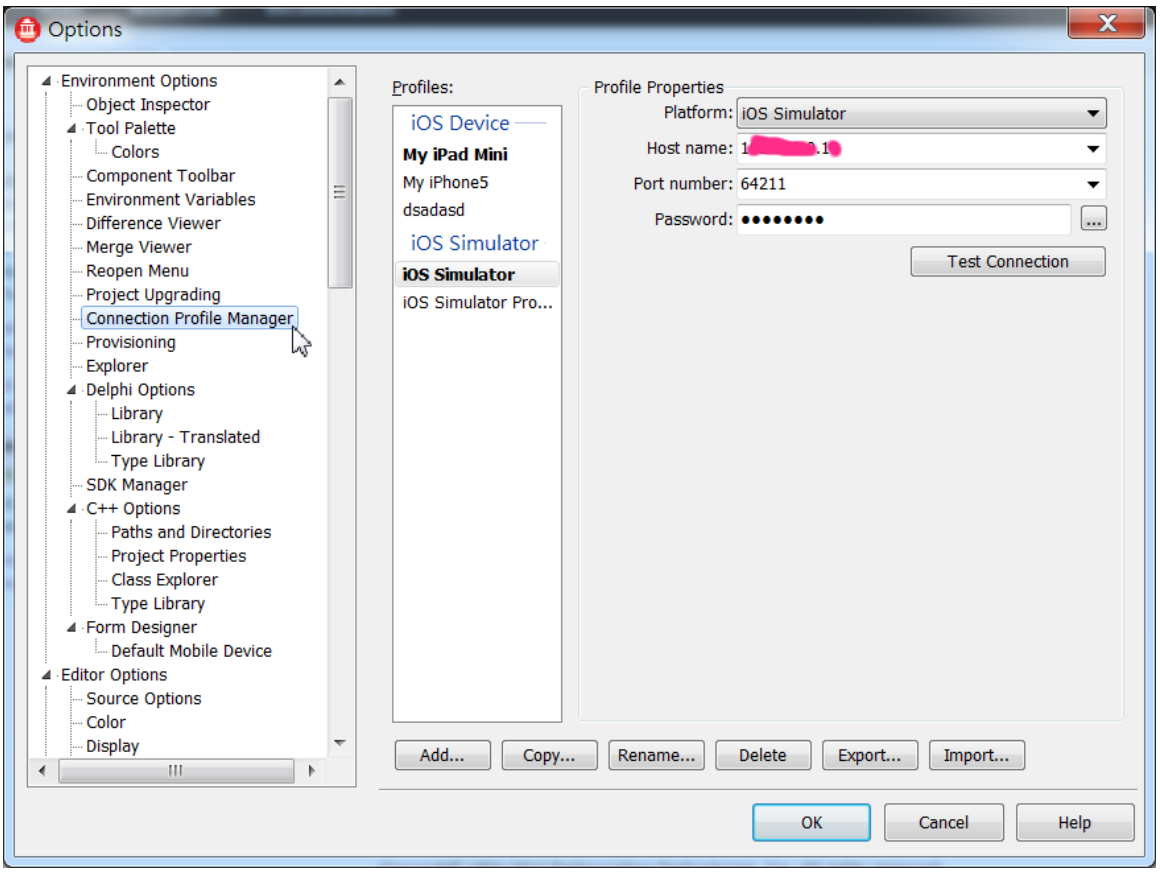
再点选 OK 按钮之后

(笔者建议读者在建立完 iOS 设备的远程组态之后可以先关闭整个 Options 对话盒，再使用 Tools|Options 菜单开启 Options 对话盒，再到『SDK Manager』选项建立 SDK 组态)

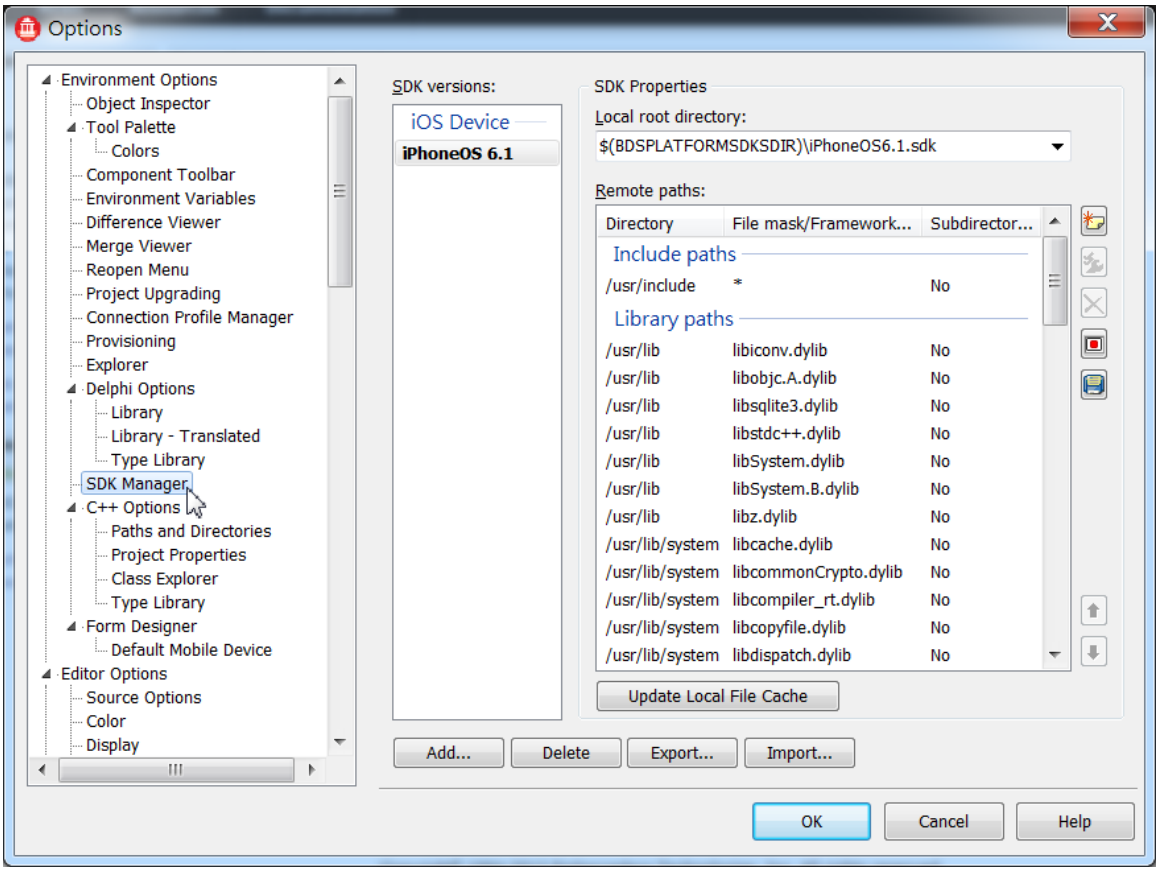
此时 C++Builder for iOS 整合发展环境就会自动根据您的设定拷贝和更新相关的档案以帮助您分发 iOS App，如下所示：



在完成上面的步骤后设定 iOS 设备远程组态的工作就完成了，此时在您的 Connection Profile Manager 中就应该有类似如下的组态，分别是分发到 iOS 仿真器中的组态和分发到 iOS 实际设备中的组态:



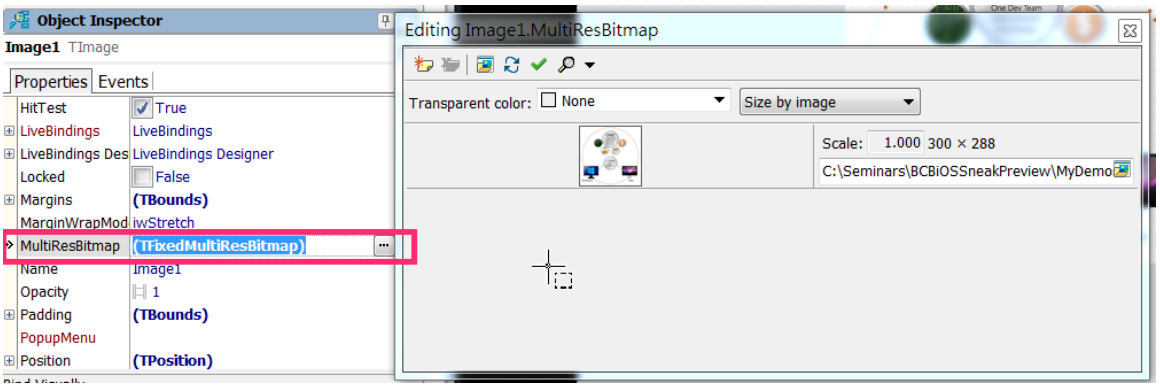
而 SDK Manager 中也会显示您的 iOS 设备使用的 SDK 版本信息和相关的档案:

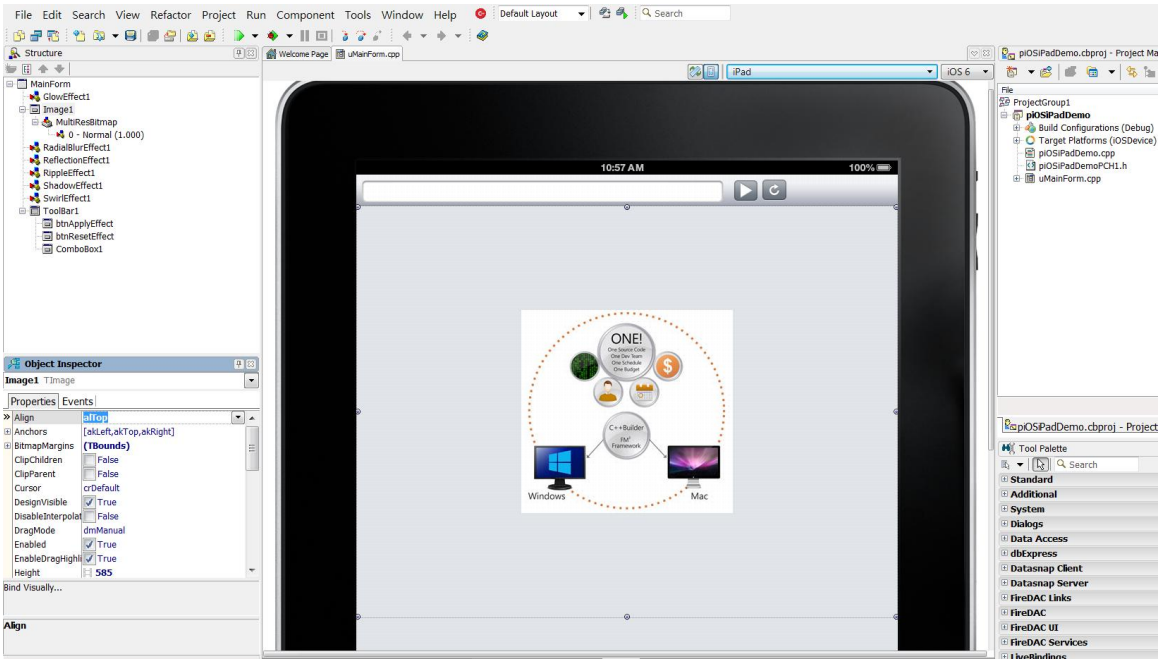


开发 iPad Mini 范例 App

现在请在 C++Builder for iOS 整合发展环境中建立一个 FireMonky Mobile Application 项目，在主窗体右上方选择建立 iPad 窗体，如下所示:

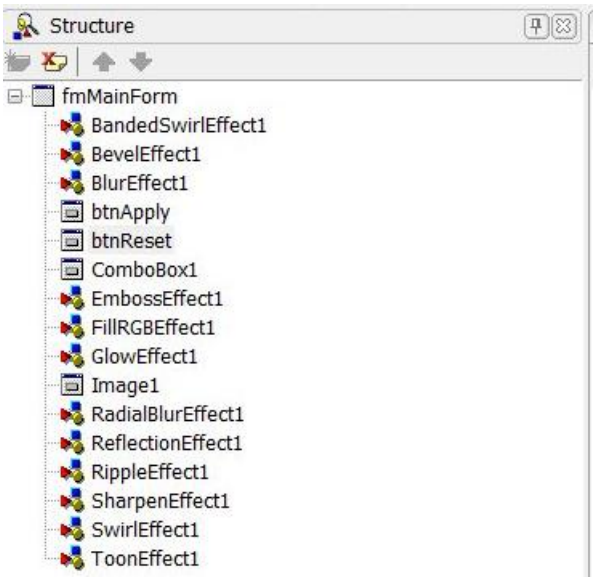
接着在主窗体中放入 TImage，TEdit，TComboBox 和 2 个按钮组件，再从工具盘中随便选择数个 Effect 组件，例如 TRippleEffect TShadowEffect 等。最后再使用 TImage 组件的 MultiResBitmap 特性的特性值编辑器加载一个影像，如下所示:





这个 iPad 范例 App 的功能是让用户可以从 TComboBox 中选择要对主窗体 TImage 组件中的影像执行各种影像效果，并且在 TImage 组件中显示各种效果执行的结果。

因此现在如果您检视整合发展环境左上方的树状架构窗口，应该可以看到类似如下的结果，我们在主窗体中放入的各种效果组件都是属于主窗体的子组件。



现在让我们撰写一些程序代码让这个 iPad App 能够工作。首先建立主窗体的 OnCreate 事件处理函式，它呼叫了 GetAllAvailabelEffects()方法取得主窗体中所有效果组件:

```
void __fastcall TMainForm::FormCreate(TObject *Sender)
{
   //取得 FireMonkey 的 Effect
   GetAllAvailabelEffects(ComboBox1->Items);
}
```

GetAllAvailabelEffects 方法检视主窗体中所有属于 TEffect 的衍生组件，并且把它的类别名称和组件参考加入到 TComboxBox 中:

```
void TMainForm::GetAllAvailabelEffects(TStrings *pSL)
{
  for (int iCount = 0; iCount < this->ComponentCount; iCount++)
  {
    TComponent *pComponent = this->Components[iCount];
    if (pComponent->InheritsFrom(__classid(TEffect)))
    {
      TEffect *pEffect = (TEffect *) (this->Components[iCount]);
      pSL->AddObject(pEffect->ClassName(), (TObject *) pEffect);
    }
  }
}
```

接着为主窗体中第一个按钮实作如下的 OnClick 事件处理函式。当用户点选此按钮之后 005 行就从 TComboBox 中取得前面储存在 TComboBox 中的效

果组件参考，008 行先呼叫 ResetEffect 方法以清除以前使用过的效果组件。要让效果组件影响主窗体中的 TImage 组件中的影像，我们只需要在 009 行设定效果组件的 Parent 特性值为 TImage 组件，010 行再设定效果组件的 Enabled 特性值为 true 即可。最后 011 行把目前作用中的效果组件储存在 pOldEffect 对象变量中，以便稍后用户使用其他效果组件时先移除目前效果组件的作用。

```
001   void __fastcall TMainForm::btnApplyEffectClick(TObject *Sender)
002   {
003     if (ComboBox1->ItemIndex != -1)
004     {
005       TEffect *pEffect = (TEffect*)
(ComboBox1->Items->Objects[ComboBox1->ItemIndex]);
006       if (pEffect != NULL)
007       {
008         ResetEffect(pOldEffect);
009         pEffect->Parent = Image1;
010         pEffect->Enabled = true;
011         pOldEffect = pEffect;
012       }
013     }
014   }
```

第二个按钮的 OnClick 事件处理函式是在移除效果组件的作用:

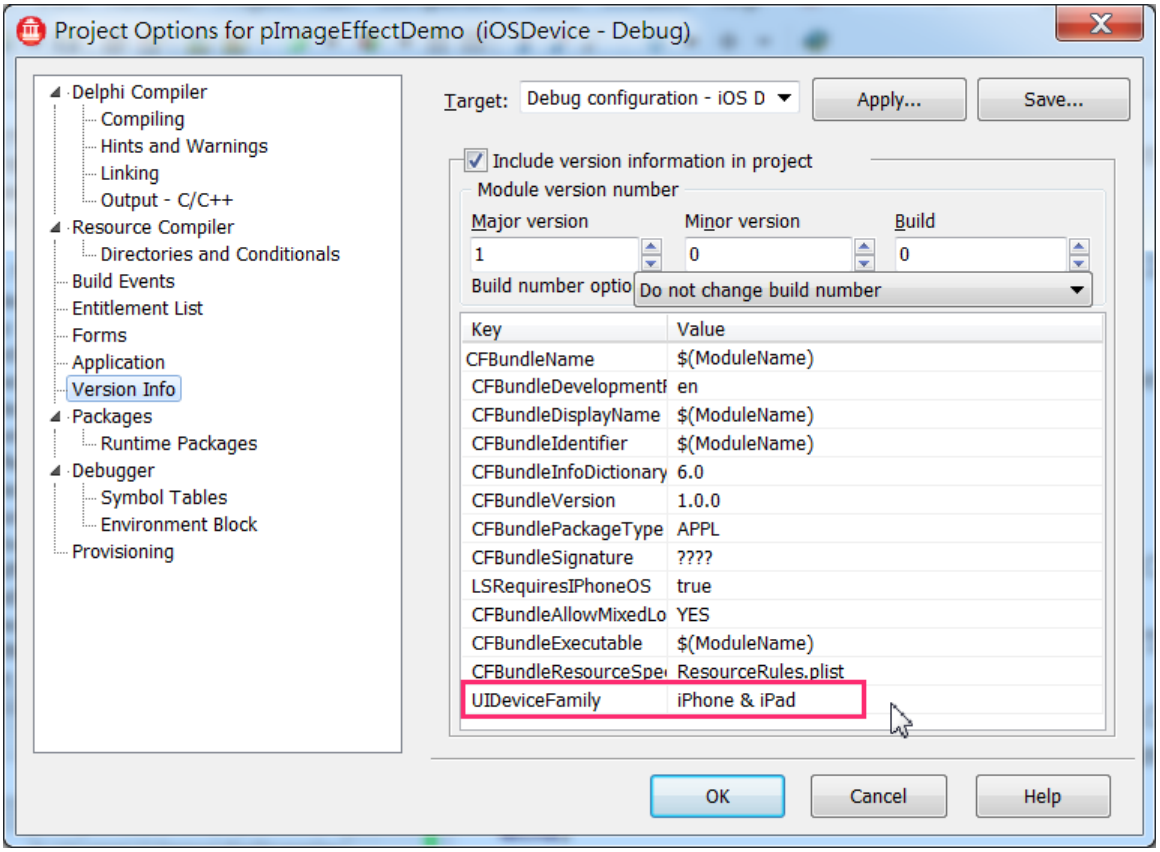
```
void __fastcall TMainForm::btnResetEffectClick(TObject *Sender)
{
  ResetEffect(pOldEffect);
}
```

ResetEffect 方法判断目前是否有任何效果组件在作用中，如果是的话就先把作用中的效果组件关闭，再设定它的 Parent 特性值为 nil 以便让 TImage 组件中的影像回复原始的状态，最后再把 oldEffect 对象变量设定为 nil。

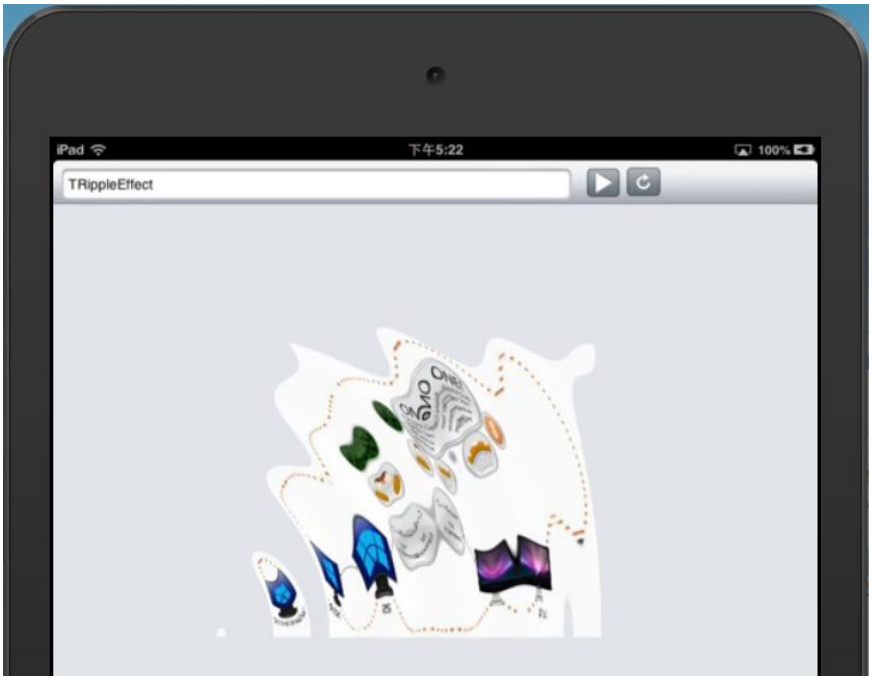
```
void TMainForm::ResetEffect(TEffect *pEffect)
{
  if (pEffect != NULL)
  {
    pEffect->Enabled = false;
    pEffect->Parent = NULL;
    pOldEffect = NULL;
  }
```

```
}
```

不我们可以准备执行这个范例 iPad App 了，但在这之前请在项目管理员中点选鼠标右键，选择 Options|Version Info 选项，确定其中的『UIDevice Family』的选项值为『iPhone & iPad』，如下所示：



如果一切顺利读者就可以在 iOS 仿真器中看到它启动 iPad 仿真器，请选择 TComboBox 中的效果组件，接着点选第一个按钮就可以看到类似下面的执行结果了：



现在我们已经成功的开发了一个 iPad App，但我们如何才能够把这个 iPad App 分发到真正的 iPad Mini 中执行呢?

取得 Apple 认证

在实际能够分发您使用 C++Builder for iOS 开发的 App 到 iOS 设备之前，您需要具备 iOS 开发人员计划资格。您可以使用 2 种方式取得 iOS 开发人员计划资格:

- 一是自行加入，您需要支付每年 99 美元的费用以加入这个计划
- 或是加入您公司的 iOS 企业开发计划，例如笔者就是藉由这个方法取得认证，Embarcadero 允许笔者参加 Embarcadero 的 iOS 企业开发计划

虽然有 2 种不同的方法可以取得您的 iOS 开发资格，但一旦您取得了资格之后接下来取得认证的步骤就差不多，因此本书将说明笔者如何取得企业开发计划并且据以取得分发认证以部署笔者使用 C++Builder for iOS 开发的 App 到 iPad Mini 设备之中。

这整个流程如下:

- 先申请 Apple ID
- 向 iOS 企业计划管理者发 EMail 要求加入 iOS 企业计划
- iOS 企业计划管理者会把您加入 iOS 企业计划，Apple 会自动寄一封 EMail 告诉您已经加入了 Apple 的 iOS 企业开发计划:

Welcome to the iOS Developer Enterprise Program

Dear Gordon Li,

Thank you for joining the iOS Developer Enterprise Program. You now have access to a comprehensive set of development tools and resources to assist you in developing and distributing innovative apps for iPhone, iPad, and iPod touch.

How to get started
Learn how to take advantage of your program resources and benefits.
iOS Enterprise Developer Program Getting Started Guide

Access your account
Log in to Member Center to manage your account and access technical resources.

Best regards,

Apple Developer Program Support

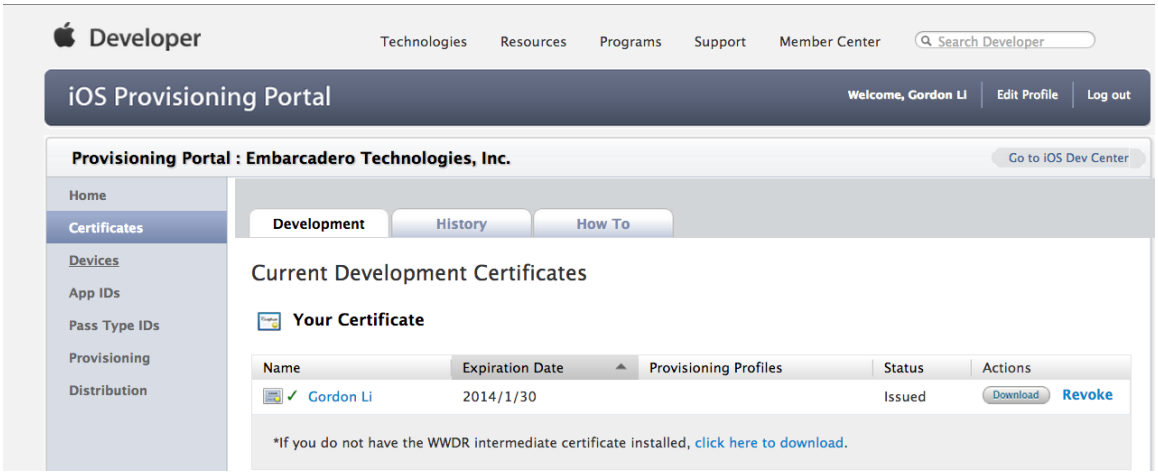
- 请使用浏览器前往:

```
https://developer.apple.com/devcenter/ios/index.action
```

并且使用您的 Apple ID 登录:

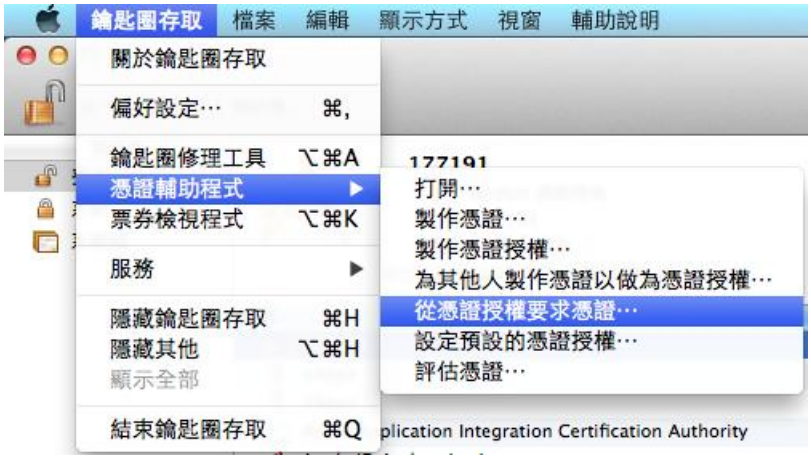
在成功登录之后请点选浏览器右上方的『iOS Provisioning Portal』准备向 Embarcadero 的 iOS 企业计划管理者申请笔者 iPad Mini 的认证。

首先点选浏览器左方的 Certificates 项目，然后点选下载 AppleWWWDRCA.cer 档案：

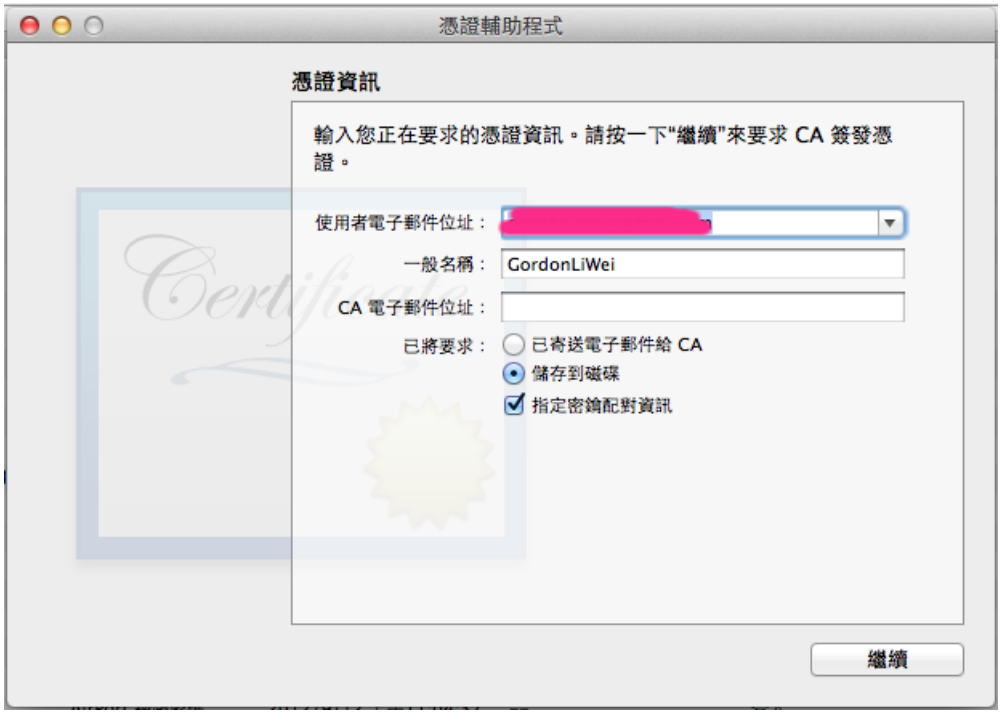


下载之后双击 AppleWWWDRCA.cer 档案就会自动启动钥匙圈存取程序，在其中读者可以看到 Apple WorldWide 开发信息已经注册：

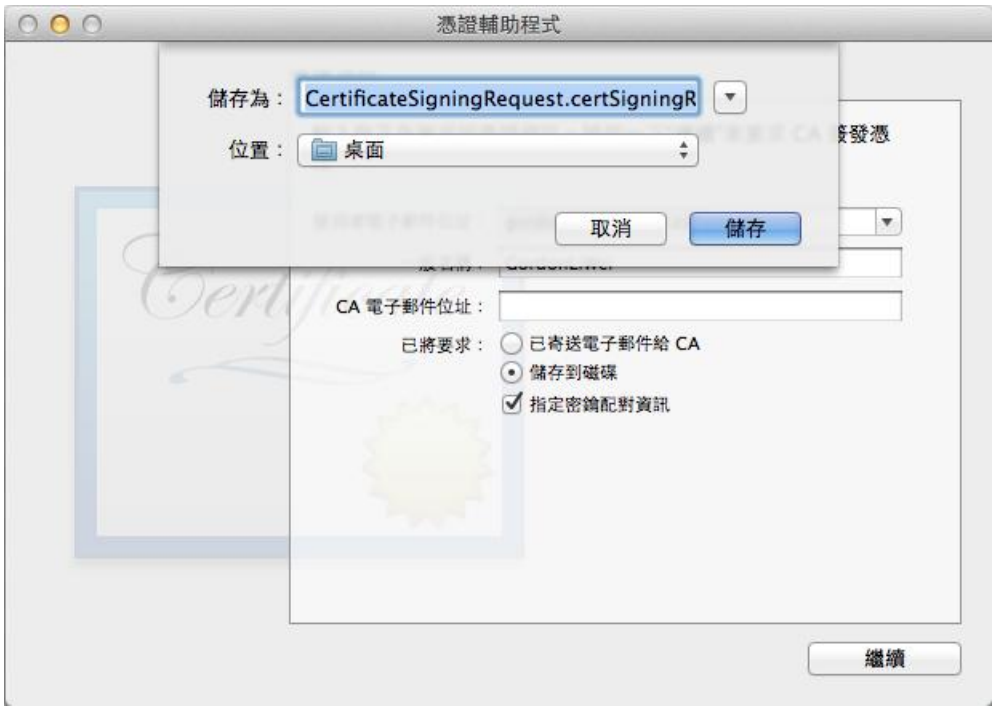
接着请点选钥匙圈存取程序菜单，选择从证书颁发机构机构(对笔者来说就是 Embarcadero)要求凭证，如下所示：



在凭证辅助程序中输入您的信息:



点选继续凭证辅助程序会在桌面储存一个档案:CertificateSigningRequest.certSigningRequest:

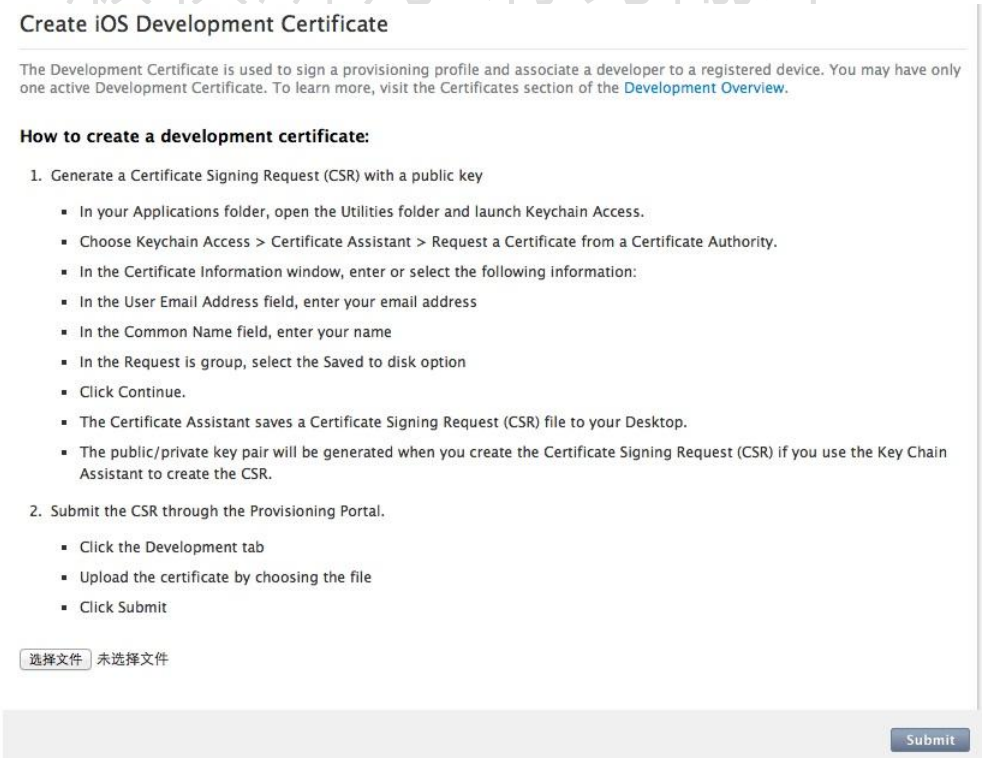


请选择 RSA 算法，点选继续

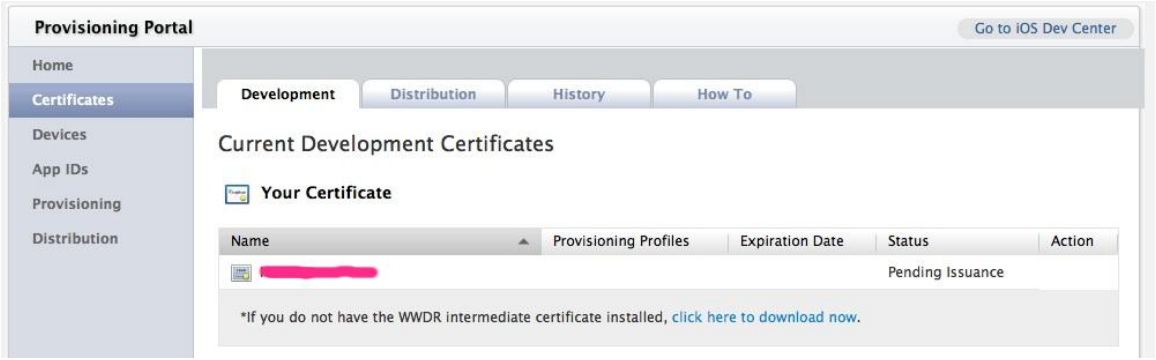
点选继续之后就会在 Mac 的桌面看到产生的 CertificateSigningRequest.certSigningRequest:

回到浏览器的 Certificates 项目，点选其中的『Request Certificate』按钮：

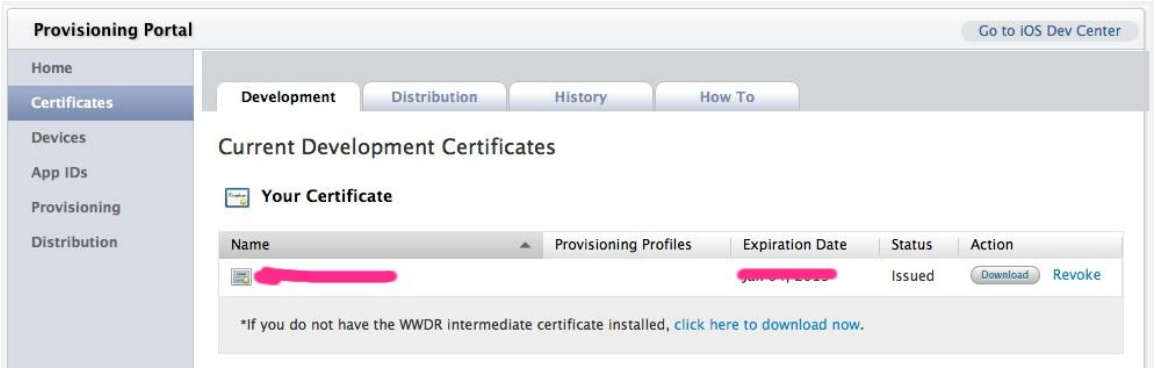
在下面的画面中选择前面储存的 CertificateSigningRequest.certSigningRequest 档案：



然后点选『Submit』按钮提出授权要求现在您就需要等待了，等待您的管理员核准您的授权请求。此时在 Certificates 项目中您会看到您的授权请求在 Pending 状态，等待管理员批准:



一旦您的管理员批准您的授权请求之后您就可以在 Certificates 项目中看到授权被核发了(Issued):



现在您就可以点选其中的『Download』按钮正式下载您的 iOS 开发授权认证，笔者的许可证文件是 ios_development.cer:

最后双击此授权认证档案之后认证信息就会成功写入您的 Mac 机器中，最后一个步骤就是连结您的授权认证档案和您的 iOS 设备，在笔者的范例中就是笔者使用的 iPad Mini。

Please register my iPad Mini

Gordon Li

寄件日期: 2013年1月31日 下午 02:09

收件者: [redacted]

Hi: [redacted]

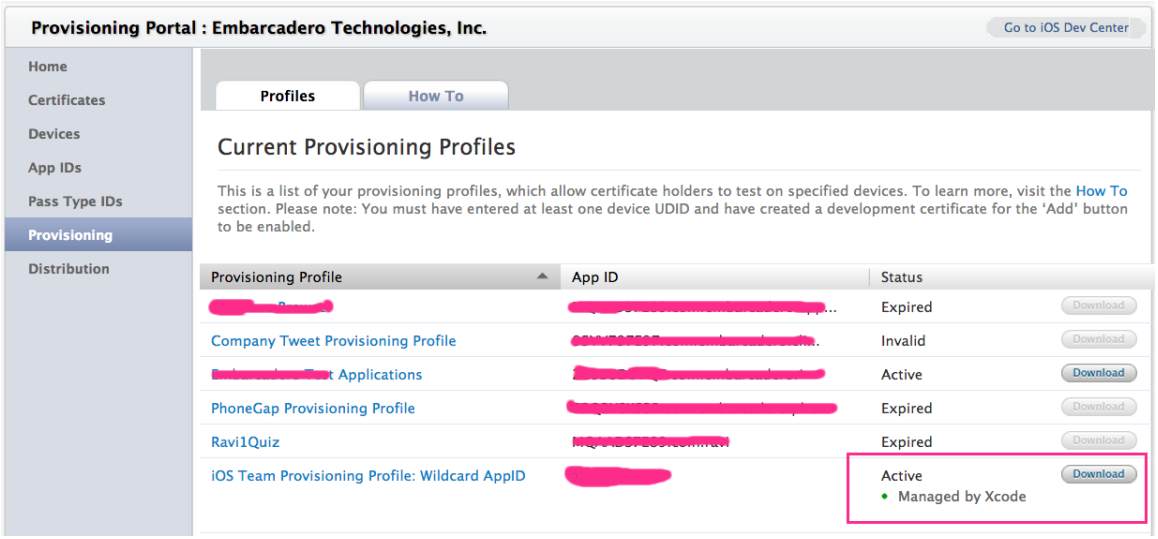
I have downloaded my ios_development.cer, so, could you register it at iOS portal?

Device Name: Gordon iPad Mini
UUID: a[redacted]1

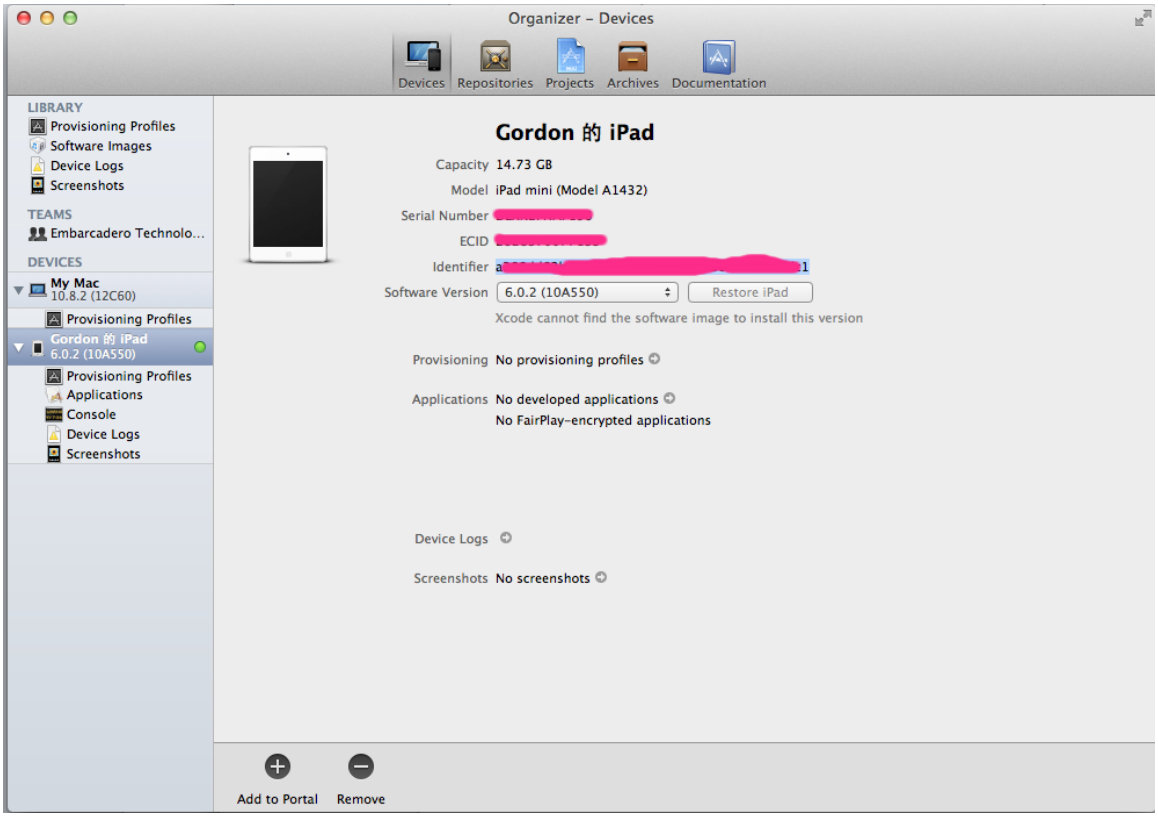
Thanks.

Cheers
Gordon

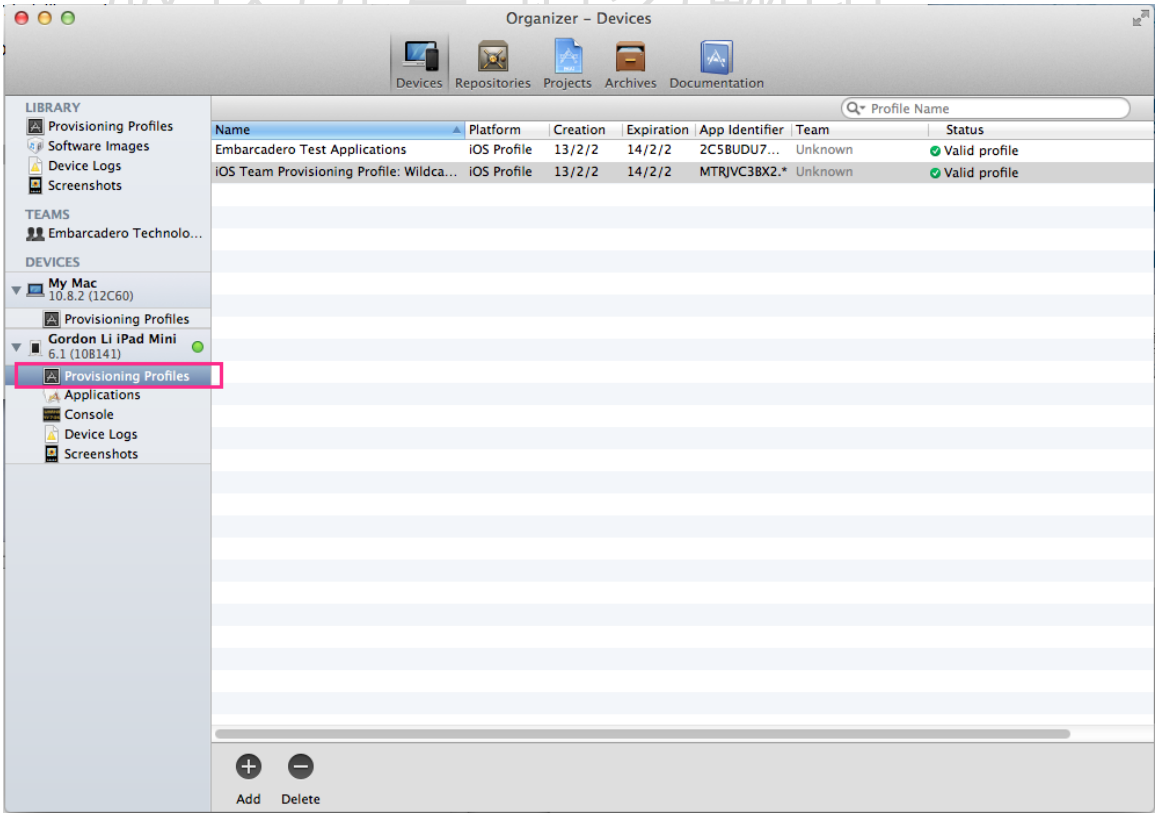
最后到 Provisioning 页次下载可使用的认证档案，再双击它以便把认证信息汇入到 XCode 中:



一旦完成这些步骤之后请读者执行 XCode 并且连结您的 iOS 设备到 Mac 机器，点选 XCode 的 Windows | Organizer 菜单，就可以看到类似如下的画面，XCode 成功的连结了笔者的 iPad Mini:



点选它的『Provisioning Profiles』页次就可以看到授权认证信息:



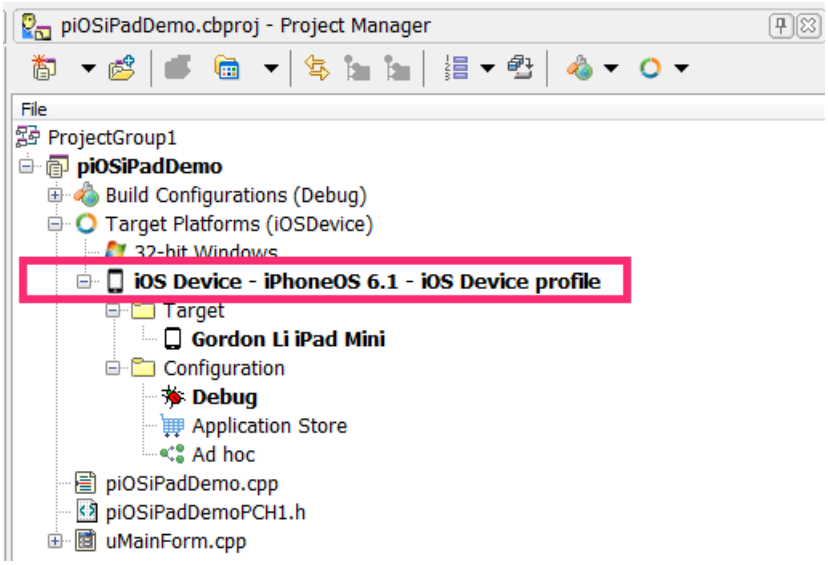
笔者使用 iPad Mini 执行设定程序，于一般 | 描述文件项目就可以看到授权认证信息被分发到笔者的 iPad Mini 中:

现在笔者的 iPad Mini 已经处于『已验证』状态，也代表笔者可以正式使用 C++Builder for iOS 部署和分发 iOS App 了。

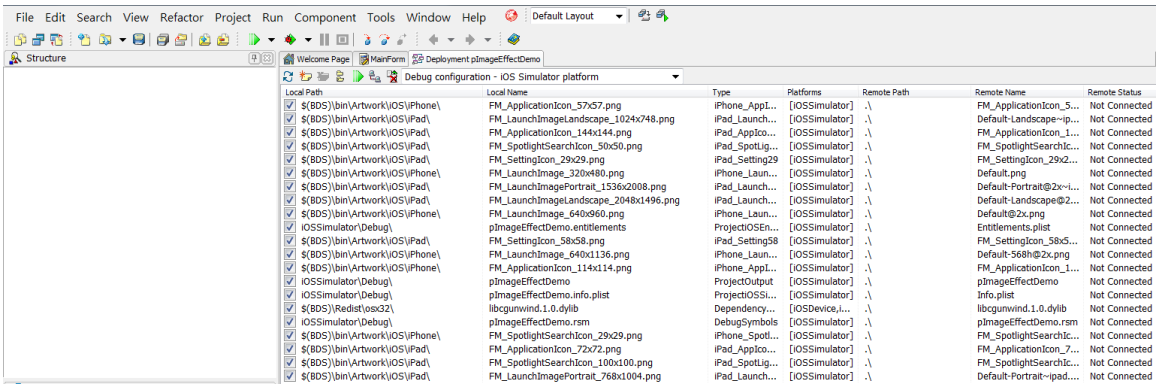
使用 C++Builder for iOS 部署和分发 iOS App 非常的简单，下一节就会说明。

使用部署管理员分发您的 iOS App

对于简单的iOS App项目，读者只需要在,。项目管理员的Target Platforms节点中选择使用 iOS 远程组态，再编译执行 iOS App 项目即可:



当然对于需要随着 iOS App 项目分发的功能和额外的档案，读者需要使用整合发展环境中的分发精灵来帮助分发复杂的 iOS App 项目，读者可以点选 Project|Deployment 菜单来启动分发精灵:



例如对于本节的范例 iPad App 由于只使用 FireMonkey 框架并没有使用数据库功能或是其他额外的档案，因此我们只需要在项目经理中选择 iOS 远程组态，再执行此范例 iPad App 之后，C++Builder for iOS 就会自动编译和分发了

> 笔者是使用 Mac 的 Reflector 软件截取 iPad Mini 的执行画面。如果读者对于 Reflector 有兴趣，可以参考 http://www.reflectorapp.com

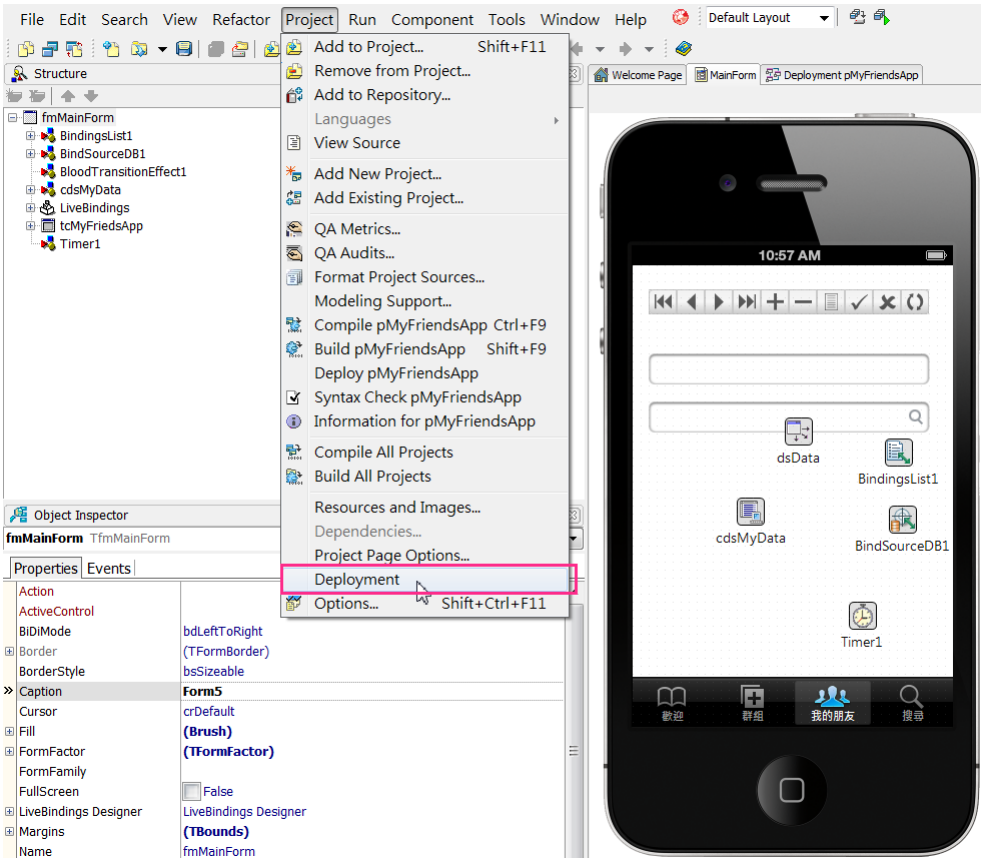
好了，现在读者应该了解了使用 C++Builder for iOS 分发 iOS App 是非常简单的工作了。

7 分发复杂的 iOS App

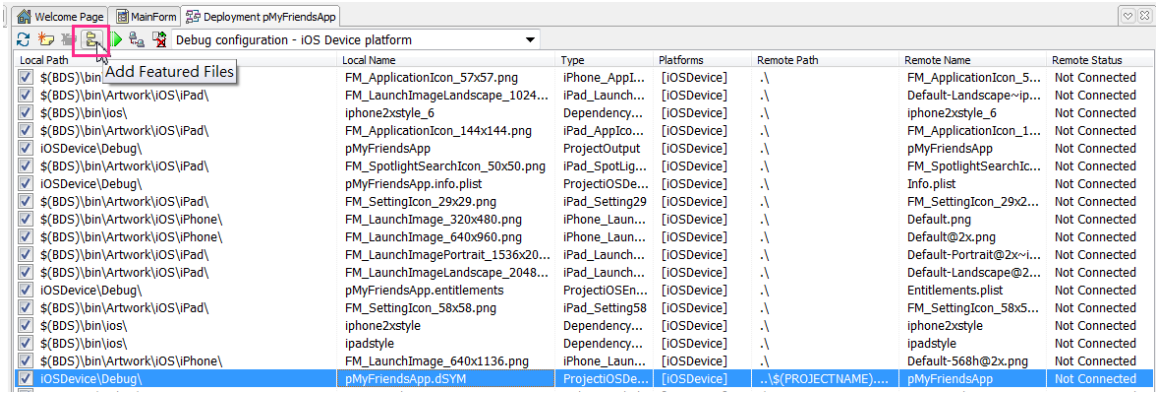
我们即将进入本书最后的小节，讨论如何分发除了 App 本身之前还需要分发其他额外的档案。

请回到前面讨论的 iPhone 范例 App。由于这个范例使用了 DataSnap 技术来处理资料，因此当我们要分发此范例 App 到仿真器中或是到实际的 iOS 设备中时，都需要分发 DataSnap 相关的函式库和档案，这就需要使用整合发展环境中分发管理员的帮忙了。

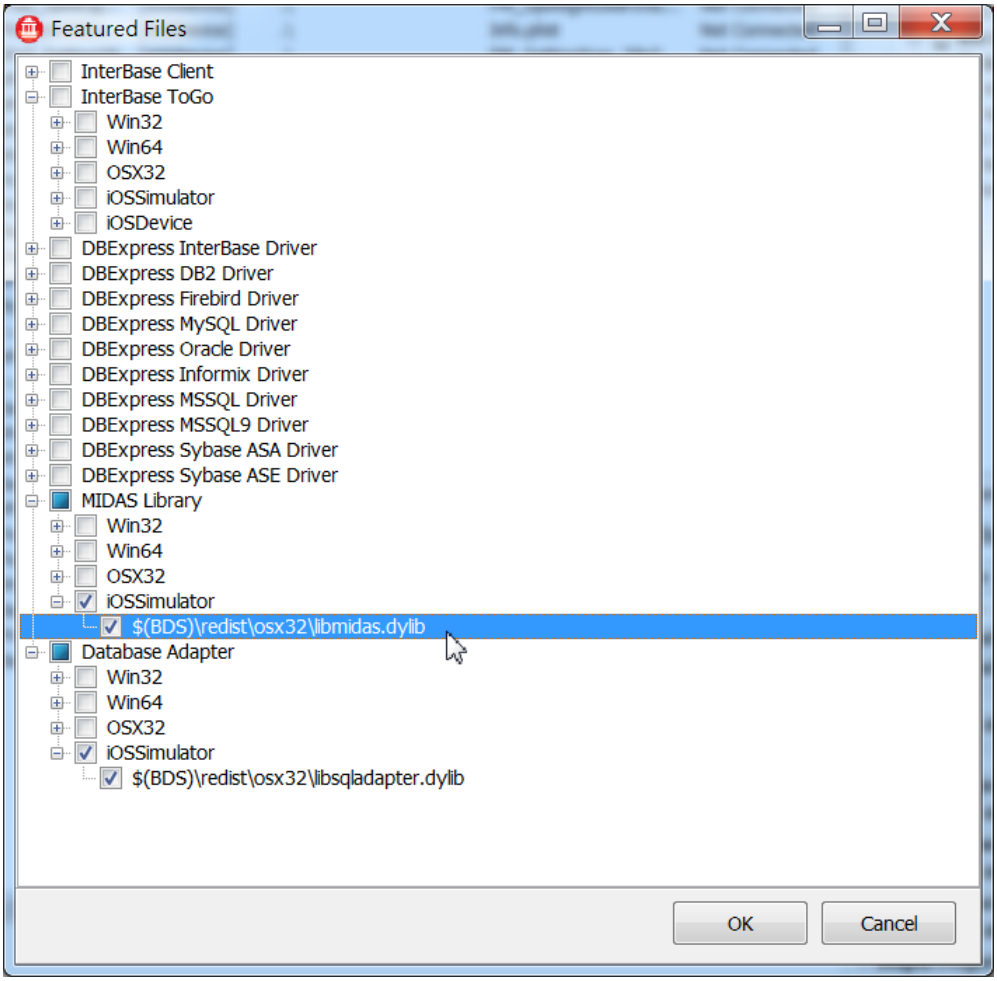
请在整合发展环境中重新开启 pMyFriendsApp 范例 App，然后点选主菜单的 Project|Deployment 启动分发管理员，如下所示:



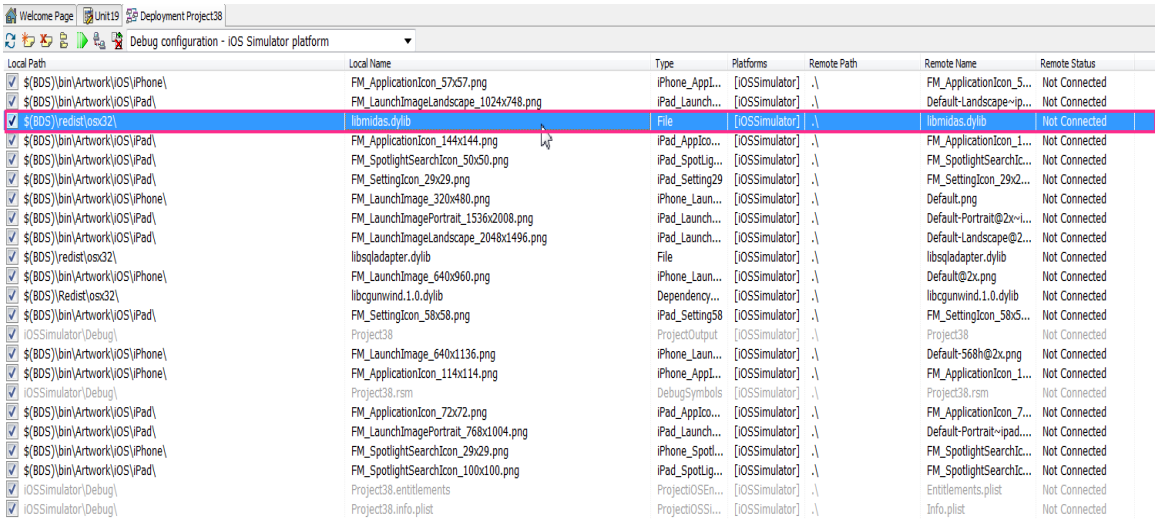
分发管理员接着会显示如下的窗口，显示目前这个范例 App 在分发时所有相关的档案。请点选分发管理员窗口左上方的『Add Featured Files』图像，如下所示：



点选『Add Featured Files』图像之后整合发展环境会显示 Featured Files 对话盒，您可以从其中点选您需要分发的功能档案。例如现在我们需要分发 DataSnap 功能的相关档案，因此请点选 Featured Files 对话盒中的 Midas Library 选项，从其下再勾选 iOSSimulator 项目，在 iOSSimulator 项目下您就可以看到分发管理员会分发『$(BDS)\redist\osx32\libmidas.dylib』档案，如下所示：



接着点选 Featured Files 对话盒的 OK 按钮后，在分发管理员中就会看到 libmidas.dylib 已经加入到分发档案的列表中，如下所示：



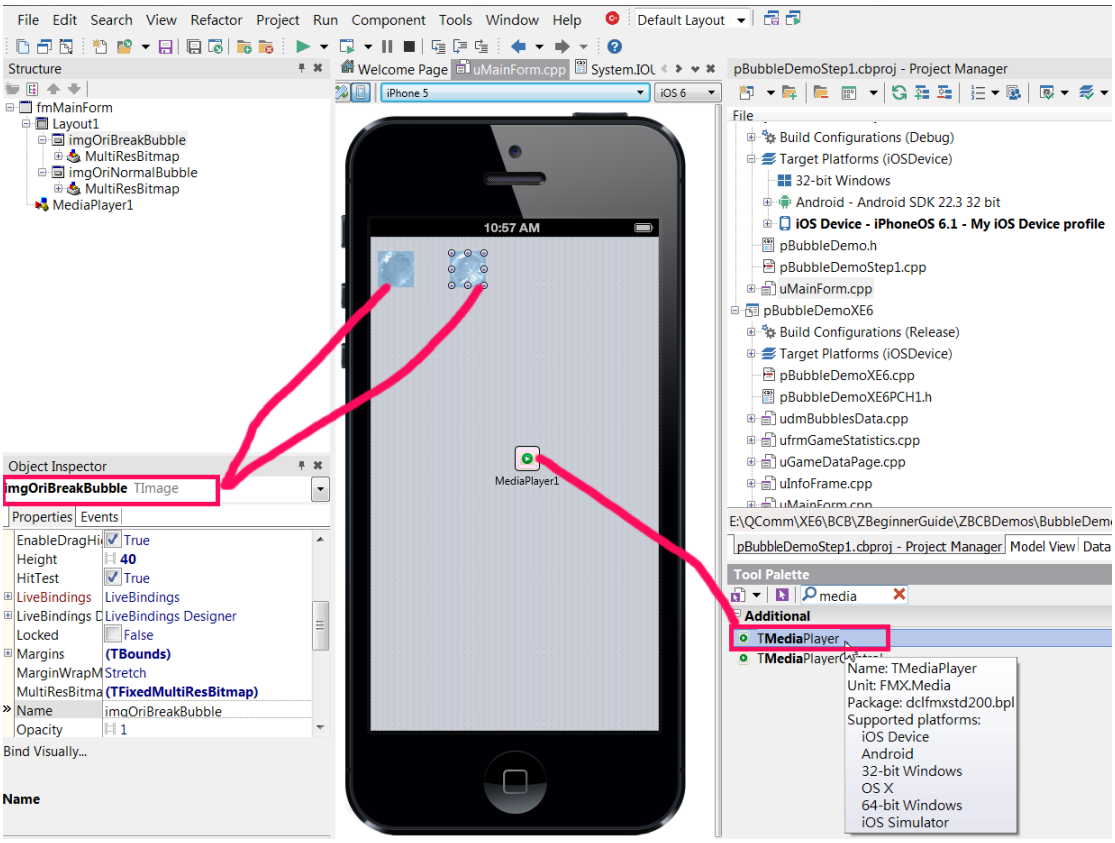
如此就完成了分发额外功能档案的工作，现在请在项目管理员中点选 iOS 设备的组态，编译并且执行此范例 App，那么这个范例 App 应该就可以分发到

您的 iOS 设备中执行了。例如下面的画面就是笔者把这个范例 App 分发到笔者的 iPad Mini 中执行的结果，这个范例 App 成功的在 iPad Mini 中执行，处理数据并且浏览数据指定的网站了。

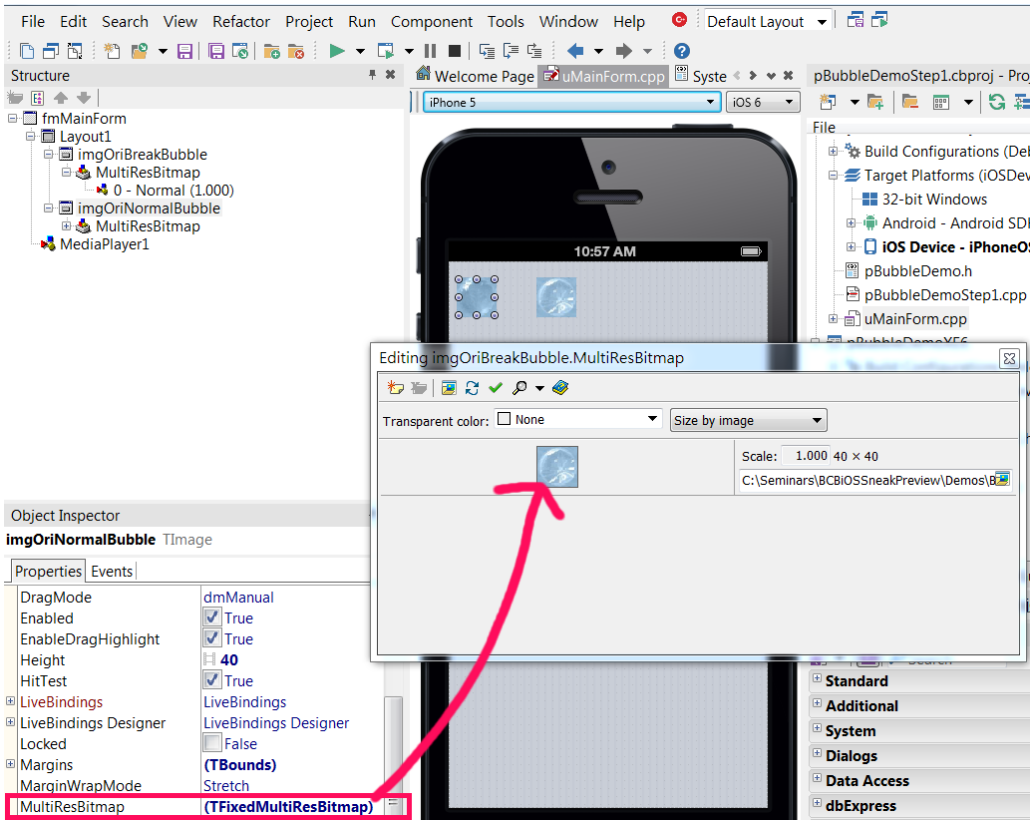
8 用 C++Builder 写个游戏吧

C/C++的长处之一就是快速的执行速度，为了展示 BCB 开发 iOS App 的高生产力，让我们来写个 iOS App 的小游戏吧，这个小游戏就是小时候捏泡泡的游戏，在这个过程中我们也会讨论许多使用 BCB 开发 iOS App 的技巧。

首先在 BCB IDE 中建立一个 FireMonkey Mobile Application 项目，先在主窗体中放入一个 TLayout 组件并且设定它的 Align 特性值为 alClient，接着在主表格的 TLayout 组件中放入一个 TMediaPlayer 组件，二个 TImge 组件如下所示:



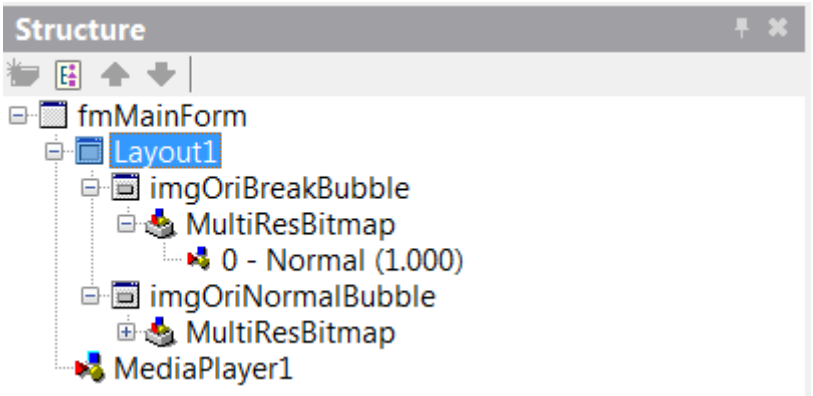
并且在对象查看器中点选 TImage 组件的 MultiResBitmap 特性加载 2 个泡泡图像，如下所示：



并且特定 2 个 TImage 组件的 Name 特性值如下:

Name特性值	说明
imgOriNormalBubble	显示正常泡泡的图像
imgOriBreakBubble	显示捏破泡泡的图像

现在主窗体中所有先件之间的关系如下图显示在 IDE 左上方的架构窗口中:



8-1 让泡泡充满画面吧

在主窗体中的 imgOriNormalBubble 和 imgOriBreakBubble 只是做为显示正常泡泡和捏破泡泡的母图像，这个小游戏首先将使用

imgOriNormalBubble 画满整个画面，之后当玩者点选了画面中任一正常泡泡之后就代表要捏破这个泡泡，那么我们就需要把这个正常泡泡图像改成显示捏破泡泡的图像并且播放一捏破泡泡的声音檔。

第一个工作就是在主窗体的 OnActivate 事件处理及式中呼叫 SetupBubbleSound() 方法设定捏破泡泡声音文件的位置，接着呼叫 SetupBubbles() 方法在主窗体中绘满正常泡泡的图像，最后我们把 imgOriNormalBubble 和 imgOriBreakBubble 隐藏起来：

```
void __fastcall TMainForm::FormActivate(TObject *Sender)
{
  SetupBubbleSound();
  SetupBubbles();
  imgOriNormalBubble->Visible = false;
  imgOriBreakBubble->Visible = false;
}
```

SetupBubbles()方法的工作就是在 011 行呼叫 imgOriNormalBubble 组件的 Clone()方法拷贝正常泡泡图像的对象，012 行设定拷贝正常泡泡图像对象的 Parent 是 Layout1，012/013 行设定这个拷贝正常泡泡图像对象的正确位置，015 行设定它的 OnClick 事件处理及式以便在被点选时切换成被捏破的图像，016 行把拷贝的正常泡泡图像对象暂存在 pBubbles 数组中，最后在代表这个拷贝正常泡泡图像对象是否已被捏破的 BubblePoppedStatus 布尔值数组中设定为 false 以代表目前为正常的状态。


```
001   void TMainForm::SetupBubbles()
002   {
003     if (!bSetup)
004     {
005       TImage *pCloneImage;
006
007       for (int iRow = 0; iRow < IROWS; iRow++)
008       {
009         for (int iCol = 0; iCol < ICOLS; iCol++)
010         {
011           pCloneImage = (TImage *) imgOriNormalBubble->Clone(this);
012           pCloneImage->Parent = Layout1;
013           pCloneImage->Position->X = iCol * 40;
014           pCloneImage->Position->Y = iRow * 40;
015           pCloneImage->OnClick = OnBubbleClick;
```

```
016           pBubbles[iRow][iCol] = pCloneImage;
017           BubblePoppedStatus[iRow][iCol] = false;
018         }
019       }
020       bSetup = true;
021     }
022   }
```

而 BubblePoppedStatus 和 pBubbles 则是宣告在表头档中：

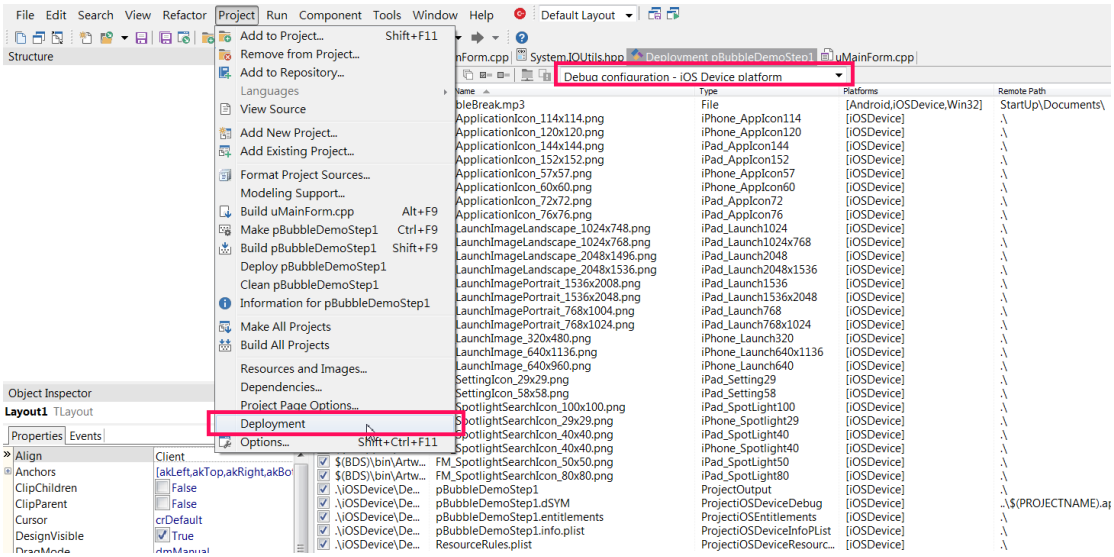
```
    bool BubblePoppedStatus[IROWS][ICOLS];
    TImage* pBubbles[IROWS][ICOLS];
```

SetupBubbleSound()方法的工作则是设定捏破泡泡时要播放的声音文件位置，稍后我们会说明如何使用 IDE 部署声音档，因此 SetupBubbleSound()方法就是设定主窗体中 TMediaPlayer 组件加载声音文件正确的路径。由于部署 iOS App 时只能把 App 需要使用的其他档案部署在 App 的沙盒中而不能随意部署 App 要使用的额外档案，因此对于像本范例使用的声音文件，我们应该把它部署在 App 的 Documents 目录中。

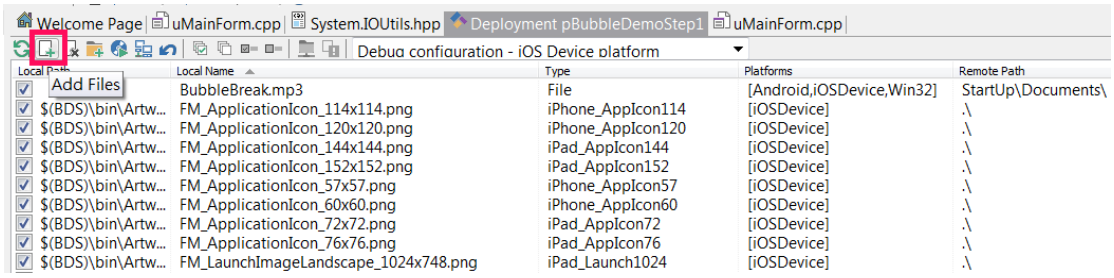
因此在 SetupBubbleSound()方法的 003 行我们可呼叫 GetHomePath()方法取得 iOS App 本身的根目录，接着再于它的 Documents 目录下加载声音文件"BubbleBreak.mp3"，004 行判断如果声音档"BubbleBreak.mp3"存在的话就设定 TMediaPlayer 组件的 FileName 特性值为此声音文件，否则就显示一警告讯息：


```
001   void TMainForm::SetupBubbleSound()
002   {
003     String sFileName =System::Ioutils::TPath::GetDocumentsPath() +
PathDelim + "BubbleBreak.mp3";
004     if (FileExists(sFileName))
005     MediaPlayer1->FileName = sFileName;
006     else
007     ShowMessage(sFileName  + "Not Exist");
008   }
```

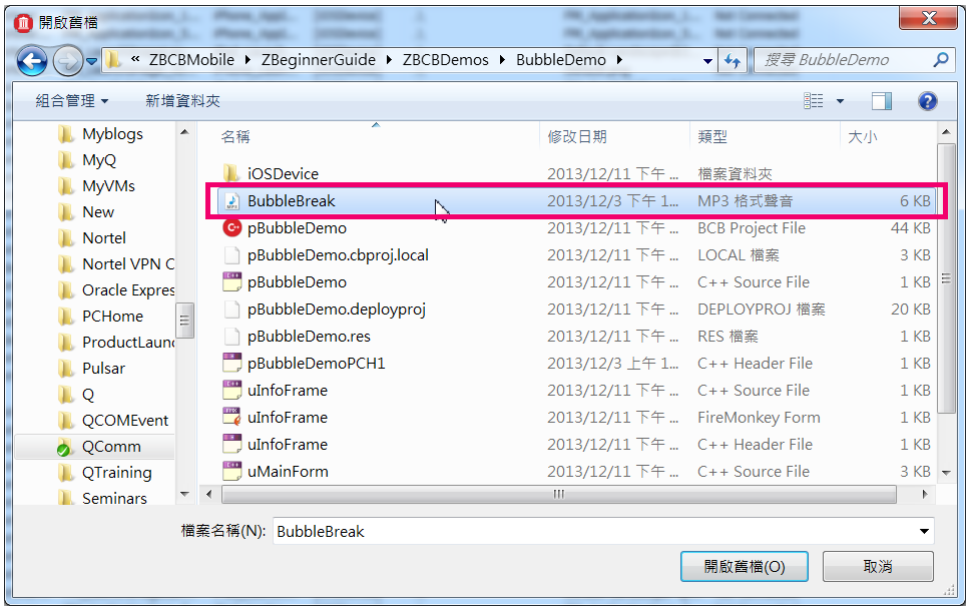
好了，到这里此范例 App 就可以先执行看看了，但在执行之前我们需要连同声音档"BubbleBreak.mp3"一起部署到 iPhone 5 手机中。请在 IDE 中点选 Project|Deployment 选单，IDE 会启动部署精灵，如下所示：



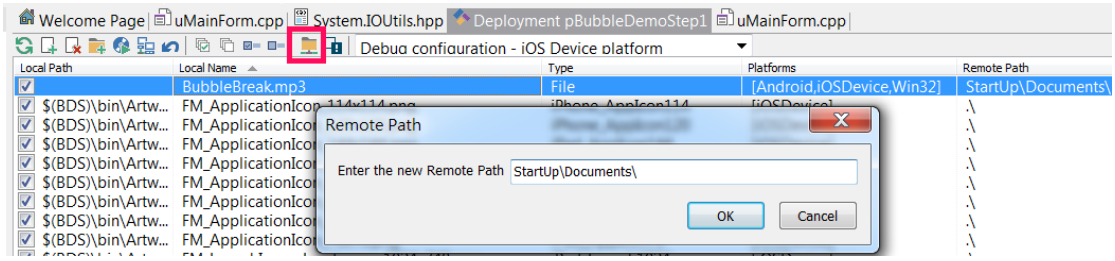
在部署精灵中点选左方上的『Add Files』按钮  以便加入声音档"BubbleBreak.mp3"：



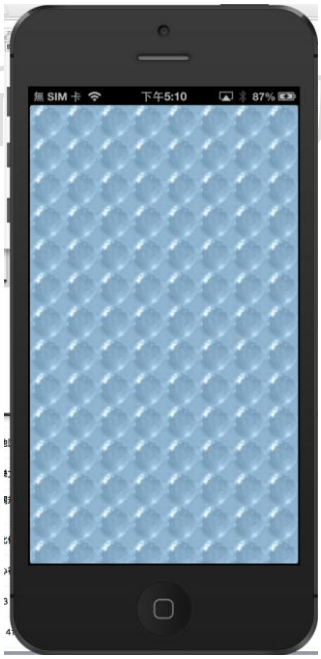
部署精灵此时会显示开启旧文件对话盒让您加载声音档：



加入了声音档"BubbleBreak.mp3"之后请点选部署精灵的『Change Remote Path For Selected Items』按钮 把声音文件的远程部署位置设定为『StartUp\Documents\』，如下所示：



