|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **记录名称** |  | | **记录编码** |  | |
| **密 级** |  | | | | |
| **起草/日期** |  | **审核/日期** |  | **批准/日期** |  |

**记录修改履历**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **序号** | **修订前版本** | **现行**  **版本** | **修订内容概述** | **起草** | **修订日期** |
| 1 | N/A | 1.0 |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**目 录**

[1 NET c# wpf介绍 7](#_Toc43881653)

[1.1 NET 7](#_Toc43881654)

[1.2 .net开源 7](#_Toc43881655)

[1.3 NET Framework各版本汇总以及之间的关系 8](#_Toc43881656)

[1.3.1 版本关系 9](#_Toc43881657)

[1.4 JIT 10](#_Toc43881658)

[1.5 第三方库 10](#_Toc43881659)

[1.5.1 WPFtools 10](#_Toc43881660)

[2 常用知识点 11](#_Toc43881661)

[2.1 未处理异常的捕获，定义自己的异常捕获 11](#_Toc43881662)

[2.2 NuGet，联机安装，使用常用的类库。 11](#_Toc43881663)

[2.3 DLL 插件方式的开发MEF,MVVM插件的开发思想。 11](#_Toc43881664)

[2.4 VS自带工具 IL 中间语言的反汇编 12](#_Toc43881665)

[2.5 接口和抽象类有什么区别 12](#_Toc43881666)

[2.6 代理，委托和回调，泛型 12](#_Toc43881667)

[2.7 LOG NLOG 13](#_Toc43881668)

[2.8 线程池 13](#_Toc43881669)

[2.8.1 多个并发的线程，采用线程池的方式。 13](#_Toc43881670)

[2.8.2 同种线程，大并发情况。 13](#_Toc43881671)

[2.9 Dictionary，list，arraylist，hashcode，queue，stack， 13](#_Toc43881672)

[2.10 C#操作符??和?: 13](#_Toc43881673)

[2.11 对象CLONE 15](#_Toc43881674)

[2.12 ORM 15](#_Toc43881675)

[2.13 E-R图 15](#_Toc43881676)

[2.14 构造函数定义为private，protected 16](#_Toc43881677)

[2.15 => 18](#_Toc43881678)

[2.16 As IS 19](#_Toc43881679)

[2.17 P/Invoke 20](#_Toc43881680)

[2.18 IOC控制反转 20](#_Toc43881681)

[2.19 MEF 21](#_Toc43881682)

[2.19.1 AvalonDock 21](#_Toc43881683)

[2.19.2 MahApps.Metro 21](#_Toc43881684)

[2.20 stringbuild appendformat 中的转义 21](#_Toc43881685)

[2.21 C#知识普及网站比微软人性化 22](#_Toc43881686)

[2.22 针对C#作用域 22](#_Toc43881687)

[2.23 Linq及其介绍 22](#_Toc43881688)

[2.24 不同数组间的copyP 22](#_Toc43881689)

[2.25 JIT编译 23](#_Toc43881690)

[2.26 重载运算符 23](#_Toc43881691)

[2.27 VS 拷贝和后期处理 23](#_Toc43881692)

[2.28 对象的引用和销毁 23](#_Toc43881693)

[2.29 C#子类继承父类 24](#_Toc43881694)

[2.30 GetManifestResourceStream得到的Stream是null的解决 24](#_Toc43881695)

[2.31 循环迭代遍历递归 25](#_Toc43881696)

[2.32 资源文件 25](#_Toc43881697)

[2.33 GUID，关于GUID， 26](#_Toc43881698)

[2.34 IaaS\PaaS\Saa 26](#_Toc43881699)

[2.35 GetManifestResourceStream得到的Stream是null的解决 27](#_Toc43881700)

[2.36 字符串操作 27](#_Toc43881701)

[2.36.1 转义输出 27](#_Toc43881702)

[2.37 C# datetime操作全 27](#_Toc43881703)

[2.38 C#采用正则表达式计算字符串数学公式计算类 39](#_Toc43881704)

[2.39 C# 常用函数库 39](#_Toc43881705)

[2.40 杂项 40](#_Toc43881706)

[2.41 GUID 42](#_Toc43881707)

[2.42 循环迭代递归遍历 42](#_Toc43881708)

[2.43 Datatable数据操作 43](#_Toc43881709)

[2.44 反射应用 44](#_Toc43881710)

[2.44.1 详解C#中的反射 44](#_Toc43881711)

[**(1)程序集基础知识** 49](#_Toc43881712)

[**(2)强名称程序集与数字证书** 54](#_Toc43881713)

[**(3)程序集加载 Assembly类** 57](#_Toc43881714)

[**(4)绑定程序集配置策略** 59](#_Toc43881715)

[**(5)CLR 运行时探测程序集引用的步骤** 60](#_Toc43881716)

[**(6)程序集加载上下文** 64](#_Toc43881717)

[**(7)动态程序集加载Load()** 65](#_Toc43881718)

[**(8)程序集反射 Type 类** 70](#_Toc43881719)

[**(9)程序集的加载和反射** 81](#_Toc43881720)

[3 异步调用，并行开发 85](#_Toc43881721)

[3.1 C#中Func与Action的理解 85](#_Toc43881722)

[3.2 异步和线程并行方面讲解得很详细 87](#_Toc43881723)

[3.3 异步处理参考 88](#_Toc43881724)

[3.4 lock相关 88](#_Toc43881725)

[3.5 同步/异步， 阻塞于非阻塞 91](#_Toc43881726)

[3.6 Invoke beginivoke 96](#_Toc43881727)

[3.7 线程的信号机制 96](#_Toc43881728)

[3.8 泛型委托 97](#_Toc43881729)

[4 专题 102](#_Toc43881730)

[4.1 SVN同步 102](#_Toc43881731)

[4.1.1 拷贝 102](#_Toc43881732)

[4.1.2 修改文件内容 102](#_Toc43881733)

[4.1.3 属性生成事件，预先生成命令 103](#_Toc43881734)

[4.2 可伸缩性编程 103](#_Toc43881735)

[4.2.1 可伸缩性（scalability）常见问题 103](#_Toc43881736)

[4.2.2 可扩展性（extensibility）： 103](#_Toc43881737)

[4.2.3 横向伸缩scale out 103](#_Toc43881738)

[4.2.4 纵向伸缩scale up 104](#_Toc43881739)

[4.2.5 横向伸缩和纵向伸缩的综合分析 104](#_Toc43881740)

[5 界面展现技术 105](#_Toc43881741)

[5.1 控件和窗口的常用方法调用 105](#_Toc43881742)

[5.2 窗口覆盖最前面的问题 105](#_Toc43881743)

[5.3 常用Ui技术 105](#_Toc43881744)

[5.3.1 辅助程序代码插件。 105](#_Toc43881745)

[5.4 WPF的绑定技术 105](#_Toc43881746)

[5.5 样式 106](#_Toc43881747)

[5.6 模板 106](#_Toc43881748)

[5.7 Datagrid编辑 106](#_Toc43881749)

[5.8 布局面板 107](#_Toc43881750)

[5.9 资源字典 107](#_Toc43881751)

[5.9.1 DynamicResource与StaticResource的区别 108](#_Toc43881752)

[5.9.2 WPF 命名空间 108](#_Toc43881753)

[5.10 auto规则 108](#_Toc43881754)

[6 软件工程 109](#_Toc43881755)

[6.1 规范化编程 109](#_Toc43881756)

[6.1.1 XML注释规范 109](#_Toc43881757)

[6.1.2 命名规范 111](#_Toc43881758)

[6.2 过度设计和设计不足 120](#_Toc43881759)

[6.3 代码混淆保密 121](#_Toc43881760)

[6.4 UML图例 123](#_Toc43881761)

[6.5 接口和抽象类设计 133](#_Toc43881762)

[6.5.1 概念 133](#_Toc43881763)

[6.5.2 相同点和不同点 137](#_Toc43881764)

[6.5.3 经典示例 139](#_Toc43881765)

[6.5.4 相关概念 139](#_Toc43881766)

[6.5.5 附录 139](#_Toc43881767)

[6.6 异常和状态码 141](#_Toc43881768)

[6.6.1 c#异常处理之性能 141](#_Toc43881769)

[6.6.2 如何处理 143](#_Toc43881770)

[6.6.3 异常对性能的影响 146](#_Toc43881771)

[6.6.4 结论 148](#_Toc43881772)

[6.6.5 关于异常的处理等。 153](#_Toc43881773)

[6.7 回收机制 154](#_Toc43881774)

[7 新观点 163](#_Toc43881775)

[7.1 接口和抽象类区别 163](#_Toc43881776)

[7.2 多重继承 163](#_Toc43881777)

[7.3 静态类，静态方法，静态变量与非静态之前的区别 163](#_Toc43881778)

[7.4 C++ 与C#对比学习 166](#_Toc43881779)

[7.5 Const readonly 166](#_Toc43881780)

[7.6 sealed 修饰符表示密封 167](#_Toc43881781)

[8 技术网站 167](#_Toc43881782)

