



白皮書

Delphi 及 Unicode

作者：Marco Cantù

2008 年 11 月

公司總部
100 California Street, 12th Floor
San Francisco, California 94111

EMEA 總部
York House
18 York Road
Maidenhead, Berkshire
SL6 1SF, United Kingdom

亞太地區總部
L7.313 La Trobe Street
Melbourne VIC 3000
Australia

簡介：DELPHI 2009 及 UNICODE

Delphi 2009 的其中一項最相關新功能是 Unicode 字元集的完整支援。特別針對英文及 26 個字母字元撰寫的 Delphi 應用程式適用於 Delphi 2009，針對全球其他語文撰寫的應用程式將因此一變革而受惠。

對於以西歐或南美洲語文撰寫用於特定地區設定的應用程式確實如此，對於以全世界其他語文撰寫的應用程式也有極大的助益。即使以英文撰寫應用程式，也可以考慮能夠更方便進行翻譯及本地化，而且，透過簡易、統一且好用的字元及，即可處理以任何語文撰寫的文字資料，包括資料庫備忘錄欄位，這些語文包括 Unicode 支援的阿拉伯文、中文、日文及古斯拉夫文等世界各地語文。

Windows 作業系統提供 API 層級的 Unicode 廣泛支援，而 Delphi 則填補空缺，並開啓銷售程式及開發特定新應用程式的新市場。

本白皮書將說明一些新概念及注意事項，可預期的是，變革將帶來許多機會。如果您需要提升相容性，仍然可以保留一部分程式碼使用傳統的字串格式。現在先不要談論過多的不同主題，先從頭開始切入。有一句話需要先提醒：Unicode 背後的概念以及 Delphi 2009 提供的某些新功能都需要時間才能瞭解，但是您可以先開始使用 2009，並且立即轉換現有的 Delphi 應用程式，不需要等到完全瞭解才進行。在 Delphi 2009 使用 Unicode 比想像中更容易！

什麼是 UNICODE ？

Unicode 是國際通用字元集的名稱，涵蓋現今與過去全世界的所有書寫字母以及其他符號。Unicode 也包含技術符號、標點符號，以及其他許多不屬於任何字母的書寫文字字元。Unicode 標準 (先前稱為 ISO/IEC 10646) 是由 Unicode Consortium 訂定完成，其中包含 100,000 個以上的字元。其主要網站位在：<http://www.unicode.org>。

採用 Unicode 是 Delphi 2009 的主要項目，並且解決其他多項問題。

Unicode 背後的概念 (易解之處) 是每一個字元都有各自的編號 (以就是 Unicode 術語所稱的字碼指標)。本文不計劃解說 Unicode 的整個理論，將擇要說明其中的重點。

UNICODE 轉換格式

Unicode 背後的困惑 (難解之處) 是透過實際儲存或實際位元組以多種方式表示相同的字碼指標 (或 Unicode 字元數值)。如果以簡易且一致的方式表示所有 Unicode 字碼指標只能使用以四位元組表示各個字碼指標 (在 Delphi 中, Unicode 字碼指標是以 `ucs4char` 資料類型表示), 大多數的開發人員這會耗用大量的記憶體及處理時間。

很少人知道相當常見的「UTF」一詞是「Unicode 轉換格式」的英文縮語。這些是屬於 Unicode 標準一部分的演算法對應, 可將各個字碼指標 (表示字元的絕對數值) 對應到表示特定字元的一組互不重複位元組。請注意, 對應可用於來回轉換不同表示的兩個方向。

此標準會定義三種編碼或格式, 這會視需要使用多少位元表示字元集的起始部分 (起始的 128 個字元): 8、16 或 32。值得注意的是, 所有三種編碼格式都需要最多 4 位元組的資料表示各個字碼指標。

- UTF-8 會將字元轉換成 1 至 4 位元組的可變長度編碼。UTF-8 常用於 HTML 及類似的通訊協定, 因為大多數的字元 (例如 HTML 的標記) 都涵蓋在 ASCII 子集中, 因此較為精簡。
- UTF-16 常用於許多作業系統 (包括 Windows) 及開發環境 (例如 Java 及 .NET)。由於大多數的字元都是 2 位元組, 因此相當精簡且處理速度快, 使用上相當簡便。
- UTF-32 會增加大量的處理量 (所有的字碼指標都具有相同的長度), 但是相當耗用記憶體, 因此實用性受到限制。

另一個與多位元組表示 (UTF-16 及 UTF-32) 相關的問題是先使用哪些位元組。根據標準, 可允許使用所有形式, 因此您可以使用 UTF-16 BE (位元組由大到小) 或 LE (位元組由小到大), 對於 UTF-32 也是如此。

位元次序標記

儲存 Unicode 字元的檔案通常使用稱為位元次序標記 (BOM) 的起始標頭作為簽章, 以指示使用的 Unicode 格式及位元順序形式 (BE 或 LE)。下表提供可以是 2、3 或 4 位元組的各種 BOM 摘要:

00 00 FE FF	UTF-32, 位元組由大到小
FF FE 00 00	UTF-32, 位元組由小到大
FE FF	UTF-16, 位元組由大到小
FF FE	UTF-16, 位元組由小到大
EF BB BF	UTF-8

用於 WIN32 的 UNICODE

從早期以來, Win32 API (可回溯到 Windows NT) 都含有 Unicode 字元的支援。大多數的 Windows API 函式都有兩種版本可用, 分別是以字母 A 標示的 ASCII 版本, 以及以字母 W 標示的寬字串版本。以下為 Delphi 2009 中 Windows.pas 的片段範例:

```

function GetWindowText(hwnd:HWND; lpString:PWideChar;
  nMaxCount:Integer):Integer; stdcall;
function GetWindowTextA(hwnd:HWND; lpString:PAnsiChar;
  nMaxCount:Integer):Integer; stdcall;
function GetWindowTextW(hwnd:HWND; lpString:PWideChar;
  nMaxCount:Integer):Integer; stdcall;

function GetWindowText; external user32
  name 'GetWindowTextW';
function GetWindowTextA; external user32
  name 'GetWindowTextA';
function GetWindowTextW; external user32
  name 'GetWindowTextW';

```

宣告部分都相同，不過分別使用 **PAnsiChar** 或 **PWideChar** 指涉字串。請注意，對於兩者而言，不含字串格式指示的一般版本只是預留位置；如上所示，在過去的 Delphi 版本一直是「A」版本，在 Delphi 2009 則預設為「W」版本。