现在请确定您的 iPhone 或是 iPad/iPad Mini 手机已经藉由 USB 连结到 Mac 机器上，那么请在 IDE 中编译和执行此范例 App，之后就应该可以在 iPhone 手机中看到如下成功执行的画面了：



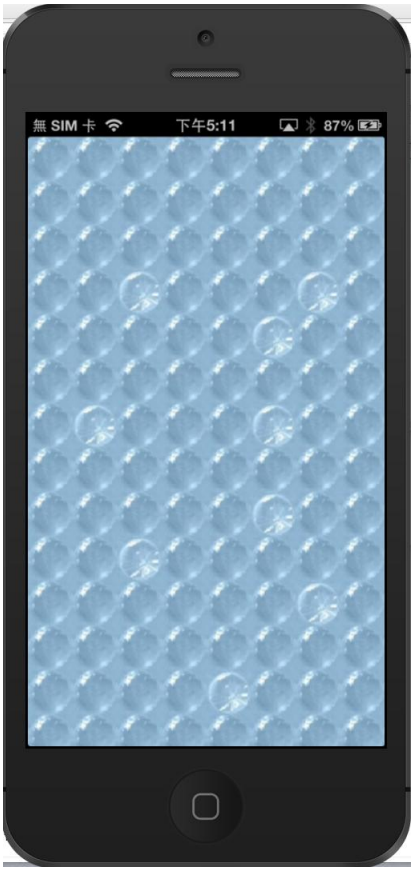
8-2 点选捏破泡泡

当用户点选主窗体中的正常泡泡图像时就会执行前面设定的 OnBubbleClick 事件处理及式，在 OnBubbleClick 中我们需要把正常泡泡图像改成捏破泡泡的图像并且播放前面部署的声音档"BubbleBreak.mp3"。因此在下面的 OnBubbleClick 事件处理中在 003/004 行我们先算出是那一个正常泡泡图像被点选，007 行把声音文件的播放位置重置回开始的位置，008 行判断

如果目前被点选的泡泡图像是正常泡泡图像的话就把目前传入的 Sender 参数的型态转换成 TImage*型态，再从 imgOriBreakBubble 中加载捏破泡泡的图像，最后再 014 行藉由 TMediaPlayer 组件播放声音文件"BubbleBreak.mp3"：

```
001   void __fastcall TMainForm::OnBubbleClick(TObject *Sender)
002   {
003     int iRow = ((TImage *) Sender)->Position->Y / BUBBLESIZE;
004     int iCol= ((TImage *) Sender)->Position->X / BUBBLESIZE;
005
006     MediaPlayer1->Stop();
007     MediaPlayer1->CurrentTime = 0;
008     if (!BubblePoppedStatus[iRow][iCol])
009     {
010       BubblePoppedStatus[iRow][iCol] = true;
011       ((TImage *)
Sender)->MultiResBitmap->Assign(imgOriBreakBubble->MultiResBitmap);
012       MediaPlayer1->Play();
013     }
014   }
```

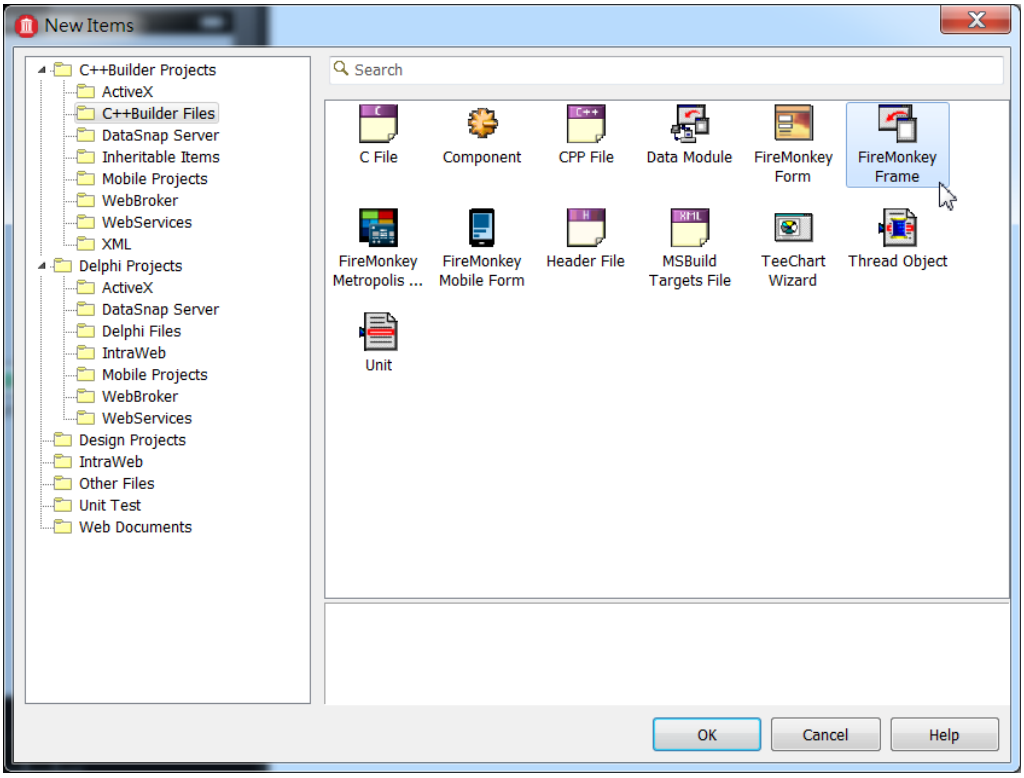

再次编译和执行此范例 App 并且随意点选泡泡就可以看到如下的画面被点选的泡泡就会显示被捏破并且会听到泡泡被捏破的声音了。



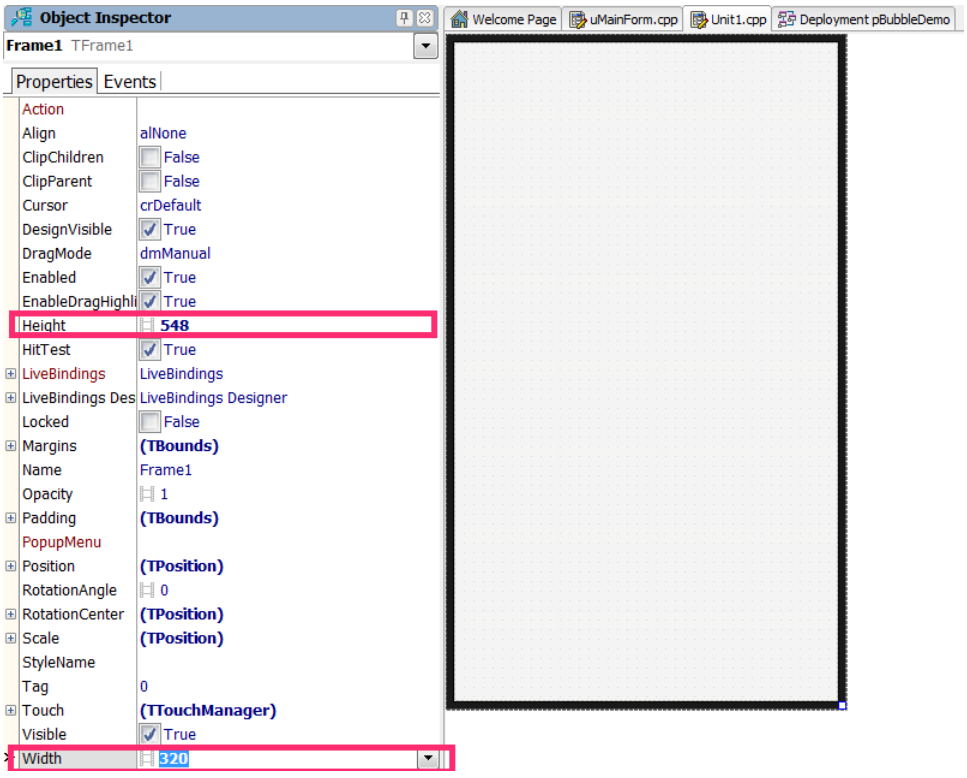
8-3 加入手势功能

接下来让我们在这个范例 App 中加入使用手势的功能，当玩家在主窗体中使用手势向左方滑动时，让我们藉由 FireMonkey 的动画功能动态的显示一个此游戏的关于信息，同时让我们说明如何使用 FireMonkey 的 Frame 功能来显示关于信息。

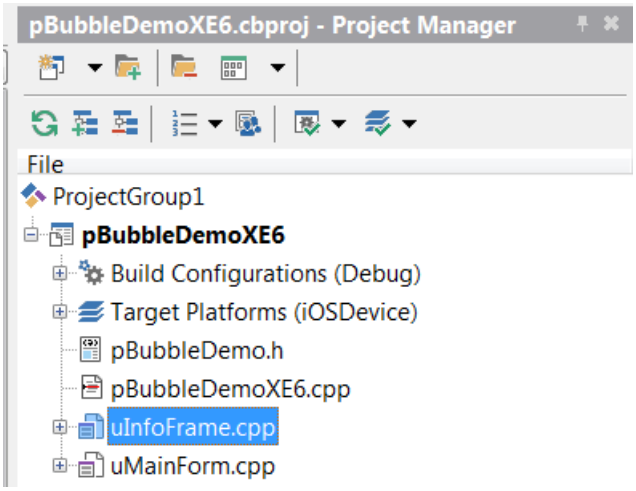
现在请在 IDE 中点选 New Items 快捷键于 New Items 对话盒中 C++Builder Files 项目中选择建立 FireMonkey Frame 对象，如下图所示：



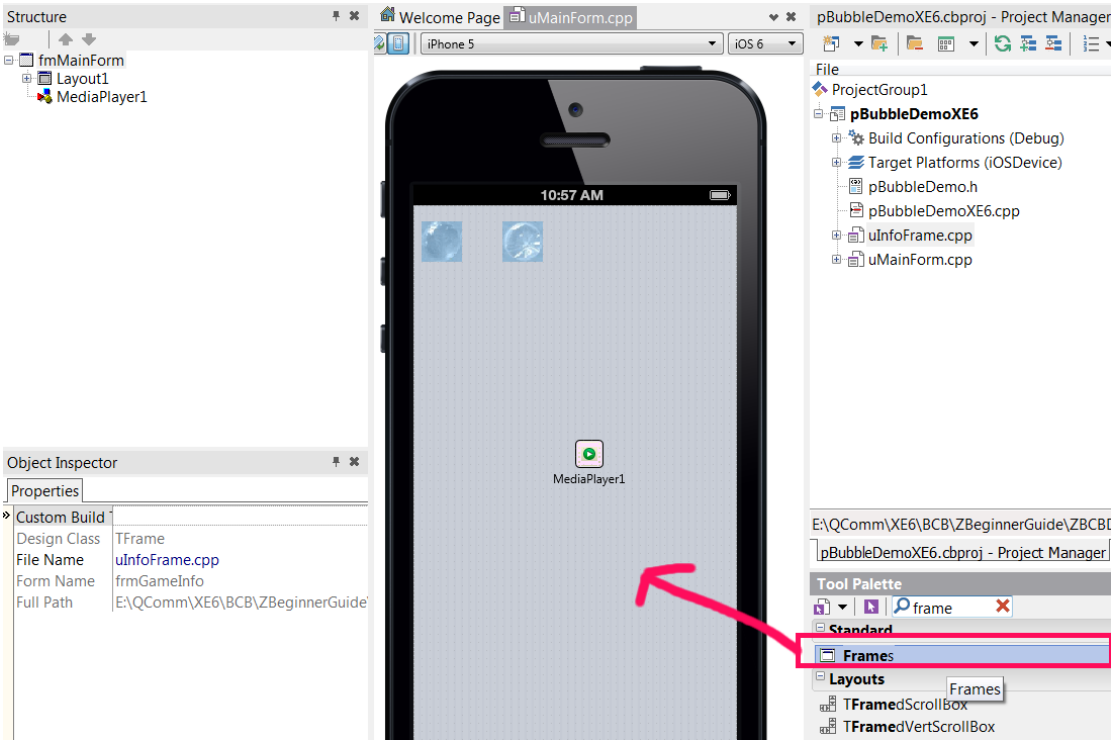
点选建立 FireMonkey Frame 对象之后 IDE 便会建立一个空白的 Frame 窗体，请在对象查看器中设定它的 Height 和 Width 特性值和主窗体中 Layout 组件一样的 Height 和 Width 特性值，如下所示：



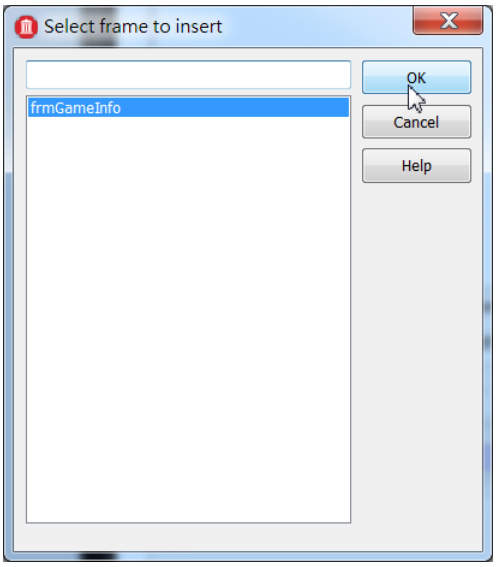
最后设定它的 Name 特性值为 frmGameInfo 并且以 uInfoFrame 储存这个 FireMonkey Frame 对象，此时项目管理员的内容应如下所示：



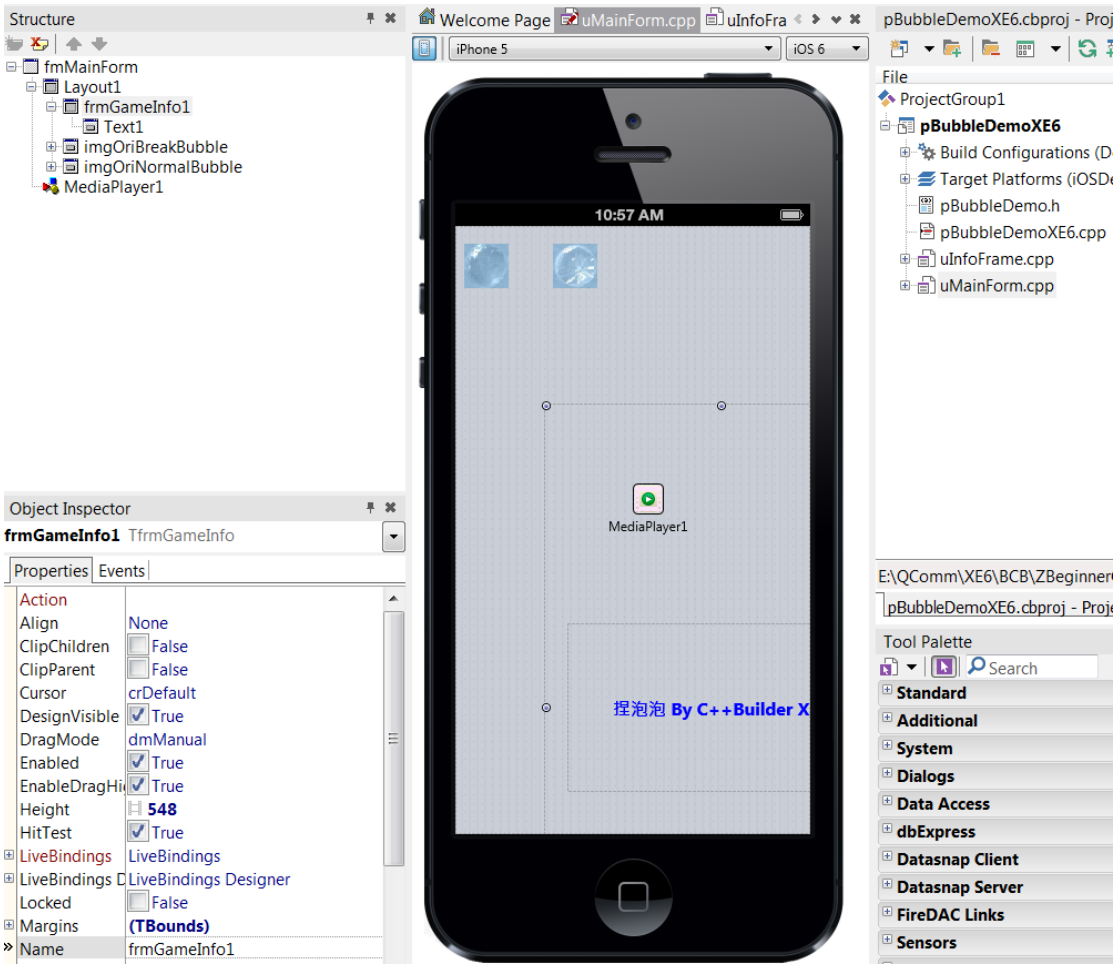
接着回到项目中的主窗体，在工具盘中找到 Frame 组件并且把它拖曳到主窗体中如下所示：



当您拖曳 Frame 组件到主窗体后 IDE 便会显示如下的对话盒询问您这个 Frame 组件要使用的 FireMonkey Frame 对象是什么，由于在前面我们已经建立了 frmGameInfo，因此就请在对话盒中选择 frmGameInfo，如下所示：

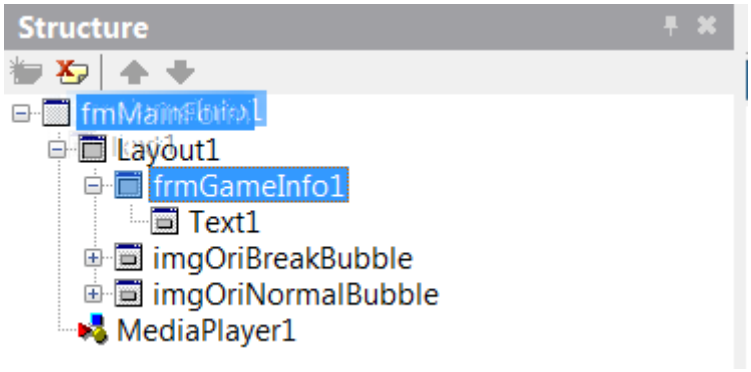


在上面的对话盒中选择了 frmGameInfo 之后我们就可以在主窗体中看到 frmGameInfo 也显示在主窗体中了，如下所示：

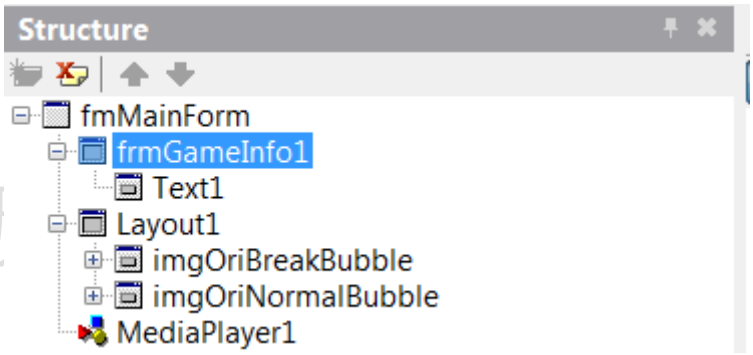


现在请在 IDE 左上方的架构窗口中查看 frmGameInfo 的父代组件是什么，如果 frmGameInfo 是在 Layout1 组件之下就代表 Layout1 是它的父代组件，

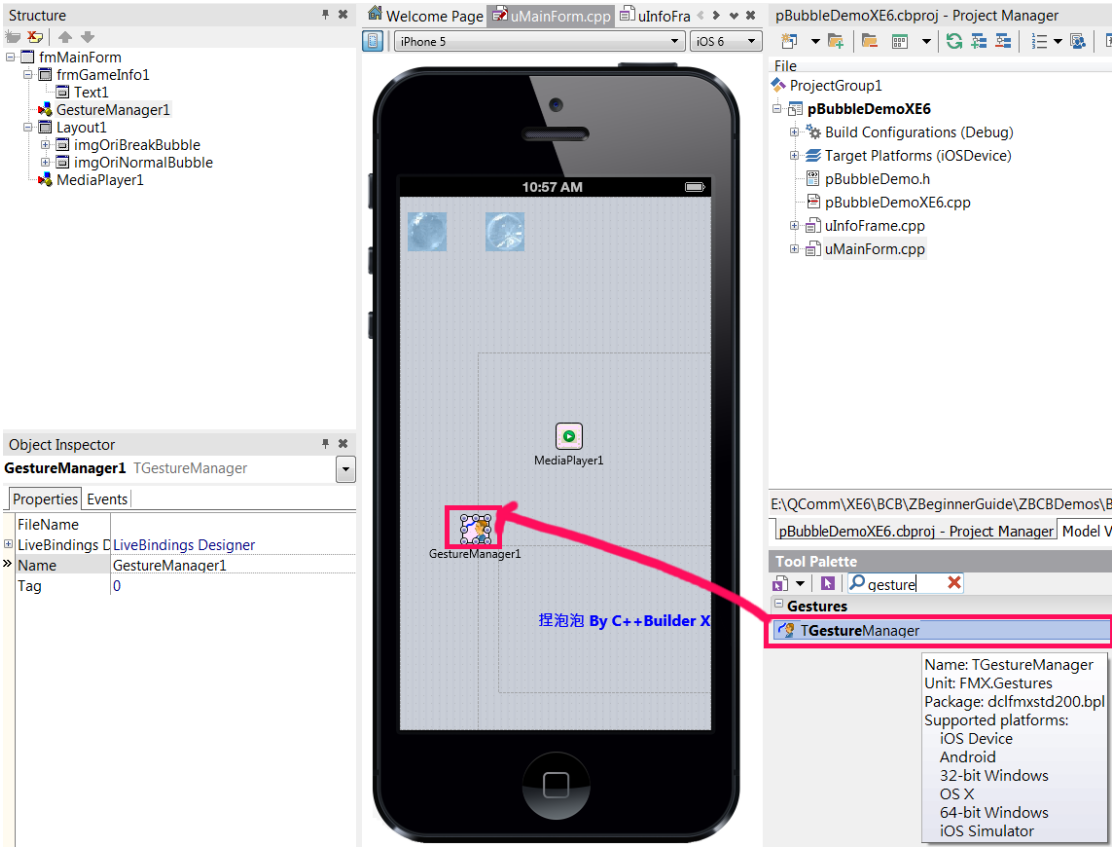
那么就请在架构窗口中点选 frmGameInfo，在保持按着鼠标左键的状况下拖曳 frmGameInfo 到 MainForm 之下再释放鼠标左键让 MainForm 成为 frmGameInfo 的父代组件，如下所示：



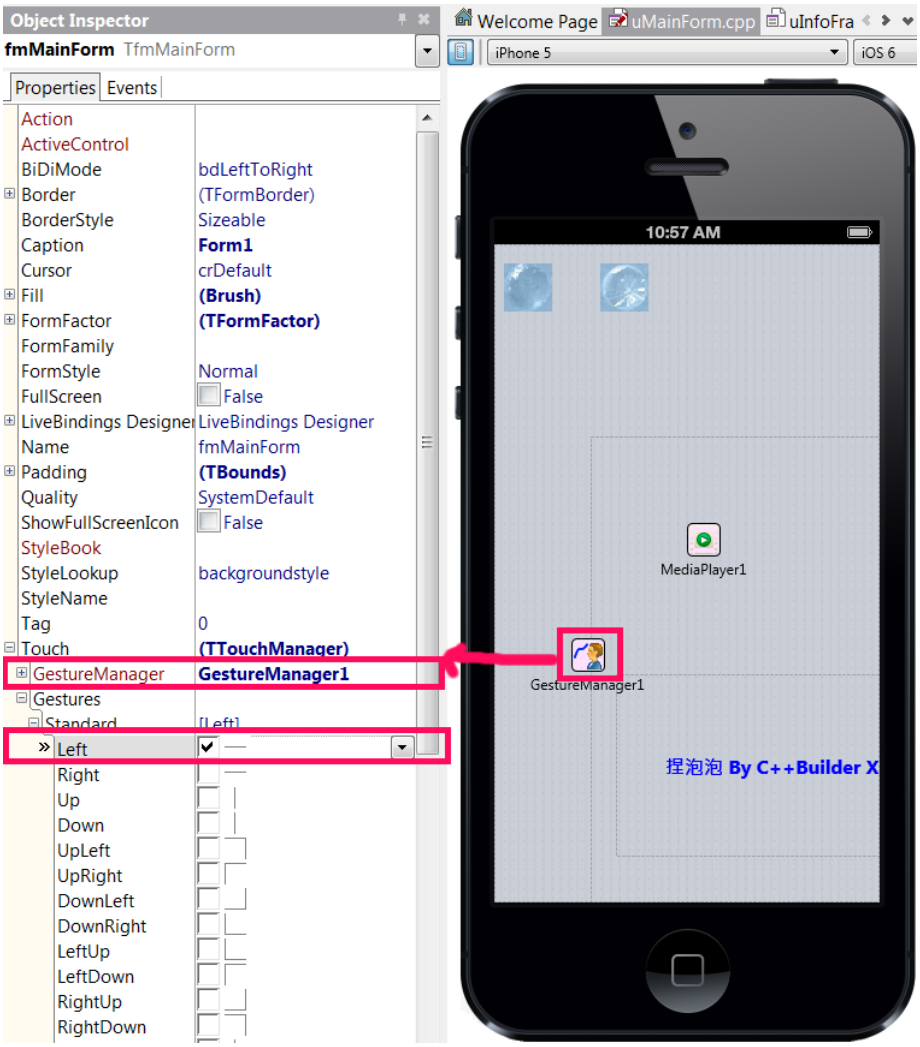
拖曳完成之后架构窗口应该如下所示：



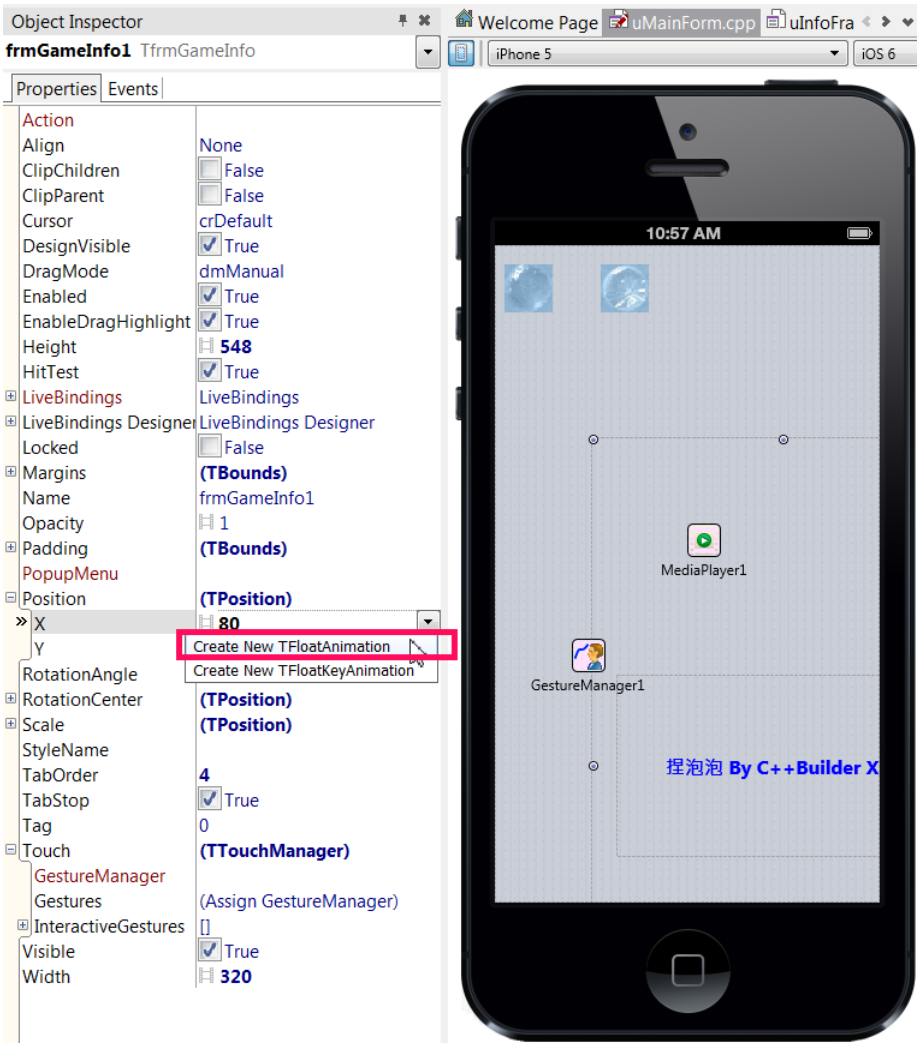
好了，接下来再于主窗体中拖入 TGestureManager 组件准备加入处理手势的功能，如下所示：



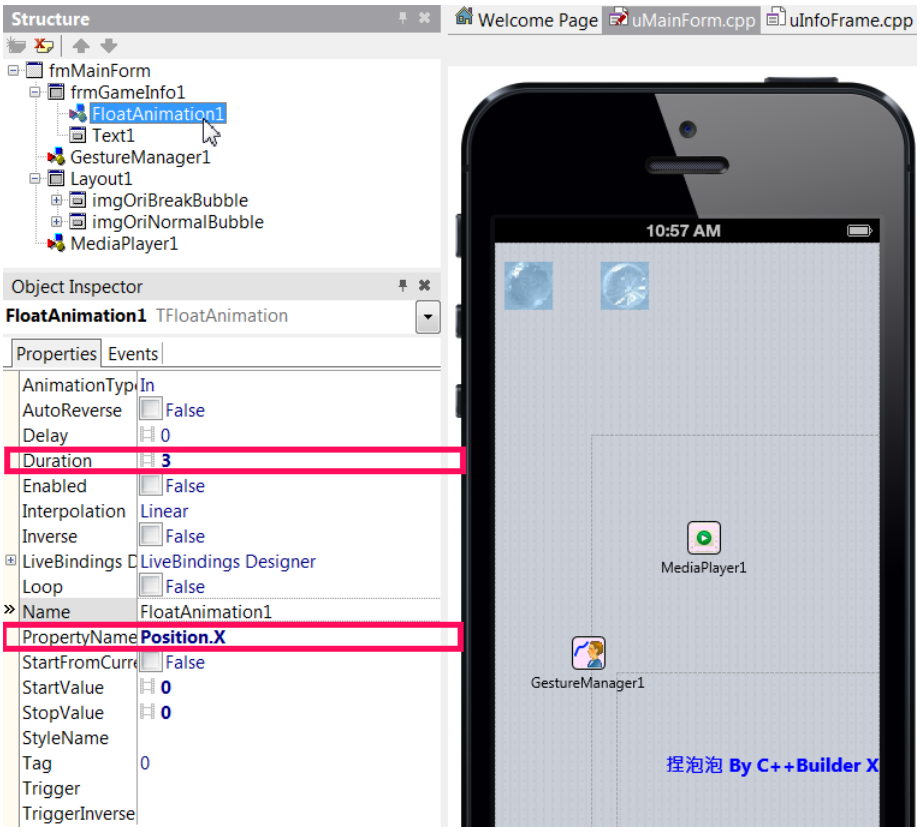
在对象查看器中设定 MainForm 的 GestureManager 特性值为刚才拖入的 TGestureManager 组件，展开 MainForm 的 Gestures 特性在它的 Standard 子特性中勾选 Left 子特性代表主窗体 MainForm 将处理向左滑动的手势，如下所示：



为了让 frmGameInfo 对象在玩家向左滑动手势后动态的显示出来，让我们为 frmGameInfo 对象的 X 轴特性加入一个动画的功能。请点选主窗体中的 frmGameInfo，在对象查看器中点选它的 Position 特性下的 X 子特性，在 X 子特性的下拉盒中选择建立一个 TFloatAnimation 对象，如下所示：



在 IDE 左上方的架构窗口中点选新加入的 FloatAnimation1 对象，再于对象查看器中设定它的 Duration 特性为 3 代表这个动画将持续 3 秒的时间，如下所示：



再把此此 FloatAnimation1 对象的 Name 特性值更改为 faniFrameX。

现在就可以撰写处理向左滑动的手势了，点选主窗体的 MainForm 对象在对象查看器中建立它的 OnGesture 事件处理函式，并且在其中撰写如下的程序代码：

```
001    void __fastcall TMainForm::FormGesture(TObject *Sender, const
TGestureEventInfo &EventInfo,
002          bool &Handled)
003    {
004      if (EventInfo.GestureID.get() == sgiLeft )
005      {
006          faniFrameX->Enabled = false;
007          faniFrameX->StartValue = Layout1->Width;
008          faniFrameX->StopValue = 0;
009          faniFrameX->Enabled = true;
010          Handled = true;
011      }
012    }
```

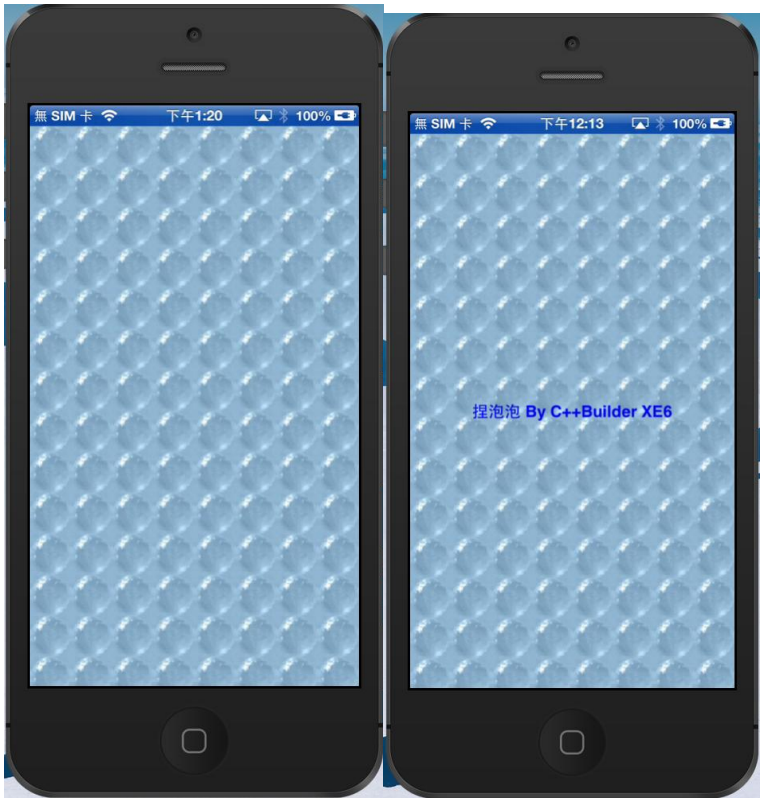
当玩家在手机屏幕做出向左滑动的手势后 TGestureManager 就会触发 MainForm 的 OnGesture 事件处理函式。OnGesture 事件处理函式的第 2 个参数 TGestureEventInfo 包含此手势的相关信息，在 TGestureEventInfo 中的 GestureID 变量代表用户做出的手势种类，因此我们只要判断 GestureID 就可以知道玩家是不是做出了左向滑动的手势。

FireMonkey 框架以 sgiLeft 常数代表左向滑动的手势，因此在上面的 004 行判断玩家做出的手势是不是 sgiLeft，如果是的话就设定 faniFrameX 动画对象开始执行的位置在 X 轴 Lauout1 的宽度处，也就是手机屏幕的最右方，向手机屏幕的最左方向出现，设定好动画的方向和位置之后 009 行就启动 faniFrameX 动画对象开始执行。

完成了 OnGesture 事件处理函式之后我们需要在此 App 启动时先把 frmGameInfo 对象设定在屏幕的最右方，如此一来 faniFrameX 动画对象才能正确的工作。因此我们需要在 MainForm 的 OnActivate 事件处理函式中加入一行程序代码呼叫 SetupGameInfoFrame()方法，而它的工作就是设定 frmGameInfo 对象的正确启动位置：

```
void TfmMainForm::SetupGameInfoFrame()
{
  frmGameInfo1->Position->X = Layout1->Width;
  frmGameInfo1->Position->Y = 0;
}
```

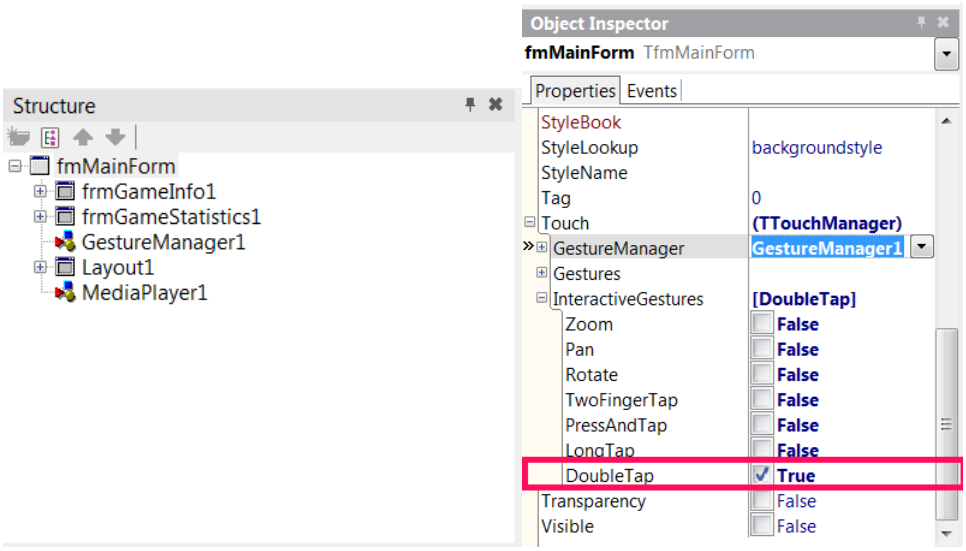
现在编译和执行此范例 App 并且随意点选泡泡，再使用手指向手机屏幕的左方滑动，就可以看到 frmGameInfo 对象从右向左慢慢的滑动出现了，如下所示：



8-4 重新开始游戏

现在再让我们为这个小游戏加入重新开始的功能，当玩家想重新开始时只需要在游戏中轻点 2 次屏幕就可以让游戏回到起始的状态，要如此做我们回需要再让这个 App 支持 DoubleTap 手势即可。

回到 MainForm 在它的 GestureManager|InteractiveGestures 特值中勾选 DoubleTap 选项代表 App 要处理用户在 App 轻点 2 次屏幕的手势：



接着在 MainForm 的 FormGesture 事件处理函式中加入如下处理 DoubleTap 手势的程序代码：

```
      if (EventInfo.GestureID.get() == igiDoubleTap )
      {
        ResetGame();
      }
```

而 ResetGame()方法只是呼叫 SetupGameInfoFrame()重新设定 2 个 TFrame 对象的位置，再把画面中已经被捏破的泡泡恢复原状而已：

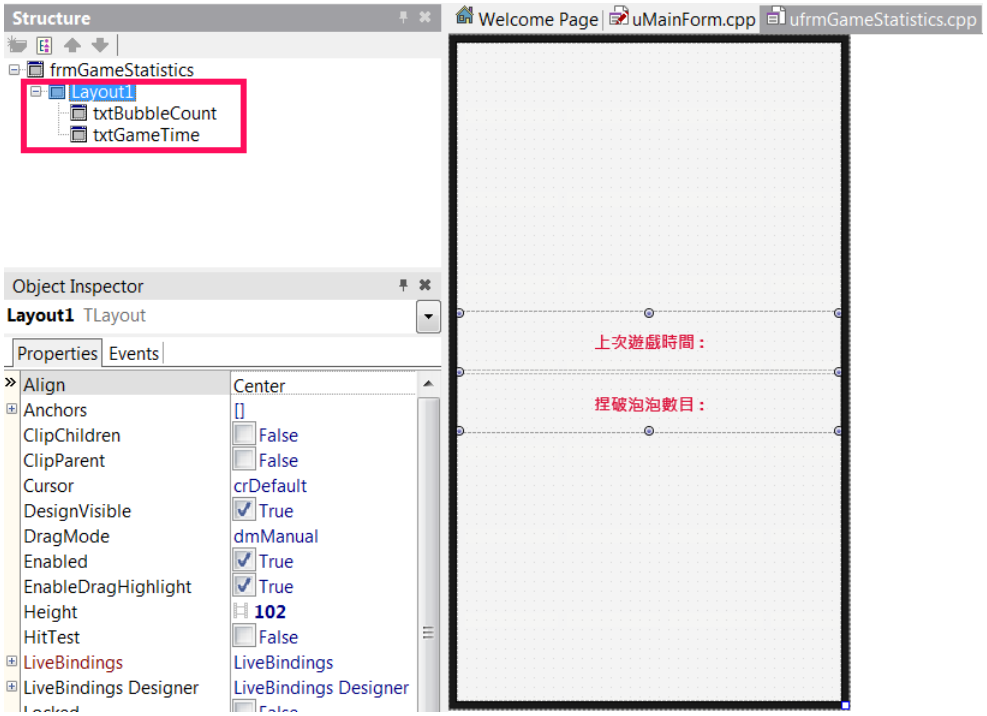
```
void TfmMainForm::ResetGame()
{
  SetupGameInfoFrame();
  for (int iRow = 0; iRow < IROWS; iRow++)
  {
    for (int iCol = 0; iCol < ICOLS; iCol++)
    {
      if (BubblePoppedStatus[iRow][iCol])
      {

    pBubbles[iRow][iCol]->MultiResBitmap->Assign(imgOriNormalBubble->
MultiResBitmap);
        BubblePoppedStatus[iRow][iCol] = false;
      }
    }
  }
}
```

现在再次执行此范例 App，试着捏破一些泡泡再轻点 2 次屏幕，您会发现此范例 App 又恢复到了原始的执行状态了。

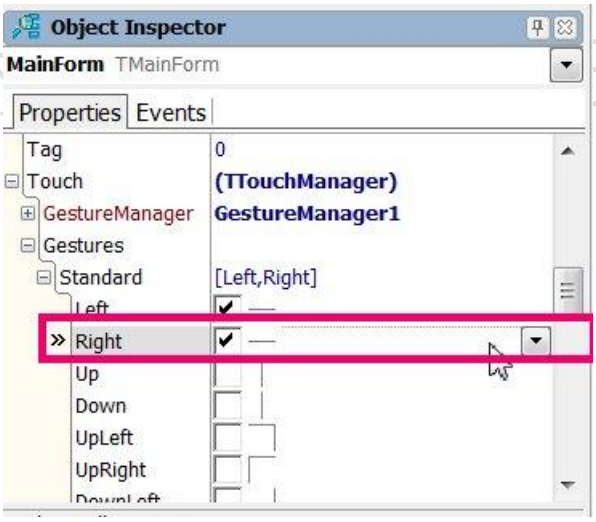
8-5 处理游戏信息

现在让我们试着把玩家每次玩捏泡泡的信息储存起来，并且让玩家能够使用向右的手势来显示上次玩游戏的信息。

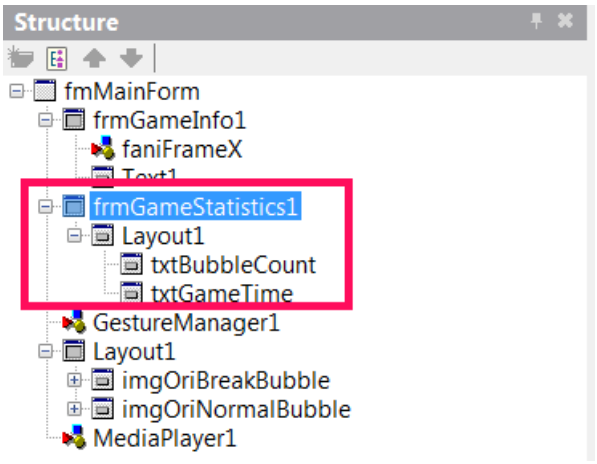
再次让我们使用一个 Frame 对象来显示游戏的信息，请使用前面介绍的方式再建立项目中第 2 个 FireMonkey Frame 对象，设定它的 Height 和 Width 特性和主窗体中的 Layout1 一样，再于其中放入一个 TLayout 组件再于其中放入 2 个 Text 对象并且设定 TLayout 组件的 Align 特性为 alCenter，设定 Name 特性为 frmGameStatistics，如下所示：



回到主窗体让主窗体支持由左向右滑动的手势：

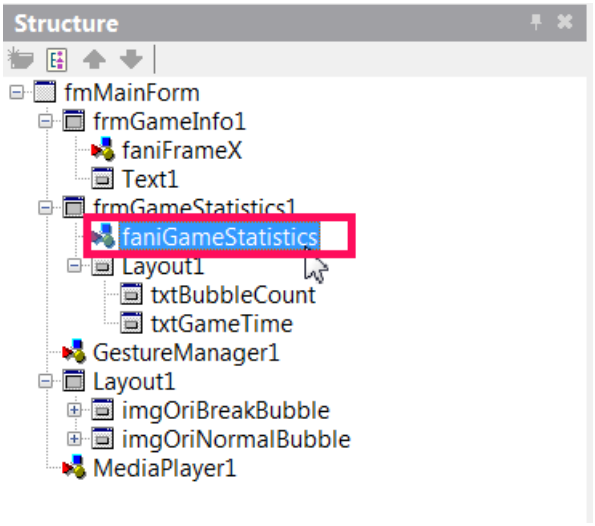


在 IDE 左上方的架构窗口中拖曳 frmGameStatistics 的父代组件为 MainForm，如下所示：



此时主窗体就会出现第 2 个 Frame 对象 frmGameStatistics，如下所示：

接着如同前面为 frmGameInfo 对象的 X 轴特性加入一个动画的功能，现在也请为 frmGameStatistics 对象的 X 轴特性加入一个动画的功能对象 faniGameStatistics：



修改 SetupGameInfoFrame() 方法，把第 2 个 Frame 对象 frmGameStatistics1 的起始位置设定为手机屏幕最左方之外：

```
void TMainForm::SetupGameInfoFrame()
{
  frmGameInfo1->Position->X = Layout1->Width;
  frmGameInfo1->Position->Y = 0;
  frmGameInfo1->Parent = this;

  frmGameStatistics1->Position->X = -Layout1->Width;
  frmGameStatistics1->Position->Y = 0;
  frmGameStatistics1->Parent = this;
}
```

WriteGameInfo()方法则是使用 TIniFile 对象把上次游戏的时间以及上次玩家捏破了多少个泡泡的数目写到名为"BubbleGameInfo.ini"的档案中，请注意"BubbleGameInfo.ini"也必须储存在这个 App 沙盒的 Documents 目录下。

因此在下面的 010 行先取得游戏信息组态档案的正确路径之后 011 行便根据游戏信息组态档案建立 TIniFile 对象，然后呼叫 TIniFile 对象的 WriteString() 方法把现在的时间和计算出来的捏破泡泡的数目写入这个档案中：

```
001    const String sBubbleGameInfo = "BubbleGameInfo.ini";
002
003    String TMainForm::GetGameInfoFile()
004    {
005      return System::Ioutils::TPath::GetDocumentsPath() + PathDelim +
sBubbleGameInfo;
```

```
006   }
007
008   void TMainForm::WriteGameInfo()
009   {
010     String sGameInfoFile = GetGameInfoFile();
011     TIniFile *pConfigFile = new TIniFile(sGameInfoFile);
012     try
013     {
014       pConfigFile->WriteString("游戏", "时间",
DateTimeToStr(Now()));
015       pConfigFile->WriteString("游戏", "捏破泡泡数目",
IntToStr(GetBrokenBubles())) ;
016     }
017     __finally
018     {
019       delete pConfigFile;
020     }
021   }
022
023   int TMainForm::GetBrokenBubles()
024   {
025     int iResult = 0;
026
027     for (int iRow = 0; iRow < IROWS; iRow++)
028     {
029     for (int iCol = 0; iCol < ICOLS; iCol++)
030       if (BubblePoppedStatus[iRow][iCol])
031         iResult++;
032     }
033
034     return iResult;
035   }
```

当然我们还需要加入处理玩家由左向右滑动的手势，因此我们需要修改 MainForm 的 OnGesture 事件处理函式，请在 OnGesture 中加入如下的程序代码：

```
       if (EventInfo.GestureID.get() == sgiRight)
       {
```

```
    frmGameStatistics1->LoadGameStatistics(GetGameInfoFile());
            faniGameStatistics->Enabled = false;
            faniGameStatistics->StartValue = -Layout1->Width;
            faniGameStatistics->StopValue = 0;
            faniGameStatistics->Enabled = true;
            Handled = true;
        }
```

在上面的程序代码中先判断是由左向右滑动的手势的话就先呼叫frmGameStatistics的LoadGameStatistics()方法从游戏信息组态档案中读出上次储存的信息。

现在在 IDE 中开启第 2 个 Frame 对象 frmGameStatistics，并且在其中加入 LoadGameStatistics()方法并实作程序代码如下：

```
001   void TfrmGameStatistics::LoadGameStatistics(const String
sGameInfoFile)
002   {
003     if (FileExists(sGameInfoFile))
004     {
005         TIniFile *pConfigFile = new TIniFile(sGameInfoFile);
006         String sDT;
007         String sBubbles;
008         try
009         {
010           sDT = pConfigFile->ReadString("游戏", "时间", "");
011           sBubbles = pConfigFile->ReadString("游戏", "捏破泡泡数目",
"") ;
012         }
013         __finally
014         {
015           delete pConfigFile;
016         }
017
018         txtGameTime->Text = "上次游戏时间 : " + sDT;
019         txtBubbleCount->Text = "捏破泡泡数目 : " + sBubbles;
020     }
021     else
022     {
```

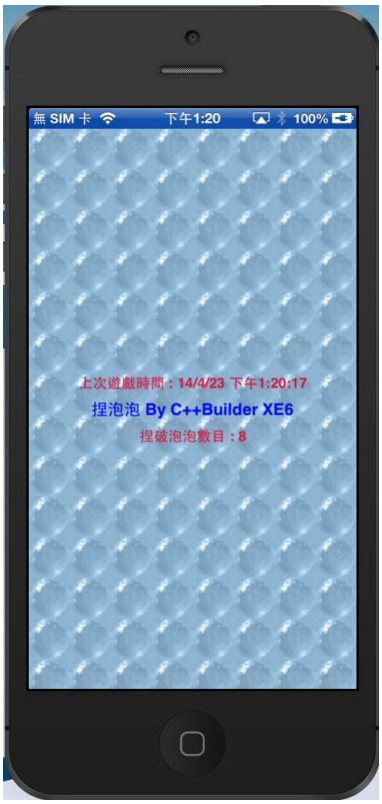
```
023          txtGameTime->Text = "上次游戏时间 : 无";
024          txtBubbleCount->Text = "捏破泡泡数目 : 0";
025
026      }
027    }
```

LoadGameStatistics()同样使用 TIniFile 对象从游戏信息组态档案中读出上次储存的游戏时间和捏破泡泡的数目并且显示在 Frame 的 2 个 Text 组件中。

最后当然要在 ResetGame()方法中加入呼叫 WriteGameInfo()方法把上次捏破泡泡的数目等信息写入 ini 档案中：

```
void TfmMainForm::ResetGame()
{
  WriteGameInfo();
  SetupGameInfoFrame();
  …
```

现在编译和执行此范例 App 并且随意点选泡泡，再摇动手机重新开始游戏并且储存游戏信息，使用手指向手机屏幕的左方滑动，再用手指向手机屏幕的右方滑动，就可以看到 2 个 Frame 对象从右向左以及从左向右慢慢的滑动出现了，如下所示：



到此我们已经成功的说明了如何在 iOS App 中储存信息到档案的方法了，但目前只能储存一次的信息，可不可以储存任意数目的信息呢？当然可以，让我们顺便说明如何在 iOS App 中使用数据库的功能以便用数据库来储存信息吧。

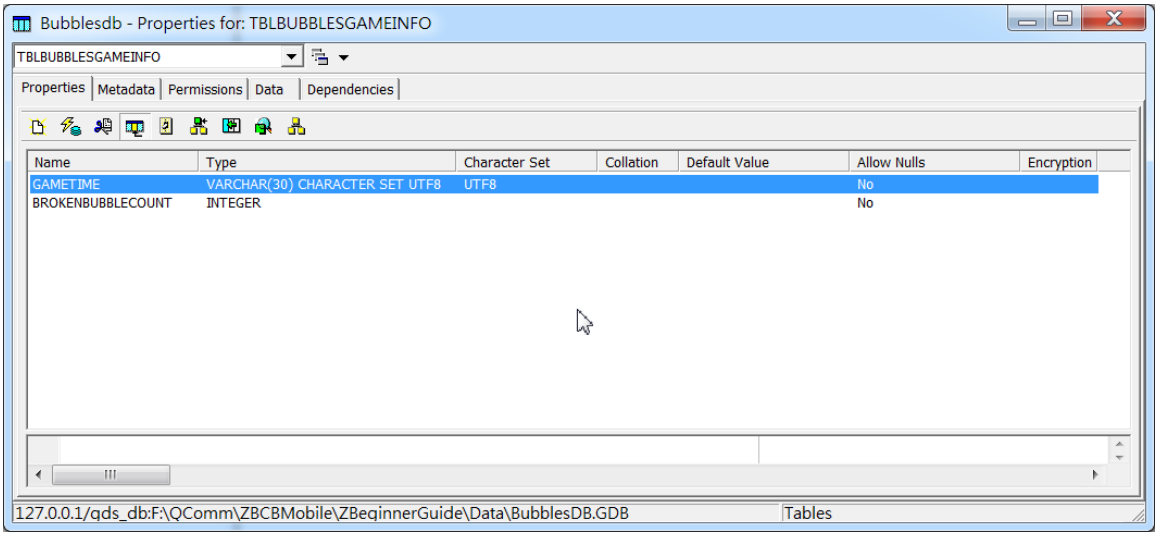
8-6 把游戏信息储存到数据库吧

C++Builder 中包含了新的信息库存取组件框架 FireDAC，由于 FireDAC 比 dbExpress 更适合于使用来开发移动 App，因此笔者强烈建议读者使用 FireDAC，本小节也将使用 FireDAC 来做为开发说明使用的数据库存取技术，同时本小节使用的数据库是包含在中的 InterBase ToGo。

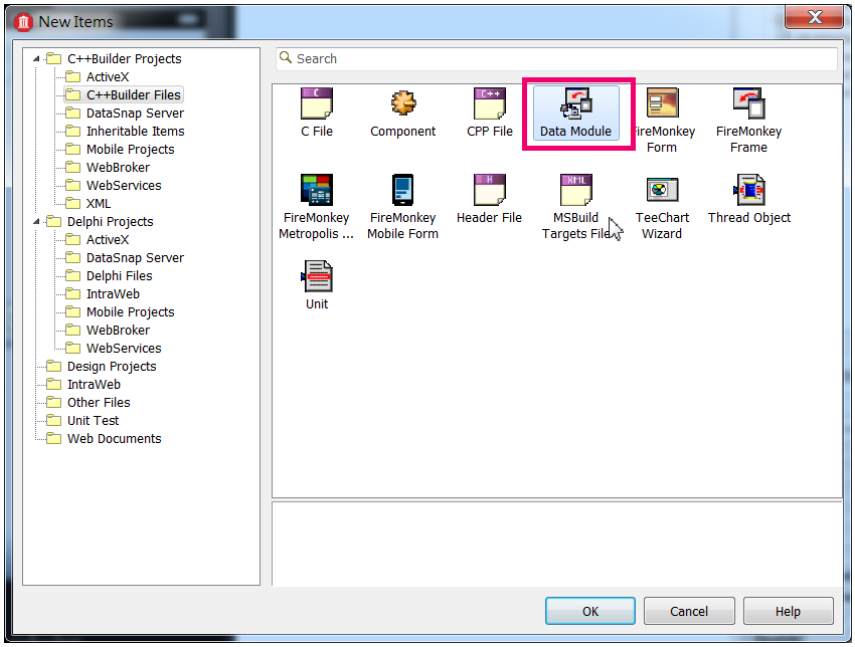
要使用 FireDAC 存取数据库非常的简单，基本上只需要使用 4 个 FireDAC 组件即可，下面的表格说明了本小节使用的 FireDAC 组件：

组件	名称	说明
	TFDConnection	使用来链接数据库的组件
	TFDQuery	使用来执行 SQL 命令的组件
	TFDPhysIBDriverLink	链接到 Interbase 的驱动程序组件
	TFDGUIxWaitCursor	控制 UI 等候光标的组件

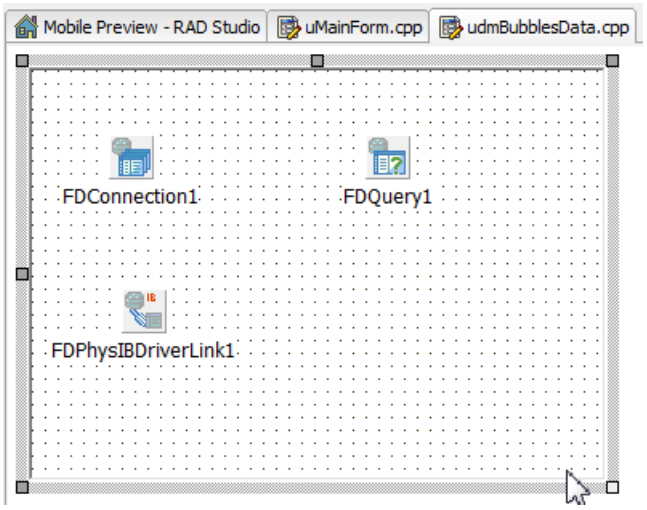
在开始之前我们需要建立一个范例 InterBase 数据库读者可以使用 IDE 中的 Explorer 或是 IBConsole 建立 BUBBLESDB.GDB 范例数据库，在其中建立 TBLBUBBLESGAMEINFO 数据表并且在其中建立下面的 2 个字段对象(读者也可以在本书的范例中找到这个数据库)：



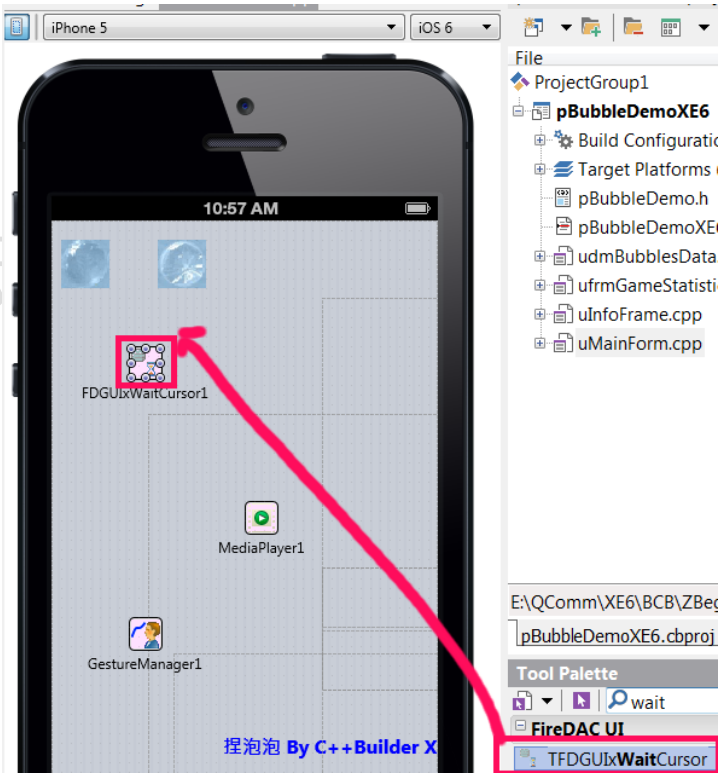
接着在 IDE 中为范例项目加入一个数据模块以便在其中使用 FireDAC 组件存取数据库，您可以在 IDE 的 New Items 对话盒中选择建立数据模块，如下所示：



点选了上面的数据模块后 IDE 便会建立一个空白的数据模块，请在其中放入 TFDConnection，TFDQuery 和 TFDPhysIBDriverLink 组件，如下所示：

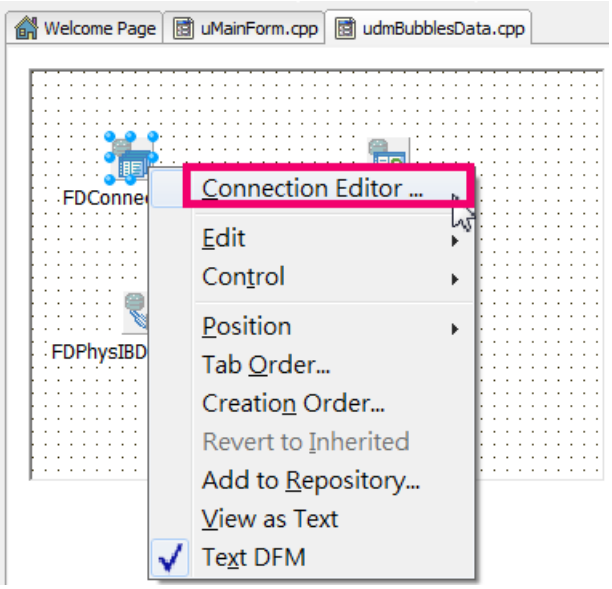


再把 TFDPhysIBDriverLink 元年放入到主窗体之中：

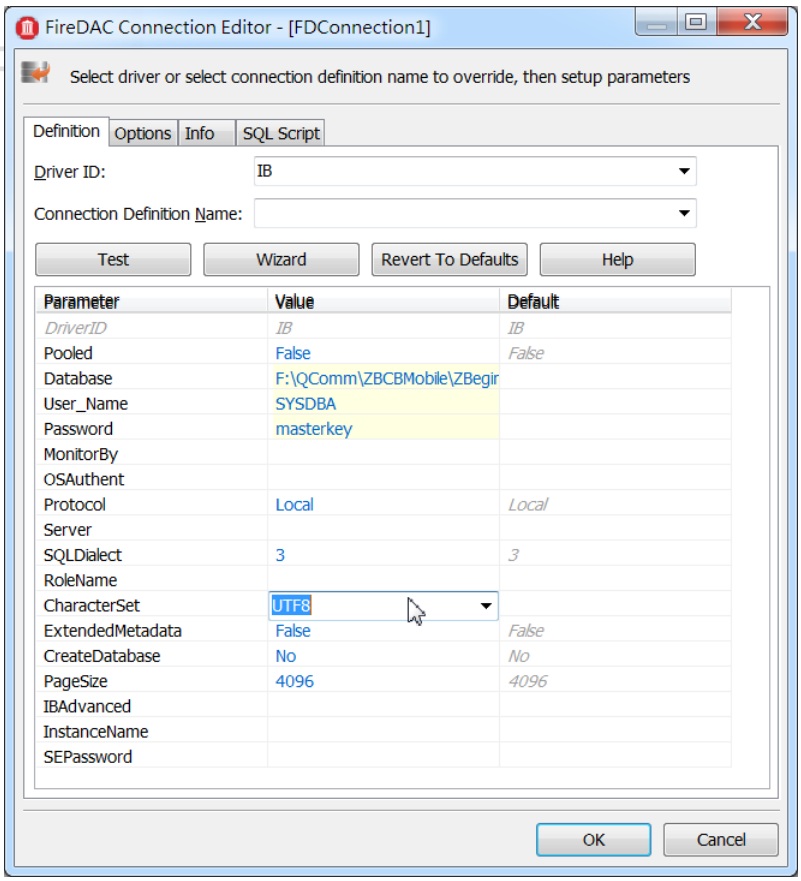


回到数据模块，现在我们要用 TFDConnection 来链接 InterBase ToGo 数据库，由于现在我们是在设计时期因此可以先直接使用 TFDConnection 连结 InterBase ToGo，等设计好基本的工作之后再于执行时期动态链接真正部署到手机中的 BUBBLESDB.GDB 范例数据库。

请在数据模块中点选 TFDConnection 再右击鼠标，此时便会出现 TFDConnection 组件的快显选单，请选择其中的 Connection Editor...选项：



TFDConnection 组件便会显示下面的组件编辑器，请在下面的组件编辑器 Database 字段加载 BUBBLESDB.GDB 范例数据库，并且在 User_Name，Password，CharacterSet 字段设定如下的数值：



在对象查看器中设定 TFDConnection 组件的 LoginPrompt 为 false。接着点选数据模块中的 TFDQuery 组件在对象查看器中设定它的 SQL 特性值为

```
select * from TBLBUBBLESGAMEINFO
```

以便从 TBLBUBBLESGAMEINFO 数据表中存取数据。

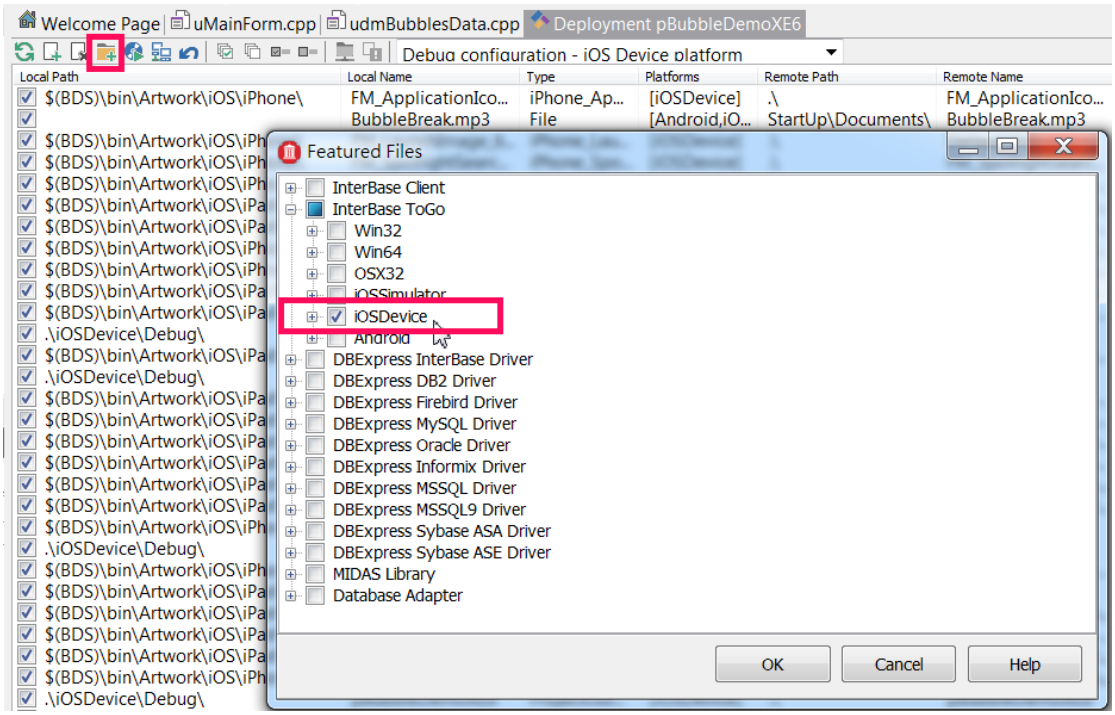
接着在数据模块的 OnCreate 事件处理函式和 OnDestry 事件处理函式中撰写如下的程序代码开启和 InterBase ToGo 的连结并且取得 TBLBUBBLESGAMEINFO 中的数据：

```
void __fastcall TDataModule1::DataModuleCreate(TObject *Sender)
{
  FDConnection1->Connected = true;
  FDQuery1->Active = true;
}

void __fastcall TDataModule1::DataModuleDestroy(TObject *Sender)
{
  FDConnection1->Connected = false;
}
```

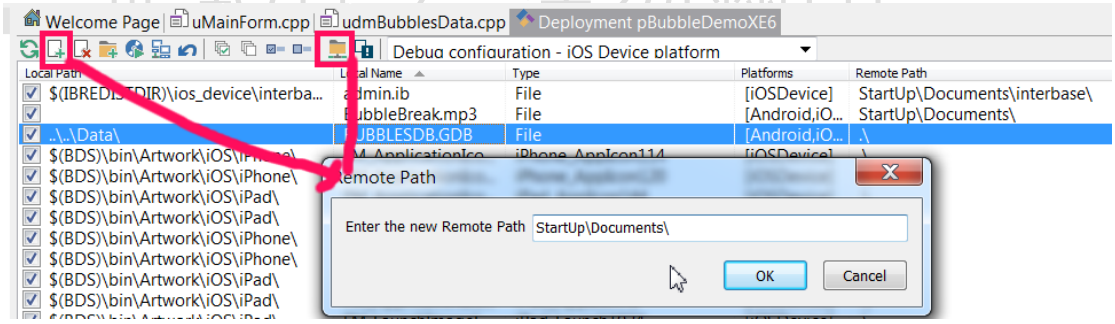
再为 TFDConnection 组件的 OnBeforeConnect 事件处理函式撰写如下的程序代码，在 OnBeforeConnect 我们需要从 App 的沙盒 Documents 目录中加载 BUBBLESDB.GDB 范例数据库：

```
void __fastcall TDataModule1::FDConnection1BeforeConnect(TObject
*Sender)
{
  FDConnection1->Params->Values["Database"] =
System::Ioutils::TPath::GetDocumentsPath() + PathDelim
+"BUBBLESDB.GDB";
}
```

因此我们也需要使用 IDE 的部署精灵把 BUBBLESDB.GDB 范例数据库部署到 App 沙盒的 Documents 目录中。请在 IDE 中先启动部署精灵，先点选上方的 Add Featured Files 快捷键加入要在此 iOS App 中使用 InterBase ToGo 的功能，C++Builder 就会在部署 App 时也顺便部署 InterBase ToGo 的相关函式库：



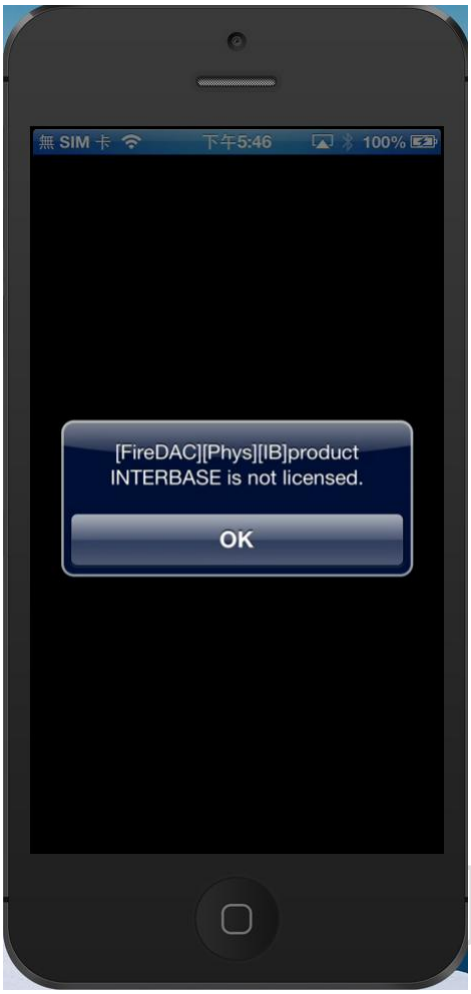
接着点选 Add Files 快捷键加入部署 BUBBLESDB.GDB 到 StartUp\Documents\目录中：



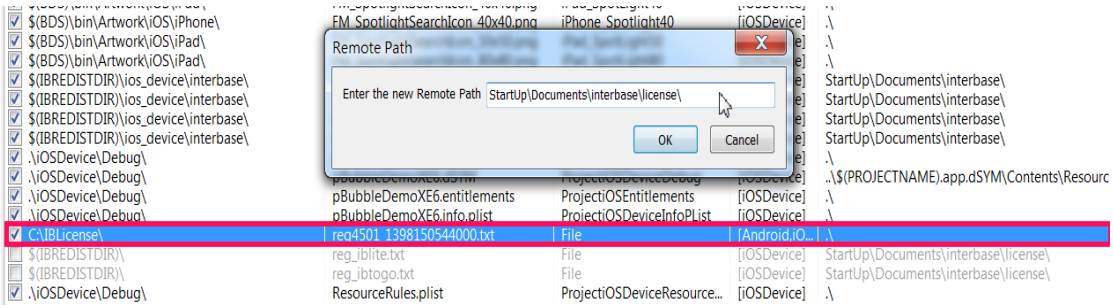
此外您需要到下面的ＵＲＬ取得 InterBase ToGo 的部署授权档一起部署：

```
http://docwiki.embarcadero.com/RADStudio//en/IBLite_and_IBToGo_Test_Deployment_Licensing
```

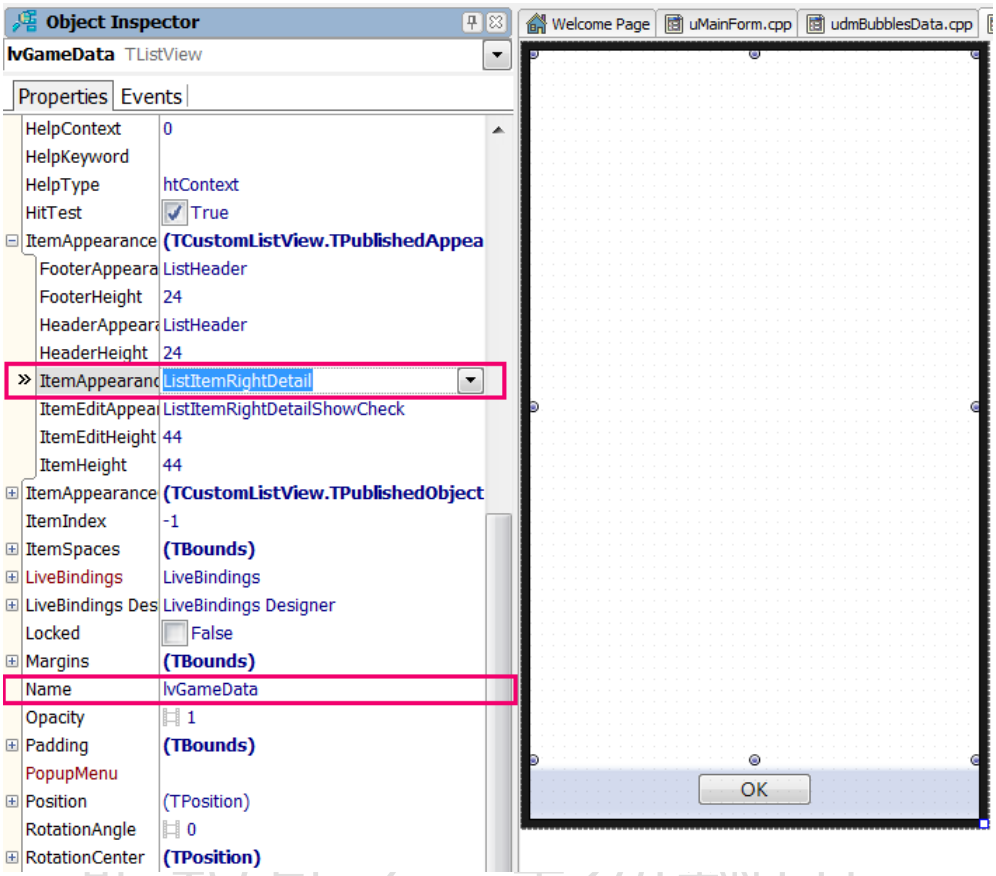
否则在此范例 App 执行时您便会看到下面的错误讯息：



最得了 InterBase ToGo 的部署授权档之后请再使用 Add Files 快捷键加入部署 InterBase ToGo 的部署授权文件到 StartUp\Documents\interbase\license\目录中，如下所示：

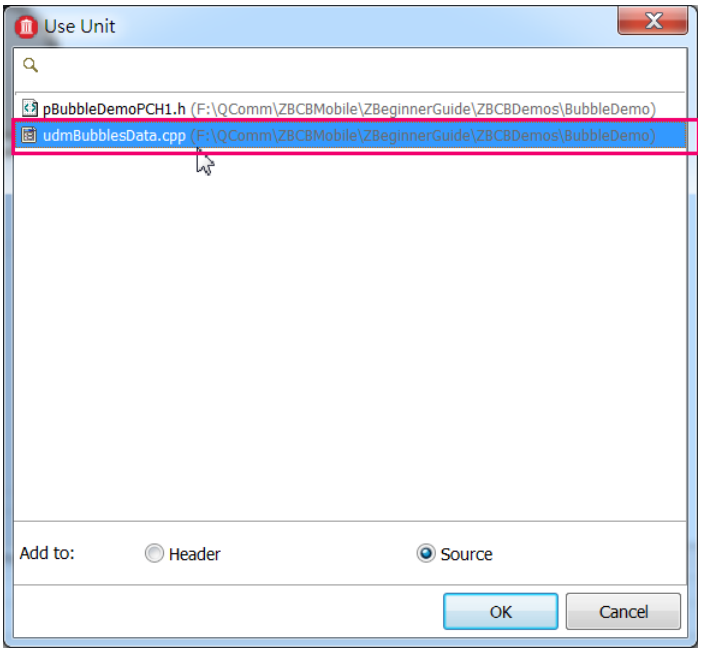


好了， 现在我们还需要一个ＵＩ来显示数据库中的数据，请于项目中再建立一个新的 Frame 对象设定它的 Name 特性值为 frmGameData，在其中放入 TListView，ToolBar 和 TButton 组件，设定 TListView 的 Align 特性值为 alClient，再设定它的 ItemAppearance 特性值为 ListItemRightDetail，设定它的 Name 特性值为 lvGameData，如下所示：



同时在主窗体中为这个 Frame 对象的 Y 轴加入一个 TFloatAnimation 对象，稍后当玩家使用由上往下的手势时就动态显示此 Frame 对象。

当然我们也需要在主窗体中使用 Frame 组件把这个 Frame 对象加入到主窗体，接着在主窗体中点选 File|Use Unit...把数据模块加入到主窗体中：



最后的工作就是撰写实作程序代码了，首先在 ResetGame 方法中加入呼叫 WriteGameData()方法把游戏数据写入数据库中：

```
void TfmMainForm::ResetGame()
{
  WriteGameInfo();
  WriteGameData(
  SetupGameInfoFrame();
  for (int iRow = 0; iRow < IROWS; iRow++)
  {
     for (int iCol = 0; iCol < ICOLS; iCol++)
     {
       if (BubblePoppedStatus[iRow][iCol])
       {

     pBubbles[iRow][iCol]->MultiResBitmap->Assign(imgOriNormalBubble->
MultiResBitmap);
         BubblePoppedStatus[iRow][iCol] = false;
       }
     }
  }
}
```

WriteGameData()方法是呼叫数据模块的 WriteGameData()方法同时传入目前游戏时间和掐破的泡泡数传给它：

```
void TMainForm::WriteGameData()
{
  dmGameData->WriteGameData(Now(), GetBrokenBubles());
}
```

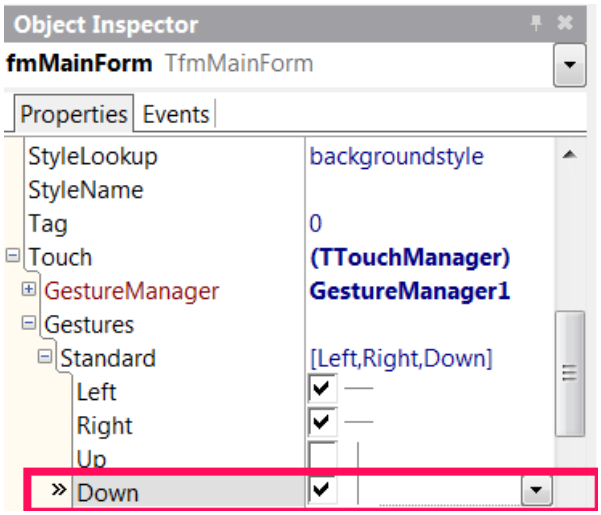
数据模块的 WriteGameData()方法非常简单，它使用 TFDQuery 组件把数据新增到 TBLBUBBLESGAMEINFO 数据表中：

```
void TdmGameData::WriteGameData(const TDateTime dt, const int iBubbles)
{
    FDQuery1->Insert();
    FDQuery1->Fields->Fields[0]->Value = dt;
    FDQuery1->Fields->Fields[1]->Value = iBubbles;
    FDQuery1->Post();
    FDQuery1->Refresh();
}
```

再回到主窗体的 OnGesture 事件处理函式中加入呼叫 FillGameData()方法，FillGameData()方法的目的是把 TBLBUBBLESGAMEINFO 数据表中的数据显示在由上往下出现的 frmGameData 对象中。

```
            if (EventInfo.GestureID == sgiDown)
            {
              FillGameData();
              faniGameDataY->Enabled = false;
              faniGameDataY->StartValue = -iScreenHeight;
              faniGameDataY->StopValue = 0;
              faniGameDataY->Enabled = true;
              Handled = true;
            }
```

