第四章
用 Visual Studio.net 來
操控 Java 虛擬機器

愚公移山的故事，是中國人自掃門前雪的最佳典範。
天神把山移到別人家門口，如果上天不買這家人的帳，
恐怕這個倒楣者和他的可憐後代都要一輩子在那裡移山了。
就算上天又感動了，則又是另外一家人倒楣的故事。

前言

在前一章中，筆者帶大家用 Borland C++ Builder，配合 JNI 技術，使得 Java 程式可以存取 Microsoft Office 所提供的軟體元件資源。其實熟悉 C++和 Windows Programming 的朋友一定知道，這樣的工作並非只有 Borland C++ Builder 才能做到，用 Microsoft Visual C++一樣可以完成相同的事情。

因此在本章中，我們將利用 Microsoft 所推出的 Visual Studio.NET 正式版，一樣配合 JNI 技術，撰寫可以操控 Java 虛擬機器，並且能夠使用 Java 類別函式庫的 C++程式碼。

為何會有這個主題的出現呢? 這是因為只要您使用上一章的技巧，就可以善用 Java ➔ JNI ➔ C++ 的方式來擴充 Java 應用程式的能力。但是反過來呢? 有些已經用 Java 撰寫好的類別函式庫，再用 C++重新撰寫划不划算? 如果可以在 C++中重複使用 Java 程式碼，也是一件重複使用軟體元件的典範。

.NET 和 Java 的戰爭似乎有水火不容的趨勢，而 Microsoft 於在 2002 年 1 月發表 Visual Studio.NET 正式版，似乎正式地點燃了戰火。但是在本章中，我們利用 Visual Studio.NET 來開發 C++程式，並藉此操控 Java 2 SDK/JRE 1.3.x 版或 Java 2 SDK/JRE 1.4.x 版內含的 Java 虛擬機器(Java Virtual Machine)，當我們控制了 Java 虛擬機器，我們就能夠在 C++之中重複使用已經用 Java 撰寫完成的類別函式庫。有了這樣的經驗，您將發現，兩家子在某種程度上還是可以相處的非常愉快。

當然，要讓.NET 和 Java 融合在一起運作，至少要讓 C#撰寫的程式碼可以存取 Java 所撰寫的程式碼(managed code 存取 byte code)，或是讓 Java 撰寫的程式碼可以使用 C#所撰寫的程式碼(byte code 存取 managed code)才算真正的融合在一起。這兩個議題在本書中完全沒有涉獵，筆者會在本書的下一版之中告訴各位該如何完成這兩件事情。所以您幾乎可以認定，本章的標題只是個幌子，但希望這是個兩大陣營大和解時代來臨的第一步。

簡介

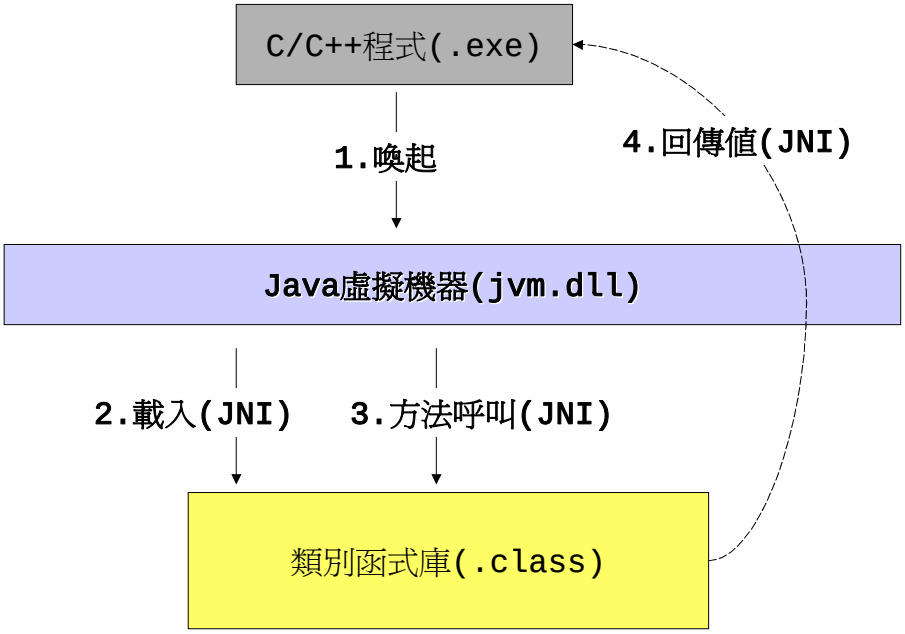
想像有一天，您自己用 Java 撰寫了底下這樣一個簡單的工具函式庫:

檔案:MyToolkit.java

```
1  public class MyToolkit
2  {
3      public static void function1(String param)
4      {
5          System.out.println("You input:" + param) ;
6      }
7  }
8
```

您會不會希望可以從 C++程式碼中重複使用這段程式碼呢? 本文的目的就就是教您如何利用 C++來操控 Java 虛擬機器，再利用 Java 虛擬機器來載入我們所需要的類別函式庫，然後使用我們想重複利用的功能函式。其架構如下圖所示:

整體程序概觀

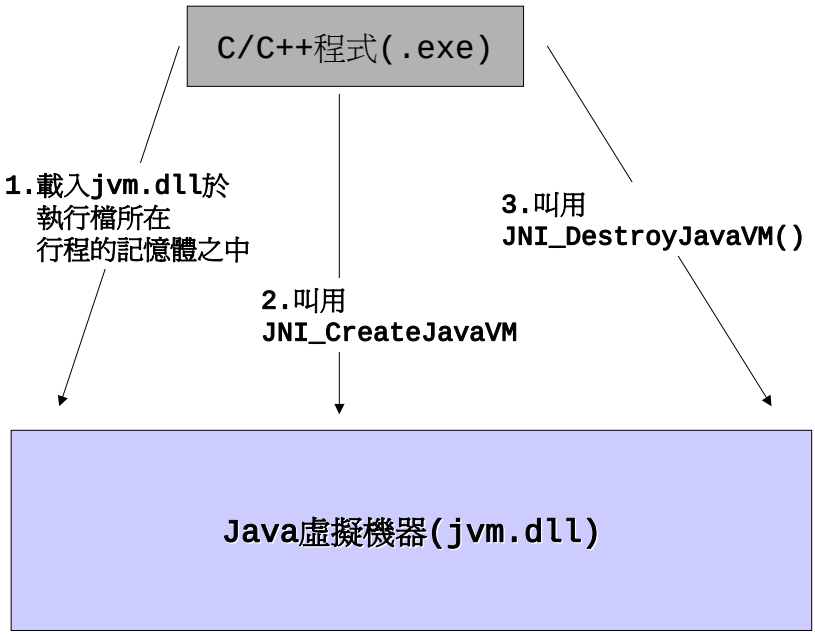


如上圖所示，整個程序的首要之先，就是先從用 C/C++所撰寫的程式碼之中，喚起 Java 虛擬機器。由於 Java 虛擬機器在 Windows 平台上是個動態聯結函式庫(.dll,即 dynamic linking library)，所以喚起程序的第一件事情，就是執行檔把 jvm.dll 這個動態聯結函式庫載入記憶體，並和我們的執行檔執行時所屬的行程(process)連接起來(attach)，如此一來我們才可以利用 jvm.dll 所開放出來的函式來操控 Java 虛擬機器。這個複雜的程序有兩種方法可以做到，第一種為 explicit 式，也就是在程式執行時期使用 Win32 API 裡頭的 LoadLibrary() 與 GetProcAddress()來做，第二種為 implicit 式，也就是在編譯時期就依靠標頭檔(.h)與符號表檔(.lib)來解決外部參考的問題。由於第一種比較有彈性，但是

比較不易除錯，所以本文採用第二種方式。因此，只要在編譯程式的時候讓編譯器可以找到標頭檔(.h)與符號表檔(.lib)兩種檔案即可。

當動態聯結函式庫接駁上我們的行程之後，我們就可以利用 JNI_CreateJavaVM()以在記憶體中建立一個 Java 虛擬機器的實體，然後我們就可以操控它來載入我們希望使用的類別函式庫。當我們不需要這個 Java 虛擬機器的時候，我們也要利用 JNI_DestroyJavaVM()來清除它。整個喚起 Java 虛擬機器的程序如下圖所示:

