|  |
| --- |
|  |
| Code Convention |
| Backend |
|  |
|  |
|  |

|  |  |  |  |
| --- | --- | --- | --- |
| 版本編號 | 日期 | 描述 | 修訂者 |
| V1.0.0 | 2023-04-24 | 第一版 | Ray Chiu |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目錄

[第一章 程式碼命名篇 4](#_Toc133220413)

[一、 基本命名原則(Naming Policy) 4](#_Toc133220414)

[1. 大駱駝峰命名(Upper Camel Case) 4](#_Toc133220415)

[2. 小駱駝峰命名(Lower Camel Case) 4](#_Toc133220416)

[3. 明確的名稱 5](#_Toc133220417)

[4. 單字單詞 6](#_Toc133220418)

[5. 快速了解 7](#_Toc133220419)

[6. 適用範圍 7](#_Toc133220420)

[二、 空間命名原則(Naming Policy for Namespace) 8](#_Toc133220421)

[1. 基礎類別庫 8](#_Toc133220422)

[2. 專案類別庫 8](#_Toc133220423)

[三、 類別命名原則( Naming Policy for Class) 9](#_Toc133220424)

[1. 基底類別(Basic Class) 9](#_Toc133220425)

[2. 一般類別(General Class) 9](#_Toc133220426)

[3. 集合類別(Collection Class) 9](#_Toc133220427)

[4. 工廠類別(Factory Class) 10](#_Toc133220428)

[5. 工具類別(Utility Class) 10](#_Toc133220429)

[6. Helper類別 10](#_Toc133220430)

[7. 具有Partial修飾詞的類別 10](#_Toc133220431)

[四、 欄位命名原則(Naming Policy for Field) 12](#_Toc133220432)

[五、 屬性命名原則(Naming Policy for Property) 12](#_Toc133220433)

[六、 方法命名原則(Naming for Method) 13](#_Toc133220434)

[1. 擷取 13](#_Toc133220435)

[2. 修改 16](#_Toc133220436)

[3. 建立 18](#_Toc133220437)

[4. 刪除 21](#_Toc133220438)

[5. 轉換 22](#_Toc133220439)

[6. 其他 23](#_Toc133220440)

[七、 參數命名原則 24](#_Toc133220441)

[八、 介面命名原則(Naming Policy for Interface) 24](#_Toc133220442)

[九、 列舉物件命名原則(Naming Policy for Enumerations) 25](#_Toc133220443)

[十、 常數命名原則(Naming Policy for Constants) 26](#_Toc133220444)

[第二章 程式碼開發篇 27](#_Toc133220445)

[一、 編排美學 27](#_Toc133220446)

[1. 調整斷行/縮排 28](#_Toc133220447)

[2. 大小括弧的使用 34](#_Toc133220448)

[3. 有意義的順序 38](#_Toc133220449)

[4. 組織成區塊/段落 41](#_Toc133220450)

[5. 一致的風格 43](#_Toc133220451)

[6. 善用工具 44](#_Toc133220452)

[二、 常見型別的使用 46](#_Toc133220453)

[1. 實質型別 46](#_Toc133220454)

[2. 參考型別 47](#_Toc133220455)

[3. 常用的資料型別 48](#_Toc133220456)

[4. 型別的轉型 49](#_Toc133220457)

[5. 常數的陷阱 50](#_Toc133220458)

[三、 變數與可讀性 51](#_Toc133220459)

[1. 消除變數 52](#_Toc133220460)

[2. 縮限變數的範圍 55](#_Toc133220461)

[3. 偏好單次寫入的變數 57](#_Toc133220462)

[四、 簡化迴圈與邏輯 58](#_Toc133220463)

[1. 條件式中的條件順序 58](#_Toc133220464)

[2. if / else 區塊順序 59](#_Toc133220465)

[3. 三元運算式 60](#_Toc133220466)

[4. 避免使用do / while 迴圈 與 goto 61](#_Toc133220467)

[5. 儘早由函數中返回 61](#_Toc133220468)

[6. 減少巢狀結構 62](#_Toc133220469)

[五、 常用存取修飾詞 64](#_Toc133220470)

[1. Public 64](#_Toc133220471)

[2. Private 64](#_Toc133220472)

[3. Protected 65](#_Toc133220473)

[4. Internal / Friend 65](#_Toc133220474)

[5. Static / Shared 65](#_Toc133220475)

[六、 註解規定與技巧 66](#_Toc133220476)

[1. 基本註解範圍 66](#_Toc133220477)

[2. 良好的註解技巧 69](#_Toc133220478)

[3. 移除不需要註解 72](#_Toc133220479)

[4. 常用工作清單註解標記 75](#_Toc133220480)

# 程式碼命名篇

# 基本命名原則(Naming Policy)

## 大駱駝峰命名(Upper Camel Case)

當名稱是由二個或二個以上的單字連結在一起，而構成的唯一識別字，用以增加程式碼的可讀性。單字之間不以底線(underline，”\_”)或破折號(dash，“-”)連結，第一個單字首字字母採用大寫字母；後續單字的首字母亦用大寫字母。(特殊情況或是常數命名不在此限制)

* 錯誤範例

string Company\_Name;

string First\_Name;

string schedulemailproxy;

違反Upper Camel Case規則，嚴禁使用。

* 正確範例

int Quantity;

string CompanyName;

string ScheduleMailProxy;

符合Upper Camel Case規則。

## 小駱駝峰命名(Lower Camel Case)

當名稱是由二個或二個以上的單字連結在一起，而構成的唯一識別字，用以增加程式碼的可讀性。單字之間不以底線(underline，”\_”)或破折號(dash，“-”)連結，第一個單字首字字母採用小寫字母；後續單字的首字母亦用大寫字母。(特殊情況或是常數命名不在此限制)

* 錯誤範例

string company\_Name;

string first\_Name;

string schedulemailproxy;

違反Lower Camel Case規則，嚴禁使用。

* 正確範例

int quantity;

string companyName;

string scheduleMailProxy;

符合Lower Camel Case規則。

## 明確的名稱

不論是何種物件（類別、屬性、方法、事件、函數、委派或其他物件等），只要是非專屬於迴圈（Loops）中使用的，一律採取明確名稱方式命名，其名稱需要明白的表明用途且須可發音，非必要不可使用無意義的名稱。

* 錯誤範例

int X;

string Y;

上述兩種變數命名方式完全看不出該變數所代表的意義，嚴禁使用。

* 正確範例1

int Quantity;

string CompanyName;

上述兩種變數的命名方式的命名就很明確，完全不需要註解，一看便知其所代表的意義。

* 正確範例2

for (int i = 0; i < 10; i++)

{

    //do stuff.

}

只有在迴圈中能使用單一字母作為計數器。

* 正確範例3

DataView students = new DataView();

foreach (DataRow dr in Students)

{

    //do stuff.