记得要也要让 MainForm 支援向下的手势：



当然我们也需要在 SetupGameInfoFrame()方法中设定 frmGameData 的起始位置，我们把它设定在主窗体的上方位置：

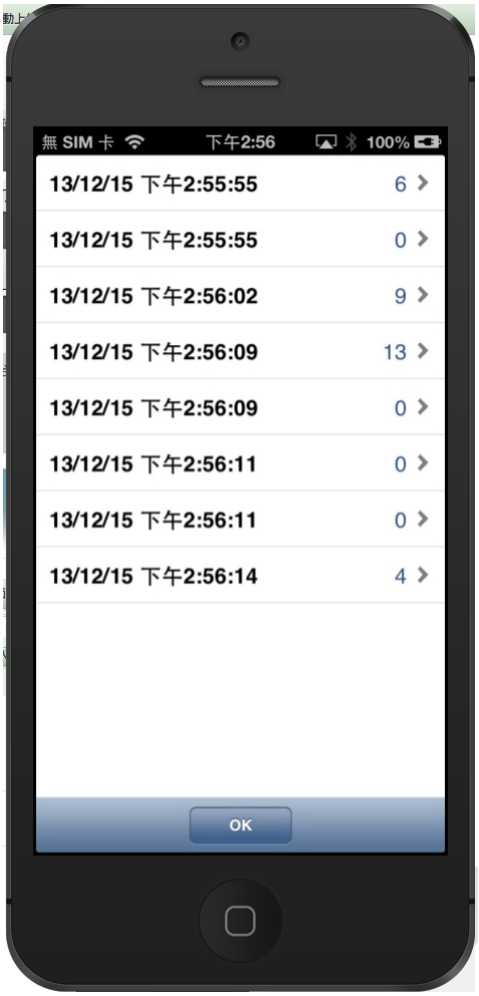
```
void TMainForm::SetupGameInfoFrame()
{
  frmGameInfo1->Position->X = Layout2->Width;
  frmGameInfo1->Position->Y = 0;
  frmGameInfo1->Parent = this;

  frmGameStatistics1->Position->X = -Layout2->Width;
  frmGameStatistics1->Position->Y = 0;
  frmGameStatistics1->Parent = this;

  frmGameData1->Position->X = 0;
  frmGameData1->Position->Y = -iScreenHeight;
  frmGameData1->Parent = this;
}
```

FillGameData() 则使用 TFDQuery 组件一一的从 TBLBUBBLESGAMEINFO 数据表中取出数据并且显示在 frmGameData 的 lvGameData 之中：

```
void TMainForm::FillGameData()
{
  frmGameData1->lvGameData->Items->Clear();
  frmGameData1->lvGameData->Items->BeginUpdate();
  try
  {
```

```
    dmGameData->FDQuery1->First();
    while (!dmGameData->FDQuery1->Eof)
    {
      TListViewItem *plvi = frmGameData1->lvGameData->Items->Add();
      plvi->Text = dmGameData->FDQuery1->Fields->Fields[0]->Value;
      plvi->Detail = dmGameData->FDQuery1->Fields->Fields[1]->Value;
      dmGameData->FDQuery1->Next();
    }
  }
  __finally
  {
      frmGameData1->lvGameData->Items->EndUpdate();
  }
}
```

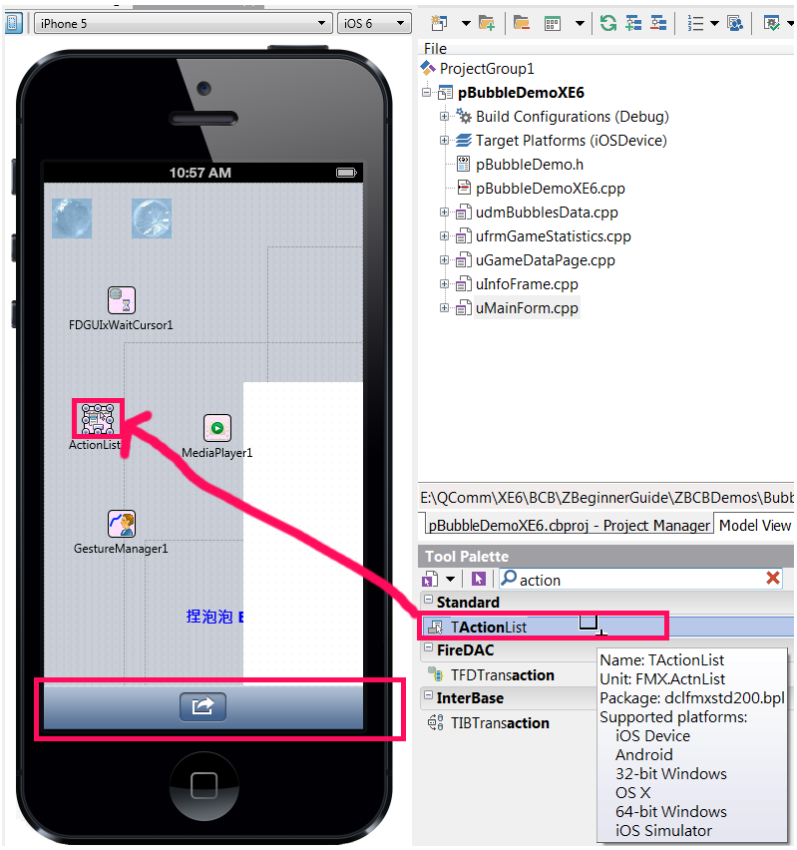
现在请编译范例 App 点选泡泡再摇动手机储存并重新开始游戏,如此反复数次之后再使用手指从手机上方往下方滑动，就可以看到类似如下的画面，范例 App 的游戏信息果然成功的储存到 InterBase ToGo 数据库并且能够显示出来了：



8-7 分享游戏的乐趣吧

在结束本小节之前，让我们再加入一个分享的功能，让我们能够把玩这个游戏的乐趣分享给您的好友吧，这只需要用 C++Builder 中的 ShareSheet 功能就可以轻易的完成，而 ShareSheet 功能则内含在 TActionList 组件。

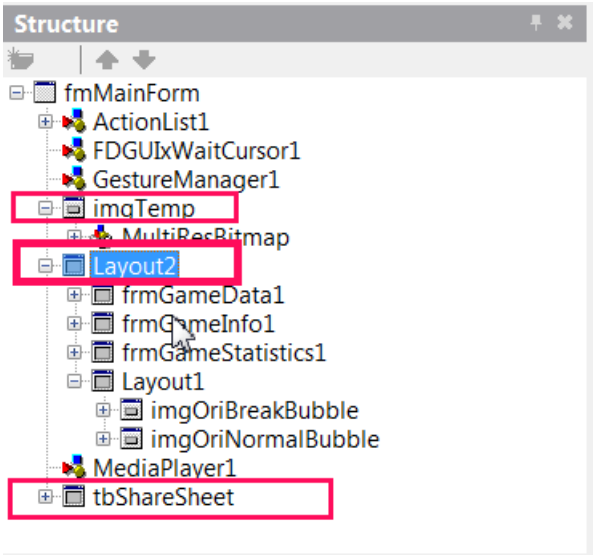
首先在主窗体中加入一个 ToolBar 组件设定它的 Align 特性值为 alBottom，再于其中放入一个 TButton 组件设定它的 StyleLookup 特性值为 actiontoolbuttonbordered 以及 Name 特性值为 btnShareGameInfo，再放入一个 TActionList 组件，此时主窗体如下所示：



接着继续在主窗体中加入一个 TLayout 组件，一个 TImage 组件设定它的特性值如下：

特性	特性值
Name	imgTemp
Align	Client
Visible	False

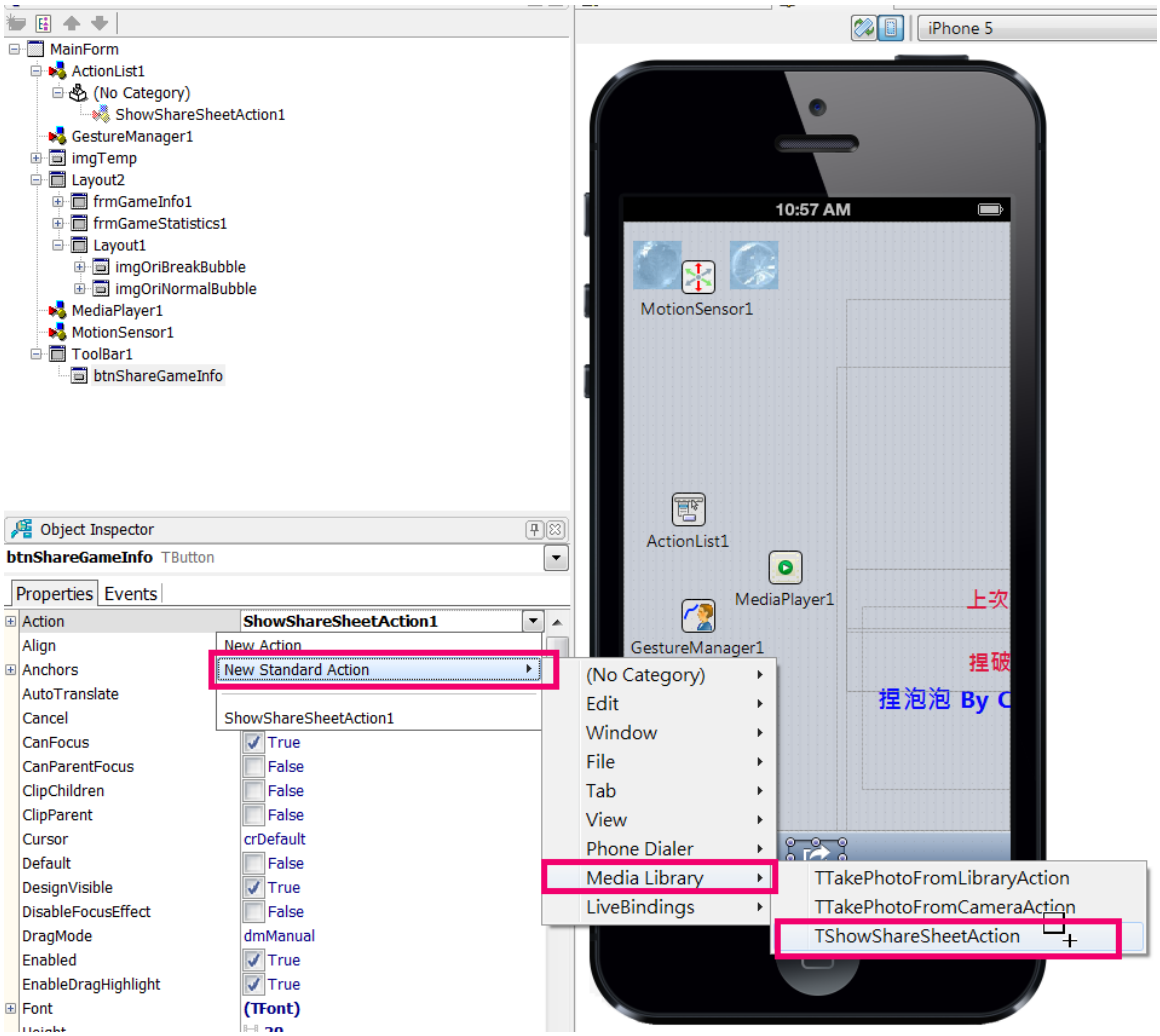
最后再于 IDE 左上方的架构窗口中移动前面 3 个 Frame 对象以及 Layout1 对象于新的 Layout2 对象之中，此时架构窗口会显示主窗体中所有组件的关系如下所示：



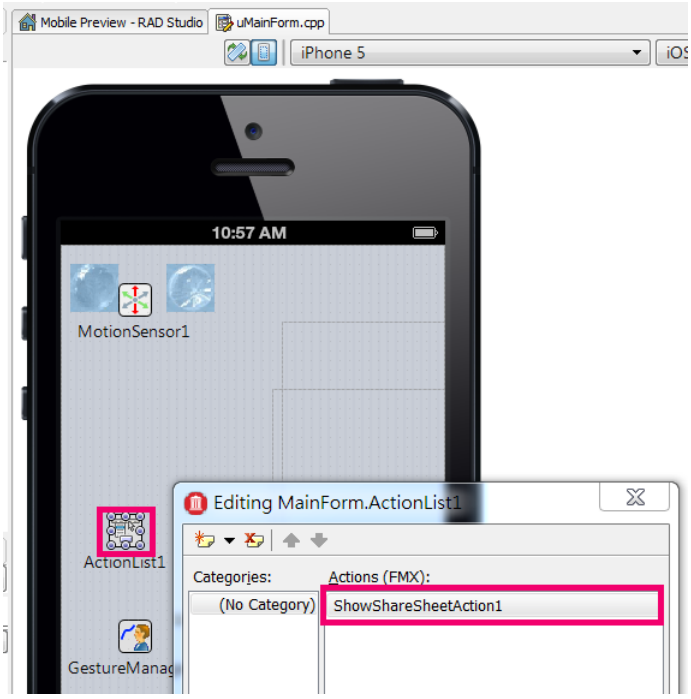
把前面 3 个 Frame 对象以及 Layout1 对象移动到新的 Layout2 对象之中是因为稍后我们要把手机游戏画面截取下来并且以图形的方式分享出去，您很快会看到如何做到。

接着点选 ToolBar 中的 btnShareGameInfo 组件，我们希望玩家稍后点选这个 TButton 组件之后就可以把游戏的信息分享出去，这很容易就能做到。

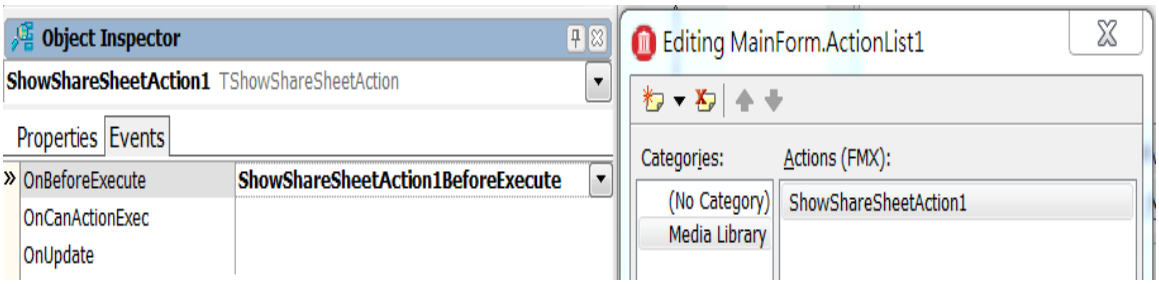
点选了 btnShareGameInfo 之后在对象查看器中于它的 Action 特性中点选 New Standard Action | Media Library | TShowShareSheetAction 选项让 btnShareGameInfo 的被点选时就触发 TShowShareSheetAction 的动作，也就是分享的动作，如下所示：



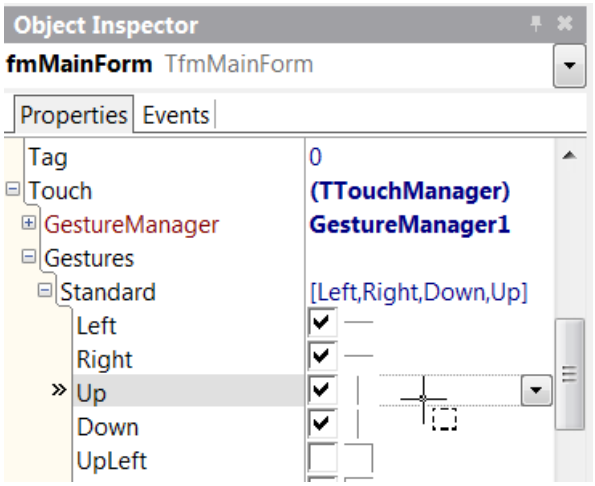
再双击主窗体中的 TActionList 组件，此时在其中就会自动产生一个 TShowShareSheetAction 对象，如下所示。



请再点选 TShowShareSheetAction 对象，在对象查看器的 Events 页次中为它建立 OnBeforeExecute 事件处理函式，如下所示。OnBeforeExecute 事件处理函式会在 TShowShareSheetAction 对象分享行动之前执行，因此我们就可以在这个事件处理函式中把游戏信息准备好并且放入 TShowShareSheetAction 对象中再让 TShowShareSheetAction 对象分享出去。



最后回到主窗体，在它支持的手势种类中再加入支持由下往上的手势动作，如下所示：



好了，现在就可以开始撰写实作程序代码了，首先到主窗体的 OnGesture 事件处理函式中加入处理由下往上的手势动作。在下面的实作程序代码中当范例 App 查觉玩家做出了由下往上的手势动作之后就呼叫 GetScreenImage()和 SetupToolBarPosition(true)方法。

```
        if (EventInfo.GestureID == sgiUp)
        {
          GetScreenImage();
          SetupToolBarPosition(true);
        }
```

GetScreenImage()方法的工作就是把目前手机的游戏画面截取出来并且指定给前面加入的 imgTemp 对象。要截取手机画面很简单，因为现在显示泡泡的图形以及 2 个 Frame 对象的内容都是包含在 Layout2 对象中，因此只需要呼叫 Layout2 对象的 MakeScreenshot()方法就可以快照手机游戏画面，而且 MakeScreenshot()方法会回传代表手机游戏画面的 TBitmap 对象，因此我们只需要再直接把它指定给 imgTemp 对象的 Bitmap 特性即可：

```
void TMainForm::GetScreenImage()
{
  imgTemp->Bitmap->Assign(Layout2->MakeScreenshot());
}
```

SetupToolBarPosition()方法控制前面加入的 ToolBar 是否要出现在手机画面之中，如果玩家做出了由下往上的手势动作就代表玩家要分享游戏信息，就显示出 ToolBar 好让玩家可以点选其中的分享按钮组件把游戏信息分享出去：

```
001    void TMainForm::SetupToolBarPosition(const bool bVisible)
002    {
003      if (bVisible)
```

```
004     {
005     ToolBar1->Align = TAlignLayout::alBottom;
006     ToolBar1->BringToFront();
007     }
008     else
009     {
010     ToolBar1->Align = TAlignLayout::alNone;
011     ToolBar1->Position->Y = Layout1->Height;
012     }
013     ToolBar1->Visible = bVisible;
014   }
```

在上面的程序代码中如果参数 bVisible 是 true 的话 005 行就设定 Toolbar 的 Align 特性值为 alBottom 让它出现在手机画面的下方，006 行把 Toolbar 提升到手机画面的最上方让它不会被其他的对象遮挡。如果参数 bVisible 是 false 的话就代表分享动作已完成或是取消了，010 行就设定 Toolbar 的 Align 特性值为 alNone，再把 Toolbar 的位置移出手机画面之外让玩家看不到它。

接受我们需要修改 SetupGameInfoFrame()方法把 3 个 Frame 对象的父代设定为 Layout2。


```
void TfmMainForm::SetupGameInfoFrame()
{
  frmGameInfo1->Position->X = Layout2->Width;
  frmGameInfo1->Position->Y = 0;
  frmGameInfo1->Parent = Layout2;

  frmGameStatistics1->Position->X = -Layout2->Width;
  frmGameStatistics1->Position->Y = 0;
  frmGameStatistics1->Parent = Layout2;

  frmGameData1->Position->X = 0;
  frmGameData1->Position->Y = -Layout2->Height;
  frmGameData1->Parent = Layout2;
}
```

当玩家点选分享按钮之后就会触发 TShowShareSheetAction 对象的 OnBeforeExecute 事件处理函式，在其中我们先把目前玩家捏破的泡泡数指定给 TShowShareSheetAction 对象的 TextMessage 特性，再把刚才截取的手机游戏画面指定给 TShowShareSheetAction 对象的 Bitmap 特性，这样 TShowShareSheetAction 对象就会把这些讯息分享出去了。

```
void __fastcall TMainForm::ShowShareSheetAction1BeforeExecute(TObject
*Sender)
{
    ShowShareSheetAction1->TextMessage = "这次捏破了" +
IntToStr(GetBrokenBubles()) + "泡泡";
    ShowShareSheetAction1->Bitmap->Assign(imgTemp->Bitmap);
    SetupToolBarPosition(false);
}
```

现在编译和执行此范例 App 并且随意点选泡泡，再摇动手机重新开始游戏并且储存游戏信息，使用手指向手机屏幕的左方滑动，再用手指向手机屏幕的右方滑动，就可以看到 2 个 Frame 对象从右向左以及从左向右慢慢的滑动出现，再使用手指向手机屏幕的上方滑动就可以看到 ToolBar 出现在手机画面的下方：

点选下方 Toolbar 中的分享按钮对象就可以看到手机自动出现可以分享信息的目的地，让我们选择分享到 Facebook，

之后就可以看到如下的画面，您可以看到在ShowShareSheetAction1BeforeExecute 事件处理函式中设定的文字信息和截取的手机游戏画面都正确的出现了：

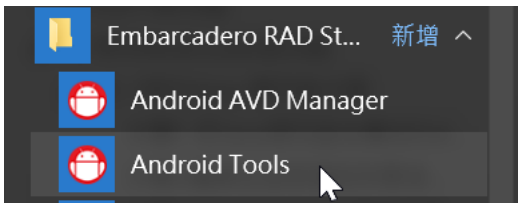
最近如果点选上面画面中的发布按钮之后就可以真的在 Facebook 的网页中看到如下的执行结果画面，我们成功的使用 C++Builder 的 ShareSheet 功能分享范例 App 的游戏信息了，Cool!

C++Builder for Android入门指引手册

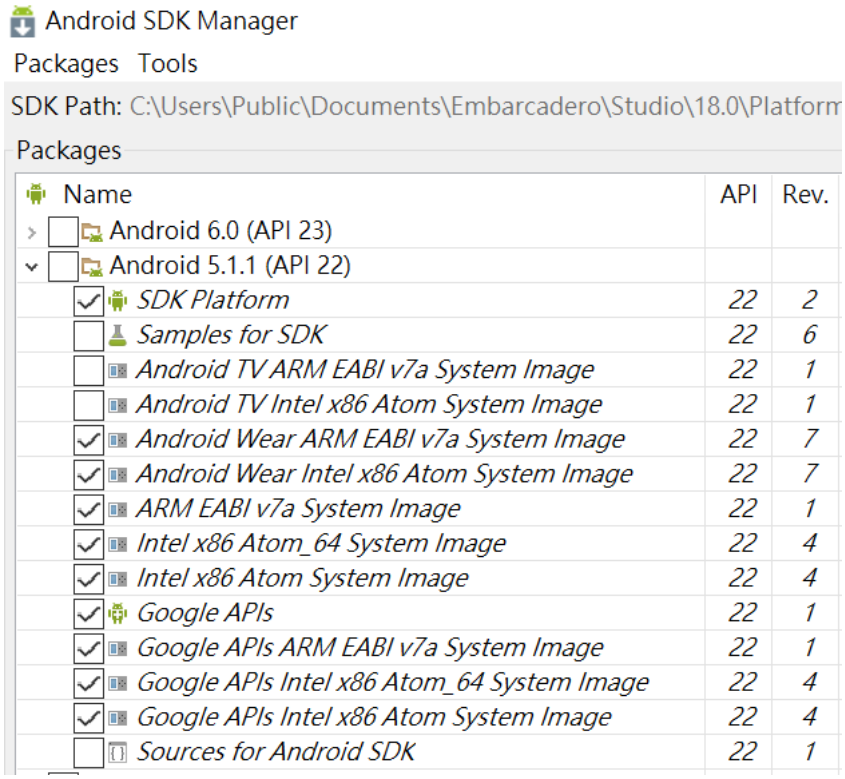
强调的功能就是一套源代码就可同时开发多个平台的应用程序，因此在前面已经说明了如何使用开发 iOS App，那么您现在就可以使用相同的技巧来开发 Android App。不过开发 Android 和开发 iOS 不同的地方是在开发 Android 之前您需要先安装好 Android SDK 和 Android NDK，并且在 C++Builder IDE 中设定好开发环境。

9.安装和设定 BCB for Android 开发环境

由于 Google 授权政策的改变因此安装时不再会同时安装 Android SDK 和 Android NDK，读者可在安装完后在它的程序群组中执行 Android Tools 程序安装 Android SDK 和 Android NDK:



例如笔者使用 Android Tools 程序安装了 Android 5.1.1:



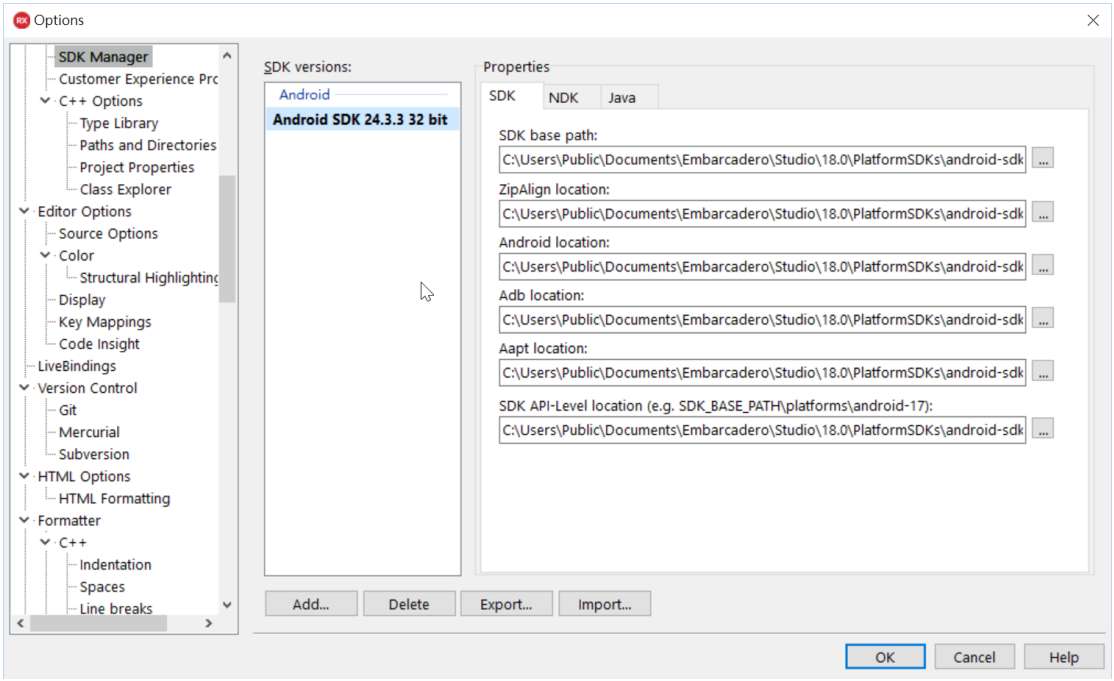
完成安装 Android SDK 和 Android NDK 后需要 C++Builder IDE 中设定 Android SDK 和 Android NDK 的安装路径，以便让 IDE 能够找到相关的档案和工具。例如 Android Tools 把 Android SDK 安装在：

```
c:\Users\Public\Documents\Embarcadero\Studio\18.0\PlatformSDKs\android-sdk-windows\
```

Android NDK 安装在：

```
c:\Users\Public\Documents\Embarcadero\Studio\18.0\PlatformSDKs\android-ndk-r9c\
```

接着请回到的 IDE 中，点选 Tools|Options...菜单，并且点选 Options 对话盒中的 SDK Manager 选项，如下所示：

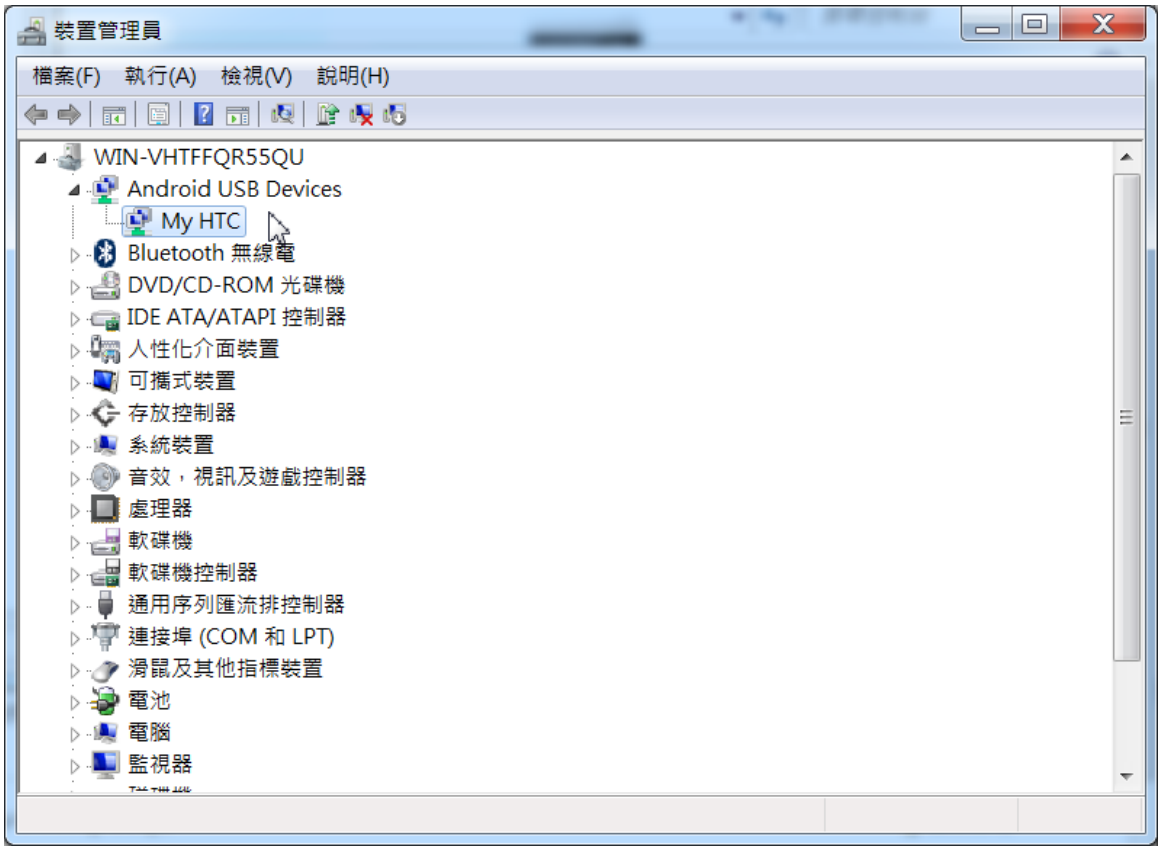


我们只需要把上 SDK 工具设定成 Android Tools 安装的路径即可。

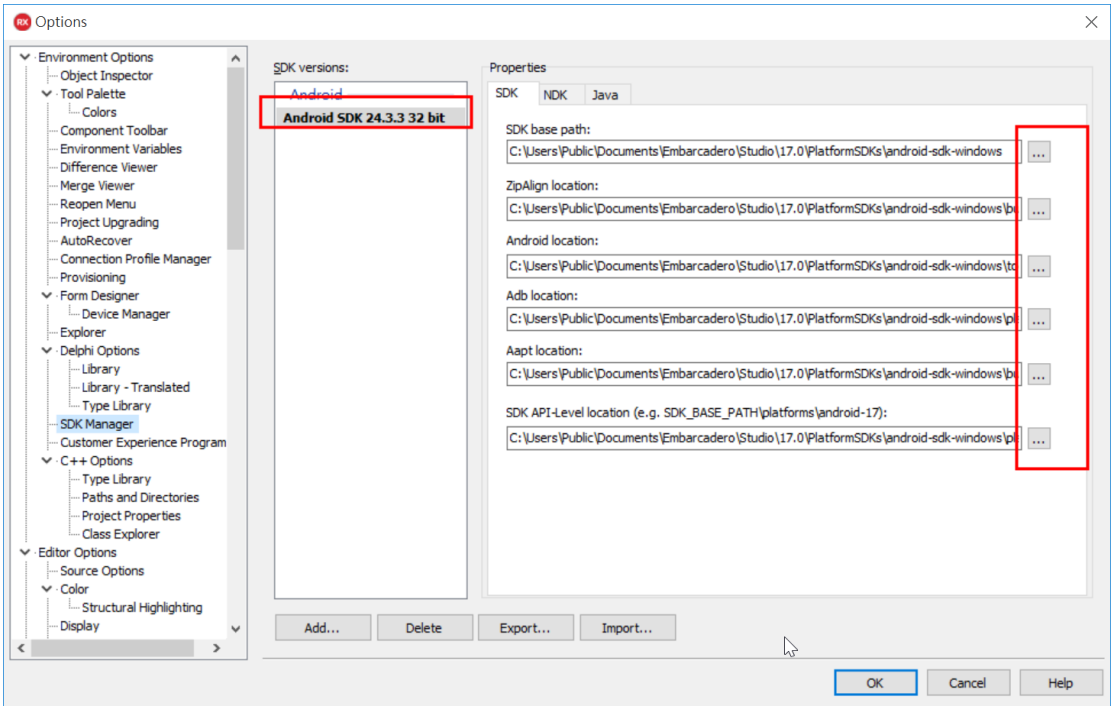
之后读者需要安装使用开发的 Android 手机的驱动程序，才能让 C++Builder IDE 部署和测试。读者需要到您的手机厂商网站下载，例如笔者是使用 hTC Incredible S，因此笔者到 hTC 下列的 URL 下载驱动程序：

http://www.htc.com/tw/software/htc-sync-manager/

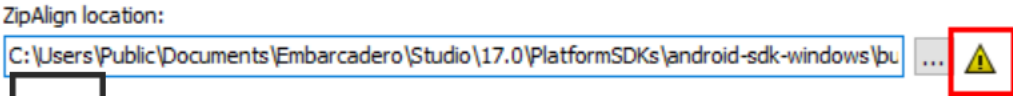
一旦安装了 hTC 的 Sync Manager 之后笔者在操作系统的设备管理器中就可以看到 HTC 手机：



在 XE7 之后到的版本中，当您安装完 C++Builder 之后 IDE 会在您启动 Android 开发时自动安装 Android SDK，并且会正确替您设定好 SDK Manager 中的设定，只要您开启 Options 对话盒并在 SDK Manager 选项中看到右方没有显示这个⚠符号即代表一切安装和设定是正常的，那您就可以开始 Android 的开发工作：



而如果您在任一选项的右方看到⚠符号，那代表此设定不正确，通常的错误原因是因为在左方的目录中找不到需要的 exe 檔。



例如上方图形有⚠符号就代表在左方的

```
C:\Users\Public\Documents\Embarcadero\Studio\18.0\PlatformSDKs\android-sdk-windows\build-tools\23.0.2\zipalign.exe
```

目录中找不到 ZipAlign.exe 这个档案，您只需要在左方指到包含 ZipAlign.exe 档案的目录即可修正此错误。

接下来我们就可以开始开发 Android App 了。

10.开发 C++Builder for Android App

从之后 C++Builder 正式支持 Android 的开发，而且 iOS 的程序代码可以再使用于 Android 平台，例如在前面章节示范的 iOS 范例 App 您可以试着把它们移植到 Android 平台中执行。

在本小节中我们将示范如何使用 C++Builder 开发 Android 的 App，由于前面的章节已经说明了基本的概念和技巧，因此在本小节中让我们试着来开发一些比较有趣的 Android App，我们将使用台北市政府提供的公共信息做为范例。

台北市政府在下面的 URL 提供了许多的公共信息让市民能够查询使用：

```
http://data.taipei.gov.tw/opendata
```

在其中有许多的公共信息已经是封装成 JSON 的形式，例如在下面的 URL 中台北市政府提供了台北市旅馆数据库的信息让市民或是旅游人士能够查询台北市旅馆的相关信息：

```
http://data.taipei.gov.tw/opendata/apply/NewDataContent?oid=4418C600-D7F5-46D6-BB1F-1DA29DAB91E9
```

现在让我们使用 C++Builder 来开发一个台北市旅馆查询 App，在这个范例 Android App 中将提供如下的功能：

- 查询台北市旅馆
- 使用部份关键词搜寻旅馆

这个范例将触及许多移动平台开发的技术，在下面的章节中我们将试着一一的完成上面的功能。

10-1 开发查询台北市旅馆 App

在下面的 URL 中：

```
http://data.taipei.gov.tw/opendata/apply/NewDataContent?oid=4418C600-D7F5-46D6-BB1F-1DA29DAB91E9
```

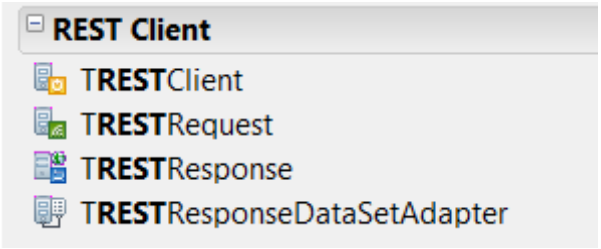
提供的旅馆信息是使用 JSON 封装的数据，下面是取得结果的部份资料：

```
[{"rownumber":"33","ref_wp":"6","cat1":"住宿","cat2":"一般旅馆
","serial_no":"B0225
","memo_tel":"0227735177","memo_fax":"0227727569","memo_cost":"1
280 以上","memo_time":"","stitle":"友统旅馆
","xbody":"","avbegin":"2008-10-20","avend":"2010-03-17","idpt":
"台北旅游网","xurl":"http://yotong.ffh.com.tw/","address":"台北市大
安区忠孝东路四段 197 号 13 楼
","xpostdate":"2010-03-17","langinfo":"10","poi":"Y","info":"","
```

```
longitude":"121.551654","latitude":"25.041705","file":"<file><im
g description=\"友统旅馆
1\">http://www.taipeitravel.net/d_upload_ttn/frontsite/tw/hotel/
B0225/B0225_1.jpg</img><img description=\"友统旅馆
2\">http://www.taipeitravel.net/d_upload_ttn/frontsite/tw/hotel/
B0225/B0225_2.jpg</img><img description=\"友统旅馆
3\">http://www.taipeitravel.net/d_upload_ttn/frontsite/tw/hotel/
B0225/B0225_3.jpg</img></file>"},{"rownumber":"364","ref_wp":"6"
,"cat1":"住宿","cat2":"一般旅馆","serial_no":"B0138
","memo_tel":"0225971281","memo_fax":"0225971288","memo_cost":"1
400 以上","memo_time":"","stitle":"庆天阁大饭店
","xbody":"","avbegin":"2008-10-20","avend":"2009-07-21"
…
```


从上面的结果中我们可以了解整个数据是以 JSON 数组对象封装的，而其中每一笔旅馆信息是使用一个 JSON 对象封装的。

了解了这个台北市政府公共服务的数据封装规则之后要解析其中的信息就非常的简单了，我们可以使用 Indy HTTP 组件取得这个数据再使用 Data.DBXJSON 中的 JSON 相关类别来解析其中的信息。但是在 C++Builder XE5 之后提供了新的 REST Client 组件组让我们可以更轻松的成为 REST 客户端以使从任何提供 REST 公共服务的来源取得需要的数据。

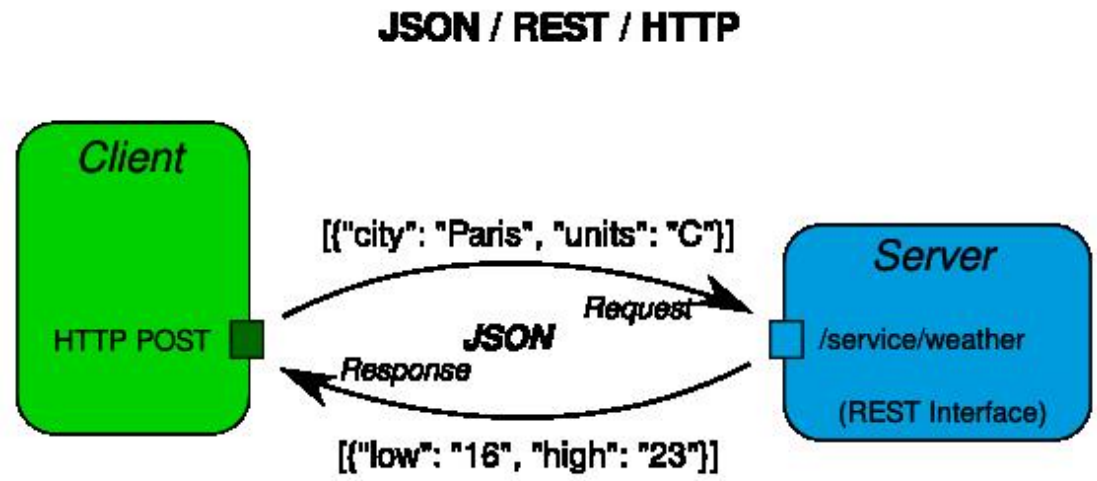


下面的表格说明了上面 4 个 REST Client 组件组的功能：

组件	说明
TRESTClient	指定提供 REST 服务的 REST 服务器
TRESTRequest	提出 REST 请求向 REST 服务器要求服务
TRESTResponse	REST 服务器执行结果回传到此组件
TRESTResponseDataSetAdapter	把 TRESTResponse 组件中的 JSON 结果转换成数据集(DataSet)的形式

我们可以使用面的图形来简单的说明如何使用上面的组件。在下图说明了一个 RESTful 架构的基本执行流程。左方的 REST 客户端藉由 HTTP/HTTPS 向 REST 服务器要求服务，这个要求的服务也是使用 JSON 封装的，在 REST 服务器接受到要求并且执行完毕之后就会把执行结果再封装成 JSON 的形式回传给 REST 客户端。

因此我们可以使用上表中的 TRESTClient 组件指定右方 REST 服务器的所在地，使用 TRESTRequest 提出要求服务，右方 REST 服务器会把执行结果回传到 TRESTResponse 组件中，最后我们可以使用 TRESTResponseDataSetAdapter 组件把 JSON 结果换成数据集再储存于 TClientDataSet 等组件中就可以存取其中的数据了。

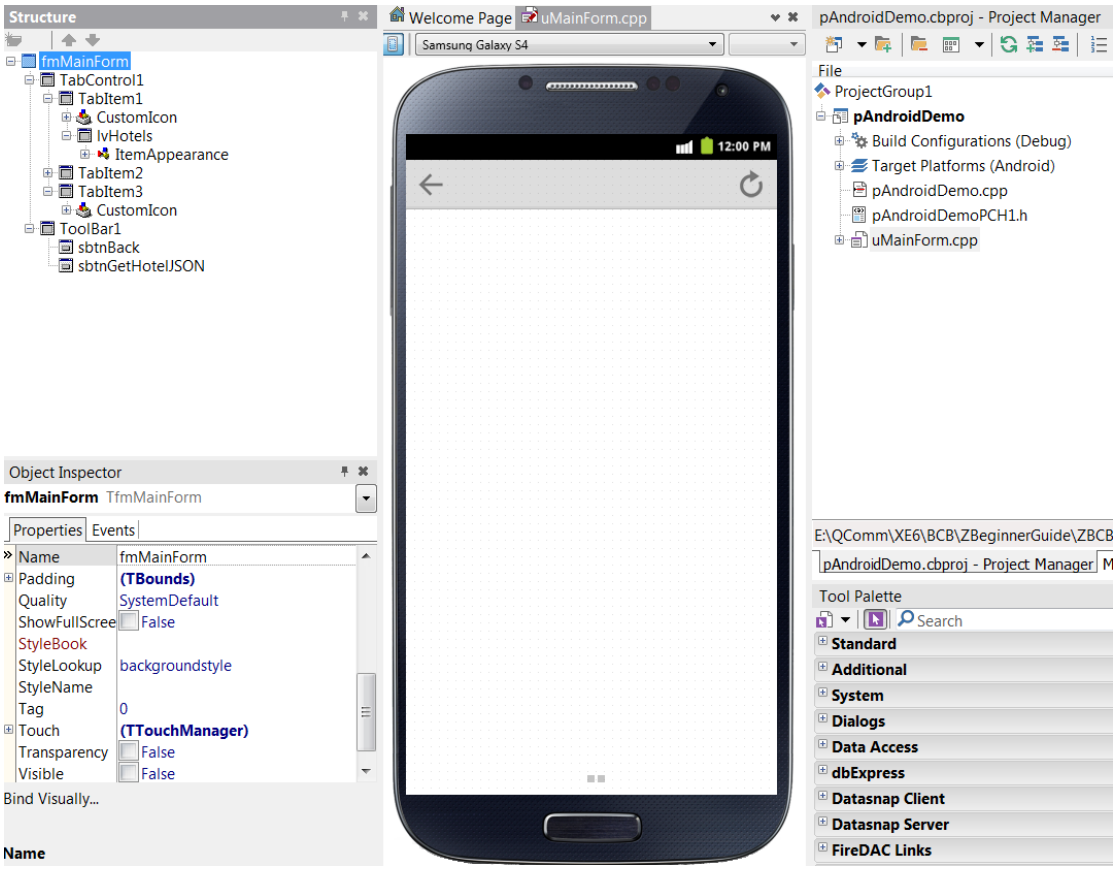


了解了上面的概念之后我们就可以轻易的使用 REST Client 中的组件完成查询台北市旅馆信息的工作：

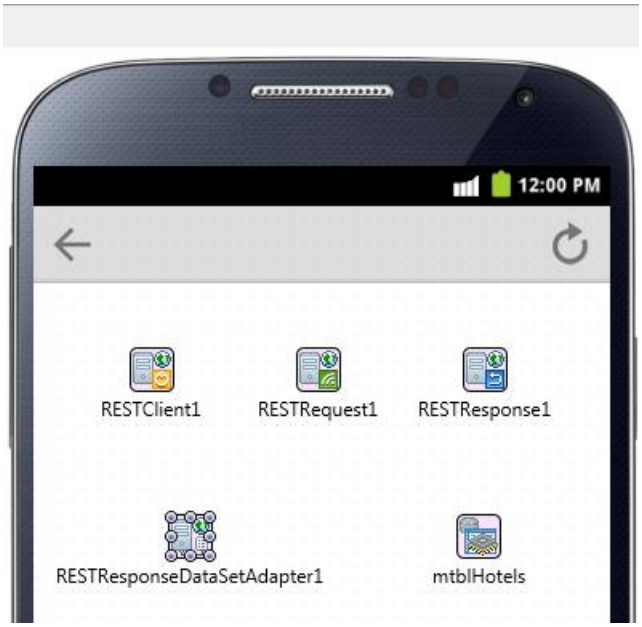
1. 使用 TRESTClient 组件设定指向 http://data.taipei.gov.tw/opendata/apply/NewDataContent?oid=4418C600-D7F5-46D6-BB1F-1DA29DAB91E9 取得服务
2. 使用 TRESTRequest 组件提出 REST 请求
3. REST 请求结果会自动回传到 TRESTResponse 组件
4. 再使用 TRESTResponseDataSetAdapter 组件把 TRESTResponse 组件中的 REST 执行结果换成数据集

5. 把 TRESTResponseDataSetAdapter 组件的结果储存在 TClientDataSet 或是 FireDAC 的 TFDMemTable 组件中

现在请使用 C++Builder 建立一个 FireMonkey Mobile 空白项目，在主窗体中放入一个 TabControl 并在其中建立个 TabItem，在第 1 个 TabItem 中放入名为 lvHotels 的 TListView 组件并且设定此 TListView 组件的 ItemAppreance|ItemAppeance 子特性值为 ListItemRightDetail。最后再放入 ToolBar 组件并且在其中放入 2 个 TSpeedButton，此时主窗体如下所示：



再于主窗体中放入前面介绍的 4 个 REST Client 中的组件以及一个 TFDMemTable 组件，如下所示：



然后设它如下的特性值：

TRESTClient 组件：

特性	特性值
BaseURL	http://data.taipei.gov.tw/opendata/apply/query/NDQxOEM2MDAtRDdGNS00NkQ2LUJCMUYtMURBMjlEQUI5MUU5?$format=json

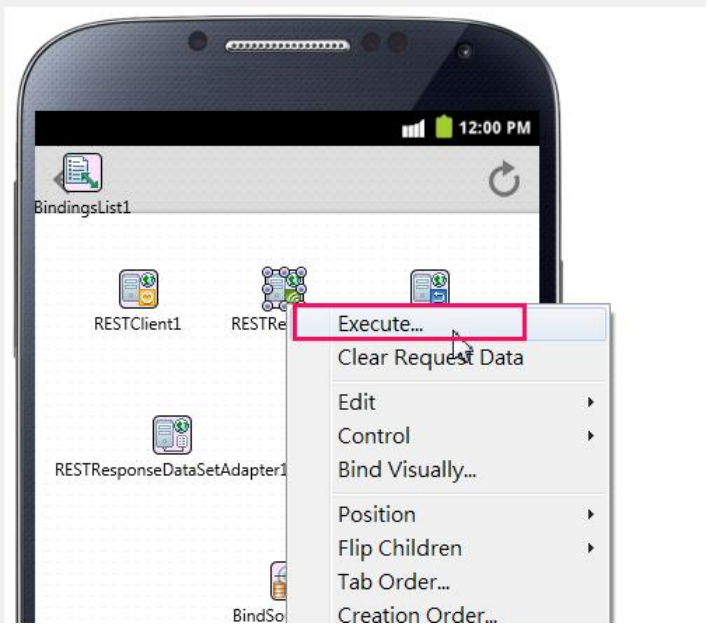
TFDMemTable 组件：

特性	特性值
Name	mtblHotels

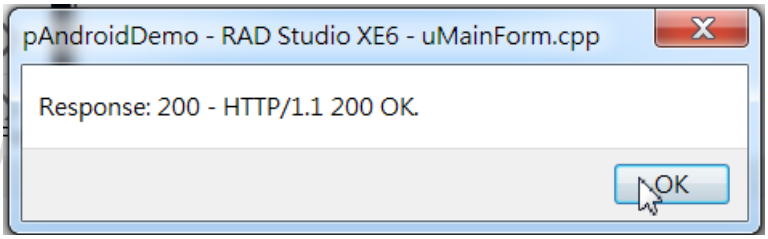
RESTResponseDataSetAdapter1 组件：

特性	特性值
ResponseJSON	RESTResponse1
DataSet	mtblHotels

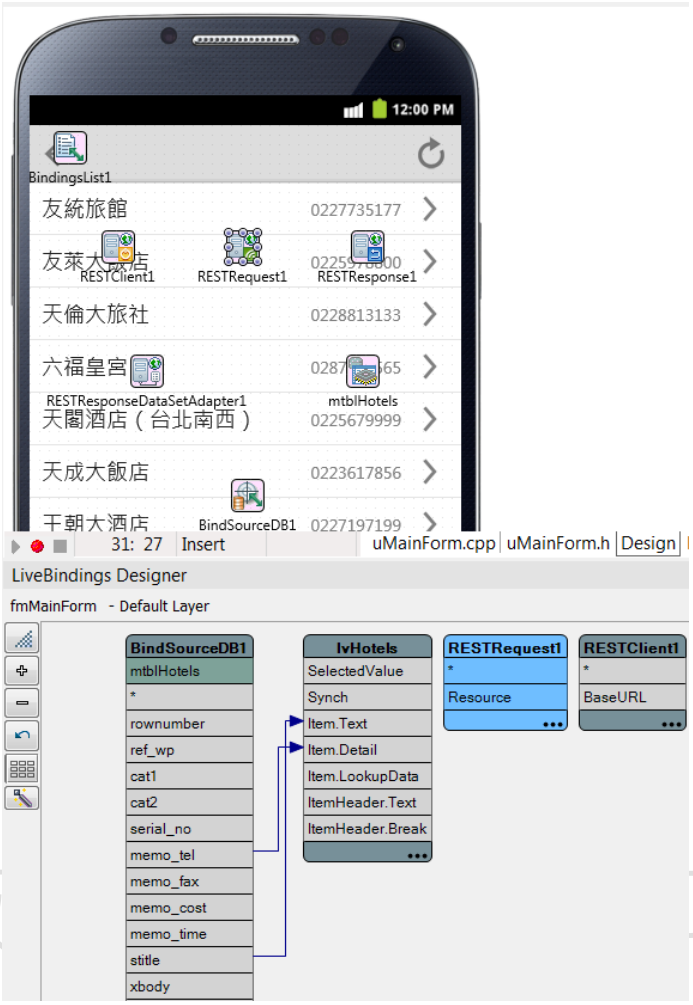
接着我们就可以开始设计台北市旅馆显示在 lvHotels 之中了，为了方便设计 UI，现在请点选主窗体中的 RESTReuqest1 组件，点选鼠标右键选择 Execute...选项(请确定您的网络链接在工作中)：



过了数秒之后您会看到IDE显示如下的讯息代表RESTReuqest1组件提出的REST请求已成功执行且结果已回传到RESTResponse1组件中：



接着设定RESTResponseDataSetAdapter1组件的Active特性值为true，开启LiveBinding设计家，把mtblHotels组件的stitle字段链接到lvHotels的Item.Text，再把memo_tel字段连结到lvHotels的Item.Detail，此时就可以在lvHotels组件中看到台北市旅馆的信息了：



完成初步的设计之后就可以开始撰写些简单的程序代码让这个范例 App 工作了。首先在主窗体的 OnActivate 事件处理函式中关闭 RESTResponseDataSetAdapter1：

```
void __fastcall TfmMainForm::FormActivate(TObject *Sender)
{
  RESTResponseDataSetAdapter1->Active = false;
}
```

接着在主窗体的中加上方的 TSpeedButton 的 OnClick 事件处理函式中呼叫 GetTaipeiHotelsJSON()方法提出 REST 请求：

```
void __fastcall TfmMainForm::sbtnGetHotelJSONClick(TObject *Sender)
{
  GetTaipeiHotelsJSON();
}
```

GetTaipeiHotelsJSON()方法先呼叫 RESTRequest1 的 Execute()方法正式提出 REST 请求，在执行完毕之后就开启 RESTResponseDataSetAdapter1 以显示最及时的台北市旅馆的信息：

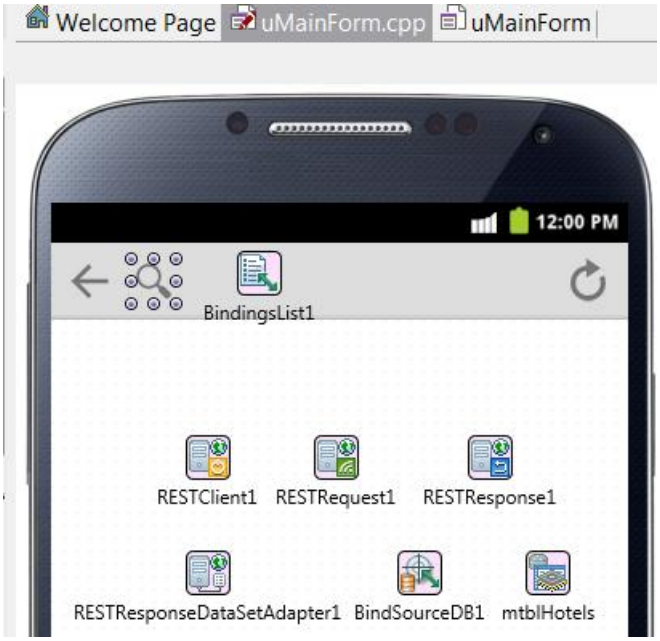
```
void TfmMainForm::GetTaipeiHotelsJSON()
{
  RESTRequest1->Execute();
  RESTResponseDataSetAdapter1->Active = true;
}
```

现在编译并且执行您应该就可以在您的 Android 手机中点选右上方的按钮取得台北市旅馆的信息，例如下图就是此时的范例 App 执行在笔者的 S4 和 HTC Inredible S 手机中：

10-2 加入查询旅馆功能

能够成功显示所有旅馆信息之后当然我们会希望能查询特定的旅馆，为了让此范例程序更有趣和有用，让我们试着加入查询的功能，让这个范例 App 能够允许用户藉由写出旅馆的部份名称后就可以自动帮助使用者查询旅馆。

回到主窗体并且在 ToolBar 中放入一个新的 TSpeedButton 做为搜寻旅馆功能：

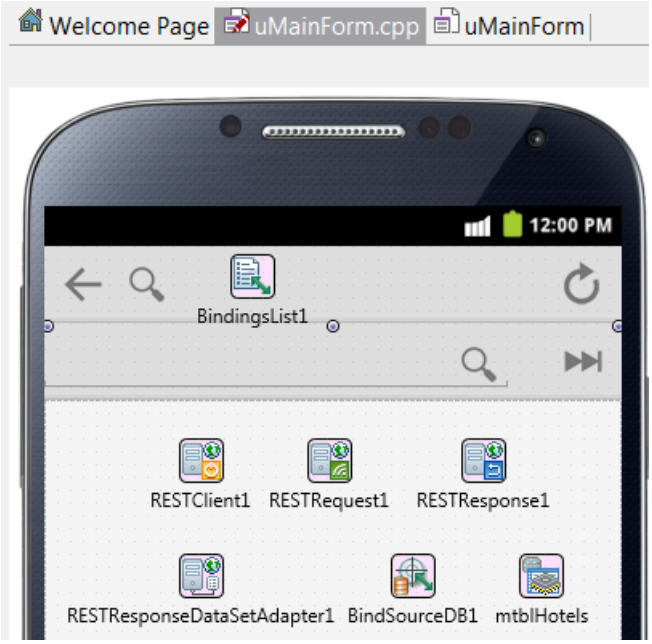


在这个 TSpeedButton 的 OnClick 事件处理函式中撰写如下的程序代码：

```
void __fastcall TfmMainForm::sbtnSpeechSearchClick(TObject *Sender)
{
  if (!mtblHotels->Active)
    GetTaipeiHotelsJSON();
  TabControl1->ActiveTab = TabItem2;
  edtHotelName->SetFocus();
}
```

如果此时尚未取得所有旅馆资料就先呼叫 GetTaipeiHotelsJSON()方法，接着显示 TabControl 的第 2 个 TabItem 页面。

在主窗体的 TabControl 的第 2 个 TabItem 中再放入一个 ToolBar，一个 TEdit，一个 TSpeedButton 和一个 TWebBrowser 组件：



在 TEdit 组件的右方的搜寻 TSpeedButton 的 OnClick2 事件处理函式中撰写如下的程序代码：

```
void __fastcall TfmMainForm::SearchEditButton1Click(TObject *Sender)
{
  if (edtHotelName->Text != "")
    SearchHotel(edtHotelName->Text);
}
```

SearchEditButton1Click 先判断使用者是否有输入任何旅馆名称的字符，如果有的话就呼叫 SearchHotel()方法在 mtblHotels 中搜寻旅馆信息。

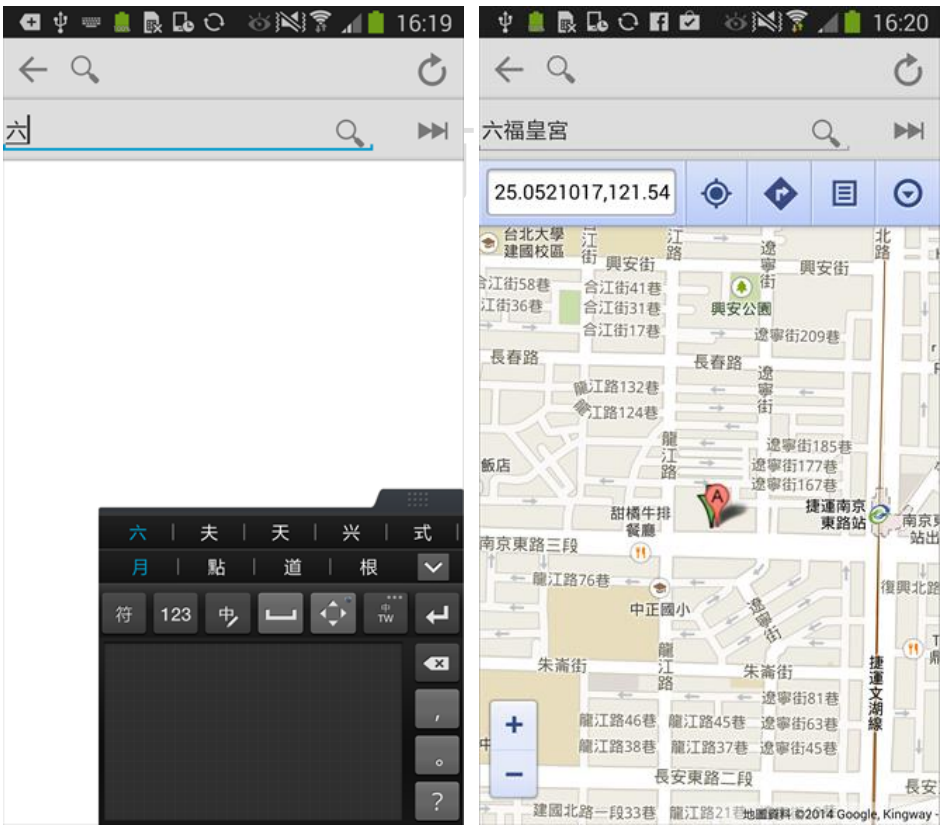
SearchHotel()方法使用 TFDMemTable 组件的 LocateEx()方法在 mtblHotels 中搜寻数据，一旦找到了用户输入的旅馆名称之后就呼叫 DisplaySearchedHotel()方法显示目标旅馆：

```
void TfmMainForm::SearchHotel(const String sData)
{
  Firedac::Comp::Dataset::TFDDataSetLocateOptions Opts;
  Opts << lxoCaseInsensitive;
  Opts << lxoPartialKey;

  if (mtblHotels->LocateEx("stitle", sData, Opts) )
    DisplaySearchedHotel();
}
```

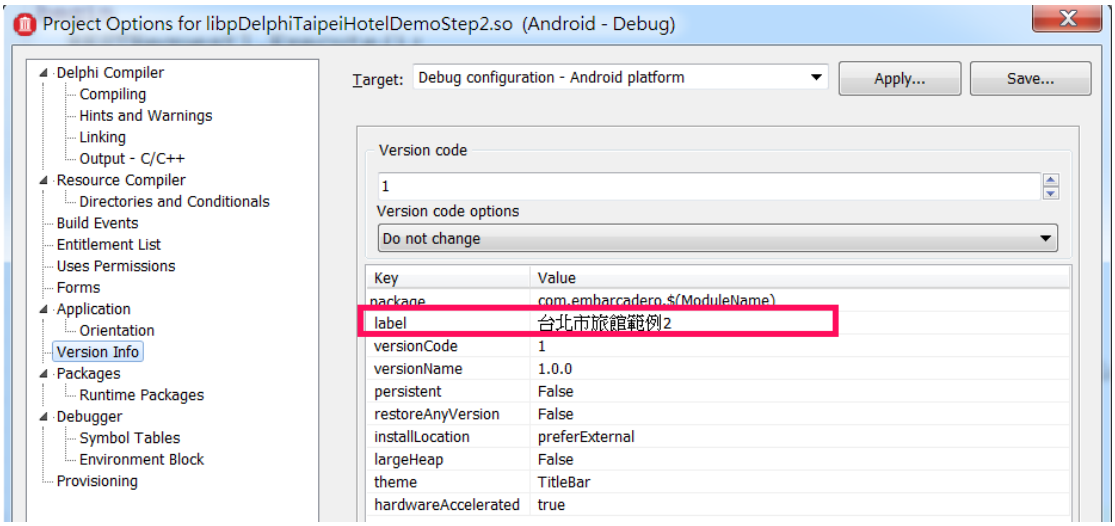
DisplaySearchedHotel()方法藉由从 mtblHotels 中取出搜寻旅馆的经/纬度信息再使用 TWebBrowser 组件显示目标旅馆的所在地：

```
void TfmMainForm::DisplaySearchedHotel()
{
  edtHotelName->Text = mtblHotels->FieldByName("stitle")->Value;
  String sHotelLongitude = mtblHotels->FieldByName("longitude")->Value;
  String sHotelLatitude = mtblHotels->FieldByName("latitude")->Value;
  String URLString = Format("https://maps.google.com/maps?q=%s,%s",
ARRAYOFCONST((sHotelLatitude, sHotelLongitude)));
  WebBrowser1->Navigate(URLString);
  TabControl1->ActiveTab = TabItem2;
}
```

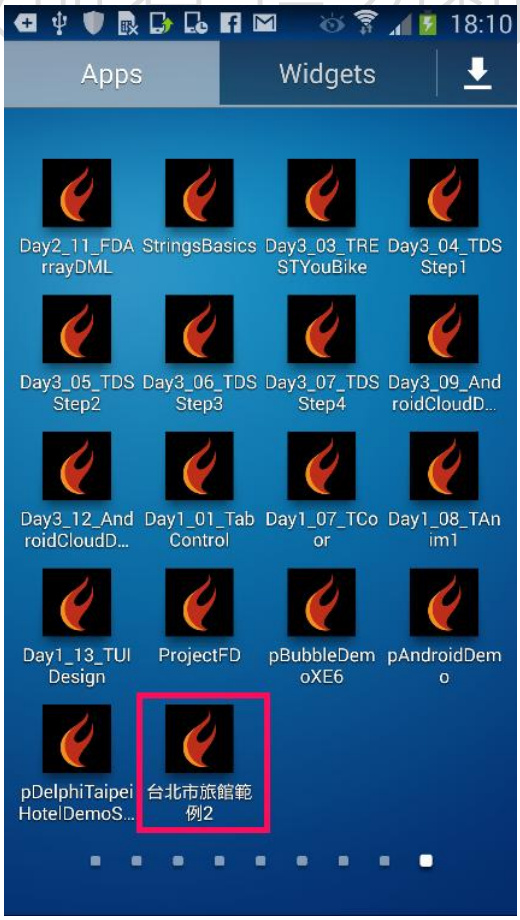
下图是笔者在 S4 上搜寻六福皇宫的画面，读者可以看到笔者只输入了”六”之后点选旁边的搜寻按钮之后就可以正确的找到” 六福皇宫”了：



到现在为止当我们部署范例 App 到手机中时看到的范例 App 名称都是英文的项目名称，现在让我们为它设定中文的部署名称吧。请在项目管理员中右击项目开启项目的 Options...对话盒，在 Version Info 选项中的 Label 选项中输入您要设定的中文部署名称，例如下图使用了"台北市旅馆范例 2":

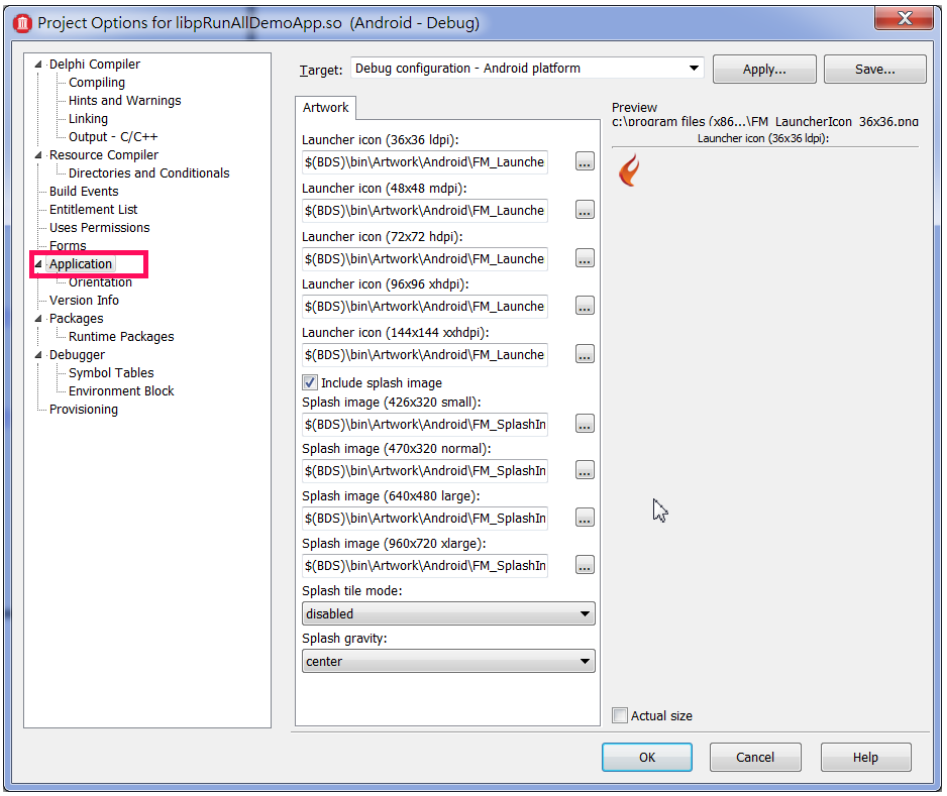


现在您可再次部署此范例 App 到手机中，您应该就可以看到中文的 App 名称了，例如下图就是此范例 App 部署到 S4 手机中的结果画面：

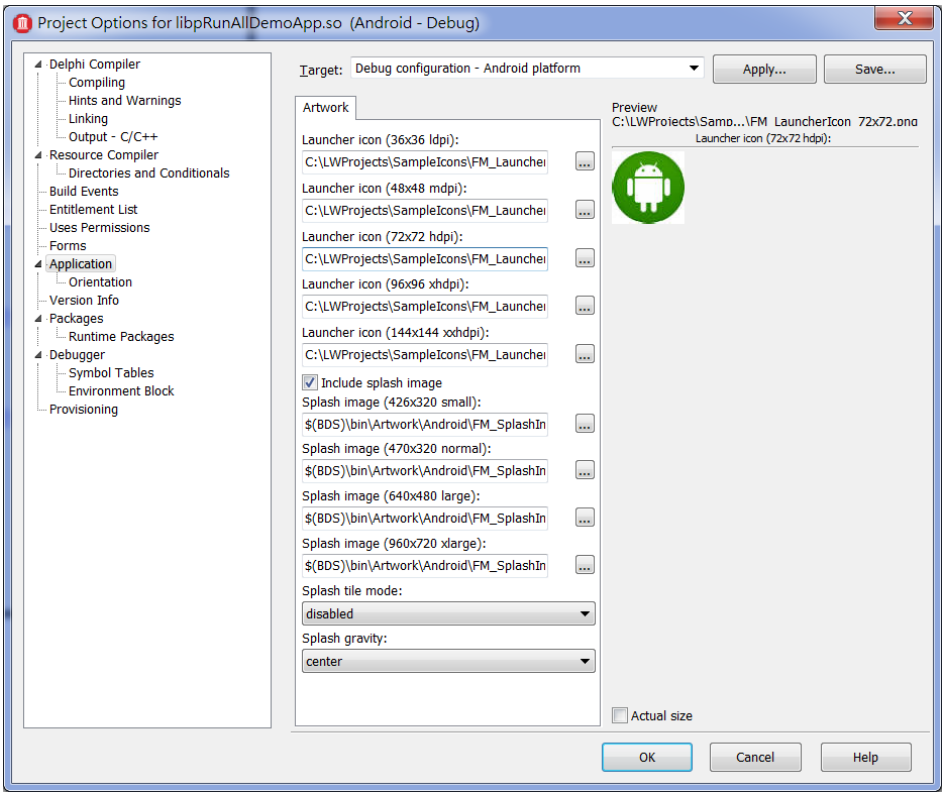


10-3 为您的 App 设计个图像吧

当您使用 C++Builder 开发完 App 之后需要部署到手机中，那么您一定希望能使用客制化的图像来代表您的 App 而不是使用 C++Builder 内定的图像。在中要为 App 更改客制化图像非常的简单，请点选 Project | Options 选单，在显示的对话盒 Application 项目中可以看到 C++Builder 内定使用的各种大小的 App 图像，如下所示：



您可以设计同样大小的图像来使用，例如 36x36，48x48 等图像，然后点选每个图像旁的...按钮并且加载使用您的客制化图像。例如下图就是笔者在此对话盒 Application 项目中加载客制化图像：

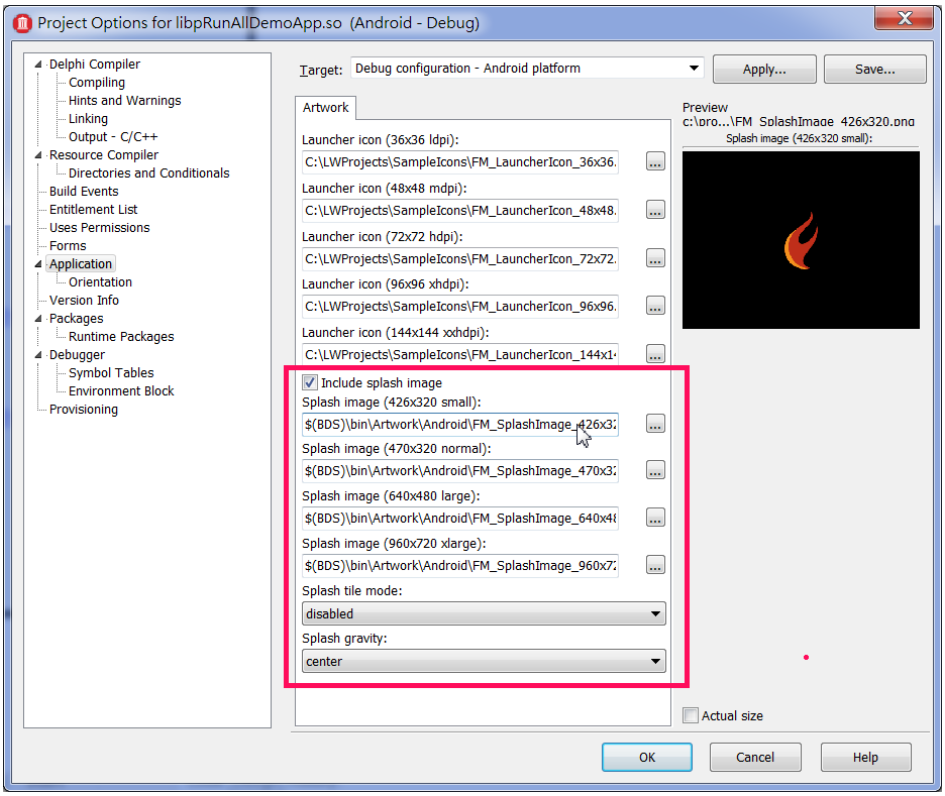


接着重新编译并且部署您的 App，您就可以在手机中看到您的 App 现在使用了您自己的客制化图像：

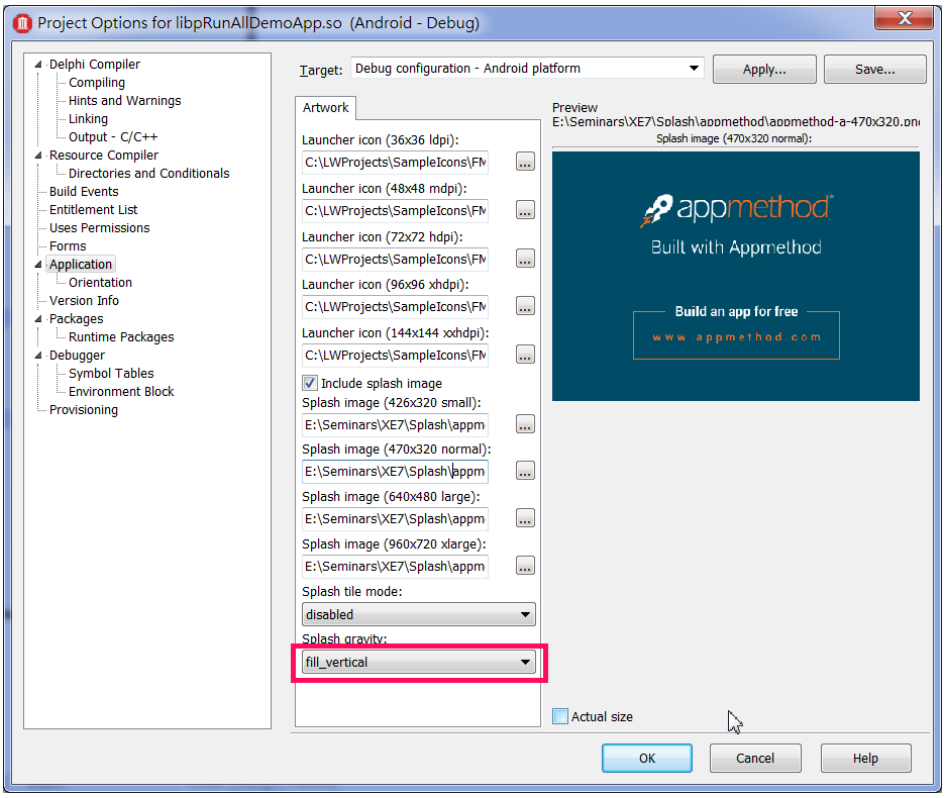
10-4 为您的 App 加入启动屏幕吧

除了 App 图像之外，也允许开发人员为 App 加入启动时的欢迎画面，由于编译出的 App 是原生的 App，App 体积比较大一点因此需要多一点的加载时间，为了让您的 App 在一开始执行时有比较快的反应时间，您可以为 App 加入欢迎画面。

您同样可以藉由点选 Project|Options 选单，在刚才加入客制化图像的下方可以看到 Splash Image 的选项：



同样的您可以自行设计您的启动屏幕影像，然后点选每个图像旁的...按钮并且加载使用您的客制化影像即可，例如下图就是加载客制化影像并且设定使用 fill_vertical 方式显示：



部署此 App 并且执行就可以看到如下的启动屏幕：

10-5 为您的 App 加入 Provisioning 信息吧

当您开发完并且决定部署您的 App 并且如上加入了客制化图像和启动屏幕影像后，请记得再对 App 进行部署开通服务(Provisioning)的设定，一旦完成了部署开通服务您的 App 不但能够部署到非开始模式的手机，也可以部署到 Application Store 中。

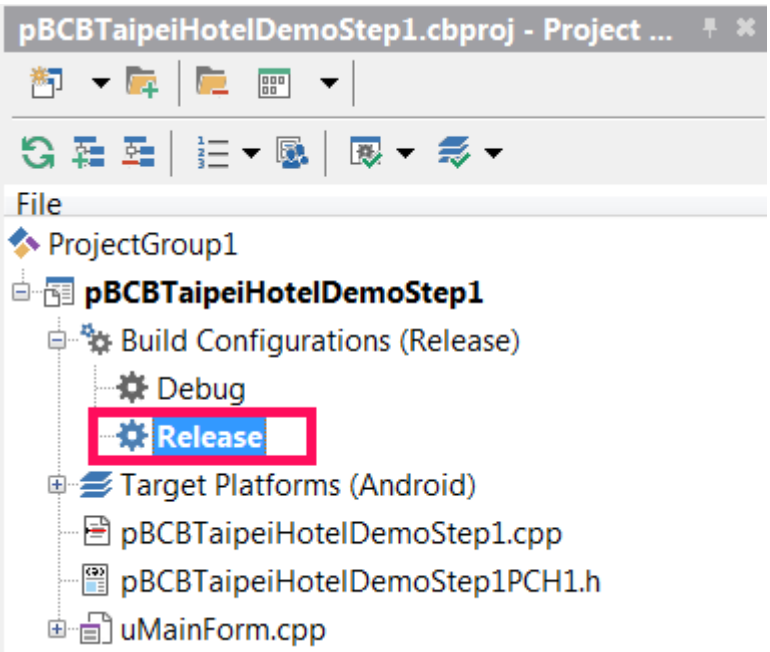
要对 App 进行 部署开通服务，您需要完成下列的工作：

1. 设定 Build Configurations 为 Release
2. 开启 Target Platforms 设定为 Android 平台并在 Configuration 中选择 Application Store
3. 到 Project > Options > Provisioning 页面填写必要的信息
4. 维护 App 版本信息

下面进行详细的说明。

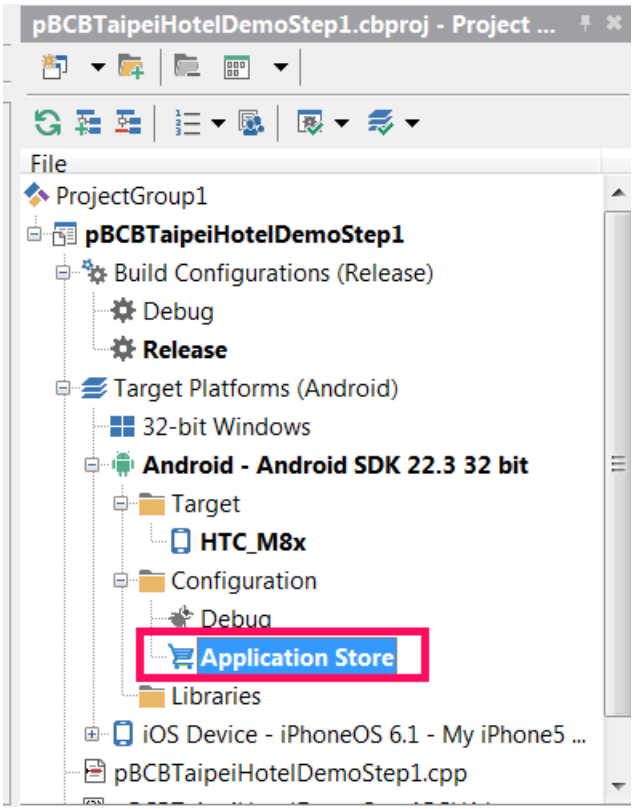
设定 Build Configurations 为 Release

我们在 IDE 中开发 App 时都是使用 Debug 模式，但在实际部署时一定要改为 Release 模式重新编译一次再部署。要设定为 Release 模式，请在项目管理员中双击 Build Configurations 中的 Release 节点：