# **概述**

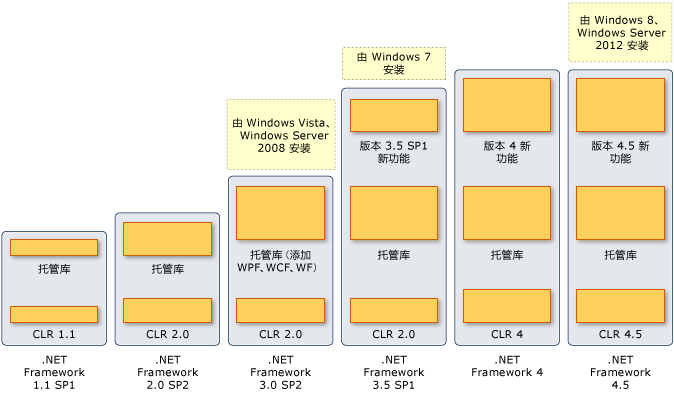
## **NET**

.Net是微软设计的独立于[**操作系统**](http://lib.csdn.net/base/operatingsystem)之上的平台，可以将它看成一套虚拟机，无论机器运行的是什么操作系统，只要该系统安装了.Net框架，便可以运行.Net可执行程序，享受基于.Net的各类[服务](http://www.91now.com/)。上句话是从用户角度出发的观点，如果从Windows系统的角度来理解，.Net就是一系列运行于Ring 3层的DLL文件。  
（1）统一了[编程](http://www.91now.com/)语言：无论程序是用C#，还是C++，或是VB编写，最终都被编译为.Net中间语言IL；  
（2）扩展了PE文件的格式：可执行文件中不再保存机器码，而是IL指令和元[数据](http://www.91now.com/)，部分结构也被改变用于保存.Net的相关信息；  
（3）改变了程序的运行方式：Windows不再直接负责程序的运行，而由.Net框架进行管理，框架中的JIT引擎负责在运行时将IL代码即时编译为本地汇编代码再执行。

### [**NET Framework各版本汇总以及之间的关系**](http://blog.csdn.net/kingmax54212008/article/details/25886345)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **.NET版本** | **1.0** | **1.1** | **2.0** | **3.0** | **3.5** | **4.0** | **4.5** |
| 完整版本 | 1.0.3705.0 | 1.1.4322.573 | 2.0.50727.42 | 3.0.4506.30 | 3.5.21022.8 | 4.0.30319.1 | 4.5.40805 |
| 发布时间 | 2002-02-13 | 2003-04-24 | 2005-11-07 | 2006-11-06 | 2007-11-19 | 2010-04-12 | 2012-05-24 |
| VS开发版本 | VS2002 | VS2003 | VS2005 |  | VS2008 | VS2010 | VS2012 |
| Windows默认安装 |  | Windows Server 2003 | Windows Server 2003 Windows Server 2008 | Windows Vista Windows Server 2008 | Windows 7 Windows Server 2008 R2 |  | Windows 8 Windows Server 2012 |
| 下载 | [.NET Framework 1.0 (SP3)](http://www.microsoft.com/zh-cn/download/details.aspx?id=13181) | [.NET Framework 1.1 (SP1)](http://www.microsoft.com/zh-cn/download/details.aspx?id=33) | [.NET Framework 2.0 (SP2)](http://www.microsoft.com/zh-cn/download/details.aspx?id=1639) | [.NET Framework 3.0 (SP2)](http://msdn.microsoft.com/netframework/bb264589) | [.NET Framework 3.5 (SP1)](http://www.microsoft.com/zh-cn/download/details.aspx?id=25150) | [.NET Framework 4.0](http://www.microsoft.com/zh-cn/download/details.aspx?id=17718) | [.NET Framework 4.5](http://go.microsoft.com/fwlink/?linkid=262043) |
| 说明 | Microsoft Internet Explorer 5.01 或更高版本 | Microsoft Internet Explorer 5.01 或更高版本 | Windows Installer 3.1 或更高版本 Internet Explorer 6.0 或更高版本 |  | 包括 .NET Framework 2.0 Service Pack 2 和 .NET Framework 3.0 Service Pack 2 累积更新 | Windows Installer 3.1 或更高版本 Internet Explorer 5.01 或更高版本 | .NET Framework 4.5 RC 是一个针对 .NET Framework 4 的高度兼容的就地更新。 |
| 支持的windows版本 | Windows 98 Windows NT Windows Server 2000 Windows Server 2003 Windows XP | Windows Server 2000 Windows Server 2003 Windows XP | Windows Server 2003 Windows XP | Windows Server 2003 | Windows Server 2003 Windows Server 2008, Windows Vista Windows XP | Windows XP SP3 Windows Server 2003 SP2 Windows Vista SP1 Windows Server 2008 Windows 7 | Windows Vista SP2 Windows 7 Windows 8 Windows Server 2008 Windows Server 2012 |

#### **版本关系**



.NET Framework 版本 2.0、3.0 和 3.5 是使用 CLR (CLR 2.0) 的相同版本生成的。 每个版本增量地生成于早期 .NET Framework 版本。 在计算机上不可能并排运行版本 2.0、3.0 和 3.5。 在安装 .NET Framework 3.5 SP1 时，会自动安装 2.0 和 3.0 层。 但是，.NET Framework 4 关闭此分层方法。 以 .NET Framework 4 开始，可使用进程内并行承载来在单独的进程中运行多个公共语言运行时 (CLR) 版本。 Apps 的 2.0 ，3.0 和 3.5 版本可以全部在 3.5 版运行，但是，它们在 4 版或更高版本将不起作用。.NET Framework 4.5 是就地更新，替换您的计算机上的 .NET Framework 4。 在安装此更新后，您的 .NET Framework 4 应用程序在无需重新编译的情况下应继续运行。 但是，.NET Framework 中的某些更改可能需要更改您的应用程序代码。 有关更多信息，在 .NET Framework 4.5 中运行现有应用程序前，请参见 在 .NET Framework 4.5 中的应用程序兼容性。 有关安装当前版本的更多信息，请参见 安装 .NET Framework 4.5。 有关对 .NET Framework 的支持的信息，请参见 Microsoft 支持网站上的 Microsoft .NET Framework Support Lifecycle Policy（Microsoft .NET Framework 支持生命周期策略）。

以上是微软官方解释。

我理解的意思是，3.5包含了2.0和3.0，安装了3.5就自动安装了2.0和3.0。这样也说明了为什么3.5的安装包会很大的原因。win7以后好像就不能直接安装2.0只能通过安装3.5来支持2.0的程序。

4.0是一个过渡版本，安装4.5是会覆盖4.0。并且4.0可在4.5环境中运行。

值得注意的是4.5不支持XP系统了。

### **JIT**

JIT 编译 (JIT compilation)，运行时需要代码时，将 Microsoft 中间语言 (MSIL) 转换为机器码的编译。

#### **VS自带工具 IL 中间语言的反汇编**

**查看反汇编的输出。**

## 官方参考

### C#知识普及网站比微软人性化

<http://www.runoob.com/csharp/csharp-attribute.html>

http://c.biancheng.net/view/2878.html

## **第三方库**

### **WPFtools**

1. 在线网普及

<http://www.runoob.com/csharp/csharp-attribute.html>

1. 加解密各种常用算法

http://tool.oschina.net/encrypt?type=4

加解密各种常用算法，DES， base64等。

c#等知识的普及性网站非常好

http://www.runoob.com/csharp/csharp-attribute.html

https://github.com/xceedsoftware/wpftoolkit

https://archive.codeplex.com/?p=wpftoolkit

http://xceed.com/xceed-toolkit-plus-for-wpf/

https://www.cnblogs.com/gnielee/archive/2011/05/10/extended-wpf-toolkit.html

WPFtools界面使用

https://archive.codeplex.com/?p=wpftoolkit

WPFTOOLS的工程下载。

https://blog.csdn.net/lweiyue/article/details/82491173

直接用GRID做表格控制。

## **术语**

time stamp 时间戳

# **常用知识点**

## **nuget程序包还原失败:未能解析此远程名称**

https://blog.csdn.net/u010935620/article/details/99427939

## **未处理异常的捕获，定义自己的异常捕获**

**捕获处理异常后，程序仍会继续运行，而不会打断返回。**

## **NuGet，联机安装，使用常用的类库。**

　　简单的说NuGet可以是我们的工作更方便，当我们的项目里要引用到的一些库时候，比如JQuery、Newtonsoft.Json、 log4net等，我们需要从网上下载这些库，然后依次拷贝到各个项目中，当有的类库有更新时又不得不再重复一遍很是繁琐 ，这时就可以考虑使用NuGet来帮我们管理和更新这些类库，而且更新类库时会自动添加类库的相关引用，方便至极。当然网上一些我们常用的类库更新频率不 是很高而且即便出了新版本我们也没必要总是保持最新，故这点对我们的帮助比较有限，个人认为NuGet最大的好处在于可以搭建自己的类库服务器，想想一下 吧，在一些较大些的公司里面有很多的项目，然后其中有一些是整个组，甚至整个公司通用的类库，当这些类库有更新后我们需要依次拷贝到我们的项目，甚至于有 时候我们自己都搞不清楚各个项目里的版本是否一致，有时偶尔一两个项目忘了复制更新出现莫名其妙的错误，为此头疼不已，现在有了NuGet，世界从此而不 同。

## **接口和抽象类有什么区别**

你选择使用接口和抽象类的依据是什么？   
接口和抽象类的概念不一样。接口是对动作的抽象，抽象类是对根源的抽象。

抽象类表示的是，这个对象是什么。接口表示的是，这个对象能做什么。比如，男人，女人，这两个类（如果是类的话……），他们的抽象类是人。说明，他们都是人。

所以，在高级语言上，一个类只能继承一个类（抽象类）(正如人不可能同时是生物和非生物)，但是可以实现多个接口(吃饭接口、走路接口)。

第一点． 接口是抽象类的变体，接口中所有的方法都是抽象的。而抽象类是声明方法的存在而不去实现它的类。  
第二点． 接口可以多继承，抽象类不行  
第三点． 接口定义方法，不能实现，而抽象类可以实现部分方法。  
第四点． 接口中基本数据类型为static 而抽类象不是的。

当你关注一个事物的本质的时候，用抽象类；当你关注一个操作的时候，用接口。

抽象类的功能要远超过接口，但是，定义抽象类的代价高。因为高级语言来说（从实际设计上来说也是）每个类只能继承一个类。在这个类中，你必须继承或编写出其所有子类的所有共性。虽然接口在功能上会弱化许多，但是它只是针对一个动作的描述。而且你可以在一个类中同时实现多个接口。在设计阶段会降低难度的