}

在迴圈中能使用約定成俗的簡易表達方式作為物件變數名稱。例如：dr、ds

## 單字單詞

名稱排列以單字或單詞為原則，盡量不要超過五個單字，例如MyName、StockQuantity、ArticleId等，但若業界或慣例中有相對應的縮寫時，可以使用縮寫。另外若縮寫只有二個字母則應全用大寫，若縮寫超過三個字母則第一個字母為大寫，其餘仍為小寫。

* 正確範例

int Quantity; 可縮寫成 int Qty;

int Amount; 可縮寫成 int Amt;

string SerialNumber; 可縮寫成 string SN;

string ExtensibleMarkupLanguage 可縮寫成 string Xml;

## 快速了解

命名要可以讓開發或維護的人員很快的就了解其意義，若物件仍難以直接由名稱了解時，則必須要在宣告處加上註解以協助閱讀。

每個概念只可使用單一字詞，避免造成誤解且不可使用雙關語、模稜兩可的字詞(一個字詞代表兩種不同的目的)。

* 範例1

void CreateAccount()

void GenerateAccount()

void NewAccount()

上述三種命名都沒有錯，但是在同一專案或是同一類別裡，應該統一採取一致的命名方式。

* 範例2

void Add()

void Insert()

void Append()

上述哪個是插入？哪個是附加？哪個是增加呢？

答案：Add 🡪 增加 Insert 🡪 插入 Append 🡪 附加

## 適用範圍

不管是VB或C#一律採取此規範。

# 空間命名原則(Naming Policy for Namespace)

一律採取大駱駝峰命名(Upper Camel Case)

## 基礎類別庫

若開發的是基礎類別庫（Framework）中的組件時，其命名空間必須要以組織名稱作為根命名空間，然後以元件的大類（應用範圍）來命名，接著再以元件的小類（或專案名稱）給定。

* 範例

Evertrust.ClassLibs.Basic

Evertrust.ClassLibs.Web

Evertrust.ClassLibs.Web.ETrust

Evertrust.ClassLibs.WebControl

## 專案類別庫

若開發的是專案，則請以專案名稱作為根命名空間，再以子專案|類別|用途|功能等作為命名空間。

* 範例1

EEPS.Permiss.ClassLibsProxy

EEPS.Permiss.Repository

EEPS.Permiss.Common

EEPS.Permiss.Common.Helpers

# 類別命名原則( Naming Policy for Class)

各式類別的命名規則除了遵循基本命名的原則之外，應使用名詞或名詞片語來命名，類別不再使用任何前綴字元(例如：c/u)來標示。類別名稱需與檔案名稱相同。

一律採取大駱駝峰命名(Upper Camel Case)

## 基底類別(Basic Class)

基底類別除遵循基本命名規則外，需要在字尾加上Base加以區分識別。

* 範例1

public class ProductBase 代表產品的基底類別。

public class StaffBase  代表員工的基底類別。

## 一般類別(General Class)

一般類別除遵循基本命名規則外，應依實際代表的角色來命名。抽象類別(Abstract Class)與一般類別的命名相同。

* 範例1

public class Product  代表產品類別。

public class Customer  代表客戶類別。

public class Advertise  代表廣告類別。

## 集合類別(Collection Class)

集合類別除遵循基本命名規則外，需要在字尾加上Collection加以區分識別。

* 範例1

public class ProductCollection 代表產品集合類別。

public class CustomerCollection  代表客戶集合類別。

## 工廠類別(Factory Class)

工廠類別除遵循基本命名規則外，需要在字尾加上Factory加以區分識別。

* 範例1

public class ProductFactory 代表產品工廠類別。

## 工具類別(Utility Class)

工具類別除遵循基本命名規則外，應以工具名稱、工具性質或輔助性質方式命名，例如FileUtility，HttpUtility等。

* 範例1

public class FileUtility 代表檔案工具類別。

public class HttpUtility 代表HTTP工具類別。

## Helper類別

Help類別除遵循基本命名規則外，應以Help名稱、Help性質方式命名，例如：EnumHelper，SystemTimeHelper等。

* 範例1

public class EnumHelper 代表類舉幫助類別。

public class SystemTimeHelper 代表系統時間幫助類別。

## 具有Partial修飾詞的類別

另外若該類別有使用partial修飾詞，其檔名需按照類別階層命名之。

Partial名詞解釋：類別 (Class)、結構 (Struct)、介面或方法的定義，都可以分割為兩個或兩個以上的原始程式檔 (Source File)。 每個原始程式檔都包含型別或方法定義的區段，而且所有部分會在應用程式進行編譯時組合起來。

* 範例

現有一隻功能龐大的Core類別，需將類別分散到個別的檔案，如：Album, Banner…，可讓多位程式設計人員同時在該專案上進行撰寫，並在編譯階段時組合起來。

檔案1 檔名需為：Core.Album.vb

namespace Evertrust.Library

{

public partial class Core

{

internal Public class Album

{

//do stuff.

}

}

}

檔案2 檔名需為：Core.Banner.vb

namespace Evertrust.Library

{

public partial class Core

{

internal Public class Banner

{

//do stuff.

}

}

}

# 欄位命名原則(Naming Policy for Field)

欄位的命名必須遵守基本命名原則，且整個專案必須具有一致性。

private：\_+小駱駝峰命名(Lower Camel Case)

public：禁止使用

* 範例1

/// <summary>

/// 測試

/// </summary>

public class Class1

{

    private string \_info = "測試資訊";

}

# 屬性命名原則(Naming Policy for Property)

在屬性命名須遵循基本命名規則。

private：(建議使用私有欄位替代)小駱駝峰命名(Lower Camel Case)

Public ：大駱駝峰命名(Upper Camel Case)

* 範例1

/// <summary>

/// 測試

/// </summary>

public class Class1

{

    private string \_info = "測試資訊";

    public string Info

    {

        get { return this.\_info; }

        set { this.\_info = value; }

    }

Public string GetName()

{

//TODO:

}

}

# 方法命名原則(Naming for Method)

除遵循基本命名規則外，應以它要執行的動作來命名，可以是動詞、動名詞或動詞片語。若動作的對象是複數時(例如回傳一份清單或是預期會多於一個的物件)，則應以對象的複數方式來表示之。以下介紹幾種常用的方法命名範例。一律採取大駱駝峰命名(Upper Camel Case)

## 擷取

在一個很明確知道傳入某個參數或是一定條件下，會從資料庫抓取回傳特定資料列或是資料集合的情況之下，方法命名建議採用『Get』前綴詞，讓開發人員可以一看名稱便知該方法的作用。

* 範例1

/// <summary>

/// 從資料庫取得會員資料

/// </summary>

/// <param name="memberId">會員編號</param>

/// <returns>DataRow</returns>

private DataRow GetMember(Guid memberId)

{

    DataRow result = null;

    //do stuff.

    return result;

}

* 範例2

/// <summary>

/// 從資料庫取得會員資料列表

/// </summary>

/// <param name="memberId">會員編號</param>

/// <returns>DataTable</returns>

private DataTable GetMembers(List<Guid> memberId)

{

    DataTable result = null;

    //do stuff.

    return result;

}

如果是在頁面元素或是商業邏輯很明確的定義其功能為『搜尋』時，這時候該方法就不適合冠上『Get』，而是建議採用『Search』前綴詞，好讓接手的開發人員更清楚的知道自己在做甚麼，也讓程式碼更接近人類的語意，增加程式碼的可讀性。

* 範例

/// <summary>

/// 搜尋產品資料列表

/// </summary>

/// <param name="price">價格</param>

/// <param name="color">顏色</param>

/// <returns>DataTable</returns>

private DataTable SearchProducts(int price, productColorEnum color)

{

    DataTable result = null;

    //do stuff.

    return result;

}

如果是要在一個既有的物件集合或是資料集合找尋特定資料的話，建議使用『Find』前綴詞。

* 範例

/// <summary>

/// 找尋員工名稱

/// </summary>

/// <param name="staffCollection">員工資料集合</param>

/// <param name="memberId">員工編號</param>

/// <returns>員工姓名</returns>

private string FindStaffName(List<Staff> staffCollection, int memberId)

{

    string result = staffCollection

                   .Find(x => x.MemberId.Equals(memberId))

                   .MemberName;

    return result;

}

## 修改

“修改”常用的情境一般分為兩種，一種為對資料庫的異動，另一種為現有變數、資料集合或是物件集合的異動。在這兩種狀況下建議方法的前綴詞應分別為『Update』、『Modify』請見下列範例。

* 範例1

/// <summary>

/// 修改會員資料

/// </summary>

/// <param name="memberId">會員編號</param>

/// <param name="memberName">會員姓名</param>

/// <param name="gender">性別</param>

/// <returns>bool</returns>

private bool UpdateMember(Guid memberId, string memberName, int gender)

{

    bool result = false;

    //對資料庫進行資料更新

    return result;