CHAR 現在是 WIDECHAR

過去有一段時間，Delphi 曾經包含兩種代表字串的不同資料類型：

AnsiChar，以 8 位元表示 (共有 256 個不同的符號)，根據字碼頁進行解譯；

WideChar，以 16 位元表示 (共有 64,000 個不同的符號)。

在這方面，Delphi 2009 沒有經過任何變更。不同的是，過去是 **AnsiChar** 別名的 **Char** 類型現在是 **WideChar** 別名。每次編譯器在程式碼中發現 **Char**，就會解譯成 **WideChar**。請注意，沒有任何方法可變更這個新的編譯器預設設定。(對於字串類型而言，會以固定的硬式編碼方式將 **Char** 類型對應到特定的資料類型。開發人員曾經要求能夠切換的編譯器指示詞，但是這會對 QA、支援、封裝相容性等造成重大影響。不過，您仍然可以選擇轉換程式碼使用特定類型，例如 **AnsiChar**。)

這確實是一項會影響許多原始程式碼的變更，並且會出現許多複雜的狀況。例如 **PChar** 指標現在是 **PWideChar** 的別名，而非過去的 **PAnsiChar**。

CHAR 成爲序數類型

新的 *large* **Char** 類型仍然是序數類型，因此您可以在其中使用 **Inc** 及 **Dec**，並且以 **Char** 計數器撰寫 **for** 迴圈等。

| var

```

    ch:Char;
begin
    ch := 'a';
    Inc (ch, 100);
    ...
    for ch := #32 to High(Char) do
        str := str + ch;

```

唯一可能讓您感到些許困擾的是根據 Char 類型宣告字元集時：

```

var
    CharSet = set of Char;
begin
    charSet := ['a', 'b', 'c'];
    if 'a' in charSet then
        ...

```

在此狀況下，編譯器會假設您正在將現有程式碼移植到 Delphi 2009、決定考慮使 Char 成爲 AnsiChar (字元集最多可以有 256 個字元)，並發出警告訊息：

```

W1050 wideChar reduced to byte char in set expressions. Consider
using 'CharInSet' function in 'SysUtils' unit.

```

程式碼會按照預期的方式進行，但是並非所有現有的程式碼都能夠便於對應，因爲已經不可能再取得所有字元的字元集。如果這是您需要的方式，則必須變更您的演算法 (可能需要按照警告的建議方式進行)。

如果您想要隱藏警告 (編譯 5 行程式碼會出現兩項警告)，您可以撰寫：

```

var
    charSet:set of AnsiChar; // suppress warning
begin
    charSet := ['a', 'b', 'c'];
    if AnsiChar('a') in charSet then // suppress warning
        ...

```

以 CHR 進行轉換

另請注意，您可以使用 AnsiChar 或 WideChar 等類型，將數值轉換爲字元，也可以運用 Chr 編譯器魔法函式 (可視爲與 Ord 相反的函式) 來使用典型 Pascal 技巧。這項標準魔法函式已經過擴充，可將單字視爲參數，而非位元組。

與字元常值不同的是，對於 **Chr** 的呼叫現在會以 **Unicode** 解譯。因此，如果您將下列的程式碼：

```
| chr (128)
```

從 **Delphi 2007** 移植到 **Delphi 2009**，您會感覺驚訝。如果您改為使用 **#128**，則會獲得不同的結果，這會根據您的字碼頁而定。

32 位元字元

雖然預設 **Char** 類型現在是對應到 **WideChar**，但是值得注意的是，**Delphi** 也會將 **System** 單元中定義的 4 位元組字元類型 **UCS4Char** 定義為：

```
| type  
   UCS4Char = type Longword;
```

此一類型定義及對應的 **UCS4String** (經定義成為 **UCS4Char** 陣列) 已出現在 **Delphi 2007** 中，**Delphi 2009** 中 **UCS4Char** 資料類型的相關性則大量運用於多個 **RTL** 常式中，包括以下將討論的全新 **Character** 單元。

全新 CHARACTER 單元

為了更加支援新的 **Unicode** 字元 (以及 **Unicode** 字串)，**Delphi 2009** 引入 **Character** 的全新 **RTL** 單元。此單元會定義 **TCharacter** 密封類別，這基本上是靜態類別函式的集合，並包括多個對應於類別公共 (及某些私有) 函式的全域常式。

此單元也會定義兩項有趣的列舉型別。第一個列舉稱為 **TUnicodeCategory**，這會對應多個類型的不同字元，例如控制項、空間、大小寫字母、小數位數、標點符號及數學符號等。第二個列舉稱為 **TUnicodeBreak**，這會定義多種空間、連字號及中斷的系列。

TCharacter 密封類型具有 40 種以上的方法可處理獨立字元或字串內字元，以便：

取得字元的數值表示 (**GetNumericValue**)。

要求類別 (**GetUnicodeCategory**) 或根據多個類別 (**IsLetterOrDigit**、**IsLetter**、**IsDigit**、**IsNumber**、**IsControl**、**IsWhiteSpace**、**IsPunctuation**、**IsSymbol** 及 **IsSeparator**) 的其中一個進行檢查。

檢查大小寫 (**IsLower** 及 **IsUpper**) 或轉換大小寫 (**ToLower** 及 **ToUpper**)

驗證是否屬於 UTF-16 Surrogate 字組 (**IsSurrogatePair**、**IsSurrogate**、**IsLowSurrogate** 及 **IsHighSurrogate**) 的一部分

以 UTF-32 來回轉換 (**ConvertFromUtf32** 及 **ConvertToUtf32**)

全域函式幾乎就是這些靜態類別方法，其中某些對應於現有的 Delphi RTL 函式，只是名稱不同。許多基本 RTL 函式都能夠處理字元，另外有延伸版本會呼叫適當的 Unicode 碼。例如，您可以撰寫下列程式碼，嘗試將重音字母轉換為大寫字母：

```
var
  ch1:Char;
  ch2:AnsiChar;
begin
  ch1 := 'ù';
  Memo1.Lines.Add ('WideChar');
  Memo1.Lines.Add ('UpCase ù:' + UpCase(ch1));
  Memo1.Lines.Add ('ToUpper ù:' + ToUpper (ch1));

  ch2 := 'ù';
  Memo1.Lines.Add ('AnsiChar');
  Memo1.Lines.Add ('UpCase ù:' + UpCase(ch2));
  Memo1.Lines.Add ('ToUpper ù:' + ToUpper (ch2));
```