## [C#操作符??和?:](http://www.cnblogs.com/refactor/archive/2012/05/30/2526147.html)

先看如下代码：

|  |
| --- |
| string strParam = Request.Params["param"];  if ( strParam== null )  {      strParam= "";  }    string strParam=    Request.Params["param"] == null ? "": Request.Params["param"]; |

|  |
| --- |
| 你发现，尽管实现的同一个功能，但第二段代码要比第一段代码是不是更简洁。  这里的” ？：“就是一个三元的操作符。利用这个三元操作符也称条件操作符，  就很简单的把一个if语句给实现了。  在使用条件操作符的时候要注意，假定条件表达式是这样 x？y：z，  表达式类型的只能是使y 和z的变量类型之一，换句话说就是 y和z的变量类型决定这个条件表达式的类型，  无论y和z是同一类型，还是y和z不是同一类型。 如果表达式的类型和赋值类型不一致，发生编译时错误 ，例如：如果你想实现如下代码：  string strTemp ;  object obj;  ....    if(strTemp ==  "")            obj   =   DBNull.Value;    else            obj   =   strTemp ;  然后为了简单，改成下面这种形式：  string strTemp;  object obj = strTemp == ""?DBNull.Value:strTemp;  你编译的时候，会报错。因为DBNull和string之间没隐式转换，修改成如下代码就可以了：  object obj = strTemp == ""? DBNull.Value: (object)strTemp;  另外条件操作符是向右关联的，从左到右进行计算，于是a?b:c?d:e 应理解为 a?b:(c?d:e)。  上面提到了DBNull，在c#2.0里面实现了Nullable数据类型，别小看这个数据类型，其实非常有用，  从此我们就可以放弃对object数据进行 if（object==null）这样的判断了。同时你会发现，  在数据库应用上，数据库中有的字段允许取空值，在C#世界中引入nullable类型正好与之对应  ，从（C#）对象映射到关系数据库对象，或者反过来，转换都是比较平滑。   为了实现Nullable数据类型转换成non-Nullable型数据，就有了一个这样的操作符”？？（两个问号）“，双问号操作符意思是取所赋值??左边的，如果左边为null，取所赋值??右边的，  比如int y = x ?? -1 如果x为空，那么y的值为-1.  于是这时候就可以把最上面第二段代码改成：  string strParam= Request.Params["param"]?? "";  嘿嘿，是不是更简洁了。  另外还有”？（单问号）“修饰符，是System.Nullable的缩写形式，比如int?代表是可空的整形，  例如：int? a = 1 或者 int? b=null。Nullable类型具有一个HasValue的bool类型只读属性，  当Nullable类型实例的该属性为true时，则表示该实例是非空实例，包含一个已知值Value，  HasValue为false时，访问Value属性将导致System.InvalidOperationException。 |

## 对象CLONE

<http://www.cnblogs.com/gengaixue/archive/2012/07/11/2586411.html>

**Entityframework**

ADO.NET Entity Framework

## [构造函数定义为private，protected](http://www.cnblogs.com/this-543273659/archive/2011/08/02/2125487.html)

将构造函数，析构函数声明为私有和保护的，那么对象如何创建？

已经不能从外部调用构造函数了，但是对象必须被构造，应该如何解决，麻烦大家帮忙说明，关于构造，析构函数声明为私有和保护时的用法？？？

提出这个问题，说明你已经对c++有所思考了。

从语法上来讲，一个函数被声明为protected或者private，那么这个函数就不能从“外部”直接被调用了。

对于protected的函数，子类的“内部”的其他函数可以调用之。

而对于private的函数，只能被本类“内部”的其他函数说调用。

语法上就是这么规定的，你肯定也知道的咯。

那么为什么有时候将构造函数或者析构函数声明为protected的或者private的？

通常使用的场景如下：

1。如果你不想让外面的用户直接构造一个类（假设这个类的名字为A）的对象，而希望用户只能构造这个类A的子类，那你就可以将类A的构造函数/析构函数声明为protected，而将类A的子类的构造函数/析构函数声明为public。例如：

class A

{ protected: A(){}

public: ....

};

calss B : public A

{ public: B(){}

....

};

A a; // error

B b; // ok

2. 如果将构造函数/析构函数声明为private，那只能这个类的“内部”的函数才能构造这个类的对象了。这里所说的“内部”不知道你是否能明白，下面举个例子吧。

class A

{

private:

A(){ }

~A(){ }

public:

void Instance()//类A的内部的一个函数

{

A a;

}

};

上面的代码是能通过编译的。上面代码里的Instance函数就是类A的内部的一个函数。Instance函数体里就构建了一个A的对象。

但是，这个Instance函数还是不能够被外面调用的。为什么呢？

如果要调用Instance函数，必须有一个对象被构造出来。但是构造函数被声明为private的了。外部不能直接构造一个对象出来。

A aObj; // 编译通不过

aObj.Instance();

但是，如果Instance是一个static静态函数的话，就可以不需要通过一个对象，而可以直接被调用。如下：class A

{

private:

A():data(10){ cout << "A" << endl; }

~A(){ cout << "~A" << endl; }

public:

static A& Instance()

{

static A a;

return a;

}

void Print()

{

cout << data << endl;

}

private:

int data;

};

A& ra = A::Instance();

ra.Print();

上面的代码其实是设计模式singleton模式的一个简单的C++代码实现。

还有一个情况是：通常将拷贝构造函数和operator=(赋值操作符重载)声明成private，但是没有实现体。

这个的目的是禁止一个类的外部用户对这个类的对象进行复制动作。

细节请看《effective C++》里面的一个条款。具体哪个条款不记得了。

## Lambda 表达式

Lambda 表达式（C# 编程指南）

匿名方法、代理方法。

## As IS

Is：检查对象是否与给定的类型兼容。例如，下面的代码可以确定MyObject类型的一个实例，或者对象是否从MyObject派生的一个类型：

if(obj is MyObject){}  
如果所提供的表达式非空，并且所提供的对象可以强制转换为所提供的类型而不会导致引发异常，则 is 表达式的计算结果将是 true。  
  
如果已知表达式始终是true或始终是false，则is关键字将导致编译时警告，但是通常在运行时才计算类型兼容性。  
  
注意：is运行符不能重载，is运行符只考虑引用转换、装箱转换和取消装箱转换。不考虑其它转换，如果用户定义转换。在Is运算符的左侧不允许使用匿名方法。lambda表达式属于例外。  
  
Object myObject = new Object();   
Boolean b1 = (myObject is Object); true.   
Boolean b2 = (myObject is Employee); false.  
如果对象引用是null，is运算符总是返回false，因为没有可检查其类型的对象。  
  
is运算符通常像下面这样使用：  
  
if (myObject is Employee)  
{  
  
Employee myEmployee = (Employee)myObject;  
  
}

在这段代码中，CLR实际会检查两次对象的类型。is运算符首先核实myObject是否兼容于Employee类型。如果是，那么在if语句内部执行转 换型，CLR会再次核实myObject是否引用一个Employee。CLR的类型检查增加了安全性，但这样对性能造成一定影响。这是因为CLR首先必 须判断变量(myObject)引用的对象的实际类型。然后，CLR必须遍历继承层次结构，用每个基类型去核对指定的类型（Employee）。由于这是 一个相当常用的编程模式，所以C#专门提供了as运算符，目的就是简化这种代码写法，同时提升性能。  
  
as:用于检查在兼容的引用类型之间执行某些类型的转换。  
  
Employee myEmployee = myObject as Employee;  
  
if (myEmployee != null)  
  
{ }  
  
在这段代码中，CLR核实myObject是否兼容于Employee类型；如果是，as会返回对同一个对象的一个非null  
  
的引用。如果myObject不兼容于Employee类型，as运算符会返回null。  
  
注意：as运算符造成CLR只校验一次对象的类型。if语句只是检查myEmployee是否为null。这个检查的速度比校验对象的类型快得多。  
  
as 运算符的工作方式与强制类型转换一样，只是它永远不会抛出一个异常。相反，如果对象不能转换，结果就是null。所以，正确的做法是检查最终生成的一引用 是否为null。如果企图直接使用最终生成的引用，会抛出一个System.NullReferenceException异常。以下代码对此进行了演 示：  
  
Object o = new Object(); 新建一个Object对象。  
  
Employee e = o as Employee; 将o转型为一个Employee  
  
e.ToString(); 访问e会抛出一个NullReferenceException异常  
  
备注：  
  
as运算符类似于强制转换操作。但是无法进行转换，则as返回null而非引发异常。  
  
示例：  
expression as Type它等效于以下表达式，但是只计算一次expression。  
expression is Type ?(Type)expression : (Type)null  
注意：as运算符只执行引用转换和装箱转换。as运算符无法执行其它转换，如果用户定义的转换，这类转换应使用强制转换表达式来执行。

## IOC控制反转

IOC：Inversion of Control 控制反转

DI:Dependency Injection 依赖注入

控制反转，从字面意思来看，就是控制权又被动变主动，最后又变回被动。

举个例子：

上面的整个过程就完成了一次IOC，从上面可以看出，IOC的基本思想是控制权的转换过程。

IOC模式系统中，通过引入实现IOC模式的IOC容器，即可由IOC容器来管理对象的生命周期、依赖关系等，从而使得应用程序的配置和依赖性规范与实际 的应用程序代码分开。其中一个特点就是通过文本的配置文件进行应用程序组件间相互关系的配置，而不用重新修改并编译具体的代码。

可以把IoC模式看做是工厂模式的升华，可以把IoC看作是一个大工厂，只不过这个大工厂里要生成的对象都是在XML文件中给出定义的，然后利用Java 的“反射”编程，根据XML中给出的类名生成相应的对象。从实现来看，IoC是把以前在工厂方法里写死的对象生成代码，改变为由XML文件来定义，也就是 把工厂和对象生成这两者独立分隔开来，目的就是提高灵活性和可维护性。

## MEF

Managed Extensibility Framework（MEF）是.NET平台下的一个扩展性管理框架，它是一系列特性的集合，包括依赖注入（DI）以及Duck Typing等。MEF为开发人员提供了一个工具.