}

* 範例2

/// <summary>

/// 修改學生姓名

/// </summary>

/// <param name="students">學生資料集合</param>

/// <param name="newStudentName">新的名字</param>

/// <returns>bool</returns>

public StudentStruct ModifyStudentName(Students studentCollection, string newStudentName)

{

    studentCollection.Name = newStudentName;

    return studentCollection;

}

/// <summary>

/// 學生資料結構

/// </summary>

public struct Students

{

    public string Name { get;set;}

}

## 建立

“建立”常用的情境一般分為三種，一種為對資料庫資料的建立，一種為資料集合或是物件的建立(資料集或物件建立後，會有後續動作的操作)，一種為單純產生特定資料給其它方法或屬性使用。在這三種狀況下建議方法的前綴詞應分別為『Insert』、『Create』、『Generate』請見下列範例。

* 範例1

/// <summary>

/// 新增員工資料到資料庫

/// </summary>

/// <param name="employeeStructCollection">員工資料集合</param>

/// <returns>bool</returns>

private bool InsertEmployee(EmployeeStruct employeeStructCollection)

{

    bool result = false;

    //新增一筆資料到Employee資料表

    return result;

}

/// <summary>

/// 員工資料結構

/// </summary>

private struct EmployeeStruct

{

    public Guid Id { get; set; }

    public string Name { get; set; }

    public int Gender { get; set; }

}

* 範例2

/// <summary>

/// 學生資料結構

/// </summary>

private struct StudentStruct

{

    public Guid Id { get; set; }

    public string Name { get; set; }

    public int Gender { get; set; }

}

/// <summary>

/// 建立學生資料結構資料

/// </summary>

/// <param name="dr">DataRow</param>

/// <returns>學生資料 Struct</returns>

private static StudentStruct CreateStructData(DataRow dr)

{

    StudentStruct studentData = new StudentStruct();

    studentData.Id = new Guid(dr["Id"].ToString());

    studentData.Name = dr["Name"].ToString();

    studentData.Gender = Convert.ToInt16(dr["Gender"]);

    return studentData;

}

/// <summary>

/// 建立學生資料結構資料集

/// </summary>

/// <param name="dv">DataView</param>

/// <returns>學生資料 ListCollection</returns>

private static List<StudentStruct> CreateStructData(DataView dv)

{

    List<StudentStruct> studentData = new List<StudentStruct>();

    foreach (DataRowView objdr in dv)

    {

        studentData.Add(CreateStructData(objdr.Row));

    }

    return studentData;

}

* 範例3

/// <summary>

/// 產生一組新的Guid

/// </summary>

/// <returns>guid</returns>

private Guid GenerateGuid()

{

    return Guid.NewGuid();

}

/// <summary>

/// 產生一組隨機亂數

/// </summary>

/// <returns>random integer</returns>

private int? GenerateRandom()

{

    int? result = null;

    Random rnd = new Random();

    //do staff.

    return result;

}

## 刪除

“刪除”常用的情境一般分為二種，一種為對資料庫資料的刪除，一種為資料集合或是物件集合的資料/物件移除。在這二種狀況下建議方法的前綴詞應分別為『Delete』、『Remove』請見下列範例。

* 範例1

/// <summary>

/// 刪除學生資料

/// </summary>

/// <param name="id">學生Id</param>

/// <returns>bool</returns>

private bool DeleteStudent(Guid id)

{

    bool result = false;

    //依照傳入Id 刪除資料庫特定資料列

    return result;

}

* 範例2

/// <summary>

/// 移除學生 ListCollection，依照學生姓名

/// </summary>

/// <param name="studentCollection">學生資料集合</param>

/// <param name="name">學生姓名</param>

private void Remove(List<StudentStruct> studentCollection,

                    string name)

{

    var result = studentCollection

                 .RemoveAll(p => p.Name.Contains("johnny"));

}

## 轉換

若有使用到代碼且需要將代碼轉換成有意義的文字呈現時，方法的命名應用前綴詞『Convert』，請見下列範例。

* 範例

/// <summary>

/// 轉換帳號代碼成有意義的文字

/// </summary>

/// <param name="colorCode"></param>

/// <returns>string</returns>

private string ConvertColorCodeToString(int colorCode)

{

    string result = string.Empty;

    switch (colorCode)

    {

        case 1:

            result = "紅色";

            break;

        case 2:

            result = "黃色";

            break;

        case 3:

            result = "藍色";

            break;

    }

    return result;

}

## 其他

其他常見的方法種類如：複製(Copy)、處理(Process)…等，只要遵守基本命名原則以及動詞、動名詞或動詞片語的前綴詞都是可以被接受的。

# 參數命名原則

應按照基本命名規則與小駱駝峰命名外，若為泛型型別參數則要以『T』作為前綴字。若有多個泛型型別可用 T1、T2、T3…表示。

一律採取小駱駝峰命名(Lower Camel Case)

* 範例1

private string GetStaffName(int staffId)

{

    return string.Empty;

}

* 範例2

public void PagedList(List<T> Tdatas, int currentPage)

{

    //do stuff.

}

* 範例3

public void PagedList(List<T> T1, List<T> T2)

{

    //do stuff.

}

# 介面命名原則(Naming Policy for Interface)

應按照基本命名與類別方法的命名準則為其命名，且須以『I』作為前綴詞。

一律採取大駱駝峰命名(Upper Camel Case)

* 範例1

public class IConnectionProvider

public class ICarFactory

# 列舉物件命名原則(Naming Policy for Enumerations)

列舉物件的命名須遵守基本命名規則為其命名。

一律採取大駱駝峰命名(Upper Camel Case)

* 範例1

/// <summary>

/// 產品顏色列舉

/// </summary>

private enum ProductColor

{

    紅色 = 1, 黃色 = 2, 藍色 = 3

}

/// <summary>

/// 店體系列舉

/// </summary>

private enum AccountType

{

    直營 = 1, 加盟 = 2

}

# 常數命名原則(Naming Policy for Constants)

常數的命名除遵循基本命名規則之外，要注意的是一律採取大駱駝峰命名(Upper Camel Case)。但建議使用 static readonly取替常數，此時命名就採用大駱駝峰命名(Upper Camel Case)。

* 範例1：常數

const double Pi = 3.14159;

const int SpeedOfLight = 300000;

* 範例2：static readonly

static readonly double Pi = 3.14159;

static readonly int SpeedOfLight = 300000;

# 程式碼開發篇

在大家都已經熟悉了命名原則之後，接下來我們就要開始規範撰寫程式碼應該遵守的基本規則與風格。

# 編排美學

當打開一個項目時，如果它給我的感覺是寧可重寫也不想去碰，那麼它就是糟糕的。那種熱帶雨林式的程式碼，傳遞給後面程式設計師的情緒是沮喪的。程式碼效率的高低，也許和程式設計師的實際經驗、職業技能有關係；但是不整潔的程式碼，卻可以看出程式設計師的個人習慣和工作態度。如果作為程式設計師想贏得交接者的尊重，那麼就從程式碼的整潔性做起吧。

## 調整斷行/縮排

適當的斷行與縮排可以讓程式碼在視覺上與組織上更為清楚更為一目了然，而不是亂用斷行/縮排或是不使用斷行/縮排，使得開發人員在觀看程式碼時，還必須要重新在腦海中組織，因而造成困擾。下列為幾個常見的不好習慣，應完全避免。

* 範例1：過多的斷行

/// <summary>

/// 判斷字串是否為數字字串

/// </summary>

/// <param name="number">字串</param>

/// <returns>true/false</returns>

private bool IsNumber(string value)

{

    bool result = false;

    //(斷行)

    Regex numberPattern = new Regex("[^0-9.-]");

    //(斷行)

    result = !numberPattern.IsMatch(value);

    //(斷行)

    return result;

}

上述例子中，每一行程式碼之間都有著一行多餘的空白斷行，完全沒有意義，嚴禁此行為。較為適合的程式碼如下：

private bool IsNumber(string value)

{

    bool result = false;

    //(斷行)