傳統的 Delphi 程式碼 (AnsiChar 版本的 **UpCase**) 只處理 ASCII 字元，因此無法轉換字元 (**UpperCase** 函式也是如此，只處理 ASCII，而 **AnsiUpperCase** 會處理 Unicode 的所有項目，不論名稱為何)。如果您對此傳遞 **WideChar**，運作方式也不會變更 (可能是因為向下相容所致)。**ToUpper** 函式的運作正常 (最終呼叫 Windows API 的 **CharUpper** 函式)。這是執行以上程式碼的輸出：

```
WideChar
UpCase ù:ù
ToUpper ù:Ù
AnsiChar
UpCase ù:ù
ToUpper ù:Ù
```

請注意，您可以保留現有的 Delphi 程式碼，而 **UpCase** 會呼叫 **Char**，並且保持標準的 Delphi 運作方式。

如果更加瞭解 **Characters** 單元引入的特定 Unicode 相關功能示範，請參閱以下程式碼，其中定義的字串包含 Unicode 字碼指標 \$1D11E，也就是音樂符號 *G* 譜號：

```
var
  str1:string;
```

```
begin
  str1 := '1.' + #9 + ConvertFromUtf32 (128) +
    ConvertFromUtf32($1D11E);
```

程式會針對字串的不同字元進行下列測試 (全部傳回 True) :

```
TCharacter.IsNumber(str1, 1)
TCharacter.IsPunctuation (str1, 2)
TCharacter.IsWhiteSpace (str1, 3)
TCharacter.IsControl(str1, 4)
TCharacter.IsSurrogate(str1, 5)
```

最後請注意，SysUtils 的 `IsLeadChar` 函式已經過修改，可處理 Unicode Surrogate 字組，以及用於移至字串下一個字元之類的其他相關函式。

關於字串及 UNICODESTRING

`Char` 類型的定義變更相當重要，因為這與字串類型的定義變更有關。與字元不同的是，字串會對應於先前不曾有的全新 `UnicodeString` 資料類型。如我們所見，其中的內部表示與典型 `AnsiString` 類型不同 (我使用「典型 `AnsiString` 類型」一詞來表示從 Delphi 2 到 Delphi 2007 都提供使用的字串類型；`AnsiString` 類型仍然是 Delphi 2009 的一部分，但是運作方式經過修改，因此在提及這個過去的結構時，我使用「典型 `AnsiString` 類型」一詞)。

語言中既然有 `WideString` 類型表示依據 `WideChar` 類型的字串，為何另外定義新的資料類型？`WideString` 過去 (現在仍然) 不列入參照計數，而且效能及彈性相當低 (例如，其中使用 Windows 全域記憶體配置器，而非原生 FastMM4)。

和 `AnsiString` 一樣，`UnicodeString` 會列入參照計數、使用寫入時複製語意，而且效能相當高。與 `AnsiString` 不同的是，`UnicodeString` 針對各個字元都使用 2 位元組，並且是根據 UTF-16。事實上，UTF-16 是可變長度編碼，有時候 `UnicodeString` 使用兩個 `WideChar Surrogate` 項目 (也就是 4 位元組) 來表示單一 Unicode 字碼指標。

此字串類型現在會以固定的硬式編碼方式對應到 `UnicodeString`，這與 `Char` 類型的原因相同。目前沒有任何編譯器指示詞或其他方法可變更此一方式。如果程式碼需要繼續使用此字串類型，只需要以 `AnsiString` 類型的明確宣告取代即可。

字串的內部結構

新 `UnicodeString` 類型的其中一項重要變更是內部的表示方式。

這個新的表示是由 `UnicodeString` 及 `AnsiString` 等所有參照計數字串類型所共用，而非由 `ShortString` 及 `WideString` 等非參照計數字串類型所共用。

「典型 `AnsiString` 類型」的表示如下：

-8	-4	字串參照位址
參照計數	長度	字串的第一個字元

第一個項目 (從字串本身的開端回溯計數) 是 Pascal 字串長度，第二個項目是參照計數。在 Delphi 2009 中，參照計數字串的代表會成爲：

-12	-10	-8	-4	字串參照位址
字碼頁	項目大小	參照計數	長度	字串的第一個字元

除了長度及參照計數外，新的欄位也表示項目大小及字碼頁。項目大小可用來區分 `AnsiString` 及 `UnicodeString`，而字碼頁特別適用於 `AnsiString` 類型 (可用於 Delphi 2009)，`UnicodeString` 類型的字碼頁則固定爲 1200。

在 `System` 單元中，會將對應的支援資料結構宣告爲：

```
type
  PStrRec = ^StrRec;
  StrRec = packed record
    codePage:word;
    elemSize:word;
    refCnt:Longint;
    length:Longint;
  end;
```

在進行實作時，無法將此用於程式碼中，對於針對特定實作方式且可能變動的內部資料結構而言，這是可理解的。另外有 `helper` 函式可取得您經常需要使用的資訊。

由於字串都是 8 到 12 位元組，或許有人會想，雖然新欄位比傳統的方式較精簡 (只需要調整相容性即可變更)，不過是不是較精簡的方式不見得較有效？這就是一般取捨記憶體與速度的兩難問題：將資料儲存於不同的記憶體位置 (並且不使用單一位置的部分)，便能夠使執行階段速度加快，不過您建立的各個字串會佔用額外的記憶體。

在過去，您必須使用低層級的指標型程式碼來取得參照計數，而 Delphi 2009 RTL 新增多項實用函式，可供取得不同的字串中繼資料：

```
function StringElementsize(const S:UnicodeString):word;
function StringCodePage(const S:UnicodeString):Word;
function StringRefCount(const S:UnicodeString):Longint;
```

另外，在 `SysUtils` 單元中，有一項稱為 `ByteLength` 的全新 helper 函式，可傳回 `UnicodeString` 的位元組大小，並忽略 `StringElementSize` 屬性 (不過不適用於 `UnicodeString` 以外的其他字串類型)。

舉例來說，您可以建立字串，並要求關於字串的某些資訊：

```
var
  str1:string;
begin
  str1 := 'foo';
  Memo1.Lines.Add ('SizeOf:' + IntToStr (SizeOf (str1)));
  Memo1.Lines.Add ('Length:' + IntToStr (Length (str1)));
  Memo1.Lines.Add ('StringElementSize: ' +
    IntToStr (StringElementSize (str1)));
  Memo1.Lines.Add ('StringRefCount:' +
    IntToStr (StringRefCount (str1)));
  Memo1.Lines.Add ('StringCodePage:' +
    IntToStr (StringCodePage (str1)));
  if StringCodePage (str1) = DefaultUnicodeCodePage then
    Memo1.Lines.Add ('Is Unicode');
  Memo1.Lines.Add ('Size in bytes: ' +
    IntToStr (Length (str1) * StringElementSize (str1)));
  Memo1.Lines.Add ('ByteLength:' +
    IntToStr (ByteLength (str1)));
```