### AvalonDock

  AvalonDock 是一个.NET库，用于在停靠模式布局（docking）中排列一系列WPF/WinForm控件。最新发布的版本原生支持MVVM框架、Aero Snap特效并具有更好的性能。

AvalonDock 2.0版本已经发布了，新版本是用MVVM框架重新编写，似乎也用了Command（命令）模式。2.0版的文档尚未发布，但你可以参考 Avalon.TestApp 或者2.0版源码中的Avalon.MVVMTestApp文件夹来查看新的API。

### MahApps.Metro

这是一个Metro样式的开源项目，应用该项目可以使你的软件具有metro的风格，具体就不多说了。

## 针对C#作用域

值类型的只在于函数内部等。作用域同C++

2、引用型的，一直存在于被引用型的生命空间。

在方法中生成的一个新的对象，是可以被引用到其他对象，一直在其他对象里。

3、引用型，受到GC的回收。(Garbage Collector（垃圾收集器，在不至于混淆的情况下也成为GC）

数组也是引用型传递，在方法中，可以对数组中的内容进行操作，但不能更换数组实例，也就是所不能在方法中，对数组进行对象赋值，方法返回后

还是原先的数组。

## VS 拷贝和后期处理

copy $(ProjectDir)..\..\library\\*.\* $(ProjectDir)1.txt

解释：拷贝projectdir 为基础的上一层的上一层的library目录下的问题件到

增加相对拷贝路径。

在解决方案属性中，增加依赖项，进行启动顺序控制。

## 资源文件

针对资源以及多语言调用等。

资源文件resx的默认空间，一定要看在哪个文件夹下。

在进行new ResourceManager(baseResourceName, Assembly.LoadFrom(fileName));

时

baseResourceName = 项目名称. + 所在目录名称.（如果有）+ 资源文件名.+ 多语言名称.(如果有）

### [GetManifestResourceStream得到的Stream是null的解决](http://www.cnblogs.com/Lisen/archive/2009/09/15/1566799.html)

Ans: 参考

<http://blog.csdn.net/blue_sky_blue_heart/archive/2005/12/17/555139.aspx>

System.Reflection.Assembly asm = System.Reflection.Assembly.GetExecutingAssembly();  
// Note that the name for the embedded resource is case sensitive and must match the file name.  
Bitmap bmp = new Bitmap(asm.GetManifestResourceStream(asm.GetName().Name + "." + "pic.png"));  
e.Graphics.DrawImage(bmp, 5, 8, new Rectangle(0, 0, bmp.Width, bmp.Height), GraphicsUnit.Pixel);

其实最主要的是在将图片资源的属性设置为“嵌入的资源”：“属性-生成操作-嵌入的资源”

注意GetManifestResourceStream函数的参数：asm.GetName().Name获得的只是当前命名空间的名称，如果在“pic.png”图片在当前工程的Photo目录下，那么该参数就该写成asm.GetName().Name + "." + "Photo" + "." + " pic.png"

## 字符串操作

### 转义输出

stringbuild appendformat 中的转义

如果想原样输出一个“{”的话，必须要写两个花括号“{{”。

“}”的情况也一样，必须写成“}}”。

在这里使用转义“\{”或者“\}”是不行的。

### [C#数字类型输出字符串时保留指定小数位数的方法](https://www.cnblogs.com/yubufan/p/3552789.html)

1、使用占位符：

1）  
float f = 321.12345F;  
f.ToString("0.00");  
这样做无论f是不是整数，都将加上2位小数。  
  
2）  
float f = 321.12345F;  
f.ToString(".##");  
这样做最多显示2位小数

占位符的定义如下：

|  |  |
| --- | --- |
| (**0**) | Digit placeholder. Display a digit or a zero. If the expression has a digit in the position where the 0 appears in the format string, display it; otherwise, display a zero in that position.  If the number has fewer digits than there are zeros (on either side of the decimal) in the format expression, display leading or trailing zeros. If the number has more digits to the right of the decimal separator than there are zeros to the right of the decimal separator in the format expression, round the number to as many decimal places as there are zeros. If the number has more digits to the left of the decimal separator than there are zeros to the left of the decimal separator in the format expression, display the extra digits without modification. |
| (**#**) | Digit placeholder.Display a digit or nothing. If the expression has a digit in the position where the # appears in the format string, display it; otherwise, display nothing in that position.  This symbol works like the 0 digit placeholder, except that leading and trailing zeros aren't displayed if the number has the same or fewer digits than there are # characters on either side of the decimal separator in the format expression. |
| (**.**) | Decimal placeholder. In some locales, a comma is used as the decimal separator. The decimal placeholder determines how many digits are displayed to the left and right of the decimal separator. If the format expression contains only number signs to the left of this symbol, numbers smaller than 1 begin with a decimal separator. To display a leading zero displayed with fractional numbers, use 0 as the first digit placeholder to the left of the decimal separator. The actual character used as a decimal placeholder in the formatted output depends on the Number Format recognized by your system. |
| (**,**) | Thousand separator. In some locales, a period is used as a thousand separator. The thousand separator separates thousands from hundreds within a number that has four or more places to the left of the decimal separator. Standard use of the thousand separator is specified if the format contains a thousand separator surrounded by digit placeholders (**0** or **#**). The actual character used as the thousand separator in the formatted output depends on the Number Format recognized by your system. |
| ("ABC") | Literal string. You can place literal strings on either side of numeric placeholders. For example, you could define a format expression as "The formatted number is: ###,###.#0" |

在msdn中搜索FormatString property可得到

2、使用标准数字格式字符串

如：

float f = 321.12345F;  
f.ToString("F3");

结果为321.123

标准数字格式字符串的定义如下：

### 字符格式化输出

|  |  |  |
| --- | --- | --- |
| **格式说明符** | **名称** | **说明** |
| C 或 c | 货币 | 数字转换为表示货币金额的字符串。转换由用于格式化数字的 NumberFormatInfo 对象的货币格式信息控制。精度说明符指示所需的小数位数。如果省略精度说明符，则使用 NumberFormatInfo 给定的默认货币精度。 |
| D 或 d | 十进制数 | 只有整型才支持此格式。数字转换为十进制数字 (0-9) 的字符串，如果数字为负，则前面加负号。精度说明符指示结果字符串中所需的最少数字个数。如果需要的话，则用零填充该数字的左侧，以产生精度说明符给定的数字个数。 |
| E 或 e | 科学计数法（指数） | 数字转换为“-d.ddd…E+ddd”或“-d.ddd…e+ddd”形式的字符串，其中每个“d”表示一个数字 (0-9)。如果该数字为负，则该字符串以减号开头。小数点前总有一个数字。精度说明符指示小数点后所需的位数。如果省略精度说明符，则使用默认值，即小 数点后六位数字。格式说明符的大小写指示在指数前加前缀“E”还是“e”。指数总是由正号或负号以及最少三位数字组成。如果需要，用零填充指数以满足最少 三位数字的要求。 |
| F 或 f | 固定点 | 数字转换为“-ddd.ddd…”形式的字符串，其中每个“d”表示一个数字 (0-9)。如果该数字为负，则该字符串以减号开头。精度说明符指示所需的小数位数。如果忽略精度说明符，则使用 NumberFormatInfo 给定的默认数值精度。 |
| G 或 g | 常规 | 根据数字类型以及是否存在精度说明符，数字会转换为固定点或科学记数法的最紧凑形式。如果精度说明符被省略或为零，则数字的类型决定默认精度，如下表所示。   * Byte 或 SByte：3 * Int16 或 UInt16：5 * Int32 或 UInt32：10 * Int64 或 UInt64：19 * Single：7 * Double：15 * Decimal：29   如果用科学记数法表示数字时指数大于 -5 而且小于精度说明符，则使用固定点表示法；否则使用科学记数法。如果要求有小数点，并且忽略尾部零，则结果包含小数点。如果精度说明符存在，并且结果的有 效数字位数超过指定精度，则通过舍入删除多余的尾部数字。使用科学记数法时，如果格式说明符是“G”，结果的指数带前缀“E”；如果格式说明符是“g”， 结果的指数带前缀“e”。  上述规则有一个例外：如果数字是 Decimal 而且省略精度说明符时。在这种情况下总使用固定点表示法并保留尾部零。 |
| N 或 n | 数字 | 数字转换为“-d,ddd,ddd.ddd…”形式的字符串，其中每个“d”表示一个数字 (0-9)。如果该数字为负，则该字符串以减号开头。小数点左边每三个数字之间插入一个千位分隔符。精度说明符指示所需的小数位数。如果忽略精度说明符， 则使用 NumberFormatInfo 给定的默认数值精度。 |
| P 或 p | 百分比 | 数字转换为由 NumberFormatInfo.PercentNegativePattern 属性或 NumberFormatInfo.PercentPositivePattern 属性定义的、表示百分比的字符串。如果数字为负，则产生的字符串由 PercentNegativePattern 定义并以负号开头。已转换的数字乘以 100 以表示为百分比。精度说明符指示所需的小数位数。如果省略精度说明符，则使用 NumberFormatInfo 给定的默认数值精度。 |
| R 或 r | 往返过程 | 往返过程说明符保证转换为字符串的数值再次被分析为相同的数值。使用此说明符格式化数值时，首先使用常规格式对其进行测试：Double 使用 15 位精度，Single 使用 7 位精度。如果此值被成功地分析回相同的数值，则使用常规格式说明符对其进行格式化。但是，如果此值未被成功地分析为相同数值，则它这样格式 化：Double 使用 17 位精度，Single 使用 9 位精度。虽然精度说明符可以追加到往返过程格式说明符，但它将被忽略。使用此说明符时，往返过程优先于精度。此格式仅受浮点型支持。 |
| X 或 x | 十六进制数 | 数字转换为十六进制数字的字符串。格式说明符的大小写指示对大于 9 的十六进制数字使用大写字符还是小写字符。例如，使用“X”产生“ABCDEF”，使用“x”产生“abcdef”。精度说明符指示结果字符串中所需的最 少数字个数。如果需要的话，则用零填充该数字的左侧，以产生精度说明符给定的数字个数。只有整型才支持此格式。 |