    Regex numberPattern = new Regex("[^0-9.-]");

    result = !numberPattern.IsMatch(value);

    //(斷行)

    return result;

}

* 範例2：過多的斷型

/// <summary>

/// 累積(加)

/// </summary>

/// <returns>int</returns>

private int Accumulate()

{

    int result = 0;

    //(斷行)

    for (int i = 1; i <= 10; i++)

    {

        //(斷行)

        //(斷行)

        //(斷行)

        result += i;

    }

    //(斷行)

    return result;

}

上述例子中，莫名其妙的有多行多餘的空白斷行，完全沒有意義反而造成困擾，嚴禁此行為。較為適合的程式碼如下：

private int Accumulate()

{

    int result = 0;

    //(斷行)

    for (int i = 1; i <= 10; i++)

    {

        result += i;

    }

    //(斷行)

    return result;

}

* 範例3：迴圈敘述在同一行

/// <summary>

/// 累積(加)

/// </summary>

/// <returns>int</returns>

private int Accumulate()

{

    int result = 0;

    for (int i = 1; i <= 10; i++) { result += i; }

    return result;

}

上述例子中，迴圈的部分撰寫成一行，沒有斷行沒有縮排，雖沒有錯，但是容易造成閱讀上的困難與維護上的不易，應儘量避免。較為適合的程式碼如下：

private int Accumulate()

{

    int result = 0;

    //(斷行)

    for (int i = 1; i <= 10; i++)

    {

        result += i;

    }

    //(斷行)

    return result;

}

* 範例4：if敘述在同一行

private void TestMethod()

{

    if (true) return;

}

或

private void TestMethod()

{

    int i, j;

    if (true) i = 1; else j = 2;

}

上述例子，if / return的部分撰寫成一行，沒有斷行，雖沒有錯，但是容易造成閱讀上的困難與維護上的不易，應儘量避免。較為適合的程式碼如下：

private void TestMethod()

{

    if (true)

    {

        return;

    }

}

或

private void TestMethod()

{

    int i, j;

    if (true)

    {

        i = 1;

    }

    else {

        j = 2;

    }

}

* 範例5：各方法黏在一起

private void CreateId(int id)

{

    //do stuff.

}

private void DeleteId(int id)

{

    //do stuff.

}

private void UpdateId(int id)

{

    //do stuff.

}

上述例子，方法與方法之間沒有斷行區隔，全部黏在一起，容易造成閱讀上的困難與維護上的不易，嚴禁此行為。較為適合的程式碼如下：

private void CreateId(int id)

{

    //do stuff.

}

//(斷行)

private void DeleteId(int id)

{

    //do stuff.

}

//(斷行)

private void UpdateId(int id)

{

    //do stuff.

}

* 範例6：過長的參數

private string UpdateEmployee(string name, string department, int salary, string jobTitle, string county, string district, string address)

{

    //do stuff.

}

上述例子，一個方法的參數過多，導致一行程式碼過長，在閱讀上會造成開發者無法一眼看完，必須來回拖拉水平卷軸，才能閱讀，這種狀況容易使的開發者分心或是思考中斷，所以比較好的做法是在適當的地方斷行，避免拖曳水平卷軸就能一眼望穿。較為適合的程式碼如下：

private string UpdateEmployee(string name,

string department,

                              int salary,

string jobTitle,

                              string county,

string district,

                              string address)

{

    //do stuff.

}

private string UpdateEmployee(

string name,

string department,

    int salary,

string jobTitle,

    string county,

string district,

    string address)

{

    //do stuff.

}

其他像是三元運算式、Linq運算式等皆適用此規則。

## 大小括弧的使用

在某些狀況下，程式碼的大小括弧是可以被忽略，並不會有錯誤的發生，但是還是老話一句，這樣的情況容易造成閱讀的不意，也容易造成維護上的困難與對程式碼的誤解，所以應該用括號包起來的就請不要省略。

* 範例1：沒有括號

/// <summary>

/// 比較A與B數字是否一樣

/// </summary>

/// <param name="a">a數字</param>

/// <param name="b">b數字</param>

/// <returns>bool</returns>

private bool? Compare(int a, int b)

{

    bool? result = null;

    if (a.Equals(b))

        result = true;

    else

        result = false;

    return result;

}

上述例子中，在if / else 判斷式中下方指令僅有一行，雖然可以忽略大括弧，但是容易造成閱讀困難與容易誤解，嚴禁此行為。較為適合的程式碼如下：

private bool? Compare(int a, int b)

{

    bool? result = null;

    if (a.Equals(b))

    {

        result = true;

    }

    else

    {

        result = false;

    }

    return result;

}

* 範例2：沒有括號又沒有縮排

/// <summary>

/// 比較A與B數字是否一樣

/// </summary>

/// <param name="a">a數字</param>

/// <param name="b">b數字</param>

/// <returns>bool</returns>

private bool Compare(int a, int b)

{

    bool result = false;

    if (a.Equals(b))

        result = true;

        a = b;

    return result;

}

上述例子中，是故意將 a = b; 寫在判斷條件式之後，並且不做縮排

沒有加上大括弧，像這種情況就容易發生開發人員的誤判，嚴禁此行為。較為適合的程式碼如下：

private bool Compare(int a, int b)

{

    bool result = false;

    if (a.Equals(b))

    {

        result = true;

    }

    a = b;

    return result;

}

* 範例3：沒有括號判別優先順序

/// <summary>

/// 數學計算

/// </summary>

/// <param name="a">a數字</param>

/// <param name="b">b數字</param>

/// <param name="c">c數字</param>

/// <returns>int</returns>

private int Calculate(int a, int b, int c)

{

    return a + b \* c;

}

上述例子中，若是分別帶入 1, 2, 3 結果答案會是多少呢？相信有不少人會一時之間就忘了先乘除後加減這條定律吧，為避免這樣的情況發生，建議在適當的地方加上小括弧，那麼開發人員就會先去看有括弧的地方了，建議較為適當的程式碼如下：

private int Calculate(int a, int b, int c)

{

    return a + (b \* c);

}

## 有意義的順序

在大多數的情況之下程式碼的順序並不會影響運行的正確性，但是若能夠依照使用情境給予適當的程式碼順序，對於開發人員後續的維護以及可讀性都會來的提高很多。

* 範例：參數接收順序

<form name="form1" method="post">

    <input type="text" name="ID" />

    <input type="text" name="Tel" />

    .

    (還有N個參數略)

</form>

int Id = (int.TryParse(Request.From["Page"], out Id))

        ? Id

        : 1;

string Tel = string.IsNullOrWhiteSpace(Request.From["Tel"])

            ? string.Empty

            : Request.QueryString["Tel"];

上述範例是對於POST資料接收的順序，但是如果接收的順序能夠依照對應的HTML form的<input>欄位順序排列，不管是在何種使用情境都能夠很快速很方便的找到所需要的變數。

另一種情境，在函數具有相依性時，必須把有相依性質的依序排列，且依循『降層準則』來排序，這樣開發人員在撰寫程式碼時，就不太需要跳離太多的行數或段落才能找到需要的區塊，降低出錯與減少時間成本。

* 範例：降層準則

private string GenerateWorkDistanceWord()

{

    string result = string.Empty;

    //住家座標

    Coordinate home = new Coordinate();

    home.Lat = 121.123456;

    home.Lng = 21.321654;

    //公司座標

    Coordinate company = new Coordinate();

    company.Lat = 121.789456;

    company.Lng = 22.123321;

    int distance = CalculateDistance(home, company);

    result = ConvertDistanceLevel(distance);

    return result;

}

private int CalculateDistance(Coordinate pointA, Coordinate pointB)

{

    //return 計算兩點座標距離(略)

}

private string ConvertDistanceLevel(int distance)

{

    return (distance <= 1000) ? "好近" : "好遠";

}

public struct Coordinate

{

    public double Lat { get; set; }

    public double Lng { get; set; }

}

上述的簡單範例中，我們要取得住家到公司上班的距離是”好近”還是”好遠”，一共需要3個方法加上1個結構，按前面所述，依循『降層準則』來對3個方法1個結構做一個適合的程式碼區段排序。

## 組織成區塊/段落

寫作時會因為許多原因將文章分為不同的段落，例如：

* 將相似內容放在一起，與其他概念分開。
* 提供視覺上的『墊腳石』--- 讓讀者清楚知道目前的位置。
* 提示段落間的導引。

出於相同目的，程式碼也應該分成多個段落，例如沒有人會喜歡閱讀下列這樣一大塊的程式碼：

public void GetNewFrients(User data, string password)

{

    var Friends = data.Friends;

    var FriendEmails = Friends.Select(x => x.Email).ToList();

    var AllPeople = ImportAllPeopleFromDB(data.MyEmail, data.MyPassword);

    var AllEmails = AllPeople.Select(x => x.Email).ToList();

    var NotFriendEmails = AllEmails.Except(FriendEmails);

    foreach (var mail in NotFriendEmails)

    {

       //print email list.