此程式會產生如下的輸出：

```
SizeOf: 4
Length: 3
StringElementSize: 2
StringRefCount: -1
StringCodePage: 1200
Is Unicode
Size in bytes: 6
ByteLength: 6
```

`UnicodeString` 傳回的字碼頁為 1200，這個字數是儲存在全域變數 `DefaultUnicodeCodePage` 中。在以上的程式碼 (及其輸出) 中，可以明顯看出其中未透過直接呼叫的方式決定字串的位元組長度，因為 `Length` 會傳回字元數。

當然，您 (通常) 可以使用下列運算是，將此乘以各字元的位元組大小：

```
| Length (str1) * StringElementSize (str1)
```

您不僅能夠取得字串的資訊，也能夠變更其中某些資訊。轉換字串的低層級方式是呼叫 `SetCodePage` 程序 (這僅適用於 `RawByteString`)，其中可以按照實際狀況調整字碼頁，也能夠執行完整的字串轉換。在「字串轉換」一節中，將使用此程序進行。

UNICODESTRING 及 UNICODE

眾所週知，新的字串類型 (更準確的說法是 `UnicodeString` 類型) 對應於 `Unicode` 字元集，不過，問

題是，究竟是什麼樣的 Unicode ？

新的字串類型依然使用 UTF-16，更準確的說法是，儲存在記憶體中成爲 UTF-16 字串且以位元組由小到大表示 (UTF-16 LE) 的 `UnicodeString` 類型。這在許多方面都相當有意義，最顯著的意義是，這是最新版作業系統中由 Windows API 管理的原生字串類型。

先前已說明 Delphi 2009 的 `WideChar` 類型，而新的 `TCharacter` 支援類型 (並非使用於 `WideChar` 而使用於 `UnicodeString` 處理) 完全支援 UTF-16 及 `Surrogate` 字組。不過先前未提及將 `WideChar` 項目數變更為與其中包含的 Unicode 字碼指標數不同所造成的影響，因爲一個 `Surrogate` 字組 (也就是兩個 `WideChar`) 可以表示單一 Unicode 字碼指標。

若要使用 `Surrogate` 字組建立字串，可使用 `ConvertFromUtf32` 函式，這會以適當條件透過 `Surrogate` 字組 (兩個 `WideChar`) 傳回字串。

```
var  
  str1:string;  
begin  
  str1 := 'Surr.' + ConvertFromUtf32($1D11E);
```

由於您要求字串長度，所以取得 8，這是 `WideChar` 的數字，而非字串的邏輯 Unicode 字碼指標數。如果印出字串，會產生適當的效果 (至少 Windows 一般會顯示一個方塊作爲 `Surrogate` 字組的預留位置，而非兩個方塊)。

例外，對於 `ConvertFromUtf32` (更準確的說法是 `TCharacter` 類別的 `ConvertFromUtf32` 類別) 的程式碼，您會看見用於將 Unicode 字碼指標對應到 `Surrogate` 字組的實際演算法。如果對細節有興趣，可以繼續往下讀。

字串的各字元出現迴圈時，會發生相關問題。標準的 `for` 迴圈或 `for-in` 循環只會讓您處理字串的各 `WideChar` 項目，而非各個邏輯 Unicode 字碼指標。因此，您可能必須根據 `NextCharIndex` 函式使用 `while` 迴圈，或使用 `for` 檢查 `Surrogate` 字組：

```
if TCharacter.IsHighSurrogate (str1 [I]) then  
  Memo1.Lines.Add (str1 [I] + str1 [I+1])
```

然而，在大多數情況下，您都可以使用 BMP (基本多語文字面)，將 Unicode 字串的各個 WideChar 視為單一字碼指標進行處理。

UCS4STRING 類型

另外有一個字串類型可以用來處理一連串的 Unicode 字碼指標，這就是 UCS4String 類型。這個資料類型表示 4 位元組字元的動態陣列 (UCS4Char 類型)。因此，這沒有參照計數或寫入時複製支援，並且具有極少的 RTL 支援。

雖然這個資料類型 (在 Delphi 2007 中已可供使用) 可用於特定的情況，但是並非特別適用於一般條件。這會佔用相當多的記憶體資源，因為不只字串的各個字元都使用 4 位元組，而且記憶體會重複儲存同一組字串。

多種字串類型

引入新的 UnicodeString 類型後，所有字串類型 (包括 AnsiString 類型) 共用的更新內部表示可以更高提升字串管理。Delphi R&D 團隊已運用這項新的內部表示 (以及在編譯器層級進行以提升字串管理的所有成果) 實際提供多種字串類型，以及全新的字串類型定義機制。

除了 UnicodeString 之外，預設定義的字串類型如下：

AnsiString 是各字元單一位元組的字串類型，其中是以作業系統的目前字碼頁為依據，近似於先前 Delphi 版本的*典型* AnsiString：

UTF8String 是以可變字元長度 UTF8 格式為依據的字串。

RawByteString 是未設定任何字碼頁的字元陣列，系統不會據以完成任何字元轉換 (因此，作為純粹的字元陣列使用時，某些方面相當類似*典型* AnsiString)。

在這些新字串類型的定義中，可看出類型定義機制：

```
type
  UTF8String = type AnsiString(65001);
  RawByteString = type AnsiString($FFFF);
```

下一節將說明 AnsiString 及自訂字串類型，並說明 UTF8String 類型。下一節說明字串轉換時，將著重在 RawByteStringin，這個字串類型一般是用來避免轉換之用。

新 ANSISTRING 類型

與過去相比，新的 `AnsiType` 字串可以另外容納字元的字碼頁。`DefaultSystemCodePage` 變數預設使用 `CP_ACP`，這是最新的 Windows 字碼頁，而且呼叫 `SetMultiByteConversionCodePage` 這個特殊程序即可進行修改。若要進行修改，可以強制整個程式(預設)處理指定字碼頁(作業系統必須支援此字碼頁)中的字元

一般而言，您可以沿用最新的字碼頁，也可以呼叫 `SetCodePage` 程序(前文探討字元及字碼頁時曾提及)針對個別字串變更字碼頁。有兩種不同的方式可呼叫此程序。首先，在您知道格式的情況下，可以變更字串的字碼頁(可能由個別檔案或通訊端載入)。其次，您可以呼叫此程序轉換指定字串(將字串指派給其中一個不同的字碼頁時，這會自動進行，下文將對此繼續探討)。