Java虛擬機器喚起程序

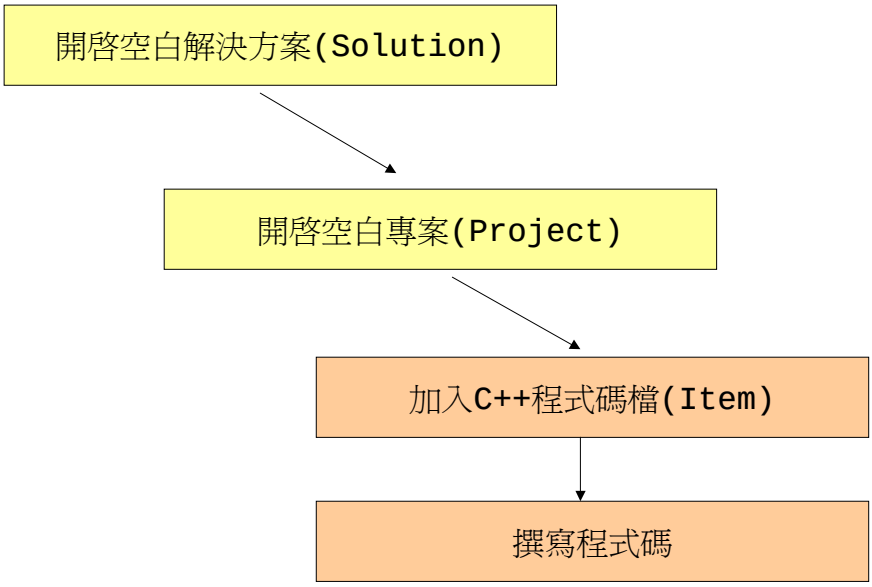


了解了整個架構之後，接下來我們就要開始進行程式碼的撰寫了。本文內容將採 step by step 的步驟，讓不熟悉工具操作的朋友也可以完成整個系統。

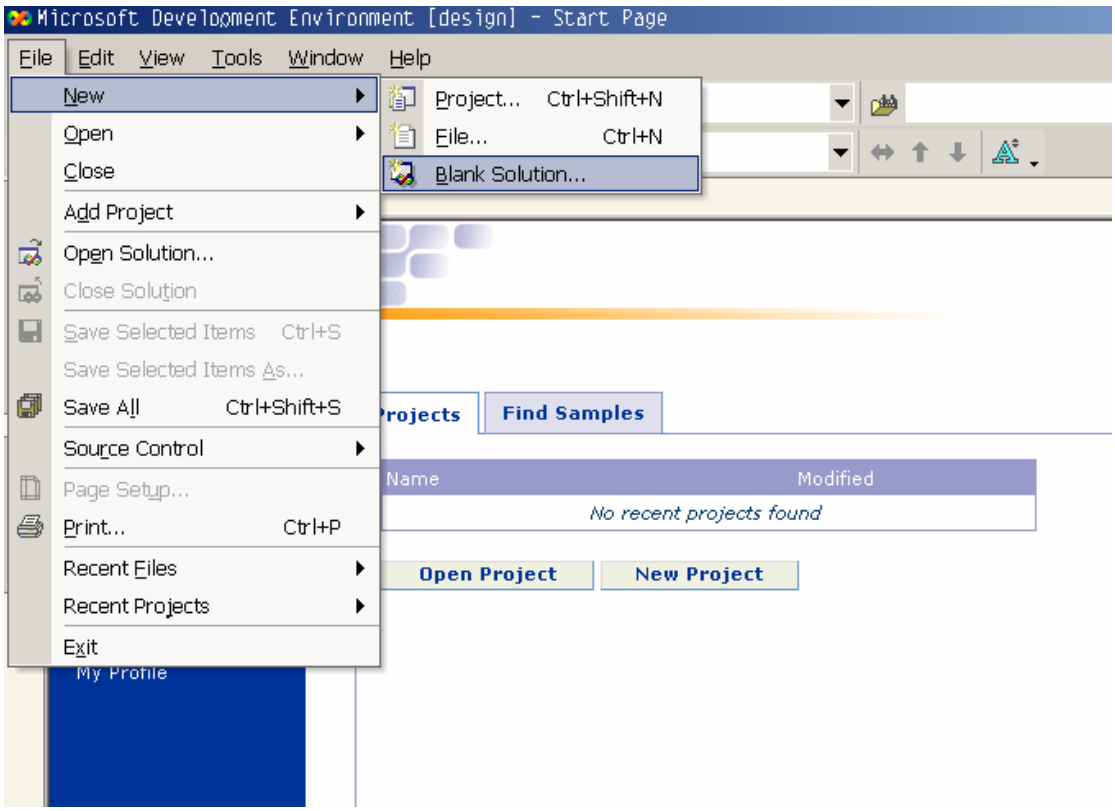
用 Visual Studio.NET 撰寫主程式

用 Visual Studio.NET 撰寫主程式的程序如下圖所示:

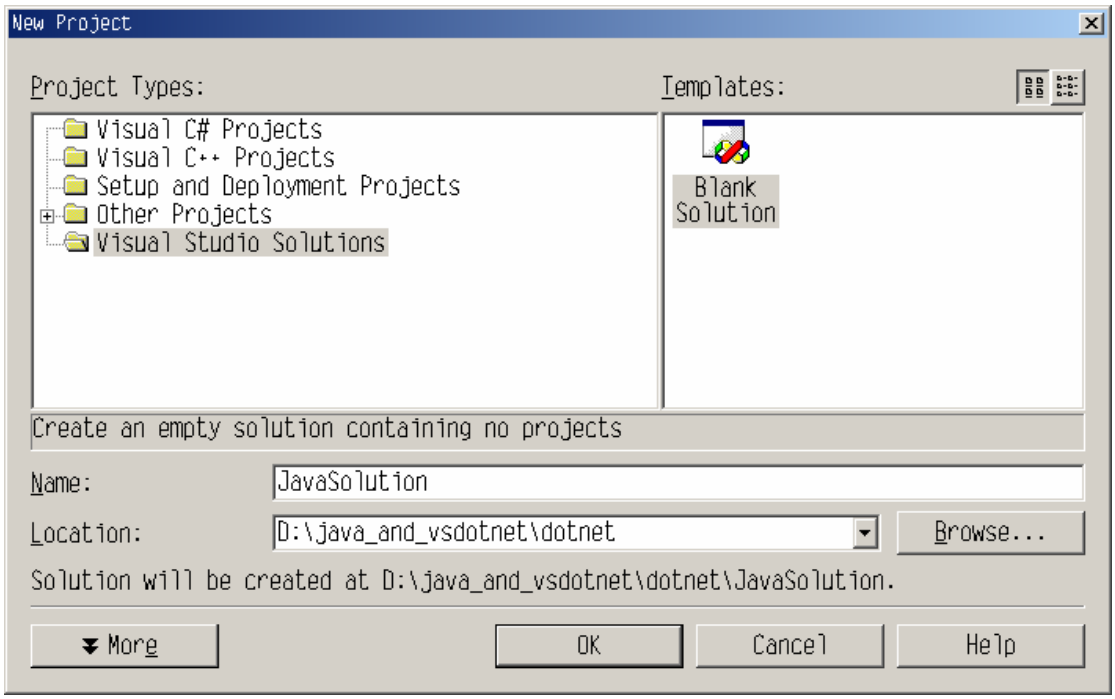
用Visual Studio.NET撰寫主程式之程序



如圖所示，首先我們必須先開啟一個空白的 Solution。請選擇主選單的 File / New 方/ Blank Solution，如下圖所示:

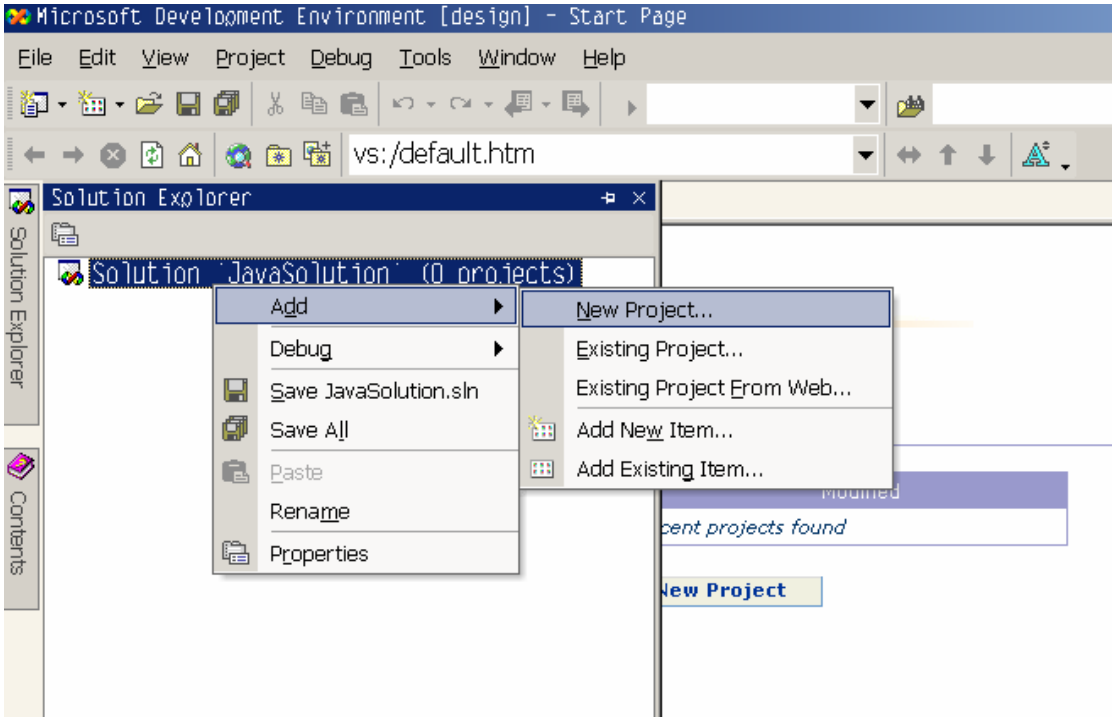


接著，在 Name 的地方打入解決方案(Solution)的名字(本文取名為 JavaSolution)，並在 Location 處填入您希望您的解決方案所儲存的位置，如下圖所示:

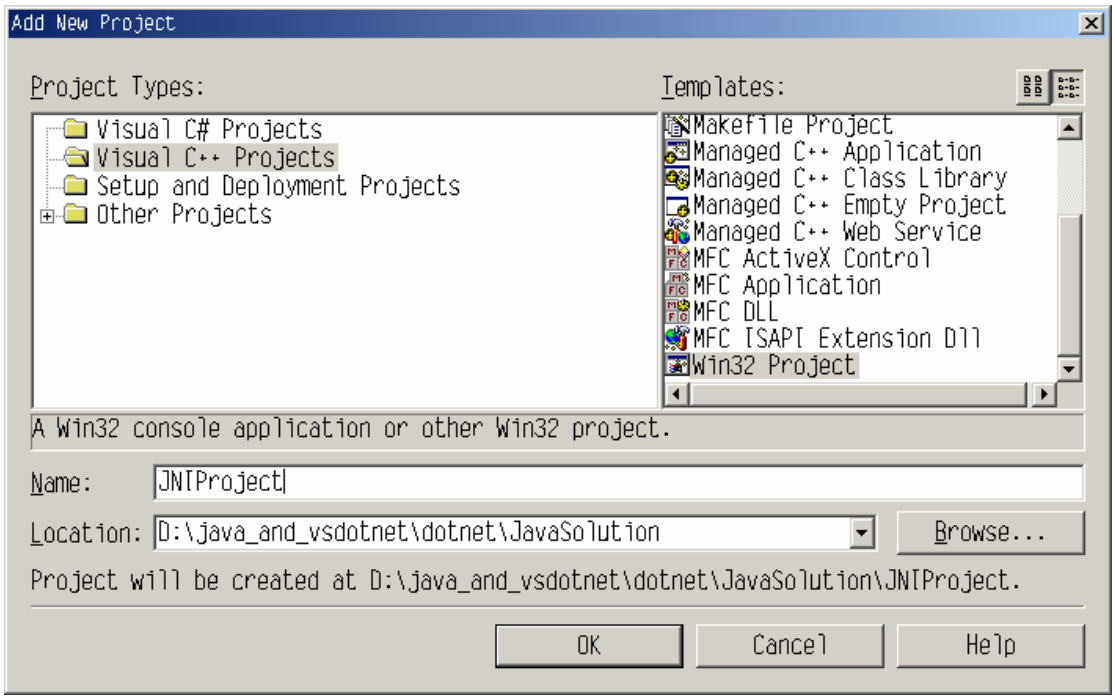


完成之後按下 OK 即可。

有了 Solution，我們才能加入專案(Project)。請開啟 Solution Explorer，在 Solution 的上方按下滑鼠右鍵以叫出內容選單，接著選擇 Add / New Project，如下圖所示。

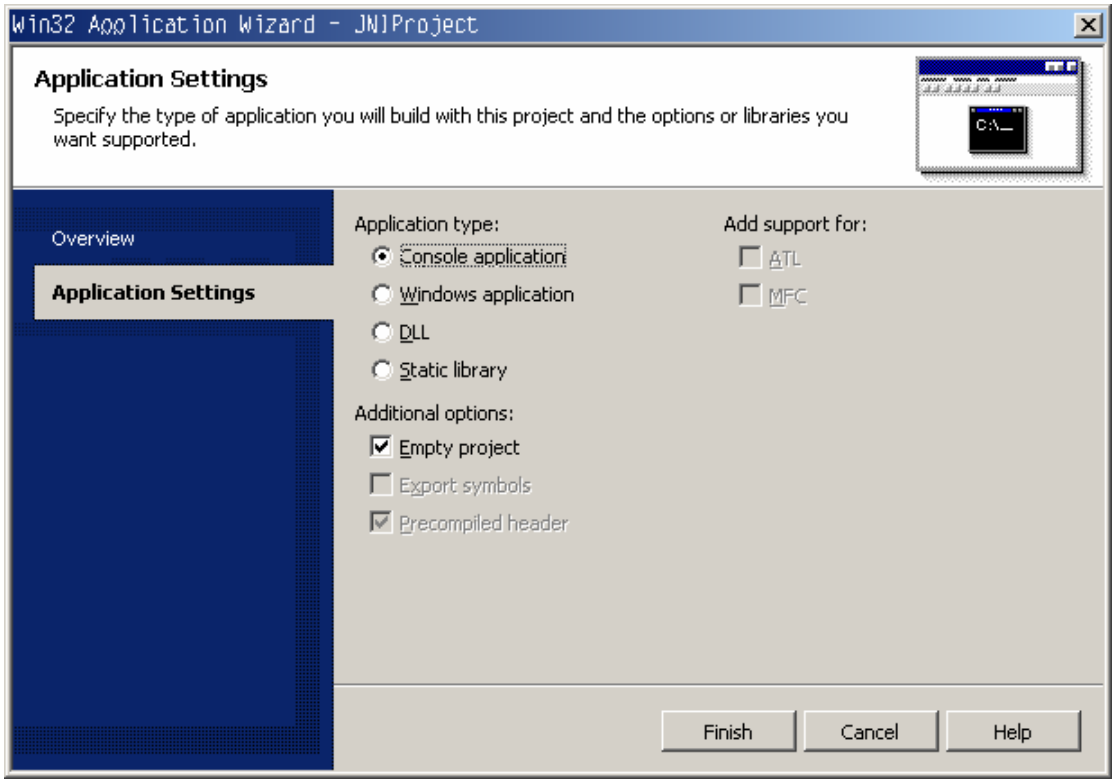


在 Add New Project 對話方塊中，請先在 Project Types 區點選 Visual C++ Projects，然後到 Templates 區選擇 Win32 Project，最後請在 Name 處填入 Project 名稱(本文使用 JNIProject)即可，如下圖所示:



完成之後，請按下 OK 鈕。

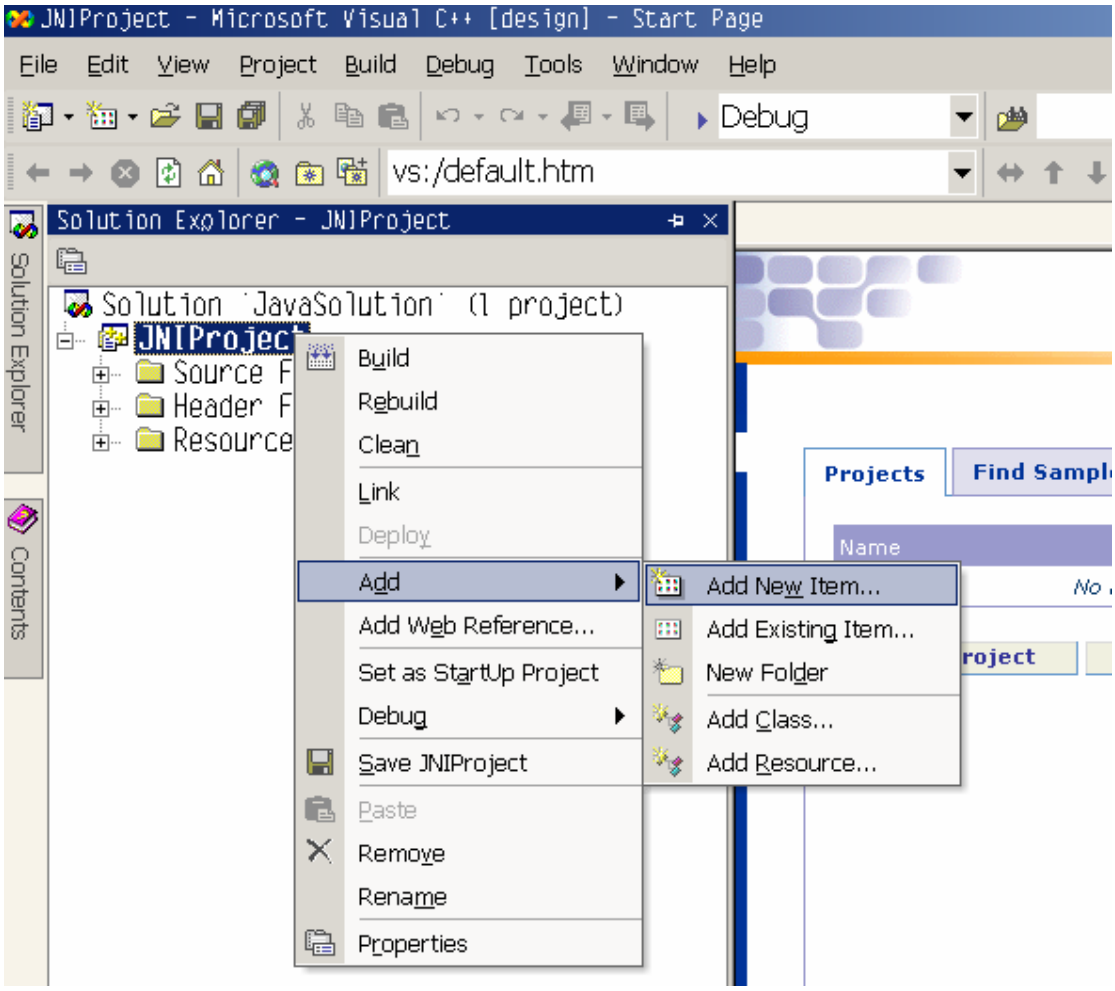
接下來，螢幕上會出現 Win32 應用程式精靈(Win32 Application Wizard)，請先選擇左方的 Application Settings，然後在 Application type 處選擇 Console application，並勾選 Additional options 中的 Empty project，如下圖所示:



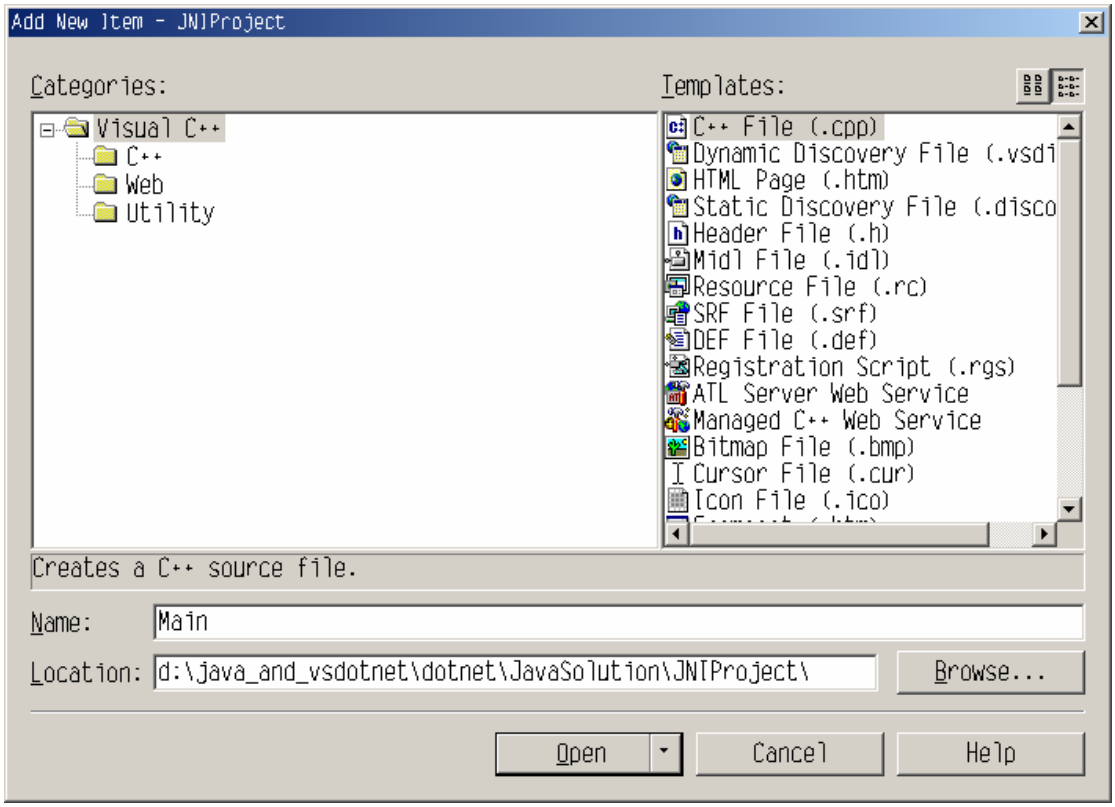
這是因為我們不希望 Visual Studio.NET 幫我們產生一堆亂七八糟的程式碼，所

以才設定成產生空白專案。

有了空白解決方案和空白專案之後，再來必須產生 C++程式碼檔(.cpp)，請開啟 Solution Explorer，在 Project 的上方按下滑鼠右鍵以叫出內容選單，點選 Add / Add New Item，如下圖所示:



出現 Add New Item 對話方塊之後，請在右邊的 Templates 處選擇 C++ File(.cpp)，然後在 Name 處輸入檔名(本文使用 Main.cpp)，最後按下 OK 鈕，就完成了 Visual Studio.NET 的專案建構程序，如下圖所示。



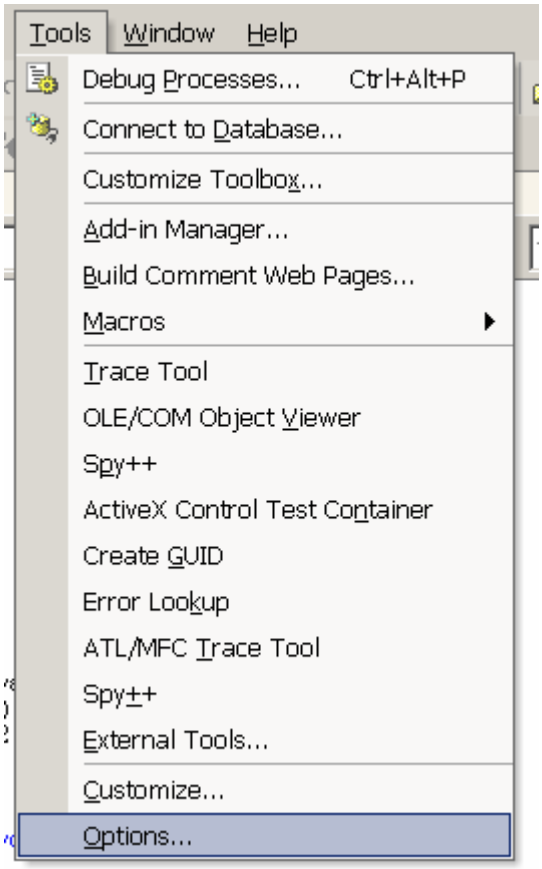
請雙擊 Solution Explorer 中的 Main.cpp，然後撰寫程式如下:

檔案:Main.cpp

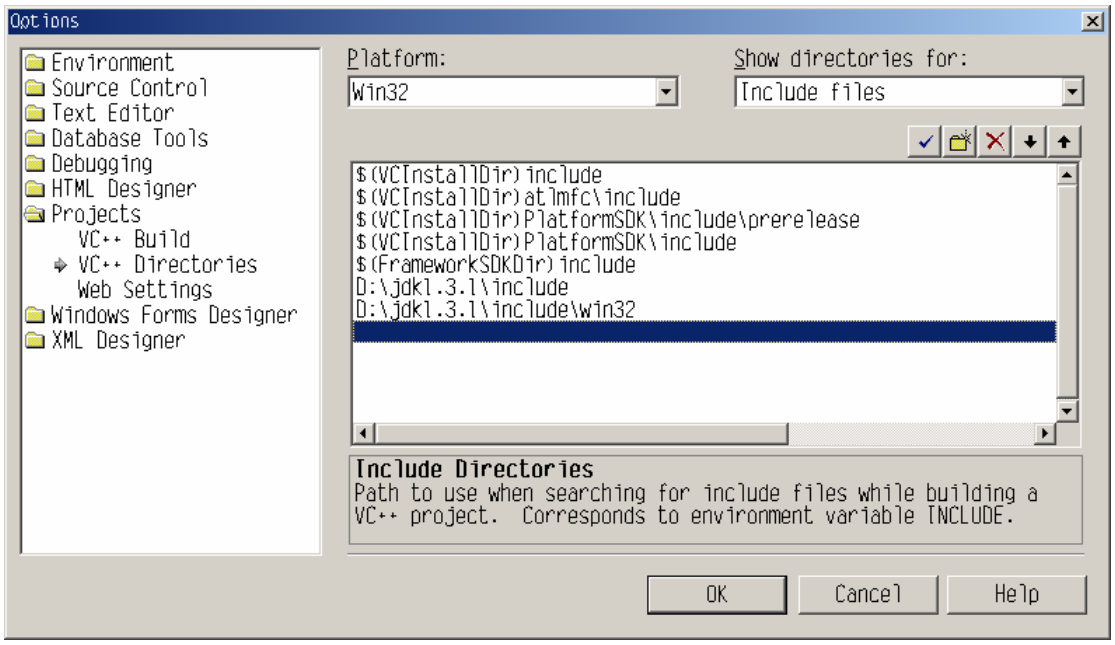
```
#include<iostream>
#include<jni.h>

using namespace std ;

int main()
{
	JavaVM* jvm ;
	JNIEnv* env ;
	JavaVMOption options[1] ;
	JavaVMInitArgs vmargs ;
	long status ;

	options[0].optionString = "-Djava.class.path=.";

	vmargs.version = JNI_VERSION_1_2;
	vmargs.options = options;
	vmargs.nOptions = 1;
```

```
        status = JNI_CreateJavaVM(&jvm,(void**)&env,&vmargs) ;
        if(status != JNI_OK)
        {
                cout << "Java虛擬機器建立失敗/不知名的錯誤" << endl ;
                cout << "錯誤代碼 : " << status ;
                return 1 ;
        }
        cout << "Java虛擬機器建立成功" ;

        jvm->DestroyJavaVM();
        return 0 ;

}
```