在msdn中搜索“标准数字格式字符串”可得到

3、使用System.Math.Round函数可以指定小数位数。

例子：

#### 字符型转换 转为字符串

12345.ToString("n"); //生成 12,345.00

12345.ToString("C"); //生成 ￥12,345.00

12345.ToString("e"); //生成 1.234500e+004

12345.ToString("f4"); //生成 12345.0000

12345.ToString("x"); //生成 3039 (16进制)

12345.ToString("p"); //生成 1,234,500.00%

### 字符串编码转换

System.Text.Encoding.Default.GetBytes(变量)

字码转换 转为比特码

如：byte[] bytStr = System.Text.Encoding.Default.GetBytes(str);

然后可得到比特长度：

len = bytStr.Length;

### char 的判断操作

#### char.IsWhiteSpce(字串变量，位数)——逻辑型

查指定位置是否空字符；

如：

string str="中国 人民";

Response.Write(char.IsWhiteSpace(str,2)); //结果为：True, 第一个字符是0位，2是第三个字符。

#### char.IsPunctuation('字符') --逻辑型

查字符是否是标点符号

如：Response.Write(char.IsPunctuation('A')); //返回：False

还有很多

### String 字符串操作

#### 字符串比较

字串对比一般都用: if(str1==str2){ } , 但还有别的方法:

(1)、

string str1; str2

//语法: str1.EndsWith(str2); \_\_检测字串str1是否以字串str2结尾,返回布尔值.如:

if(str1.EndsWith(str2)){ Response.Write("字串str1是以"+str2+"结束的"); }

(2)、

//语法:str1.Equals(str2); \_\_检测字串str1是否与字串str2相等,返回布尔值,用法同上.

(3)、

//语法 Equals(str1,str2); \_\_检测字串str1是否与字串str2相等,返回布尔值,用法同上.

#### 索引查找

IndexOf() 、LastIndexOf()

查找字串中指定字符或字串首次（最后一次）出现的位置,返回索引值，如：

str1.IndexOf("字")； //查找“字”在str1中的索引值（位置）

str1.IndexOf("字串")；//查找“字串”的第一个字符在str1中的索引值（位置）

str1.IndexOf("字串",3,2)；//从str1第4个字符起，查找2个字符，查找“字串”的第一个字符在str1中的索引值（位置）

#### Insert()

在字串中指定索引位插入指定字符。如：

str1.Insert(1,"字");在str1的第二个字符处插入“字”，如果str1="中国"，插入后为“中字国”；

#### PadLeft()、PadRight()

在字串左（或右）加空格或指定char字符，使字串达到指定长度，如：

<%

string str1="中国人";

str1=str1.PadLeft(10,'1'); //无第二参数为加空格

Response.Write(str1); //结果为“1111111中国人” ， 字串长为10

%>

#### 以什么开头

string.StartsWith()

## C#采用正则表达式计算字符串数学公式计算类

<https://blog.csdn.net/challengeshackers/article/details/7185403>

## C# datetime操作全

https://blog.csdn.net/qq\_34307432/article/details/80267000

DateTime.Now.ToShortTimeString()

DateTime dt = DateTime.Now;

dt.ToString();//2005-11-5 13:21:25

dt.ToFileTime().ToString();//127756416859912816

dt.ToFileTimeUtc().ToString();//127756704859912816

dt.ToLocalTime().ToString();//2005-11-5 21:21:25

dt.ToLongDateString().ToString();//2005年11月5日

dt.ToLongTimeString().ToString();//13:21:25

dt.ToOADate().ToString();//38661.5565508218

dt.ToShortDateString().ToString();//2005-11-5

dt.ToShortTimeString().ToString();//13:21 dt.ToUniversalTime().ToString();//2005-11-5 5:21:25

dt.Year.ToString();//2005

dt.Date.ToString();//2005-11-5 0:00:00

dt.DayOfWeek.ToString();//Saturday

dt.DayOfYear.ToString();//309

dt.Hour.ToString();//13

dt.Millisecond.ToString();//441

dt.Minute.ToString();//30

dt.Month.ToString();//11

dt.Second.ToString();//28

dt.Ticks.ToString();//632667942284412864

dt.TimeOfDay.ToString();//13:30:28.4412864

dt.ToString();//2005-11-5 13:47:04

dt.AddYears(1).ToString();//2006-11-5 13:47:04

dt.AddDays(1.1).ToString();//2005-11-6 16:11:04

dt.AddHours(1.1).ToString();//2005-11-5 14:53:04

dt.AddMilliseconds(1.1).ToString();//2005-11-5 13:47:04

dt.AddMonths(1).ToString();//2005-12-5 13:47:04

dt.AddSeconds(1.1).ToString();//2005-11-5 13:47:05

dt.AddMinutes(1.1).ToString();//2005-11-5 13:48:10

dt.AddTicks(1000).ToString();//2005-11-5 13:47:04

dt.CompareTo(dt).ToString();//0

dt.Add(?).ToString();//问号为一个时间段

dt.Equals("2005-11-6 16:11:04").ToString();//False

dt.Equals(dt).ToString();//True

dt.GetHashCode().ToString();//1474088234

dt.GetType().ToString();//System.DateTime

dt.GetTypeCode().ToString();//DateTime

dt.GetDateTimeFormats('s')[0].ToString();//2005-11-05T14:06:25

dt.GetDateTimeFormats('t')[0].ToString();//14:06

dt.GetDateTimeFormats('y')[0].ToString();//2005年11月

dt.GetDateTimeFormats('D')[0].ToString();//2005年11月5日

dt.GetDateTimeFormats('D')[1].ToString();//2005 11 05

dt.GetDateTimeFormats('D')[2].ToString();//星期六 2005 11 05

dt.GetDateTimeFormats('D')[3].ToString();//星期六 2005年11月5日

dt.GetDateTimeFormats('M')[0].ToString();//11月5日

dt.GetDateTimeFormats('f')[0].ToString();//2005年11月5日 14:06

dt.GetDateTimeFormats('g')[0].ToString();//2005-11-5 14:06

dt.GetDateTimeFormats('r')[0].ToString();//Sat, 05 Nov 2005 14:06:25 GMT

string.Format("{0:d}",dt);//2005-11-5

string.Format("{0}",dt);//2005年11月5日

string.Format("{0:f}",dt);//2005年11月5日 14:23

string.Format("{0:F}",dt);//2005年11月5日 14:23:23

string.Format("{0:g}",dt);//2005-11-5 14:23

string.Format("{0:G}",dt);//2005-11-5 14:23:23

string.Format("{0:M}",dt);//11月5日

string.Format("{0:R}",dt);//Sat, 05 Nov 2005 14:23:23 GMT

string.Format("{0:s}",dt);//2005-11-05T14:23:23

string.Format("{0:t}",dt);//14:23

string.Format("{0:T}",dt);//14:23:23

string.Format("{0:u}",dt);//2005-11-05 14:23:23Z

string.Format("{0:U}",dt);//2005年11月5日 6:23:23

string.Format("{0:Y}",dt);//2005年11月

string.Format("{0}",dt);//2005-11-5 14:23:23

string.Format("{0:yyyyMMddHHmmssffff}",dt);

计算2个日期之间的天数差

-----------------------------------------------

DateTime dt1 = Convert.DateTime("2007-8-1");

DateTime dt2 = Convert.DateTime("2007-8-15");

TimeSpan span = dt2.Subtract(dt1);

int dayDiff = span.Days + 1;

计算某年某月的天数

-----------------------------------------------

int days = DateTime.DaysInMonth(2007, 8);

days = 31;

给日期增加一天、减少一天

-----------------------------------------------

DateTime dt =DateTime.Now;

dt.AddDays(1); //增加一天

dt.AddDays(-1);//减少一天

其它年份方法类似...

Oracle SQL里转换日期函数

-----------------------------------------------

to\_date("2007-6-6",'YYYY-MM-DD");

to\_date("2007/6/6",'yyyy/mm/dd");

如下一组数据,如何查找表里包含9月份的记录:

CGGC\_STRATDATE CGGC\_ENDDATE

=========================================

2007-8-4 2007-9-5

2007-9-5 2007-9-20

2007-9-22 2007-10-5

SELECT \* FROM TABLE

(TO\_DATE('2007/9/1','yyyy/mm/dd') BETWEEN CGGC\_STRATDATE

AND CGGC\_ENDDATE OR CGGC\_STRATDATE >=TO\_DATE('2007/9/1','yyyy/mm/dd')

AND CGGC\_ENDDATE<=TO\_DATE('2007/9/30','yyyy/mm/dd') "

OR TO\_DATE('2007/9/30','yyyy/mm/dd') BETWEEN CGGC\_STRATDATE

AND CGGC\_ENDDATE) ORDER BY CGGC\_STRATDATE ASC

//本年度，用ToString的字符格式化我们也很容易地算出本年度的第一天和最后一天

DateTime.Parse(DateTime.Now.ToString("yyyy-01-01")).ToShortDateString();

DateTime.Parse(DateTime.Now.ToString("yyyy-01-01")).AddYears(1).AddDays(-1).ToShortDateString();

//上年度，不用再解释了吧

DateTime.Parse(DateTime.Now.ToString("yyyy-01-01")).AddYears(-1).ToShortDateString();

DateTime.Parse(DateTime.Now.ToString("yyyy-01-01")).AddDays(-1).ToShortDateString();

//下年度

DateTime.Parse(DateTime.Now.ToString("yyyy-01-01")).AddYears(1).ToShortDateString();

DateTime.Parse(DateTime.Now.ToString("yyyy-01-01")).AddYears(2).AddDays(-1).ToShortDateString();