    }

}

雖然不太明顯，但是這個函數包含了幾個不同的動作，如果將各步驟的程式碼區分為段落並且加上摘要式註解，會很有幫助，如下：

public void GetNewFrients(User data, string password)

{

    //取得使用者的朋友的Email

    var Friends = data.Friends;

    var FriendEmails = Friends.Select(x => x.Email).ToList();

    //用使用者的Email帳號去資料庫取得所有的人員資料

    var AllPeople = ImportAllPeopleFromDB(data.MyEmail, data.MyPassword);

    var AllEmails = AllPeople.Select(x => x.Email).ToList();

    //找出還不是朋友的Email

    var NotFriendEmails = AllEmails.Except(FriendEmails);

    //將清單列印出來

    foreach (var mail in NotFriendEmails)

    {

       //print email list.

    }

}

適時適當的對程式碼作區塊段落的區分，是有助於瞭解程式碼使其更加清晰容易閱讀與理解，如同文字寫作一樣，段落是非常重要的。

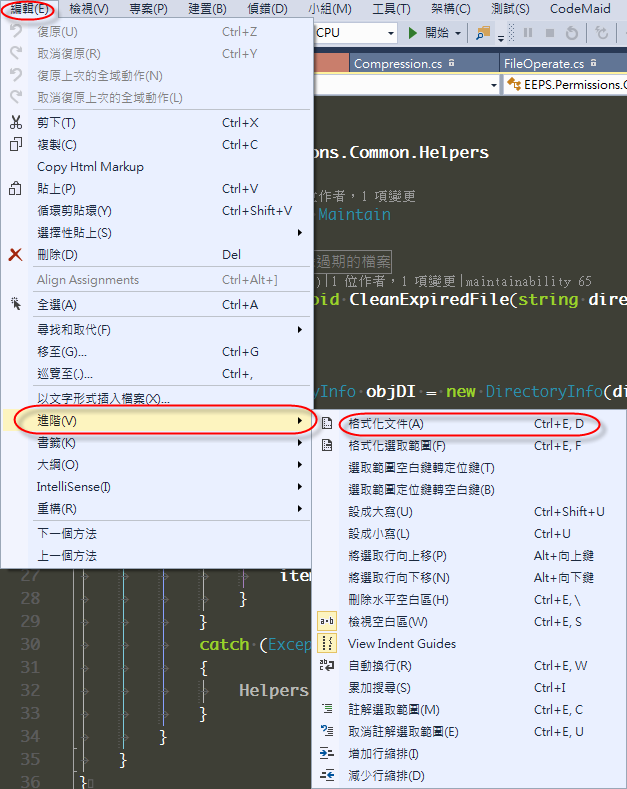
## 一致的風格

有部分美學上的選擇是屬於個人風格，這些風格對可讀性的影響並不大，但如果混合了不同風格就可能會影響可讀性。曾經參與過一些個人認為是錯誤的風格專案，但是仍然遵循專案的慣例，使其看起來風格一致容易閱讀，因為一致性是更為重要的。

## 善用工具

* Visual Studio
* CodeMaid

在我們每天使用的地表最強IDE Visual Studio 系列中，已經有提供一個功能不錯的排版功能，其路徑位於 編輯 🡪 進階 🡪 格式化文件 (如下圖所示)

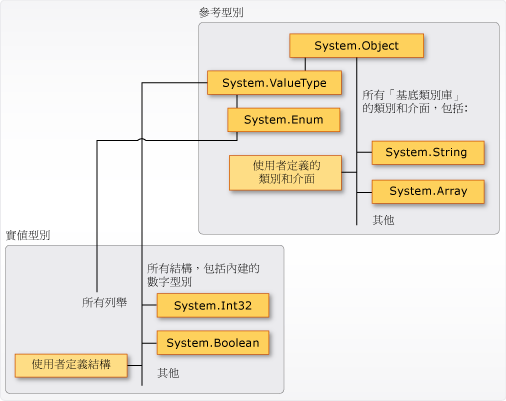


不管是 HTML、CSS、JavaScript、C#、VB等，通通可以用這個功能來做版型的編排，不過部分功能還是需要透過第三方來加強。

最為易用且免費的莫過於CodeMaid這個套件，在不增加開發人員負擔的前提下來自動整理專案中程式碼的排版，讓團隊的程式碼產出趨於一致、大幅提高程式碼的生產品質。其安裝與使用方式，我們將會在工具篇中詳細介紹。

# 常見型別的使用

總體來說，通用語言執行平台(Common Language Runtime，簡稱CLR)定義了兩種型別類型，實質型別(Value Type)和參考型別(Reference Type)，以下我們就這兩種類型與常用的幾種資料型別來做說明。



## 實質型別

**實值型別直接包含資料**，而且實值型別的執行個體 (Instance) 是在堆疊上配置，或是在結構中內嵌 (Inline) 配置。實值型別可由執行階段實作、使用者定義或列舉型別 (Enumeration)。是在其本身的記憶體配置中存放資料。

實質型別包含：

* 所有的數字型別(Numeric Type)
* 布林(Boolean)、字元(Char)及日期(Date)
* 所有結構(Structures) (包括成員屬於參考型別的結構)
* 列舉型別 (Enumeration)，因為其基礎型別一定是 Byte、Short、Integer 或 Long

實質型別的重點如下：

* 不同於參考型別，實質型別不允許Null值，不過可為Null的型別功能允許實值型別指派給Null。(要特別宣告成Nullable這個泛型類別)
* 每個實值型別都有隱含預設建構函式 (Constructor)，所以宣告執行個體時，可以不需要使用new。
* 實值型別都必需明確初始化之後才可以使用。
* 實值型別資料傳遞，是將資料複製到另外一個變數中。

## 參考型別

包含存**放在其他記憶體配置中資料的指標**，儲存變數記憶體位址的參考，並且在堆積 (Heap) 上配置。參考型別的型別可從自我描述型別的值判斷出來。自我描述型別可再分為陣列和類別型別。類別型別包括使用者定義型別、Boxed 實值型別和委派。

參考型別包含：

* 字串(String)
* 所有陣列(Array) (包括元素屬於數值型別的陣列)
* 類別型別(Class)
* 委派(Delegate)
* 介面(Interface)

參考型別的重點如下：

* 參考型別資料必須要使用new 指令建立物件實體,再將實體的參考記錄在變數中。
* 參考型別傳遞時，變數會將實體的參考複製到另外一個變數中。

## 常用的資料型別

* 數字型別(Numeric Type) – 實質型別
* 字串型別(String Type) – 參考型別
* 日期型別(Date Type) – 實質型別
* 布林型別(Boolean Type) – 實質型別
* 列舉型別(Enumeration Type) – 實質型別
* 結構型別(Struct Type) – 實質型別
* 匿名型別(Anonymous Type) – 參考型別
* 通用型別(Loose Type) – 參考型別