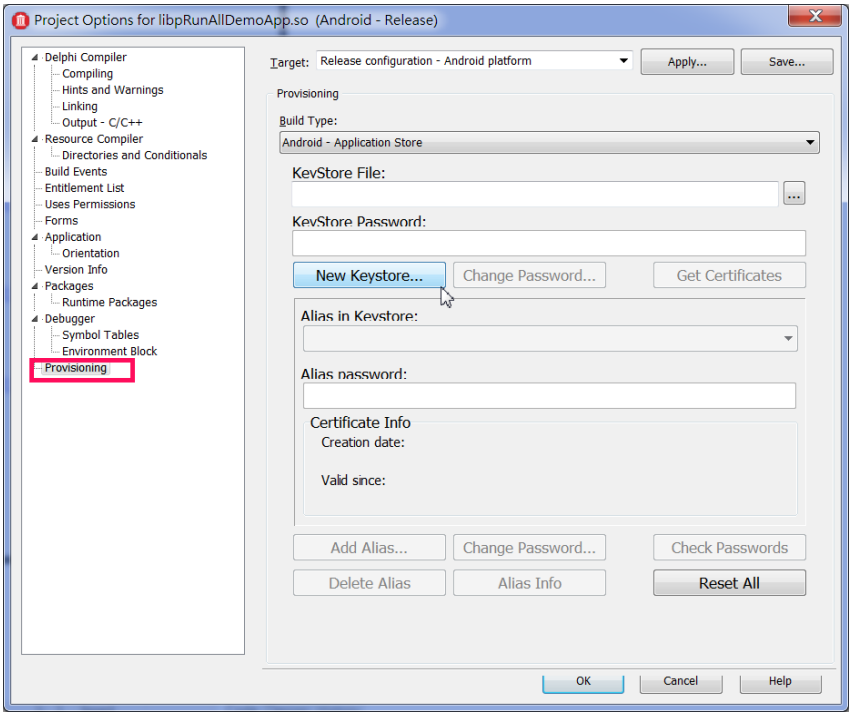
开启 Target Platforms 设定为 Android 平台并在 Configuration 中选择 Application Store

接着同样在请在项目管理员中双击 Target Platforms | Configuration 中的 Application Store 节点:

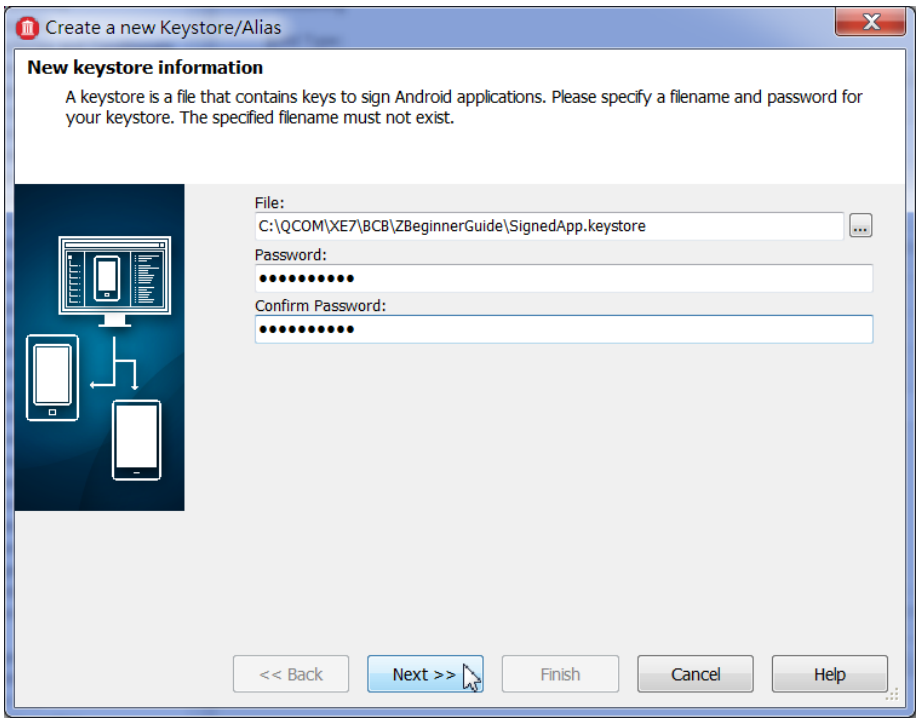


到 Project > Options > Provisioning 页面填写必要的信息

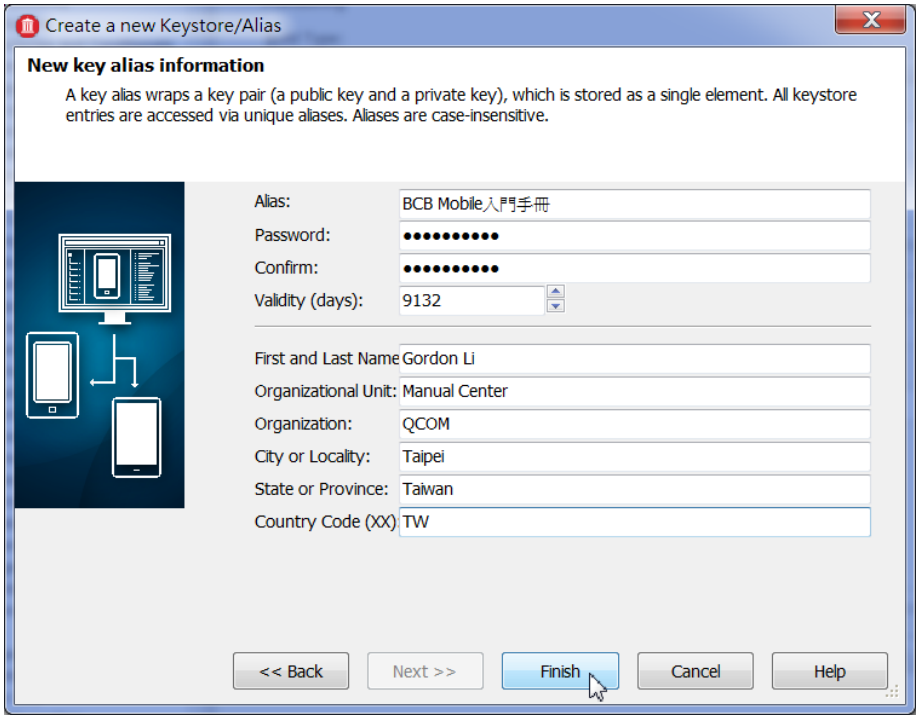
点选 Project|Options 选单，再点选 Provisioning 节点：



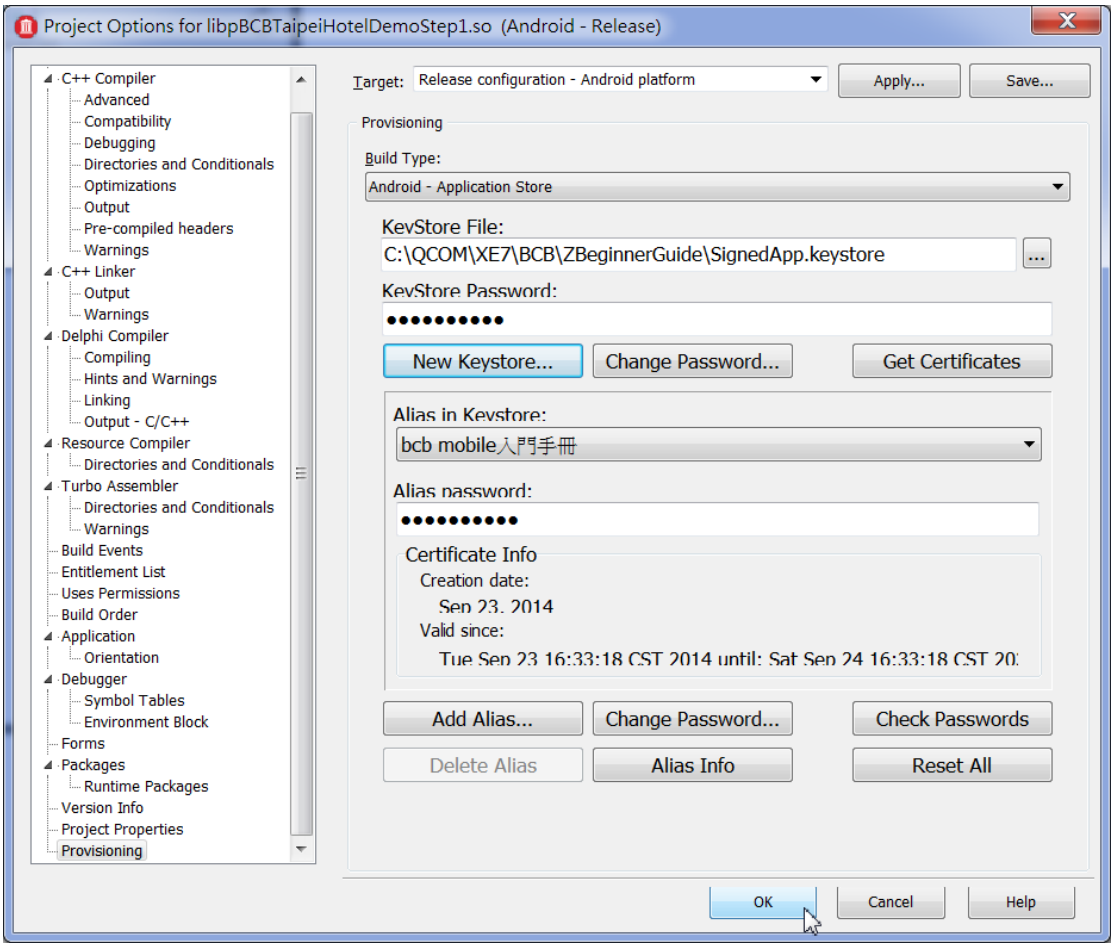
接着点选其中的 New Keystore...按钮，选择一个要建立 Keystore 文件的目录和文件名并且输入密码：



点选 Next 按钮，在下一个对话下输入他的别名(Alias)，密码等信息，如下所示：

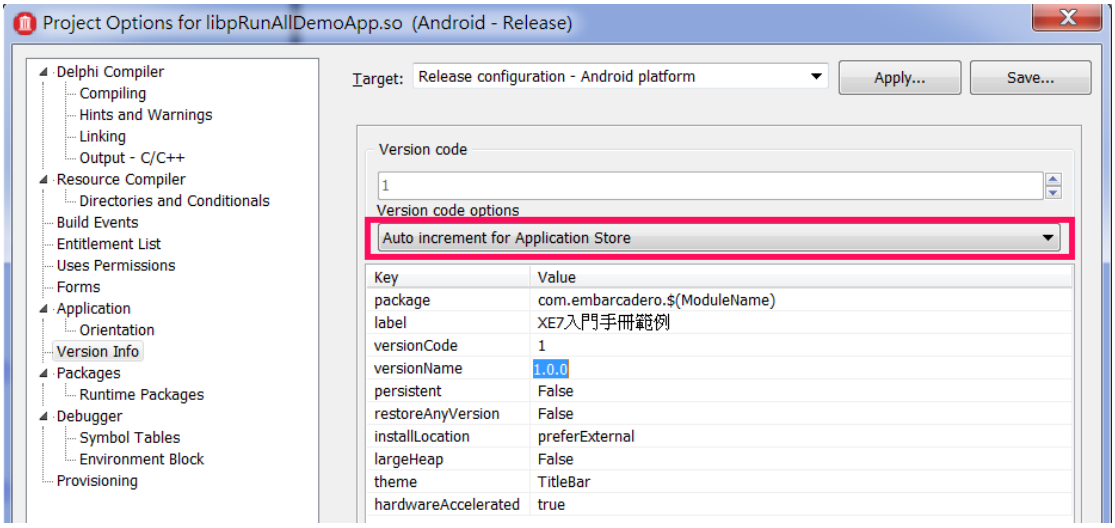


点选 Finish 按钮之后就可以回到 Provisioning 节点并且看到刚才设定的信息都已经自动带入到 Provisioning 的各个字段中了。刚才建立的 Keystore 档是这个 App 的 Provisioning 信息，请像 App 的原始程序一样妥善保存好。



维护 App 版本信息

最后请设定你的 App 版本信息，您可以点选 Project|Options 选单，再点选 Version Info 节点中找到您的 App 版本信息。要让 IDE 自动维护您的 App 版本信息,请选择 Auto increment for Application Store 选项：



那么您每一次重新 Build 您的项目 Version code 就会自动增加，当然您也可以使用 versionName 特性来显示您的 App 的版本信息，例如更改 versionName 特性值和您的 Version code 一致后就可以在手机中看到您的 App 的版本信息了：

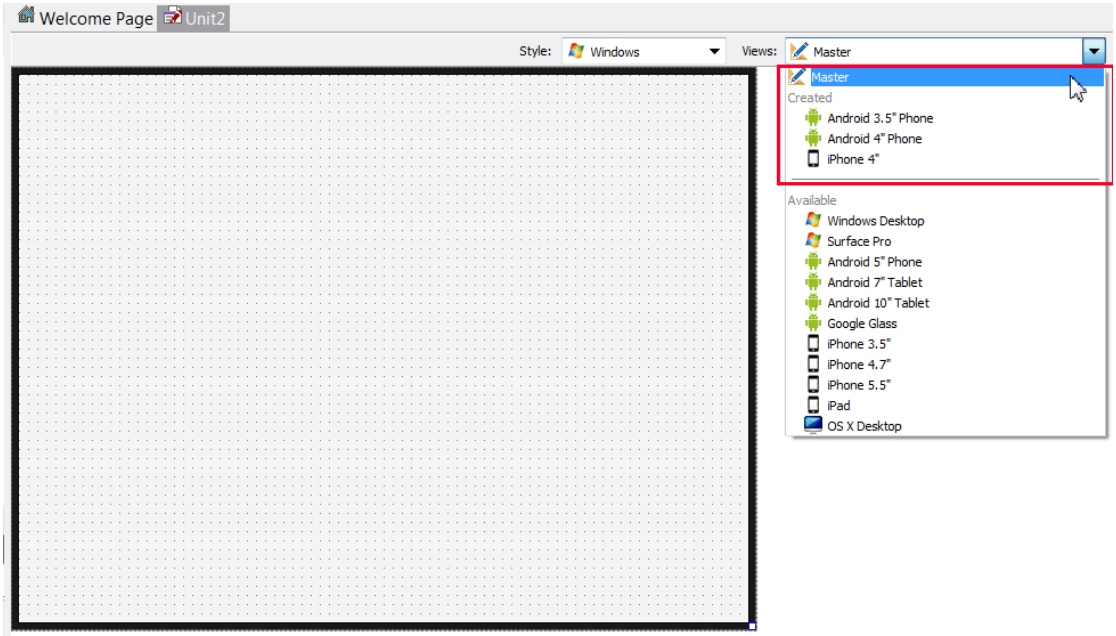
11 新功能

C++Builder 中的新功能：MultiView 设计家，版本控制功能 Git 和 DUNITX 测试框架是笔者认为非常重要的 3 大功能。这是因为 C++Builder 支持多平台的开发能力，但如何在多个平台开发时能够减少 UI 的设计工作，如何能够最大化的使用相同的程序代码便成为了开发人员最须重视的问题。而 MultiView，Git 和 DUNITX 正是这 2 个问题的解决方案，因此在本章中将先为读者介绍这 3 个功能。

11-1 MultiView

MultiView 功能是从 XE7 版本开始出现的，它的目标是让开发人员节省多平台开发的时间进而提升生产力。在中 MultiView 继续获得了强化，加入了 Multi-Device 功能允许开发人员可同时检视所有开发平台的 UI 接口。

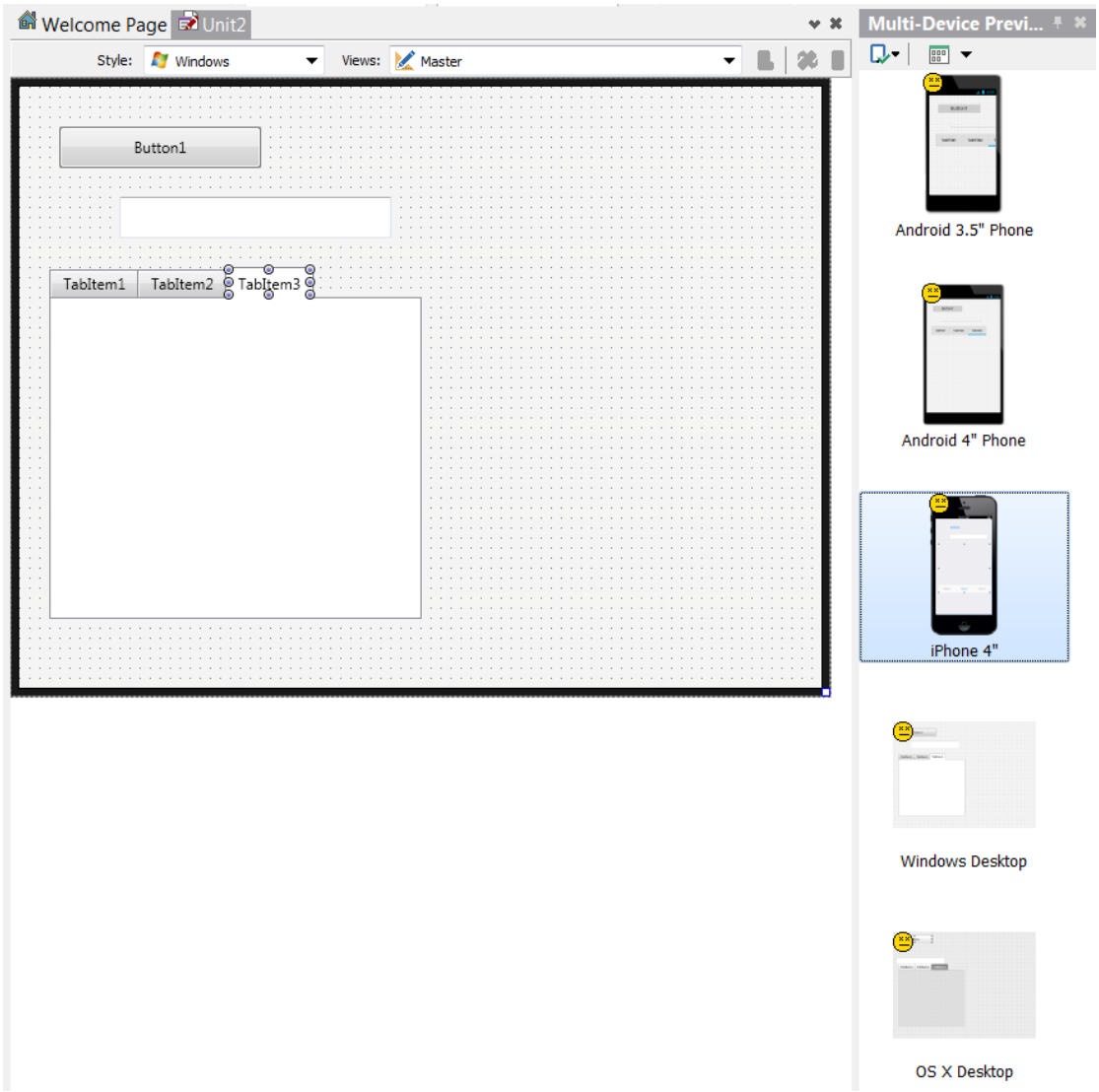
Multi-Device 在使用上非常的简单，只要在开发项目时开启 Multi-Device 设计窗口即可。例如下图是一个 Multi-Device 项目，它同时要开发 Android 3.5 吋，Android 4 吋和 iPhone 吋的设备：



在开始设计时请点选 View|Multi-Device Preview 选单并且在 Master View 中开始放入可视化组件，就可以看到类似下图的画面，View|Multi-Device

Preview 会同时显示在设计 UI 时，每一个开发设备的 UI 会如何显示。而且只要使用鼠标双击 View|Multi-Device Preview 窗口中特定的设备图像，IDE 便会切断到点选的特定设备的可视化设计接口。

View|Multi-Device Preview 功能在使用上非常的简单，但却可提供开发人员非常高的设计 UI 的生产力。



11-2 使用分布式版本控制工具-Git

C++Builder XE7 便开始支持分布式版本控制工具 Git，但 XE7 只提供了最基本的 Git 功能，到了对于 Git 的支持就比较完整了。本小节的内容将说明如何在 C++Builder IDE 中使用 Git 进行版本控制功能。

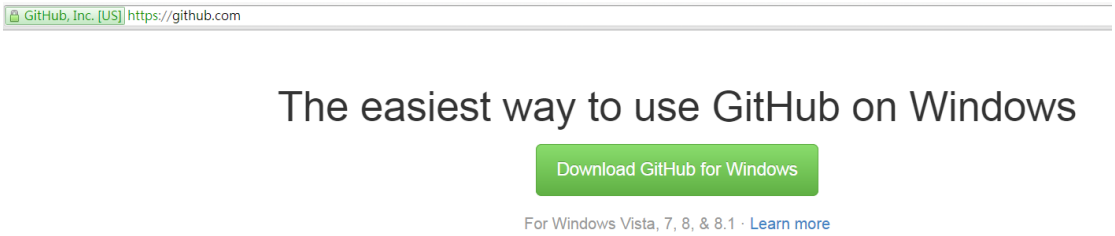
要在 IDE 中使用 Git，开发人员需要进行下面的步骤：

1. 下载和安装 Git

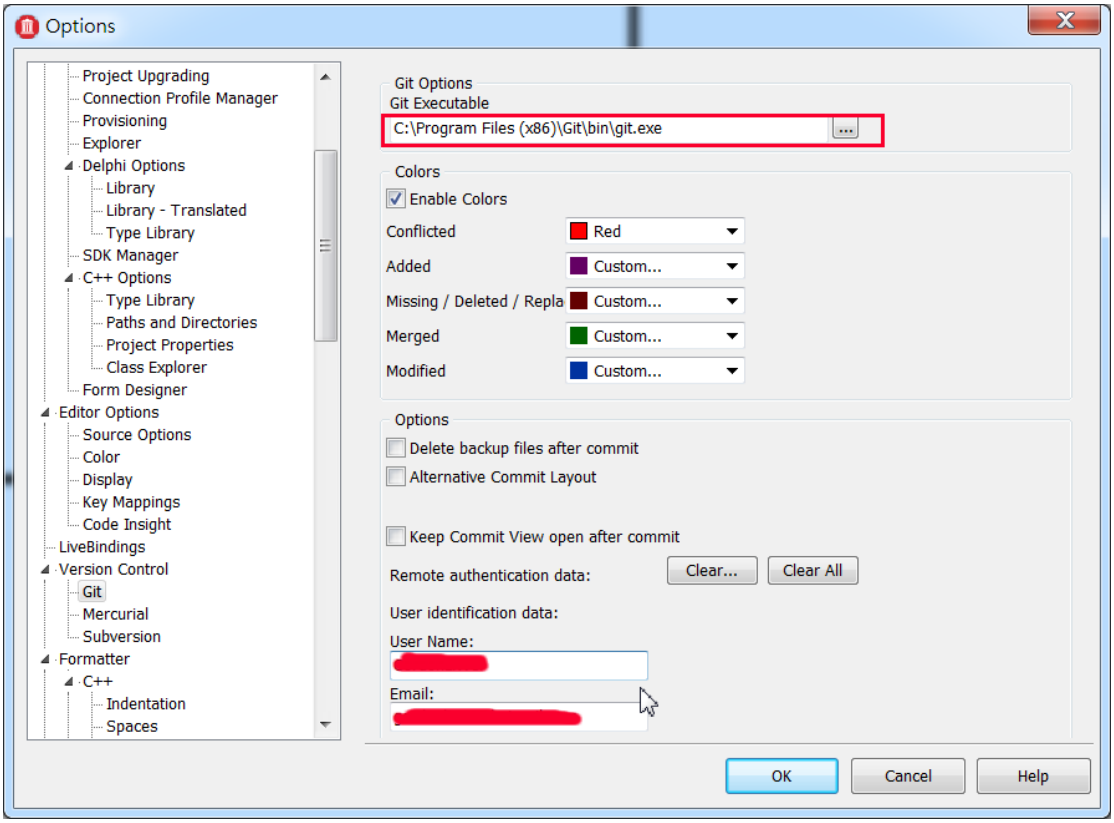
2. 在 IDE 中设定 Git

3. 申请账号

要进行第 1 步骤，读者可到 https://github.com/下载 Git：



下载 Git 之后请安装它，安装完之后请到 C++Builder IDE 中点选 Tools|Options 选项，在 Version Control|Git 类别中请在 Git Executable 字段中输入您安装的 Git 执行文件位置：



在上面的对话盒中还需要您输入您申请的 Git 帐户信息，这是第 3 个步骤。

要使用 Git，开发人员需要先到 GitHub 或是 BitBucket 申请使用空间，

读者可到

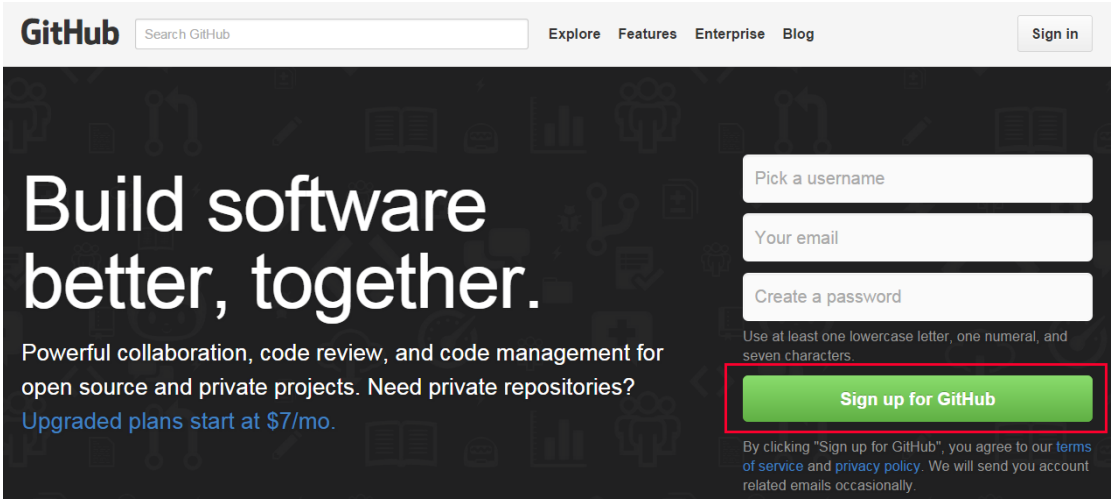
```
https://github.com/
```

申请使用 GitHub，或是到

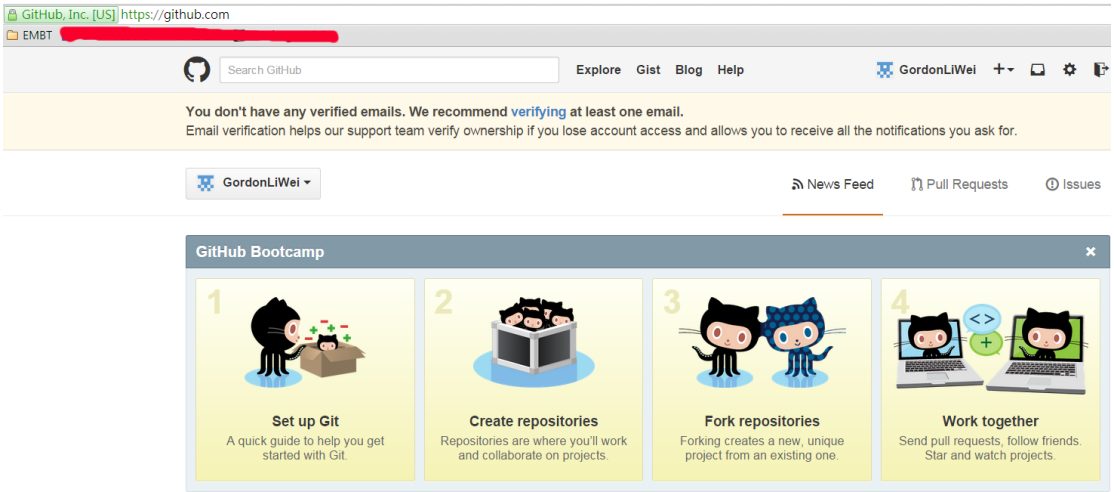
```
https://bitbucket.org/
```

申请使用 BitBucket。在下面的内容中笔者以 GitHub 做为说明。

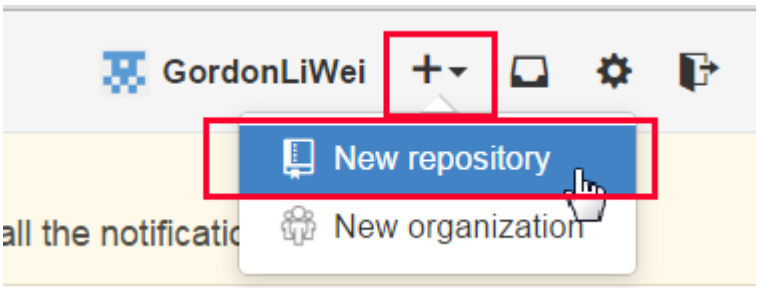
首先到 https://github.com/ 点选如下图的”Sign up for GitHub”并申请账号：



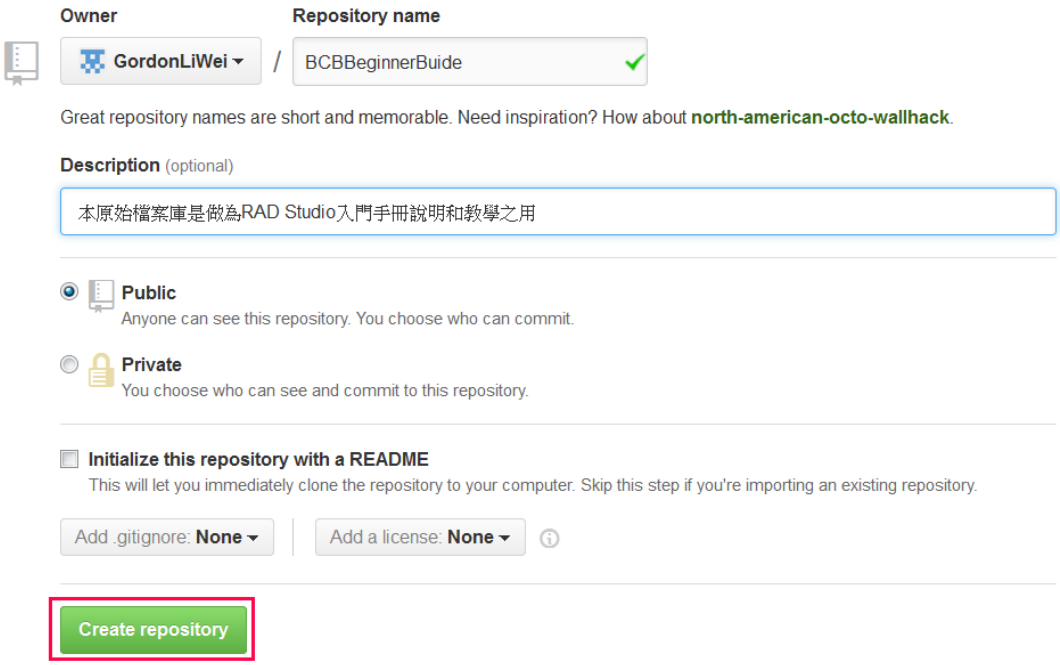
在成功申请账号并登入之后可看到如下的画面，不过申请完账号后请记得回到 IDE 的 Tools | Options | Version Control | Git 类别输入您的帐户信息：



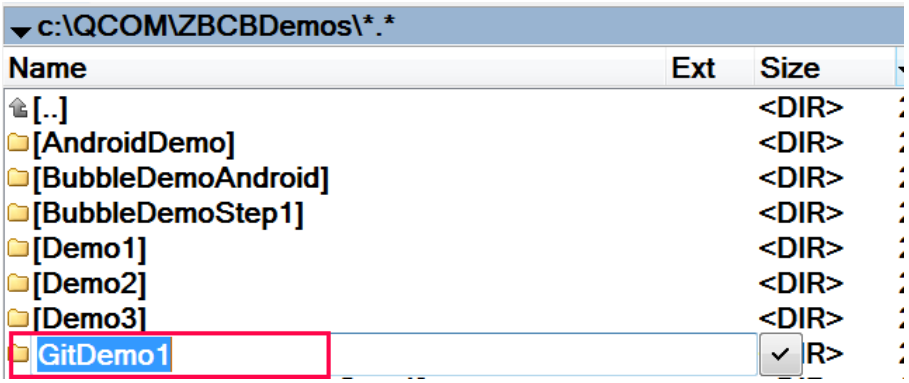
此时可点选您账号名称右边的”+”号以建立一新的链接库，如下所示：

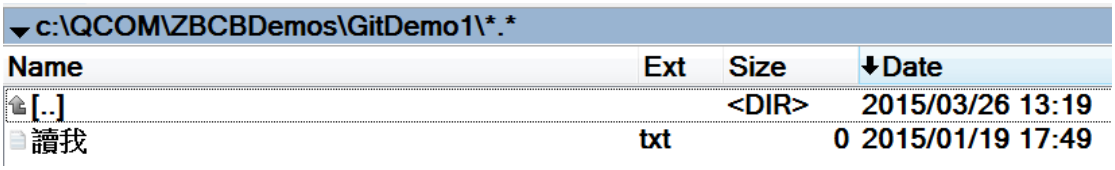


点选”+”号之后会看到如下的类似画面，您需要为新的链接库取一名称和输入一简短的说明之后点选下方的”Crate repository”按钮真正的建立链接库：

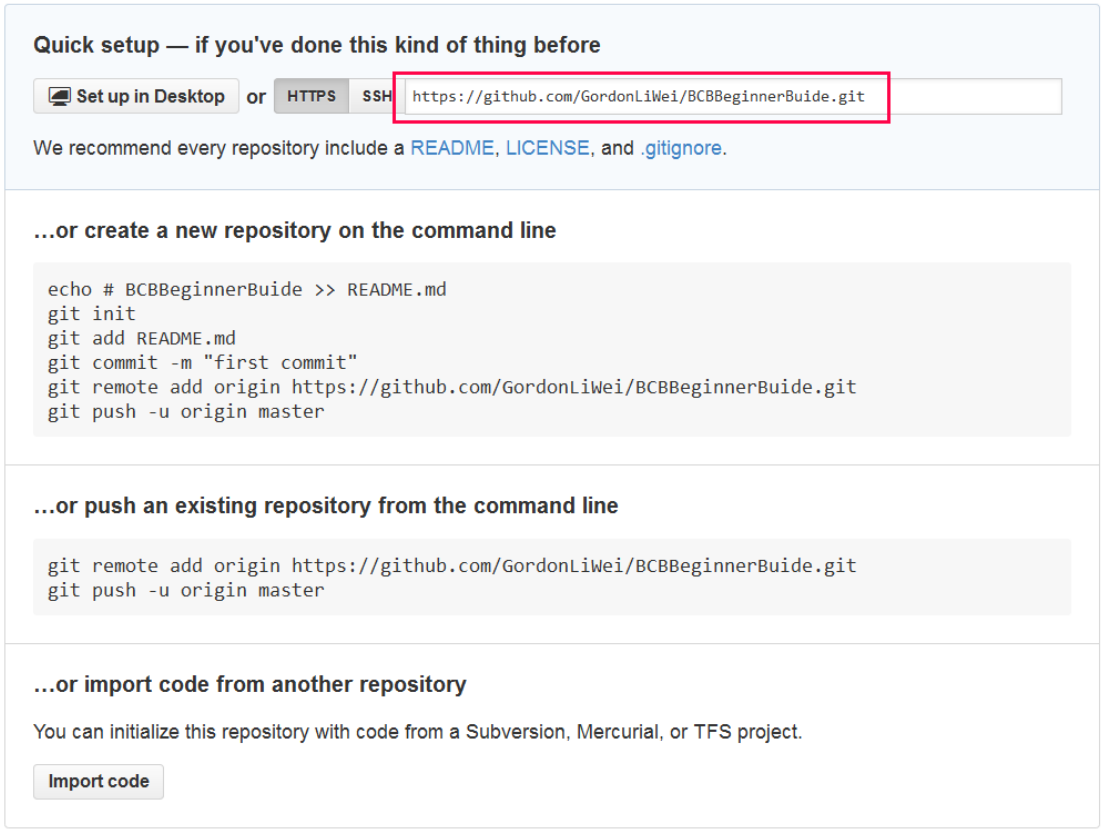


回到你的计算机中，我们现在需要在本机中先建立一个文本文件并签入到远程的刚才建立的链接库中。下图是笔者在本机的 GitDemo1 目录中建立一个”读我.txt”文本文件：



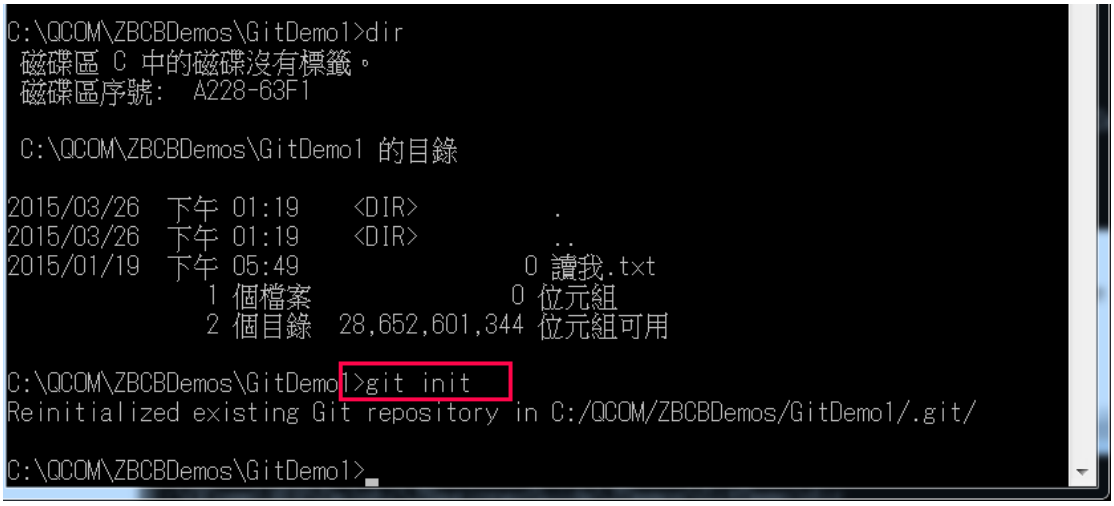


事实上当您建立了链接库时 GitHub 便会显示如下的内容教导您如何把一个档案签入到链接库中：

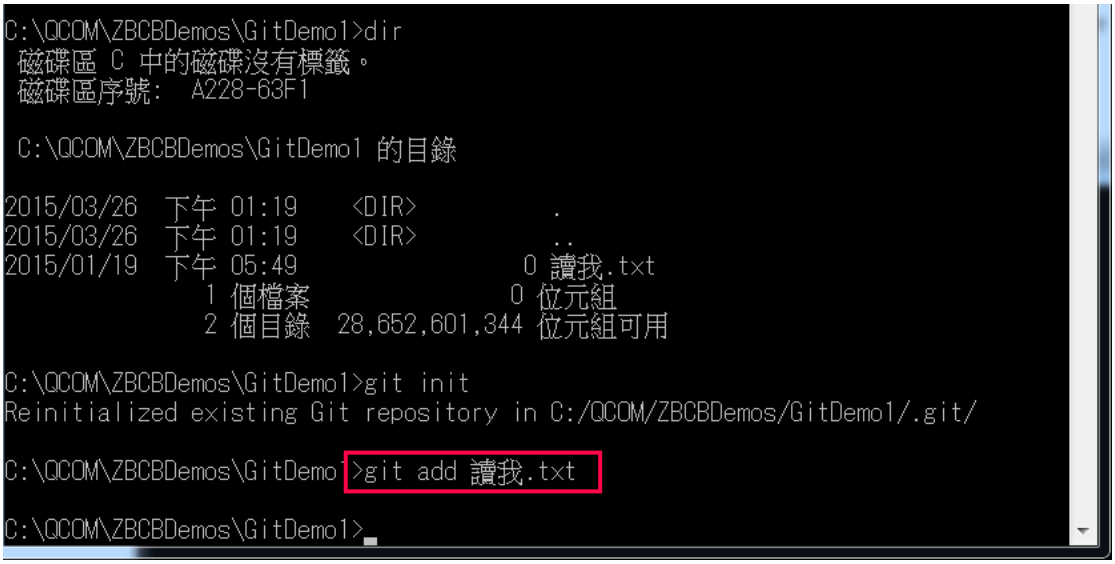


现在就让我们使用上面的说明把 GitDemo1 目录中”读我.txt”文本文件签入到远程的 GitHub 中。

开启一个命令行窗口并且到 GitDemo1 目录执行”git init”命令，Git 便会在此目录建立本机链接库信息：



再执行”git add 读我.txt”命令把”读我.txt”文本文件加入到 git 的 stage 档案串行中：



再执行”git commit –m 第 1 个 Commit”命令签入此文本文件到本机的链接库中：

接下来我们就准备把本机签入的”读我.txt”文本文件同步到远程的 GitHub 的链接库中，但在这之前您可能需要执行”git config –global user.email 您的 email 地址”让 Git 知道要签入到远程的什么地方：

```
E:\QComm\XE8\Delphi\ZBeginnerGuide\ZDemos\GitDemo1>git config --global user.emai
l "gordonliwei@hotmail.com"
```

最后执行"git remote add origin https://github.com/GordonLiWei/您的远程链接库名称"命令连结本机和远程链接库，再执行" git push -u origin master"命令把本机的"读我.txt"文本文件从本机的链接库真正签入到远程的链接库中，在这一步骤中您可能需要输入登入信息：

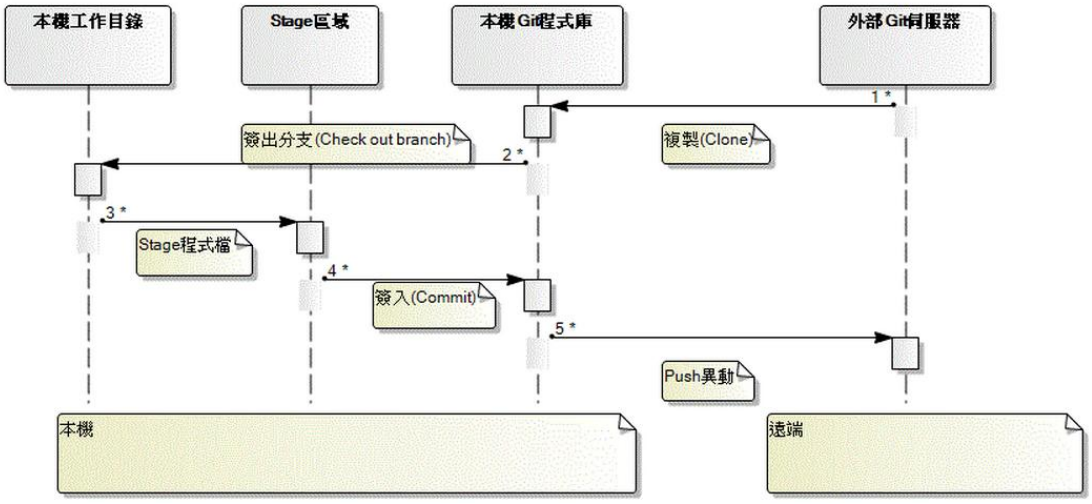
```
C:\QCOM\ZBCBDemos\BCBGitDemo1>git remote add origin https://github.com/GordonLiW
ei/BCBBeginnerBuide.git

C:\QCOM\ZBCBDemos\BCBGitDemo1>git push -u origin master
Username for 'https://github.com': GordonLiWei
Password for 'https://GordonLiWei@github.com':
Counting objects: 3, done.
Writing objects: 100% (3/3), 228 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/GordonLiWei/BCBBeginnerBuide.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.

C:\QCOM\ZBCBDemos\BCBGitDemo1>
```


最后回到浏览器并查看远程 GitHub 的链接库就可以看到"读我.txt"文本文件已经出现在其中了：

到了这里我们就可以开始使用 C++Builder IDE 来进行项目的版本控制了，而我们使用的链接库就是刚才在远程于 GitHub 中建立的链接库。但在说明如何在 C++Builder IDE 中使用 GitHub，先让我们说明一下什么是分布式版本控制，一旦读者了解之后就能明白前面和稍后在 C++Builder IDE 中执行的动作是什么意思。

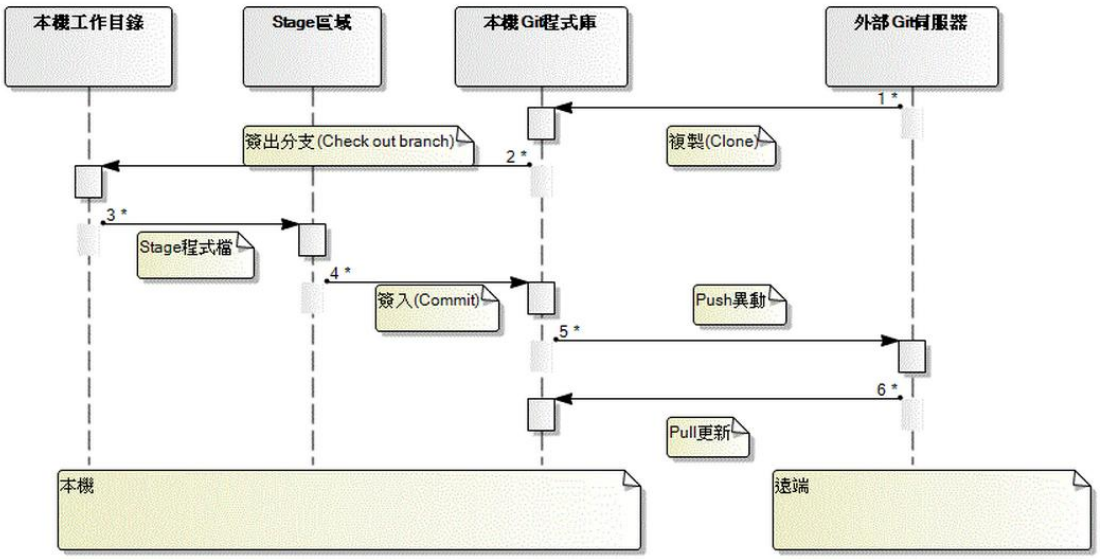
传统的版本控制工具，例如 StarTeam 和 SubVersion 等都是属于中央控制的 C/S 架构，每一个客户端的开发人员从版本控制服务器签出程序开发和异动，最后再签入回版本控制服务器。但 Git 是属于分布式的版本控制系统，Git 除了有远程版本控制服务器之外(即 GitHub)，Git 也会在本机建立本机链接库，Git 可同步远程版本控制服务器和本机的链接库。

让我们使用下面的图形简单说明 Git 基本的运作原理 ，在前面我们于 GitHub 上建立的链接库就类似下图中的外部 Git 服务器，一开始我们从外部 Git 服务器复制程序到本机 Git 链接库，接着我们签出分支，进行开发的工作。开发到一段落之后我们可 Stage 开发的程序文件，确定异动之后再签入异动到本机 Git 链接库。到此所有的开发，异动和签出/签入都是在本机之中，因此可减少外部 Git 服务器的负荷。当本机的开发到一段落之后我们可以 Push 本机签入的结果到外部 Git 服务器，如此一来团队其他的成员就可以复制/签入这些开发成果，再继续进行团队开发。



Git 基本运作原理

如果团队中其他成员开发到一段落那么外部 Git 服务器可以主动再把它们 Push 到你的本机 Git 链接库我们就可以再签出进行后续开发。当然本机 Git 链接库也可以主动执行 Pull 命令从外部 Git 服务器更新本机 Git 链接库：

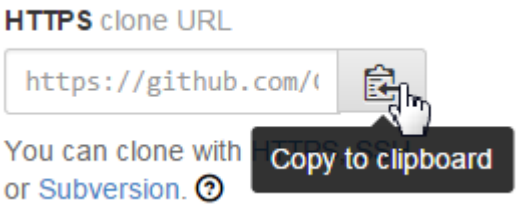


从上面的简单说明读者应该可以了解为什么现在 Git 这么流行，因为 Git 提供了分布式的版本控制能力，又能整合远程 Git 服务器和本机 Git 链接库进行团队开发。

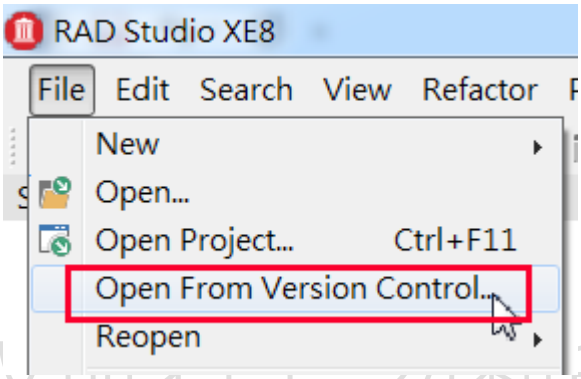
在具备了 Git 基本的观念后，我们就可以开发解释如何在中使用 Git，笔者在 GitHub 中建立了一个”BCBGitDemos”链接库做为上图中的远程服务器：

请注意上图中这个远程链接库目前的分支是”Master”，而在上图右下角的"HTTPS clone URL”处即是指向此远程链接库的 URL，稍后在 C++Builder IDE 中将使用这个 URL 复制(clone)程序到本机 Git 链接库，也就是前图” Git 基本运作原理”中的第 1 步骤。

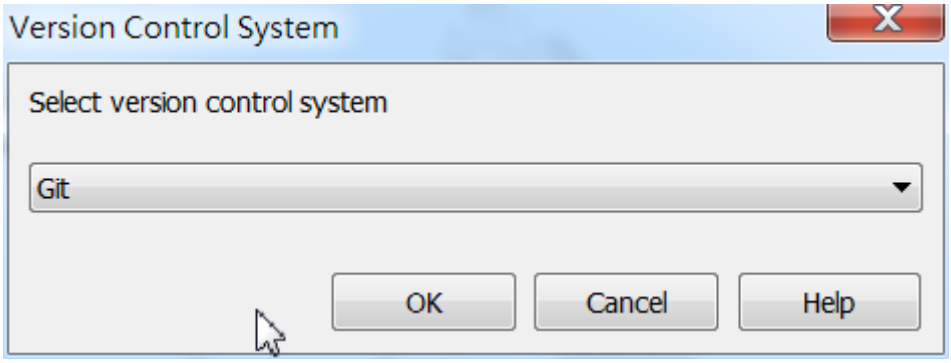
在 GitHub 的”BCBGitDemos”链接库页面中点选拷贝”BCBGitDemos”链接库 URL：



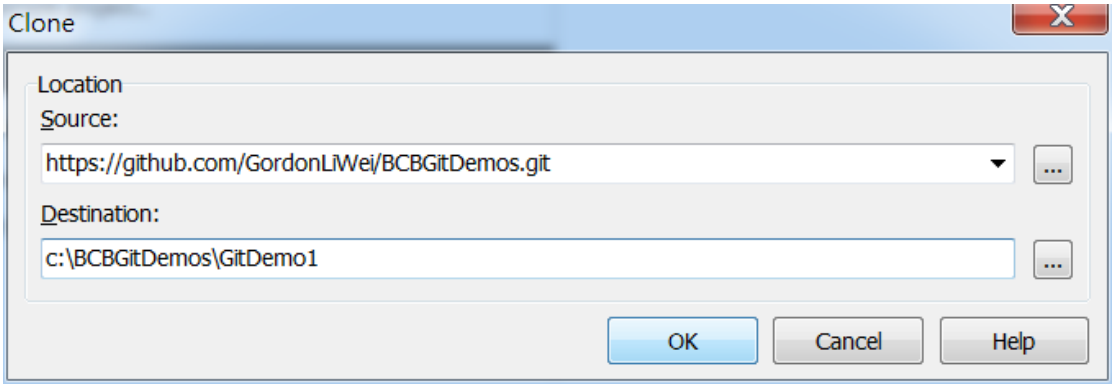
现在回到 C++Builder IDE，点选 File | Open From Version Control...选项：



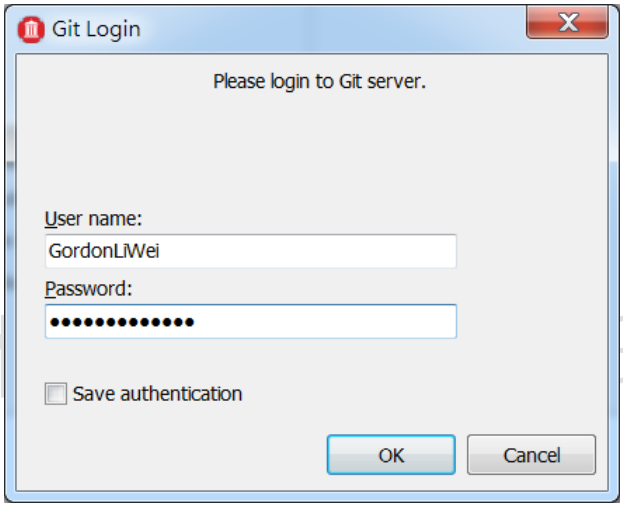
从 Version Control System 对话盒中选择 Git：



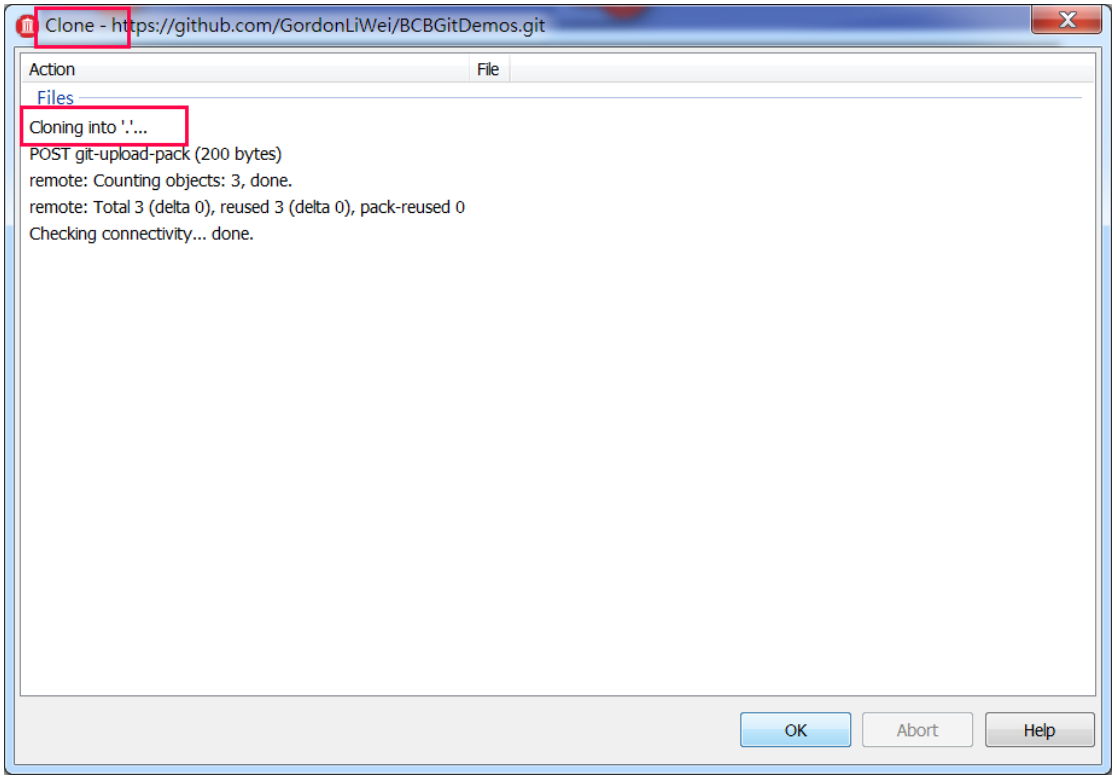
点选上图的 OK 按钮后在 Clone 对话盒中的 Source 字段中贴上前面拷贝的”BCBGitDemos”链接库 URL，再于 Destination 字段中输入复制(clone)到本机 Git 链接库的目录：



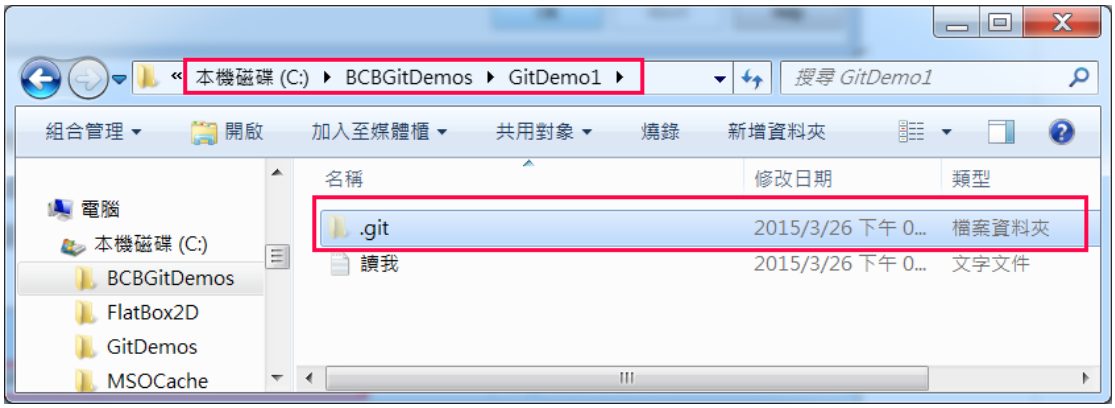
最后点选 OK 按钮，就可以看到 C++Builder IDE 显示如下的复制对话盒，此时 IDE 会要求您登入 Git：



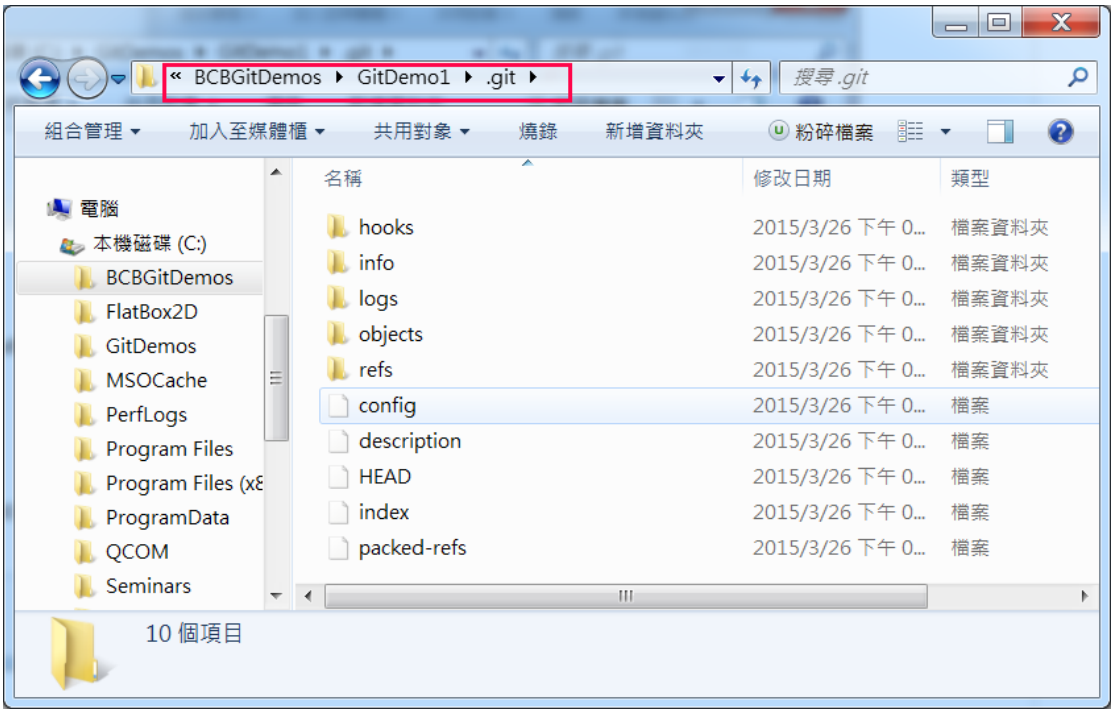
输入前面申请建立的 Git 帐户信息后再点选 OK 按钮，您可以看到它显示 Git 正在复制 (cloning) 远程的链接库内容到本机的 c:\BCBGitDemos\GitDemo1 目录中。这印证了 C++Builder IDE 正在执行前图" Git 基本运作原理"中的第 1 步骤：



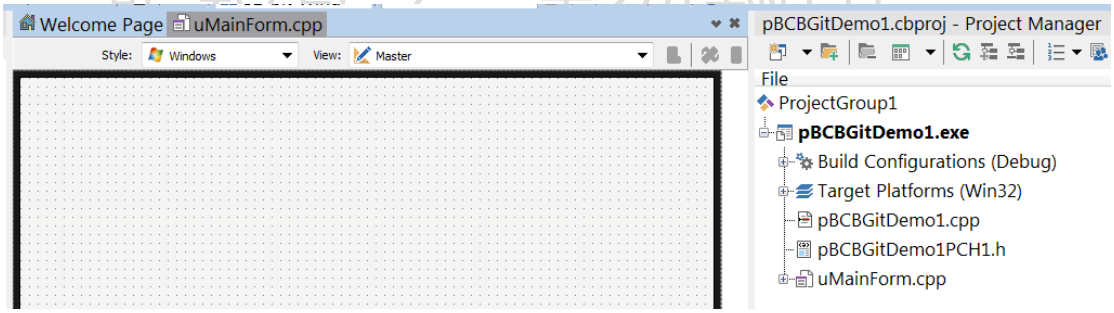
点选 OK 按钮之后到 c:\BCBGitDemos\GitDemo1 目录中查看，我们可以看到在 c:\BCBGitDemos\GitDemo1 目录中果然复制了远程"读我.txt"而且在目录中出现了一个 .git 子目录：



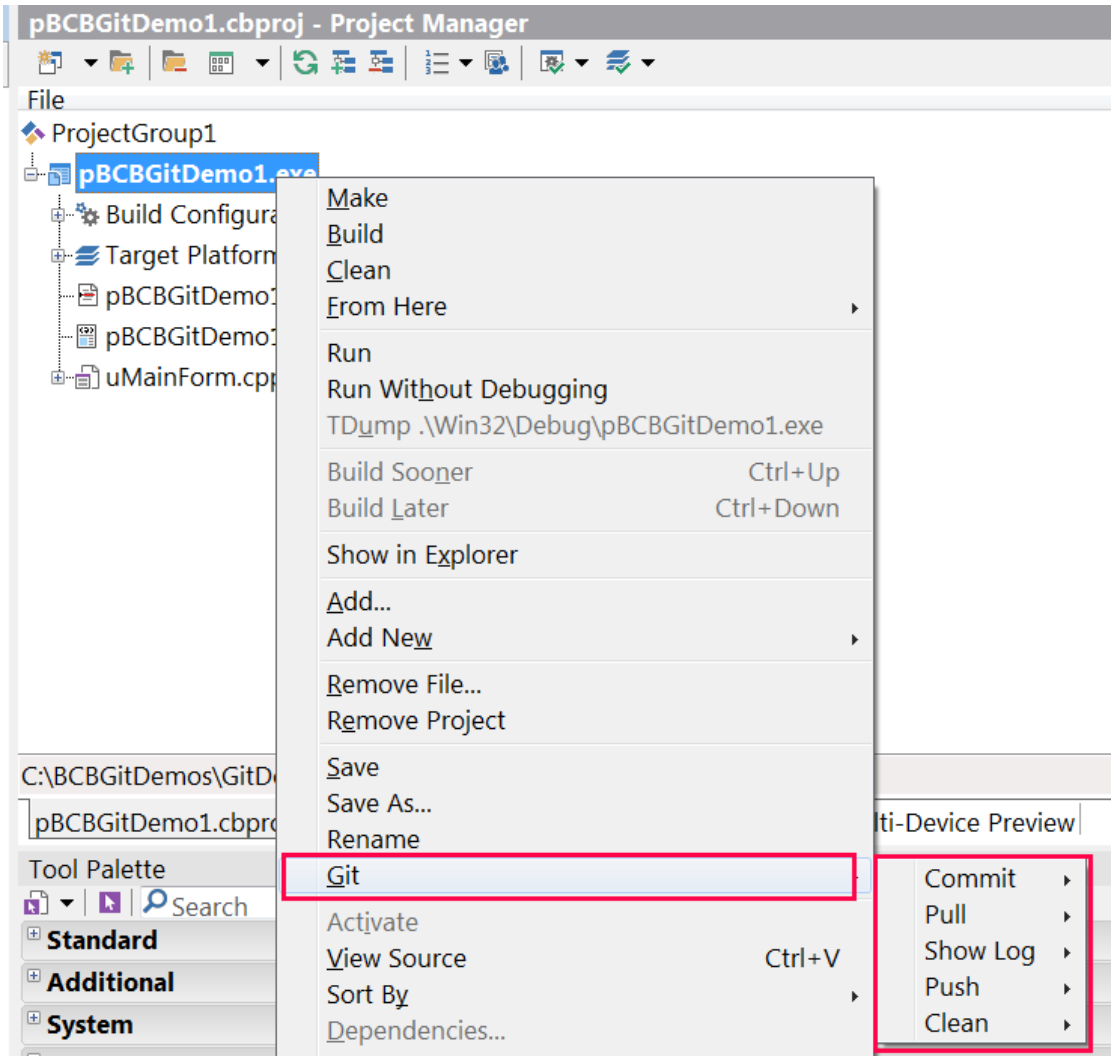
.git 子目录就是 Git 在 c:\BCBGitDemos\GitDemo1 目录中建立的本机链接库，我们如果到.git 子目录中就可以看到下面的内容，.git 子目录是由 Git 管理的本机链接库：



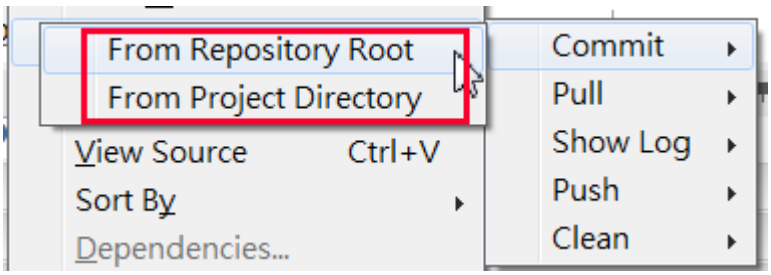
让我们建立一个 Multi-Device 项目并储存在 c:\BCBGitDemos\GitDemo1 目录，如下所示：



接着在项目管理员中点选鼠标右键可以看到有 Git 选项,在其中有数个可执行的 Git 命令：



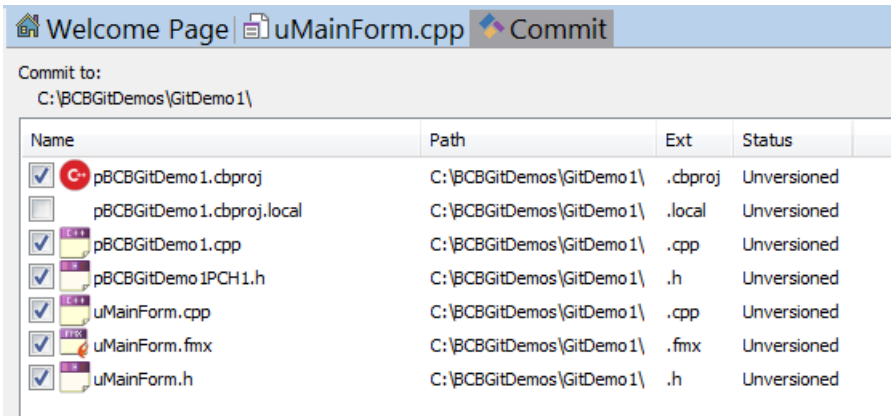
在 Git 选项中有 Commit 子选项，在其中有 2 个命令”From Repository Root”和”From Project Directory”：



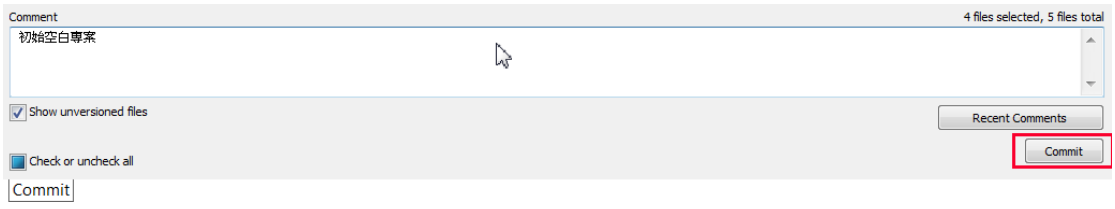
现在让我们把这个项目签入到本机的 Git 链接库中，由于现在本机中本机的 Git 链接库和 C++Builder 项目是在同一目录中，因此上图中的”From Repository Root”和”From Project Directory”2 个命令效果是一样的，所以请读者点选上图中的 Git|Commit|From Repository Root 选项，准备把项目签入到本机的 Git 链接库。

点选 Git|Commit|From Repository Root 选项后 IDE 会显示 Commit 页面让开发人员可以签入档案，请先 Commit 页面下方的 Show unversioned files 以显示尚未签入的项目文件：

此时 Commit 页面便会显示项目中所有的档案，请勾选要签入的档案，例如下图显示要签入 6 个档案，目前每一个档案的状态区位都是 Unversioned：



接着在下方的 Comment 字段输入签入说明，此时右下方的”Commit”按钮会致能，输入完毕之后请点选”Commit”按钮执行 Stage 档案和签入档案到本机的 Git 链接库中。

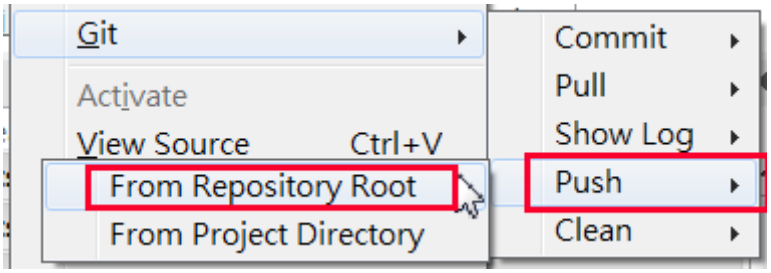


点选”Commit”按钮就是前图” Git 基本运作原理”中的第 3 和第 4 步骤。

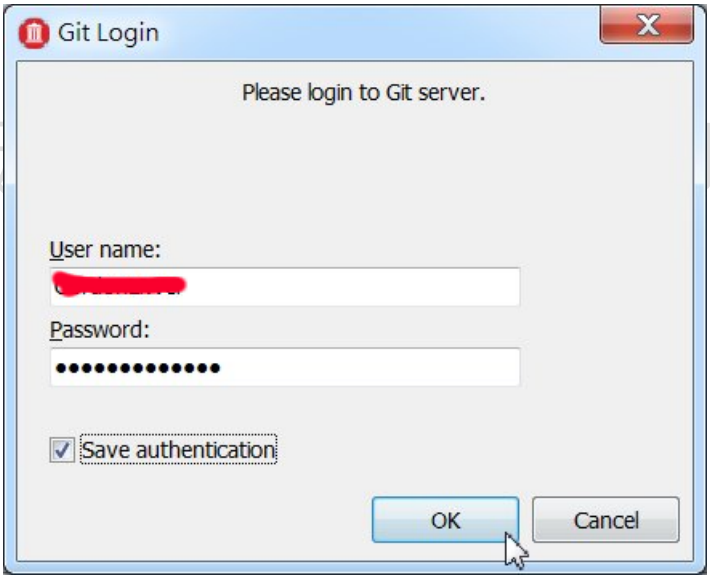
点选”Commit”按钮之后 IDE 会执行 Git 的 Commit 命令，成功之后 IDE 的讯息窗口就会显示一签入代号如下所示代表签入成功：

请注意现在我们只是把项目文件签入到本机链接库中，并没有把项目推播到远程 Git 服务器中，因此其他团体成员无法共享现在的开发成果，因此如果现在去 GitHub 的 BCBGitDemos 链接库我们看不到刚才签入的项目文件。

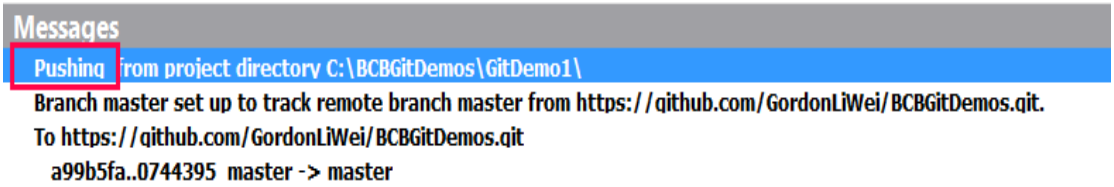
因此现在让我们把此项目推播到远程 Git 服务器中，以便进行团队开发。请点选 Git | Push | From Repository Root 选项：



由于现在要推播项目到远程 GitHub 的链接库，因此 IDE 会显示如下的对话盒要求输入您在 GitHub 的帐户信息：

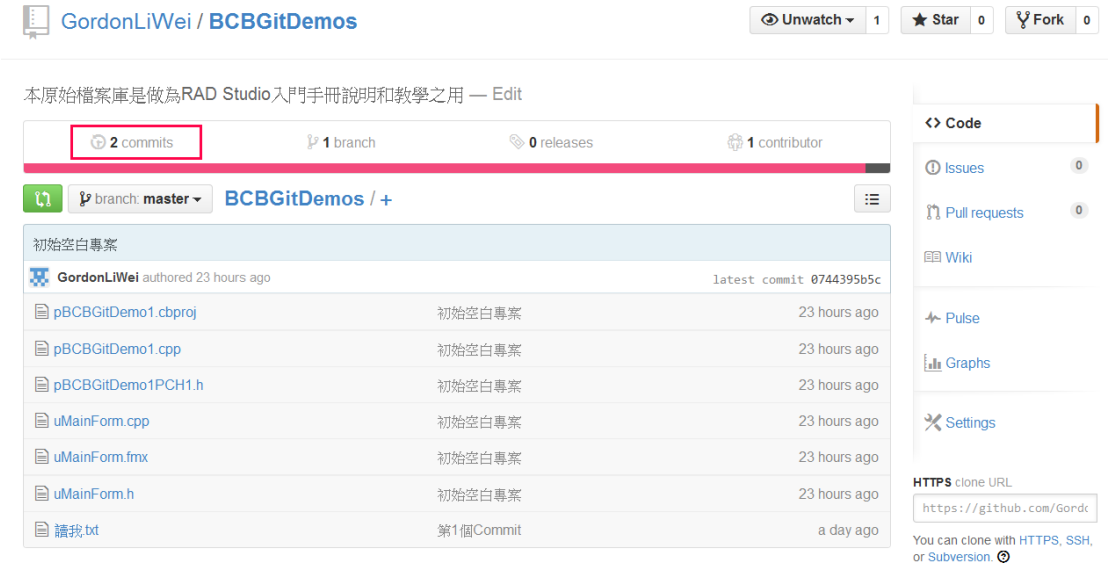


点选 OK 按钮之后 IDE 便会执行 Git 的 Push 命令，成功之后 IDE 在讯息窗口会显示推播成功的信息，从下图可以看到 IDE 的确是执行 Push 命令，而且会显示从本机的 master 分支推播到 GitHub 的链接库中 BCBGitDemos 链接库的 master 分支。

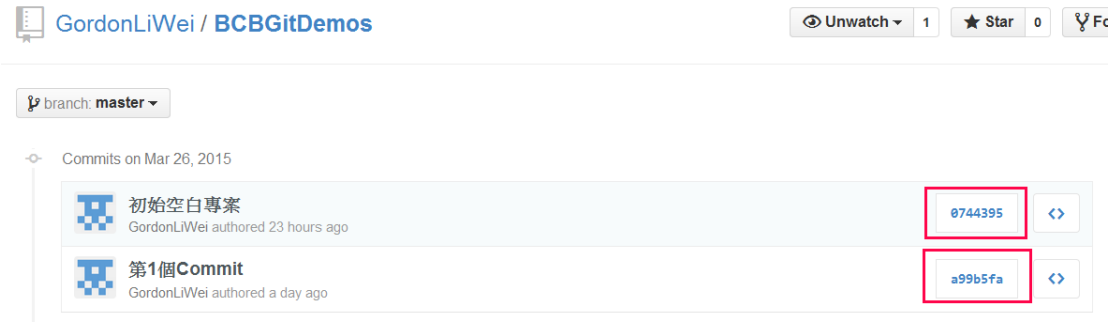


这个动作就是前图”Git 基本运作原理”中的第 5 步骤。

现在回到 GitHub，重新整理 BCBGitDemos 链接库页面我们就可以看到类似下面的结果，项目中的档案果然签入到远程的 Git 服务器了：

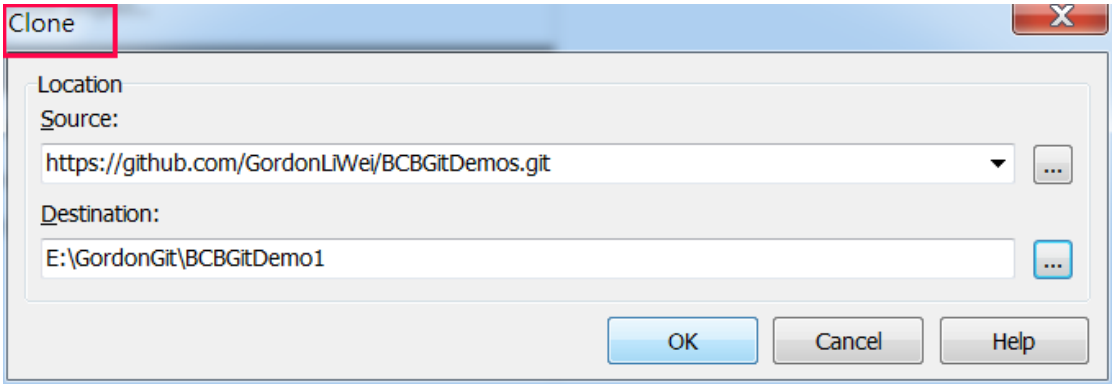


请注意上图中显示我们有 2 次签入的动作，第 1 次是签入”读我.txt”档案，第 2 次当然就是刚才签入项目的动作。如果我们点选上图中的 2 commits，那么就可以看到如下的结果：



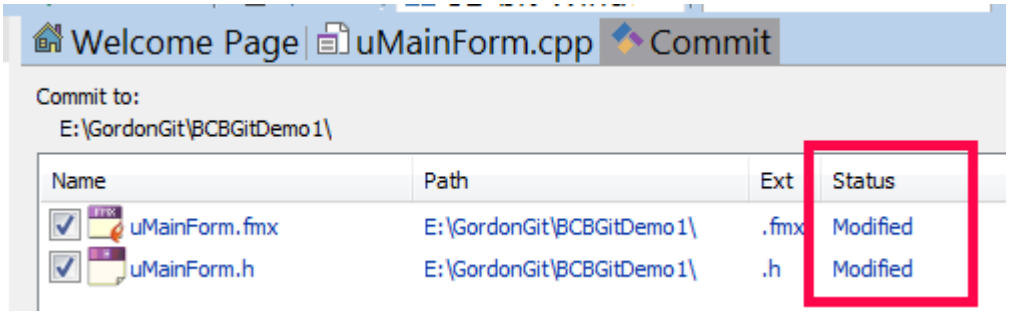
我们可以看到 2 次签入的批注和每次的签入代号。

现在假设有一个团队成员 Gordon 在另一台机器中想使用刚才推播到 GitHub 服务器中的项目，那么 Gordon 也可以在 IDE 中点选 File | Open From Version Control...选项从远程 GitHub 服务器中复制项目到 Gordon 的本机中，就像前面说明的流程一样，