完成之後，請按下 Visual Studio.NET 選單中的 Build / Build Solution 來建造整個解決方案。一開始您會先遇到編譯找不到標頭檔 jni.h 的錯誤訊息。要解決這個問題，請選擇 Visual Studio.NET 選單中的 Tools / Options 以叫出 Options 對話方塊，如下圖所示:

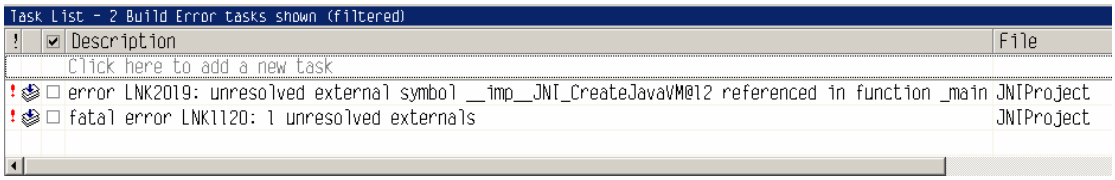


開啟 Options 對話方塊之後，請先選擇左方的 Projects 之下的 VC++ Directories。然後在 Show directories for 的地方點選 Include files，然後加入 **<jdk 安裝目錄>\include** 與 **<jdk 安裝目錄>\include\win32** 這兩個目錄，由於筆者的 Java 2 SDK 1.3.1 裝在 d:\jdk1.3.1 這個目錄下，所以必須指到 **d:\jdk1.3.1\include** 以及 **d:\jdk1.3.1\include\win32** (注意:jni.h 中參考到位於 **<jdk 安裝目錄>\include\win32** 底下的 jni_md.h，所以必須加入**<jdk 安裝目錄>\include\win32**，否則會產生找不到 jni_md.h 的錯誤訊息)，如下圖所示:

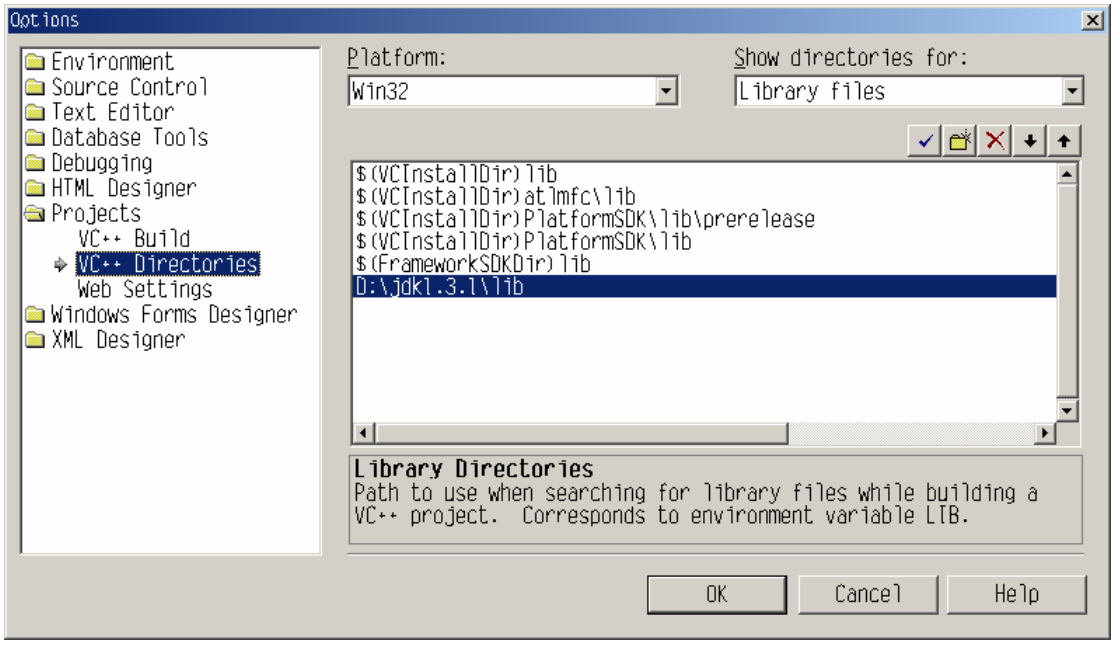


完成上述事項之後，請按下 OK 鈕，然後重新建造整個解決方案。

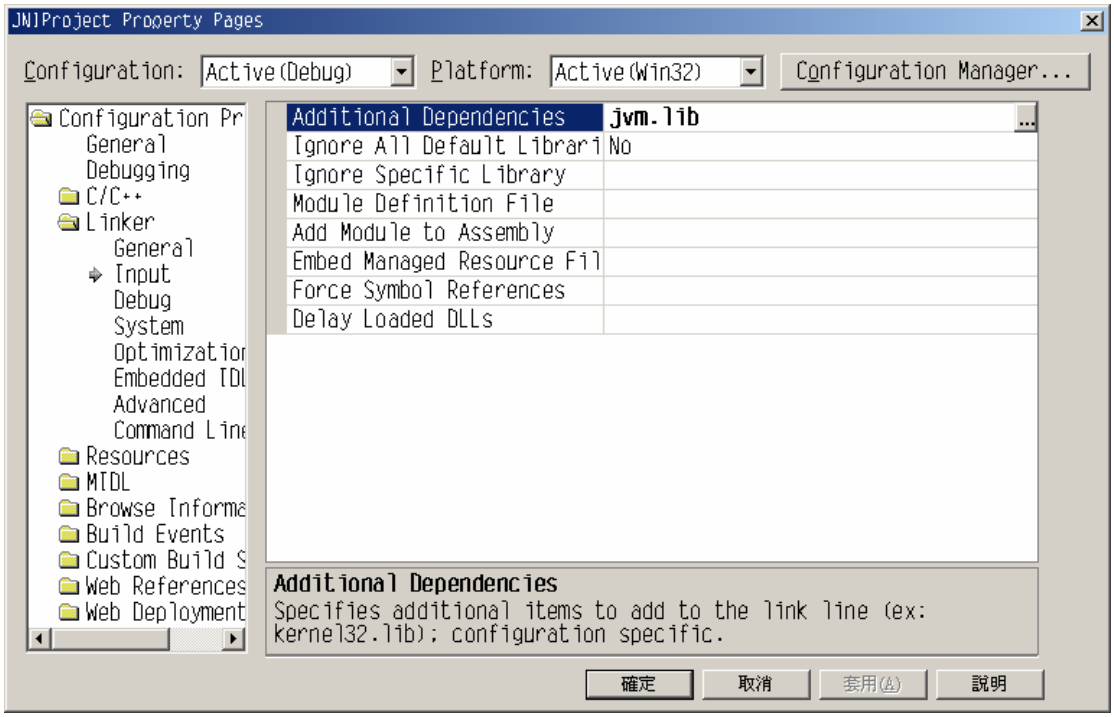
由於我們使用使用動態聯結函式庫的方式是使用 implicit 的方式，因此當編譯器完成程式的編譯程序，接著要進行連結程序(linking)的時候，就會產生錯誤訊息，如下圖所示:



為了解決這個問題，我們必須還要讓聯結器(linker)可以找到.lib 檔才行。請選擇 Visual Studio.NET 選單中的 Tools / Options 以叫出 Options 對話方塊。開啟 Options 對話方塊之後，請先選擇左方的 Projects 之下的 VC++ Directories。然後在 Show directories for 的地方點選 Library files，再加入 **<jdk 安裝目錄>\lib** 這個子目錄，由於筆者的 Java 2 SDK 1.3.1 裝在 d:\jdk1.3.1 這個目錄下，所以必須指到 **d:\jdk1.3.1\lib** 才行。