## 型別的轉型

在撰寫程式碼時，常常會需要將不同的資料型別做轉型來判斷、運算或是儲存，不同的資料型態都具有各自不同的意義與用途，所以將資料轉型成正確的型態再運用是一件非常重要的事情。

千萬不要再把字串當作是萬用的資料型態。下面將簡單介紹三種常用的型別轉換概念。

1. **隱含轉換**

不必使用特殊語法，因為這類轉換屬於型別安全轉換，而且不會遺失資料。這類範例包括從小型到大型整數型別的轉換，以及從衍生類別 (Derived Class) 到基底類別 (Base Class) 的轉換。

* 範例1

int num = 2147483647;

long bigNum = num;

* 範例2

float num = 123.456F;

double doubleNum = num;

1. **明確轉換**

明確轉換需要使用轉型運算子。當資料可能在轉換中遺失，或轉換可能因故無法成功時，進行轉換 (Cast) 有其必要。例如，對精確度較低或範圍較小型別的數值轉換以及基底類別對衍生類別的轉換都是常見的轉換。

* 範例1

double x = 1234.7;

int y= (int)x;

* 範例2

int height = 180;

short h = (short)height;

1. **使用Helper類別轉換**

為了在不相容型別 (例如整數與 System.DateTime 物件或十六進位字串與位元組陣列) 之間進行轉換，您可以使用 System.BitConverter 類別、System.Convert 類別和內建數字型別 (Numeric Type) 的 Parse 方法，例如 Int32.Parse。

* 範例1

string x = "2";

int y = System.Convert.ToInt32(x);

* 範例2

int i = 0;

string s = "108";

bool result = int.TryParse(s, out i);

* 範例3

string x = "2016-04-08";

DateTime z = System.DateTime.Parse(x);

## 常數的陷阱

如果一個變數希望它無法修改時，此時會使用常數(const)，當常數使用在動態連結庫（Dynamic Link Library），必須注意到一個小陷阱，當修改Dll裡的常數，編譯完讓專案參考它時，如果沒有建置或編譯則會導致專案裡的常數依舊沒有修改。

因此除非有必要性非使用常數不可，否則較建議使用 static readonly的變數來取代常數，可以避免掉上述的陷阱產生。

# 變數與可讀性

濫用變數會造成三個嚴重的問題：

* 變數越多越難同時記住所有變數。
* 變數存活的範圍越大，就必須記得越久。
* 變數越常改變，越難記得目前的數值。

以下介紹幾個常見濫用變數的例子，在程式碼的撰寫過程中，請避免犯下相同類似不好習慣。

## 消除變數

消除對可讀性沒有幫助的變數，消除之後，程式碼會更簡潔也更容易理解。

* 不必要的暫存變數

private void TestMethod()

{

    var now = DateTime.Now;

    var result = now;

    Console.WriteLine(result);

}

Now是個有必要的變數嗎？很顯然的並不是，原因如下：

* 不是分解複雜表示式的結果。
* 程式沒有更加明確，因為 DateTime.Now 已經夠清楚了。
* 只使用一次，沒有消除任何重複的程式碼。較為適當的程式如下：

private void TestMethod()

{

    Console.WriteLine(DateTime.Now);

}

* 消除中間結果private void TestMethod(List<string> arrayObject, string value)

{

    int indexToRemove = 0;

    for (int i = 0; i < arrayObject.Count; i++)

    {

        if (arrayObject[i] == value)

        {

            indexToRemove = i;

            break;

        }

    }

   if (indexToRemove != 0)

    {

        arrayObject.ToList().Remove(value);

    }

}

此範例是從List移除某個字串元素的函數，範例中indexToRemove變數只是用來保存中間的結果，這類變數有時候可以透過儘快處理消除，所以較適當的程式碼如下：

private void TestMethod(List<string> arrayObject, string value)

{

    for (int i = 0; i < arrayObject.Count; i++)

    {

        if (arrayObject[i] == value)

        {

            arrayObject.ToList().Remove(value);

            return;

        }

    }

}

* 消除控制流程變數

private void TestMethod()

{

    bool done = false;

    while (/\*condition\*/ && !done)

    {

        if (...)

        {

            done = true;

            continue;

        }

    }

}

有時候我們會看見這樣子的迴圈，迴圈可能有好幾個地方都會把done變數設為true，這樣的程式碼通常是為了滿足一些不言而喻的規則。像done這樣的變數就是控制流程變數，通常可以透過更結構化或是程式碼重構(程式碼抽離)的方式，進而消除，較為適當的程式如下：

private void TestMethod()

{

    while (/\*condition\*/)

    {

        if (...)

        {

            break;

        }

    }

}

* 消除沒有使用的變數

private const double PI = 3.1415926;

private void CalculateTheArea ()

{

    double PI = 3.14;

    Console.WriteLine(10 \* PI);

}

此例子已經在外部定義一個圓周率的常數，但是方法內部卻又建立一個圓周率的變數，導致外部常數有定義卻不使用，容易造成維護上的困難或誤判，較為適當的程式如下：

private const double PI = 3.1415926;

private void CalculateTheArea()

{

    Console.WriteLine(10 \* PI);

}

## 縮限變數的範圍

大家都知道應該儘量避免使用全域變數，因為太難記住程式中所有全域變數的使用地點與原因。全域變數還會汙染命名空間(放進一堆可能與區域變數衝突的全域變數名稱)，程式碼可能在無意間改變的全域變數的數值，卻自認是在修改區域變數，或是相反的情況。

但更進一步來說，應該要對所有的變數限縮範圍。

* 將類別成員退階至方法的區域變數：

假設有個很大的類別，包含了一個只用在兩個方法的成員變數。

public class Class1

{

    private string SqlCommand = string.Empty;

    private void TestMethodA()

    {

        SqlCommand = "Select .....";

    }

    private void TestMethodB()

    {

        SqlCommand = "Delete .....";

    }

    private void TestMethod1()

    {

    }

    // ...

    // ...

    private void TestMethod10()

    {

    }

}

就某方面而言，類別的成員變數就像是類別內的小型全域變數；特別是對很大的類別而言，很難記得所有的成員變數以及他們被哪一些方法修改，越少小型全域變數越好。就上面的案例，比較適當的程式碼如下：

public class Class1

{

    private void TestMethodA()

    {

        string SqlCommand = "Select .....";

    }

    private void TestMethodB()

    {

        string SqlCommand = "Delete .....";

    }

    private void TestMethod1()

    {

    }

    // ...

    // ...

    private void TestMethod10()

    {

    }

}

將SqlCommand變數成類別的成員變數，下放到各函數裡面的區域變數。

## 偏好單次寫入的變數

操作同一個變數的地方越多，就越難記得當前的數值。尤其是同一個變數在多個區段或是迴圈當中一直不斷被重新賦予新值，此種行為非常容易造成開發人員的混淆與偵錯上的難度，所以儘量使用只寫入一次的變數，來讓程式更容易理解。

# 簡化迴圈與邏輯

如果程式碼中沒有條件判斷、迴圈或是任何控制流程的敘述，讀起來會非常困難且不容易理解的。讓所有的條件判斷、迴圈與其他改變流程的敘述「儘量自然」(閱讀者不需要停下來重讀的方式撰寫程式碼)，將會對程式碼有相當程度的幫助。

## 條件式中的條件順序

重點：條件式中的條件順序： 左側「較有變化」，右側為「比較基準」。

以下兩段程式碼，哪一個好讀？

* 範例1

if(lenght >= 10)

或

if(10 <= lenght)

對大多數程式設計師而言，第一種寫法比較好讀，但是接下來的範例又如何呢？

* 範例2

while(bytesReceived < bytesExpected)

或

while (bytesExpected > bytesReceived)

同樣的，第一個版本比較好讀，為什麼呢？有通用原則嗎？只要謹記下列規則：

**左側「較有變化」，右側為「比較基準」**

這個原則符合英文日常用法，例如：我們會說「如果你的年收入至少10萬美金」、「如果你至少18歲」，這些都是十分自然的說法；換個方式說「如果18歲小於等於你的年紀」就不太自然且繞口了。