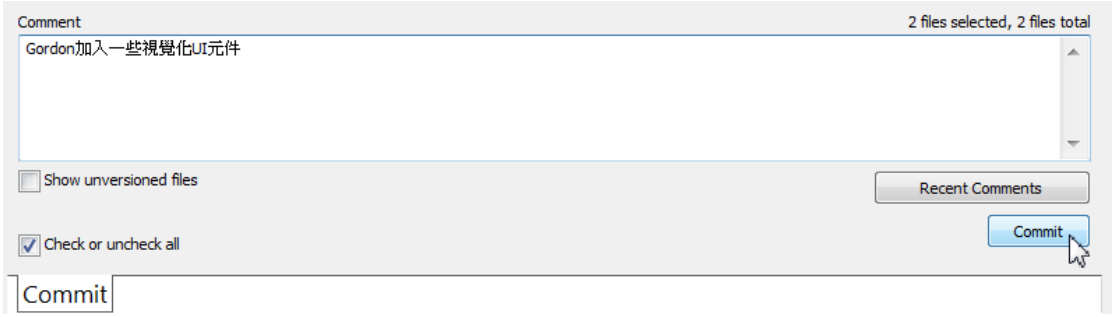


然后 Gordon 在复制到本机的项目中加入如下的 UI 组件：

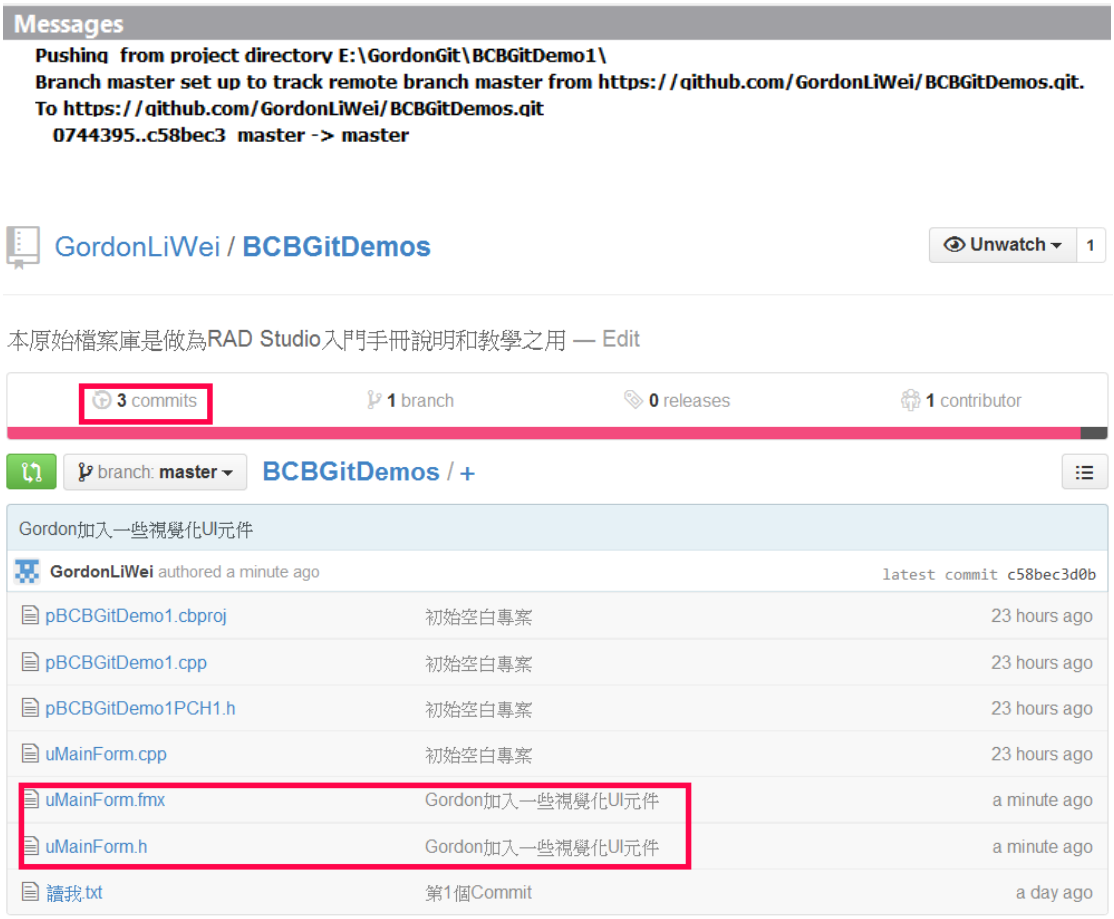
接着 Gordon 点选 Git|Commit|"From Repository Root"选项就可以在 IDE 中看到下面的 2 个项目文件被修改过了：



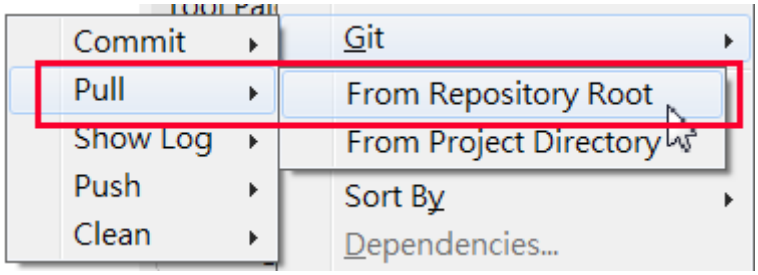
Gordon 输入新的签入说明再点选 Commit 按钮签入修改的档案：



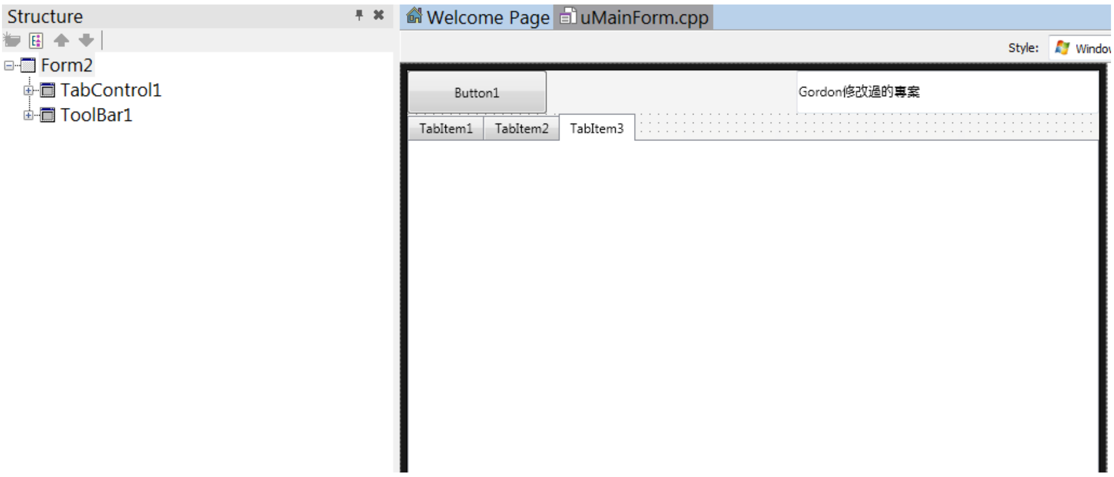
再点选 Git|Push|From Repository Root 选项把修改的项目推播回远程 Git 服务器，之后我们就可以看到类似下面的结果，Gordon 修改的项目果然签回到远程的 Git 服务器了：

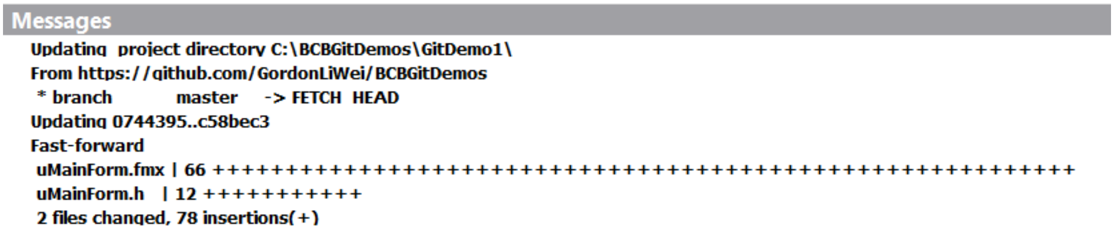


现在我们再回到原先的机器中，我们想继续 Gordon 的修改工作，因此请在 IDE 中点选 Git|Pull|From Repository Root 把项目的异动更新回原来的本机中：



数秒后我们就可以在原先的 IDE 中看到 Gordon 修改的项目已经整合到原先机器中的项目了，而我们就可以接着 Gordon 的开发工作继续往下开发，这就是分布式团队开发的范例：





请注意在上图中当我们从远程 Git 服务器中取得 Gordon 的异动时，IDE 会执行更新本机中原本项目的内容，合并我们的初始开发内容以及 Gordon 进行的开发异动。

整合的分布式版本控制工具 Git 可以让团队开发非常的方便且有效率，非常值得读者使用在日常的开发工作中。

11-3 使用 DUNITX 单元测试框架

C++Builder 从很早的版本即使用 DUNIT 框架来支持单元测试，到了开始支持 DUNITX 框架来支持单元测试，因此从 C++Builder 开发同时支持 DUNIT 和 DUNITX。开始使用 DUNITX 主要是因为：

1. DUNITX 支持跨平台的单元测试

2. DUNITX 使用 RTTI 的属性来定义单元测试类别而无需从特定的单元测试类别继承，因此任何的 C++Builder 类别都可以成为测试类别。如此一来 C++Builder 测试类别可免于受到单元测试框架的改变而需要修改

DUNITX 单元测试框架现在特别重要的原因就是因为现在 C++Builder 可开发多个平台，因此为了减少需要在多个平台测试/除错的成本，跨平台的单元测试能力就更重要了。

本小节将简单的说明如何使用 DUNITX 单元测试框架，希望读者在了解之后能够使用在每日的开发工作之中。

11-3-1 DUNITX 单元测试框架简介

DUNITX 是一开放原始码单元测试框架，DUNITX 主要是由 Vincent Parrett 先生开发的，随后有数字加入的贡献者。DUNITX 框架从 C++Builder 2010 版即开始支持，读者可以在下面的 URL 下载 DUNITX 框架：

```
https://github.com/VSoftTechnologies/DUnitX
```

到了 C++Builder 版正式加入到 C++Builder 的 IDE 中。

基本上 DUNITX 比 DUNIT 提供了更多的功能但使用上却更方便，下表简列了这 2 个单元测试框架的差异：

功能,	DUnit,	DUnitX
基础测试类别	TTestCase	无
测试方法	须宣告在 Published	宣告在 Published 或是使用[Test]属性标注
Fixture Setup 方法	无	使用[SetupFixture] 属性标注或是使用建构元(Constructor)
Test Setup 方法	覆载基础测试类别的 Setup 方法	使用 [Setup] 属性标注
Test TearDown 方法	覆载基础测试类别的 TearDown 方法	使用 [TearDown] 属性标注
命名空间	藉由注册的字符串参数	单元名称
数据驱动测试	无	使用 [TestCase(parameters)] 属性标注
Asserts	Check(X),	Assert 类别
Asserts on Containers(IEnumerable<T>),	人工撰写	Assert.Contains*, Assert.DoesNotContain*, Assert.IsEmpty*
使用正规表示法 Asserts	无	Assert.IsMatch (XE2 及以后的版本).
Stack Trace support	Jcl,	Jcl, madExcept 3,

		madExcept 4, Eurekalog 7
记忆漏失检查	FastMM4	FastMM4
IoC Container	使用 Spring 或其他框架	内建简易 IoC container
Console Logging	内建	内建
XML Logging	内建 (own format),	内建

为了让读者了解 DUNITX 测试框架使用原理，在下一小节中让我们一步一步的来说明如何撰写 DUNITX 测试框架测试类别以及上表的意义。

11-3-2 如何成为 DUNITX 测试框架的测试类别

从上面的表格说明我们可以了解在 DUNITX 测试框架中：

规则 1 任何的 C++Builder 类别都可以成为测试类别

因此下面的范例类别 TMyTestObject 就是一个 DUNITX 测试框架的测试类别：


```
#include <DUnitX.TestFramework.hpp>
#include <stdio.h>

#pragma option --xrtti

class __declspec(C++Builderrtti) TMyTestObject : public TObject
{
public:
  virtual void __fastcall SetUp();
  virtual void __fastcall TearDown();

__published:
  void __fastcall Test1();
  void __fastcall Test2();
};
```

非常的简单。

规则 2 撰写测试方法

有了测试类别之后我们需要有测试方法，在 C++Builder 的 DUNITX 中由于没有像 C++Builder 的语言属性来标志测试方法，因此 C++Builder 的测试方法都需要宣告在类别的 **__published** 部分，因此下面的 TestPushedMessageCount 和 TestUnPushedMessageCount 方法就成为了测试方法。

```
class __declspec(C++Builderrtti) TMyTestObject : public TObject
{
public:
  virtual void __fastcall SetUp();
  virtual void __fastcall TearDown();

__published:
  void __fastcall TestPushedMessageCount();
  void __fastcall TestUnPushedMessageCount();
};
```

也非常的简单。

在 C++Buidler 中 DUNITX 测试类别必须加入 2 个表头档案：

```
#include <DUnitX.TestFramework.hpp>
#include <stdio.h>
```

此外也必须使用#pragma 开启 rtti 功能：

```
#pragma option -xrtti
```

规则 3 如何使用 Test Fixture

在许多测试中我们经常需要在测试之前先进行一些设计工作，例如开启档案，建立数据库联机，建立要被测试的对象等。另外在测试完成之后则需要释放所有的资源。因此在一般测试框架中要执行测试设定的工作都是撰写在 Setup 方法中，测试完成之后释放所有资源的工作都是撰写在 TearDown 方法中。

在 DUNITX 中只要使用[Setup] 属性标注的方法就是 Setup 方法，DUNITX 在执行测试之前都会先执行使用[Setup] 属性标注的方法。而使用[TearDown] 属性标注方法就是 TearDown 方法，DUNITX 在执行完测试方法之后最后会执行使用 [TearDown] 属性标注的方法。但由于 C/C++语言没有属

性标注机制，因此在 C++Builder 中是使用 SetUp()方法和 TearDown()方法来代替：

```
 class __declspec(C++Builderrtti) TMyTestObject : public TObject
{
public:
  virtual void __fastcall SetUp();
  virtual void __fastcall TearDown();

__published:
  void __fastcall TestPushedMessageCount();
  void __fastcall TestUnPushedMessageCount();
};
```

规则 4 使用 Dunitx::Testframework::Assert 类别测试执行结果

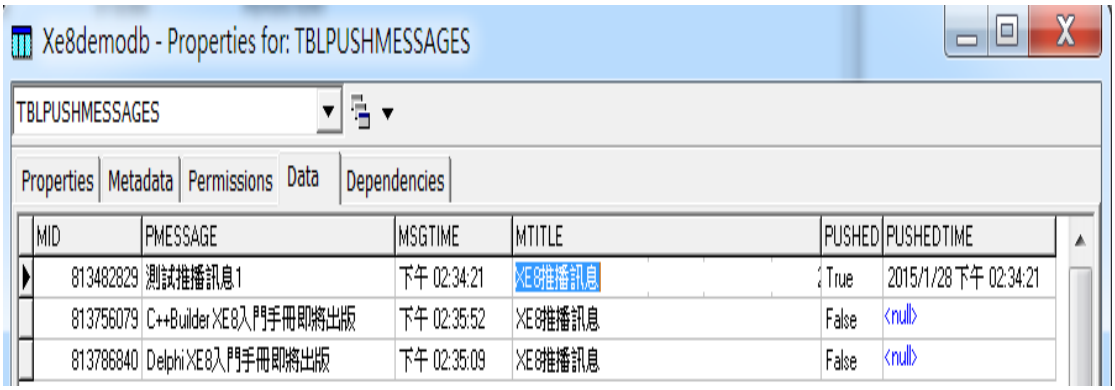
使用 DUNITX 框架最主要的目地当然就是测试开发人员撰写的程序代码是否正确，每当我们实作了一个方法之后就应该撰写一个测试方法来测试实作方法的正确性，实作第 2 个方法之后再撰写第 2 个测试方法来测试实作方法的正确性并且不断的一直执行所有先前的测试方法看看后来加入的实作方法有没有影响先前的实作程序代码(依照敏捷开发方式您应该反过来，先撰写测试方法再实作方法)。

在 DUNITX 框架的 Assert 类别中提供了许多的方法让程序员可以进行各种不同的测试，下面的表格列出了这些经常使用的测试方法：

功能	说明
Pass	检查函式是否可正常执行
Fail	检查函式是否执行失败
AreEqual	检查项目是否相等
AreNotEqual	检查项目是否不相等
AreSame	检查 2 个项目是否相同
AreNotSame	检查 2 个项目是否不相同
Contains	检查项目是否包含在一个串行对象中
DoesNotContain	检查项目是否不包含在一个串行对象中
IsTrue	检查条件是否为真
IsFalse	检查条件是否为假
IsEmpty	检查项目的数值是否是空白

IsNotEmpty	检查项目的数值是否不是空白
IsNull	检查项目是否是 null
IsNotNull	检查项目是否不是 null
WillRaise	检查一个匪法是否会产生例外
StartsWith	检查字符串是否以特定的子字符串开头
InheritsFrom	检查是否继承自特定的类别
IsMatch	检查项目是否符合特定的样式

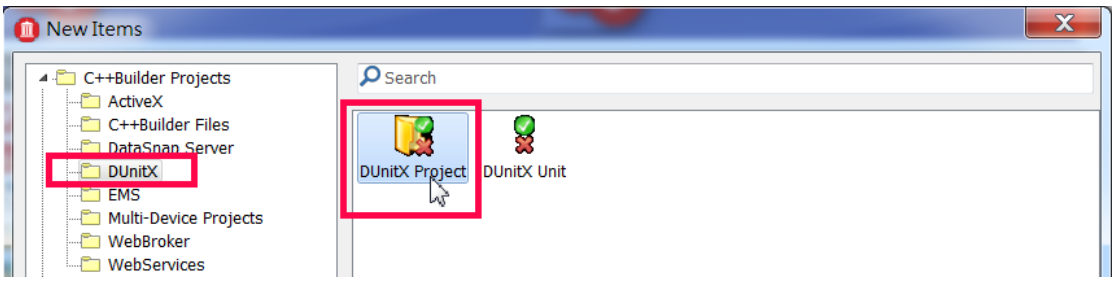
让我们使用一个数据表 TBLPUSHMESSAGES 来说明，一开始 TBLPUSHMESSAGES 中包含了下面的数据：



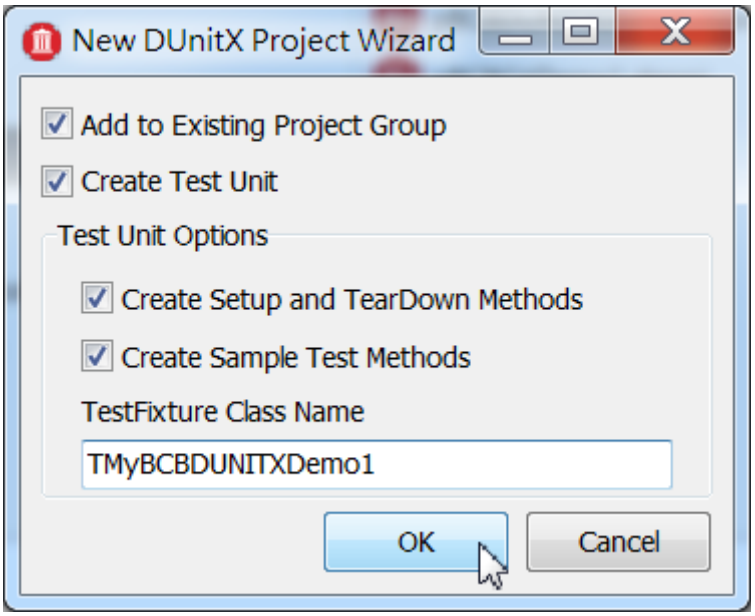
TBLPUSHMESSAGES 一共有 3 笔资料，其中已经推播的资料笔数是 1，另 2 笔数据是尚未推播的(PUSHED 字段)。接下来让我们直接使用 C++Builder IDE 中的 DUNITX 功能来说明如何进行测试。

11-3-3 使用 DUNITX 单元测试框架

在 C++Builder IDE 中使用 DUNITX 框架非常简单，只需点选 File|New|Other...选项再于 C++Builder Projects|DUnitX 项目中可看到 DUnitX Project 和 DUnitX Unit 图像。DUnitX Project 图像代表要建立 DUNITX 测试项目而 DUnitX Unit 图像代表要建立测试程序单元，现在让我们点选 DUnitX Project 图像建立 DUNITX 测试项目：



接着 IDE 会显示下面的对话盒询问您是否要建立测试程序单元，是否要建立 Setup 和 TearDown 方法，是否要建立简单的范例测试方法以及为测试类别取一个名称等信息：



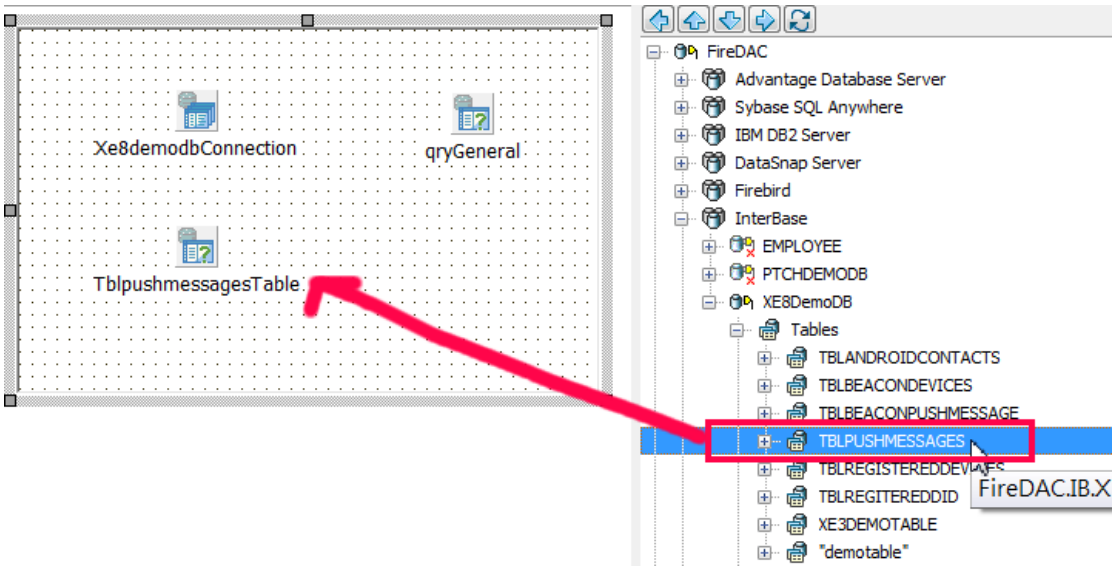
点选上面的选项和 OK 按钮后 IDE 便会产生如下的程序代码，除了前面介绍的 SetUp()和 TearDown()方法之外 DUnitX 精灵也会产生 2 个样本测试方法：


```
#pragma option --xrtti
class __declspec(C++Builderrtti) TMyBCBDUNITXDemo1 : public TObject
{
public:
  virtual void __fastcall SetUp();
  virtual void __fastcall TearDown();

__published:
  void __fastcall Test1();
  void __fastcall Test2();
};


void __fastcall TMyBCBDUNITXDemo1::SetUp()
{
}

void __fastcall TMyBCBDUNITXDemo1::TearDown()
```

```
{
}

void __fastcall TMyBCBDUNITXDemo1::Test1()
{
  // TODO
  String s("Hello");
  Dunitx::Testframework::Assert::IsTrue(s == "Hello");
}

void __fastcall TMyBCBDUNITXDemo1::Test2()
{
  // TODO
  String s("Hello");
  Dunitx::Testframework::Assert::IsTrue(s == "Bonjour"); // This
fails for illustrative purposes
}

static void registerTests()
{
  TDUnitX::RegisterTestFixture(__classid(TMyBCBDUNITXDemo1));
}
#pragma startup registerTests 33
```


我们现在想实作一个数据模块让它可以处理 TBLPUSHMESSAGES 数据表中的数据，也由于我们会撰写实作程序代码，因此我们要使用 DUNITX 框架来测试这些实作程序代码。

首先请在范例项目中加入一个数据模块，再于 IDE 的 Data Explorer 中把 TBLPUSHMESSAGES 数据表拖入到数据模块 ，再加入一个名为 qryGenral 的 TFDQuery 组件：



现在读者就可以直接在产生的测试程序单元中撰写测试程序代码了。

11-4 App Tethering

App Tethering 技术允许开发人员使用软件链接 PC 和各种客户端的设备，开发人员可藉由 WiFi 或是蓝牙连结各种不同的硬件：

App Tethering 技术在发展之初是希望帮助传统 PC 开发人员能够把 Windows 应用程序的功能移植到手机中，后来才逐渐发展成可连结各种不同的硬件的软件技术。

藉由 App Tethering 技术在不同硬件平台中的软件可以：

1. 互相远程控制执行行动命令

2. 传递和共享数据

3. 发展点对点的执行控制模式

App Tethering 的功能在 XE6 中即开始出现，一开始 App Tethering 只提供在同一个子网络区域中设备的链接，到了 XE7 App Tethering 可藉由直接提供 IP 地址链接也允许使用蓝牙协议连结，到了 App Tethering 提供了更完整的链接功能并且增进了 App Tethering 的执行效率。

由于 App Tethering 允许不同硬件平台中的软件不同硬件平台中的软件，因此用户可以把手机 App 的数据及时传递给 PC 中的应用程序，或是反之。因此 App Tethering 提供了 2 种方式传递数据：

传递数据方式	说明
共享资源	使用资源方式共享数据，并提供数据更新的能力
以暂时资源一次传送数据	从一平台 App 一次传递数据到另一平台的 App

由于 C++Builder 本身提供了数个 App Tethering 的范例供开发人员参考，但如果没有一些基本的概念的话可能不太容易了解，因此在本小节中我们将使用一个范例来说明如何使用 App Tethering 技术，这个范例就是：BDShoppingList。

基本上 App Tethering 使用了类似蓝牙的概念，也就是说要使用 App Tethering 功能，开发人员需要使用一个主要功能的组件，再使用另一个 Profile 组件来完成应用程序要提供的功能。因此这个主要功能组件就是 TetheringManager，而 Profile 组件就是 TetheringAppProfile。下面的表格说明了这 2 个组件的功能：

组件	说明
TetheringManager	负责提供最基本的功能，即侦测其他的 TetheringManager 并连结和配对远程的 TetheringManager 组件
TetheringAppProfile	提供 Tethering 的执行功能，包含执行远程命令，分享资源和传送数据等

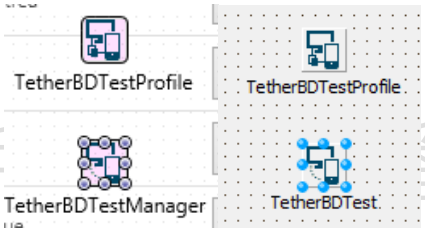
现在我们就可以开始说明 BDShoppingList 是如何工作的。请在 C++Builder IDE 中开启下面目录中的

```
c:\Users\Public\Documents\Embarcadero\Studio\16.0\Samples\CPP\RTL\Tethering\BDShoppingList\
```

BDShoppingList.groupproj 专案群组。这个项目群组中包含了一个 Windows 端的 VCL 应用程序和一个能在 Android/iOS 平台中执行的 App。在下面的小节中将说明 Windows 端和 Android/iOS 平台中的 App 如何藉由 App Tethering 技术共同执行运作。

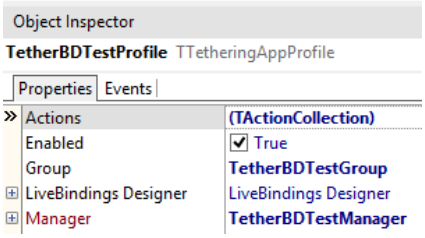
11-4-1 手机端 TetherDBClient 如何工作

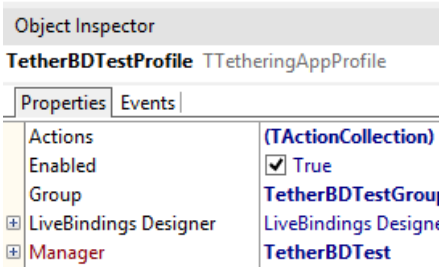
开启 TetherDBClient 项目的主窗体，您会看到下面左方的 TetheringManager 和 TetheringAppProfile 组件，如何此时再开启 Windows 端的 VCL 应用程序的主窗体就会看到到下方右边同样的 2 个组件：



使用 App Tethering 技术的 2 方应用程序各自需要这 2 个组件以侦测和链接对方。

TetheringManager 组件需要注册并使用 TetheringAppProfile 组件以启动功能，因此在对象查看器中可以看到上面的 2 个 TetheringAppProfile 组件都设定了它的 Manager 都链接到 TetheringManager 组件：





设定好上面的组件之后要让 App Tethering 能够工作，那么这 2 方一定要有一方必须执行侦测和连结对方的工作，这个工作在此范例中是由手机端执行的。

侦测和连结

要侦测和连结对方，开发人员可以呼叫 TetheringManager 组件的 AutoConnect 或是 DiscoverManagers 方法：

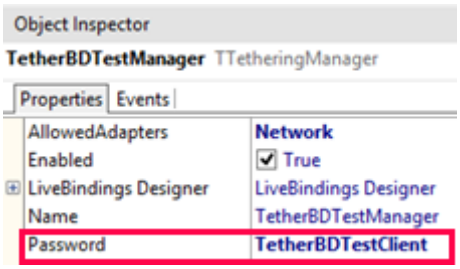
```
  void __fastcall AutoConnect(unsigned Timeout, const
System::UnicodeString ATarget = System::UnicodeString())/*
overload */;
  void __fastcall AutoConnect(const System::UnicodeString ATarget
= System::UnicodeString())/* overload */;
#ifndef _WIN64
  void __fastcall AutoConnect(unsigned Timeout, const
System::DynamicArray<System::UnicodeString> ATargetList)/*
overload */;
  void __fastcall AutoConnect(const
System::DynamicArray<System::UnicodeString> ATargetList)/*
overload */;
#else /* _WIN64 */
  void __fastcall AutoConnect(unsigned Timeout, const
System::TArray__1<System::UnicodeString> ATargetList)/* overload
*/;
  void __fastcall AutoConnect(const
System::TArray__1<System::UnicodeString> ATargetList)/* overload
*/;
#endif /* _WIN64 */

  void __fastcall DiscoverManagers(unsigned Timeout, const
System::UnicodeString ATarget = System::UnicodeString())/*
```

```
overload */;
  void __fastcall DiscoverManagers(const System::UnicodeString
ATarget = System::UnicodeString())/* overload */;
#ifndef _WIN64
  void __fastcall DiscoverManagers(unsigned Timeout, const
System::DynamicArray<System::UnicodeString> ATargetList)/*
overload */;
  void __fastcall DiscoverManagers(const
System::DynamicArray<System::UnicodeString> ATargetList)/*
overload */;
#else /* _WIN64 */
  void __fastcall DiscoverManagers(unsigned Timeout, const
System::TArray__1<System::UnicodeString> ATargetList)/* overload
*/;
  void __fastcall DiscoverManagers(const
System::TArray__1<System::UnicodeString> ATargetList)/* overload
*/;
#endif /* _WIN64 */
```


这 2 个方法都可以接受一个超时时间参数，此内定的参数值是 1500ms，在这 2 个方法于超时时间参数值到达之后就会触发 OnEndAutoConnect 或 OnEndManagersDiscovery 事件，在这 2 个事件处理函式中您会收到一列可十 2 结的远程 TetheringManager 组件，您可以选择其中您的 App 的链接对象。因此在 TetherDBClient 项目的中您会看到此范例使 AutoConnect 方法侦测并连结对方共在 OnEndAutoConnect 事件中检查侦测到的远程 TetheringManager 组件：

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  TetherBDTestManager->AutoConnect();
}

void __fastcall
TForm1::TetherBDTestManagerRemoteManagerShutdown(const TObject
*Sender,  const UnicodeString ManagerIdentifier)
{
  CheckRemoteProfiles();
}
```

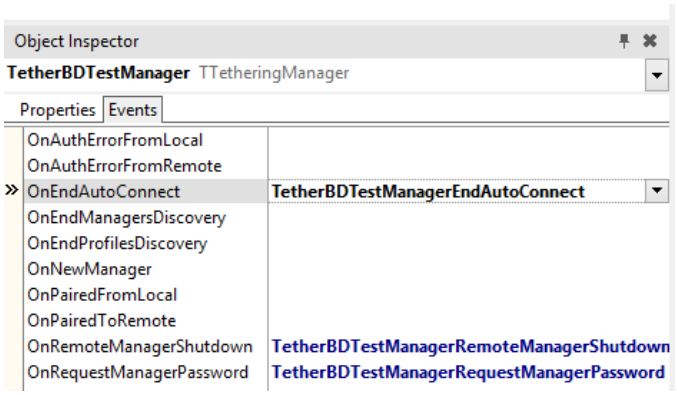
您也可以使用密码来保护合法的链接：



当使用密码时在链接远程 TetheringManager 组件时远程 TetheringManager 组件会触发 OnRequestManagerPassword 事件要求提供登入密码，因此 TetherDBClient 项目在它的 OnRequestManagerPassword 事件中提供如下的登录链接密码：

```
void __fastcall
TForm1::TetherBDTestManagerRequestManagerPassword(const TObject
*Sender,
const UnicodeString RemoteIdentifier, UnicodeString &Password)
{
  Password = "TetherBDTest";
}
```


而如果使用的密码错误就会触发 OnAuthErrorFromRemote 事件。



如果成功登录并连结，那么接着会触发 OnPairedToRemote 事件最后再触发 OnEndProfilesDiscovery 或 OnEndAutoConnect 事件。

而在远程一方，例如下面的 Windows 端 TetherDatabase 项目中，如果 TetherDBClient 使用错误的密码链接就会触发 OnAuthErrorFromLocal 事件，但如果密码正确就会触发 OnPairedFromLocal 事件。

了解了 TetherDBClient 端如何工作之后再让我们说明 Windows 端。

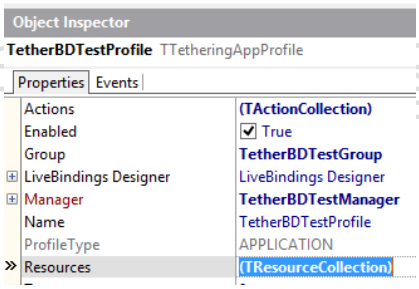
11-4-2 Windows 端 TetherDatabase 如何工作

Windows 端 TetherDatabase 项目提供了手机端 TetherDBClient App 一个远程命令让手机端可执行在远程 Windows 的 TetherDatabase 应用程序中的功能，TetherDatabase 项目也提供了手机端 TetherDBClient App 一个共享的资源让手机端 TetherDBClient App 可撷取远程 Windows 的 TetherDatabase 应用程序的执行结果。因此 Windows 端 TetherDatabase 项目提供了：

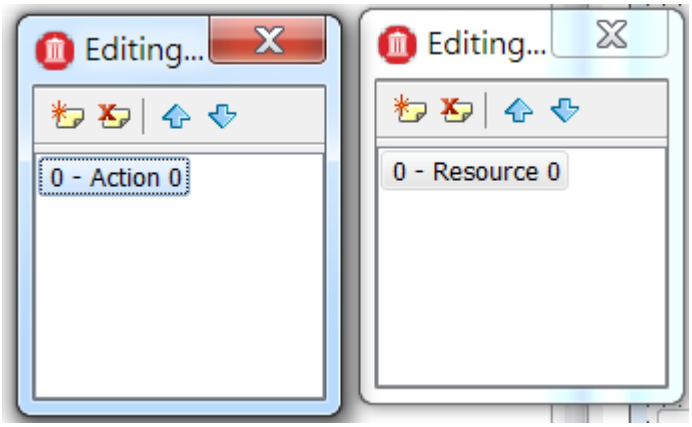
1. 一个输出执行命令

2. 一个共享资源

要提供输出执行命令和共享资源，开发人员可以藉由 TetheringAppProfile 组件的 Actions 和 Resources 特性，例如在 TetherDatabase 项目中的 TetherBDTestProfile 组件：

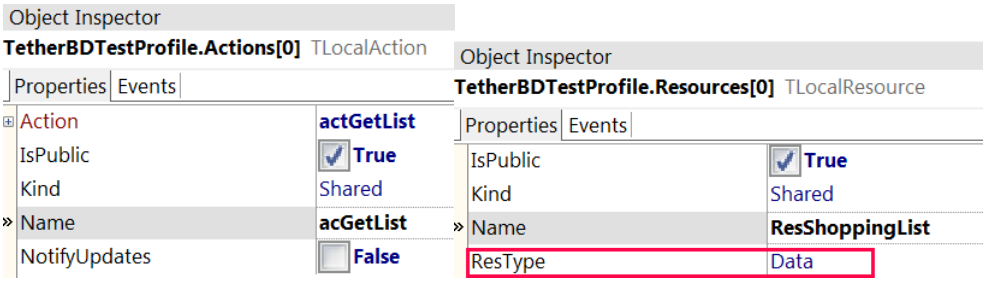


在它的 Actions 和 Resources 特性中提供了一个命令对象 TAction 和一个共享资源对象 TLocalResource：

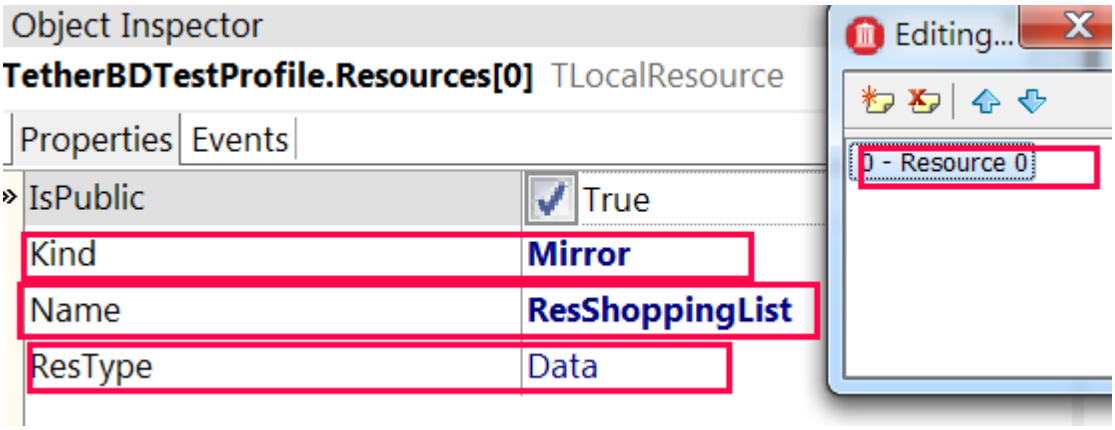


命令对象在被远程手机 App 呼叫执行之后就会把执行结果以共享资源方式让双方可存取，因此共享资源对象 TLocalResource 的 ResType 的型态是设定

为 Data， Kind 特性设定为”Shared”，而且共享资源是命名为”ResShoppingList”：

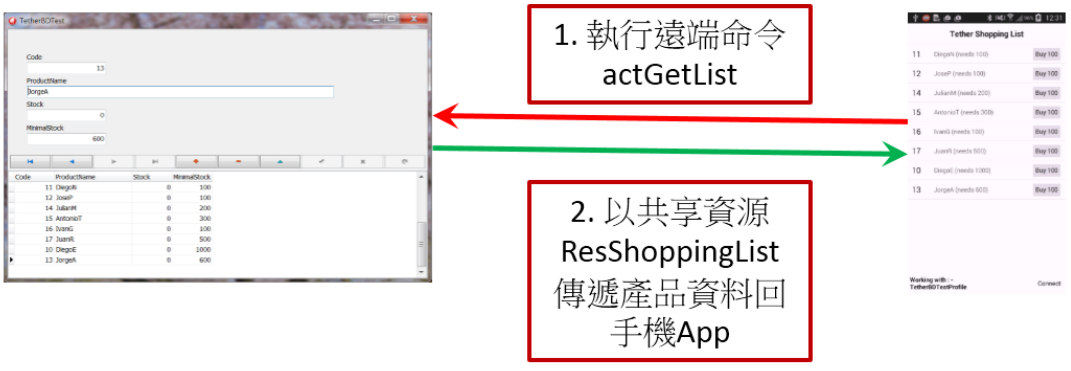


因此在手机端 TetherDBClient 项目中的 TetherBDTestProfile 组件其 Resources 特性中也要加入一个命名为”ResShoppingList”的共享资源对象但其 Kind 特性设定为”Mirror”：



11-4-3 Windows 端和手机如何共同工作

现在就可以说明 Windows 端和手机如何使用 App Tethering 技术共同工作了，请先在 Windows 中执行 TetherDatabase 项目再于手机中执行 TetherDBClient 项目，点选 TetherDBClient 项目右下方的 Connect 按钮就可以看到如下的执行结果：



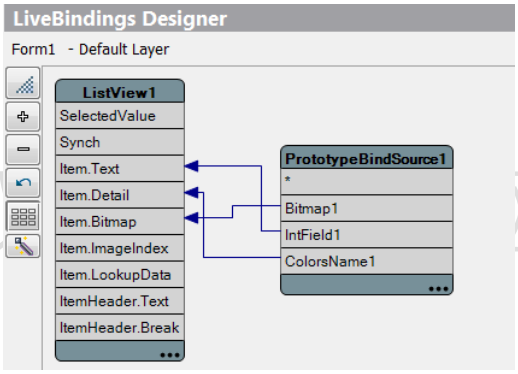
手机中的 App 可以藉由 App Tethering 技术看到需要购买那么产品，为什么？这是因为...

TetherDBClient 专案端

在手机 TetherDBClient 项目端的 Connect 按钮被点选后就会去执行远程的命令 actGetList：

```
  if (!FIsConnected)
    actGetList->Execute();
```

在远程(Windows 端)执行了此命令之后就会把执行结果储存在共享资源中，接着 App Tethering 就会触发 TetherDBClient 项目端的 OnResourceReceived 事件，TetherDBClient 项目端再使用 LiveBinding 技术把产品数据显示在手机中。



TetherDatabase 专案端

在 Windows 端的 TetherDatabase 项目的 actGetList 命令被手机端呼叫后就会执行并把执行结果存放在共享资源中等待手机端去存取执行结果：

```
void __fastcall
TForm1::TetherBDTestProfileResources0ResourceReceived(const
TObject *Sender,
      const TRemoteResource *AResource)
{
  TStringList * lStrings = NULL;
  ListView1->Items->Clear();
  if(AResource->Value.AsString != "NONE") {
```

```
      lStrings = new TStringList();
      try
      {
          lStrings->Delimiter = ':';
          lStrings->DelimitedText = AResource->Value.AsString;
          for(int i = 0; i < lStrings->Count; i++){
              TListViewItem * lItem = ListView1->Items->Add();
              TStringList * itemParts = new TStringList();
              try
              {
                  itemParts->Delimiter = '-';
                  itemParts->DelimitedText = lStrings->Strings[i];
                  lItem->Text = itemParts->Strings[1];
                  lItem->Detail = itemParts->Strings[0] + " (needs
" + itemParts->Strings[2] + ")";
              }
               __finally
              {
                  delete itemParts;
              }
          }
      }
       __finally
      {
          delete lStrings;
      }
  }
}
```


接着手机端可以决定要采购库存不足的产品，一但用户在手机端点选采购特定的产品后，就会执生下面的流程：

TetherDBClient 专案端

在手机 TetherDBClient 项目端的 OnListView1ButtonClick 事件中藉由呼叫 SendString 方法把要采购的产品同称传递给 Windows 端：

```
void __fastcall TForm1::ListView1ButtonClick(const TObject
*Sender, const TListViewItem *AItem,
        const TListItemSimpleControl *AObject)
{
  TetherBDTestProfile->SendString(TetherBDTestManager->RemotePro
files->Items[0],
      "Buy item", AItem->Text);
}
```


TetherDatabase 专案端

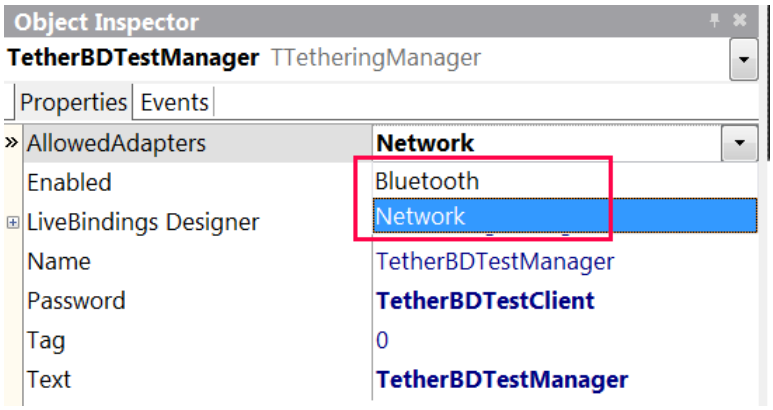
此时 Windows 端的 TetherDatabase 项目就会触发 OnResourceReceived 事件，接着就执行采购和更新数据的工作：

```
void __fastcall
TForm2::TetherBDTestProfileResources0ResourceReceived(const
TObject *Sender,
        const TRemoteResource *AResource)
{
  if(AResource->ResType == TRemoteResourceType::Data) {
      int pId = StrToInt(AResource->Value.AsString);
      CDSProducts->First();
```

```
      while(!CDSProducts->Eof) {
          if(CDSProductsCode->Value == pId) {
              CDSProducts->Edit();
              CDSProductsStock->Value =  CDSProductsStock->Value +
100;
              CDSProducts->Post();
              break;
          }
          CDSProducts->Next();
        }
    }
}
```

了解了 BDShoppingList 项目群组如何工作之后您就应该掌握了 App Tethering 技术的基本观念了，您可以再检视其他 App Tethering 的范例项目学习如何传递图形数据或是串行流的数据。

在 XE7 之后 App Tethering 不但可以使用 Wi-Fi 连结，也可以使用改用蓝牙连结，开发人员只需要设定 TTetheringManager 组件的"AllowedAdapters"特性即可，设定 AllowedAdapters 特性值为 Network 代表使用 Wi-Fi，设定 Bluetooth 代表使用蓝牙连结：



另外 XE7 之后也允许开发人员直接链接特定的 IP 地址，例如在呼叫 TTetheringManager 组件的 AutoConnect 方法时直接使用一个 IP 地址参数即可：

```
TetherBDTestManager->AutoConnect(2000,"192.168.0.27");
```

11-5 蓝牙开发

C++Builder 从 XE7 开始便支持开发传统蓝牙和低耗电蓝牙 BLE 的功能，并且提供了 TBlueToothLE 组件封装低耗电蓝牙 BLE 的功能，但对于传统蓝芽则只提供类别支持并没有提供封装传统蓝牙的组件。但到了 C++Builder 终于同时提供了 TBlueTooth 和 TBlueToothLE 这 2 个组件来封装传统蓝牙和低耗电蓝牙 BLE 功能。

要开发蓝牙应用程序您必须了解不同平台对于蓝牙技术的支持差异，下面的表格列出了 4 个平台对传统蓝牙和低耗电蓝牙 BLE 的支持程度：

	传统蓝牙	低耗电蓝牙BLE
Android	✔	✔(Android 4.3(含)以上版本)
iOS	✖	✔(iPhone 4s+ and iPad2+)
Windows	✔	Windows 8
Mac	✔	✔

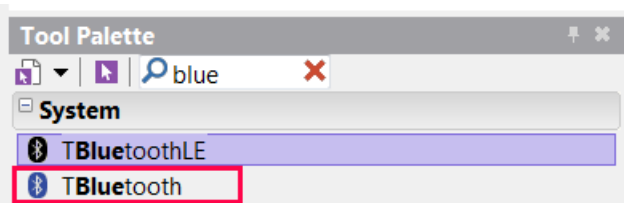
在本小节中我们将说明如何使用 TBlueTooth 组件来开发蓝牙 App。

使用 TBlueTooth 组件

开发蓝牙功能的 App 基本上开发人员要执行下列的步骤：

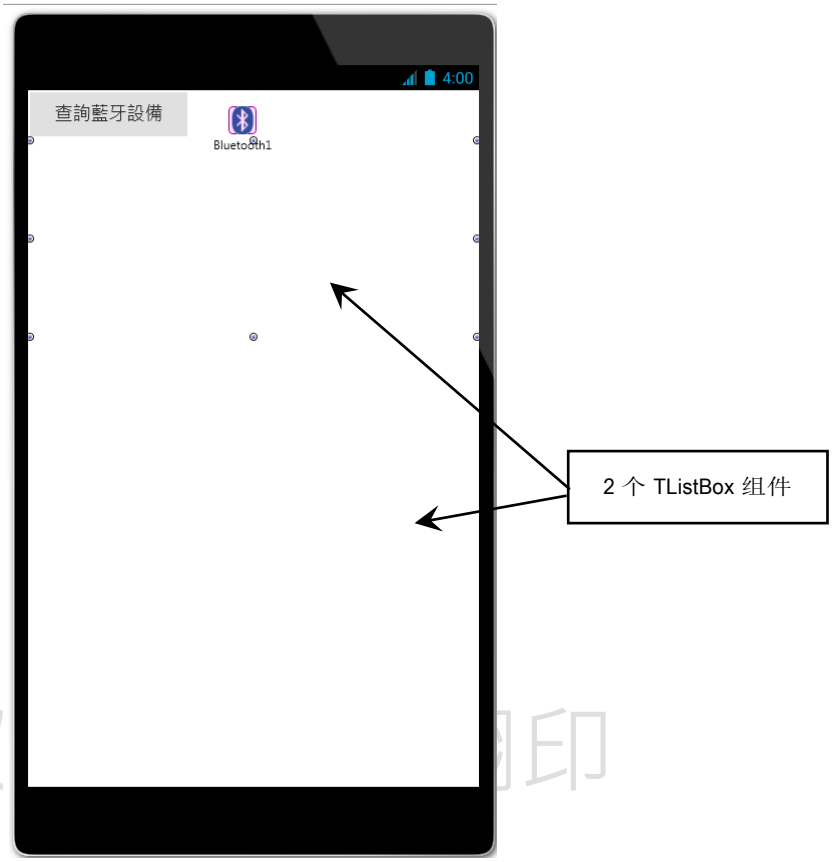
1. 搜寻附近的蓝牙设备
2. 搜寻蓝牙设备提供的服务
3. 配对蓝牙设备
4. 建立配对蓝牙设备之间的通讯管道
5. 传递数据

有了下面 TBlueTooth/TBlueToothLE 组件之后这些工作就非常简单了：



现在就让我开发一个简单的范例蓝牙 App 来说明如何完成上面的开发步骤。

首先建立一个 Multi-Device 项目并在主窗体中加入如下的组件：



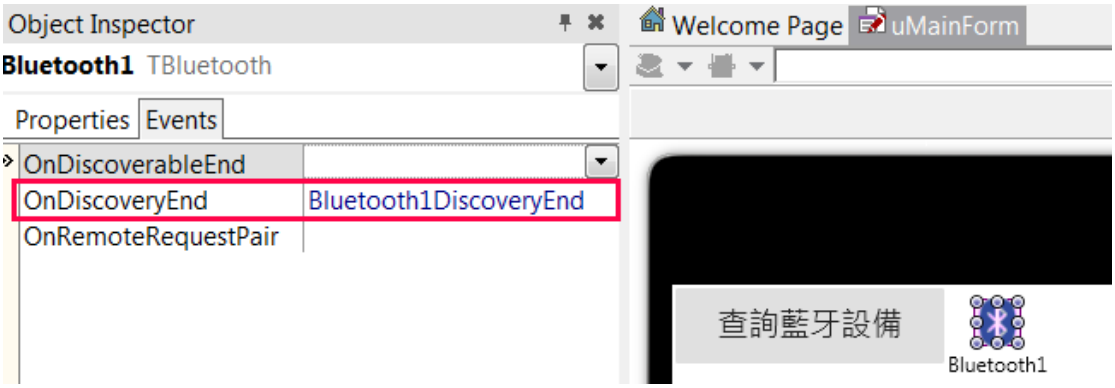
要查询蓝牙设备很简单，只要呼叫 TBlueTooth 组件的 DiscoverDevices 方法即可，DiscoverDevices 接收一个搜寻蓝牙设备时间的参数：

```
   void __fastcall DiscoverDevices(int ATimeout);
```

因此要实作主窗体 " 查询蓝牙设备 " 按钮的功能只需要使用下面的程序代码让范例 App 使用 10 秒的时间查询蓝牙设备：

```
void __fastcall TfmMainForm::Button1Click(TObject *Sender)
{
  Bluetooth1->DiscoverDevices(10000);
}
```

在 TBlueTooth 组件查询完毕之后它会触发 OnDiscoveryEnd 事件：

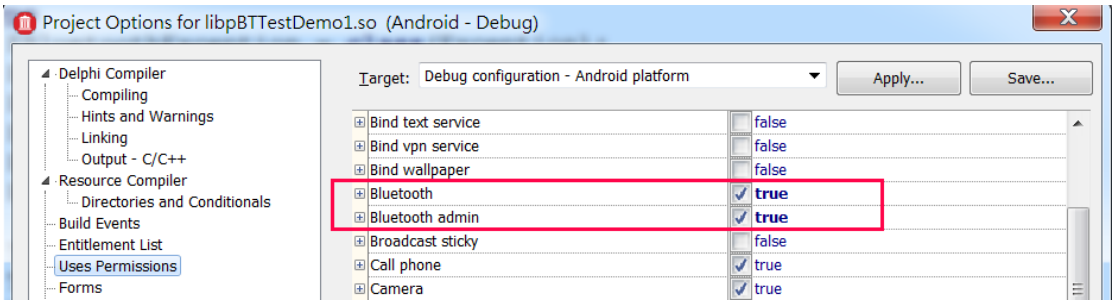


在 OnDiscoveryEnd 事件会传入所有找到的蓝牙设备参数 ADeviceList ，它的型态是 TBluetoothDeviceList。TBluetoothDeviceList 的 Items 特性是 TBluetoothDevice 类别对象，每一个 TBluetoothDevice 类别对象代表一个搜寻到的蓝牙设备。TBluetoothDevice 类别对象即可提供蓝牙设备的名称，地址等信息，而且我们也可以使用它来获得此蓝牙设备提供的服务信息。

因此在 OnDiscoveryEnd 事件中我们即可以取出每一个 TBluetoothDevice 类别对象并显示此它的名称到主窗体的 TListBox 中：

```
void __fastcall TfmMainForm::Bluetooth1DiscoveryEnd(TObject *
const Sender, TBluetoothDeviceList * const ADeviceList)
{
  lbBTDevices->Items->Clear();
  for (int iCount = 0; iCount < ADeviceList->Count; iCount++)
  {
  lbBTDevices->Items->Add(ADeviceList->Items[iCount]->DeviceName
+ ":" + ADeviceList->Items[iCount]->Address);
  }
  FDiscoverDevices = ADeviceList;
  TabControl1->TabIndex = 0;
}
```


现在可以准备执行范例 App 来看看它是否能搜寻蓝牙设备，但在编译和执行之前必须先开启范例 App 存取蓝牙的权限。请点选 Project|Options...选项并如下图勾选 Bluetooth 和 Bluetooth admin 权限：



下面是执行开发到现在的范例 App 画面，我们可以看到范例 App 是可以找到附近的蓝牙设备：

现在再让我们实作在找到蓝牙设备之后如果在 TListBox 中点选这个蓝牙设备就可以找到它提供的服务信息，这可以在 TListBox 的 OnItemClick()事件中先找到被点选的蓝牙设备，然后呼叫 DisplayDeviceServices()方法：

```
void __fastcall TfmMainForm::lbBTDevicesItemClick(TCustomListBox *
const Sender, TListBoxItem * const Item)
{
  DisplayDeviceServices(FDiscoverDevices->Items[Item->Index]);
}
```

DisplayDeviceServices()方法藉由呼叫 TBlueTooth 组件的 GetServices()方法并传入被点选的蓝牙设备做为参数：

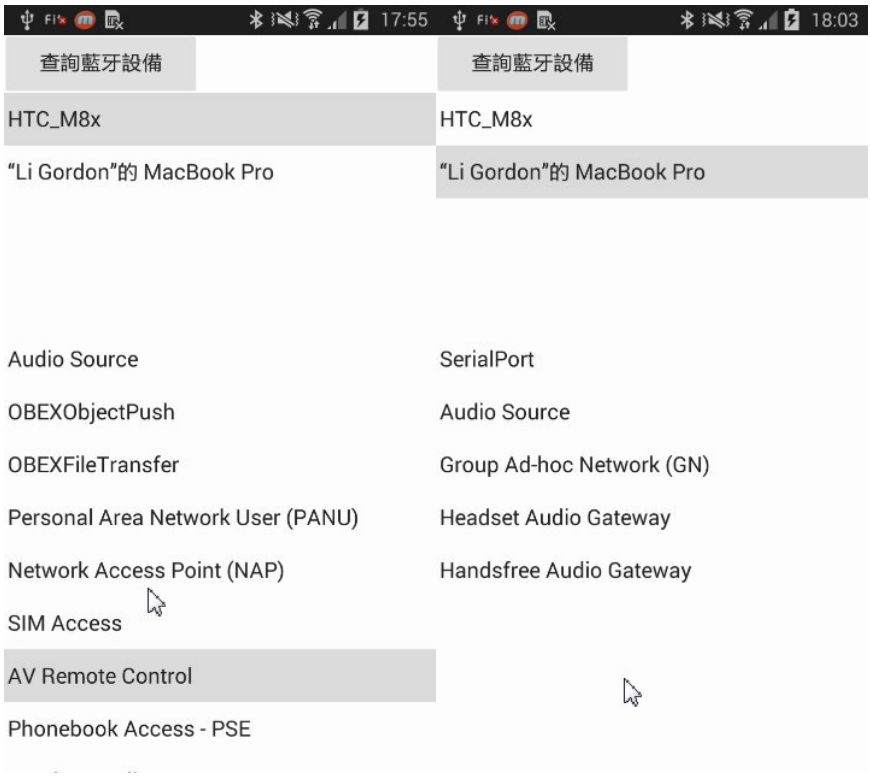
```
System::Bluetooth::TBluetoothServiceList* __fastcall
GetServices(System::Bluetooth::TBluetoothDevice* const ADevice);
```

GetServices()方法会回传蓝牙设备提供的所有服务，在回传的 TBluetoothServiceList 对象中每一个蓝牙设备提供的服务都以一个 TBluetoothService 对象代表，因此在 DisplayDeviceServices()方法中我们就

可以取出每一个 TBluetoothService 对象并显示此服务名称到主窗体的第 2 个 TListBox 中：

```
void TfmMainForm::DisplayDeviceServices(TBluetoothDevice*
aDevice)
{
  BTDeviceServices->Items->Clear();
  TBluetoothServiceList * sl = Bluetooth1->GetServices(aDevice);
  for (int iCount = 0; iCount < sl->Count; iCount++)
  {
  BTDeviceServices->Items->Add(sl->Items[iCount].Name);
  }
}
```

再执行范例 App 并点选找到的蓝牙设备就可以如下图看到每个蓝牙设备都提供了不同的服务：



接下来让我们实作第 2 个步骤＂配对蓝牙设备＂。

要实作配对蓝牙设备非常简单，只需要呼叫 TBlueTooth 组件的 Pair()方法即可，而 UnPair()方法则是解除配对：

```
function Pair(const ADevice: TBluetoothDevice): Boolean;
```

```
function UnPair(const ADevice: TBluetoothDevice): Boolean;
```

Pair()方法也是接受一个代表要配对的蓝牙设备的 TBluetoothDevice 对象。因此让我们在主窗体中加入一个配对按钮，当用户在主窗体的 TListBox 中选择了一个搜寻到的蓝牙设备后就可以点选配对按钮来进行蓝牙设备之间的配对工作。

下面是配对按钮的 OnClick 实作程序代码，它呼叫了 Pair()方法并且把点选的 TBluetoothDevice 对象传入做为参数。如果 Pair()方法执行成功就会回传 true 值，那么我们就把成功配对的蓝牙设备名称加入到成功配对的 TComboBox 组件中：

```
void __fastcall TfmMainForm::btnPairClick(TObject *Sender)
{
  System::Bluetooth::TBluetoothDevice* pDevice =
FDiscoverDevices->Items[lbBTDevices->ItemIndex];
  if (Bluetooth1->Pair(pDevice))
  {
  cbPairDevices->Items->Add(FDiscoverDevices->Items[lbBTDevices-
>ItemIndex]->DeviceName);
  cbPairDevices->ItemIndex =
cbPairDevices->Items->IndexOf(FDiscoverDevices->Items[lbBTDevice
s->ItemIndex]->DeviceName);
  }
}
```


再次执行范例程序代码，在下面的画面中可以看到我们要在范例 App 执行的 Samsung S4 中配对 HTC M8 手机：

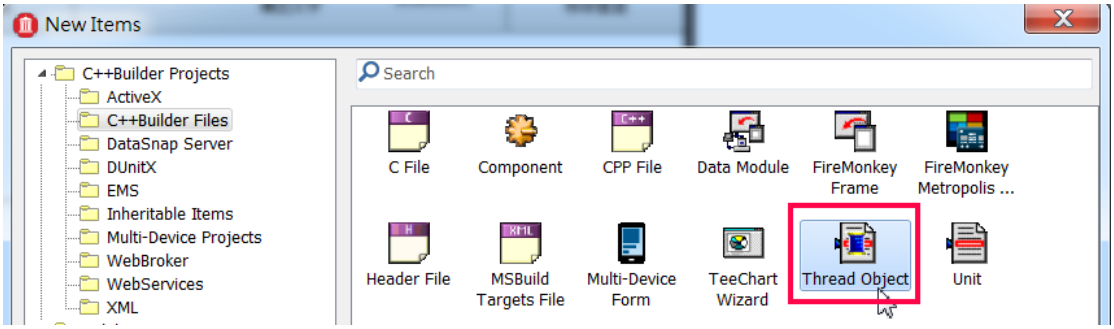
点选配对按钮之后可以看到 Samsung S4 显示的配对要求：

按下确定之后可以看到成功和 HTC M8 配对成功了：

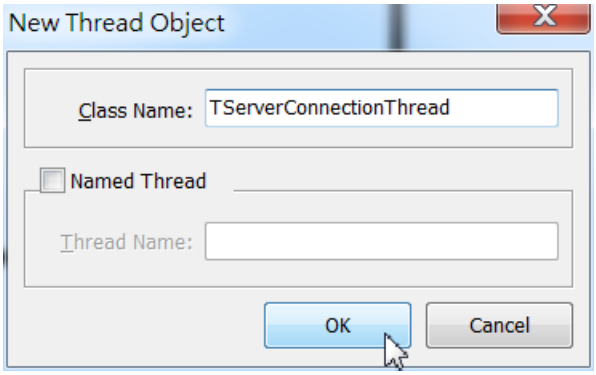
完成了配对步骤之后我们就可以开始下一个步骤＂建立配对蓝牙设备之间的通讯管道＂，一旦通讯管道建立成功之后就可以在蓝牙设备之间传递数据了。

要建立通讯管道并且传递数据我们需要使用一个独立的线程，如此一来才不会影响主线程的执行。先让我们在主窗体中加入 3 个按钮，1 可输入数据的 TEdit 组件和一个 TMemo 组件，如下所示：

接着在项目中建立一个 Thread Object：



取名为 TServerConnectionThread 并且在主窗体中使用它：



要在配对的蓝牙设备之间通讯，开发人员需要使用下面的步骤：

1. 一方的蓝牙设备必须以伺服端建立通讯管道

2. 另一方的蓝牙设备必须以客户端建立通讯管道

3. 伺服端的蓝牙设备先建立 TBluetoothServerSocket 对象，并使用它监听客户端的链接请求，一旦伺服端蓝牙设备接受链接便可取得 TBluetoothSocket 对象。伺服端取得 TBluetoothSocket 对象之后就可以使用它读取客户端传送来的数据

4. 客户端蓝牙设备建立 TBluetoothSocket 对象并使用它传递数据给伺服端的蓝牙设备

了解了如何建立通讯管道和传递数据的原理之后我们就可以开始实作了，首先让我们实作伺服端。

在伺服端中我们要使用前面建立的 TServerConnectionThread 对象中于一个独立的线程来监听客户端的连结请求并读取客户端传送来的数据。因此在主窗体的”建立通讯管道’按钮的 OnClick 事件处理函式中呼叫了 CreateBTTextService()方法来堤行这些工作：

```
void __fastcall TfmMainForm::btnCreateTextServiceClick(TObject
*Sender)
{
  CreateBTTextService();
}
```

CreateBTTextService()方法主要是建立 TServerConnectionThread 对象并藉由 TBluetoothServerSocket 对象取得可读取数据的 TBluetoothSocket 对象。要取得 TBluetoothServerSocket 对象我们可以藉由 TBlueTooth 组件的 CurrentAdapter 特性先得 TBluetoothAdapter 对象，再

呼叫 TBluetoothAdapter 对象的 CreateServerSocket() 方法取得 TBluetoothServerSocket 对象：

```
TBluetoothServerSocket* __fastcall CreateServerSocket(const
System::UnicodeString AName, const GUID &AUUID, bool Secure);
```

CreateServerSocket()方法接受 3 个参数，第 1 个是此通讯管道的名称，第 2 个参数是代表此通讯管道的 GUID 值，最后一个参数代表是否要建立一个安全的通讯管道。为了传递给 CreateServerSocket()方法第 1 和第 2 个参数，范例 App 定义了下面的常数(在 IDE 中 GUID 值可使用 ctrl-shift-g 产生)：

```
const String ServiceName = "范例蓝牙传递数据服务";
const String ServiceGUI =
"{9827B0D2-66FF-4B30-A3C3-59F38ED59B61}";
```

下面就是 CreateBTTextService() 方法的实作程序代码，005 行建立 TServerConnectionThread 对象，006 行建立 TBluetoothServerSocket 对象并指定给 TServerConnectionThread 对象的 ServerSocket 特性以便 TServerConnectionThread 对象使用来接受客户端的连络请求，最后在 007 行启示执行 TServerConnectionThread 对象代表的独立线程：


```
001   void TfmMainForm::CreateBTTextService()
002   {
003     try
004     {
005       scThread = new TServerConnectionThread(true);
006       scThread->ServerSocket =
Bluetooth1->CurrentAdapter->CreateServerSocket(ServiceName,
StringToGUID(ServiceGUI), false);
007       scThread->Start();
008       PostBTMessage(" - 成功建立服务 : '" + ServiceName + "'");
009     }
010     catch(Exception& ex)
011     {
012       PostBTMessage(ex.Message);
013     }
014   }
015
016   void TfmMainForm::PostBTMessage(const String sMessage)
```

```
017    {
018      mmMessages->Lines->Add(sMessage);
019      mmMessages->GoToTextEnd();
020    }
```

TServerConnectionThread 是从 TThread 继承下来的线程类别，它的功能是在一个独立的线程中建立 TBluetoothSocket 对象准备接受客户端蓝牙的链接并接收资料。

因此下面即是 TServerConnectionThread 类别的宣告，它的 FSocket 即是使用来接受客户端蓝牙链接并接收数据的 TBluetoothSocket 对象变量，而 FData 则是 FSocket 从客户端接收的数据：

```
 #ifndef uTServerConnectionThreadH
#define uTServerConnectionThreadH
//---------------------------------------------------------------------
#include <System.Classes.hpp>
#include <System.Bluetooth.hpp>
#include <System.Bluetooth.Components.hpp>
//---------------------------------------------------------------------
class TServerConnectionThread : public TThread
{
private:
  TBluetoothServerSocket *FServerSocket;
  TBluetoothSocket *FSocket;
  TBytes FData;

  void __fastcall TThreadMethod(void);
  void __fastcall TThreadMethodException(void);
protected:
  void __fastcall Execute();
public:
  __fastcall TServerConnectionThread(bool CreateSuspended);
    __fastcall virtual ~TServerConnectionThread(void);

__property TBluetoothServerSocket* ServerSocket =
{read=FServerSocket, write=FServerSocket, nodefault};
};
```

当 TServerConnectionThread 对象在前面 CreateBTTextService()方法的第 7 行启动之后就会执行它的 Execute()方法，在 013 行先接受客户端蓝牙链接以取得 TBluetoothSocket 对象，如果链接成功就在 020 行呼叫 ReadData()方法接受客户端蓝牙传递来的数据，接着在 023 行藉由 Synchronize()方法把接受来的数据显示在主窗体的 UI 中，由于传递来的数据型态是 TBytes，因此我们需要藉由 TEncoding::UTF8->GetString()方法把 TBytes 转成字符串型态：

```
001   void __fastcall TServerConnectionThread::Execute()
002   {
003     TBluetoothSocket *ASocket;
004     String Msg;
005
006     while (!Terminated)
007     {
008       try
009       {
010         ASocket = NULL;
011         while ( (!Terminated) && (ASocket == NULL) )
012         {
013           ASocket = FServerSocket->Accept(100);
014         }
015         if(ASocket != NULL)
016         {
017           FSocket = ASocket;
018           while (!Terminated)
019           {
020             FData = ASocket->ReadData();
021             if(FData.Length > 0)
022             {
023               Synchronize(TThreadMethod);
024             }
025             Sleep(100);
026           }
027         }
028       }
029       catch (Exception& ex)
```

```
030       {
031         Msg = ex.Message;
032         Synchronize(TThreadMethodException);
033       }
034     }
035   }
036
037   void __fastcall
TServerConnectionThread::TThreadMethod(void)
038   {
039
  fmMainForm->PostBTMessage(TEncoding::UTF8->GetString(FData));
040   }
041
042   void __fastcall
TServerConnectionThread::TThreadMethodException(void)
043   {
044       fmMainForm->PostBTMessage("伺服端链接已关闭: " + Msg);
045   }
```


由于 TServerConnectionThread 类别中拥有 2 个建立的对象 FSocket 和 FServerSocket，因此在它的解构元中必备释放这 2 个对象：

```
__fastcall
TServerConnectionThread::~TServerConnectionThread(void)
{
  delete FSocket;
  delete FServerSocket;
}
```

另外在主窗体的”移除通讯管道” 按钮中也需要释放我们建立的对象和资源，因此它的 OnClick 事件呼叫 FreeBTTextService()方法：

```
void __fastcall TfmMainForm::btnRemoveTextServiceClick(TObject
*Sender)
{
  FreeBTTextService();
}
```

FreeBTTextService 方法先判断是否建立过 TServerConnectionThread 对象接受数据，如果有的话就先停止它的执行再释放它：

```
void TfmMainForm::FreeBTTextService()
{
  if (scThread != NULL)
  {
  scThread->Terminate();
  scThread->WaitFor();
  delete scThread;
  PostBTMessage(" - 连结服务已移除 -");
  }
}
```

现在我们已完成蓝牙伺服端的开发工作，再来需要完成客户端的开发工作，客户端需要建立一个 TBluetoothSocket 对象要求链接，在伺服端接受之后就可以使用这个 TBluetoothSocket 对象传送数据给伺服端。

因此在主窗体的”传送文字”按钮中我们需要先找到配对需要传递数据的伺服端蓝牙，再建立客户端的 TBluetoothSocket 对象以传递数据给伺服端蓝牙。因此在”传送文字”按钮的 OnClick 事件中先呼叫 FillPairDeviceInfo()方法找到配对蓝牙设备，再呼叫 SendDataToServer()方法开始传递数据：


```
procedure TfmMainForm.btnSendTextClick(Sender: TObject);
begin
  FillPairDeviceInfo;
  SendDataToServer;
end;
```

下面是 SendDataToServer() 方法的主程序代码部份，SendDataToServer() 方法先呼叫 CreateClientSocket() 方法建立客户端 TBluetoothSocket 对象，再呼叫 SendData()方法传递数据：

```
 void TfmMainForm::SendDataToServer()
{
  try
  {
  CreateClientSocket();
  SendData();
  }
```

```
  catch (Exception& Ex)
  {
    PostBTMessage(Ex.Message);
    delete FClientSocket;
  }
}
```

CreateClientSocket()方法先于 007 行找到用户选择要传递数据的配对蓝牙设备，再呼叫代表配对蓝牙设备的 pairedDevice 对象的 CreateClientSocket()方法建立客户端的 TBluetoothSocket 对象，请注意 CreateClientSocket()方法的第 1 个参数是伺服端和客户端使用来建立连结的相同 GUID 值。

在建立了客户端 TBluetoothSocket 对象之后就可以于 012 行呼叫它的 Connect()方法链接伺服端蓝牙设备：

```
001    void TfmMainForm::CreateClientSocket()
002    {
003      TBluetoothDevice *pairedDevice;
004
005      if (FClientSocket == NULL)
006      {
007        pairedDevice =
GetThePairedDevice(cbPairDevices->Items->Strings[cbPairDevices->
ItemIndex]);
008        PostBTMessage(GetServiceName(pairedDevice,
ServiceGUI));
009        FClientSocket =
pairedDevice->CreateClientSocket(StringToGUID(ServiceGUI),
false);
010        if (FClientSocket != NULL)
011        {
012          FClientSocket->Connect();
013          PostBTMessage(L"蓝牙已连结");
014        }
015        else
016          PostBTMessage(L"发生错误, 蓝牙连结超时!");
017      }
```

```
018    }
```

上面的 FClientSocket 变量当然是宣告为 TBluetoothSocket 型态的对象变量：

```
   TBluetoothSocket *FClientSocket;
```

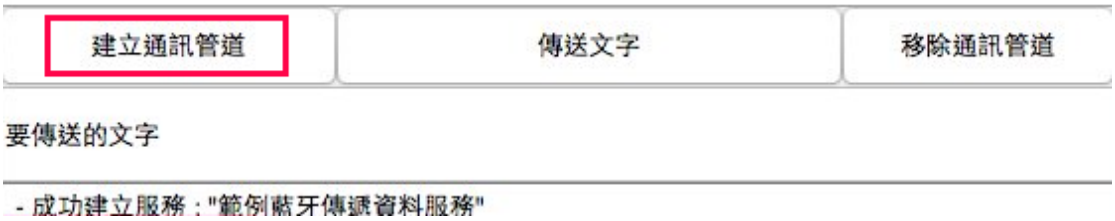
最后的 SendData()方法则非常的简单，它使用刚才建立的 FClientSocket 对象呼叫它的 SendData()方法传递数据，由于 SendData()方法需要传递 TBytes 型态的数据，因此在 003 行需要藉由 TEncoding::UTF8->GetBytes()方法把主窗体中 TEdit 组件中的文字字符串型态数据先转换成 TBytes 型态的数据：

```
001    void TfmMainForm::SendData()
002    {
003      TBytes ToSend = TEncoding::UTF8->GetBytes(edtText->Text);
004      FClientSocket->SendData(ToSend);
005      PostBTMessage(L"讯息已传送!");
006    }
```


到这里我们也完成了客户端的开发工作，接下来我们就可以试着执行范例程序来看看它是否能成功的工作，在下面我们就使用 Mac 做为蓝牙伺服端，它执行 OSX Yosemite 10.10.2 版，另外再使用 HTC M8 做为蓝牙客户端，M8 执行了 Android 4.4 以上的版本。

首先部署此范例程序到 OSX 中执行，先如下图点选”查询蓝牙设备”按钮找到 M8 手机，再点选 M8 然后再点选”配对”按钮以配对 M8：

最后再点选”建立通讯管道”按钮建立伺服端 TBluetoothSocketh 对象等待稍后客户端链接：



接着在客户端 M8 手机中执行范例 App，找到配对的 Mac 机器，再点选”传送文字”按钮建立客户端 TBluetoothSocketh 对象，链接伺服端并传递数据给伺服端：

下面的 2 个画面就是伺服端和客户端的执行结果，我们可以看到 M8 手机中的数据果然藉由蓝牙传递到 Mac 机器中了。

12 呼叫 Android 系统功能

在 Android 平台 C++Builder 产生的 App 是原生机械码而不是 Java 的虚拟程序代码，不过 Android 的功能大多是用 Java 撰写的因此在一些应用场合我们仍然需要在 C++Builder 中呼叫 Java 的程序代码或是 API。在前面的章节中已经有 2 个范例说明如何在 C++Builder 中呼叫 Java 的功能，在本小节中将做比较完整的说明并使用数个范例来展示。

开发人员在使用 C++Builder 呼叫 Java 的 API 或是程序代码时需要了解一些基本的知识才可以顺利的完成呼叫工作。这些知识有些是使用在呼叫 Java 的 API 需要的，有的则是在呼叫 C++Builder 宣未封装的 API 时需要的，下面的表格说明了这些最重要的知识：

需要了解的知识	说明
JObjectClass 界面	C++Builder 使用来封装 Java API 类别的内容，例如类别常数(class constant)，类别方法等
JObject 界面	C++Builder 使用来封装 Java API 对象的内容，例如对象方法等
TJavaGenericImport 类别	C++Builder 使用来合成 JObjectClass 和 JObject 接口成为 C++Builder 类别，如此一来开发人员可建

	立此类别物来呼叫 Java API。此类别是泛型类似因此可使用封装任何的 JObjectClass 和 JObject 接口
JavaSignature 属性	使用来指定 JObject 接口封装的 Java 类别
init 方法	TJavaGenericImport 类别的建构元(constructor)，由于 TJavaGenericImport 类别是封装 Java 类别，因此要真正建立 JVM 中的 Java 对象，C++Builder 程序代码要呼叫此 init 方法
cdecl 呼叫方式宣告	在使用 JObject 接口封装的 Java API 时，一定要使用 cdecl 的呼叫方式

由于在 RAD Studio 中 Android 的 SDK 大都是使用 C++Builder 程序语言定义的，因对于 C++Builder 的开发人员来说在了解了上面的内容后更需要知道如何藉由 C/C++来呼叫 Android 的 SDK。一般来说 C++Builder 开发人员可以藉由

```
_di_C++Builder 接口名称
```

和

```
_di_C++Builder 类别名称
```


来呼叫 Android 的 SDK，也许让我们使用一个简单的范例来说明就很容易了解了。

例如在下面即将说明的范例中我们需要呼叫 Android SDK 的 JContentResolver 接口中的方法，而 JContext 是由 DC++Builder 程序语言定义如下：

```
  [JavaSignature('android/content/ContentResolver')]
  JContentResolver = interface(JObject)
    ['{774C50C1-66DC-489E-9CAC-5434A5DE7CE0}']
    function acquireContentProviderClient(uri: Jnet_Uri):
JContentProviderClient; cdecl; overload;
    …
```

在 C++Buider 中对于 C++Builder 定义的 JContentResolver 接口，我们使用 typedef 复位义了 C++Builder 的接口变量：

```
__interface JContentResolver;
typedef System::C++BuilderInterface<JContentResolver>
_di_JContentResolver;
```

因此在 C++Builder 中我们只使用使用_di_JContentResolver 就可以呼叫 Android SDK 的 JContentResolver 接口中的方法。

同样的对于 C++Builder 定义的 JContentResolverClass 类别接口：

```
  JContentResolverClass = interface(JObjectClass)
    ['{29F2ED97-64A0-435B-A79C-7B8F80E6659A}']
    {class} function _GetCURSOR_DIR_BASE_TYPE: JString; cdecl;
…
```

在 C++Builder 中也只需要在 JContentResolverClass 名称之前加上"_di_"即可：

```
__interface JContentResolverClass;
typedef System::C++BuilderInterface<JContentResolverClass>
_di_JContentResolverClass;
```

如果您不知道_di_的意义，其实它就是代表"**C++Builder I**nterface"。

接着 C++Builder 便用 TJContentResolver 类别封装这 2 个接口，这也意味 C++Builder 的开发人员可以使用 TJContentResolver 类别来取得这 2 个接口。例如要取得_di_JContentResolverClass，那我们可以使用如下的程序代码：


```
_di_JContentResolverClass  jcrc = TJContentResolver::JavaClass;
```

现在就让我们使用一个范例来说明如何使用上面的知识来存取 Android 手机中的联系人信息。

12-1 呼叫 Java 类别程序代码

首先让我们使用一个简单的 Java 类别来展示如何让 C++Builder 可以呼叫 Java 程序代码，在这个讨论的过程中您将学习到数个重要的技术来帮助您了解如何能够不直接使用 JNI 方式而让 C/C++可以呼叫 Java。

下面是一个非常简单的 Java 类别，让我们用 C++Builder 写一个 Android App 来呼叫其中的 setIntValue()方法设定 RTWrapClassTest1 类别对象的 storedintvalue 数值，再呼叫其中的 getIntValue ()方法取出这个数值看看从 C++Builder 设定 Java 数值是否能成功：