//本季度，很多人都会觉得这里难点，需要写个长长的过程来判断。其实不用的，我们都知道一年四个季度，一个季度三个月

//首先我们先把日期推到本季度第一个月，然后这个月的第一天就是本季度的第一天了

DateTime.Now.AddMonths(0 - ((DateTime.Now.Month - 1) % 3)).ToString("yyyy-MM-01");

//同理，本季度的最后一天就是下季度的第一天减一

DateTime.Parse(DateTime.Now.AddMonths(3 - ((DateTime.Now.Month - 1) % 3)).ToString("yyyy-MM-01")).AddDays(-1).ToShortDateString();

//下季度

DateTime.Now.AddMonths(3 - ((DateTime.Now.Month - 1) % 3)).ToString("yyyy-MM-01");

DateTime.Parse(DateTime.Now.AddMonths(6 - ((DateTime.Now.Month - 1) % 3)).ToString("yyyy-MM-01")).AddDays(-1).ToShortDateString();

//上季度

DateTime.Now.AddMonths(-3 - ((DateTime.Now.Month - 1) % 3)).ToString("yyyy-MM-01"); DateTime.Parse(DateTime.Now.AddMonths(0 - ((DateTime.Now.Month - 1) % 3)).ToString("yyyy-MM-01")).AddDays(-1).ToShortDateString();

//如何获得当月有多少天

int m=System.DateTime.DaysInMonth(System.DateTime.Now.Year,System.DateTime.Now.Month);

//获取某月的实际工作日(即不包括周六日)

//调用

//int days =getDays(System.DateTime.Now));

private int getDays(System.DateTime date1)

{

int m=System.DateTime.DaysInMonth(date1.Year,date1.Month);

int mm=0;

for(int i=1;i<=m;i++)

{

System.DateTime date=Convert.ToDateTime(date1.Year+"-"+date1.Month+"-"+i);

switch (date.DayOfWeek)

{

case System.DayOfWeek.Monday:

case System.DayOfWeek.Thursday:

case System.DayOfWeek.Tuesday:

case System.DayOfWeek.Wednesday:

case System.DayOfWeek.Friday:

mm=mm+1;

break;

}

}

return mm;

}

获得任意两日期之间的有效工作日(不包括周六日) 获得任意两日期之间的有效工作日(不包括周六日)#region 获得任意两日期之间的有效工作日(不包括周六日)

//调用

//DateTime date1=Convert.ToDateTime("2005-10-20");

//DateTime date2=Convert.ToDateTime("2005-11-01");

//int days =getDays(date1,date2);

private int getDays(System.DateTime date1,System.DateTime date2)

{

string m=DateDiff(EnumDateCompare.day,date1,date2).ToString("f0");

int mm=0;

for(int i=0;i<=Convert.ToInt32(m);i++)

{

System.DateTime date=Convert.ToDateTime(date1.AddDays(i));

switch (date.DayOfWeek)

{

case System.DayOfWeek.Monday:

case System.DayOfWeek.Thursday:

case System.DayOfWeek.Tuesday:

case System.DayOfWeek.Wednesday:

case System.DayOfWeek.Friday:

mm=mm+1;

break;

}

}

return mm;

}

#endregion

格式输出

格式输出#region 格式输出

private void Page\_Load(object sender, System.EventArgs e)

{

System.Globalization.DateTimeFormatInfo myDTFI = new System.Globalization.CultureInfo( "en-US", false ).DateTimeFormat;

//中国用zh-cn

DateTime myDT =System.DateTime.Now;

Response.Write(myDT.ToString("f",myDTFI));

/\*\*//\*\*//\*\*//\*

This code produces the following output.

FORMAT en-US EXAMPLE

CHAR VALUE OF ASSOCIATED PROPERTY, IF ANY

d 1/3/2002 M/d/yyyy (ShortDatePattern)

D Thursday, January 03, 2002 dddd, MMMM dd, yyyy (LongDatePattern)

f Thursday, January 03, 2002 12:00 AM

F Thursday, January 03, 2002 12:00:00 AM dddd, MMMM dd, yyyy h:mm:ss tt (FullDateTimePattern)

g 1/3/2002 12:00 AM

G 1/3/2002 12:00:00 AM

m January 03 MMMM dd (MonthDayPattern)

M January 03 MMMM dd (MonthDayPattern)

r Thu, 03 Jan 2002 00:00:00 GMT ddd, dd MMM yyyy HH':'mm':'ss 'GMT' (RFC1123Pattern)

R Thu, 03 Jan 2002 00:00:00 GMT ddd, dd MMM yyyy HH':'mm':'ss 'GMT' (RFC1123Pattern)

s 2002-01-03T00:00:00 yyyy'-'MM'-'dd'T'HH':'mm':'ss (SortableDateTimePattern)

t 12:00 AM h:mm tt (ShortTimePattern)

T 12:00:00 AM h:mm:ss tt (LongTimePattern)

u 2002-01-03 00:00:00Z yyyy'-'MM'-'dd HH':'mm':'ss'Z' (UniversalSortableDateTimePattern)

U Thursday, January 03, 2002 8:00:00 AM

y January, 2002 MMMM, yyyy (YearMonthPattern)

Y January, 2002 MMMM, yyyy (YearMonthPattern)

\*/

}

#endregion

获得本周的周六和周日

ConvertDateToWeek#region ConvertDateToWeek

public static void ConvertDateToWeek(DateTime date,out DateTime firstdate,out DateTime lastdate)

{

DateTime first=System.DateTime.Now;

DateTime last=System.DateTime.Now;

switch (date.DayOfWeek)

{ case System.DayOfWeek.Monday:

first=date.AddDays(-1);

last=date.AddDays(5);

break;

case System.DayOfWeek.Tuesday:

first=date.AddDays(-2);

last=date.AddDays(4);

break;

case System.DayOfWeek.Wednesday:

first=date.AddDays(-3);

last=date.AddDays(3);

break;

case System.DayOfWeek.Thursday:

first=date.AddDays(-4);

last=date.AddDays(2);

break;

case System.DayOfWeek.Friday:

first=date.AddDays(-5);

last=date.AddDays(1);

break;

case System.DayOfWeek.Saturday:

first=date.AddDays(-6);

last=date;

break;

case System.DayOfWeek.Sunday:

first=date;

last=date.AddDays(6);

break;

}

firstdate=first;

lastdate=last;

}

#endregion

//调用

DateTime firstdate=System.DateTime.Now;

DateTime lastdate=System.DateTime.Now;

ConvertDateToWeek(date,out firstdate,out lastdate);

获得当前日期是该年度的第几周

DateTime dt = Convert.ToDateTime("2006-05-01");

int weeks = dt.DayOfYear / 7 + 1;

## C# 常用函数库

<https://blog.csdn.net/slim_cx/article/details/6399412>

## c#重载和多态性

### object类

Object 类是 [C#](http://c.biancheng.net/csharp/) 语言中最原始、最重要的类，是所有类的“祖先”，每个 C# 类都是它的子类，它实现了每个类都必须具有的基本方法。  
  
这里指的“所有类”，即不管是 C# 系统所提供的标准类，还是用户自行編写的类，都是从Object类直接或间接继承而来，它是类层次结构中的顶级类，即 C# 树型类层次结构的“根”。  
  
Object 类中的属性和方法可以被用到任何类。  
  
当編程者定义一个类时没有使用关键字 Extends 指明它的父类，则編译器认为该类从 Object 类继承而来。  
  
但 Object 类的方法所实现的功能很有限，都是由编程者在自行设计的 class 类型的类体内编写与原型完全相同的方法来覆盖它们，以实现用户所要求的有用功能。  
  
在 Object 类中提供了 4 个常用的方法，即 Equals、GetHashCode、GetType 以及 ToString 方法。

#### GetHashCode

[C#](http://c.biancheng.net/csharp/) GetHashCode 方法返回当前 System.Object 的哈希代码，每个对象的哈希值都是固定的。

会有重复字段？

### 重载字段

/// <summary>

///

/// </summary>

public class Cdxmc4500Base : CDeviceBase

{

private int \_id;

public PhysicsIdSetting SmidIndex { get; set; }

public new int Id

{

get; set;

}

public Cdxmc4500Base()

{

Id = 100;

}

}

//通过new 可以改变基类的属性类型。

public new int Id

{

get; set;

}

### 重载运算符

https://docs.microsoft.com/zh-cn/dotnet/csharp/programming-guide/statements-expressions-operators/overloadable-operators

<https://www.cnblogs.com/52XF/p/yunsuanfuchognzai.html>

### c#重载和多态性

https://www.cnblogs.com/zhangkai2237/archive/2012/12/20/2826734.html

### C#子类继承父类

（1）实例化子类MyClass时，先调用了父类的构造函数，然后再调用子类的构造函数。这就是“子类默认调用父类的构造函数”。

（2）“子类不继承父类的构造函数”的含义是：子类的构造函数与父类的构造函数之间没有“重写”和“覆盖”关系；只有“调用”关系，也就是实例化子类时先调用父类的构造函数，再调用子类的构造函数，如图所示以及如（1）所述。

sealed 修饰符表示密封,即不能再被重写。

用于类时，表示该类不能再被继承，不能和 abstract 同时使用，因为这两个修饰符在含义上互相排斥

用于方法和属性时，表示该方法或属性不能再被重写，必须和 override 关键字一起使用，因为使用 sealed 修饰符的方法或属性肯定是基类中相应的虚成员

通常用于实现第三方类库时不想被客户端继承，或用于没有必要再继承的类以防止滥用继承造成层次结构体系混乱

恰当的利用 sealed 修饰符也可以提高一定的运行效率，因为不用考虑继承类会重写该成员

## GUID

“GUID 是一个 128 位整数（16 字节），可用于所有需要唯一标识符的计算机和网络。此标识符重复的可能性非常小。”

GUID 的格式为“xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx”