## if / else 區塊順序

撰寫if / else敘述時，通常能夠自由交換區塊順序。

* 範例1

if (a == b)

{

    //狀況1

}

else

{

    //狀況2

}

也可以寫成

if (a != b)

{

    //狀況2

}

else

{

    //狀況1

}

大多數的程式設計師對區塊順序不會多做考慮，但是有些條件有助於決定區塊的順序：

* 先處理**肯定**條件而非否定條件。
* 先處理**簡單的**情況，比較能在畫面中同事呈現if與else區塊，在閱讀上很有幫助。
* 先處理比較**有趣**或**明顯**的情況。

這些條件可能互相衝突，設計師必須自行判斷，但是大部分的情況下都是很明確的，運用這三個原則來撰寫 if / else，將會使的程式碼流程更容易被正確解讀。

## 三元運算式

在一些簡單卻繁瑣的條件式上，建議使用「三元運算式」來減少程式碼的行數，並增加可讀性，例如：

* 範例1

string meridiem = (DateTime.Now.Hour > 12) ? "PM" : "AM";

string gender = ("M" == "F") ? "女性" : "男性";

但是有一些較為複雜的表示式如果使用三元運算式來撰寫，雖然程式碼行數會變少，但是卻會相當難以閱讀與偵錯，所以應適時使用，避免濫用。

* 範例2

int result = (exponent >= 0)

    ? mantissa \* (1 << exponent)

    : mantissa / (1 << -exponent);

不好閱讀，如果以 if / else 敘述展開這段程式是比較自然的作法。

if (exponent >= 0)

{

    return mantissa \* (1 << exponent);

}

else

{

    return mantissa / (1 << -exponent);

}

## 避免使用do / while 迴圈 與 goto

do / while 迴圈奇怪的地方在於區塊中的程式碼是根據底下的條件決定是否重複執行，一般來說邏輯判斷都在區塊上方，if、while與for都是如此。因為程式碼是由上讀至下，這讓 do / while 不太自然，許多人都得讀兩次程式碼。

另一個避免使用 do / while 迴圈的原因是 do / while 迴圈內的 continue很難理解。總而言之，請勿再使用 do / while迴圈在任何一個地方。

goto非常容易使程式碼失控以及讓程式碼難以追蹤，尤其是在具有多個goto的目標，彼此的路徑交錯，很容易造成如意大麵般破碎的程式碼，這一些應該都可以用其他的流程控制來解決，所以請勿再使用 goto在任何一個地方。

## 儘早由函數中返回

一些程式設計師認為函數只應該有一個 return 敘述，因為他想執行函式尾端的清除程式碼，但其實不太正確，因為現代的程式語言提供更成熟的機制來達成相同的效果，所以儘早從函數返回式完全沒有問題的，通常也是一個比較好的做法。

* 範例1

private bool TestMethod(string stringValue, string subValue)

{

    if (stringValue == null || subValue == null) return false;

    if (subValue.Equals("")) return true;

    ....

}

## 減少巢狀結構

多層巢狀結構的程式碼十分難懂，每層巢狀結構都會在使用者的「心理堆疊」中加入額外條件，當使用者看到右大括號( } )時，很難彈出堆疊資訊，想起上一層的條件。這種波動拳的撰寫方法將十分的難以閱讀與持續維護，最簡易的修正辦法就是儘早返回函數，消除巢狀結構或是將其拆解成多個函數。

* 範例1

if (result == SUCCESS)

{

    if (permission != SUCCESS)

    {

        reply.WriteError(permission);

reply.Done();

        return;

    }

    reply.WriteError ("...");

}

else

{

    reply.WriteError (result);

}

reply.Done();

利用儘早返回函數的手法可以非常簡單的將此範例修正成較佳的程式結構，所以比較適當的程式碼如下：

if (result != SUCCESS)

{

    reply.WriteLine(result);

    reply.Done;

    return;

}

if (permission != SUCCESS)

{

    Console.WriteLine(permission);

    reply.Done("...");

    return;

}

reply.WriteError("...");

reply.Done;

如何？運用簡單的手法即可獲得不錯的效果，在著手撰寫程式的時候，記得多想想是否有更好的寫法來消除巢狀式結構。

* 範例2

for (int i = 0; i < results.Length; i++)

{

    if (results[i] != null)

    {

        int total++;

        if (results[i].name != "")

        {

            ....

        }

    }

}

好多層，看得眼睛都花了，利用 continue 來簡化一下，所以比較適當的程式碼如下：

for (int i = 0; i < results.Length; i++)

{

    if (results[i] == null) continue;

    int total++;

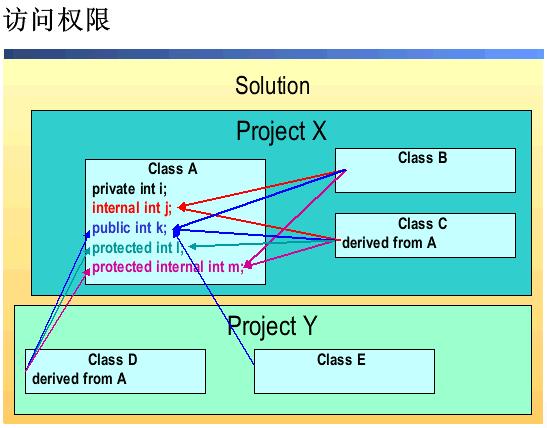
    if (results[i].name == "") continue;

    ....

}

# 常用存取修飾詞

不管任何的程式語言，其所有型別與型別成員都有存取層級，用來控制您的組件或其他組建中的其他程式碼是否可以使用這些型別與成員。您可以在宣告型別或成員時，使用下列存取修飾詞指定存取範圍，這是很重要的，可以限定資源的使用範圍，避免誤用以及其他例外錯誤。



## Public

型別或成員可由相同組件或參考該組件的另一個組件中的任何其他程式碼存取。

## Private

型別或成員只能由相同類別或結構中的程式碼存取。

## Protected

型別或成員只能由相同類別或結構，或是在衍生自該類別的類別中存取。

## Internal / Friend

型別或成員可由相同組件中的任何程式碼存取，但是不包括其他組件中的程式碼。

## Static / Shared

可用來宣告靜態成員，此成員屬於型別本身，並不隸屬任何一個物件。 static 修飾詞可以用於類別、欄位、方法、屬性、運算子、事件及建構函式 (Constructor)，但是不能用於索引子 (Indexer)、解構函式 (Destructor) 或類別以外的型別。

# 註解規定與技巧

程式開發過程中，有時候很難光是靠程式碼來讓別人了解這段程式的用意或是為什麼需要這樣做，甚至時間久了就連原本的作者也忘了當初怎麼會這樣寫，這時候在適時的加入一些正確的註解輔佐，對於日後的維護、重構與文件製作都是一項不可或缺的重要事情。下面將會說明註解的規範以及較為適當的作法。

## 基本註解範圍

下列幾項是最基本的一定要寫註解的地方，在VB下可以連按三下「’」，C#下可以連按三下「/」來產生固定格式，方便填寫。

* **類別**：

必須清楚寫明該類別的用途。

/// <summary>

/// 列舉靜態擴充類別

/// </summary>

public static class EnumExtensions

{

}

/// <summary>

/// 資料庫連線類別

/// </summary>

public class DatabaseConstants : IDatabaseConstants

{

}

* **函數、介面**：

必須寫明函數用途、參數說明與回傳資訊。

/// <summary>

/// 以地址/地標取得 google map 地理編碼資訊

/// </summary>

/// <param name="address">地址/地標</param>

/// <returns>google map地理編碼資訊(GoogleLocateViewModel)</returns>

GoogleLocateViewModel GetGeocoding(string address);

/// <summary>

/// Gets the database connection.