雖然您可以繼續使用 `AnsiString` 類型，以達到更精簡的字串記憶體內部表示，但是，在大多數的情況下，您會想要將程式碼轉換為使用全新的 `UnicodeString` 類型，也就是以一般字串類型宣告字串。另外，在某些條件下需要使用特定字串類型。例如，載入或儲存檔案時，或者移動資料進出資料庫時，使用程式碼的各個字元格式必須維持 8 位元的網際網路通訊協定。在所有這些情況下，請轉換程式碼使用 `AnsiString`。

建立自訂字串類型

除了使用編譯應用程式時與使用的預設字碼頁相關聯的全新 `AnsiString` 類型之外，您也可以使用相同的機制來定義您自己的自訂字串類型。例如，您可以撰寫下列程式碼來定義 Latin-1 字串：

```
type
  Latin1String = type AnsiString(28591);

procedure
  TFormLatinTest.btnNewTypeClick( sender:
    TObject);
var
  str1:Latin1String;
begin
  str1 := 'a string with an accent:Cantù';
  Log ('String:' + str1);
```

您可以按照其他任何字串的使用方式使用此字串，但是這會與特定字碼頁產生關聯。因此，如果您使用此字串類型，當您將 `Latin1String` 轉換成爲 `UnicodeString` (例如在以上的 `Log` 呼叫中顯示時)，Delphi 編譯器會新增轉換呼叫。程式碼片段的最後一行有 `_UStrFromLStr` 的隱藏呼叫，這會呼叫 `System` 單元的多個內部函式，直到 `MultiByteToWideChar` Windows API 執行真正的轉換作業爲止。以下是呼叫序列：

```

procedure _UStrFromLStr(var Dest:UnicodeString;
  const Source:AnsiString);
procedure InternalUStrFromPCharLen(
  var Dest:UnicodeString; Source:PAnsiChar;
  Length:Integer; CodePage:Integer);
function WCharFromChar(WCharDest:PWideChar;
  DestChars:Integer; const CharSource:PAnsiChar;
  SrcBytes:Integer; CodePage:Integer):Integer;
function MultiByteToWideChar(CodePage, Flags:Integer;
  MBStr:PAnsiChar; MBCount:Integer;
  WCStr:PWideChar; WCCount:Integer):Integer; stdcall;
external kernel name 'MultiByteToWideChar';

```

Windows API 可以執行適當的轉換，但是這些可能是不準確的轉換，這是因為不同 Windows 字碼頁的某些字元無法以 Latin1 表示，例如歐元貨幣符號和智慧引號。

以上的 `btnNewTypeClick` 方法繼續顯現字串的其他某些細節：

```

  Log ('Last char:' + IntToStr (
    Ord (str1[Length(str1)]));
  Log ('ElemSize:' + IntToStr (StringElementSize (str1)));
  Log ('Length:' + IntToStr (Length (str1)));
  Log ('CodePage:' + IntToStr (StringCodePage (str1)));

```

執行此程式碼會產生下列輸出：

```

Last char: 249
ElemSize: 1
Length: 30
CodePage: 28591

```

爲了證明處理這個全新自訂字串類型的方式與標準 `AnsiString` 類型不同（至少在我的電腦及我的地區設定中是如此），我已經撰寫測試方法，將相同的上端字元（從 #128 到 #255）新增到 `AnsiString` 及 `Latin1String`，以便在群組的備忘錄中顯示：

```

procedure
  TFormLatinTest.btnCompareCharSetClick(Sender:T
  Object);
var
  str1:Latin1String;
  str2:AnsiString;
  I:Integer;
begin
  for I := 128 to 255 do
  begin

```

```

    str1 := str1 + AnsiChar (I);
    str2 := str2 + AnsiChar (I);
end;

for I := 0 to 15 do
begin
    Log (IntToStr (128 + I*8) + ' - ' +
        IntToStr (128 + I*8 + 7));
    Log ('Lati:' + Copy (str1, 1 + i*8, 8));
    Log ('Ansi:' + Copy (str2, 1 + i*8, 8));
end;
end;

```

輸出的起始部分標示兩個字元集之間的差異 (同樣地，您看到的結果會因為您的地區設定而有所不同)：

```

128 - 135
Lati:7E,f".77
Ansi:€E,f,,...†‡
136 - 143
Lati:ˆ7S<OEZE
Ansi:ˆ%OS<E<E
144 - 151
Lati:E'""".--
Ansi:E'""".---
152 - 159
Lati:~Ts>oEzY
Ansi:~™š>œEžŸ

```

除此之外，另外有個更有趣的範例是使用非拉丁文字母的字碼頁，例如古斯拉夫文。以下範例是我定義的第二個自訂字串類型：

```

type
    CyrillicString = type Ansistring(1251);

```

您可以按照先前程式碼片段的方式使用此字串，但是有趣的是使用高序位字元，也就是數值超過 127 的字元。我已經透過 for 迴圈使用其中一些字元：

```

procedure
    TFormLatinTest.btnCyrillicClick( sender:
        TObject);
var
    str1:CyrillicString;
    I:Integer;
begin
    str1 := 'a string with an accent:Cantù';
    Log ('String:' + str1);
    Log ('Lasf char:' + IntToStr (

```

```

    Ord (str1[Length(str1)]));
Log('ElemSize:' + IntToStr (StringElementSize (str1)));
Log('Length:' + IntToStr (Length (str1)));
Log ('CodePage:' + IntToStr (StringCodePage (str1)));

str1 := '';
for I := 150 to 250 do
    str1 := str1 + CyrillicString(AnsiChar (I));
Log ('High end chars:' + str1);
end;

```

此方法的輸出如下：

```

String:a string with an accent:Cantu
Last char: 117
ElemSize: 1
Length: 30
CodePage: 1251
High end chars:--Е™ъ>ьќћц њџѡѓ|§Ё€«¬-
®İ°±İıǧμ¶·ё№є»jSsїАБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮяабвгдежзийклмно
прстуфхцчшщъ

```

您會注意到重音字母已轉換為對應的非重音字母，這是因為無法使用原始值所致。(轉換之後的 `wideCharToMultiByte` 會嘗試在某些情況下以受控制的方式失效。例如，智慧引號會降轉為單引號而非問號，而範例程式碼的重音字母會去除重音符號。)字串常數是 Unicode 字串，而 `str1` 的指派會執行隱含轉換。事實上，最後字元的數值不一樣。

另外，高端字元此時也完全不一樣。為達到需要的效果，試想您必須另外撰寫：

```

| CyrillicSfring(AnsiChar (I))

```

如果您只是將字元連接成字串，然後轉換該字串，則會將這些字元視為 Unicode 字元。

管理 UTF-8 字串

此字串類型的全新內部結構有一項影響，也就是我們現在也可以使用更原生的方式管理 UTF-8 的字串。過去 `UTF8String` 只是字串類型的別名，現在則能夠完全辨識此字串類型：轉換會自動進行，所有現有 UTF-8 字串運用常式都經過移植，可使用全新的特定類型。