但是光是設定這裡，只能告訴連結器去哪裡找 lib，這是不夠的，因為連結器還需要知道使用哪一個.lib 檔來解決外部參考才行。所以請重新開啟 Solution Provider，在 Project 的上方按下滑鼠右鍵以叫出內容選單，點選 Properties。叫出 JNIProject Property Pages 對話方塊之後，請先點選左方 Linker 底下的 Input，然後在右邊 Additional Dependences 的地方加入 jvm.lib，如下圖所示:



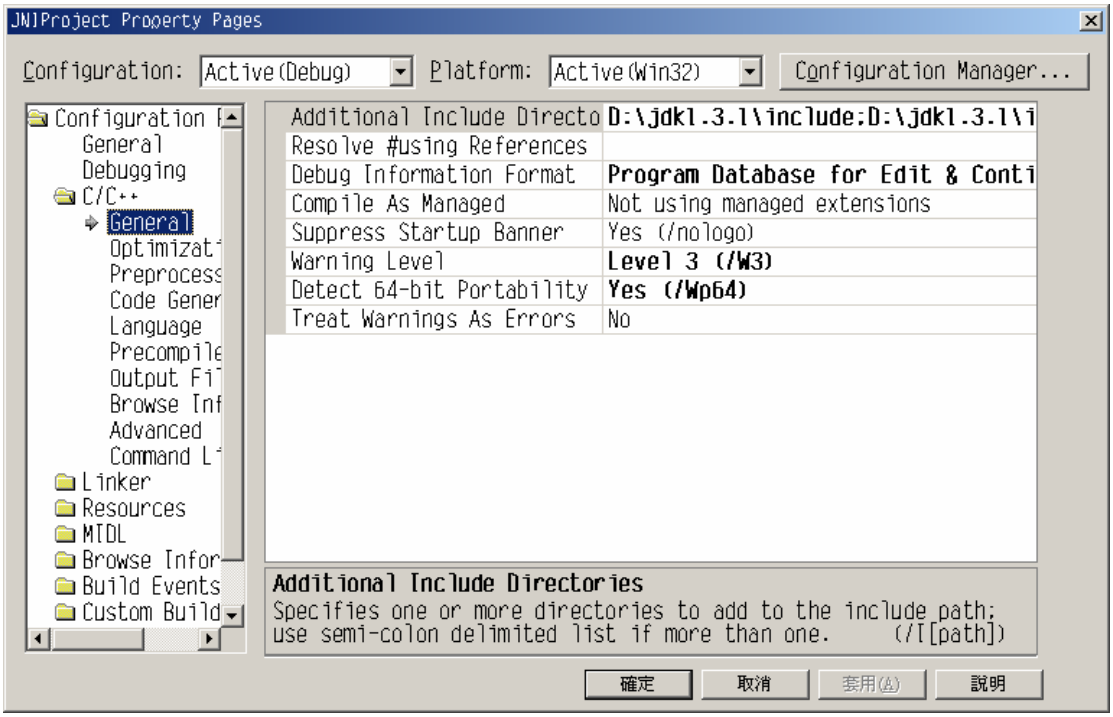
完成上述事項之後，請按下 OK 鈕，然後重新建造整個解決方案。如此就可以順利地造出可執行檔。

注意:

上述加入引入檔(.h)和符號表檔(.lib)參考路徑的設定，其實也可以在 JNIProject Property Pages 對話方塊中的選項來設定。

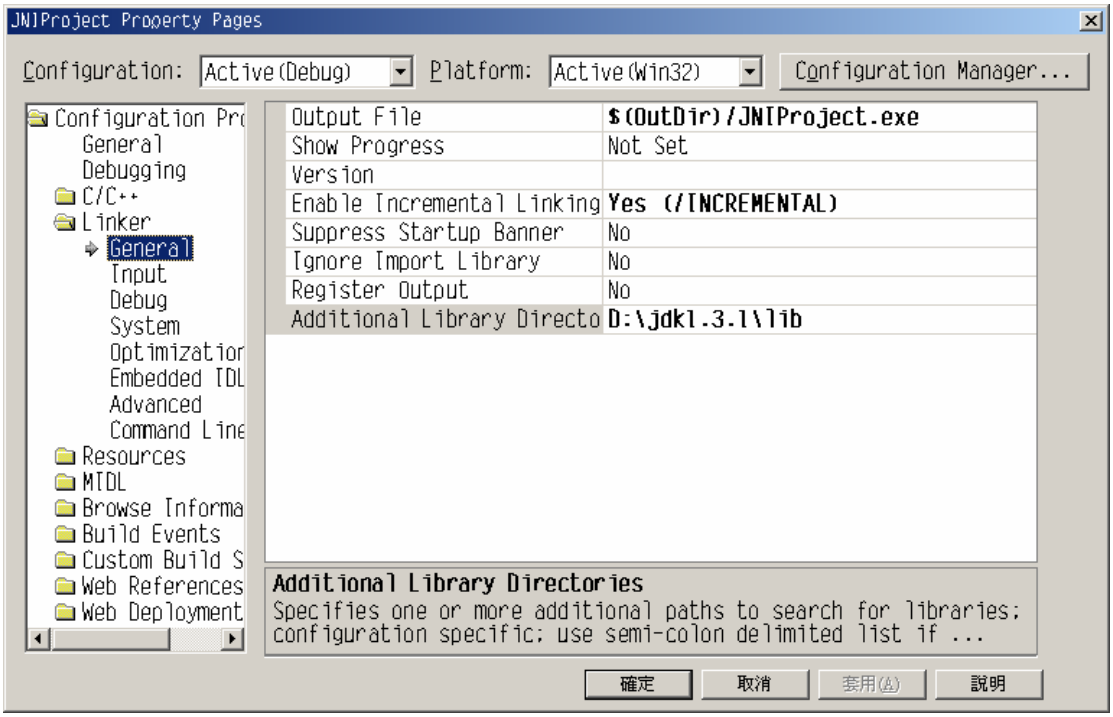
加入引入檔參考路徑:

在 C/C++底下的 General 中的 Additional Include Directories 裡。



加入符號表檔參考路徑:

在 Linker 底下的 General 中的 Additional Library Directories 裡頭:

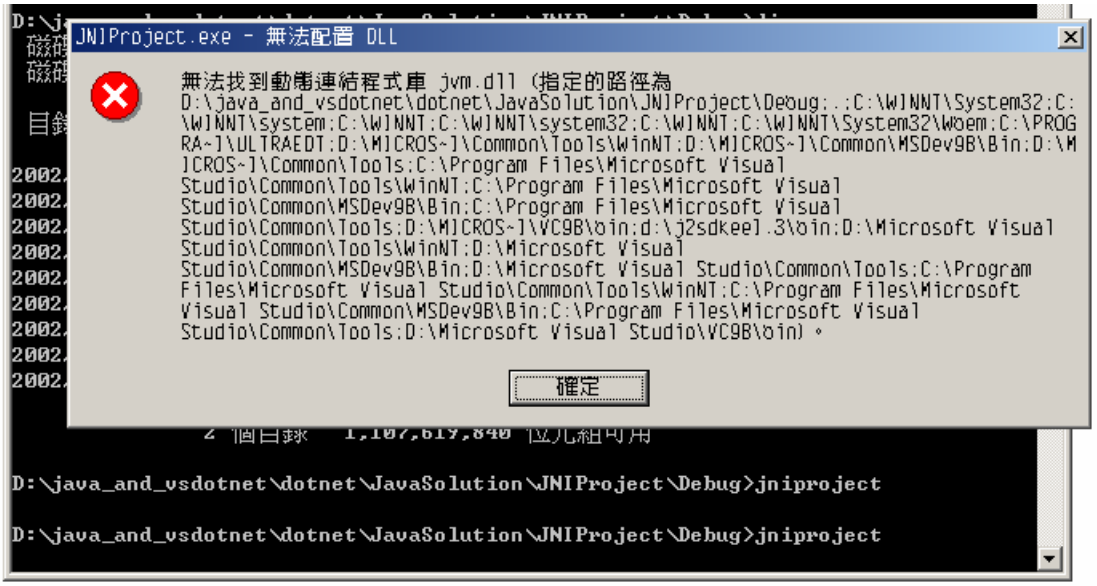


這種設定方式和從 Visual Studio.NET 系統選單 Tools / Options 叫出 Options

對話方塊來設定的差別在於，使用 Tools / Options 叫出 Options 對話方塊來設定會影響往後所有的專案，而在 JNIProject Property Pages 的設定只會影響到目前這個專案(即:JNI Project)。