/// </summary>

/// <param name="dbConnectionName">連線字串名稱</param>

/// <returns>System.String.</returns>

string GetDbConnection(string dbConnectionName);

* **資料結構**：

必須寫明結構名稱與成員說明。

/// <summary>

/// google定位資料

/// </summary>

public class GoogleLocateModel

{

    /// <summary>

    /// 縣/市

    /// </summary>

    public string County { get; set; }

    /// <summary>

    /// 鄉/鎮/市/區

    /// </summary>

    public string Distirct { get; set; }

}

* **列舉**：

必須寫明列舉名稱與成員說明。

/// <summary>

/// 伺服器類別

/// </summary>

public enum ServerProdution

{

    /// <summary>

    /// 開發機

    /// </summary>

    Development = 0,

    /// <summary>

    /// 測試機

    /// </summary>

    Test = 1,

    /// <summary>

    /// 封測機

    /// </summary>

    Staging = 2,

    /// <summary>

    /// 正式機

    /// </summary>

    Production = 3

}

* **常數**：

必須寫明常數用途、意義與限制。

/// <summary>

/// 圓周率

/// </summary>

const double Pl = 3.1415;

## 良好的註解技巧

* **維持註解的簡潔**：

應使用清楚簡短的語意來描述，切勿長篇大論。

* **避免模擬兩可的代名詞**：

/// <summary>

/// 將資料插入串列，並會先檢查它是否過大

/// </summary>

使用者需要解讀代名詞的意義，上段註解中的「它」可能是指插入的資料也可能是指串列，雖然可以從程式碼中找出「它」的意義，但如此一來，註解就失去意義了。最安全的方式是直接寫出代名詞所代表的對象，避免任何可能的困擾，修正過後的註解如下：

/// <summary>

/// 將資料插入串列，並會先檢查資料是否過大

/// </summary>

* 修整草率的語句：

很多時候，修改註解為更精準的語句，會讓註解更加簡潔。

/// <summary>

/// 根據是否已經爬過這個url，給予不同的優先權

/// </summary>

這句註解還算可以，但是下面的版本更為簡單、直接，同時也解釋了較高的優先權是給還未爬過的url，這是上面註解並沒有包含的資訊。

/// <summary>

/// 給予不曾爬過的url較高的權限

/// </summary>

* **精確描述函數行為**：

有一個計算檔案行數的函數

/// <summary>

/// 計算檔案的行數

/// </summary>

/// <param name="fileName">檔案名稱</param>

/// <returns>integer</returns>

int CountLines(string fileName);

這樣的註解不十分精確，因為一行有許多不同的定義方式，例如：

* “”(空檔案) 0或1行？
* “hello\n” 1或2行？
* “hello\n world” 1或2行？
* “hello\n\r cruel\n world\r” 2、3或4行？

所以較好的註解應為：

/// <summary>

/// 計算檔案中的換行位元個數(\n)

/// </summary>

/// <param name="fileName">檔案名稱</param>

/// <returns>integer</returns>

int CountLines(string fileName);

* **使用具代表性的輸入/輸出範例**：

/// <summary>

/// 從輸入字串'src'中移除'chars'字首/字尾

/// </summary>

/// <param name="src">輸入字串</param>

/// <param name="chars">移除字串</param>

/// <returns>string</returns>

public string Strip(string src, string chars)

註解不夠精準，無法回答以下疑問：

* Chars是會被移除的子字串，或只是個沒有特定順序的字元集合？
* 如果src的結尾有多組chars重複出現會如何？

這時候，只需要一個範例就能夠回答這一些問題

/// <summary>

/// 從輸入字串'src'中移除'chars'字首/字尾

/// </summary>

/// <param name="src">輸入字串</param>

/// <param name="chars">移除字串</param>

/// <returns>string</returns>

/// <example>範例：Strip("abaa/a/ba", "ab") 傳回："/a"</example>

public string Strip(string src, string chars)

## 移除不需要註解

若是看見屬於這樣的註解…請勇敢的直接刪除它。

* **日誌型註解**：

在很久之前，因為沒有原始碼管理系統，所以有很好的理由，維護這樣的日誌項目。然而如今這些又臭又長的日誌，只會使得模組更加複雜和混淆，它們應該徹底的被消滅。

''' <summary>

''' Sql 類別

''' Date : 2010/7/21 Justin

''' </summary>

''' <remarks>

''' 2015/06/04 亨利 增加 GetTransactionSQL()、GetDataTableBySQL()

''' 2012/1/31 Jacky 修正 因應上稿系統產生的SQL會自動加 Order By ，增加自動判斷 Order By 功能，如未指定 Top 指令，自動移除 Order By 後的所有指令

''' 2011/12/16 Ray 修正 Execute CoommandText 的錯誤

''' 2011/12/15 Ray 增加 GetDataSetBySQL 傳入 SqlParameter 模式

''' 2011/12/14 Ray Execute增加回傳異動筆數

''' 2011/12/14 Ray 增加GetRandomDataTable 產生隨機排序的DataTable

''' 2011/8/9 Ray 只有 Text Mode 才加上執行環境註解

''' 2011/8/9 Ray 為SQL加上執行頁面註解 AppendComment

''' 2010/8/3 Justin 強化 GetDataString 系列，增加傳入 SqlCommand 的方式，讓程式開發接近 .net 的底層。

''' 2010/8/3 Justin 增加 UDL 轉 Sql Client 的 Connection String 字串部分的功能，提供給 AP 系統使用。

''' 2010/7/23 Justin 修正 GetDataString 取回 nothing 時會發生錯誤

''' </remarks>

* **干擾型註解**：

有時候會發現某些註解毫無用處，就只會干擾我們。它們陳述很明顯的事物，又沒有提供新的資訊，這樣的註解也應該被消滅殆盡。

/\*\* the day of the month. \*/

private int DayOfMonth();

/\*\*

\* returns the day of the month.

\*

\* @return the day of the month.

\*/

public int GetDayOfMonth()

{

    return DayOfMonth;

}

* **位置的標誌物**：

有時候程式設計師喜歡在原始檔裡標記某個特別的地方。如下：

//某某某區塊////////////////////////////////////////////////////

or

//某某某區塊----------------------------------------------------

只有在某些極特殊的情況，且具有相當大的顯著效果之下，才允許使用，否則請即刻移除。

* **右大括號後面的註解**：

有一些設計師會在右括號的地方留下一些特別的註解，如下面例子，這對於一個較為複雜的深層巢狀結構的函示而言，可能是有意義的，但過於複雜的巢狀結構只會造成凌亂，所以要在括號右處留下一些註解，倒不如試著簡短函數來取代這樣的行為。

try

{

    while (true)

    {

        try

        {

        } //try

        catch (Exception)

        {

            throw;

        } //catch

    } //while

}

catch (Exception)

{

    throw;

} //catch

* **出處及署名**：

現在的原始碼管控系統非常專精於程式碼修改過程的記錄，對於某人於甚麼時間加入了甚麼程式碼，都能記得清清楚楚，實在沒有必要用小小的署名來汙染你的程式碼，所以此類的資訊就交由程式碼管控系統來記錄即可。

* **被註解起來的程式碼**：

既然都已經是被註解掉的程式碼，就大膽放心的把它們都給移除掉吧，因為那真的很讓人討厭。況且也都可以從版控系統中找到，實在沒有必要留著。

## 常用工作清單註解標記

多加利用工作清單註解標記來標示所遇到的問題，可以利用Visual Studio的工作清單視窗來快速檢視，並可快速分類。

倘若預設工作清單標記不敷使用，各單位可自行擴充，詳情可參閱：(建立工作清單註解) <https://msdn.microsoft.com/zh-tw/library/zce12xx2(v=vs.90).aspx>

|  |  |
| --- | --- |
| 預設標記 | 意義 |
| TODO | 還沒有處理的部分。 |
| UNDONE | 尚未完成 |
| HACK | 承認解決方法不夠優雅。 |