考慮以下複雜的程式碼：

```
var
  str8:Utf8String;
  str16:string;
begin
  str8 := 'Cantù';
  Memo1.Lines.Add ('UTF-8');
  Memo1.Lines.Add('Length:' + IntToStr (Length (str8)));
  Memo1.Lines.Add('5:' + IntToStr (Ord (str8[5])));
  Memo1.Lines.Add('6:' + IntToStr (Ord (str8[6])));

  str16 := str8;
  Memo1.Lines.Add ('UTF-16');
  Memo1.Lines.Add('Length:' + IntToStr (Length (str16)));
  Memo1.Lines.Add('5:' + IntToStr (Ord (str16[5])));
```

如您所預期，**str8** 字串的長度為 6 (表示 6 位元組)，而 **str16** 字串的長度為 5 (表示 10 位元組)。請注意，**Length** 一定會傳回字串項目數，以避免可變長度表示不符合字串表示的 Unicode 字碼指標數。以下是程式的輸出：

```
UTF-8
Length: 6
5: 195
6: 185

UTF-16
Length: 5
5: 249
```

原因是 UTF-8 字串使用可變長度實作，因此超出起始 7 位元 ANSI 空間的字元會使用至少兩個字元。對於以上的重音字母 *u*，情況也是如此。將相同的 UTF-8 字串指派給 **AnsiString** 變數，並且執行類似的程式碼，得到以下的輸出：

```
ANSI
Length: 5
5: 249
```

然而，這次的字串長度 5 確實是表示 5 位元組，而非 5 個字元。

UTF-8 格式的支援可能不如 UTF-16 的 Delphi 2009 原生字串實作支援完備，但是已經過大幅提升。在 **WideStrUtils** 單元進行 UTF-8 運用有特定的常式，另外也完整支援此格式的串流文字檔 (「串流及編碼」一節將談及 **TEncoding** 類別及文字檔轉換)。重要的是，您可以處理這類字串，並且以任何控制項顯示，完全不需要執行明確的轉換 (必須記得是否於何時執行轉換)，這相當有助益。

由於額外的 **UnicodeString** 類型來回轉換，使得 UTF-8 字串的某些作業可能相當緩慢，但是使用特定的資料類型，而非使用編譯器強制使用的別名類型，對於必須處理這類編碼的任何 Delphi 開發人員而言，會有極大的不同。

您可以使用這個特定的字串類型撰寫現有常式 (或新常式) 的大量執行版本，以避免任何額外的轉換。

轉換字串

您已經能夠將 `UnicodeString` 值指派為 `AnsiString` 或 `UTF8String`，而且能夠進行適當的轉換。同樣地，當您將使用指定字碼頁的 `AnsiString` 指派為根據不同字碼頁的其他字串，會進行轉換。您也可以將字串指派為不同的字碼頁，要求進行轉換，以便轉換字串：

```
type
  Latin1String = type AnsiString(28591);

procedure
  TFormStringConvert.btnLatin1Click( Sender:
    TObject);
var
  str1:AnsiString;
  str2:Latin1String;
  rbs:RawByteString;
begin
  str1 := 'any string with a €';
  str2 := str1;

  Memo1.Lines.Add (str1);
  Memo1.Lines.Add (IntToStr (Ord (str1[19])));

  Memo1.Lines.Add (str2);
  Memo1.Lines.Add (IntToStr (Ord (str2[19])));
  rbs := str1;
  SetCodePage(rbs, 28591, True);
  Memo1.Lines.Add (rbs);
  Memo1.Lines.Add (IntToStr (Ord (rbs[19])));
end;
```

在以上兩種情況下，進行的轉換是不準確的轉換，因為無法以 `Latin1` 字碼頁表示歐元符號。其中必須注意 `SetCodePage` 常式的使用，這僅適用於 `RawByteString` 參數，因此才能進行指派。產生的輸出是：

```
| any string with a €
| 128
```

```
any string with a 7  
63  
any string with a 7  
63
```

轉換可能使程式碼執行緩慢

背景執行的自動轉換相當方便，因為系統會自動執行作業，但是，如果您未審慎考量您執行的作業，可能會因為連續轉換及字串複製作業，最終產生某些執行緩慢的程式碼。考慮以下的程式碼：

```
str1 := 'Marco ';  
str2 := 'Cantù ';  
for I := 1 to 10000 do  
    str1 := str1 + str2;
```

演算法可以會相當迅速或相當緩慢，這需視兩個字串的實際字串類型而定。示範在第一次執行時使用字串 (也就是 `UnicodeString`)，並且在第二次執行時使用 `AnsiString` 及 `UTF8String` 的組合 (這是最嚴重的狀況，因此這兩者必須針對各項指派來回轉換 `UnicodeString` 類型)。這是 10,000 次循環的結果：

```
plain: 00.001  
mixed: 01.717
```

是的，您讀取的是正確的數字，這大約 1,000 次，也就是 10 的 3 次方！如果一切都沒問題，可以考慮 50,000 個串連：

```
plain: 00:00.003  
mixed: 00:42.879
```

這又是一個幾次方的數字！(由於需要多次在記憶體中重新配置愈來愈大的字串，因此呈現次方數增加。程式碼執行緩慢的狀況只發生在轉換時，但是通常需要建立新的大量暫時字串，而非持續增加現有字串的大小。)換句話說，偶爾使用隱含轉換不至於發生問題，但是千萬不要在迴圈或遞迴常式中使用隱含轉換！

必須要謹記的是，您可以在啟用字串轉換警告 (預設為啟用) 的情況下編譯程式，並檢視編譯器在哪些地方新增轉換程式碼。在用於串連不同類型字串的一行程式碼中，您會看見下列警告：

```
W1057 Implicit string cast from 'UTF8String' to 'string'
```

```
w1057 Implicit string cast from 'AnsiString' to 'string'  
w1058 Implicit string cast with potential data loss from 'string'  
to 'UTF8String'
```

由於並非所有的字串都能夠以所有格式表示，因此會發生「潛在資料遺失」的問題。例如，如果將 `UnicodeString` 指派給 `AnsiString`，就有可能無法進行作業。由於字串轉換作業相當常見，因此對應的兩個警告(隱含字串轉換和潛在資料遺失的隱含字串轉換)預設為關閉。