無法喚起 Java 虛擬機器

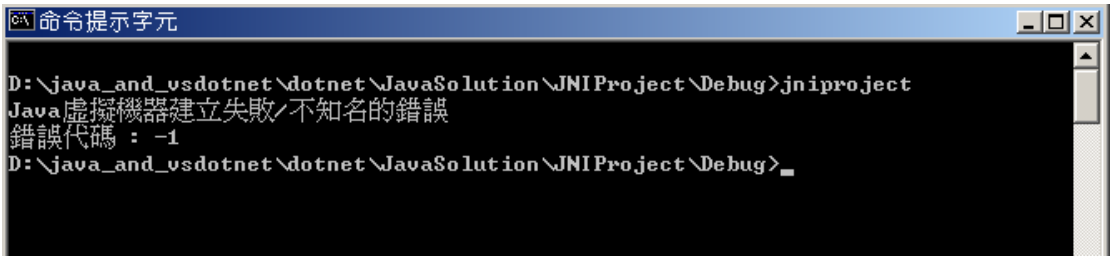
執行檔建造完成之後，您一定會急著想要執行看看，一執行，就會產生錯誤訊息如下:



這是因為使用 implicit 的方式來連接動態聯結函式庫的程式，都會在一開始執行時找出動態聯結函式庫。前面我們說過，Java 虛擬機器被放置在 jvm.dll 裡頭，可是我們的執行檔找遍了所有的設定路徑(環境變數 PATH 的設定)，就是找不到 jvm.dll，因此產生了上述的錯誤訊息。那麼您不禁要問:jvm.dll 到底藏在哪裡?

如果您是一位經驗豐富的 Windows Programmer，您一定會使用 Windows 的搜尋功能找尋 jvm.dll，您將可以在您的電腦中發現好幾個 jvm.dll (隨個人機器的不同而不同，但是如果您安裝的 Java 2 SDK 1.3.1，則應該至少會發現 4 個左右，但是如果您安裝的 Java 2 SDK 1.4.0 RC，則應該至少會發現 3 個左右，)。

此時問題來了，我們該使用哪一個 jvm.dll 呢? 每個大小都不同，也放在不同目錄底下。如果您想用暴力法一個一個來 try and error，我們把每個 jvm.dll 依序拷貝到與我們的執行檔同一個目錄下，執行之後，雖然不會出現找不到 jvm.dll 的錯誤訊息了，可是您的命令列模式下永遠出現底下錯誤訊息:



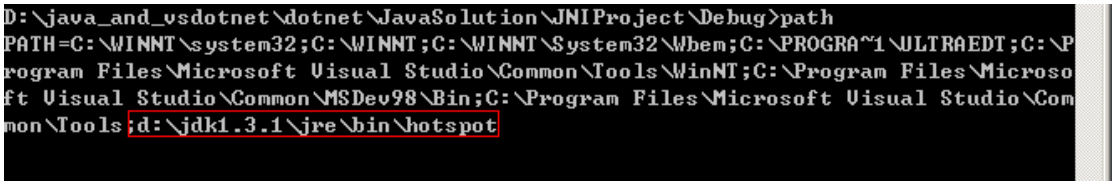
這下可好了，連 Java 虛擬機器都無法喚起，更不要說重複使用 Java 的類別函式庫了。這個問題該怎麼解決呢?

順利喚起 Java 虛擬機器

請您回頭複習我們在第一章所提到的概念。在第一章中，我們可以知道 java.exe 用何種方式找到 JRE，然後再選擇所使用的 Java 虛擬機器(jvm.dll)。其實，jvm.dll 無法單獨存在，當 jvm.dll 啟動之後，會使用 explicit 的方式(即使用 Win32 API 之中的 LoadLibrary()與 GetProcAddress())來載入輔助用的動態聯結函式庫，而這些輔助用的動態聯結函式庫都必須位於 jvm.dll 所在目錄的父目錄之中。這也是為什麼之前單單把 jvm.dll 拷貝到我們的執行檔所在目錄卻依然發生錯誤的原因。那麼，您是不是開始想著:"除了把 jvm.dll 拷貝到我們執行檔的所在目錄之外，也把那些位於**<JRE 所在目錄>\bin** 底下的輔助用動態聯結函式庫都拷貝到執行檔所在目錄的父目錄"呢? 這樣做雖然可行，可是實在太麻煩!! 最簡單的方式，就是讓環境變數 PATH 指向 JRE 所在目錄底下的 jvm.dll，這樣一來就不用如此大費周章。

假設我們要使用 Hotspot Client Virtual Machine，我們只要把 PATH 設定中加入 **<JRE 所在路徑>\bin\hotspot** 即可，筆者想用 JDK 底下的那組 JRE，而筆者的 JDK 又灌在 d:\jdk1.3.1\底下，所以要執行指令:

path=%path%;d:\jdk1.3.1\jre\bin\hotspot



注意:
如果您拷貝了 jvm.dll 到執行檔所在目錄，此時請記得刪除它。因為 Windows 作業系統會以執行檔所在目錄所找到的動態聯結函式庫為優先。

設定完成之後，請重新執行我們的程式，您就可以看到 Java 虛擬機器被成功地喚起了，如下圖所示:

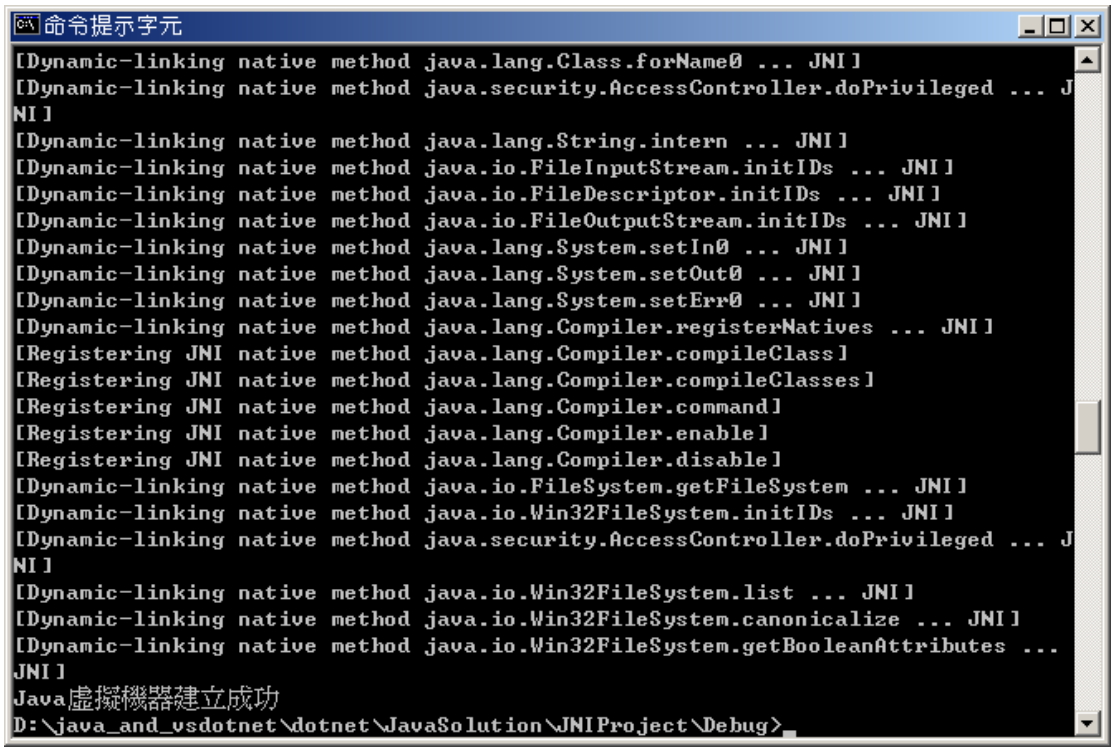
如果您希望可以看到整個喚起 Java 虛擬機器的詳細資料，請將程式修改如下:

檔案:Main.cpp

```
#include<iostream>
#include<jni.h>

using namespace std ;

int main()
{
	JavaVM* jvm ;
	JNIEnv* env ;
	JavaVMOption options[2] ;
	JavaVMInitArgs vmargs ;
	long status ;

	options[0].optionString = "-Djava.class.path=.";
	options[1].optionString = "-verbose:jni";

	vmargs.version = JNI_VERSION_1_2;
	vmargs.options = options;
	vmargs.nOptions = 2;



	status = JNI_CreateJavaVM(&jvm,(void**)&env,&vmargs) ;
	if(status != JNI_OK)
	{
		cout << "Java虛擬機器建立失敗/不知名的錯誤" << endl ;
		cout << "錯誤代碼 : " << status ;
		return 1 ;
	}
	cout << "Java虛擬機器建立成功" ;

	jvm->DestroyJavaVM();
	return 0 ;
```

```
}
```

您的命令列視窗將輸出許多 jvm.dll 連結輔助用動態聯結函式庫的相關訊息，如下圖:



(注:使用 –verbose:jni 時，可以加上 –Xcheck:jni 以取得更多進階資訊)

叫用 Java 類別函式庫

最後，這些雜七雜八的問題都搞定了，我們可以開始叫用 Java 類別函式庫，程式如下:

檔案:Main.cpp

```
#include<iostream>
#include<jni.h>

using namespace std ;

int main()
{
	JavaVM* jvm ;
	JNIEnv* env ;
	JavaVMOption options[1] ;
	JavaVMInitArgs vmargs ;
```

```
long status ;

options[0].optionString = "-Djava.class.path=D:\ \java_and_vsdotnet\ \java";

vmargs.version = JNI_VERSION_1_2;
vmargs.options = options;
vmargs.nOptions = 1;



status = JNI_CreateJavaVM(&jvm,(void**)&env,&vmargs) ;
if(status != JNI_OK)
{
        cout << "Java虛擬機器建立失敗/不知名的錯誤" << endl ;
        cout << "錯誤代碼 : " << status << endl ;
        return 1 ;
}
cout << "Java虛擬機器建立成功" << endl;

jclass toolclass ;
toolclass = env->FindClass("MyToolkit") ;
if(toolclass == NULL)
{
        cout << "找不到MyToolkit類別" ;
        return 1 ;
}

jstring jstr = env->NewStringUTF("My first class!");



jmethodID function ;
function = env->GetStaticMethodID(toolclass,"function1","(Ljava/lang/String;)V") ;
if(function == NULL)
{
        cout << "找不到function1函式" ;
        return 1 ;
}
env->CallStaticVoidMethod(toolclass, function, jstr);
```

```
    jvm->DestroyJavaVM();
    return 0 ;

}
```

首先，請先將 java.class.path 指向 MyToolkit.class 的所在位置，如下圖所示:

```
JavaVMOption options[1] ;
JavaVMInitArgs vmargs ;
long status ;

options[0].optionString = "-Djava.class.path=D:\\java_and_vsdotnet\\java";

vmargs.version = JNI_VERSION_1_2;
vmargs.options = options;
vmargs.nOptions = 1;
```

其原因已經在前面的章節裡提過。

再來，我們要請虛擬機器載入 MyToolkit.class，方法如下:

```
jclass toolclass ;
toolclass = env->FindClass("MyToolkit") ;
if(toolclass == NULL)
{
    cout << "找不到MyToolkit類別" ;
    return 1 ;
}
```

由於 MyToolkit.class 裡面的方法為靜態方法(static method)，因此不用產生實體物件就可以直接使用。

名為 function1 的方法需要一個字串做參數，所以我們要產生一個 UTF 字串，方法如下:

```
jstring jstr = env->NewStringUTF("My first class!");
```

將類別載入之後，我們還必須指引 Java 虛擬機器找到類別裡的靜態方法，方法如下:

```
jmethodID function ;
function = env->GetStaticMethodID(toolclass,"function1","(Ljava/lang/String;)V") ;
if(function == NULL)
{
    cout << "找不到function1函式" ;
    return 1 ;
}
```

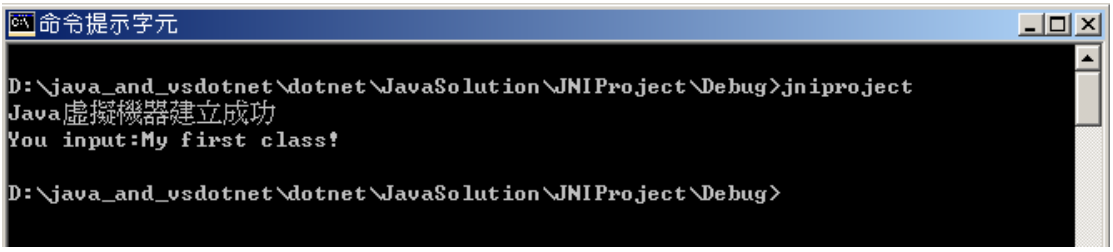
GetStaticMethodID 的最後一個參數為函式的傳回值和參數，如果您不曉得這個參數怎麼下，請使用 UltraEdit 開啟 MyToolkit.class 即可，如下圖:

```
00000000h: CA FE BA BE 00 03 00 2D 00 27 0A 00 0A 00 13 09 ; ■  ?..-.'......
00000010h: 00 14 00 15 07 00 16 0A 00 03 00 13 08 00 17 0A ; ................
00000020h: 00 03 00 18 0A 00 03 00 19 0A 00 1A 00 1B 07 00 ; ................
00000030h: 1C 07 00 1D 01 00 06 3C 69 6E 69 74 3E 01 00 03 ; .......<init>...
00000040h: 28 29 56 01 00 04 43 6F 64 65 01 00 0F 4C 69 6E ; ()V...Code...Lin
00000050h: 65 4E 75 6D 62 65 72 54 61 62 6C 65 01 00 09 66 ; eNumberTable...f
00000060h: 75 6E 63 74 69 6F 6E 31 01 00 15 28 4C 6A 61 76 ; unction1...(Ljav
00000070h: 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 3B 29 56 ; a/lang/String;)V
00000080h: 01 00 0A 53 6F 75 72 63 65 46 69 6C 65 01 00 0E ; ...SourceFile...
00000090h: 4D 79 54 6F 6F 6C 6B 69 74 2E 6A 61 76 61 0C 00 ; MyToolkit.java..
000000a0h: 0B 00 0C 07 00 1E 0C 00 1F 00 20 01 00 16 6A 61 ; .......... ...ja
000000b0h: 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 42 75 ; va/lang/StringBu
000000c0h: 66 66 65 72 01 00 0A 59 6F 75 20 69 6E 70 75 74 ; ffer...You input
000000d0h: 3A 0C 00 21 00 22 0C 00 23 00 24 07 00 25 0C 00 ; :..!."..#.$..%..
000000e0h: 26 00 10 01 00 09 4D 79 54 6F 6F 6C 6B 69 74 01 ; &.....MyToolkit.
000000f0h: 00 10 6A 61 76 61 2F 6C 61 6E 67 2F 4F 62 6A 65 ; ..java/lang/Obje
00000100h: 63 74 01 00 10 6A 61 76 61 2F 6C 61 6E 67 2F 53 ; ct...java/lang/S
00000110h: 79 73 74 65 6D 01 00 03 6F 75 74 01 00 15 4C 6A ; ystem...out...Lj
00000120h: 61 76 61 2F 69 6F 2F 50 72 69 6E 74 53 74 72 65 ; ava/io/PrintStre
00000130h: 61 6D 3B 01 00 06 61 70 70 65 6E 64 01 00 2C 28 ; am;...append..,(
00000140h: 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E ; Ljava/lang/Strin
00000150h: 67 3B 29 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 ; g;)Ljava/lang/St
00000160h: 72 69 6E 67 42 75 66 66 65 72 3B 01 00 08 74 6F ; ringBuffer;...to
00000170h: 53 74 72 69 6E 67 01 00 14 28 29 4C 6A 61 76 61 ; String...()Ljava
00000180h: 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 3B 01 00 13 ; /lang/String;...
00000190h: 6A 61 76 61 2F 69 6F 2F 50 72 69 6E 74 53 74 72 ; java/io/PrintStr
000001a0h: 65 61 6D 01 00 07 70 72 69 6E 74 6C 6E 00 21 00 ; eam...println.!.
000001b0h: 09 00 0A 00 00 00 00 00 02 00 01 00 0B 00 0C 00 ; ................
000001c0h: 01 00 0D 00 00 00 1D 00 01 00 01 00 00 00 05 2A ; ...............*
000001d0h: B7 00 01 B1 00 00 00 01 00 0E 00 00 00 06 00 01 ; ?.?...........
000001e0h: 00 00 00 01 00 09 00 0F 00 10 00 01 00 0D 00 00 ; ................
000001f0h: 00 36 00 03 00 01 00 00 00 1A B2 00 02 BB 00 03 ; .6........?.?.
00000200h: 59 B7 00 04 12 05 B6 00 06 2A B6 00 06 B6 00 07 ; Y?...?.*?.?.
00000210h: B6 00 08 B1 00 00 00 01 00 0E 00 00 00 0A 00 02 ; ?.?...........
00000220h: 00 00 00 05 00 19 00 06 00 01 00 11 00 00 00 02 ; ................
```

最後，萬事具備，只欠東風，我們呼叫該方法為我們服務，方法如下:

```
env->CallStaticVoidMethod(toolclass, function, jstr);
```

如果執行順利，螢幕會輸出底下畫面:



就這樣，我們完成了我們的目的。

■結論

在上一章中，筆者為大家說明了如何從 Java ➔ C++ ，而本章反過來說明如何從 C++ ➔ Java。 相信大家都已經見識到，其實使用 Java 來開發程式是不寂寞的。希望往後 Java 的愛好者對 Java 更有信心。