```
package com.xe5test.myjavaclasastest;


import android.util.Log;
```

```
public class RTWrapClassTest1 {
    int storedintvalue;

    public int getIntValue {
        Log.d("classtesting","getIntValue called" );
        return storedintvalue + 3;
    }

    public void setIntValue(int newvalue) {
        Log.d("classtesting", "setIntValue called");
        storedintvalue = newvalue;
    }

    public void stunt(String s, int n) {
        Log.d("classtesting", "stunt called");

        for (int x = 10; x < 12; x = x + 1)
            Log.d("classtesting", s + ' ' + n);
    }
}
```


这个 Java 类别被执编译并封装在 XE5ClassTesting1.jar 中，现在我们就可以准备来呼叫了。

在 RAD Studio 中我们可以使用如下的步骤来让 C/C++直接呼叫 Java 程序代码：

1. 使用 Java2OP 工具把 Java 类别宣告转成 C++Builder 类别宣告
2. 使用 dccaarm.exe 编译程序和编译程序指令把 C++Builder 类别宣告直接转成 C/C++的表头宣告
3. 在 C++Builder 项目中 include 上面步骤产生的 C/C++的表头宣告，编译成.O 的档案并直接连结到最后的 App 中
4. 使用 C++Builder 项目管理员在 App 中加入 XE5ClassTesting1.jar 档并部署

在下面的内容中我们将一一的说明如何完成每一个步骤。

步骤 1 把 Java 类别宣告转成 C++Builder 类别宣告

首先您可以使用 RAD Studio 内附的 Java2OP 工具把 Java Jar 檔中的 Java 类别转成 C++Builder 类别宣告，例如您可以使用如下的命令把 mylib.jar 中的 Java 类别转成 C++Builder 类别宣告：

```
Java2OP.exe -jar mylib.jar
```

因此在此步骤中您只需要使用下面的指令：

```
Java2OP.exe -jar XE5ClassTesting1.jar
```

就可以产生类似如下的 C++Builder 类别宣告：

```
unit com.xe5.ClassTesting1.RTClassTesting1;

interface

uses
  AndroidAPI.JNIBridge,
  Androidapi.JNI.JavaTypes;

type
  JRTClassTesting1 = interface;

  JRTClassTesting1Class = interface(JObjectClass)
    ['{8EF97555-32BC-4262-B17E-7A5157944E97}']
    function getIntValue : Integer; cdecl;
// ()I A: $1
    function init : JRTClassTesting1; cdecl;
// ()V A: $1
    procedure setIntValue(newvalue : Integer) ; cdecl;
// (I)V A: $1
    procedure stunt(s : JString; n : Integer) ; cdecl;
// (Ljava/lang/String;I)V A: $1
  end;

  [JavaSignature('com/xe5/ClassTesting1/RTClassTesting1')]
  JRTClassTesting1 = interface(JObject)
    ['{983668BC-BD74-4142-8A1F-3DDA43F3E29F}']
```

```
    function getIntValue : Integer; cdecl;
// ()I A: $1
    procedure setIntValue(newvalue : Integer) ; cdecl;
// (I)V A: $1
    procedure stunt(s : JString; n : Integer) ; cdecl;
// (Ljava/lang/String;I)V A: $1
  end;

  TJRTClassTesting1 =
class(TJavaGenericImport<JRTClassTesting1Class,
JRTClassTesting1>)
  end;

implementation

end.
```

让我们把这个 C++Builder 类别宣告储存为 com.xe5.ClassTesting1.RTClassTesting1.pas。

步骤 2 把 C++Builder 类别宣告直接转成 C/C++的表头宣告

有了 Java 类别的 C++Builder 类别宣告之后接下来我们可使用 RAD Studio 中的 dccaarm.exe 这个编译程序直接把 C++Builder 类别宣告直接转成 C/C++的表头宣告，非常的简单。

dccaarm.exe 有一个编译程序令 jphne 就可以帮助我们完成这个工作，因此我们可以简单的使用下面的指令把上面的 com.xe5.ClassTesting1.RTClassTesting1.pas 直接转成 C/C++的表头宣告：

```
dccaarm -jphne  com.xe5.ClassTesting1.RTClassTesting1.pas
```

我们就可以立刻得到 com.xe5.ClassTesting1.RTClassTesting1.hpp 表头档：

```
001    // CodeGear C++Builder
002    // Copyright (c) 1995, 2015 by Embarcadero Technologies, Inc.
003    // All rights reserved
004
```

```
005   // (DO NOT EDIT: machine generated header)
'com.xe5.ClassTesting1.RTClassTesting1.pas' rev: 29.00 (Android)
006
007   #ifndef Com_Xe5_Classtesting1_Rtclasstesting1HPP
008   #define Com_Xe5_Classtesting1_Rtclasstesting1HPP
009
010   #pragma C++Builderheader begin
011   #pragma option push
012   #pragma option -w-     // All warnings off
013   #pragma option -Vx     // Zero-length empty class member
014   #pragma pack(push,8)
015   #include <System.hpp>
016   #include <SysInit.hpp>
017   #include <Androidapi.JNIBridge.hpp>
018   #include <Androidapi.JNI.JavaTypes.hpp>
019
020   //-- user supplied
-----------------------------------------------------------
021
022   namespace Com
023   {
024   namespace Xe5
025   {
026   namespace Classtesting1
027   {
028   namespace Rtclasstesting1
029   {
030   //-- forward type declarations
-----------------------------------------------
031   __interface JRTClassTesting1Class;
032   typedef System::C++BuilderInterface<JRTClassTesting1Class>
_di_JRTClassTesting1Class;
033   __interface JRTClassTesting1;
034   typedef System::C++BuilderInterface<JRTClassTesting1>
_di_JRTClassTesting1;
035   class C++BUILDERCLASS TJRTClassTesting1;
036   //-- type declarations
-------------------------------------------------------
```

```
037    __interface
INTERFACE_UUID("{8EF97555-32BC-4262-B17E-7A5157944E97}")
JRTClassTesting1Class  : public
Androidapi::Jni::Javatypes::JObjectClass
038    {
039        virtual int __cdecl getIntValue(void) = 0 ;
040        HIDESBASE virtual _di_JRTClassTesting1 __cdecl init(void)
= 0 ;
041        virtual void __cdecl setIntValue(int newvalue) = 0 ;
042        virtual void __cdecl
stunt(Androidapi::Jni::Javatypes::_di_JString s, int n) = 0 ;
043    };
044
045    __interface
INTERFACE_UUID("{983668BC-BD74-4142-8A1F-3DDA43F3E29F}")
JRTClassTesting1  : public Androidapi::Jni::Javatypes::JObject
046    {
047        virtual int __cdecl getIntValue(void) = 0 ;
048        virtual void __cdecl setIntValue(int newvalue) = 0 ;
049        virtual void __cdecl
stunt(Androidapi::Jni::Javatypes::_di_JString s, int n) = 0 ;
050    };
051
052    #pragma pack(push,4)
053    class PASCALIMPLEMENTATION TJRTClassTesting1 : public
Androidapi::Jnibridge::TJavaGenericImport__2<_di_JRTClassTesting
1Class,_di_JRTClassTesting1>
054    {
055        typedef
Androidapi::Jnibridge::TJavaGenericImport__2<_di_JRTClassTesting
1Class,_di_JRTClassTesting1>  inherited;
056
057    public:
058        /* TObject.Create */ inline __fastcall
TJRTClassTesting1(void) :
Androidapi::Jnibridge::TJavaGenericImport__2<_di_JRTClassTesting
1Class,_di_JRTClassTesting1> () { }
059        /* TObject.Destroy */ inline __fastcall virtual
```

```
~TJRTClassTesting1(void) { }
060
061    };
062
063    #pragma pack(pop)
064
065    //-- var, const, procedure
-------------------------------------------------
066    }  /* namespace Rtclasstesting1 */
067    }  /* namespace Classtesting1 */
068    }  /* namespace Xe5 */
069    }  /* namespace Com */
070    #if !defined(C++BUILDERHEADER_NO_IMPLICIT_NAMESPACE_USE)
&& !defined(NO_USING_NAMESPACE_COM_XE5_CLASSTESTING1_RTCLASSTEST
ING1)
071    using namespace Com::Xe5::Classtesting1::Rtclasstesting1;
072    #endif
073    #if !defined(C++BUILDERHEADER_NO_IMPLICIT_NAMESPACE_USE)
&& !defined(NO_USING_NAMESPACE_COM_XE5_CLASSTESTING1)
074    using namespace Com::Xe5::Classtesting1;
075    #endif
076    #if !defined(C++BUILDERHEADER_NO_IMPLICIT_NAMESPACE_USE)
&& !defined(NO_USING_NAMESPACE_COM_XE5)
077    using namespace Com::Xe5;
078    #endif
079    #if !defined(C++BUILDERHEADER_NO_IMPLICIT_NAMESPACE_USE)
&& !defined(NO_USING_NAMESPACE_COM)
080    using namespace Com;
081    #endif
082    #pragma pack(pop)
083    #pragma option pop
084
085    #pragma C++Builderheader end.
086    //-- end unit
-----------------------------------------------------------------
087    #endif // Com_Xe5_Classtesting1_Rtclasstesting1
```

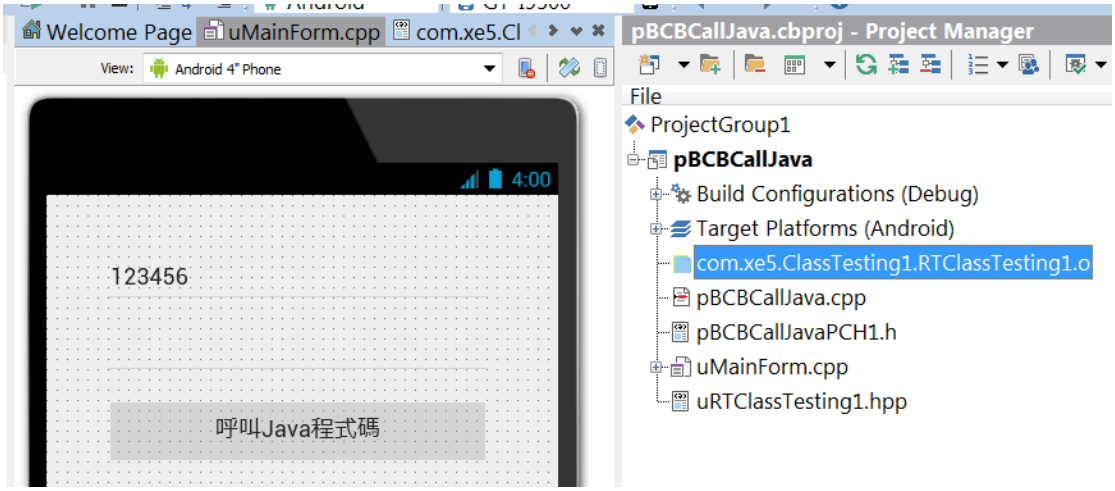

请花些时间仔细观察一下上面表头档的内容，并且对照前面本书说明 C++Builder 和 C++Builder 如何封装 Java 类别和 Android SDK 的规则。例

如在 032 行有一个封装 Java 类别的_di_JRTClassTesting1Class，034 行有一个封装 Java 对象的_di_JRTClassTesting1，最后在 058 行使用了 Androidapi::Jnibridge::TJavaGenericImport__2 封装 JRTClassTesting1Class 和 JRTClassTesting1。整个表头档程序代码都符合本书前面说明的规则。

让我们把这个 C/C++ 表头宣告储存为 com.xe5.ClassTesting1.RTClassTesting1.hpp。

步骤 3 编译成.O 的档案并直接连结到最后的 App 中

现在在 C++Builder IDE 中建立一个 Multi-Device Application 项目如下：



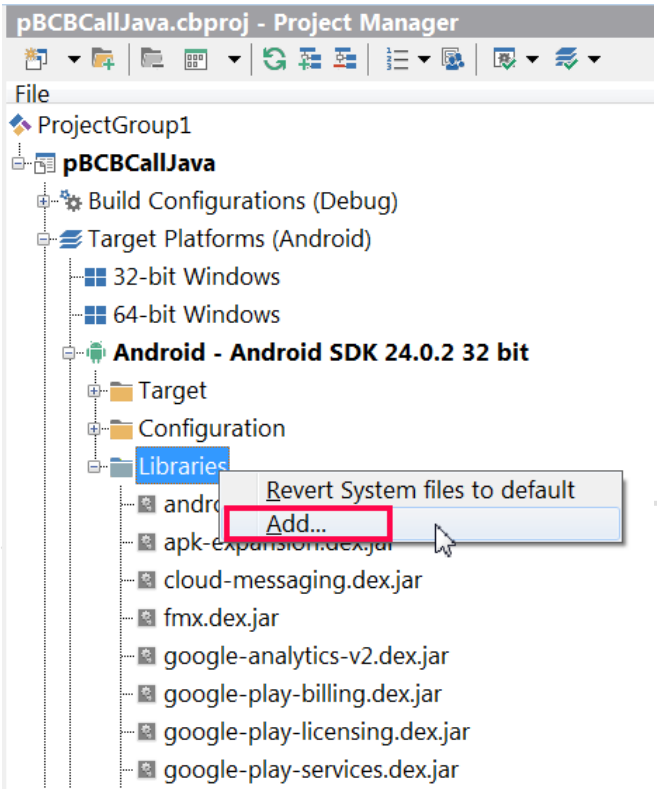
接着在主窗体程序代码中只需要 include 在步骤 2 产生的 C/C++表头宣告档：

```
#include <fmx.h>
#pragma hdrstop

#include "uMainForm.h"
#include "com.xe5.ClassTesting1.RTClassTesting1.hpp"
…
```

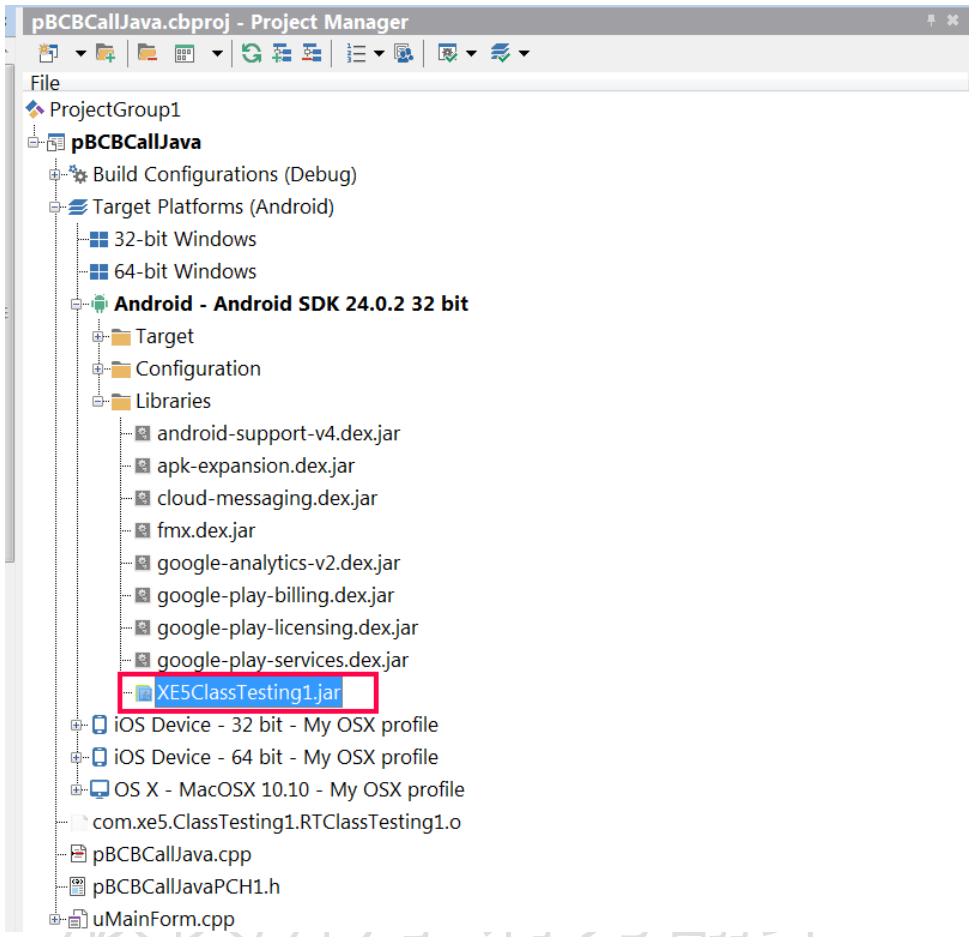
就可以得到.O 的档案了。

步骤 4 加入 Java 的.jar 档并部署

提供了可让程序员在 C++Builder 使用的在 IDE 中 Java 及函式馆中加入客制化的 jar 档以便让 C/C++程序代码可呼叫其中的 Java 程序代码。由于现在我们要呼叫 XE5ClassTesting1.jar 中的 Java 程序代码，因此请到 IDE 的项目管理员中展开 Android 平台，再展开 Libraries 节点，右击鼠标再从突显选单中点选 Add...选项并选择加入 XE5ClassTesting1.jar：



下图就是加入 XE5ClassTesting1.jar 的结果：



在加入了 XE5ClassTesting1.jar 之后 IDE 便会把它封装到 classes.dex 并准备部署到 Android 平台中。

完成了上面的步骤后要让 C/C++呼叫 Java 就简单了，对于 C++Builder 来说就像是呼叫一般的 C/C++程序代码一样。我们只需要使用如下的程序代码即可：

```
001    void __fastcall TForm5::Button1Click(TObject *Sender)
002    {
003      _di_JRTClassTesting1 ijt = TJRTClassTesting1::Create();
004      ijt->setIntValue(StrToInt(Edit1->Text));
005      Edit2->Text = IntToStr(ijt->getIntValue());
006      ijt = NULL;
007    }
```

在 003 行我们藉由呼叫 TJRTClassTesting1::Create() 建立 _di_JRTClassTesting1 型态的对象，而这也等于建立了 Java 的

RTWrapClassTest1 类别对象。接着在 004 和 005 行就可以藉由 ijt 直接呼叫 RTWrapClassTest1 类别对象的 setIntValue()和 getIntValue()方法了，就像直接呼叫由 C/C++语言写的程序代码一样。

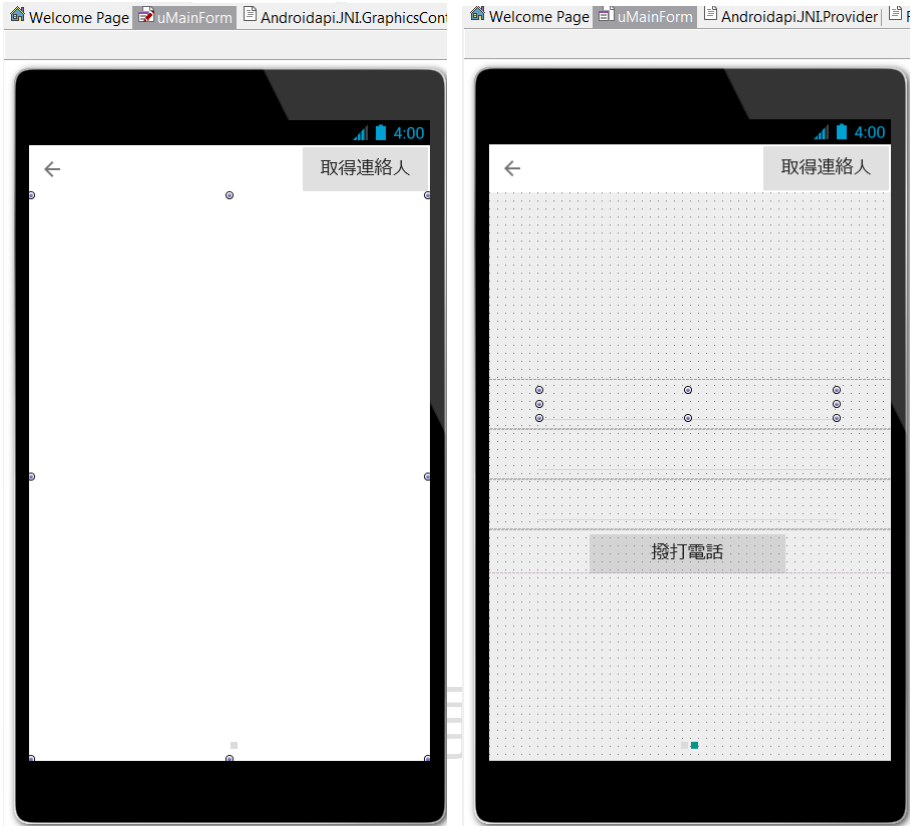
最后编译并部署到 Android 手机中执行这个范例 App，从下面的执行画面可以看到我们成功的使用 C++Builder 呼叫了由 Java 撰写的程序代码。

12-2 呼叫 Java API 存取 Android 联系人信息

上面的范例是说明如何使用 C++Builder 呼叫尚未封装的 Android API 的方法，在本小节中再让我们说明如何使用 C++Builder 呼叫已封装的 Android API 来存取 Android 上的信息。

在许多的 App 应用中经常会需要存取手机中联系人的信息，但目前 C++Builder 尚未封装联系人信息为组件和类别让开发人员使用，不过这也正好让我们使用这个需求做为说明如何在 C++Builder 中呼叫 Java API 的范例。

首先建立一个 Multi-Device 项目并设计如下的 UI，在 TPageControl 的第 1 个页面将显示联系人姓名和电话，点选任何联系人就会在第 2 个页面显示 Email 的信息：



因此我们需要使用 C++Builder 程序代码存取联系人的姓名，电话和 Email，让我们宣告 TContactPerson 类别以储存每一个联系人的信息：

```
class TContactPerson
{
private:
  String FID;
  String FName;
  String FPhone;
  String FEMail;
public:
  __property String ID = {read=FID, write=FID};
  __property String Name = {read=FName, write=FName};
  __property String Phone = {read=FPhone, write=FPhone};
  __property String EMail = {read=FEMail, write=FEMail};
```

```
};
```

在 TContactPerson 类别中的 ID 字段将储存使用来查询 Android 联系人的查询键值，使用这个查询键值才能够查询到联系人的延伸信息，例如电话号码和 Email 等，我们马上就会说明如何能取得此查询键值。

现在让我们开发先说明如何取得联系人的查询键值和显示名称信息，再根据查询键值取得其他需要的信息。在主窗体的”取得联系人”按钮中呼叫 GetContactsInfo()方法：

```
void __fastcall TfmMainForm::btnGetContactsClick(TObject *Sender)
{
  GetContactsInfo();
}
```

GetContactsInfo() 方法会呼叫 GetContentResolver() ，QueryContactInfo()和 DisplayContactInfo()等 3 个方法，它们分别取得查询联系人的_di_JCursor 接口，实际进行查询信息的工作以及显示查询的结果：

```
void TfmMainForm::GetContactsInfo()
{
  GetContentResolver();
  QueryContactInfo();
  DisplayContactInfo();
}
```


要查询 Android 的联系人信息我们需要呼叫 Android API 的 android.content.ContentResolver 的 query()方法：

```
  virtual _di_JCursor __cdecl
query(Androidapi::Jni::Net::_di_Jnet_Uri uri,
Androidapi::Jnibridge::TJavaObjectArray__1<Androidapi::Jni::Java
types::_di_JString> * projection,
Androidapi::Jni::Javatypes::_di_JString selection,
Androidapi::Jnibridge::TJavaObjectArray__1<Androidapi::Jni::Java
types::_di_JString> * selectionArgs,
Androidapi::Jni::Javatypes::_di_JString sortOrder) = 0 /* overload
*/;
  virtual _di_JCursor __cdecl
query(Androidapi::Jni::Net::_di_Jnet_Uri uri,
```

```
Androidapi::Jnibridge::TJavaObjectArray__1<Androidapi::Jni::Java
types::_di_JString> * projection,
Androidapi::Jni::Javatypes::_di_JString selection,
Androidapi::Jnibridge::TJavaObjectArray__1<Androidapi::Jni::Java
types::_di_JString> * selectionArgs,
Androidapi::Jni::Javatypes::_di_JString sortOrder,
Androidapi::Jni::Os::_di_JCancellationSignal cancellationSignal)
= 0 /* overload */;
```

第 1 个参数 uri 代表要查询的内容，第 2 个参数 projection 类似要取得的信息字段，第 3 个参数 selection 类似 SQL 命令的 where 条件，第 4 个参数 selectionArgs 类似 SQL 命令的动态参数值，selectionArgs 的内容会动态带入第 3 个参数 selection 中，最后一个参数 sortOrder 代表查询出来的数据的排序方式。

但要怎么取得 ContentResolver 以呼叫 query 方法呢？

GetContentResolver() 方法藉由存取 C++Builder 定义的 SharedActivityContext() 公共方法取得 _di_JCursor 接口再呼叫 _di_JCursor 接口中的 getContentResolver() 方法即可取得代表 android.content.ContentResolver 的界面_di_JContentResolver。


```
void TfmMainForm::GetContentResolver()
{
  jcr = SharedActivityContext()->getContentResolver();
}
```

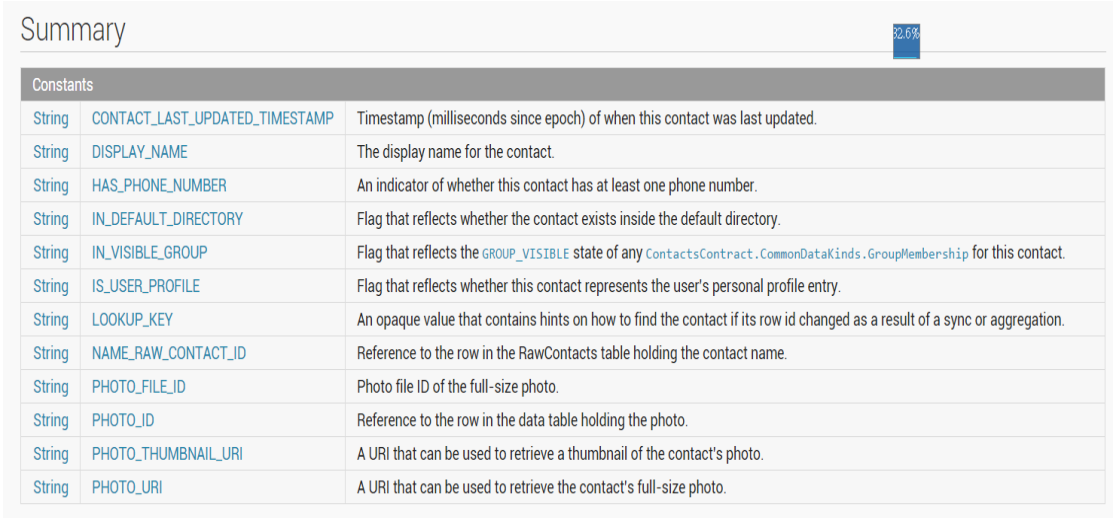
上面的 jcr 是宣告在窗体类别 private 部份的_di_JContentResolver 接口变量：

```
private:  // User declarations
  _di_JContentResolver jcr;
  _di_JCursor jc;
  TList *contacts;
```

取得了_di_JContentResolver 接口之后就可以呼叫它的 query()方法查询联系人信息了，根据下面 URL 的 Android API 的说明

```
http://developer.android.com/reference/android/provider/Contacts
Contract.ContactsColumns.html
```

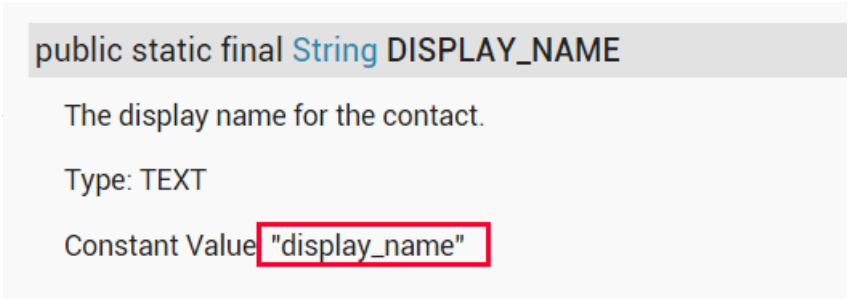
我们可以看到下面的信息，其中的 DISPLAY_NAME 正是我们要查询的联系人名称，而 LOOKUP_KEY 正是查询键值：



Summary

Constants		
String	CONTACT_LAST_UPDATED_TIMESTAMP	Timestamp (milliseconds since epoch) of when this contact was last updated.
String	DISPLAY_NAME	The display name for the contact.
String	HAS_PHONE_NUMBER	An indicator of whether this contact has at least one phone number.
String	IN_DEFAULT_DIRECTORY	Flag that reflects whether the contact exists inside the default directory.
String	IN_VISIBLE_GROUP	Flag that reflects the GROUP_VISIBLE state of any ContactsContract.CommonDataKinds.GroupMembership for this contact.
String	IS_USER_PROFILE	Flag that reflects whether this contact represents the user's personal profile entry.
String	LOOKUP_KEY	An opaque value that contains hints on how to find the contact if its row id changed as a result of a sync or aggregation.
String	NAME_RAW_CONTACT_ID	Reference to the row in the RawContacts table holding the contact name.
String	PHOTO_FILE_ID	Photo file ID of the full-size photo.
String	PHOTO_ID	Reference to the row in the data table holding the photo.
String	PHOTO_THUMBNAIL_URI	A URI that can be used to retrieve a thumbnail of the contact's photo.
String	PHOTO_URI	A URI that can be used to retrieve the contact's full-size photo.

点选上面 2 个字段可以看到代表这 2 个字段的常数值是”display_name”和”lookup”：

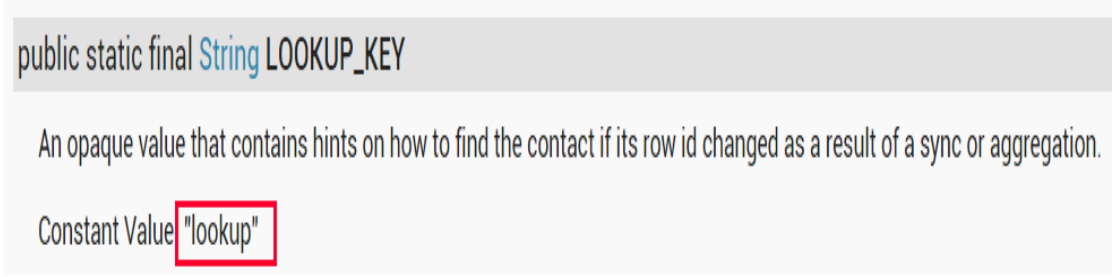


public static final String DISPLAY_NAME

The display name for the contact.

Type: TEXT

Constant Value: "display_name"



public static final String LOOKUP_KEY

An opaque value that contains hints on how to find the contact if its row id changed as a result of a sync or aggregation.

Constant Value: "lookup"

查询 Android 联系人的信息有一个规则，那就是先要查询到您要查询数据的字段索引值，再根据此字段索引值来查询真正的字段内容信息。例如如果我们要查询联系人名称，那要先根据上面的”display_name”常数查询它的字段索引值，再根据此字段索引值查询到真正的联系人名称。

了解了这些基本的观念后就可以准备呼叫_di_JContentResolver 接口的 query()方法查询联系人信息了，但我们要传入正确的参数给 query()方法。

Query()方法第 1 个参数是表示要查询什么内容，根据 Android API 文件联系人是定义在 ContactsContract.Contacts 中，而 C++Builder 使用 TJContactsContract_Contacts 定义了这个类别(C++Builder 原始定义)：

```
  [JavaSignature('android/provider/ContactsContract$Contacts')]
  JContactsContract_Contacts = interface(JObject)
    ['{F174D654-2BBC-4854-BB3A-23F403CE38D8}']
  end;
  TJContactsContract_Contacts =
class(TJavaGenericImport<JContactsContract_ContactsClass,
JContactsContract_Contacts>) end;
```

在 C++Builder 的 Androidapi.Jni.Provider.hpp 表头档中定义如下：

```
__interface
INTERFACE_UUID("{3BA68A03-57B0-426D-B452-93635322BBDE}")
JContactsContract_ContactsClass  : public
Androidapi::Jni::Javatypes::JObjectClass
{
…
```


在 JContactsContract_ContactsClass 中的 CONTENT_URI 特性正是 query()方法需要的第 1 个参数值，代表我们要查询联系人信息：

```
__property Androidapi::Jni::Net::_di_Jnet_Uri CONTENT_URI =
{read=_GetCONTENT_URI};
```

所以在下面的 QueryContactInfo()方法中我们使用 jcr 呼叫它的 query()方法，由于现在我们要取得所有联系人信息，因此 query()方法的 2，3，4 参数都使用 NULL。Query()方法最后一个参数我们藉由呼叫 StringToJString()把 C++Builder 的字符串转成 Java 字符串要求所有联系人信息以名称排序：

```
void TfmMainForm::QueryContactInfo()
{
  TContactPerson *aContact;

  jc =
jcr->query(TJContactsContract_Contacts::JavaClass->CONTENT_URI,
NULL, NULL, NULL, StringToJString("display_name ASC"));
  jc->moveToFirst();
```

```
  while (jc->moveToNext())
  {
  aContact = new TContactPerson();
  aContact->ID = GetContactID();
  aContact->Name = GetContactName();
  aContact->Phone = GetContactPhone(aContact);
  aContact->EMail = GetContactEMail(aContact);
  contacts->Add(aContact);
  }
}
```

query()方法成功执行之后即会回传_di_JCursor 接口(C++Builder 原始定义)：

```
  [JavaSignature('android/database/Cursor')]
  JCursor = interface(JCloseable)
```

在 C++Builder 中重新定义如下：

```
typedef System::C++BuilderInterface<JCursor> _di_JCursor;
class C++BUILDERCLASS TJCursor;
```

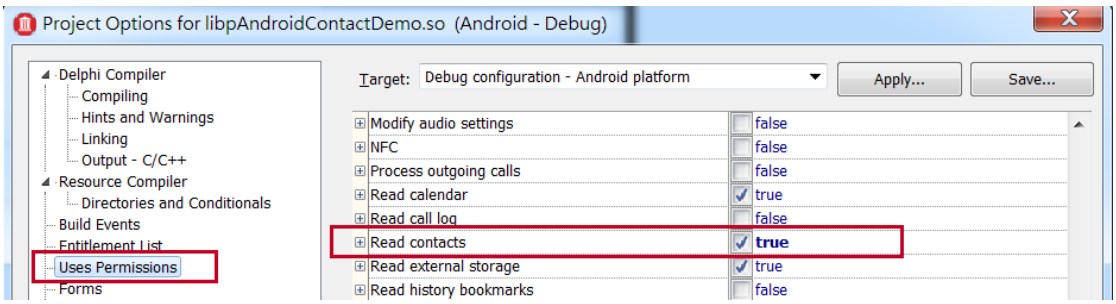

我们就可以使用_di_JCursor 接口中的方法再存取我们需要的信息值，由于query()方法回传的类似一个数据表，因此我们先呼叫_di_JCursor 接口中的moveToFirst()方法定位到第 1 笔数据，再进入循环藉由呼叫 moveToNext()方法依序的存取每一笔数据。

GetContactID()，GetContactName()方法就是使用回传的_di_JCursor 接口变量 jc 来实际查询我们需要的查询键值和联系人名称。从下面的程序代码我们可以很清楚的看到它使用的规则，先使用 jc 的 getColumnIndex()方法以字段常数取得字段索引值，接着再次使用 jc 的 getString()方法藉由字段索引值来取得字段内容值，最后呼叫 JStringToString()把 Java 字符串转回 C++Builder 字符串：

```
String TfmMainForm::GetContactID()
{
  return
UTF8String( JStringToString( jc->getString(jc->getColumnIndex(St
ringToJString("lookup"))) ) );
}
```

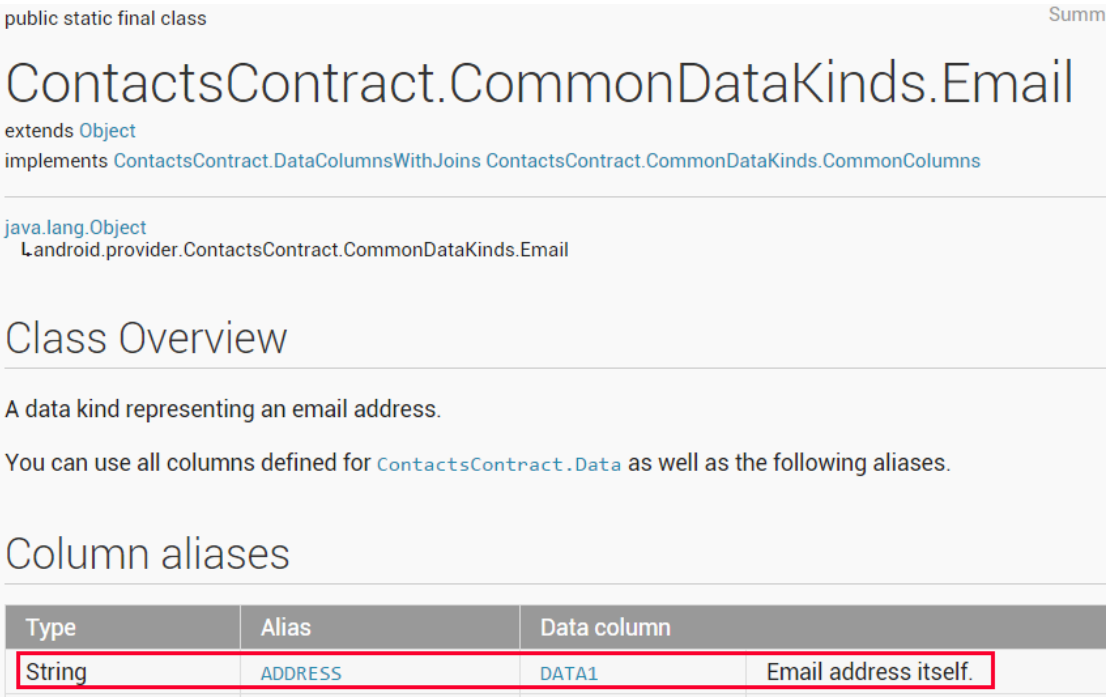
```
String TfmMainForm::GetContactName()
{
  return
UTF8String( JStringToString( jc->getString(jc->getColumnIndex(St
ringToJString("display_name"))) ) );
}
```

在试着执行此范例 App 之前您必须开启 App 读取连络信息的权限，请点选 Tools | Options...选单，在 User Permissions 项目中如下勾选设定 Read contacts 为 true：



完成上面的程序代码和权限设定后执行此范例程序就可以看到类似如下的结果，果然可以存取到 Android 中联系人的信息了：

那么要如何再查询到电话和 Email 等其他的信息呢？这就需要使用前面取得的查询键值了，因为这些额外的数据 Android 是定义在 ContactsContract.CommonDataKinds.Email，ContactsContract.CommonDataKinds.Phone 和 ContactsContract.CommonDataKinds.Photo 等类别中。例如下图是 ContactsContract.CommonDataKinds.Email 类别的定义，我们可以看到它的 ADDRESS 一栏正是我们需要的联系人 EMAIL：


public static final class Summ

ContactsContract.CommonDataKinds.Email

extends Object

implements ContactsContract.DataColumnsWithJoins ContactsContract.CommonDataKinds.CommonColumns

java.lang.Object
↳android.provider.ContactsContract.CommonDataKinds.Email

Class Overview

A data kind representing an email address.

You can use all columns defined for ContactsContract.Data as well as the following aliases.

Column aliases

Type	Alias	Data column	
String	ADDRESS	DATA1	Email address itself.

所以要查询特定联系人的 Email ,我们必须使用前面那个特定联系人的查询键值到 ContactsContract.CommonDataKinds.Email 类别中查询，这类似数据库的 Foreign Key 的概念。在 C++Builder 中 ContactsContract.CommonDataKinds.Email 类别已经由 TJCommonDataKinds_Email 定义了(C++Builder 原始定义)：

```
[JavaSignature('android/provider/ContactsContract$CommonDataKinds$Email')]
  JCommonDataKinds_Email = interface(JObject)
    ['{E1AC9554-07BA-4399-8907-B92FA714B486}']
  end;
  TJCommonDataKinds_Email =
class(TJavaGenericImport<JCommonDataKinds_EmailClass,
```

```
JCommonDataKinds_Email>) end;
```

在 C++Builder 中重新定义如下：

```
__interface
INTERFACE_UUID("{E1AC9554-07BA-4399-8907-B92FA714B486}")
JCommonDataKinds_Email  : public
Androidapi::Jni::Javatypes::JObject
{

};

#pragma pack(push,4)
class PASCALIMPLEMENTATION TJCommonDataKinds_Email : public
Androidapi::Jnibridge::TJavaGenericImport__2<_di_JCommonDataKind
s_EmailClass,_di_JCommonDataKinds_Email>
{
  typedef
Androidapi::Jnibridge::TJavaGenericImport__2<_di_JCommonDataKind
s_EmailClass,_di_JCommonDataKinds_Email> inherited;

public:
  /* TObject.Create */ inline __fastcall
TJCommonDataKinds_Email(void) :
Androidapi::Jnibridge::TJavaGenericImport__2<_di_JCommonDataKind
s_EmailClass,_di_JCommonDataKinds_Email> () { }
  /* TObject.Destroy */ inline __fastcall virtual
~TJCommonDataKinds_Email(void) { }

};

#pragma pack(pop)
```


因此在下面的 GetContactEMail() 方法中 005~007 我们使用 Androidapi::Jnibridge::TJavaObjectArray__1<Androidapi::Jni::Javatypes::_di_JString>泛型类别建立要查询的 Email 地址字段，008~010 建立查询条件，013 代表要查询 Email 信息内容，010 则带入先前已取得的此联系人的查询键值。012 行同样取得代表查询结果的 _di_JCursor 接口变量

emailCursor。015 行判断如果查询到此联系人的 Email 信息就在 021 行取得他的 Email 地址：

```
001    String TfmMainForm::GetContactEMail(TContactPerson
*aContact)
002    {
003      String sResult = "";
004
005
Androidapi::Jnibridge::TJavaObjectArray__1<Androidapi::Jni::Java
types::_di_JString> *sProjection = new
Androidapi::Jnibridge::TJavaObjectArray__1<Androidapi::Jni::Java
types::_di_JString>(1);
006      sProjection->Items[0] =
TJCommonDataKinds_Email::JavaClass->ADDRESS;
007
008
Androidapi::Jnibridge::TJavaObjectArray__1<Androidapi::Jni::Java
types::_di_JString> *sWhere = new
Androidapi::Jnibridge::TJavaObjectArray__1<Androidapi::Jni::Java
types::_di_JString>(2);
009      sWhere->Items[0] =
TJCommonDataKinds_Email::JavaClass->CONTENT_ITEM_TYPE;
010      sWhere->Items[1] = StringToJString(aContact->ID);
011
012      _di_JCursor emailCursor =
jcr->query(TJContactsContract_Data::JavaClass->CONTENT_URI,sProj
ection, StringToJString("mimetype = ? AND lookup = ?"), sWhere,
NULL);
013      try
014      {
015        if (emailCursor->getCount() > 0)
016        {
017          emailCursor->moveToNext();
018          try
019          {
020            sResult =
UTF8String( JStringToString(emailCursor->getString(0)) );
```

```
021         }
022         catch(...)
023         {
024             ;
025         } }
026     }
027     __finally
028     {
029       emailCursor->close();
030       emailCursor = NULL;
031     }
032
033     return sResult;
034   }
```

要查询联系人的电话使用的方法和刚才查询 Email 的方式类似，只是我们需要到 ContactsContract.CommonDataKinds.Phone 类别中查询：

ContactsContract.CommonDataKinds.Phone

extends Object
implements ContactsContract.DataColumnsWithJoins ContactsContract.CommonDataKinds.CommonColumns

java.lang.Object
  ↳android.provider.ContactsContract.CommonDataKinds.Phone

Class Overview

A data kind representing a telephone number.

You can use all columns defined for `ContactsContract.Data` as well as the following aliases.

Column aliases

Type	Alias	Data column
String	NUMBER	DATA1

而 ContactsContract.CommonDataKinds.Phone 类别已经封装在 C++Builder 的 TJCommonDataKinds_Phone 类别中(C++Builder 原始定义)：

```
[JavaSignature('android/provider/ContactsContract$CommonDataKind
```

```
s$Phone')]
  JCommonDataKinds_Phone = interface(JObject)
    ['{B3BC4EC6-2FEB-4F10-9D45-275FDE754A78}']
  end;
  TJCommonDataKinds_Phone =
class(TJavaGenericImport<JCommonDataKinds_PhoneClass,
JCommonDataKinds_Phone>) end;
```

在 C++Builder 中重新定义如下：

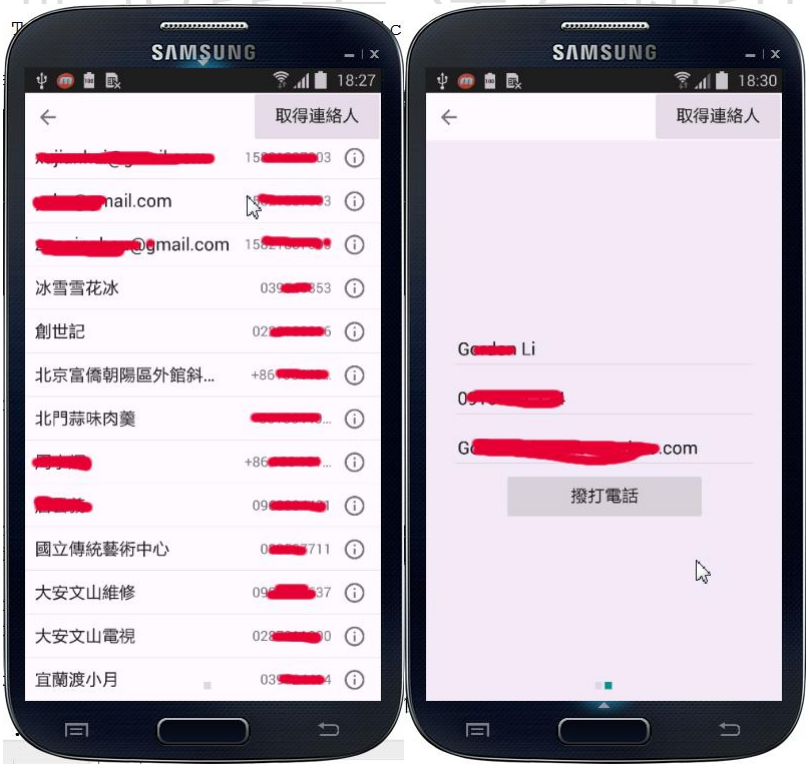
```
__interface
INTERFACE_UUID("{26C70162-6E79-45BB-9BE5-6C1EF293FD07}")
JCommonDataKinds_PhoneClass  : public
Androidapi::Jni::Javatypes::JObjectClass
{
…
}

#pragma pack(push,4)
class PASCALIMPLEMENTATION TJCommonDataKinds_Phone : public
Androidapi::Jnibridge::TJavaGenericImport__2<_di_JCommonDataKind
s_PhoneClass,_di_JCommonDataKinds_Phone>
{
  typedef
Androidapi::Jnibridge::TJavaGenericImport__2<_di_JCommonDataKind
s_PhoneClass,_di_JCommonDataKinds_Phone>  inherited;

public:
  /* TObject.Create */ inline __fastcall
TJCommonDataKinds_Phone(void) :
Androidapi::Jnibridge::TJavaGenericImport__2<_di_JCommonDataKind
s_PhoneClass,_di_JCommonDataKinds_Phone> () { }
  /* TObject.Destroy */ inline __fastcall virtual
~TJCommonDataKinds_Phone(void) { }

};

#pragma pack(pop)
```

因此 GetContactPhone()方法使用的技巧和 GetContactEMail()方法类似，我们就不再赘述了：

```
001    String TfmMainForm::GetContactPhone(TContactPerson
*aContact)
002    {
003      String sResult = "";
004
005
Androidapi::Jnibridge::TJavaObjectArray__1<Androidapi::Jni::Java
types::_di_JString> *sProjection = new
Androidapi::Jnibridge::TJavaObjectArray__1<Androidapi::Jni::Java
types::_di_JString>(1);
006      sProjection->Items[0] =
TJCommonDataKinds_Phone::JavaClass->NUMBER;
007
008
Androidapi::Jnibridge::TJavaObjectArray__1<Androidapi::Jni::Java
types::_di_JString> *sWhere = new
Androidapi::Jnibridge::TJavaObjectArray__1<Androidapi::Jni::Java
types::_di_JString>(2);
009      sWhere->Items[0] =
TJCommonDataKinds_Phone::JavaClass->CONTENT_ITEM_TYPE;
010      sWhere->Items[1] = StringToJString(aContact->ID);
011
012      _di_JCursor PhoneCursor =
jcr->query(TJContactsContract_Data::JavaClass->CONTENT_URI,sProj
ection, StringToJString("mimetype = ? AND lookup = ?"), sWhere,
NULL);
013      try
014      {
015        if (PhoneCursor->getColumnCount() > 0)
016        {
017          PhoneCursor->moveToNext();
018          try
019          {
020            sResult =
UTF8String( JStringToString(PhoneCursor->getString(0)) );
```

```
021          }
022          catch(...)
023          {
024              ;
025          }
026        }
027      }
028      __finally
029      {
030        PhoneCursor->close();
031        PhoneCursor = NULL;
032      }
033
034      return sResult;
035    }
```

最后编译范例 App 并执行在第 1 个页面可查询到所有联系人信息，点选任一联系人就可以在第 2 个页面中看到他的电话和 Email 信息了。

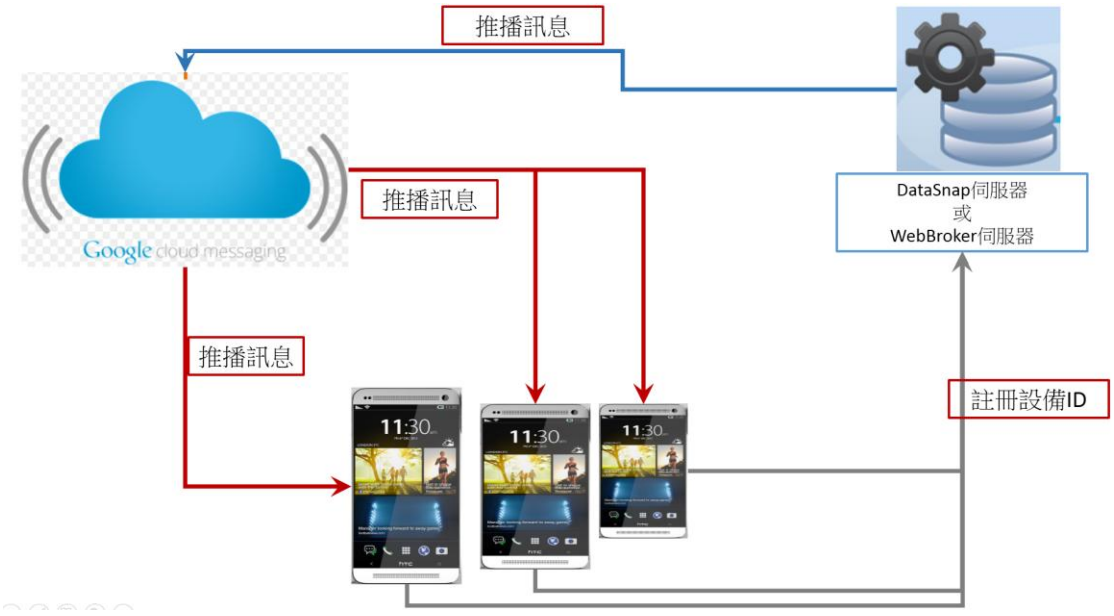


12-3 使用 Google GCM 实作推播功能

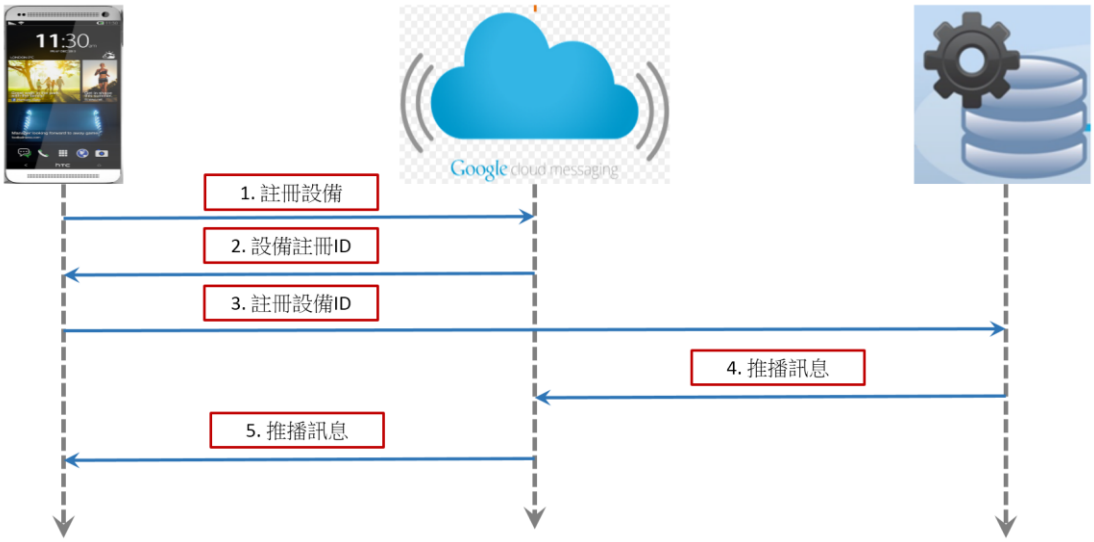
在最后一个小节中让我们使用一个更复杂范例，那就是如何使用 C++Builder 实作推播的功能，这个范例不但需要呼叫 Java 程序代码，也需要从 Java 接收回传通知。但在说明如何开发之前让我们简单的介绍一下 GCM 的架构。

GCM 是 Google Cloud Messaging 的简写，它是 Google 提供的免费服务可主动推播讯息给 Android 设备。在 GCM 架构中是由 3 个对象组成的，第 1 即是 GCM 本身，第 2 是客户端的 Android 设备以及一个伺机器。在 C++Builder 技术领域中我们可以使用下图来说明 GCM 的架构连作。

在 C++Builder 中我们可以使用 DataSnap 或是 WenBroker 技术撰写一个服务器，这个服务器可使用 HTTP 推播讯息给 GCM，GCM 就会把此讯息自动推播给已经注册要接收推播讯息的 Android 客户端。至于下图中的 Android 客户端就是使用 C++Builder 撰写的 App，它要执行工作最多，首先它需要向 GCM 注册表明要接收推播讯息，接着它也要向 DataSnap/WenBroker 服务器注册以便服务器将来指定要推播讯息给那一个 Android 客户端。



下图是这 3 者之间基本的互动流程图，一开始 Android 客户端 App 须要先向 GCM 注册以取得注册 ID，再把此注册 ID 向服务器注册。接着服务器向 GCM 推播讯息，在此推播讯息动作中服务器可指明推播给那一个 Android 客户端，那一群 Android 客户端或是所有的 Android 客户端。



了解了 GCM 架构的基本原理后我们就可以开始进行开发的工作了，要完成完整的 GCM 开发我们需要完成 3 个工作，它们是

1. 启动使用 GCM 功能
2. 开发 GCM 客户端 App
3. 开发 DataSnap 服务器

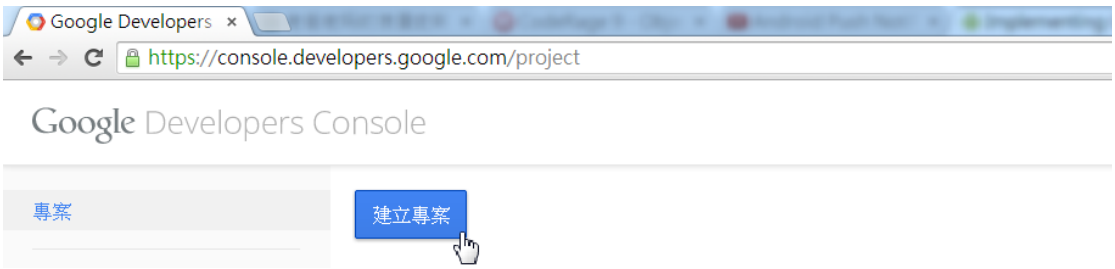
在下面的小节中将分别说明如何完成每一个步骤。

12-3-1 启动使用 GCM 功能

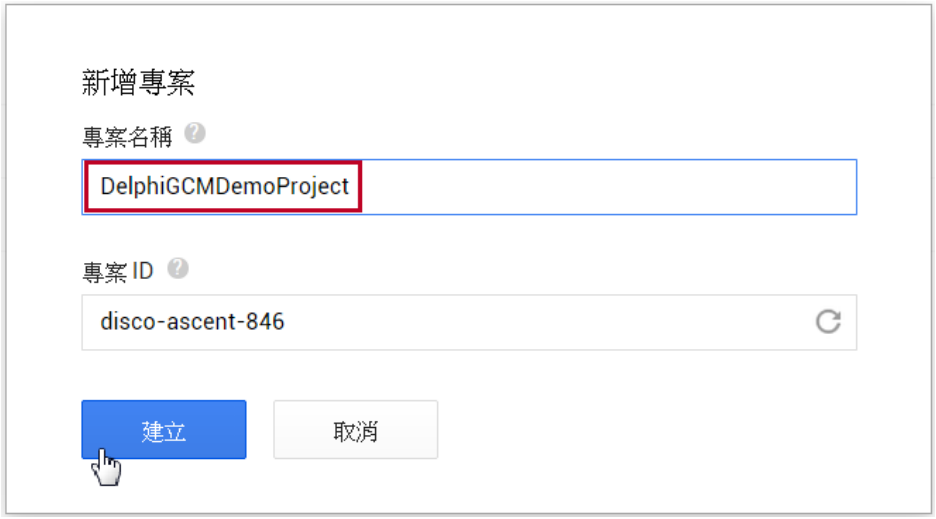
在使用 GCMi 之前您必须先到

```
https://console.developers.google.com
```

建立项目并开启 GCM 的功能，请点选下图中的”建立项目”按钮：



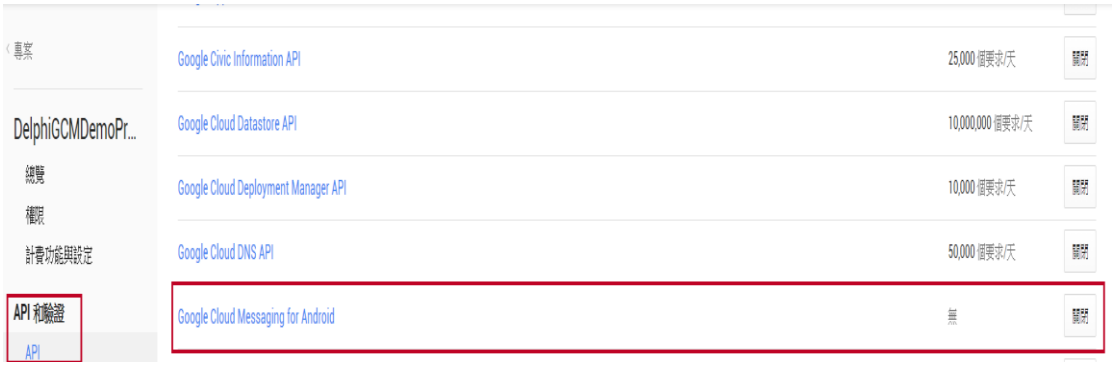
取一个项目名称，例如”C++BuilderGCMDemoProject”，并点选”建立”按钮：



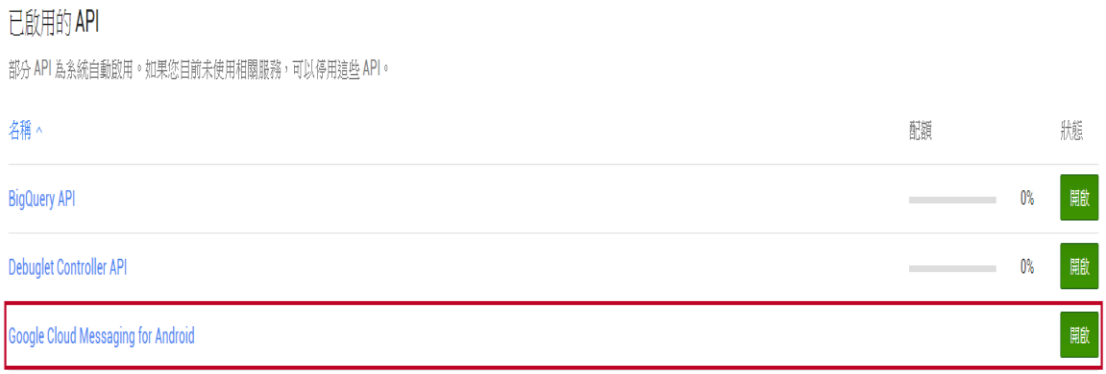
成功建立项目之后在”总览”页面可看到一个项目编号，此编号在稍后的开发中非常的重要请保存下来：

接下来我们要开启 GCM 的功能，请到 API 和验证下的 API 子项在右方找到如下图的 Google Cloud Messaging for Android 并点选右方的”关闭”按钮以开启使用这项服务：

开启

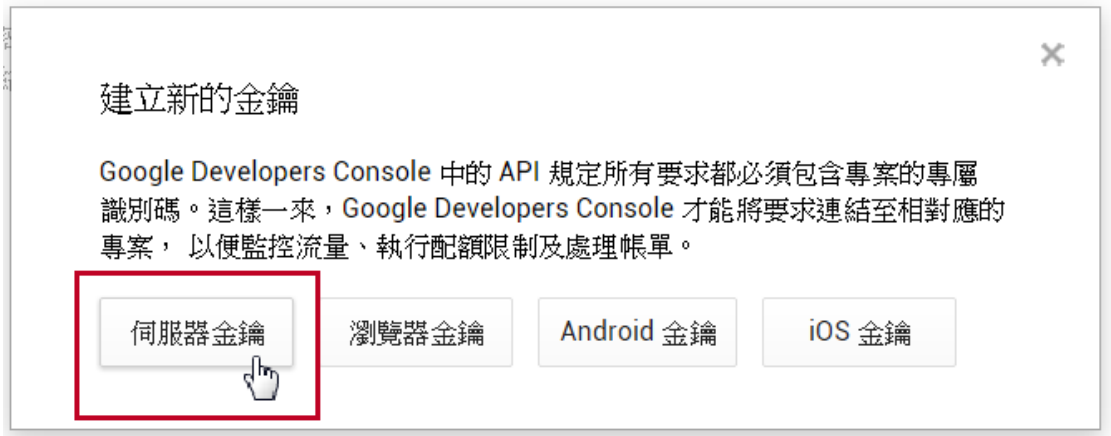


开启之后在此页面最前面的已启用的 API 中应该就可以看到如下图的”开启”状态了：



接着我们需要公开这个 API 让 Android 的客户端设备可以使用，请到 API 和验证下的凭证子项中点选”建立新的密钥”：

再点选”服务器密钥”：



接着会出现下图的画面要求您输入稍后执行 DataSnap 服务器的硬件 IP 以便 Google 控制流量和进行计费的计算(GCM 在 1000 人以内是免费的)。由于笔者是在个人 PC 中实作这个范例因此可以暂时不输入 IP：

最后点选上面”建立”按钮，就会出现下面重要的信息页面，其中的 API 密钥是稍后开发必要的信息，请和前面的项目编号一样把它保存下来。

到了这里我们已经完成了申请，开启和设定 GCM 服务的工作了，我们可以开始进行实际的开发工作了，但在说明之前请再次确定您已经有了：

- 专案编号
- API 密钥

12-3-2　开发 GCM 客户端 App

在开发 GCM 架构中 GCM 客户端是比较复杂的，因为 GCM 客户端需要完成下列的数项工作：

1. 向 GCM 注册客户端设备并取得设备注册 ID
2. 向服务器注册从 GCM 取得的设备注册 ID
3. 接收来自 GCM 的通知讯息

其中第 1 项工作需要呼叫 GCM API，这在前面的章节中已经说明如何呼叫 Java 程序代码所以不困难。第 1 项工作需要实作一个 DataSnap 服务器并呼叫它的服务方法，这也不困难。最困难的是第 3 项工作，因为 GCM 会推播讯息通知给 Java 的回叫函式，我们必须把这个回叫的执行导引到我们的 C++Builder 程序代码中。但是记得 C++Builder 是一个 RAD 工具，因此只要我们了解了 GCM 的工作原理之后再想通如何在 C++Builder 中根据这些原理来开发，那么就很简单了，而且简单到您会很吃惊。

在下面的小节中将一一说明如何完成这上面列出的开发步骤。

建立 Multi-Device 项目

让我们从简单的地方开始，先开发一个能自我接受 GCM 讯息的 App，成功之后再加入 DataSnap 服务器提供所有客户端设备的功能，最后再开发一个 VCL 客户端能够推播讯息到客户端设备。

在 C++Builder IDE 中先建立一个 Multi-Device 项目，在主窗体中加入 ToolBar，2 个 TButton 和一个 TListView 组件如下所示：

接着我们就要实作步骤 1 向 GCM 注册客户端设备并取得设备注册 ID，一般来说要完成步骤 1，开发人员需要完成下面 2 项工作：

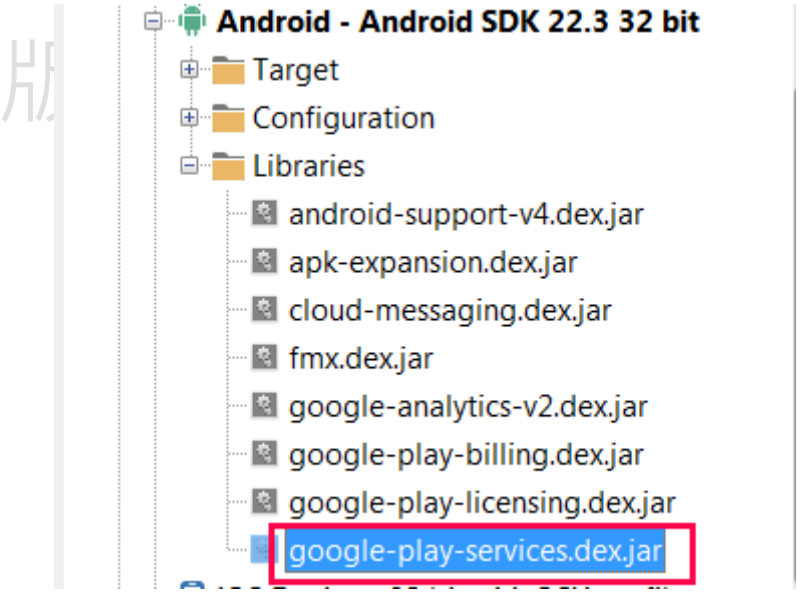
1. 查询是否有 Google Play Service

2. 有的话就可以向 GCM 注册

要查询是否有查询是否有 Google Play Service，我们可以使用 C++Builder 中的 TJGooglePlayServicesUtil 类别，它的 JGooglePlayServicesUtilClass 接口中的 isGooglePlayServicesAvailable 类别方法可提供查询：

```
virtual int __cdecl
isGooglePlayServicesAvailable(Androidapi::Jni::Graphicscontentvi
ewtext::_di_JContext context) = 0 ;
```

如果 isGooglePlayServicesAvailable 回传 true，那应就可以开始注册的工作，在这应该一定回传 true，因为项目的 Libraries 中已经包含了提供 Google Play Service 的 jar 档案了：

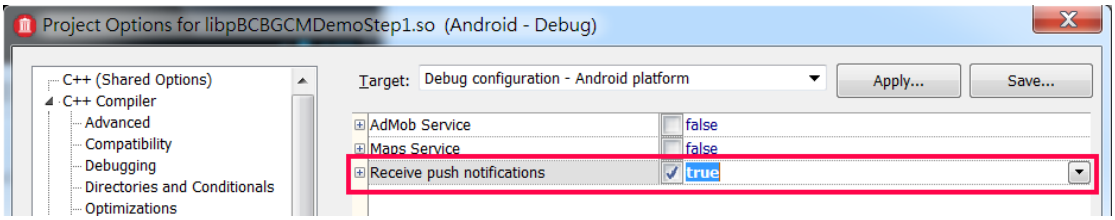


要向 GCM 注册我们可以使用 TJGoogleCloudMessaging 类别中的 register 方法，它接受一个 Androidapi::Jnibridge::TJavaObjectArray__1<Androidapi::Jni::Javatypes::_di_JString>型态的参数：

```
 __interface
INTERFACE_UUID("{A4F0B5B0-FCB2-45F8-A3CB-D37481F2FBD8}")
```

```
JGoogleCloudMessaging  : public
Androidapi::Jni::Javatypes::JObject
{
  virtual void __cdecl close(void) = 0 ;
  virtual Androidapi::Jni::Javatypes::_di_JString __cdecl
getMessageType(Androidapi::Jni::Graphicscontentviewtext::_di_JIn
tent intent) = 0 ;
  virtual Androidapi::Jni::Javatypes::_di_JString __cdecl
Register(Androidapi::Jnibridge::TJavaObjectArray__1<Androidapi::
Jni::Javatypes::_di_JString> * senderIds) = 0 ;
  virtual void __cdecl
send(Androidapi::Jni::Javatypes::_di_JString to_,
Androidapi::Jni::Javatypes::_di_JString msgId,
Androidapi::Jni::Os::_di_JBundle data) = 0 /* overload */;
  virtual void __cdecl
send(Androidapi::Jni::Javatypes::_di_JString to_,
Androidapi::Jni::Javatypes::_di_JString msgId, __int64 timeToLive,
Androidapi::Jni::Os::_di_JBundle data) = 0 /* overload */;
  virtual void __cdecl unregister(void) = 0 ;
};
```


很复杂吗？别担心，马上告诉您非常简单的方法来完成这个步骤，在解释之前我们需要先开启此项目能接收 GCM 讯息的能力，请点选项目，点选鼠标加键选择 Options...选项在 Entitlement List 子项中勾选 Receive push notifications，如下所示：

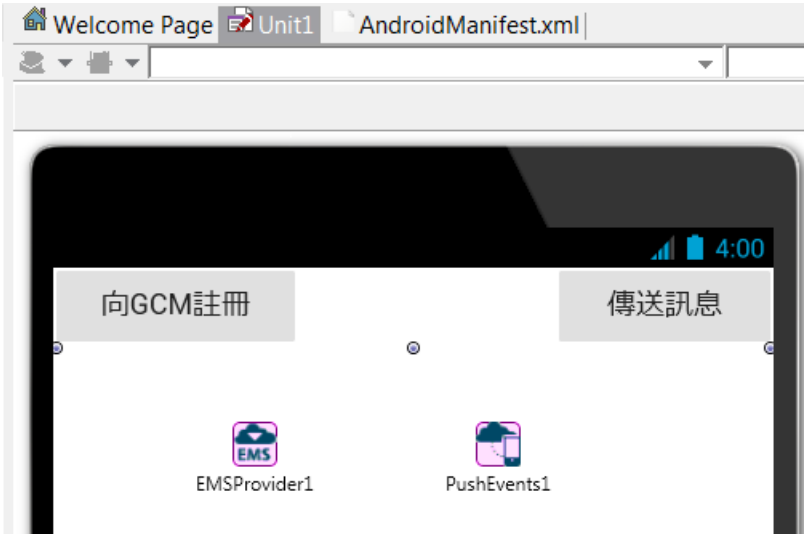


如此一来 C++Builder 就会在项目的 AndroidManifest.xml 文件中加入正确的权限和接收 GCM 讯息的接收器了。

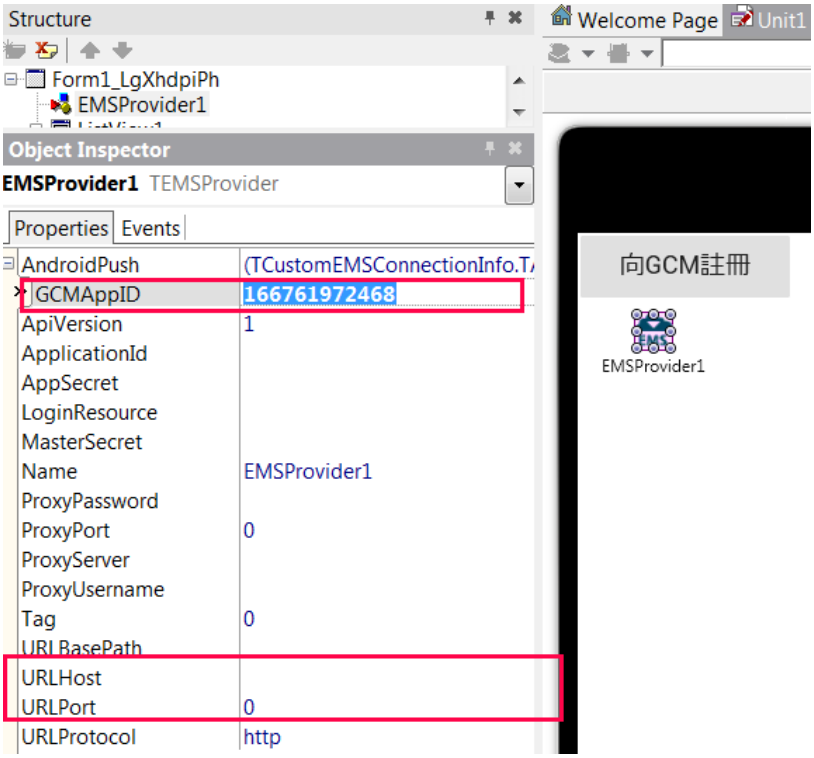
好了，那么要如何前面的工作呢？非常简单我们只需要转个脑筋藉由使用中的 BAAS Client 组件就可以了。在 BAAS Client 组件组中有 TEMSProvider 和 TPushEvents 这 2 个组件，虽然 C++Builder 的英文手册中说明 TEMSProvider 组件是链接到 EMS 服务器使用的，而 TPushEvents 则是需要使用 Parse 或 Kinvey 等第 3 方供货商才能使用的，但不用管手册说什么，因为

只需要转个脑筋把TEMSProvider组件想成是链接到Google，把TPushEvents想成是连结到 GCM 不就可以了吗？

所以请在主窗体中加入这 2 个组件：

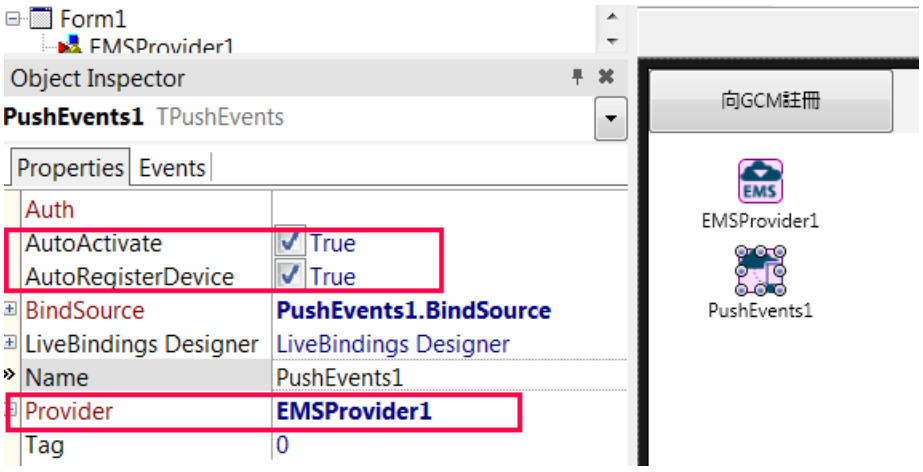


接下来要设定这 2 个组件，首先设定 TEMSProvider 组件。设定TEMSProvider 组件的关键点是完全忽略 TEMSProvider 组件链接 EMS 服务器的相关设定，我们只需要设定它的 AndroidPush 特性下的 GCMAppID 子特性，GCMAppID 的设定值就是我们前面说明的 Google 项目 ID 值，至于其他的特性，例如 MasterScret，AppSecret 等特性请都留成空白。这样的设定等于让 TEMSProvider 组件链接链接到 Google 的 GCM 服务服务器而不是 EMS 服务器，如下图所示：



接着对 TPushEvents 组件进行如下的设定：

特性	设定值
AutoActivate	True
AutoRegisterDevice	True
Provider	EMSProvider1



TPushEvents 组件的 AutoRegisterDevice 特性就可以向 Google GCM 自动注册客户端移动设备并取得设备 ID 和 GCM 注册 ID。这是因为 TPushEvents 组件是从 TCustomPushEvents 类别继承下来，在 TCustomPushEvents 类

别中有一个关键的特性 PushConnection，它是 TPushServiceConnection 类别对象：

```
__property System::Pushnotification::TPushServiceConnection*
PushConnection = {read=GetPushServiceConnection};
```

另外还有下面 3 个关键特性：

```
__property bool DeviceRegistered = {read=GetDeviceRegistered,
nodefault};
__property System::UnicodeString DeviceToken =
{read=GetDeviceToken};
__property System::UnicodeString DeviceID = {read=GetDeviceID};
```

上面的 DeviceToken 特性值即是客户端移动设备的 GCM 注册 ID，而 DeviceID 特性值则是客户端移动设备 ID。所以一旦 AutoRegisterDevice 特性值设定为 True 之后，那么在程序代码中就可以使用下面的方式取得客户端移动设备的 GCM 注册 ID：

```
if (PushEvents->DeviceRegistered)
  sGCMID = PushEvents->DeviceToken;
```


在 PushConnection 中有一 Service 特性它是 TPushService 类别对象：

```
__property TPushService* Service = {read=FService};
```

当我们用 TEMSProvider 连结 Google GCM 时，这个 Service 其实会是 TGCMPushService 对象：

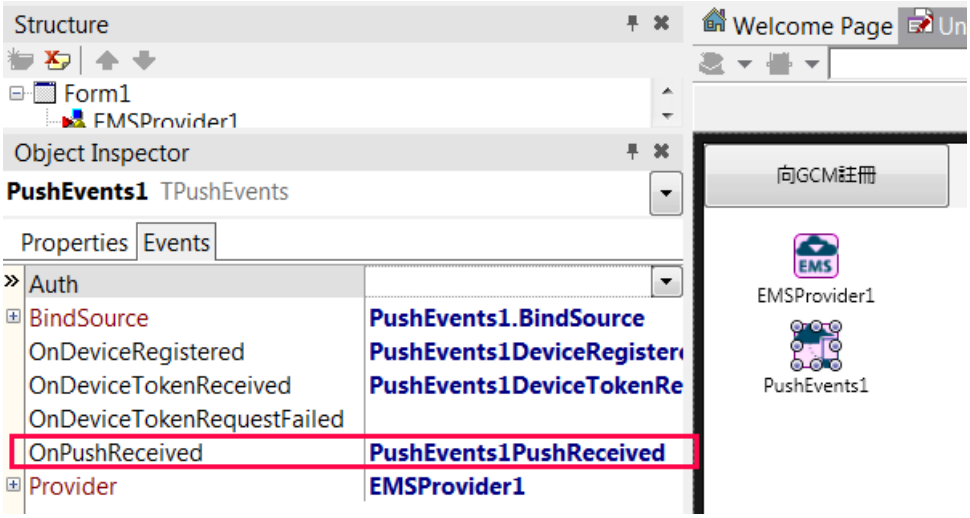
```
  TGCMPushService = class(TPushService)
```

在 TGCMPushService 类别中的 Register 方法就会自动帮助我们呼叫前面说明的使用 JGoogleCloudMessaging 等接口方法向 GCM 注册(以下为 C++Builder 的实作码)：

```
procedure TGCMPushService.Register(const LGCMAppId: string);
var
  LGCM: JGoogleCloudMessaging;
  LSenderIds: TJavaObjectArray<JString>;
  LToken: JString;
begin
  /// prepare sender-ids (this is usually your app-id)
```

```
  LSenderIds := TJavaObjectArray<JString>.Create(1);
  LSenderIds.SetRawItem(0, (StringToJString(LGCMAppId) as
ILocalObject).GetObjectID);