一个16字节，128位的整数限定的变化率就是，GUID 的总数达到了2^128（3.4×10^38）个，那么18字节，32字节的连续状态变化率更是无穷。

所以这种排列性质的遍历是无解的。

* GUID的重复概率非常非常小。

1、如果两个不认识的人，每个人随机写一个GUID，则生成一样的几率非常非常小。除非这两个人都知道一个GUID，背下来并写下来。

两个人在不互相沟通的情况下，产生同一种相同的结果是非常小。

2、一个人持续写GUID，几亿年才会有写完所有的。

## 循环迭代递归遍历

　loop、iterate、traversal和recursion这几个词是计算机技术书中经常会出现的几个词汇。众所周知，这几个词分别翻译 为：循环、迭代、遍历和递归。乍一看，这几个词好像都与重复（repeat）有关，但有的又好像不完全是重复的意思。那么这几个词到底各是什么含义，有什么区别和联系呢？下面就试着解释一下。

　　1，循环（loop），指的是在满足条件的情况下，重复执行同一段代码。比如，while语句。 循环则技能对应集合，列表，数组等，也能对执行代码进行操作。

　　2，迭代（iterate），指的是按照某种顺序逐个访问列表中的每一项。比如，for语句。 　　迭代只能对应集合，列表，数组等。不能对执行代码进行迭代。

　　3，遍历（traversal），指的是按照一定的规则访问树形结构中的每个节点，而且每个节点都只访问一次。 遍历同迭代一样，也不能对执行代码进行遍历。

4，递归（recursion），指的是一个函数不断调用自身的行为。比如，以编程方式输出著名的斐波纳契数列(http://www.cnblogs.com/feichengwulai/articles/3587848.html中第1题就有用递归方法输出斐波那契数列解题。20-23,41题都用递归，41题进行了详细的解释！！！)。

递归分类：线性递归和尾递归。

## Datatable数据操作

datatable基本操作，统计查询

C# DataTable Compute方法的使用

在开发中需要对DataTable的数据进行处理，比如累加，求最大最小及平均值等，以前都采用手工对DataTable进行循环并计算的方式，现在发现DataTable的Compute方法可以轻松实现这些功能。

https://msdn.microsoft.com/zh-cn/library/system.data.datatable.compute.aspx

语法：

DataTable.Compute("expression","filter");

其中：Expression是要计算的表达式，Filter是要限制在表达式中进行计算的行的筛选器，均是string类型。

Expression：

sum(列名)，avg(列名)，sum(列名)，min(列名)，max(列名)，count(列名)

例：

DataTable有列"Column1"和"Column2”

DataTable.Compute("Sum(Column1)","") //对Column1进行求和

DataTable.Compute("Sum(Column1)","Column2=5") //对Column2的值为5的Column1进行求和

## 反射应用

<http://www.jb51.net/article/49034.htm>

### [详解C#中的反射](http://blog.csdn.net/educast/article/details/2894892)

反射的用途：

（1）使用Assembly定义和加载程序集，加载在程序集清单中列出模块，以及从此程序集中查找类型并创建该类型的实例。

（2）使用Module了解包含模块的程序集以及模块中的类等，还可以获取在模块上定义的所有全局方法或其他特定的非全局方法。

（3）使用ConstructorInfo了解构造函数的名称、参数、访问修饰符（如pulic 或private）和实现详细信息（如abstract或virtual）等。

（4）使用MethodInfo了解方法的名称、返回类型、参数、访问修饰符（如pulic 或private）和实现详细信息（如abstract或virtual）等。

（5）使用FiedInfo了解字段的名称、访问修饰符（如public或private）和实现详细信息（如static）等，并获取或设置字段值。

（6）使用EventInfo了解事件的名称、事件处理程序数据类型、自定义属性、声明类型和反射类型等，添加或移除事件处理程序。

（7）使用PropertyInfo了解属性的名称、数据类型、声明类型、反射类型和只读或可写状态等，获取或设置属性值。

（8）使用ParameterInfo了解参数的名称、数据类型、是输入参数还是输出参数，以及参数在方法签名中的位置等。

反射用到的命名空间：

System.Reflection

System.Type

System.Reflection.Assembly

反射用到的主要类：

System.Type 类－－通过这个类可以访问任何给定数据类型的信息。

System.Reflection.Assembly类－－它可以用于访问给定程序集的信息，或者把这个程序集加载到程序中。

System.Type类：

System.Type 类对于反射起着核心的作用。但它是一个抽象的基类，Type有与每种数据类型对应的派生类，我们使用这个派生类的对象的方法、字段、属性来查找有关该类型的所有信息。

获取给定类型的Type引用有3种常用方式：

●使用 C# typeof 运算符。

Type t = typeof(string);

●使用对象GetType()方法。

string s = "grayworm";

Type t = s.GetType();

●还可以调用Type类的静态方法GetType()。

Type t = Type.GetType("System.String");

上面这三类代码都是获取string类型的Type，在取出string类型的Type引用t后，我们就可以通过t来探测string类型的结构了。

string n = "grayworm";

Type t = n.GetType();

foreach (MemberInfo mi in t.GetMembers())

{

Console.WriteLine("{0}/t{1}",mi.MemberType,mi.Name);

}

Type类的属性：

Name 数据类型名

FullName 数据类型的完全限定名(包括命名空间名)

Namespace 定义数据类型的命名空间名

IsAbstract 指示该类型是否是抽象类型

IsArray 指示该类型是否是数组

IsClass 指示该类型是否是类

IsEnum 指示该类型是否是枚举

IsInterface 指示该类型是否是接口

IsPublic 指示该类型是否是公有的

IsSealed 指示该类型是否是密封类

IsValueType 指示该类型是否是值类型

Type类的方法：

GetConstructor(), GetConstructors()：返回ConstructorInfo类型，用于取得该类的构造函数的信息

GetEvent(), GetEvents()：返回EventInfo类型，用于取得该类的事件的信息

GetField(), GetFields()：返回FieldInfo类型，用于取得该类的字段（成员变量）的信息

GetInterface(), GetInterfaces()：返回InterfaceInfo类型，用于取得该类实现的接口的信息

GetMember(), GetMembers()：返回MemberInfo类型，用于取得该类的所有成员的信息

GetMethod(), GetMethods()：返回MethodInfo类型，用于取得该类的方法的信息

GetProperty(), GetProperties()：返回PropertyInfo类型，用于取得该类的属性的信息

可以调用这些成员，其方式是调用Type的InvokeMember()方法，或者调用MethodInfo, PropertyInfo和其他类的Invoke()方法。

查看类中的构造方法：

NewClassw nc = new NewClassw();

Type t = nc.GetType();

ConstructorInfo[] ci = t.GetConstructors(); //获取类的所有构造函数

foreach (ConstructorInfo c in ci) //遍历每一个构造函数

{

ParameterInfo[] ps = c.GetParameters(); //取出每个构造函数的所有参数

foreach (ParameterInfo pi in ps) //遍历并打印所该构造函数的所有参数

{

Console.Write(pi.ParameterType.ToString()+" "+pi.Name+",");

}

Console.WriteLine();

}

用构造函数动态生成对象：

Type t = typeof(NewClassw);

Type[] pt = new Type[2];

pt[0] = typeof(string);

pt[1] = typeof(string);

//根据参数类型获取构造函数

ConstructorInfo ci = t.GetConstructor(pt);

//构造Object数组，作为构造函数的输入参数

object[] obj = new object[2]{"grayworm","hi.baidu.com/grayworm"};

//调用构造函数生成对象

object o = ci.Invoke(obj);

//调用生成的对象的方法测试是否对象生成成功

//((NewClassw)o).show();

用Activator生成对象：

Type t = typeof(NewClassw);

//构造函数的参数

object[] obj = new object[2] { "grayworm", "hi.baidu.com/grayworm" };

//用Activator的CreateInstance静态方法，生成新对象

object o = Activator.CreateInstance(t,"grayworm","hi.baidu.com/grayworm");

//((NewClassw)o).show();

查看类中的属性：

NewClassw nc = new NewClassw();

Type t = nc.GetType();

PropertyInfo[] pis = t.GetProperties();

foreach(PropertyInfo pi in pis)

{

Console.WriteLine(pi.Name);

}

查看类中的public方法：

NewClassw nc = new NewClassw();

Type t = nc.GetType();

MethodInfo[] mis = t.GetMethods();

foreach (MethodInfo mi in mis)

{

Console.WriteLine(mi.ReturnType+" "+mi.Name);

}

查看类中的public字段

NewClassw nc = new NewClassw();

Type t = nc.GetType();

FieldInfo[] fis = t.GetFields();

foreach (FieldInfo fi in fis)

{

Console.WriteLine(fi.Name);

} (http://hi.baidu.com/grayworm)

用反射生成对象，并调用属性、方法和字段进行操作

NewClassw nc = new NewClassw();

Type t = nc.GetType();

object obj = Activator.CreateInstance(t);

//取得ID字段

FieldInfo fi = t.GetField("ID");

//给ID字段赋值

fi.SetValue(obj, "k001");

//取得MyName属性

PropertyInfo pi1 = t.GetProperty("MyName");

//给MyName属性赋值

pi1.SetValue(obj, "grayworm", null);

PropertyInfo pi2 = t.GetProperty("MyInfo");

pi2.SetValue(obj, "hi.baidu.com/grayworm", null);

//取得show方法

MethodInfo mi = t.GetMethod("show");

//调用show方法

mi.Invoke(obj, null);

System.Reflection.Assembly类

Assembly类可以获得程序集的信息，也可以动态的加载程序集，以及在程序集中查找类型信息，并创建该类型的实例。

使用Assembly类可以降低程序集之间的耦合，有利于软件结构的合理化。

通过程序集名称返回Assembly对象

Assembly ass = Assembly.Load("ClassLibrary831");

通过DLL文件名称返回Assembly对象

Assembly ass = Assembly.LoadFrom("ClassLibrary831.dll");

通过Assembly获取程序集中类

Type t = ass.GetType("ClassLibrary831.NewClass"); //参数必须是类的全名

通过Assembly获取程序集中所有的类

Type[] t = ass.GetTypes();