啟動這些警告後，您會發現許多潛在的問題，而一般的程式也有許多警告，即使明確的類型轉換也無法移除這些警告，但是只要將這些警告轉換為另一組警告即可(明確字串轉換和潛在資料遺失的明確字串轉換)。完成檢查後，就將這些警告關閉！

在無法轉換某些字元的情況下，將字串字串常數指派給字串時，會出現第五個類似的警告。這個情況下的警告略有不同：

```
[DCC warning] StringConvertForm.pas(63):w2455 Narrowing given  
wide string constant lost information
```

您不需理會這個警告，因為這個作業沒有太大的意義。

另一個狀況是隱含(或可說是隱藏)轉換使得程式執行緩慢，請考慮下列的程式碼片段：

```
str1 := 'Marco Cantù';  
for I := 1 to MaxLoop2 do  
    str1 := AnsiUpperCase (str1);
```

在 `str1` 變數為 `UnicodeString` 的情況下，一切都相當平順，但是，在該變數為 `AnsiString` 的情況下，則會導致兩項轉換。這個狀況不如先前的情況嚴重(因為字串很短，但是需要複製字串)，不過會出現略微的負荷情形(對於一百萬次循環)：

```
AnsiUpperCase (string): 00:00.289  
AnsiUpperCase (AnsiString): 00:00.540
```

使用 RAWBYTESTRING

萬一需要將 `AnsiString` 作為參數傳遞到常式，該怎麼辦？進行編碼時，將參數指派給特定字串，會將參數轉換為適當的類型，不過會發生潛在資料遺失。這就是為什麼 Delphi 2009 導入另一個稱為 `RawByteString` 的自訂字串類型，其定義如下：

```
type  
    RawByteString = type AnsiString($ffff);
```

這個定義會建立不含編碼的字串類型，更準確的說法是其中包含表示「無編碼」的預留位置

\$ffff。RawByteString 可被視為位元組字串，在進行自動轉換且指派為 AnsiString 的情況下，這會忽略附加的編碼。換句話說，剖析各字元字串 1 位元組成爲 RawByteString 參數時，不會進行任何轉換，這與其他任何 AnsiString 衍生類型不同。您可以呼叫 **SetCodePage** 常式以執行特定轉換，如先前「轉換字串」一節所示。

因此，這可便於取代程式碼的字串 (或 AnsiString) 類型，以便使用字串進行維持各字元 1 位元組的一般和自訂資料處理。(切勿與各字元 Ansi 相容字串 1 位元組的延伸支援相混淆：偏好的解決方案是將字串處理程式碼移轉爲 UnicodeString 類型。切勿使用這些新的額外字串類型。)

盡量不要宣告類型 RawByteString 的變數儲存實際的字串。對於未經定義字碼頁，這會導致不確定的運作方式和潛在資料遺失。另一方面，如果您想要使用類似字串的記憶體配置和表示來儲存二進位資料，可以使用 RawByteString，使用方法與在舊版 Delphi 使用 AnsiString 相同。以 RawByteString 取代使用 AnsiString 的非字串程式碼是相當有趣的移轉過程。

現在，讓我們著重探討可以使用 RawByteString 類型作爲參數的典型範例。如果您要顯示關於 8 位元字串的某些資訊，您可以撰寫以下兩項宣告的其中一項 (這些都是 RawTest 示範主表單的方法)：

```
procedure DisplayStringData (str:AnsiString);  
procedure DisplayRawData (str:RawByteString);
```

這兩種方法的程式碼都相同 (這裡只列出兩者之中的一種)：

```
procedure TFormRawTest.DisplayRawData(  
    str:RawByteString);  
begin  
    Log ('DisplayRawData(str:RawByteString)');  
    Log ('String:' + UnicodeString(str));  
    Log ('CodePage:' + IntToStr (StringCodePage (str)));  
    Log ('Address:' + IntToStr (Integer (Pointer (str))));  
end;
```

請注意，轉換為 `UnicodeString` 都會顯示適當的字串，這是必須進行的轉換，以避免因為與未於編譯時定義字碼頁的字串進行字串常值串連，而將資料視為一般的 `AnsiString`。(直接使用 `Log (str)` 也有效，因為其中不需要任何串連。)

串流及編碼

如果在使用 RTL 和 VCL 而且能夠使用 Windows API 時，將應用程式中所有的字串轉換為 Unicode，則會顯得較為複雜，因為您必須在檔案來回讀取和撰寫字串。以 `TStrings` 檔案作業為例，會發生什麼情形？

Delphi 2009 導入另一項稱為 `TEncoding` 的全新類別來處理檔案編碼，這近似於 .NET 架構的 `System.Text.Encoding` 類別。在 `SysUtils` 單元中定義的 `TEncoding` 類別有多個子類別表示 Delphi 自動支援的編碼 (這些都是您可以自行新增的標準編碼)：

```
type
  TEncoding = class
    TMBCSEncoding = class(TEncoding)
      TUTF7Encoding = class(TMBCSEncoding)
        TUTF8Encoding = class(TUTF7Encoding)
          TUnicodeEncoding = class(TEncoding)
            TBigEndianUnicodeEncoding = class(TUnicodeEncoding)
```

在 `TEncoding` 類別中，這些類別都個別有一個物件可供使用，例如類別資料，而且都有對應的 `getter` 函式和類別屬性：

```
type
  TEncoding = class
  public
    class property ASCII:TEncoding read GetASCII;
    class property BigEndianUnicode:TEncoding
      read GetBigEndianUnicode;
    class property Default:TEncoding read GetDefault;
    class property Unicode:TEncoding read GetUnicode;
    class property UTF7:TEncoding read GetUTF7;
    class property UTF8: TEncoding read GetUTF8;
```

`TEncoding` 類別有讀取和撰寫位元組串流字元的方法可執行轉換，並且有特殊的函式可處理名稱為 `GetPreamble` 的 BOM。因此，您可以撰寫 (在程式碼的任何位置)：

```
| TEncoding.UTF8.GetPreamble
```

串流 TSTRING

透過編碼可呼叫 `TStrings` 類別的 `ReadFromFile` 和 `WriteToFile` 方法。如果您撰寫文字檔的字串清單，而未提供特定的編碼，類別將使用 `TEncoding.Default`，這會使用第一次進行時由目前 Windows 字碼頁擷取的內部 `DefaultEncoding`。換句話說，如果您儲存檔案，便會取得與先前相同的 ANSI 檔案。

當然，您也可以強制使檔案使用不同的格式，例如 UTF-16 格式：

```
Memo1.Lines.SaveToFile('test.txt',  
    TEencoding.Unicode);
```