  FDeviceToken := '';
  LGCM :=
TJGoogleCloudMessaging.JavaClass.getInstance(SharedActivityConte
xt);
  try
    LToken := LGCM.Register(LSenderIds);
    FDeviceToken := (JStringToString(LToken));
    FStatus := TPushService.TStatus.Started;
    GCMPushService.DoChange([TPushService.TChange.Status,
TPushService.TChange.DeviceToken]);
  except
    on E: Exception do
    begin
      FStatus := TPushService.TStatus.StartupError;
      GCMPushService.FStartupError := E.Message;
      GCMPushService.DoChange([TPushService.TChange.Status]);
    end;
  end;
end;
```


如何？很棒吧，只要脑筋转一下就可以叫 C++Builder 自动帮忙我们完成 15-3-2 小节中的第 1，2 个步骤啦。最后的第 3 个步骤更简单完全不需要再使用 Java 程序代码，只要使用 TPushEvents 组件的 OnPushReceived 事件即可：



之后 C++Builder 提供了一个方法可直接把 GCM 传来的讯息串接到 TPushEvents 组件的 OnPushReceived 事件，那就是 GCMIntentService。因此要让 OnPushReceived 事件接收 GCM 传来的讯息，请开启您的项目目录下的 AndroidManifestTemplate.xml 档案(如果没有的话就请先 Build 一下你的项目即会自动产生)，在它的 application 元素中加入如下的子元素：

```
  <service
android:name="com.embarcadero.gcm.notifications.GCMIntentService
" />
```


例如下图就是笔者在 AndroidManifestTemplate.xml 档案中加入定的画面：

```
<%activity%>
<service android:name="com.embarcadero.gcm.notifications.GCMIntentService" />
<receiver android:name="com.embarcadero.firemonkey.notifications.FMXNotificationAlarm" />
<%receivers%>
```

加入了 GCMIntentService 并储存 AndroidManifestTemplate.xml 档案后就可以在 OnPushReceived 事件撰写接收 GCM 讯息的程序代码。OnPushReceived 事件会收到 TPushData 对象参数，其中的 GCM 特性就是 Google GCM 传来的讯息：

```
void __fastcall TForm3::PushEvents1PushReceived(TObject *Sender,
TPushData * const AData)
```

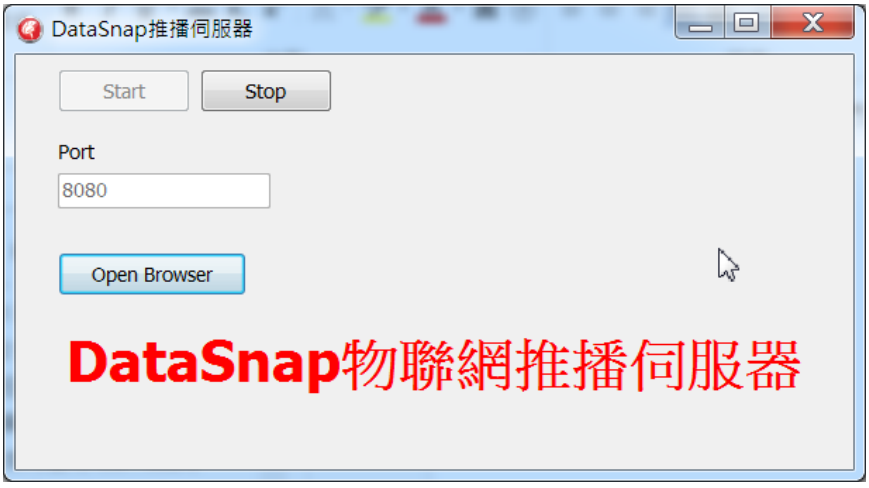
```
{
  ListView1->Items->Add()->Text = AData->GCM->Title;
  ListView1->Items->Add()->Text = AData->GCM->Msg;
  ListView1->Items->Add()->Text = AData->GCM->Message;
}
```

接着我们在主窗体的”向 GCM 注册”按钮中存取设备 ID 和 GCM 注册 ID：

```
void __fastcall TForm3::Button1Click(TObject *Sender)
{
  ListView1->Items->Add()->Text = PushEvents1->DeviceID;
  ListView1->Items->Add()->Text = PushEvents1->DeviceToken;
}
```

现在在手机中执行此范例 App 并点选”向 GCM 注册”按钮就可以看到如下的结果画面，我们已经可以成功取得这 2 个 ID 了：

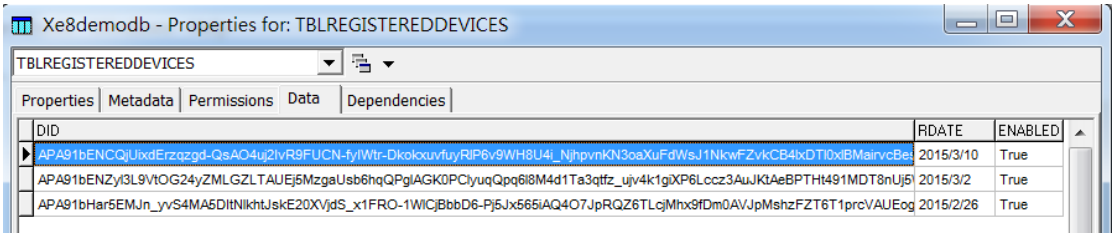
接着我们要测试 TPushEvents 组件的 OnPushReceived 事件是否可真的接收 GCM 讯息，在这里让我们使用一个已经写好的 DataSnap 推播服务器：



在这个 DataSnap 推播服务器中提供了一个方法可以让客户端的 App 向它注册客户端手机的 ID，

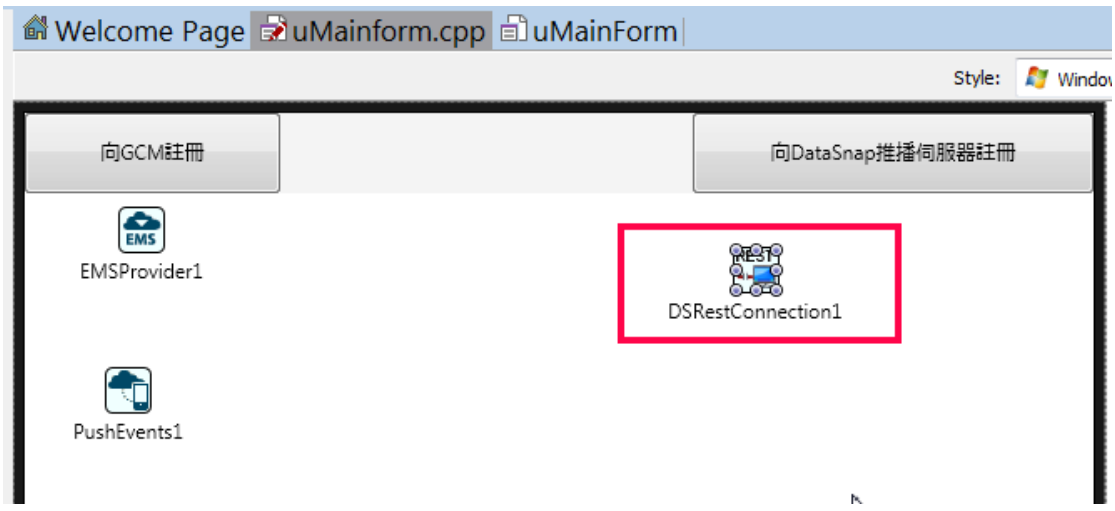
```
bool __fastcall RegisterdeviceID(System::UnicodeString sID, const
String& ARequestFilter = String());
```

一旦客户端的 App 呼叫 RegisterdeviceID 方法之后，DataSnap 推播服务器就会在一个名为 TBLREGISTEREDDEVICES 数据表中注册此客户端手机的 ID，之后 DataSnap 推播服务器就可以藉由这个 ID 向客户端手机的 ID 推播信息了。例如在下面的 TBLREGISTEREDDEVICES 数据表中目前注册了 3 个手机 ID，稍后我们实作完成并呼叫 RegisterdeviceID 方法后在 TBLREGISTEREDDEVICES 数据表就会出现新的 BCB 客户端 App 的 ID：

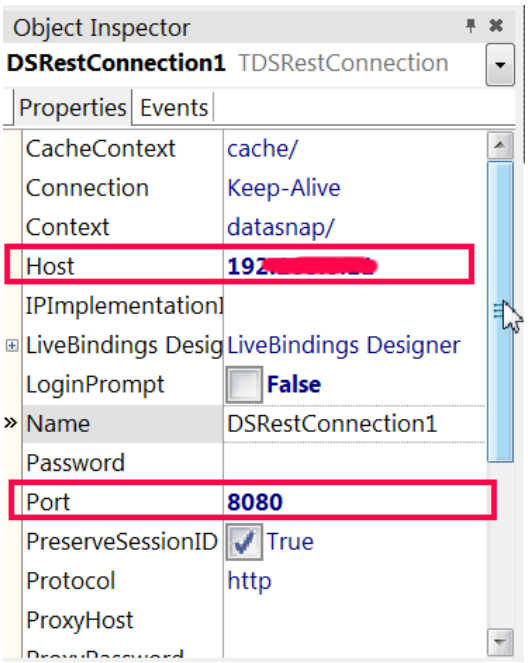


12-3-4 向 DataSnap 服务器注册设备 ID

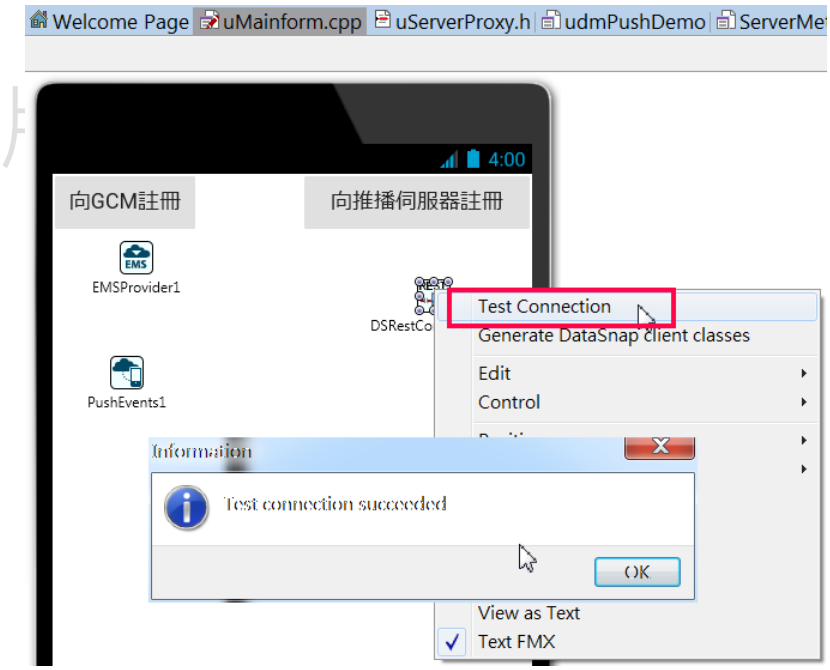
回到前面的范例 BCB App 准备加入向 DataSnap REST 推播服务器注册 GCM ID 的功能。首先要让范例 App 链接范例 DataSnap REST 服务器，请在主窗体中加入 TDSRestConnection 组件：



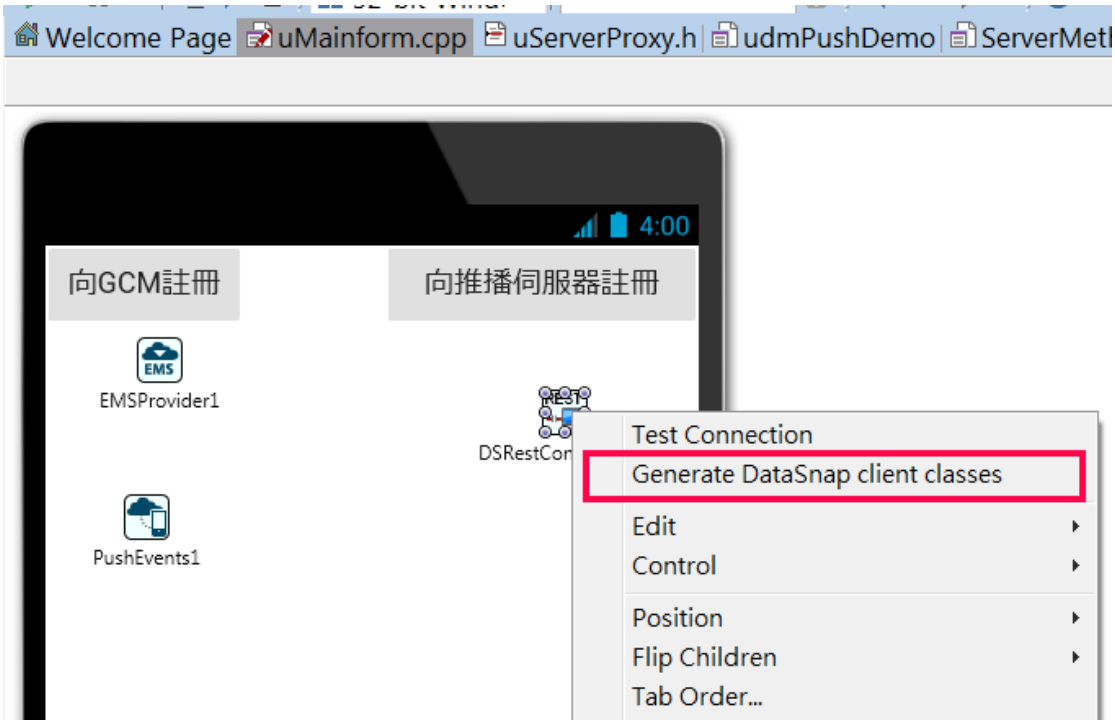
设定 DataSnap REST 服务器的 IP 地址以及使用的 Port 特性值 8080：



再使用鼠标右键测试是否成功连结到 DataSnap REST 推播服务器：



再使用鼠标右键产生客户端链接程序码：

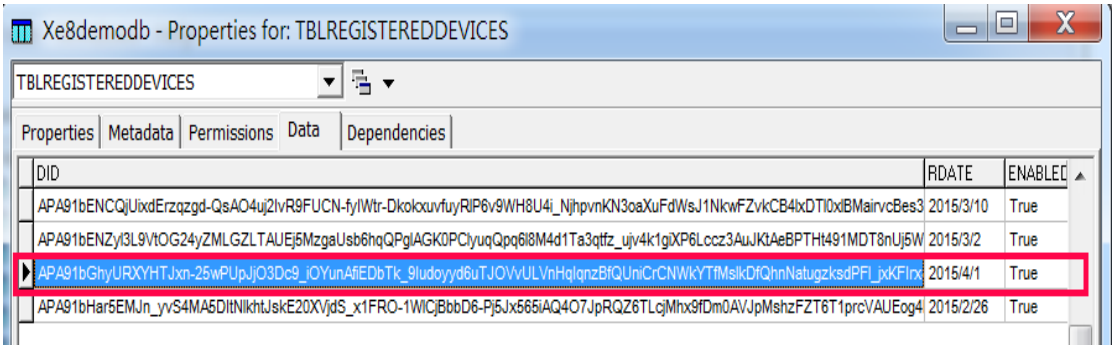


接着在”向推播服务器注册”按钮中使用客户端链接程序码呼叫 DataSnap REST 推播服务器提供的 RegisterdeviceID()方法注册：

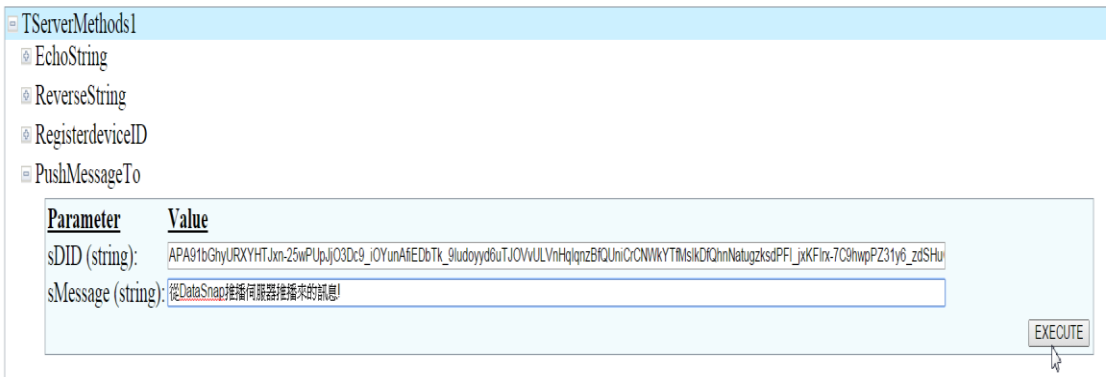

```
void __fastcall TForm3::Button2Click(TObject *Sender)
{
  TServerMethods1Client *pDSServer = new
TServerMethods1Client(DSRestConnection1);
  try
  {
  if (pDSServer->RegisterdeviceID(PushEvents1->DeviceToken))
    ListView1->Items->Add()->Text = L"向服务器注册成功";
  else
    ListView1->Items->Add()->Text = L"向服务器注册失败";
  }
  __finally
  {
  delete pDSServer;
  }
}
```

执行范例 App 点选” 向推播服务器注册”按钮就可以看到下面”注册成功”的讯息：

此时如果开启 TBLREGISTEREDDEVICES 范例数据表就可以看到客户端设备的 GCM ID 已经成功写入了：

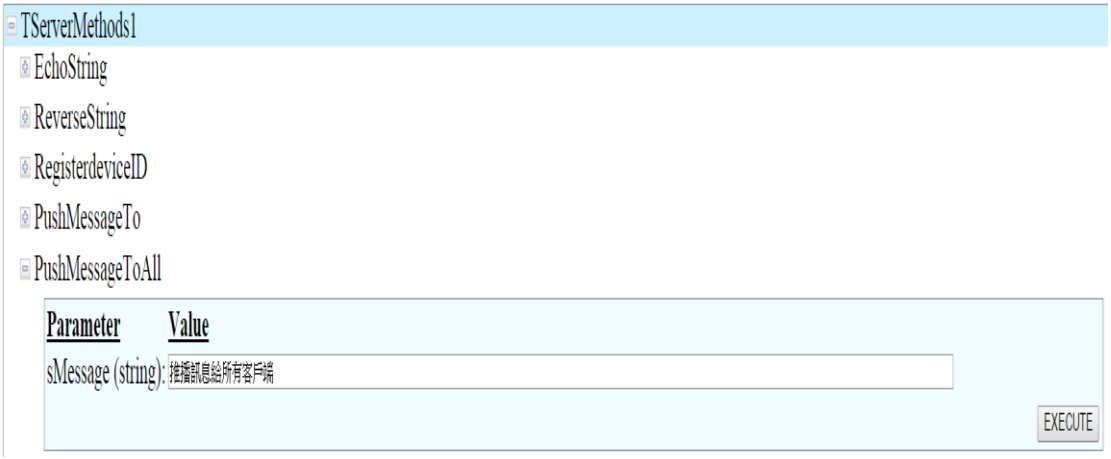


此时如果使用浏览器执行 DataSnap REST 推播服务器的 PushMessageTo 方法并使用范例 BCB App 的客户端 ID 值如下所示：



那就应该会在注册的客户端设备收到推播讯息，例如下面的画面就是笔者的 Samsung S4 收到范例 DataSnap REST 推播服务器藉由 GCM 推播来的讯息：

此时如果使用浏览器执行 PushMessageToAll 方法如下所示：



那就应该会在所有注册的客户端设备收到推播讯息，例如下面的 2 个画面就是笔者的 Samsung S4 和 HTC M8 收到范例 DataSnap REST 服务器藉由 GCM 推播来的讯息，一个 App 是使用 C++Builder 开发的，另一个是使用 C++Builder 开发的，但都能够藉由 GCM 收到 DataSnap REST 服务器推播来的讯息：


SAMSUNG
11:28 3月 2日, 週一
Wi-Fi
位置
聲音
畫面旋轉
藍牙
+1
自動
通知
清除
作為媒體裝置連接
輕觸 USB 選項。
無 SIM 卡
請插入 SIM 卡
Mobizen
11:27
遠端連接WIN-ROTR0B6E9D9和WiFi
pDelphiGCMStep1
11:25
推播訊息給所有客戶端 -傳遞時間 : 15-03-02 ..
僅限緊急通話

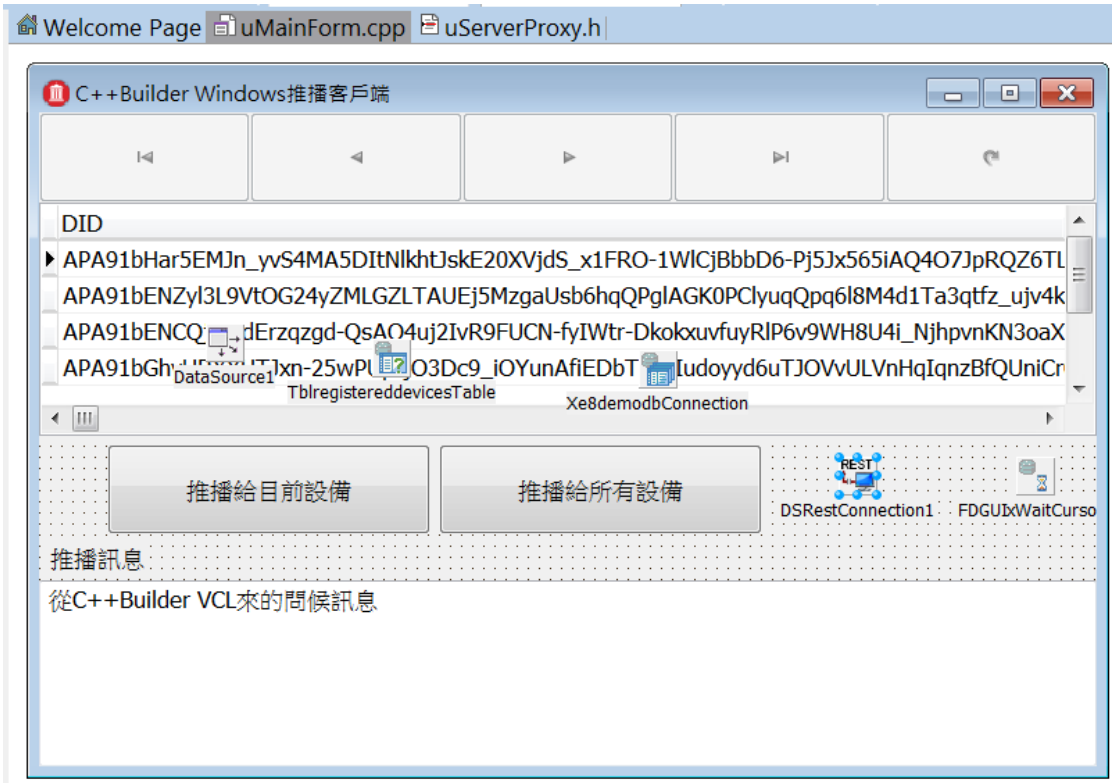

上午11:25
向GCM註冊
向伺服器註冊
傳送訊息
12EBDB10677ABD41CF9D2427AE3BA168
APA91bENZyl3L9VtOG24yZMLGZLTAUEj5MzgaUsb6hqQPglAG...
註冊成功
推播訊息給所有客戶端 -傳遞時間 : 15-03-02 11:23:01

12-3-5 开发 Windows 客户端

要开发一个可推播讯息的 Windows 客户端也很简单，我们只要使用前面讨论的技术，仍然藉由 TDSRestConnection 组件呼叫范例 DataSnap 推播服务器的服务方法即可完成。在 DataSnap 推播服务器中的 PushMessageTo 方法可把讯息推播给特定的客户端，而 PushMessageToAll 方法则可把讯息推播给所有的客户端：

```
bool __fastcall PushMessageTo(System::UnicodeString sDID,
System::UnicodeString sMessage, const String& ARequestFilter =
String());
bool __fastcall PushMessageToAll(System::UnicodeString sMessage,
const String& ARequestFilter = String());
```

例如下面的 2 个画面就是范例 Windows 客户端推播讯息给手机的结果：



在”推播给目前设备”按钮的 OnClick 事件中呼叫 PushMessageTo 方法推播讯息给特定的客户端手机：

```
void __fastcall TfmMainForm::btnPushMessageClick(TObject *Sender)
{
  TServerMethods1Client *pDSServer = new
```

```
TServerMethods1Client(DSRestConnection1);
  try
  {
  pDSServer->PushMessageTo(GetCurrentID(),
mmMessages->Lines->Text);
  }
  __finally
  {
  delete pDSServer;
  }
}
//---------------------------------------------------------------

String TfmMainForm::GetCurrentID()
{
  return TblregistereddevicesTable->FieldByName("DID")->AsString;
}
```

在”推播给目前设备”按钮的 OnClick 事件中呼叫 PushMessageToAll 方法推播讯息给所有的客户端手机：


```
void __fastcall TfmMainForm::btnPushMessageToAllClick(TObject
*Sender)
{
  TServerMethods1Client *pDSServer = new
TServerMethods1Client(DSRestConnection1);
  try
  {
  pDSServer->PushMessageToAll(mmMessages->Lines->Text);
  }
  __finally
  {
  delete pDSServer;
  }
}
```

从下面的执行结果画面可看到 C++Builder 的 VCL Windows 应用程序果然可藉由 DataSnap 推播服务器推播讯息给 Android 的手机：

13 物联网开发

物联网(IoT, Internet of Things)技术正如火如荼的席卷信息系统的开发，特别在各种穿戴式设备于 2015 年逐渐成熟和成为主流客户端设备之后，把所有设备都连结起来提供更方便，更丰富，更全面和更及时的信息就成为现今信息系统最重要的目前，而 C++Builder 从版就开始提供强大的物联网开发功能。

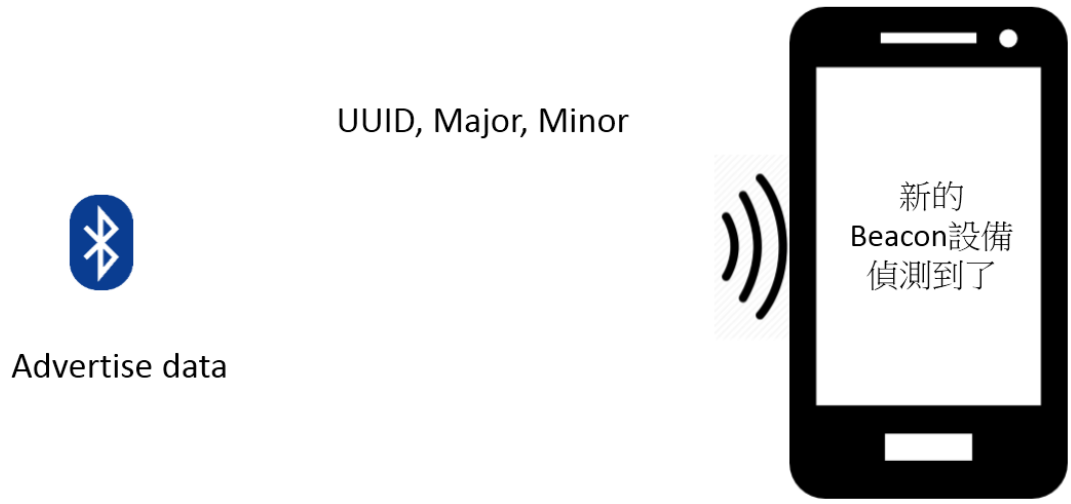
由于物联网包含的设备众多，在本小节中我们将讨论如何使用 C++Builder 开发可使用 Beacon 设备的物联网架构。

13-1 什么是 Beancon 技术

Beacon 是一个可提供制造商特定数据的低功耗蓝牙技术的设备，当 App 进入特定 Beacon 设备的范围入之后就可以进行连结，当 App 进入到 Beacon 有效的范围内，Beacon 就会发送一串识别信息给 App， App 侦测到识别信息后便会触发一连串的动作，例如是从云端查询/下载信息，也可能是开启其他 App 或连动装置。目前在市面上已可购买到 Beacon 设备，例如下图就是一些 Beacon 设备：

一般来说 Beacon 则可将定位范围精准到 2～100 公尺内，但 Beacon 会受到实体对象的影响，例如墙面，桌子，皮包等等的阻隔而影响信号的强度，有效距离和精确度。

当 App 进入特定 Beacon 设备的信号范围内时，App 可以取得 Beacon 设备的识别信息，例如 UUID，Beacon 设备的 Major ID 和 Minor ID，如下图所示：



因此 App 可以精确的掌握位置和此 Beacon 设备的详细信息，接着 App 便可以根据这些信息链接到云端或是远程服务器查询数据，或是由端服务器主动推播数据给 App。

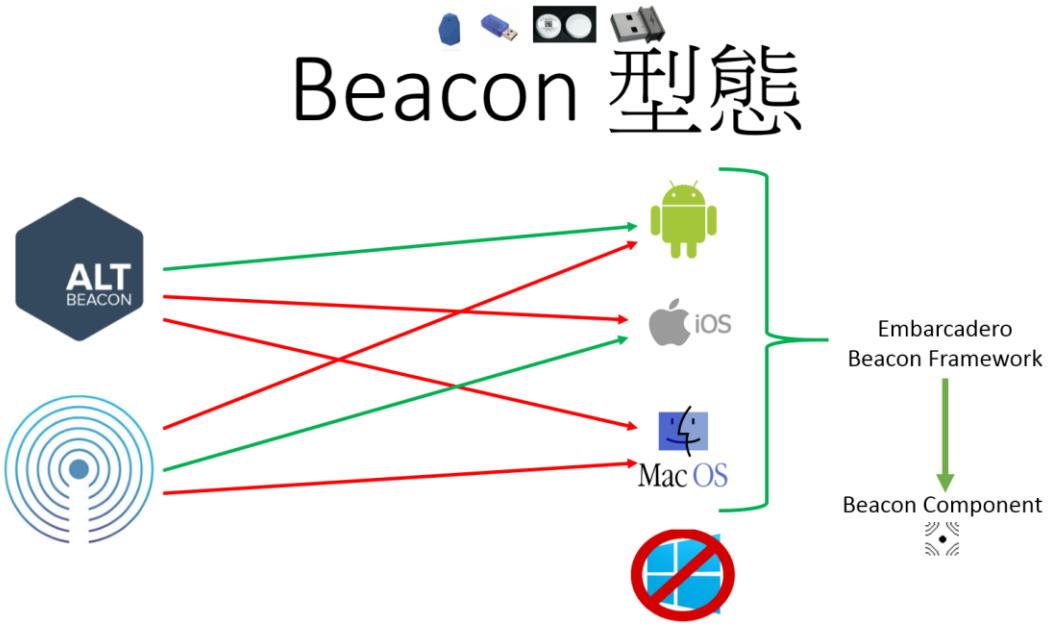
13-2 Beancon 种类

目前 Beacon 技术分为 2 大类，但彼此非常的相像只有一点点的不同，它们是：

种类	说明
iBeacon	使用 Apple 格式的设备，使用 iBeacon 格式的设备必须先向 Apple 注册
AltBeacon	开放的格式，它的规格公开在 http://altbeacon.org/

C++Builder 在版开始同时支持 iBeacon 和 AltBeacon。

由于 Beacon 技术是根基于低功耗蓝牙技术，因此不是每一个平台都可支持 iBeacon/AltBeacon，下面的图形说明了目前在中每一个平台对于支持 Beacon 技术的现况：



从上图可知由于 Windows 平合对于 iBeacon/AltBeacon 的限制，因此 Windows 平台尚不能开发 Beacon 的 App。

此外 C++Builder 虽然在 XE7 便可以支持低功耗蓝牙技术的开发，但由于 Apple 限制要开发 iBeacon App 必须使用 Core Location Services API 而不能用 BLE API，因此提供了新的 Beacon 框架和 Beacon 组件来帮助 C++Builder 开发人员开发 iBeacon/AltBeacon 的 App。

13-3 Beancon 数据格式和意义

每一个 Beacon 设备都需要提供如下的数据：

字段	说明
UUID	独特的 ID，代表唯一公司的 Beacon 设备
Major ID	代表唯一公司的 Beacon 设备中的特定 Beacon 群组
Minor ID	代表唯一公司的 Beacon 设备中的特定 Beacon 群组中的特定 Beacon 设备
TxPower	此常数值代表从一公尺接收 Beacon 设备的信号强度，TxPower 结合 RSSI(Received Signal Strength Indicator)后即可

C++Builder App 可在取得 UUID，Major ID 和 Minor ID 后扫瞄 TxPower 和 RSSI 值来决定距离 Beacon 设备有 3 远。

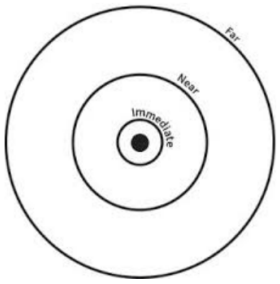
例如下面是 Beacon 设备可提供数据的范例，Beacon 设使用一个唯一的 UUID 来识别特定公司，用 Major ID 做为地区的识别，最后用 Minor ID 做为不同服装部门的识别：

百货地点		台北	台中	高雄
UUID		1D614A54-0149-4361-9BF1-42389A2AE58B		
Major		1	2	3
Minor	男装	10	10	10
	女装	20	20	20
	童装	30	30	30

因此当 C++Builder App 接近这些 Beacon 设备并取得 UUID，Major ID 和 Minor ID 后充就可以知道是在什么地方的什么部门，接着就可以再藉由例如 RESTful 技术连到远程服务器，例如 DataSnap，取得此部门的优惠活动信息。

在一个场合中多个 Beacon 设备可以形成一个所谓的 Region，当然也可以同时形成多个 Regions。例如我们可以 Major ID 做为 Region 的识别，而 Proximity 则代表 App 对于 Region 或是特定 Beacon 设备的距离：

Regions 和Proximity



RSSI
Received Signal Strength Indicator

下面的表格实际的列出了 iBeacon/AltBeacon 的数据格式：

长度	说明	iBeacon范例	AltBeacon范例
1 个字节	数据长度	1A	1B
1 个字节	FF(一定是 FF,代表是制造商资料)	FF	FF

2 个字节	公司 ID	4C 00 *APPLE INC	18 01 *Creative Technology Ltd.
2 个字节	BEACON 型态	02 15 *iBEACON	BE AC *Beac-on
16 个字节	UUID	2F 23 44 54 CF 6D 4A 0F AD F2 F4 91 1B A9 FF A6	2F 23 44 54 CF 6D 4A 0F AD F2 F4 91 1B A9 FF A6
2 个字节	Major ID – Beacon 群组	00 01	00 01
2 个字节	Minor ID – Beacon 单位	00 01	00 01
1 个字节	TX Power – Rssi	BE	BE
1 个字节	制造商保留数据(iBeacon 无此数据)	无	00

13-4 Beancon 接近状态

当 App 接近 Beacon 设备会根据 Proximity 的值来决定 App 和 Beacon 设备的距离，一般来说可分为 4 种状态：

接近状态	说明
Immediate	App 距离 Beacon 设备 0 到 0.5 公尺
Near	App 距离 Beacon 设备 0.5 到 1.5 公尺
Far	App 距离 Beacon 设备 1.5 公尺以外
Unkonw	未知距离(可能太遥远或是有东西阻挡讯号)

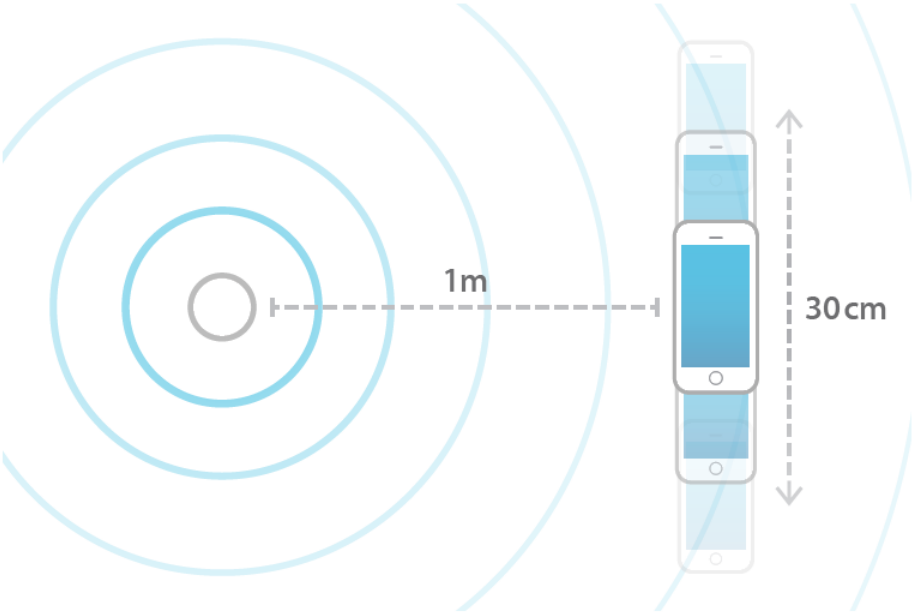
因此 App 可根据 Proximity 来决定要采取什么行动。

13-5 Beancon 设备校正

在开发 Beacon App 时我们需要对 Beancon 设备进行校正的工作以取得精确的 RSSI 数值好计算 App 距离 Beancon 设备有多远。要校正 Beancon 设备 Apple 建议使用下面的步骤：

1. 安装 Beancon 设备并让它开发发射讯号
2. 使用移动设备并执行 Bluetooth 4.0 radio 在距离 Beancon 设备 1 公尺左右花至少 10 秒时间测量信号强度
3. 以如下图的方式缓慢前后移动手机/平板等移动设备 30 分分
4. 取得 RSSI 值
5. 平均计算 RSSI 值

6. 在 Beancon 设备组态信息中记录此 RSSI 值



13-6 开发 Beacon App 和物联网应用架构

在对什么是 Beacon 以及对 Beacon 和 Beacon 设备有了基本的了解之后我们就可以开始使用 C++Builder 来开发物联网型态的 App 了。

在使用 C++Builder 开发 Beacon App 时基本上开发人员需要掌握 2 个基本能力：

- 自动侦测 Beacon 设备：第 1 种开发物联网应用的架构是自动侦测附近的Beacon 设备虽然再透过网络根据附近的Beacon 设备到云端或是到远程服务器查询数据或服务。

- 开发已知 Beacon 设备 App：第 2 种开发物联网应用的架构是一个公司或企业已经知道使用所有的 Beacon 设备，并只允许 App 侦测这些 Beacon 设备来定位并提供服务。

掌握了上面 2 项技术之后就可以再结合其他技术来开发连网应用架构了。在下面的小节中将一一说明如何完成上述的每一项开发工作。

13-6-1 自动侦测 Beacon 设备

如果要开发的物联网 App 事先不知道会使用什么 Beacon 设备，那么 App 就需要先自动侦测附近的 Beacon 设备，再对侦测到的 Beacon 设备进行扫瞄/监督等后续的处理。

由于 Beacon 设备就是一种低功耗蓝牙设备，因此在中我们可以使用 TBluetoothLEManager 类别来自动侦测 Beacon 设备，再把侦测到的 Beacon 设备注册给 TBeacon 组件进行扫瞄/监督等后续的处理就可以很简单的完成物联网 App 的开发工作。在本小节中我们将说明如何使用 TBluetoothLEManager 类别来自动侦测 Beacon 设备，于下一小节再说明如何使用 TBeacon 组件进一步的处理。

TBluetoothLEManager 类别提供了 2 个事件让开发人员可以侦测找到的低功耗蓝牙设备，其中 OnDiscoveryEnd 事件会在所有低功耗蓝牙设备侦测完毕后触发而 OnDiscoverLEDevice 事件则会在每一个低功耗蓝牙设备侦测到时触发：

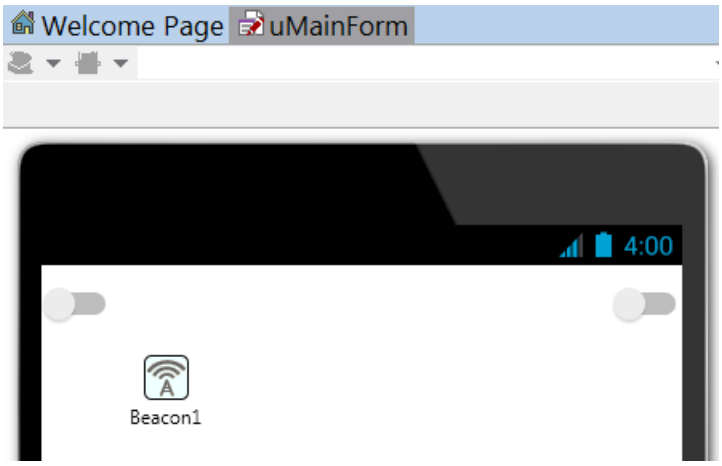
```
__property TDiscoveryLEEndEvent OnDiscoveryEnd =
{read=FOnDiscoveryLEEnd, write=FOnDiscoveryLEEnd};
__property TDiscoverLEDeviceEvent OnDiscoverLEDevice =
{read=FOnDiscoverLEDevice, write=FOnDiscoverLEDevice};
```


要侦测低功耗蓝牙设备，开发人员只需要呼叫 TBluetoothLEManager StartDiscovery 方法即可：

```
bool __fastcall StartDiscovery(unsigned Timeout,
TBluetoothUUIDsList* const AFilterUUIDList =
(TBluetoothUUIDsList*)(0x0));
```

因此要侦测 Beacon 设备我们只需要指定一个事件处理函式给 OnDiscoverLEDevice 特性，再呼叫 StartDiscovery()方法。

请在 IDE 中建立一个 Multi-Device 应用程序，在主窗体中加入 2 个 TSwitch 组件再加入一个 TBeacon 组件，如下所示：



主窗体中左上方的 TSwitch 组件目的是开始自动侦测 Beacon 设备，右上方的 TSwitch 组件则是开始对侦测到的 Beacon 设备主进行扫瞄/监督等工作。

因此在上方的 TSwitch 组件的 OnSwitch 事件中呼叫 DoAutoScanBeacons()方法开始自动侦测 Beacon 设备：

```
void __fastcall TfmMainForm::swtScanBeaconsSwitch(TObject *Sender)
{
  DoAutoScanBeacons();
}
```


接着在主窗体表头文件中宣告一个 TBluetoothLEManager 类别对象变量，以便使用它进行自动侦测 Beacon 设备：

```
  TBluetoothLEManager *FManager;
```

DoAutoScanBeacons()使用的方法就是前面我们说的，先使用 TBluetoothLEManager 类别取得一个目前的 TBluetoothLEManager 类别对象并指定给 FManager，再指定事件处理函式 DiscoverLEDevice()给它的 OnDiscoverLeDevice 触发事件，如此一来当 TBluetoothLEManager 每侦测到一个 Beacon 设备就会呼叫 DiscoverLEDevice()，最后呼叫 StartDiscovery()方法使用 10 秒的时间自动侦测 Beacon 设备：

```
void TfmMainForm::DoAutoScanBeacons()
{
  if (FManager == NULL)
  {
  FManager =  TBluetoothLEManager::Current;
  FManager->OnDiscoverLEDevice = DiscoverLEDevice;
```

```
  }
  FManager->StartDiscovery(10000);
}
```

DiscoverLEDevice()方法会在 TBluetoothLEManager 侦测到 Beacon 设备时被呼叫，它首先检查被侦测到的低功耗蓝牙设备是否包含制造商数据（$FF），如果是的话就应该是一个 Beacon 设备（请回头参考前面的表格说明，iBeacon/AltBeacon 的第 1 个数据一定是 FF）。在 System.Bluetooth.HPP 表头文件中定义了 TScanResponseKey 数据型态，其中的 ManufacturerSpecificData 定义值是 255，也就是 16 进位的 FF：

```
enum class DECLSPEC_DENUM TScanResponseKey : unsigned short { Flags
= 1, IncompleteList16SCUUID,… ManufacturerSpecificData = 255 };
```

因此 DiscoverLEDevice()方法先检查扫瞄到的低功耗蓝牙设备是否包含 FF，如果是的话就建立一个 MyThreadProcedure 对象再呼叫 Synchronize()方法来显示数据：

```
void __fastcall TfmMainForm::DiscoverLEDevice(System::TObject*
const Sender, TBluetoothLEDevice* const ADevice, int Rssi,
  TScanResponse* const ScanResponse)
{
  if
(ScanResponse->ContainsKey(TScanResponseKey::ManufacturerSpecifi
cData))
  {
      _di_TThreadProcedure mtp = new MyThreadProcedure(ADevice,
Rssi, ScanResponse);
      System::Classes::TThread::Synchronize(NULL, mtp);
  }
}
```


但为什么要建立一个 MyThreadProcedure 对象？为什么要呼叫 Synchronize()方法来显示数据呢？

这是因为在 C++Builder 中扫瞄低功耗蓝牙设备要使用额外独立的线程来进行，由于此独立的线程不是主线程，因此在立的线程扫瞄完成要显示数据时就需要呼叫 Synchronize()方法来处理同步的问题。

为了维护范例 App 中扫瞄到的低功耗蓝牙设备，让我们先定义一个 struct 数据结构来储存扫瞄到的低功耗蓝牙设备：

```
  struct TBeaconDevice{
    TBluetoothLEDevice* ADevice;
  TGUID GUID;
  Word Major;
  Word Minor;
  Integer TxPower;
  Integer Rssi;
  Double Distance;
  System::Boolean Alt;
  };
  typedef std::list<TBeaconDevice> TBeaconDeviceList;
```

接着定义 MyThreadProcedure 类别，请注意 MyThreadProcedure 类别是从 TCppInterfacedObject 类别继承下来：

```
class MyThreadProcedure : public
TCppInterfacedObject<TThreadProcedure>
{
public:
  MyThreadProcedure(TBluetoothLEDevice* const _ADevice, int _Rssi,
TScanResponse* const _ScanResponse);
  void __fastcall Invoke(void);
private:
  TBluetoothLEDevice* const ADevice;
  int Rssi;
  TScanResponse* const ScanResponse;
};
```


因此我们需要一个接口让 TCppInterfacedObject 封装，这个接口就是 TThreadProcedure，它定义了一个接口方法 Invoke()：

```
__interface TThreadProcedure  : public System::IInterface
{
  virtual void __fastcall Invoke(void) = 0 ;
};
```

接下来我们需要做的就是实作 Invoke()方法的工作就是解析扫瞄到的 Becoan 设定数据：

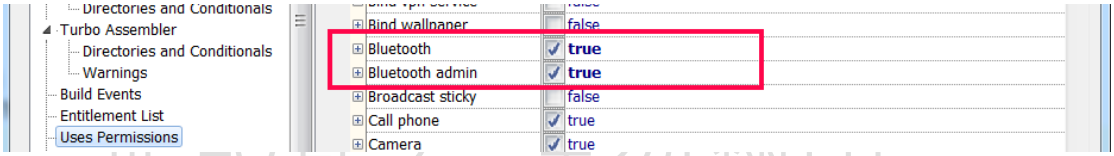
```
void __fastcall MyThreadProcedure::Invoke(void)
{
  TBeaconDevice LBeaconDevice =
fmMainForm->DecodeScanResponse(ScanResponse);
  LBeaconDevice.Rssi = Rssi;
  LBeaconDevice.Distance =
fmMainForm->FManager->RssiToDistance(Rssi, LBeaconDevice.TxPower,
0.5);
  LBeaconDevice.ADevice = ADevice;
  int NewBeacon = 0;
  bool BeaconFound = False;
  if (fmMainForm->BeaconDeviceList.size() > 0)
  {
      for (std::list<TBeaconDevice>::iterator
BD=fmMainForm->BeaconDeviceList.begin(); BD !=
fmMainForm->BeaconDeviceList.end(); ++BD)
      {
          NewBeacon++;
          if (IsEqualGUID(BD->GUID, LBeaconDevice.GUID) &&
(BD->Major == LBeaconDevice.Major) && (BD->Minor ==
LBeaconDevice.Minor))
          {
              *BD = LBeaconDevice;
              BeaconFound = True;
              break;
          }
       }
  }
  TVarRec v[] = { LBeaconDevice.Distance };
  if (!BeaconFound)
  {
      fmMainForm->BeaconDeviceList.push_back(LBeaconDevice);

  fmMainForm->lbBeacons->Items->Add("----------------------------
--");
```

```
      String BeaconName("Beacon Found: "+ ADevice->DeviceName);
      if (LBeaconDevice.Alt)
        BeaconName = BeaconName + "; AltB";
      else
        BeaconName = BeaconName + "; iB";
      fmMainForm->lbBeacons->Items->Add(BeaconName);

      BeaconName = "Device Complete name: ";
      if
(ScanResponse->ContainsKey(TScanResponseKey::CompleteLocalName))
      {
          TByteDynArray value;

  ScanResponse->TryGetValue(TScanResponseKey::CompleteLocalName,
value);
          BeaconName = BeaconName +
(TEncoding::UTF8->GetString(value));
      }
      else
        BeaconName = BeaconName + "No Name";

      fmMainForm->lbBeacons->Items->Add(BeaconName);
      fmMainForm->lbBeacons->Items->Add( " UUID: " +
GUIDToString(LBeaconDevice.GUID) );
      fmMainForm->lbBeacons->Items->Add( " Major:" +
IntToStr(LBeaconDevice.Major) +
                                    ",  Minor:" +
IntToStr(LBeaconDevice.Minor) +
                                    ",  txPower: " +
IntToStr(LBeaconDevice.TxPower) );
      fmMainForm->lbBeacons->Items->Add(" Rssi: " +
IntToStr(LBeaconDevice.Rssi) + Format(" Distance: %f m", v, 0) );
  }
  else
  {
      String BeaconName("Beacon Found: "+ ADevice->DeviceName);
      if (LBeaconDevice.Alt)
```

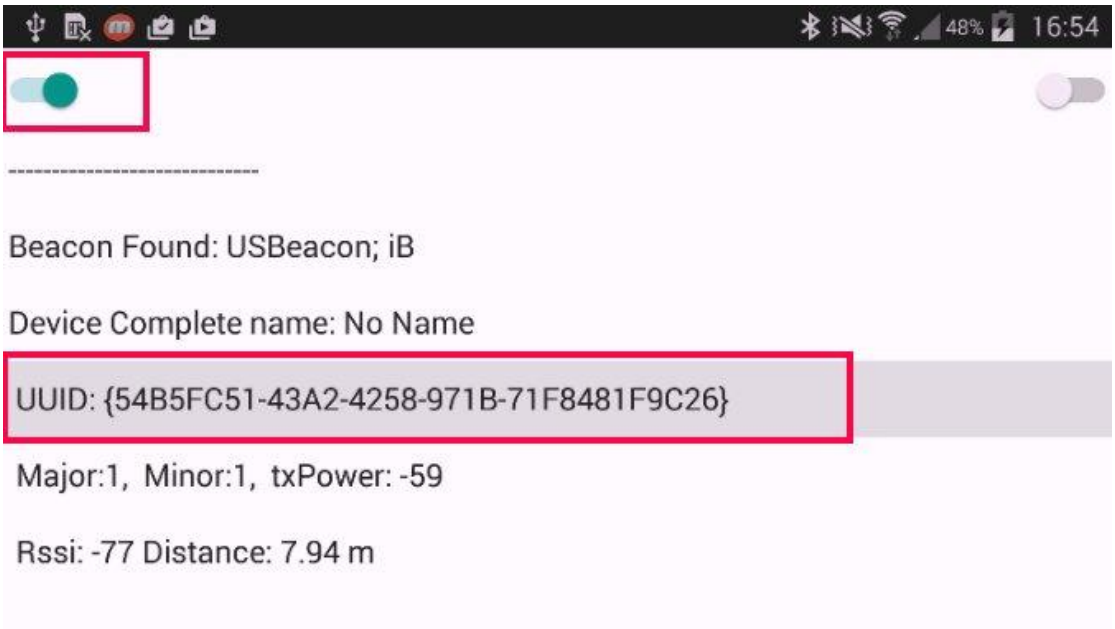
```
        BeaconName = BeaconName + "; AltB";
      else
        BeaconName = BeaconName + "; iB";
      fmMainForm->lbBeacons->Items->Strings[(NewBeacon-1)*6+1] =
(BeaconName);
      fmMainForm->lbBeacons->Items->Strings[(NewBeacon-1)*6+5] =
(" Rssi: " +  IntToStr(LBeaconDevice.Rssi) + Format(" Distance: %f
m", v, 0));
  }
}
```

由于 MyThreadProcedure 对象被封装在 Synchronize()方法中，因此它可以放心的更新主窗体中的 UI。

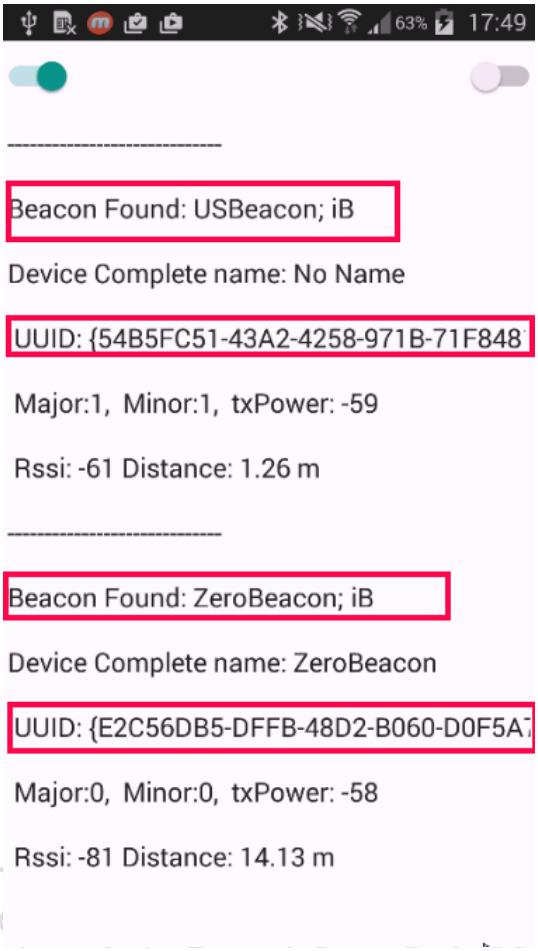
在执行范例 App 之前当然不要忘记开启蓝牙的存取权：



现在如果编译和执行范例 App 就可以看到如下的画面：



下图是此范例 App 扫瞄到多个 Beacon 设备的画面：



了解了如何自动侦测 Beacon 设备后，我们就可以进一步说明如何处理 App 和 Beacon 设备的互动了。

13-6-2 开发已知 Beacon 设备 App

对于企业应用而言，在布建物联网架构时使用的 Beacon 设备的 UUID 和 MajorID，MinorID 应该都已经确定了，因此不需要像上一小节要处理自动侦测 Beacon 设备的工作。

为了方便开发人员完成此开发的工作，在 System.Beacon.Hpp 程序单元中定义了 TBeaconManager 类别可使用进行 Beacon 的开发工作。下面说明了如何使用 TBeaconManager 类别对象(此程序代码是以 C++Builder 实作的)：

```
001    BeaconManager :=
TBeaconManager.GetBeaconManager(TBeaconScanMode.Standard); // 或
TBeaconScanMode.Alternative
002    try
003      //指定 Beacon Manager 事件处理函式
```

```
004     BeaconManager.OnBeaconEnter := BeaconEnter;
005     BeaconManager.OnBeaconExit := BeaconExit;
006     BeaconManager.OnEnterRegion := EnterRegion;
007     BeaconManager.OnExitRegion := ExitRegion;
008     BeaconManager.OnBeaconProximity := BeaconProximity;
009
010     //注册要监督的 Beacon 区域
011     BeaconManager.RegisterBeacon(TGUID.Create('
{D1E10B90-38A6-4BEB-99BB-DF5A6F51F7AE}'));
012     BeaconManager.RegisterBeacon(TGUID.Create('
{F490070D-7341-40EB-B28C-43CBDBFBAA17} '));
013
014     //开始扫瞄
015     BeaconManager.StartScan;
016
017     //在此时就会触发前面设定的事件
018
019     //停止扫瞄
020     BeaconManager.StopScan;
021
022   finally
023   //释放 TBeaconManager 对象
024     BeaconManager.Free;
025   end;
```


从上面的程序代码中我们可以知道，要使用 TBeaconManager 类别对象，开发人员要使用下面的步骤：

1. 根据是要扫瞄 iBeacon 或是 AltBeacon，取得相对的 BeaconManager 对象(001 行)
2. 设定不同 Beacon 设备的件处理函式(004~008 行)
3. 设定要扫瞄的 Beacon 区域(Beacon 设备的 UUID 值)(011~012 行)
4. 开始扫瞄 Beacon 区域中的 Beacon 设备(014 行)
5. 处理完成之后停止扫瞄(020 行)
6. 最后释放 TBeaconManager 对象(024 行)

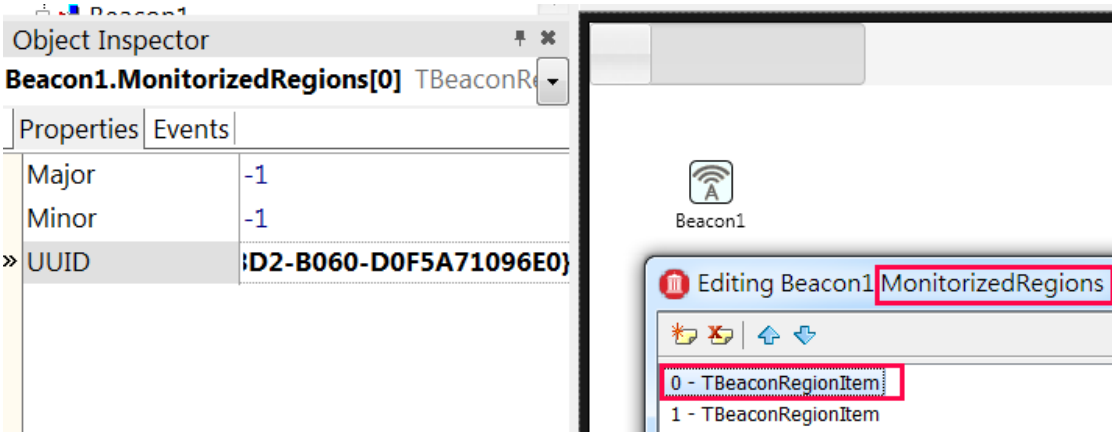
为了进一步简化开发人员的工作，封装了 TBeacon 组件,开发人员只需要使用 TBeacon 组件就可以更简单的完成上面的工作。TBeacon 组件提供了下面的特性设定：

特性	说明
Mode	设定使用 iBeacon 或是 AltBeacon 设备。设定 Standard 代表 iBeacon，设定 Alternative 代表 AltBeacon 设备
MonitorizedRegions	设定设定要扫瞄的 Beacon 区域，即 Beacon 设备的 UUID 值

TBeacon 组件也提供了下面的事件处理函式：

事件处理函式	说明
OnBeaconEnter	在进入要扫瞄的 Beacon 设备范围时触发
OnBeaconExit	在离开要扫瞄的 Beacon 设备范围时触发
OnBeaconProximity	在和 Beacon 设备的距离改变时触发
OnEnterRegion	在进入要扫瞄的 Beacon 区域范围时触发
OnExitRegion	在离开要扫瞄的 Beacon 区域范围时触发
OnCalcDistance	在要计算和 eacon 设备的距离时触发
OnParseManufacturerData	在解析 Beacon 设备的制造商数据时触发

现在就可以说明如何使用 TBeacon 组件来开发已知 Beacon 设备 App 了，请在 IDE 中建立一个 Multi-Device 应用程序，在主窗体中加入 TToolBar，TTabControl，TSwitch，TRectangle 和 TBeacon 组件，并在对象查看器中点选 TBeacon 组件的 MonitorizedRegions 特性，在其中加入 2 个 TBeaconRegionItem 对象，在每一个 TBeaconRegionItem 对象的 UUID 特性中输入你要监督扫瞄的 Beacon 设备的 UUID 值，如下所示：



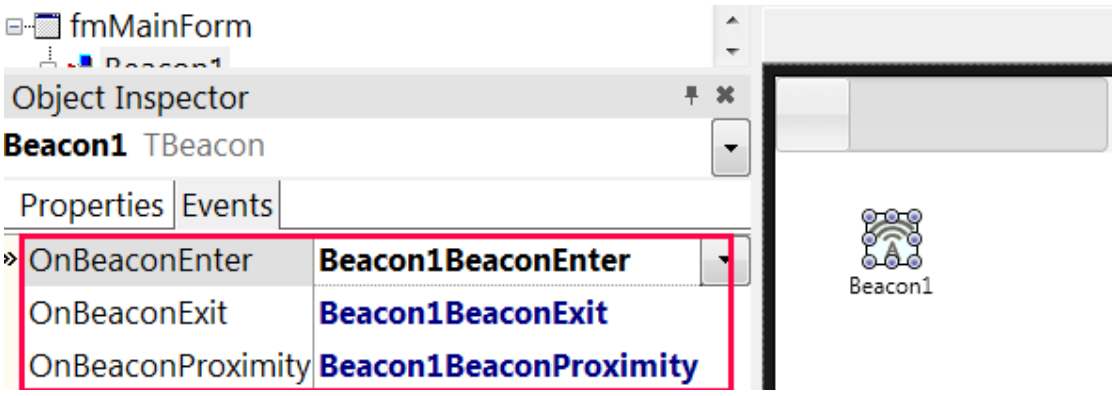
在上面的 TBeaconRegionItem 对象中其 Major 和 Minor 特性值都是-1，-1 代表要监督扫瞄具有相同 UUID 值的 Beacon 设备，不管其 Major 和 Minor 特性值是什么。

先在主窗体中 TSwitch 组件的 OnSwitch 事件处理函式根据 TSwitch 组件的开启状况开始扫瞄或是停止扫瞄：

```
void __fastcall TfmMainForm::Switch1Switch(TObject *Sender)
{
  if (Switch1->IsChecked)
  StartScanBeacons();
  else
  StopScanBeacons();

}

void TfmMainForm::StartScanBeacons()
{
  Beacon1->StartScan();
}

void TfmMainForm::StopScanBeacons()
{
  Beacon1->StopScan();
}
```


要开始扫瞄或是停止扫瞄只需要呼叫 TBeacon 组件的 StartScan 和 StopScan 方法即可。

接着为主窗体中的 TBeacon 组件设定如下的事件处理函式：



在的 TBeacon 组件的 OnBeaconEnter 和 OnBeaconExit 事件中我们只是显示一讯息代表进入或是离开 Beacon 设备的有效范围：

```
void __fastcall TfmMainForm::Beacon1BeaconEnter(TObject * const
Sender, IBeacon * const ABeacon,
       const TBeaconList CurrentBeaconList)
{
  DisplayBeaconInfo(ABeacon, L"进入 : ");
}
//---------------------------------------------------------------
void __fastcall TfmMainForm::Beacon1BeaconExit(TObject * const
Sender, IBeacon * const ABeacon,
       const TBeaconList CurrentBeaconList)
{
  DisplayBeaconInfo(ABeacon, L"离开 : ");
}
//---------------------------------------------------------------
void TfmMainForm::DisplayBeaconInfo(IBeacon * const ABeacon, const
String sStatus)
{
  TMonitor::Enter(FLock);
  try
  {
  TListViewItem *alvi = lvBeacons->Items->Add();
  alvi->Text = sStatus + GUIDToString(ABeacon->GUID);
  alvi->Detail = "Major: " + IntToStr(ABeacon->Major) + " Minor: "
+ IntToStr(ABeacon->Minor)  + " Time :" + TimeToStr(Now());
  }
  __finally
  {
  TMonitor::Exit(FLock);
  }
}
```


上面的 Flock 对象变量是 TObject 型态的对象：

```
private:  // User declarations
  TObject *FLock;
```

它在主窗体的 OnCreate 事件中建立并在 OnDestroy 事件中建立释放：

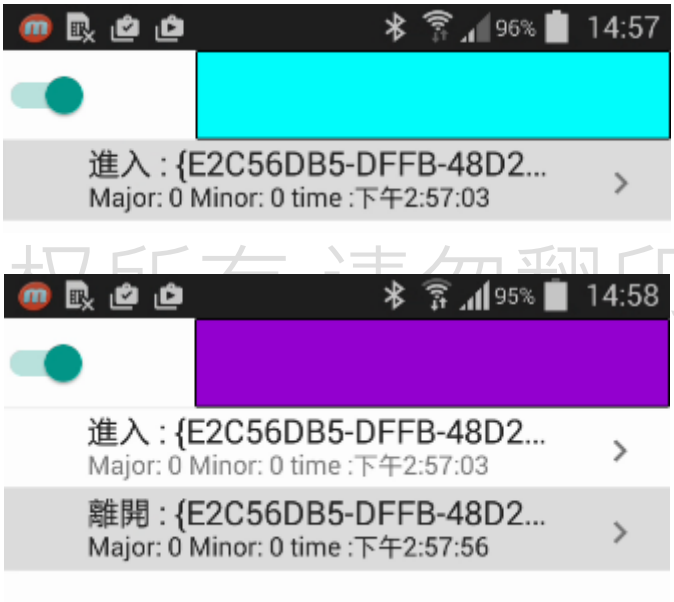
```
void __fastcall TfmMainForm::FormCreate(TObject *Sender)
{
  FLock = new TObject();
}
//---------------------------------------------------------------
void __fastcall TfmMainForm::FormDestroy(TObject *Sender)
{
  Beacon1->Enabled = false;
  delete FLock;
}
```

最后在 TBeacon 组件的 OnBeaconProximity 事件中我们根据 Beacon 设备的距离来改变主窗体中 TRectangle 组件的颜色：

```
void __fastcall TfmMainForm::Beacon1BeaconProximity(TObject *
const Sender, IBeacon * const ABeacon,
       TBeaconProximity Proximity)
{
  TMonitor::Enter(FLock);
  try
  {
  switch (ABeacon->Proximity)
  {
      case TBeaconProximity::Immediate :
        this->rtProximity->Fill->Color =
TAlphaColorRec::Springgreen;
        break;
      case TBeaconProximity::Near :
        this->rtProximity->Fill->Color = TAlphaColorRec::Aqua;
        break;
      case TBeaconProximity::Far :
        this->rtProximity->Fill->Color =
TAlphaColorRec::Slateblue;
        break;
      case TBeaconProximity::Away :
        this->rtProximity->Fill->Color =
TAlphaColorRec::Darkviolet;
```

```
        break;
    }
    }
    __finally
    {
    TMonitor::Exit(FLock);
    }
}
```

请编译并执行此范例 App 并搭配使用的 Beacon 设备就可以看到类似如下的执行画面：



TBeacon 组件果然又简单又实用，可快速帮助开发人员开发物联网的相关 App。

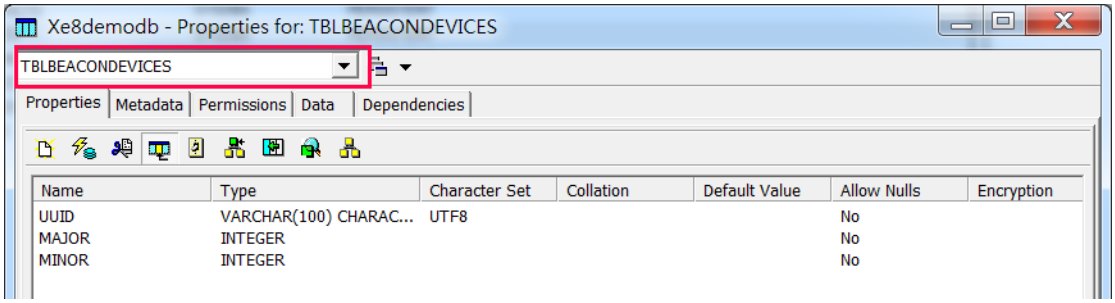
14 开发有趣的物联网应用架构

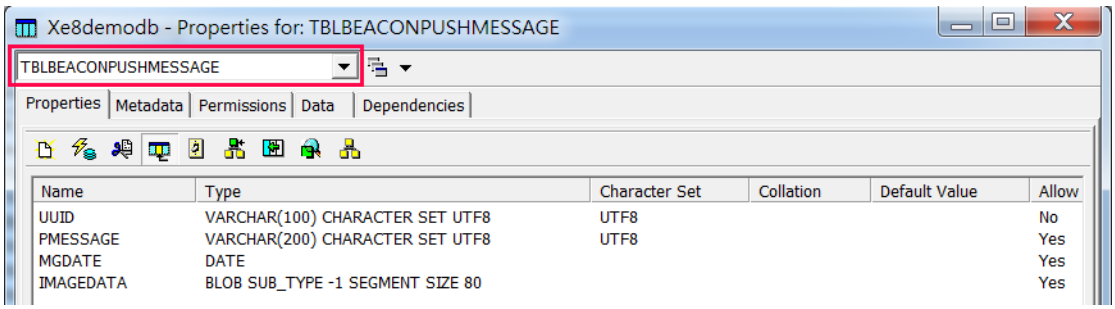
在前面的下节中已经说明了如何藉由 TBeacon 组件开发物联网相关的 App，但只是侦测和扫瞄/监督 Beacon 设备并没有什么实际的做用。开发人员应该再结合其他的技术来提供有意义的服务，例如在本书前面说明的推播技术，开发人员可以结合 Beacon 设备和推播技术以及后端的服务服务器来提供类似下图的架构：

当前端 App 侦测和扫瞄/监督 Beacon 设备之后可以呼叫后端的 DataSnap 服务来撷取更完整的数据或是呼叫后端的云端服务，而远程的 Windows 客户端也可以藉由推播技术把需要的信息主动推播到前端的 App 中，读者可以回头参考本书前面讨论的内容来开发更有趣的物联网应用 App，下面简单说明如何融合前面章节讨论的推播技术，DataSnap 技术和 Beacon 技术来开发一有趣又实用的物联网范例架构。

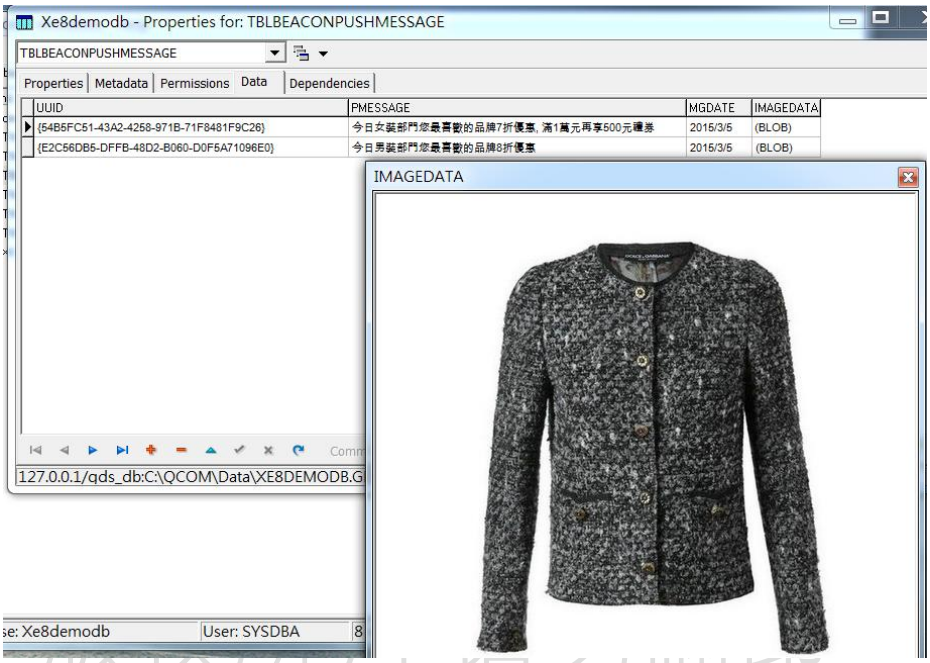
14-1 范例数据表

首先在 DEMODB.GDB 中让我们加入 2 个数据表：TBLBEACONDEVICES 和 TBLBEACONPUSHMESSAGE：



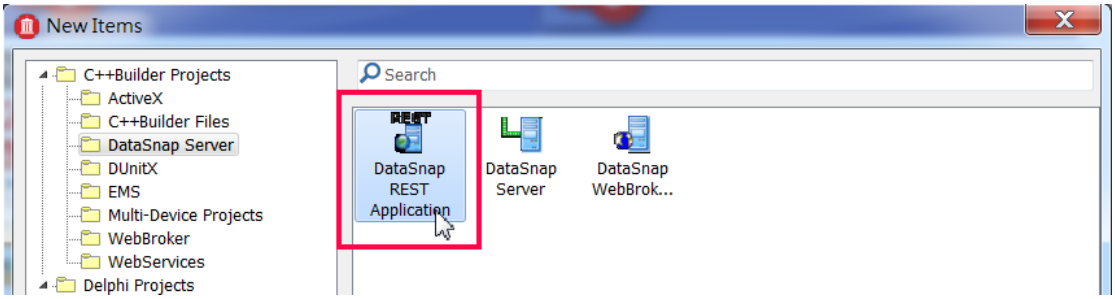


其中的 TBLBEACONDEVICES 数据表是使用来注册客户端 App 侦测到的 Beacon 设备而 TBLBEACONPUSHMESSAGE 数据表则储存了要推播到特定 Beacon 设备附近 App 的推播讯息。如下所示在范例 TBLBEACONPUSHMESSAGE 数据表中尚包含了图形的数据在用户收到推播讯息之后还可以进一步看到详细的信息：

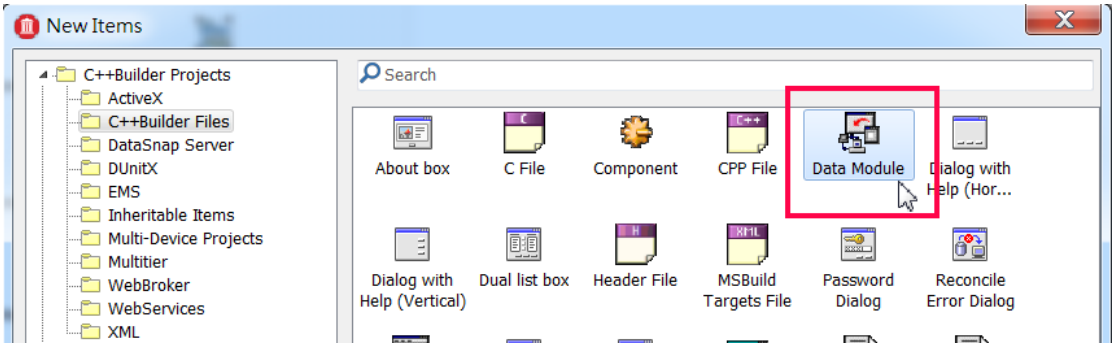


14-2 中介 DataSnap 服务器

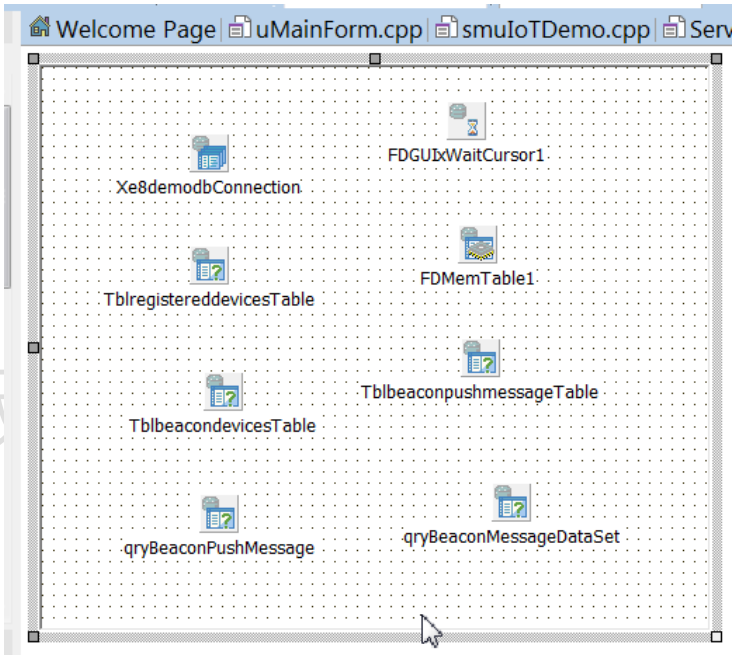
接着建立一个 DataSnap REST Application 项目：



在项目中建立一个数据模块：



在数据模块中加入如下的 FireDAC 组件准备存取前面的范例数据表：



接着在项目的 ServerMethodUnit 中加入如下的 2 个服务方法：

```
//For Beacon Services
  bool RegisterBeacon(const String sID, const String  sUUID, const
int iMajorID, const int iMinorID);
  TDataSet* GetBeaconMessage(const String sMessage);
```

RegisterBeacon()方法是在客户端侦测到 Beacon 设备时向 DataSnap 服务器使用的，而 GetBeaconMessage()方法则可以根据特定的 Beacon 设备取得包含推播讯息的 TDataSet 对象。

RegisterBeacon()方法的实作如下，它很简单只是藉由范例程序中的数据模块把侦测到的 Beacon 设备写入 TBLBEACONDEVICES 数据表中：

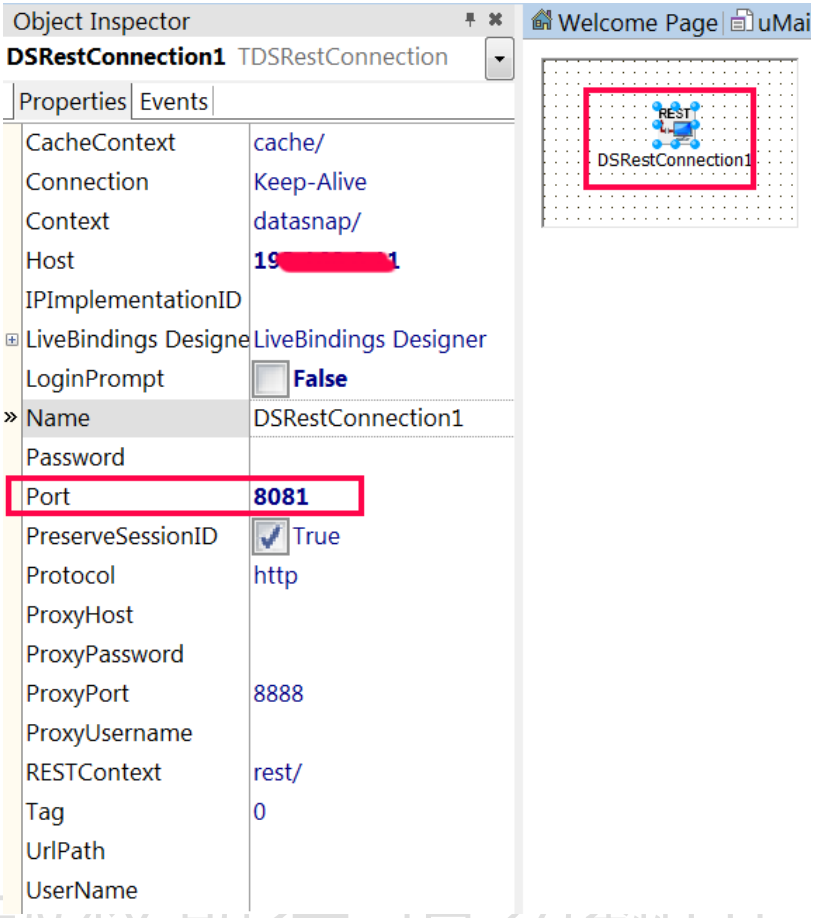
```
bool TsmBCBIoTDemo::RegisterBeacon(const String sID, const String
sUUID, const int iMajorID, const int iMinorID)
{
  bool bResult;

  System::TMonitor::Enter(dmIoTPushDemo);
  try
  {
  bResult = dmIoTPushDemo->RegisterBeacon(sID, sUUID, iMajorID,
iMinorID);
  }
  __finally
  {
  System::TMonitor::Exit(dmIoTPushDemo);
  }
  PushBeaconMessage(sID, sUUID);

  return bResult;
}
```


一旦客户端 App 侦测到特定的 Beacon 设备并呼叫 RegisterBeacon()方法注册 Beacon 设备之后就会继续呼叫 PushBeaconMessage()方法，而 PushBeaconMessage()方法则是呼叫 PushMessageTo()推播方法把 TBLBEACONPUSHMESSAGE 数据表中属于侦测到特定的 Beacon 设备推播讯息推播到客户端的 App 中：

```
bool TsmBCBIoTDemo::PushBeaconMessage(const String sDID, const
String sUUID)
{
  TFDQuery *aDataSet = NULL;
  bool bResult = true;
  System::TMonitor::Enter(dmIoTPushDemo);
  try
  {
  aDataSet = dmIoTPushDemo->GetBeaconMessage(sUUID);
  if (aDataSet != NULL)
  {
      while (!aDataSet->Eof)
```

```
      {
        bResult = PushMessageTo(sDID,
aDataSet->FieldByName("PMESSAGE")->AsString) && bResult;
        aDataSet->Next();
      }
  }
  else
     bResult = false;
  }
  __finally
  {
  System::TMonitor::Exit(dmIoTPushDemo);
  }

  return bResult;
}
```

由于在前面章节的已有有了可执行推播功能的 GCM 服务器，因此我们只需要藉由它推播 Beacon 设备的讯息即可。因此请在 ServerMethodUnit 中加入一个 TDSRestConnection 组件并设定它链接到前面章节的 GCM 服务器并产生 C++Builder 客户端的表头文件和程序文件：

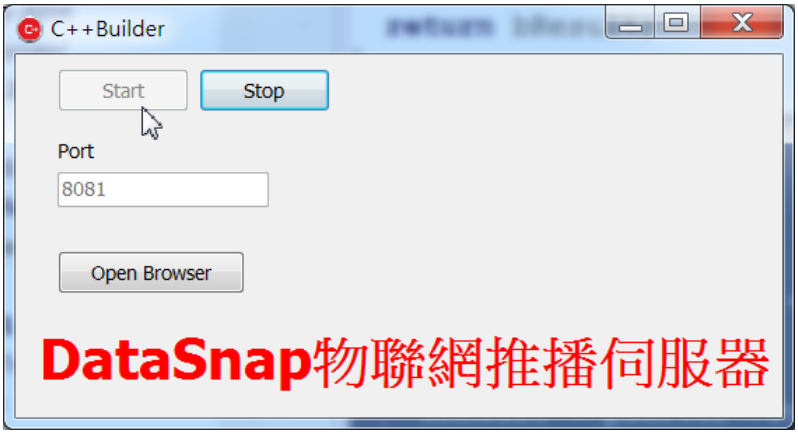


接着实作 PushMessageTo()方法呼叫前面章节的 GCM 服务器中的 PushMessageTo()方法把 Beacon 设备的讯息推播到客户端的手机中：

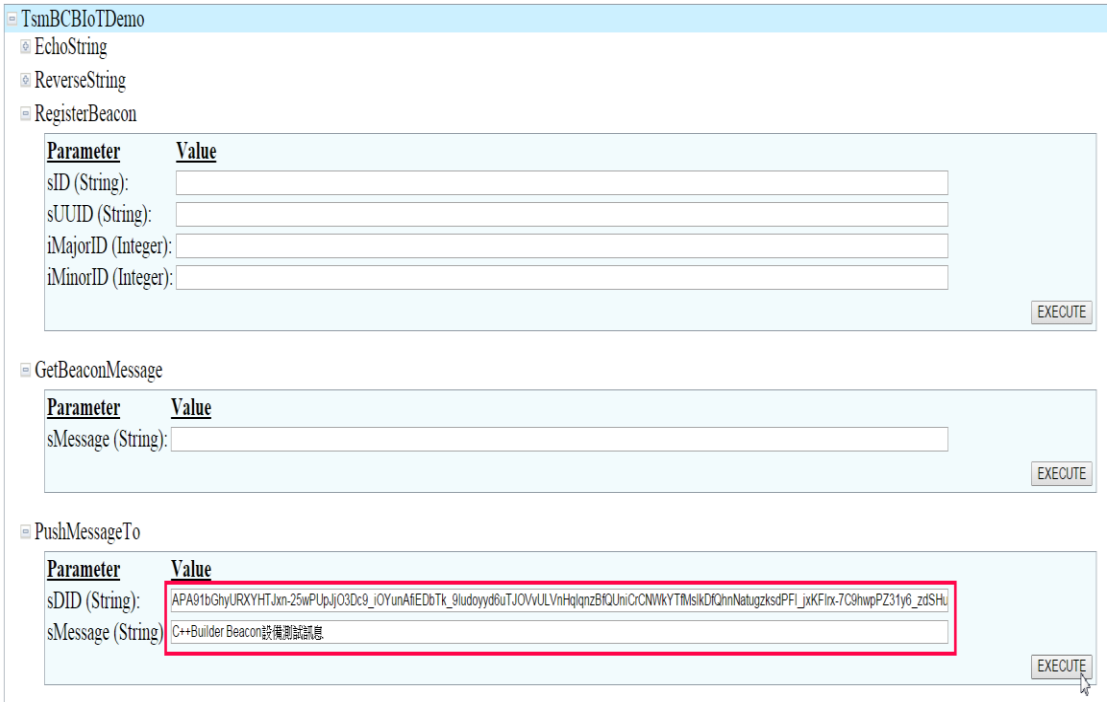
```
bool TsmBCBIoTDemo::PushMessageTo(const String sDID, const String
sMessage)
{

  bool bResult = false;

  TServerMethods1Client *pServer = new
TServerMethods1Client(this->DSRestConnection1);
  try
  {
  bResult = pServer->PushMessageTo(sDID, sMessage);
  }
  __finally
  {
  delete pServer;
```

```
  }

  return bResult;
}
```

现在请执行此范例 C++Builder DataSnap 服务器：



开启浏览器就可以看到范例 C++Builder DataSnap 服务器输出的服务方法，如果我们在浏览器中呼叫 PushMessageTo()方法：



就可以看到如下范例 C++Builder DataSnap 服务器成功的呼叫前面的 GCM 服务器：

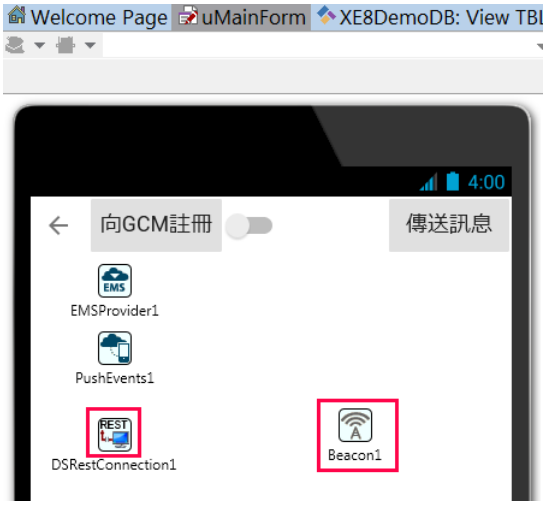
```
Executed: TsmBCBIoTDemo.PushMessageTo
        {"sDID":"APA91bGhyURXYHTJxn-
Result: 25wPUpJjO3Dc9_iOYunAfiEDbTk_9Iudoyyd6uTJOVvULVnHqIqnzBfQUniCrCNWkYTfMslkDfQhnNatugzksdPF1_jxKFIrx-
        7C9hwpPZ31y6_zdSHuCGo2sqtspTo4oBQ1Q7yx5YhiTBw","sMessage":"C++Builder Beacon設備測試訊息","result":true}
```

而范例 C++Builder DataSnap 服务器也藉由 GCM 服务器推播讯息到客户端的手机中：

接下来就来开发移动客户端 App 并结合 Beacon 的功能。

14-3 移动客户端

最后让我们在客户端 App 中使用 TBeacon 组件侦测 Beacon 设备，使用 TDSRestConnection 组件链接范例 DataSnap 服务器再使用 TEMSProvider 和 TPushEvents 组件启动推播功能：



在用户开启主窗体中的 TSwitch 组件后就启动 TBeacon 组件开始侦测 Beacon 设备：

```
 void __fastcall TfmMainForm::swActivateBeaconSwitch(TObject
*Sender)
{
  if (swActivateBeacon->IsChecked)
  Beacon1->Enabled = true;
  else
  Beacon1->Enabled = false;
}
```


在 TBeacon 组件侦测到 Beacon 设备之后就呼叫范例 DataSnap 服务器的 RegisterBeaconToServer() 方法注册 Beacon 设备并开始接收属于此特定 Beacon 设备的推播讯息：

```
void __fastcall TfmMainForm::Beacon1BeaconEnter(TObject * const
Sender, IBeacon * const ABeacon,
       const TBeaconList CurrentBeaconList)
{
  try
  {
  ListView1->Items->Add()->Text = L"发现 Beacon : " +
GUIDToString( ABeacon->GUID);
  RegisterBeaconToServer(ABeacon);
  }
  catch (const Sysutils::Exception& ex)
```

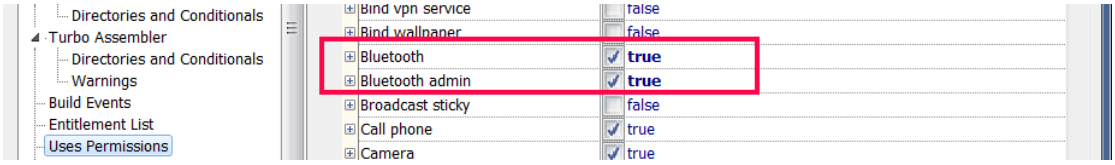
```
  {
  ListView1->Items->Add()->Text = ex.Message;
  }
  TabControl1->TabIndex = 1;
}
```

RegisterBeaconToServer()方法藉由 TDSRestConnection 组件自动产生的客户端链接程序码以链接范例 DataSnap 服务器：

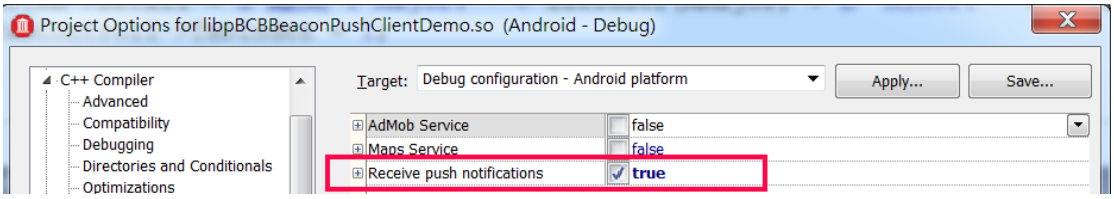
```
void TfmMainForm::RegisterBeaconToServer(IBeacon * const ABeacon)
{

  TsmBCBIoTDemoClient *aServer = new
TsmBCBIoTDemoClient(this->DSRestConnection1);
  try
  {
  try
  {
    if (aServer->RegisterBeacon(PushEvents1->DeviceToken,
GUIDToString(ABeacon->GUID), ABeacon->Major, ABeacon->Minor))
      ListView1->Items->Add()->Text = L"Beacon 注册成功";
  }
  catch (const Sysutils::Exception& ex)
  {
    ;
  }
  }
  __finally
  {
  delete aServer;
  }
}
```


14-4 执行 DataSnap 服务器和客户端 App

在编译并执行此 Beacon 推播 App 之前，记得要把蓝牙存取许可打开：



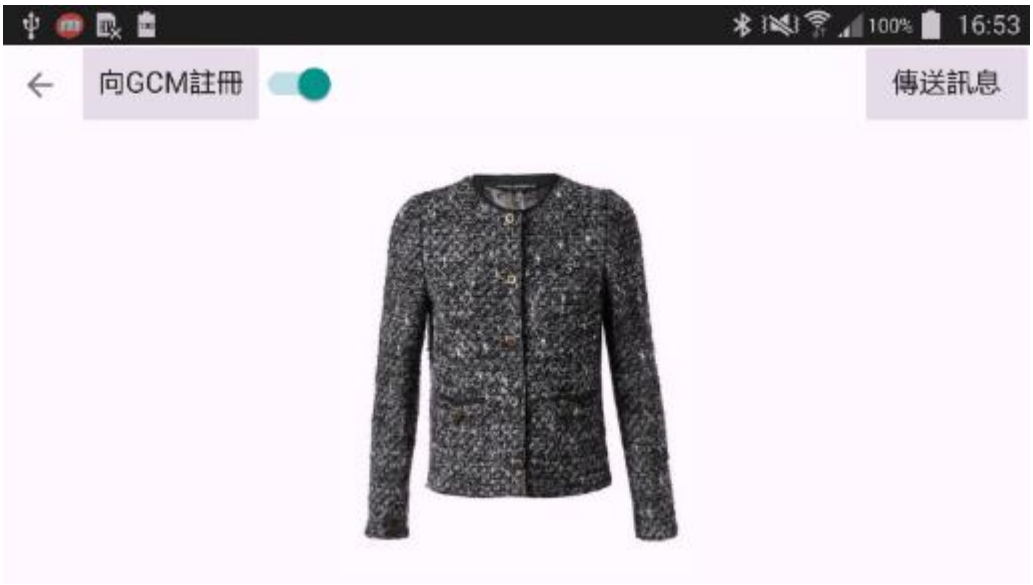
也需要把推播功能开启：



最后要在范例项目的 AndroidManifest.template.xml 档中加入如下的服务：

```
<%activity%>
<service
android:name="com.embarcadero.gcm.notifications.GCMIntentService
" />
```


完成上面的步骤后就可以编译并执行 DataSnap 服务器和客户端 App，就可在客户端 App 看到如下的执行结果，客户端 App 能够自动侦测 Beacon 设备，自动接收推播讯息，并且可在客户端 App 中查询到图形的推播数据了：



15 新的 JSON 类别库

现今的应用软件愈来愈开放也愈来愈依赖使用 JSON/RESTful 的架构，C++Builder 早在数个版本之前就支持 JSON 的开发，例如 System.JSON 程序单元中的各个 JSON 相关类别。但由于 System.JSON 中的类别属于早期的设计，到今日应用程序更积极趋向更快，更小的方向发展，因此提供一个更有弹性和更快速的 JSON 框架并应用于 C++Builder 的 DataSnap，REST 和开发人员的 App 中就更显需求了。因此在 C++Builder 中开始提供新一代的 JSON 框架。

这个新的 JSON 框架设计目标如下：

- 更快，更节省资源
- 可提供 JSON 对象和 C++Builder 对象之间的串行化功能
- 可提供开发人员客制化的能力
- 支援 BSON

新的 JSON 框架主要是由数个类别所组成，它们包含了：

- TJSONReader 以及其衍生类别，例如 TJSONTextReader，TBSONReader
- TJSONWriter 以及其衍生类别，例如 TJSONTextWriter，TBSONWriter
- TJSONObjectBuilder，TJSONArrayBuilder 等封装 TJSONWriter 的辅助类别
- TJSONConverter 提供型态转换的类别

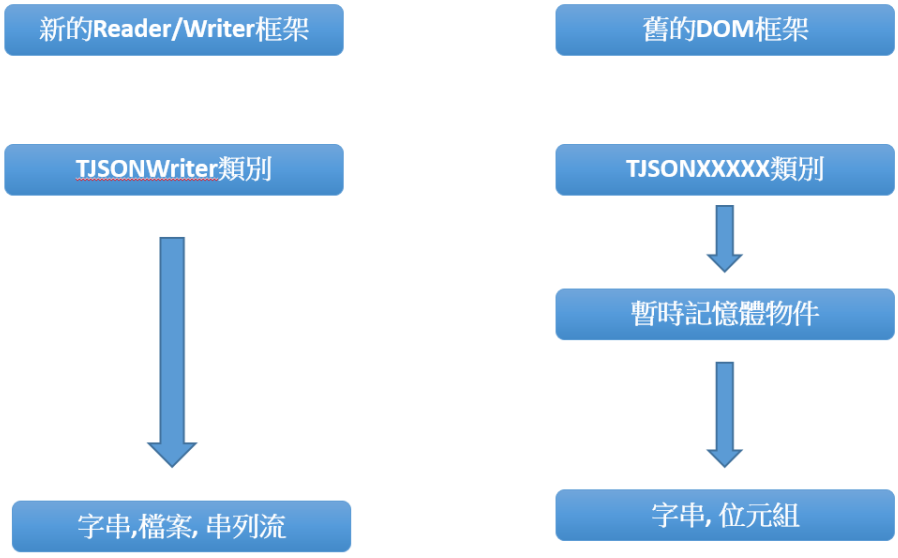
当然其中还有更多的类别无无一一列出，不过开发人员基本上只需要使用上述的类别应该就能完成大部份的工作。

15-1 JSON Reader/Writer 类别

新的 JSON 类别库提供了 Reader/Writer 新类别，所谓的 Reader 是指从字符串或是 2 进位串行流读取 JSON 的数据，而 Writer 则是指把字符串数据封装成 JSON 对象。

在之前的版本中 C++Builder 是使用 TJSONValue，TJSONObject，TJSONArray 等类别来处理 JSON 相关的开发，这些类别是使用 DOM 模型实作的。而新的 JSON Reader/Writer 类别则改用 Stream 模型实作。JSON Reader 类别是由 TJsonReader 这个抽象类开始，真正的实作类别是 TJsonTextReader，它是从 TJsonReader 继承下来。

而 JSON Writer 类别是由 TJsonWriter 这个抽象类开始，接着有衍生类别 TJsonTextWriter 和 TJsonObjectWriter。TJsonTextWriter 是直接可把数据撰写成 JSON 格式的数据，而 TJsonObjectWriter 则是把数据直接写成 JSON 对象的格式。下面的图形说明了新的 JSON 框架和旧的 JSON DOM 框架在实作上的不同：



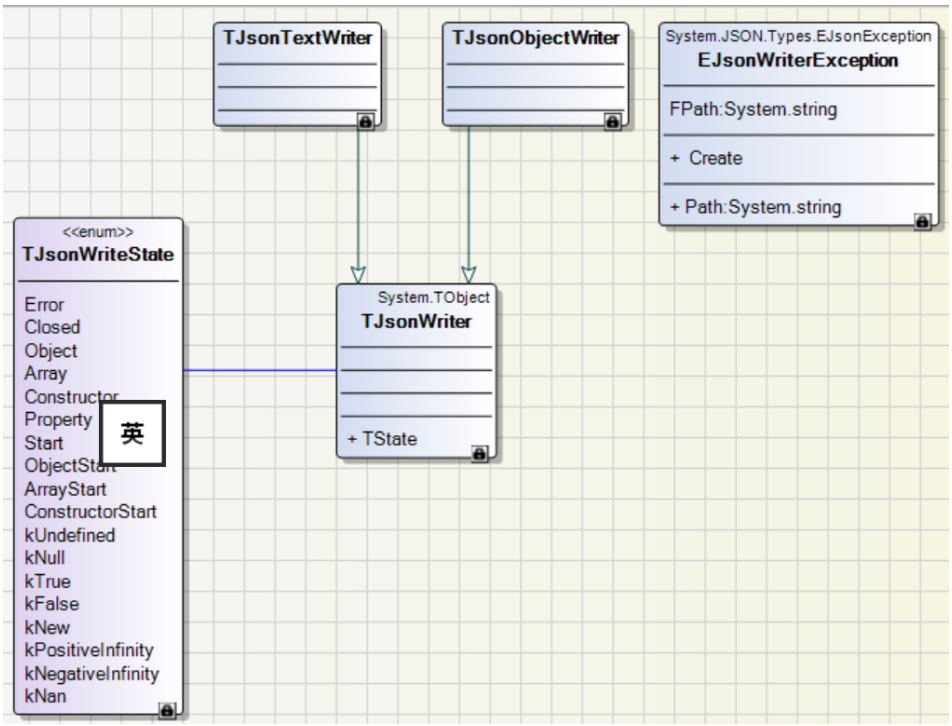
其实新的 JSON 框架使用上非常的简单，让我们使用下的 JSON 数据来说明如何藉由 TJsonWriter 的相关类别输出下面 JSON 格式的数据，在下一小节再使用 TJsonrReader 的相关类别再把数据读出：

```
{
        "name" : "王小华",
        "account" : "WSH",
        "country" : "tw",
        "age" : "22",
        "email" : [
                "wsh@gmail.com.tw",
                "wsh@hotmail.com"
        ]
}
```


上面的范例 JSON 格式的数据也使用在 FireDAC 数据库中 MongoDB 的章节中，因为 MongoDB 就是使用 JSON/BSON 格式的数据，而 FireDAC 也使用了新的 JSON 框架来处理 JSON/BSON 格式的数据。

15-1-1 Writer 类别

新的 JSON Writer 类别是由 TJsonWriter，TJsonTextWriter 和 TJsonObjectWriter 等类别组成，下图是它们的类别图：



下表说明了每个类别的功能：

类别	说明
TJsonWriter	根抽象类，实际功能由它的衍生类别实作
TJsonTextWriter	把数据写成字符串或是串行流格式的 JSON 数据
TJsonObjectWriter	把数据写成的 JSON Object 格式的数据

要把数据写成 JSON 的格式我们可以直接使用 TJsonTextWriter 类别，TJsonTextWriter 的建构元接受一个 TTextWriter 的对象：

```
__fastcall TJsonTextWriter(System::Classes::TTextWriter* const
TextWriter);
```

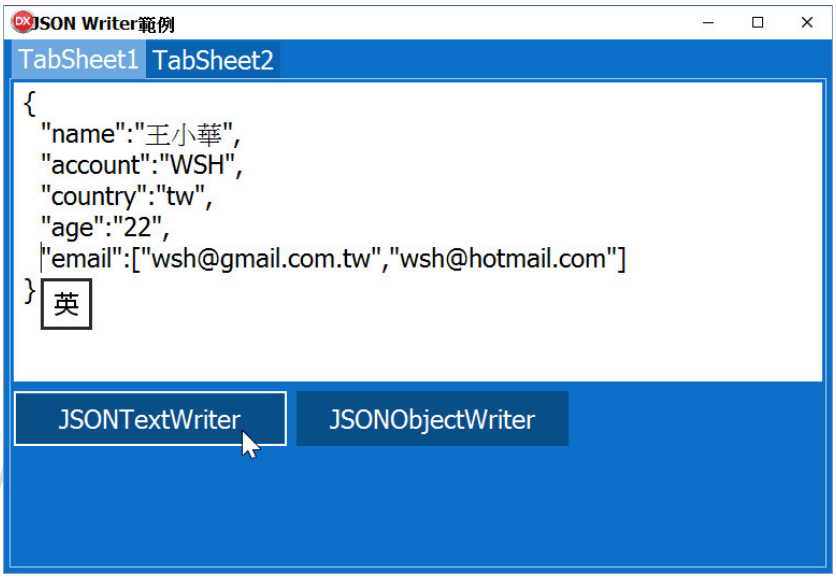
TJsonTextWriter 类别提供了许多的 Write 方法让程序员呼叫，例如上面的数据格式是 JSON 对象，因此 TJsonTextWriter 类别提供了 WriteStartObject/WriteEndObject 方法，如果要写出 JSON 数组对象，那可呼叫 WriteStartArray/WriteEndArray 方法。要写出 JSON Pair 的名称可用 WritePropertyName，而要写出 JSON Pair 的数值部分则可用各种复载的 WriteValue 方法。

例如要使用 TJsonTextWriter 写出前面”王小华”的 JSON 资料，我们可以使用下面的程序代码：

```
001    void TfmMainForm::WritePair(TJsonTextWriter *jtw, const
```

```
String sName, const String sValue)
002   {
003     jtw->WritePropertyName(sName);
004     jtw->WriteValue(sValue);
005   }
006
007   void __fastcall TfmMainForm::Button1Click(TObject *Sender)
008   {
009     TJsonTextWriter *jtw;
010     TStringWriter *sw = new TStringWriter();
011     try
012     {
013       jtw = new TJsonTextWriter(sw);
014
015       jtw->WriteStartObject();
016       WritePair(jtw, "name", L"王小华");
017       WritePair(jtw, "account", "WSH");
018       WritePair(jtw, "country", "tw");
019       WritePair(jtw, "age", "22");
020
021       //JSON 数组
022       jtw->WritePropertyName("emails");
023       jtw->WriteStartArray();
024       String eMail1 = "wsh@gmail.com.tw";
025       jtw->WriteValue(String("wsh@gmail.com.tw"));
026       jtw->WriteValue(String("wsh@hotmail.com"));
027       jtw->WriteEndArray();
028       jtw->WriteEndObject();
029       FDemoData = sw->ToString();
030       Memo1->Lines->Text = FDemoData;
031     }
032     __finally
033     {
034       jtw->Close();
035       delete jtw;
036       delete sw;
037     }
038   }
```

010 行先建立一个 TStringWriter 对象，准备把 JSON 数据写入，013 行建立 TJsonTextWriter 对象。由于”王小华”是使用 JSON 对象封装，因此 015 行呼叫 WriteStartObject()方法写出’{’，接着呼叫 WritePair()方法呼叫 TJsonTextWriter 类别的 WritePropertyName()和 WriteValue()方法写出 JSON Pair 数据，最后写出 email 这个 JSON Pair 数据，但由于 email 的 Value 值又是使用 JSON 数组封装，因此 023 行再呼叫 WriteStartArray()方法写出’[‘，最后 027，028 行分别呼叫 WriteEndArray()和 WriteEndObject()写出’]’和’}’即可完成，025 行呼叫 TJsonTextWriter 类别的 Close()方法完成写入 TStringWriter 对象的工作。下图是此范例执行在 Windows 10 的结果：



同样的工作我们也可以使用 TJsonObjectWriter 类别来完成：

```
001    void TfmMainForm::WritePair(TJsonObjectWriter *jow, const
String sName, const String sValue)
002    {
003      jow->WritePropertyName(sName);
004      jow->WriteValue(sValue);
005    }
006
007    void __fastcall TfmMainForm::Button2Click(TObject *Sender)
008    {
009      TJsonObjectWriter *jow = new TJsonObjectWriter(true);
010      try
011      {
012        jow->WriteStartObject();
013        WritePair(jow, "name", L"王小华");
```

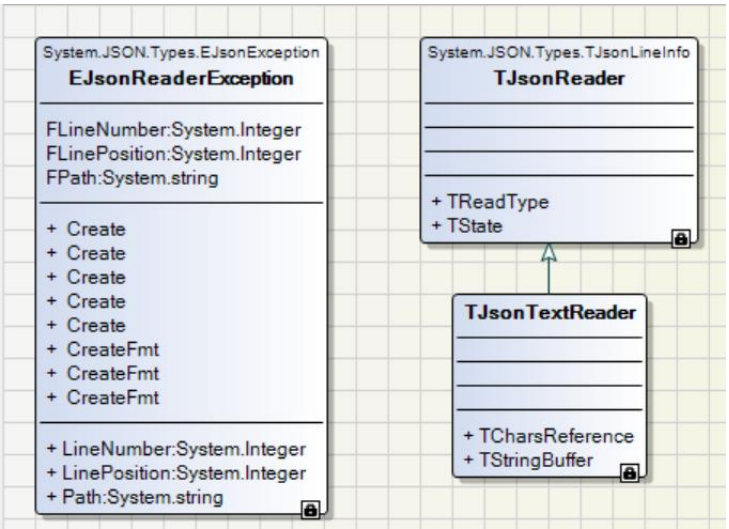
```
014       WritePair(jow, "account", "WSH");
015       WritePair(jow, "country", "tw");
016       WritePair(jow, "age", "22");
017
018       //JSON 数组
019       jow->WritePropertyName("emails");
020       jow->WriteStartArray();
021       String eMail1 = "wsh@gmail.com.tw";
022       jow->WriteValue(String("wsh@gmail.com.tw"));
023       jow->WriteValue(String("wsh@hotmail.com"));
024       jow->WriteEndArray();
025       jow->WriteEndObject();
026       FDemoData = jow->JSON->ToString();
027       Memo1->Lines->Text = FDemoData;
028     }
029     __finally
030     {
031       jow->Close();
032       delete jow;
033     }
034   }
```


使用 TJsonObjectWriter 对象就不需要先建立 TStringWriter 对象，我们可以直接把数据写入 TJsonObjectWriter 对象的内存中，最后在 026 行存取它的 JSON 特性值即可取得最的的结果。

TJsonWriter 在使用上非常的直觉又简单，接下来我们就可以讨论如何使用 Reader 类别。

15-1-2 Reader 类别

TJsonReader 类别的功能是从 JSON 格式的数据读出我们需要的数据，因此 TJsonReader 类别是从 JSON 的字符串中解析和读取数据，所以 TJsonReader 类别组非常的简单基本上目前只有 TJsonReader 和 TJsonTextReader 类别：



基本上TJsonReader类别在解析和读取JSON数据时使用符号触发处理的方式让程序员处理 JSON 数据中的每一个元素，而原本的 System.Json.hpp 中的类别则是需要程序员先了解 JSON 数据的实际格式再解析和读取，也许让我们使用一个简单的范例来说明会更容易了解。

例如现在我们希望从下面的 JSON 对象中读出"姓名"这个 JSON Pair 的数值：

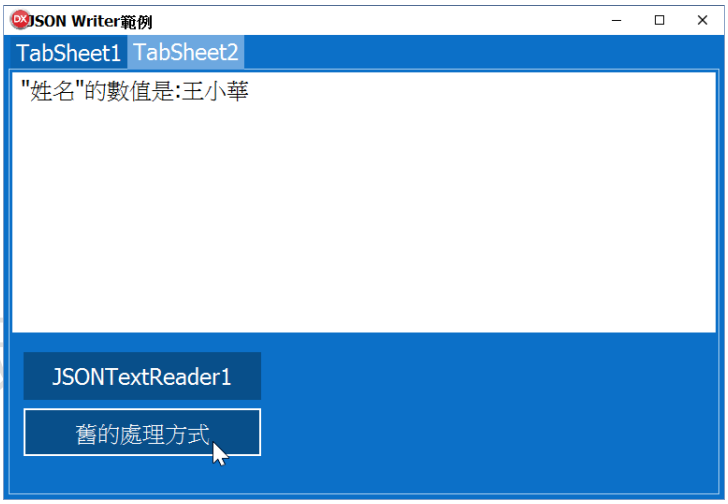

```
{
  "姓名" : "王小华"
}
```

那么在以前可以使用如下的程序代码：

```
void __fastcall TfmMainForm::Button4Click(TObject *Sender)
{
  TJSONObject *jo;

  TJSONValue *jv = TJSONObject::ParseJSONValue(SPEOPLEDATA);
  try
  {
  TJSONObject *jo = dynamic_cast< TJSONObject* > (jv);
  if (jo != 0)
  {
    TJSONPair *jp = jo->Pairs[0];
    String sData = L"\"" + jp->JsonString->Value() + L"\"的数值是:"
+ jp->JsonValue->Value();
```

```
    Memo2->Lines->Add(sData);
  }
  }
  __finally
  {
  delete jv;
  }
}
```

如果仔细看上面的程序代码可以发现程序员需要事先知道要处理的 JSON 格式：

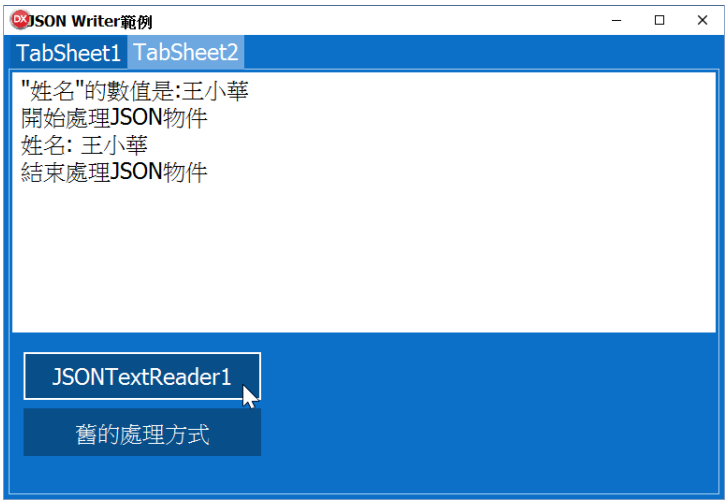


在新的 JSON 框架中可以使用如下的程序代码来解析 JSON 资料，006 行先把 JSON 数据传入 TStringReader 对象，再于 007 行建立 TJsonTextReader 对象，011 行开始呼叫 TJsonTextReader 对象的 Read()方法读取 JSON 数据，接着可使用一个 switch 子句来判断目前读取到的内容再决定要如何处理：

```
001    const String SPEOPLEDATA = L"{\"姓名\" : \"王小华\"}";
002
003    void __fastcall TfmMainForm::Button3Click(TObject *Sender)
004    {
005      String sData;
006      TStringReader *sr = new TStringReader(SPEOPLEDATA);
007      TJsonTextReader *jr = new TJsonTextReader(sr);
008      try
009      {
010        jr->Rewind();
```

```
011       while (jr->Read())
012       {
013         switch (jr->TokenType)
014         {
015           case TJsonToken::StartObject:
016             Memo2->Lines->Add(L"开始处理 JSON 对象");
017             break;
018           case TJsonToken::PropertyName:
019             sData = jr->Value.ToString() + ": " +
jr->ReadAsString();
020             break;
021           case TJsonToken::String:
022             sData = sData + jr->ReadAsString();
023             break;
024           case TJsonToken::EndObject:
025             Memo2->Lines->Add(sData);
026             Memo2->Lines->Add(L"结束处理 JSON 对象");
027             break;
028         }
029       }
030     }
031     __finally
032     {
033       delete sr;
034       delete jr;
035     }
036   }
```


例如在 023 行我们可以存取 TJsonTextReader 对象的 Value 特性值取得目前读取到的 JSON Pair 的名称部分的内容(即"姓名"), 再呼叫它的 ReadAsString()方法取得 JSON Pair 的数值部分的内容(即"王小华"),而下图是此 2 种方法执行的结果:



新的 JSON 框架让程序员可以在事先不知道 JSON 数据格式的情形下更容易的解析和读取 JSON 数据的内容。

因此我们可以使用下面的程序代码藉由 TJsonTextReader 对象来处理前面”王小华”的范例数据：


```
001    const String SDEMODATA = L"{\"name\" : \"王小华\",
\"account\" : \"WSH\", \"country\" : \"tw\", \"age\" : \"22\",
\"email\" : [\"wsh@gmail.com.tw\",\"wsh@hotmail.com\"]}";
002
003    void __fastcall TfmMainForm::Button5Click(TObject *Sender)
004    {
005      String sData;
006      TStringReader *sr = new TStringReader(SDEMODATA);
007      TJsonTextReader *jr = new TJsonTextReader(sr);
008      try
009      {
010        jr->Rewind();
011        while (jr->Read())
012        {
013          switch (jr->TokenType)
014          {
015            case TJsonToken::StartObject:
016              Memo2->Lines->Add(L"开始处理 JSON 对象");
017              break;
018            case TJsonToken::PropertyName:
```

```
019             if (jr->Value.ToString() != "email")
020             {
021               sData = jr->Value.ToString() + ": " +
jr->ReadAsString();
022               Memo2->Lines->Add(sData);
023             }
024             break;
025           case TJsonToken::EndObject:
026             Memo2->Lines->Add(L"结束处理 JSON 对象");
027             break;
028           case TJsonToken::StartArray:
029             Memo2->Lines->Add(L"开始处理 JSON 数组");
030             while (jr->Read())
031             {
032               switch (jr->TokenType)
033               {
034                 case TJsonToken::String:
035                   Memo2->Lines->Add(jr->Value.ToString());
036                   break;
037                 case TJsonToken::EndArray:
038                   Memo2->Lines->Add(L"结束始处理 JSON 数组");
039                   break;
040               }
041             }
042             break;
043         }
044       }
045     }
046     __finally
047     {
048       delete sr;
049       delete jr;
050     }
051   }
```


上面的关键点在 028 行到 042 行，一旦 TJsonTextReader 对象处理到 JSON 数组，我们就需要藉由 TJsonTextReader 对象的 Value 特性值来一一

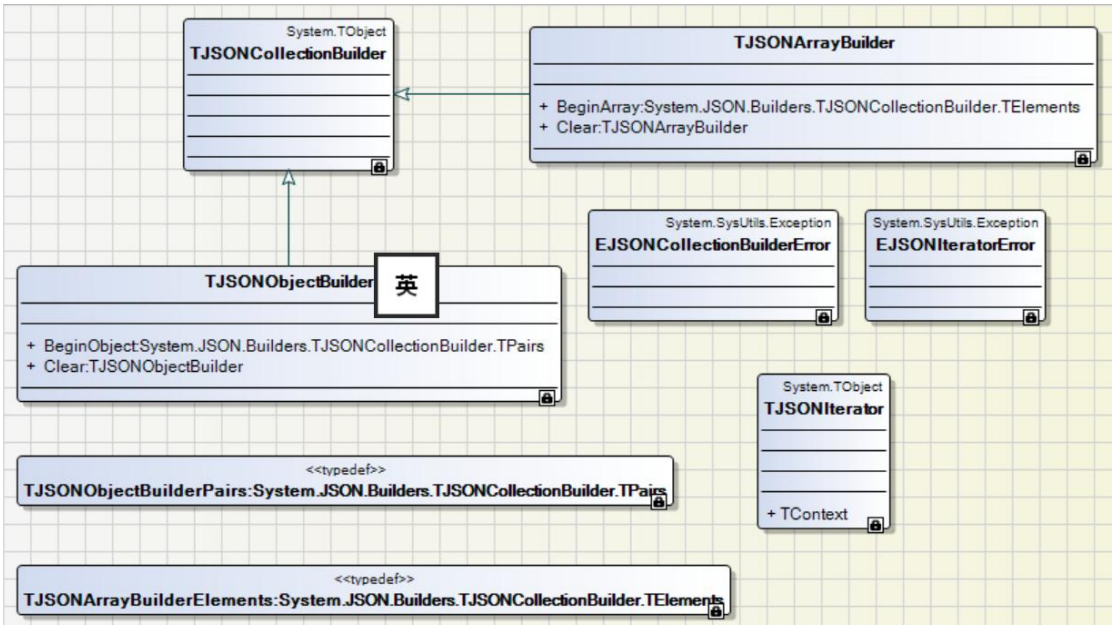
存取数组中的每一个元素值，下面就是执行的结果，我们可以看到TJsonTextReader对象可以正确解析和读取出JSON数据的内容：

15-2 JSON Builder 类别

虽然使用新的JSON Reader/Writer类别可以比以前的JSON类别更方便快速的处理JSON数据，但是撰写起程序代码来比较繁琐，为了进一步帮助开发人员能更方便处理JSON，因此在新的JSON框架中也提供了更方便使用的JSON Builder类别。

JSON Builder类别使用了类似Helper类别的设计方式，为大多数方法都回传方法的对象(Self)，因此程序员可以使用更简洁的程序代码来完成工作。

JSON Builder类别主要是由TJSONCollectionBuilder，TJSONObjectBuilder，TJSONArrayBuilder和TJSONIterator等类别组成的：



下面的表格简介了 JSON Builder 类别的功能：

类别	说明
TJSONCollectionBuilder	JSON Builder 的抽象根类别，提供衍生类别的基础功能
TJSONObjectBuilder	可建立 JSON 对象的 Builder 类别
TJSONArrayBuilder	可建立 JSON 数组的 Builder 类别
TJSONIterator	执装 TJSONReader 的类别，可读取 JSONl 数据中的内容

为了帮助读者了解如何使用这些新的 JSON Builder 类别，仍然让我们继续使用前面”王小华”的范例数据。

TJSONObjectBuilder 类别

TJSONObjectBuilder 类别是使用来建立 JSON 对象的，下面是它的宣告：

```
class PASCALIMPLEMENTATION TJSONObjectBuilder : public
TJSONCollectionBuilder
{
  typedef TJSONCollectionBuilder inherited;

public:
  HIDESBASE TJSONCollectionBuilder::TPairs* __fastcall
BeginObject(void);
  TJSONObjectBuilder* __fastcall Clear(void);
public:
```

```
  /* TJSONCollectionBuilder.Create */ inline __fastcall
TJSONObjectBuilder(System::Json::Writers::TJsonWriter* const
AJSONWriter)/* overload */ : TJSONCollectionBuilder(AJSONWriter)
{ }
  …
  /* TJSONCollectionBuilder.Destroy */ inline __fastcall virtual
~TJSONObjectBuilder(void) { }

};
```

要建立 JSON 对象程序员只需要呼叫 TJSONObjectBuilder 的 BeginObject()方法，而 BeginObject 方法会回传 TPairs 对象。TPairs 类别提供了许多 Add 复载方法可在 JSON 对象中加入数据，请注意的是 Add 复载方法又回传原先的 TPairs 对象：

```
  class PASCALIMPLEMENTATION TPairs : public
TJSONCollectionBuilder::TBaseCollection
  {
      typedef TJSONCollectionBuilder::TBaseCollection inherited;

  public:
      HIDESBASE TJSONCollectionBuilder::TPairs* __fastcall
Add(const System::UnicodeString AKey, const System::UnicodeString
AValue)/* overload */;
      HIDESBASE TJSONCollectionBuilder::TPairs* __fastcall
Add(const System::UnicodeString AKey, const int AValue)/* overload
*/;
      HIDESBASE TJSONCollectionBuilder::TPairs* __fastcall
Add(const System::UnicodeString AKey, const unsigned AValue)/*
overload */;
  …
  }
```


因此程序员可使用下面的语法在 JSON 对象中加入数据：

```
joBuilder->Add()->Add()->Add()…  ->EndAll();
```

上面的 EndAll()方法可在 JSON 对象中加入所有结尾符号，例如 JSON 数组结尾']'， JSON 对象结尾'}'，因此在使用 TJSONObjectBuilder 对象之后要记得呼叫它：

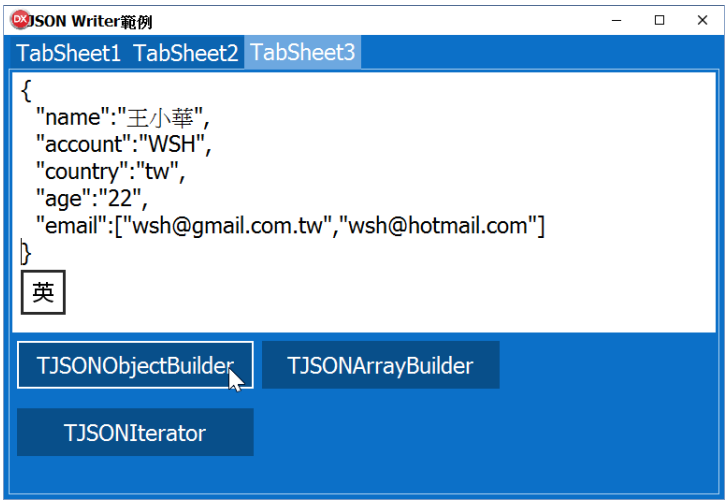
```
    void __fastcall EndAll(void);
```

因此要使用 TJSONObjectBuilder 对象写出前面"王小华"的范例 JSON 数据，我们可以使用如下的程序代码：

```
001   void __fastcall TfmMainForm::Button6Click(TObject *Sender)
002   {
003     TStringWriter *sw = new TStringWriter();
004     TJsonTextWriter *jtw = new TJsonTextWriter(sw);
005     TJSONObjectBuilder *joBuilder = new
TJSONObjectBuilder(jtw);
006     try
007     {
008       joBuilder->BeginObject()
009        ->Add("name",String(L"王小华"))
010        ->Add("account",String(L"WSH"))
011        ->Add("country",String(L"tw"))
012        ->Add("age",String(L"22"))
013        ->BeginArray("email")
014         ->Add(String(L"wsh@gmail.com.tw"))
015          ->Add(String(L"wsh@hotmail.com"))
016        ->EndAll();
017
018        Memo3->Lines->Text = sw->ToString();
019     }
020     __finally
021     {
022       delete joBuilder;
023       delete jtw;
024       delete sw;
025     }
026   }
```


003 和 004 行分别建立了 TStringWriter 和 TJsonTextWriter 对象，005 行再建立 TJSONObjectBuilder 对象并把 TJsonTextWriter 对象传入。从 008 行开始先呼叫 BeginObject()方法，因为"王小华"是要封装在 JSON 对象中，接着呼叫一连串的 Add()方法在其中加入 4 个 JSON Pair，接着呼叫 BeginArray()方法开始加入 JSON 数组，再呼叫 2 个 Add()方法在其中加入 2 个 JSON 字符串，最后呼叫 EndAll()方法完成所有写入 JSON 数据的工作，下图就是执行上

面程序代码的执行结果，我们成功的使用 TJSONObjectBuilder 对象写出了正确的 JSON 数据：



TJSONArrayBuilder 类别

TJSONArrayBuilder 类别是使用来建立 JSON 数组的，它的 BeginArray() 方法可以开始写出 JSON 数组的内容，而且它也是回传 TElements 对象：


```
class PASCALIMPLEMENTATION TJSONArrayBuilder : public
TJSONCollectionBuilder
{
  typedef TJSONCollectionBuilder inherited;

public:
  HIDESBASE TJSONCollectionBuilder::TElements* __fastcall
BeginArray(void);
  TJSONArrayBuilder* __fastcall Clear(void);
public:
  …
}
```

而 TElements 类别也提供了许多 Add()方法可在 JSON 数组中加入数据，此外 TElements 类别也提供了 BeginObject()和 BeginArray()方法，这代表 JSON 数组中的元素也可以是一个 JSON 对象或是 JSON 数组：

```
  class PASCALIMPLEMENTATION TElements : public
TJSONCollectionBuilder::TBaseCollection
  {
```

```
      typedef TJSONCollectionBuilder::TBaseCollection inherited;

  public:
      HIDESBASE TJSONCollectionBuilder::TElements* __fastcall
Add(const System::UnicodeString AValue)/* overload */;
      HIDESBASE TJSONCollectionBuilder::TElements* __fastcall
Add(const int AValue)/* overload */;
      HIDESBASE TJSONCollectionBuilder::TElements* __fastcall
Add(const unsigned AValue)/* overload */;

              ...

      HIDESBASE TJSONCollectionBuilder::TPairs* __fastcall
BeginObject(void)/* overload */;
      HIDESBASE TJSONCollectionBuilder::TElements* __fastcall
BeginArray(void)/* overload */;
      HIDESBASE TJSONCollectionBuilder::TParentCollection*
__fastcall EndArray(void);
      TJSONCollectionBuilder::TElements* __fastcall AsRoot(void);
  public:
      /* TBaseCollection.Create */ inline __fastcall
TElements(TJSONCollectionBuilder* const AOwner, const int
ARootDepth) : TJSONCollectionBuilder::TBaseCollection(AOwner,
ARootDepth) { }

  public:
      /* TObject.Destroy */ inline __fastcall virtual
~TElements(void) { }

  /* Hoisted overloads: */

  };
```

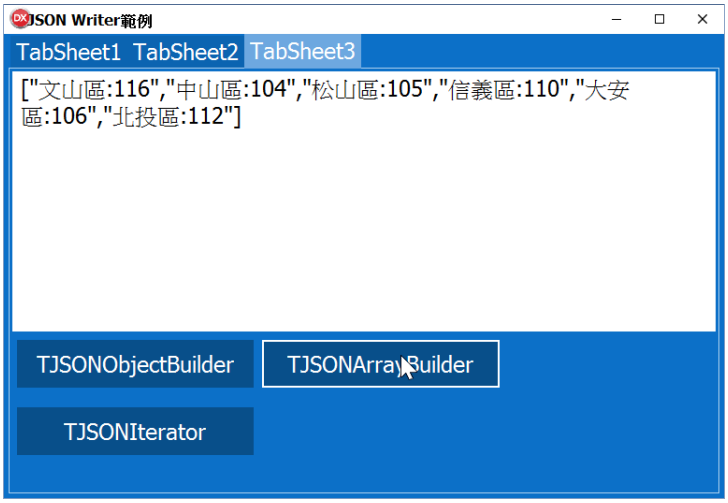

例如下面的程序代码可以建立一个包含台北市邮政编码的 JSON 数组：

```
001    void __fastcall TfmMainForm::Button7Click(TObject *Sender)
002    {
003      TStringWriter *sw = new TStringWriter();
```

```
004     TJsonTextWriter *jtw = new TJsonTextWriter(sw);
005     TJSONArrayBuilder *jaBuilder = new TJSONArrayBuilder(jtw);
006     try
007     {
008       jaBuilder->BeginArray()
009        ->Add(String(L"文山区:116") )
010        ->Add(String(L"中山区:104") )
011        ->Add(String(L"松山区:105") )
012        ->Add(String(L"信义区:110") )
013        ->Add(String(L"大安区:106") )
014        ->Add(String(L"北投区:112") )
015       ->EndAll();
016
017       Memo3->Lines->Text = sw->ToString();
018     }
019     __finally
020     {
021       delete jaBuilder;
022     }
023   }
```


003 和 004 行同样分别建立了 TStringWriter 和 TJsonTextWriter 对象，005 行再建立 TJSONArrayBuilder 对象并把 TJsonTextWriter 对象传入，从 011 行开始先呼叫 BeginArray()方法建立 JSON 数组，再呼叫一连串的 Add()方法在其中加入 JSON 字符串，最后同样呼叫 EndAll()方法结束。

下面就是执行的结果：



TJSONIterator 类别

TJSONIterator 类别是使用来更方便的读取 JSON 数据的内容，使用它的基本概念是传入一个 TJSONReader 对象给它，然后开始呼叫 Next()方法一一的读取 JSON 数据，如果遇到 JSON 对象或是 JSON 数组就呼叫 Recurse()方法再一一处理 JSON 对象或是 JSON 数组中的数据，如果没有任何剩下的 JSON 数据，那 Next()方法就回传 false。因此一个典型的程序代码架构就是：

```
while (jInerator->Next())
{
  …
}
```

当然在使用它处理 JSON 数据时，程序员仍然可以藉由它的 Type 特性值来判断目前处理的 JSON 目标是什么：

```
  __property System::Json::Types::TJsonToken Type = {read=FType,
nodefault};
```

例如如果我们想使用 TJSONIterator 类别来读取其中下面"王小华"这笔 JSON 对象中的数据：


```
const String SDEMODATA = L"{\"name\" : \"王小华\", \"account\" :
\"WSH\", \"country\" : \"tw\", \"age\" : \"22\", \"email\" :
[\"wsh@gmail.com.tw\",\"wsh@hotmail.com\"]}";
```

那么我们可以使用如下的程序代码：

```
001    void __fastcall TfmMainForm::Button8Click(TObject *Sender)
002    {
003      String sData = "";
004      TStringReader *sr = new TStringReader(SDEMODATA);
005      TJsonTextReader *jr = new TJsonTextReader(sr);
006      TJSONIterator *jInerator = new TJSONIterator(jr);
007      try
008      {
009        while (jInerator->Next())
010        {
011          if ((jInerator->Index == -1) && (jInerator->Type !=
TJsonToken::StartArray))
```

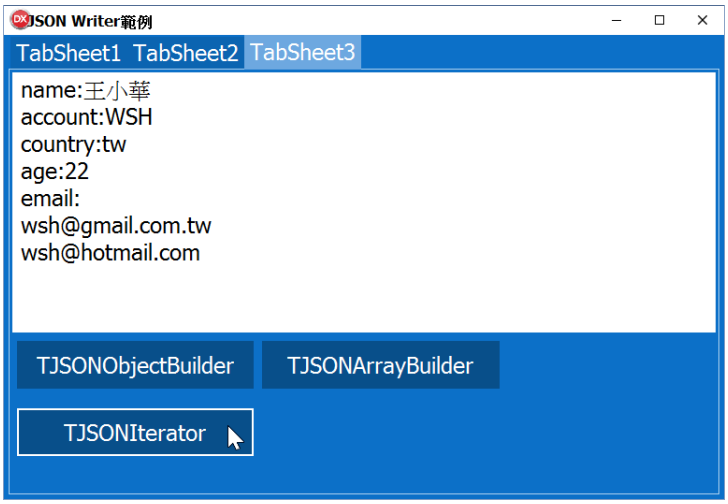
```
012           sData = sData + jInerator->Key + ":" +
jInerator->AsString + "\n";
013         else
014         {
015           if (jInerator->Index == -1)
016             sData = sData + jInerator->Key + ":";
017           else
018             sData = sData + "\n" + jInerator->AsString;
019           jInerator->Recurse();
020         }
021       }
022       Memo3->Lines->Text = sData;
023     }
024     __finally
025     {
026       delete jInerator;
027       delete jr;
028       delete sr;
029     }
030   }
```


004 行使用 TStringReader 对象输入"王小华"这笔 JSON 对象，005 行再建立 TJSONTextReader 对象，006 行建立 TJSONIterator 对象并传入 TJsonTextReader 对象做为参数，接着呼叫 Next()方法进入循环一一处理每一个 JSON 元素。对于目前的 JSON 元素名称可以存取 TJSONIterator 对象的 Key 特性值取得：

```
__property System::UnicodeString Key = {read=FKey};
```

而 JSON 元素值则可藉由存取它的各个 AsXXXX 特性值取得，例如 AsString 和 AsInteger 等。

015~019 行的程序代码主要是处理"王小华"这笔 JSON 对象中内含的 JSON 数组数据，一旦进入 JSON 对象或是 JSON 数组，那么 Key 特性值就成为元素的索引值，而且要呼叫 Recurse()方法一一处理 JSON 对象或是 JSON 数组中的每一个 JSON 元素。下图就是上面使用 TJSONIterator 对象读取出的数据结果:



15-3 BSON 类别

新的 JSON 框架中也包含了 TBsonWriter 和 TBsonReader 这 2 个能处理 BSON 的类别，基本上 BSON 就是 2 进位格式的 JSON，使用 BSON 来处理数据速度会比使用字符串格式的 JSON 来得更有效率，例如 MongoDB 就是使用 BSON 处理和储存数据。

15-3-1 TBsonWriter

TBsonWriter 类别的功能是把数据输出成 BSON 的格式，它的建构元接收 TBinaryWriter 或是 TStream 对象：

```
  __fastcall TBsonWriter(System::Classes::TBinaryWriter* const
ABinaryWriter)/* overload */;
  __fastcall TBsonWriter(System::Classes::TStream* const Stream)/*
overload */;
```

由于建构元可接收 TStream 对象，因此我们可以传入 TMemoryStream，TStringStream 和 TFileStream 等对象，TBsonWriter 便会把 BSON 格式的数据写入。

TBsonWriter 类别和前面介绍的 JSON Writer 类别一样提供了写入各种 JSON 元素的方法，例如：

```
  virtual void __fastcall WriteStartObject(void);
  virtual void __fastcall WriteEndObject(void);
  virtual void __fastcall WriteStartArray(void);
  virtual void __fastcall WriteEndArray(void);
```

因此我们可以像前面说明的一样呼叫这些方法写入 JSON 元素，当然 TBsonWriter 类别也提供了各种 WriteValue()复载方法可让程序员写入数据：

```
  virtual void __fastcall WriteValue(const System::UnicodeString
Value)/* overload */;
  virtual void __fastcall WriteValue(int Value)/* overload */;
  …
```

其中一个重要的方法就是 WriteToken()方法：

```
  void __fastcall WriteToken(System::Json::Readers::TJsonReader*
const Reader)/* overload */;
  void __fastcall WriteToken(System::Json::Readers::TJsonReader*
const Reader, bool WriteChildren)/* overload */;
```

WriteToken()方法接收一个 TJsonReader 对象，接着 WriteToken()方法就从 TJsonReader 对象中读出 JSON 数据再写入 TBsonWriter 类别的 Stream 对象中。

现在让我们继续使用"王小华"这笔 JSON 数据并把它写成 BSON 的格式再从 BSON 转回 JSON。在下面的程序代码中 SDEMODATA 包含的就是"王小华"这笔 JSON 数据，我们先呼叫 Json2Bson()方法把 JSON 格式的数据转成 BSON 格式，再呼叫 Bytes2String()方法把 BSON 格式的数据以文字的形式显示出来(记得 BSON 是 2 进位形式的资料吗?)：

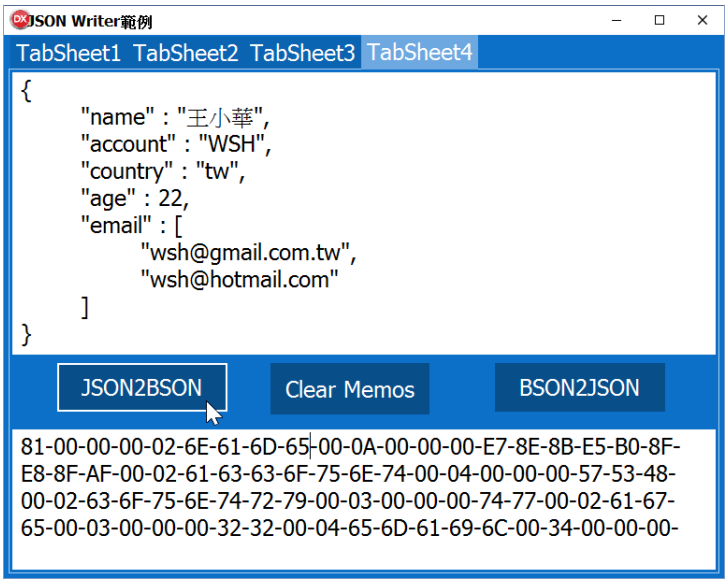

```
void __fastcall TfmMainForm::Button9Click(TObject *Sender)
{
  Memo5->Lines->Text = Bytes2String(Json2Bson(SDEMODATA));
}
```

Json2Bson()方法就使用了 TBsonWriter 类别把 JSON 格式的数据转成 BSON 格式，在下面的程序代码中首先先藉由 TJsonTextReader 对象把"王小华"这笔 JSON 数据读入，005 行建立 TBytesStream 对象，006 行再建立 TBsonWriter 对象并传入 TBytesStream 对象。接着我们只需要在 011 行呼叫 TBsonWriter 对象的 WriteToken 方法即可把 TJSONTextReader 对象中包含的 JSON 格式的数据以 BSON 的格式写入 TBytesStream 对象中，最后在 016 行以 System::TArray__1<System::Byte>型态回传转换后的 BSON 格式数据：

```
001    System::TArray__1<System::Byte>
TfmMainForm::Json2Bson(const String AJson)
```

```
002   {
003     TStringReader *sr = new TStringReader(AJson);
004     TJsonTextReader *jr = new TJsonTextReader(sr);
005     TBytesStream *Stream = new TBytesStream(NULL);
006     TBsonWriter *BsonWriter = new TBsonWriter(Stream);
007     System::TArray__1<System::Byte> bResult;
008
009     try
010     {
011       BsonWriter->WriteToken(jr);
012       bResult.set_length(Stream->Size);
013       Stream->Position = 0;
014       Stream->Read(bResult, Stream->Size);
015
016       return bResult;
017     }
018     __finally
019     {
020       delete jr;
021       delete sr;
022       delete BsonWriter;
023       delete Stream;
024     }
025   }
```


下图就是执行的结果，我们可以看到可以成功的把"王小华"这笔 JSON 数据转成下方 BSON 格式的数据：



同样的我们也可以把 BSON 格式的资料转回 JSON 格式的数据，下面的 Bson2Json 方法可把一个包含 BSON 格式数据的 System::TArray__1<System::Byte>转回 JSON：

```
001    String TfmMainForm::Bson2Json(const
System::TArray__1<System::Byte> ABson)
002    {
003      String sResult = "";
004      TBytesStream *Stream = new TBytesStream(ABson);
005      TBsonReader *BsonReader = new TBsonReader(Stream, false);
006      TStringWriter *sw = new TStringWriter();
007      TJsonTextWriter *jtw = new TJsonTextWriter(sw);
008      jtw->Formatting = TJsonFormatting::Indented;
009      try
010      {
011        jtw->WriteToken(BsonReader);
012        sResult = sw->ToString();
013      }
014      __finally
015      {
016        delete jtw;
017        delete sw;
018        delete BsonReader;
019        delete Stream;
020      }
```

```
021
022     return sResult;
023   }
```

004 行先使用 TBytesStream 接收传入的包含 BSON 格式数据的 System::TArray__1<System::Byte>，005 行使用下一小节介绍的 TBsonReader 对象读入并解析 BSON 格式数据，011 行呼叫 TJsonTextWriter 类别的 WriteToken()方法把 BSON 转成 JSON 即可。

TJsonTextWriter 类别的 WriteToken()方法宣告如下，我们只须传入 TJsonReader 对象它即可帮我们转换数据：

```
 void __fastcall WriteToken(System::Json::Readers::TJsonReader*
const Reader)/* overload */;
 void __fastcall WriteToken(System::Json::Readers::TJsonReader*
const Reader, bool WriteChildren)/* overload */;
```

而 TBsonReader 是从 TJsonReader 类别继承下来，因此它就可以把 BSON 转成 JSON：

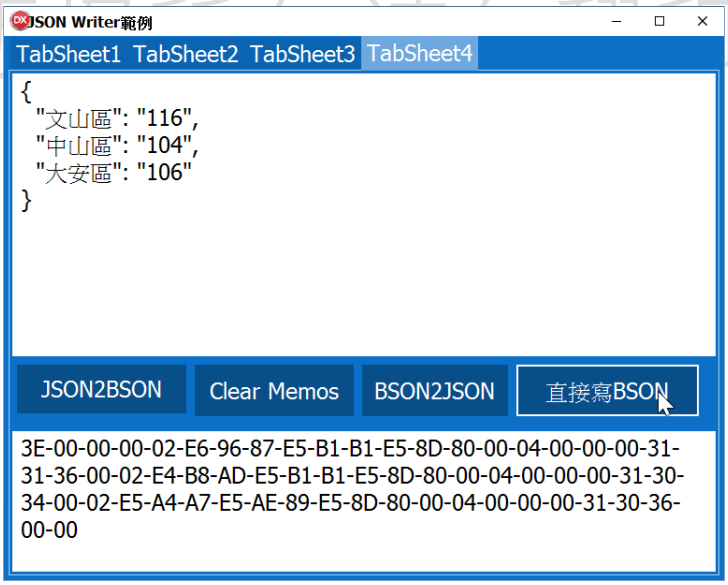

```
class PASCALIMPLEMENTATION TBsonReader : public
System::Json::Readers::TJsonReader
```

当然我们也可以直接把 JSON 格式的数据写入 TBsonWriter 对象中即可自动转成 BSON 格式，例如下面的程序代码直接呼叫 TBsonWriter 对象的各个 Write()方法把写成 BSON 格式：

```
001   void __fastcall TfmMainForm::Button12Click(TObject *Sender)
002   {
003     TBytesStream *Stream = new TBytesStream(NULL);
004     TBsonWriter *BsonWriter = new TBsonWriter(Stream);
005     try
006     {
007       BsonWriter->WriteStartObject();
008       BsonWriter->WritePropertyName(L"文山区");
009       BsonWriter->WriteValue(String(L"116"));
010       BsonWriter->WritePropertyName(L"中山区");
011       BsonWriter->WriteValue(String(L"104"));
012       BsonWriter->WritePropertyName(L"大安区");
013       BsonWriter->WriteValue(String(L"106"));
```

```
014       BsonWriter->WriteEndObject();
015       BsonWriter->Close();
016       Button11Click(Sender);
017       Stream->Position = 0;
018       Memo4->Lines->Text = Bson2Json(Stream->Bytes);
019       Memo5->Lines->Text =
Bytes2String( Json2Bson(Memo4->Lines->Text) );
020     }
021     __finally
022     {
023       delete BsonWriter;
024       delete Stream;
025     }
026   }
```

下面是藉由上面的程序代码把台北市邮政编码写成 BSON 格式之后显示出来再转回 JSON 格式。直接呼叫 TBsonWriter 类别中的方法也可正确无误的处理 JSON/BSON 格式的数据。



15-3-2 TBsonReader 类别

TBsonReader 类别在上一小节已经看到如何使用它了，也说明了它是从 TJsonReader 类别继承下来，因此我们我可以像使用 TJsonReader 类别的方式来使用它，只是它是使用来读取 BSON 格式数据的 Reader 类别。

TBsonReader 的建构元接受一个包含 BSON 数据的 TStream 类别对象：

```
__fastcall TBsonReader(System::Classes::TStream* const AStream,
bool AReadAsRootValuesArray);
```

之后我们就只要把它丢给 TJsonWriter 类别对象，那么 TJsonWriter 类别对象就会把TBsonReader对象中的BSON数据写入TJsonWriter类别对象中。如果需要从头再次读取 BSON 格式数据，那么只需要呼叫它的 Rewind 方法即可：

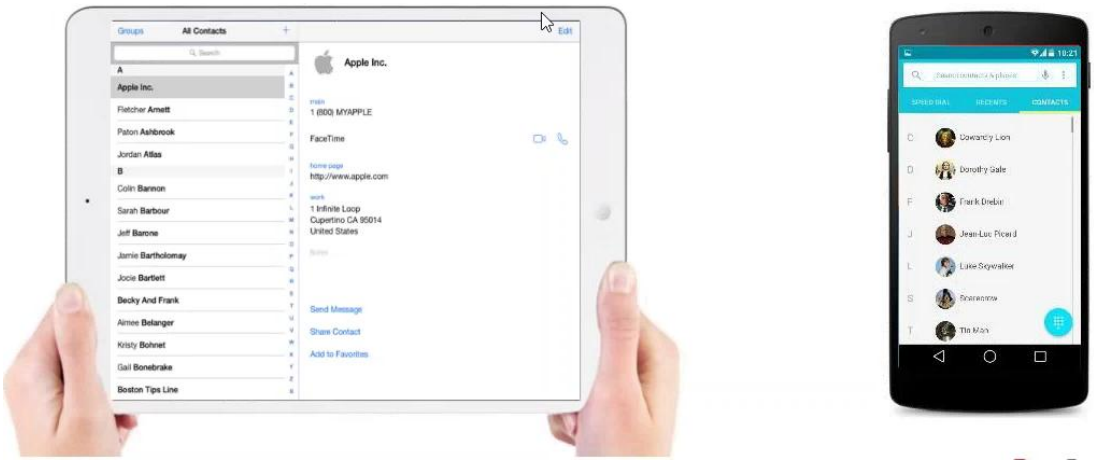
```
virtual void __fastcall Rewind(void);
```

由于上一小节已经介绍了如何使用 TBsonReader 类别，因此我们就不再赘述了。

16 新的 AddressBook 组件()

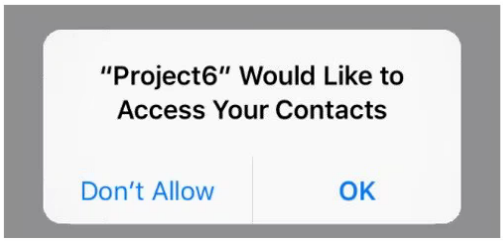
终于提供了许多开发人员多年来的要求，那就是一个跨平台的 TAddressBook 组件。

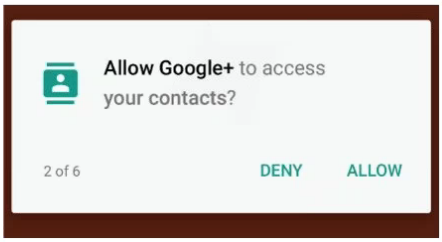
的新 TAddressBook 组件可以提供存取手机或平板的连络信息，并提供处理连络信息 CRUD 的能力，例如新增联系人，新增连络群组或是删除连络群组等功能:



如此一来开发人员就可以使用 TAddressBook 组件在 iOS 或是 Android 进行相关工作的开发。

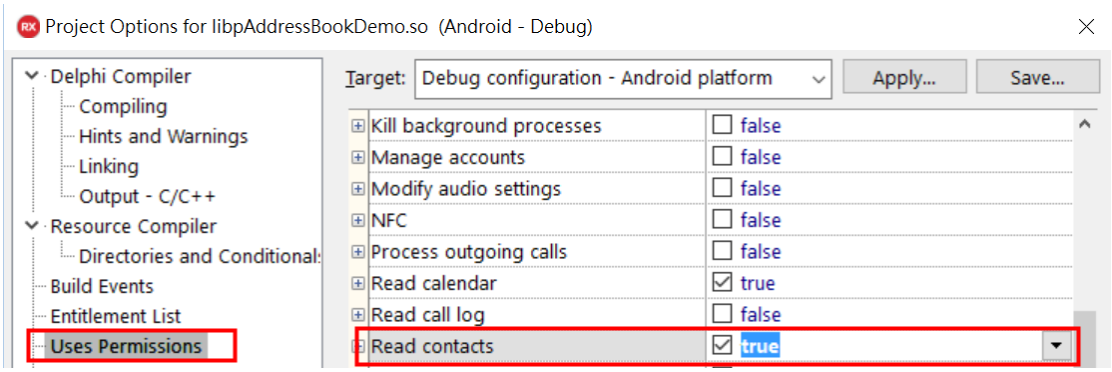
在使用 TAddressBook 组件存取联系人信息时开发人员需要设定访问权限否则无法存取联系人信息:





Permissions

在 C++Builder 中程序员需要设定 Users Permissions 中的 Read contacts 为 true：



接着呼叫 TAddressBook 组件的 RequestPermission()方法要求存取联系人的权限，之后会触发 TAddressBook 组件的 OnermissionRequest 事件：

```
void __fastcall
TfmMainForm::AddressBook1PermissionRequest(TObject *ASender,
const UnicodeString AMessage,
        const bool AAccessGranted)
{
}
```

OnermissionRequest 事件的 AAccessGranted 参数即代表使用者是否授权存取联系人信息，true 代表授权而 false 则代表否决存取。

完成访问权限设定之后就可以呼叫 TAddressBook 组件的 AllContacts()方法取得代表所有联系人信息的 TAddressBookContacts 对象：

```
  TAddressBookContacts* __fastcall AllContacts(void);
```

AllContacts()方法回传包含所有联系人信息的 TAddressBookContacts 对象：

```
class PASCALIMPLEMENTATION TAddressBookContacts : public
System::Generics::Collections::TObjectList__1<TAddressBookContac
t*>
{
  typedef
System::Generics::Collections::TObjectList__1<TAddressBookContac
t*>  inherited;
…
```

而其中每一个联系人信息则是由 TAddressBookContact 类别对象代表，请注意如下的 TAddressBookContact 类别宣告，TAddressBookContact 类别是一个抽象类：

```
class PASCALIMPLEMENTATION TAddressBookContact : public
System::TObject
{
  typedef System::TObject inherited;

protected:
  virtual System::UnicodeString __fastcall GetStringValue(const
Fmx::Addressbook::Types::TContactField AIndex) = 0 ;
  virtual void __fastcall SetStringValue(const
Fmx::Addressbook::Types::TContactField AIndex, const
System::UnicodeString AValue) = 0 ;
…
```


这代表程序员不应该直接建立 TAddressBookContact 类别对象，而应该藉由 TAddressBook 组件的 AllContacts()方法取得。

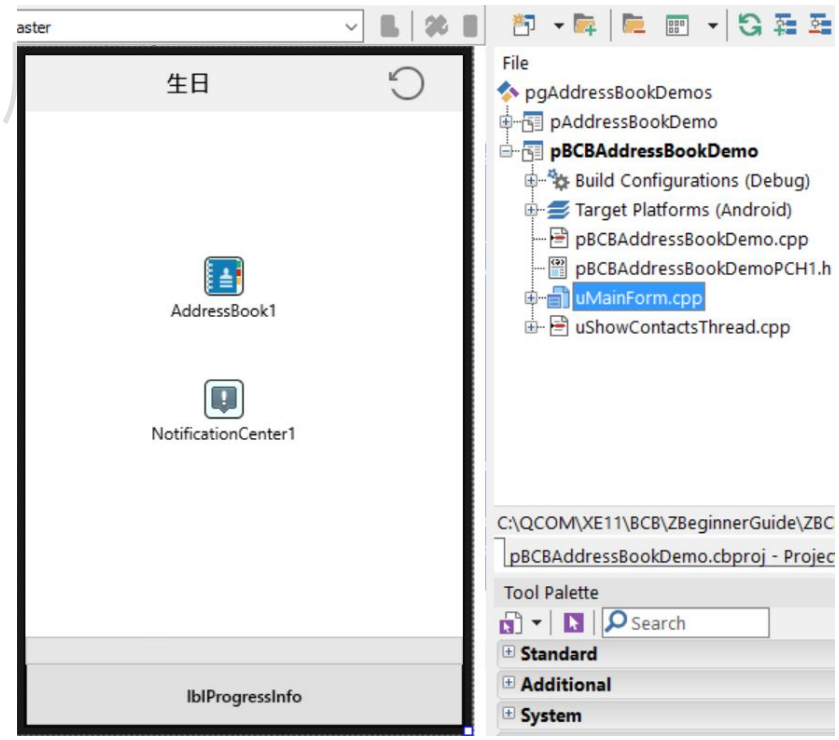
最后一旦取得了 TAddressBookContact 类别对象后，程序员就可以藉由存取它的特性值来取得联系人信息，例如联系人名称，联系人 Email 和联系人生日等重要的信息：

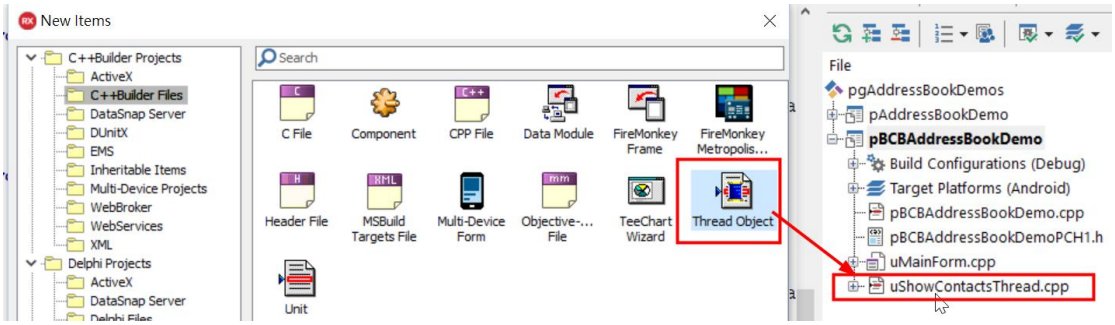
```
  __property System::UnicodeString DisplayName =
{read=GetDisplayName};
```

```
  __property Fmx::Addressbook::Types::TMultipleStringValues*
EMails = {read=GetListStringValue, write=SetListStringValue,
index=16};
  __property System::TDateTime Birthday = {read=GetDateTimeValue,
write=SetDateTimeValue, index=22};
…
```

另外在使用 TAddressBook 组件存取联系人信息时读者应具备的一个重要观念就是应该使用一个独立的线程来存取而不要使用主程序的线程，这是因为联系人信息可能数量众多而且手机中的联系人信息并不是储存在 RDBMS 的数据库中而是使用各自(iOS/Android)的架构储存。因此在存取大量联系人信息时速度可能不够快而拖累主程序的执行速度，而在 Delphi 中我们正好可使用一个额外的 TThread 类别对象来存取。

现在我们就可以说明如使用 TAddressBook 组件了，请先建立一个 Multi-Device 项目在主窗体中加入新的 TAddressBook 组件，接着再建立一个 TThread 类别 TShowContactsThread，如下所示：





首先在主窗体的 OnShow 事件中要求存取联系人信息的权限：

```
void __fastcall TfmMainForm::FormShow(TObject *Sender)
{
  AddressBook1->RequestPermission();
}
```

接着在 TAddressBook 组件的 OnPermissionRequest 事件中检查是否取得权限，如果是的话就呼叫 FillContactsInfo 方法存取所有联系人信息：

```
void __fastcall
TfmMainForm::AddressBook1PermissionRequest(TObject *ASender,
const UnicodeString AMessage,
        const bool AAccessGranted)
{
  if (AAccessGranted)
  FillContactsInfo();
  else
  lblProgressInfo->Text = u"用户不允许存取信息!";
}
```


FillContactsInfo 方法会建立前面建立的 TThread 衍生类别 TShowContactsThread 的对象并把使用 TAddressBook 组件取得的 TAddressBookContacts 对象传递给此线程对象，再启动执行此独立的线程：

```
void TfmMainForm::FillContactsInfo()
{
  delete FContacts;
  FContacts = AddressBook1->AllContacts();
  if ( (FThread != NULL) && (! FThread->Finished))
  {
  FThread->Terminate();
```

```
  FThread->WaitFor();
  }
  delete FThread;

  FThread = new TShowContactsThread(FContacts);
  FThread->OnContactLoaded = ContactedLoaded;
  FThread->OnStart = ContactLoadingBegin;
  FThread->OnTerminate = ContactLoadingEnd;
  FThread->Start();
}
```

TShowContactsThread 类别的建构元会储存主线程传递来的 TAddressBookContacts 对象：

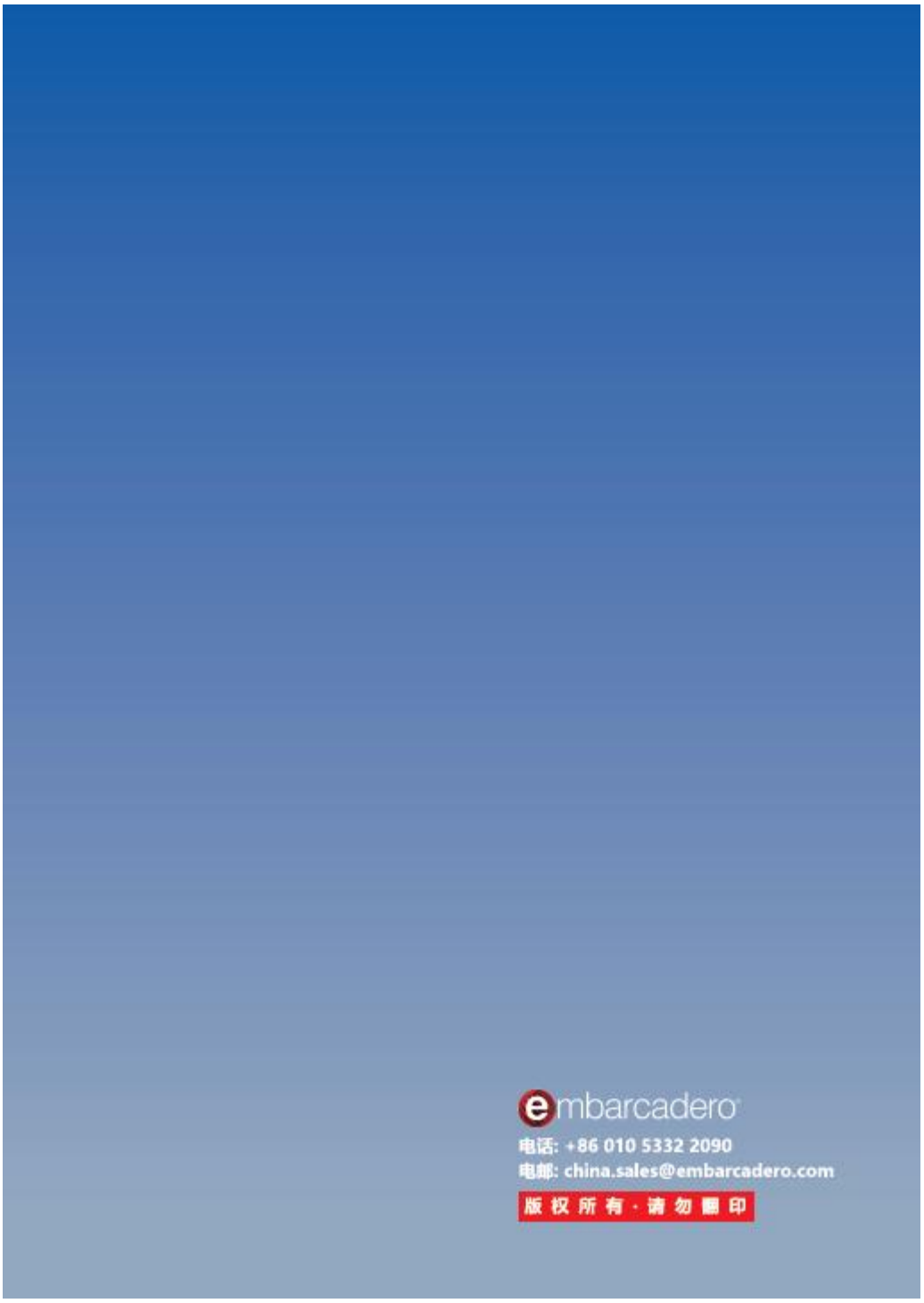
```
__fastcall
TShowContactsThread::TShowContactsThread(TAddressBookContacts
*AllContacts)
  : TThread(true)
{
  FAllContacts = AllContacts;
}
```


接着在 Execute()方法中一一取出 TAddressBookContacts 对象每一个代表联系人的 TAddressBookContact 对象，再呼叫 Synchronize()方法藉由匿 1 方法呼叫 DoContactedLoaded()方法显示每一个联系人信息：

```
void __fastcall TShowContactsThread::Execute()
{
  //---- Place thread code here ----
  Synchronize(DoStart);
  FIndex = 0;
  for (int iLoop = 0; iLoop < FAllContacts->Count; iLoop++)
  {
  FContact = FAllContacts->Items[iLoop];
  Synchronize(DoLoadContact);
  FIndex++;
  }
}
```

最后编译且执行此范例程序到 iOS 或是 Android 手机中就可以看到如下的执行结果画面，由于范例程序使用了一个独立的线程存取和显示联系人信息因此执行速度很理想也不会拖累主窗体的反应速度：

本书到此也告一段落了最后祝您使用 C++Builder 开发 iOS/Android App 愉快，现在您可以试着把前面 iOS 的捏泡泡 iOS App 转换成 Android 的版本，Have Fun！


embarcadero
电话: +86 010 5332 2090
电邮: china.sales@embarcadero.com
版 权 所 有 · 请 勿 翻 印