這會以 Unicode BOM 或前序編碼儲存檔案。進行對應的 `LoadFromFile` 作業時，如果未指定編碼，載入方法會呼叫 `TEencoding` 類別的 `GetBufferEncoding` 方法，以根據是否有 BOM 決定編碼（如果沒有，則會使用預設的 ANSI 編碼）。

萬一在 `LoadFromFile` 中指定編碼，該怎麼辦？不論檔案中是否有 BOM，都會以您提供的編碼讀取檔案，這通常會造成錯誤。我認為在這種情況下出現例外，以某個字碼頁儲存檔案，而且強制以不同的字碼上傳檔案，顯然是開發人員的錯誤。未出現例外有助於不使用 BOM 儲存編碼的檔案，但是仍不應被視為 ASCII 檔案，事實上這是 UTF 檔案。

現在討論檔案儲存作業。如果您不變更現有的 Delphi 程式碼，您的程式會將檔案儲存為 ANSI。如果您現有的程式不處理 Unicode 資料，您的程式及相關的檔案則完全向下相容。萬一程式處理 Unicode 資料，該怎麼辦？假設有一份以不同語言撰寫的字串清單，例如以下的設計時間表單：



圖 1 以不同語言顯示字串的設計表單

如果現有 Delphi 程式碼將字串清單儲存為檔案，然後重新載入檔案，則可能成為：

```

procedure
  TFormStreamEncoding.btnPlainClick( sender:
    TObject);
var
  strFileName:string;
begin
  strFileName := 'PlainText.txt';
  ListBox1.Items.SaveToFile(strFileName);
  ListBox1.Clear;
  ListBox1.Items.LoadFromFile(strFileName);
end;

```

顯然結果相當嚴重，因為只有一部分使用的字元進行 ANSI 呈現，因此清單方塊出現許多問號。簡單的替代方法是變更專案第二個按鈕的事件控制碼程式碼：

```

  strFileName := 'Utf8Text.txt';
  ListBox1.Items.SaveToFile(strFileName, TEncoding.UTF8);

```

另外，我們不需要指定載入字串清單時的編碼，因為 Delphi 會自行從 BOM 中選擇。如果您想要在必要時將資料儲存為 ANSI，可以檢查字串清單內容，以決定是否儲存為 ASCII 或 UTF-8：

```
procedure
  TFormStreamEncoding.btnAsNeededClick(Sender:T
  Object);
var
  strFileName:string;
  encoding1:TEncoding;
begin
  strFileName := 'AsNeededText.txt';
  encoding1 := TEncoding.Default;

  if ListBox1.Items.Text <>
    UnicodeString (AnsiString(ListBox1.Items.Text)) then
    encoding1 := TEncoding.UTF8;

  ListBox1.Items.SaveToFile(strFileName,Encoding1);
```

這段程式碼會檢查您是否能夠將字串轉換為 AnsiString 並回復轉換為 UnicodeString，而不會遺漏任何內容。對於相當長的字串，這項雙重轉換與比較相當耗費資源，因為您可以改用下列的替代程式碼（這不是相當準確，因為需要仰賴特定的字碼頁，但是可達到近似的效果）：

```
var
  ch:Char;
begin
  ...
  for ch in ListBox1.Items.Text do
    if Ord (ch) >= 256 then
      begin
        encoding1 := TEncoding.UTF8;
        break;
      end;
```

使用類似的程式碼，即可視情況決定使用的格式。比較好的做法是，不論實際的資料為何，將所有的檔案轉會為 Unicode 編碼 (UTF-8 或 UTF-16)。使用 UTF-16 會使檔案增大，但是會減少儲存和載入的轉換次數。

不過，由於無法指定預設轉換，因此，除非巧妙變更類別的標準運作方式，否則檔案使用 Unicode 編碼即表示需要變更各個檔案儲存作業。這類的做法能夠以 class helper 進行。考慮以下的程式碼：

| type

```
TStringsHelper = class helper for TStrings
  procedure SaveToFile (const strFileName:string);
end;

procedure TStringsHelper.SaveToFile(
  const strFileName:string);
begin
  inherited SaveToFile (strFileName, TEncoding.UTF8);
end;
```

請注意，其中的繼承不是表示呼叫基本類別，而是 class helper 協助的類別。現在只需要撰寫 (或維持如下的程式碼)：

```
| ListBox1.Items.SaveToFile(strFileName);
```

即可將檔案儲存為 UTF8 (或您選擇的其他任何編碼)。

結論：UNICODE 及 VCL

在 Delphi 語言中使用 Unicode 字串支援相當便利，使 Win32 API 重新對應為的寬字串版本可便於移轉，但是根本的變化是整個 RTL 及視覺化元件程式庫 (VCL) 現在完全採用 Unicode。元件管理的所有字串 (及字串清單) 都經過宣告成為字串，因此現在符合全新的 UnicodeString 類型。

然而，RTL 的某些低層級的內部部分使用不同的格式。例如，屬性名稱是根據 UTF-8，因此 TypInfo 單元中提供的部分 RTTI 支援也是如此。不過，除了少數極特別的例外之外，其他所有部份都已經移轉為 UnicodeString 和 UTF-16。

Unicode 支援是重要的一環，但是不是唯一能夠協助提升建構國際性應用程式支援的功能。

需要注意的是，您可以將原始程式碼檔案儲存為您需要的任何格式，但是，在原始程式碼中使用超過 255 的任何字碼指標時，必須使用 Unicode 格式 (針對識別碼名稱、字串、註解或其他任何項目)。邊其器會提示您在必要時使用這類格式，但是您都可以使用 Unicode 原始程式碼檔案。

關於作者

本白皮書是由暢銷系列 *Mastering Delphi* 的作者 Marco Cantù 為 Embarcadero Technologies 進行撰寫，其中內容摘錄自該作者最新的 *Delphi 2009 Handbook* 一書，詳見 <http://www.marcocantu.com/dh2009>。您可以在 Marco 的部落格 (<http://blog.marcocantu.com>) 閱讀關於他的相關訊息，並且寄送電子郵件到 marco.cantu@gmail.com 與他交流。



Embarcadero Technologies Inc. 能夠讓應用程式開發者與資料庫專業人員在所選擇的環境中，藉由工具設計、建立與執行軟體應用程式。全球超過三百萬使用者的社群以及「財富」雜誌一百大公司中的九十家公司都是仰賴 Embarcadero 的 DatabaseGear™ 與 CodeGear™ 系列產品增加生產力、公開協力合作與自由創新。Embarcadero 成立於 1993 年，總部位於加州舊金山，全球各地都設有辦公室。Embarcadero 的網址為 www.embarcadero.com。公司的旗艦版 CodeGear 工具包括：Delphi®、C++Builder® 和 JBuilder®。