//通过程序集的名称反射

Assembly ass = Assembly.Load("ClassLibrary831");

Type t = ass.GetType("ClassLibrary831.NewClass");

object o = Activator.CreateInstance(t, "grayworm", "http://hi.baidu.com/grayworm");

MethodInfo mi = t.GetMethod("show");

mi.Invoke(o, null);

//通过DLL文件全名反射其中的所有类型

Assembly assembly = Assembly.LoadFrom("xxx.dll的路径");

Type[] aa = a.GetTypes();

foreach(Type t in aa)

{

if(t.FullName == "a.b.c")

{

object o = Activator.CreateInstance(t);

}

}

### [程序集基础知识](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427505.html)

程序集是 .NET Framework 应用程序的构造块；程序集构成了部署、版本控制、重复使用、激活范围控制和安全权限的基本单元。最终由CLR管理这些程序集中代码的执行。这意味着必须在目标机器上安装好 .NET Framework 。

公共语言运行时（Common Language Runtime，CLR）是一个可由多种编程语言使用的“运行时”，CLR的核心功能（比如内存管理、程序集加载、安全性、异常处理和线程同步）可由面向CLR的所有语言使用。

高级语言通常只公开了CLR的所有功能的一个子集。然而，IL汇编器语言允许开发人员访问CLR的所有功能。CLR允许在不同编程语言之间方便切换，同时又保持紧密集成，所以当你使用的高级语言隐藏了你需要的CLR功能，那么可以使用“混合语言编程”或使用IL汇编语言实现。

1.         程序集执行以下功能：

a)         包含公共语言运行时执行的代码。如果可迁移可执行 (PE) 文件没有相关联的程序集清单，则将不执行该文件中的 Microsoft 中间语言 (MSIL) 代码。

b)         程序集形成安全边界。程序集就是在其中请求和授予权限的单元。

c)         程序集形成类型边界。每一类型的标识均包括该类型所驻留的程序集的名称。在一个程序集范围内加载的 MyType 类型不同于在其他程序集范围内加载的 MyType 类型。

d)         程序集形成引用范围边界。程序集的清单包含用于解析类型和满足资源请求的程序集元数据。它指定在该程序集之外公开的类型和资源。该清单还枚举它所依赖的其他程序集。

e)         程序集形成版本边界。程序集是公共语言运行时中最小的可版本化单元，同一程序集中的所有类型和资源均会被版本化为一个单元。程序集的清单描述您为任何依赖项程序集所指定的版本依赖性。

f)          程序集形成部署单元。当一个应用程序启动时，只有该应用程序最初调用的程序集必须存在。其他程序集（例如本地化资源和包含实用工具类的程序集）可以按需检索。这就使应用程序在第一次下载时保持精简。

g)         程序集提供允许同时运行多个版本的软件组件（称作并行执行）的基本结构。

h)         程序集可以是静态的或动态的。静态程序集可以包括 .NET Framework 类型（接口和类），以及该程序集的资源（位图、JPEG 文件、资源文件等）。静态程序集存储在磁盘上的可迁移可执行 (PE) 文件中。您还可以使用 .NET Framework 来创建动态程序集，动态程序集直接从内存运行并且在执行前不存储到磁盘上。您可以在执行动态程序集后将它们保存在磁盘上。

2.         程序集的优点

程序集旨在简化应用程序部署并解决在基于组件的应用程序中可能出现的版本控制问题。

最终用户和开发人员比较熟悉当今基于组件的系统所产生的版本控制和部署问题。一些最终用户曾经历过在计算机上安装新应用程序失败的事情，发现已有应用程序突然停止工作。许多开发人员花费了大量的时间来使所有必需的注册表项保持一致，以便激活 COM 类。

通过在 .NET Framework 中使用程序集，许多开发问题得以解决。因为程序集是不依赖于注册表项的自述组件，所以程序集使无相互影响的应用程序安装成为可能。程序集还使应用程序的卸载和复制得以简化。

3.         强名称的程序集

强名称是由程序集的标识加上公钥和数字签名组成的。其中，程序集的标识包括简单文本名称、版本号和区域性信息（如果提供的话）。

强名称是使用相应的私钥，通过程序集文件（包含程序集清单的文件，并因而也包含构成该程序集的所有文件的名称和散列）生成的。

1)         强名称程序集显示名称的语法：

<程序集名称>, <版本号>, <区域性>, <公钥标记>

( 如果没有区域性值，请使用 Culture=neutral )

2)         程序集版本号：

<主版本>.<次版本>.<内部版本号>.<修订号>

eg：2.5.719.2

a)         前两个编号：构成了公众对一个版本的理解。如上例为程序集的2.5版本

b)         第三个编号：是程序集的build号。如果公司每天都要生成程序集，那么每天都应该递增这个build号

c)         最后一个编号：指出当前build的修订次数。如因为重大bug当天生成了两次程序集，那么revision号就应该递增。

3)         强名称提供如下好处：

a)         强名称依赖于唯一的密钥对来确保名称的唯一性。任何人都不会生成与您生成的相同的程序集名称，因为用一个私钥生成的程序集的名称与用其他私钥生成的程序集的名称不相同。

b)         强名称保护程序集的版本沿袭。强名称可以确保没有人能够生成您的程序集的后续版本。用户可以确信，他们所加载的程序集的版本出自创建该版本（应用程序是用该版本生成的）的同一个发行者。

c)         强名称提供可靠的完整性检查。通过 .NET Framework 安全检查后，即可确信程序集的内容在生成后未被更改过。注意：强名称中或强名称本身并不暗含信任级别，例如由数字签名和支持证书提供的信任。

d)         在.NET中，只有强名称签名的程序集才能放到全局程序集缓存中。

在引用具有强名称的程序集时，您应该能够从中受益，例如版本控制和命名保护。如果此具有强名称的程序集以后引用了具有简单名称的程序集（后者没有这些好处），则您将失去使用具有强名称的程序集所带来的好处，并依旧会产生 DLL 冲突。因此，具有强名称的程序集“应当只”引用其他具有强名称的程序集。

更多关于强名称签名请参见 [《(2)强名称程序集与数字证书》](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427531.html)

4.         延迟为程序集签名（部分签名）

一个单位可以具有开发人员在日常使用中无法访问的严密保护的密钥对。公钥通常是可用的，但对私钥的访问权仅限于少数个人。开发强名称程序集时，每个引用具有强名称的目标程序集的程序集中都包含了用于为目标程序集指定强名称的公钥的标记。这要求公钥在开发过程中可用。

您可以在生成时使用延迟签名（部分签名），在可迁移可执行 (PE) 文件中为强名称签名保留空间，将实际签名延迟至后面某些阶段（通常就在传送程序集之前）。

下面的步骤说明了延时对程序集签名的过程：

使用 Windows 软件开发包 (SDK) 提供的 [强名称工具](http://www.cnblogs.com/heyuquan/archive/2012/04/01/2428005.html) (Sn.exe)  从将执行最终签名的单位获取密钥对的公钥部分。此密钥通常是 .snk 文件的形式。

使用 System.Reflection 中的两种自定义特性来批注程序集的源代码：

1)         AssemblyKeyFileAttribute，它将包含公钥的文件的名称作为参数传递给其构造函数。

2)         AssemblyDelaySignAttribute，它通过将 true 作为参数传递给其构造函数，表明正在使用延迟签名。例如：

a)         [assembly:AssemblyKeyFileAttribute("myKey.snk")]

b)         [assembly:AssemblyDelaySignAttribute(true)]

3)         编译器将公钥插入程序集清单，并在 PE 文件中为完整的强名称签名保留空间。真正的公钥必须在生成程序集时存储，以便引用此程序集的其他程序集可获取密钥以存储在它们自已的程序集引用中。

4)         由于程序集没有有效的强名称签名，所以必须关闭该签名的验证。通过将“强名称”工具与–Vr 选项一起使用来执行此操作。

下面的示例关闭名为 myAssembly.dll 的程序集的验证。

sn –Vr myAssembly.dll

-Vr选项会将程序集的身份添加到以下注册表：

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\StrongName\Verification

为了确保这些密钥的安全性，密钥值绝对不能持久存储在一个磁盘文件中。“加密服务提供程序”（Cryptographic Service Provder，CSP）提供了对这些密钥的位置进行抽象的容器。以Microsoft使用的CSP为例，一旦访问它提供的容器，就会自动从一个硬件设备中获取私钥。

5)         以后，通常是在即将交付前，重新开启强名称签名验证。通过将“强名称”工具与–R 选项来实际进行强名称签名 (这步不能省，否则会产生安全漏洞) 。

下面的示例使用 sgKey.snk 密钥对为名为 myAssembly.dll 的程序集签署强名称。

sn -R myAssembly.dll sgKey.snk

5.         附属程序集

标记了具体的语言文化的程序集称为附属程序集。附属程序集通常只包含语言文化特有的资源，不包含任何代码，为附属程序集指定的语言文化应准确反映程序集中包含的资源的语言文化。

部署一个附属程序集时，应该把它存到一个专门的子目录中，子目录的名称应该与语言文化的文本相匹配。

在运行时，可以用System.Resources.ResourceManager类来访问一个附属程序集的资源。

通常使用AL.exe工具来生成附属程序集。之所以不用编译器，是因为附属程序集中不应包含任何代码。

一般不应生成引用了附属程序集的一个程序集。换言之，程序集的AssemblyRef记录项只应应用语言文化中性的程序集。想访问包含在一个附属程序集中的类型或成员，应使用“反射技术”。

6.         全局程序集缓存

安装有公共语言运行时的每台计算机都具有称为全局程序集缓存的计算机范围内的代码缓存。全局程序集缓存中存储了专门指定给由计算机中若干应用程序共享的程序集。

1)         一个程序集可以采用两种方式来部署：私有和全局。

a)         “私有部署的程序集”是指部署到应用程序基目录或者一个子目录中的程序集。弱命名程序集只能以私有方式部署。

b)         “全局部署的程序集”是指部署到一些已知位置，即全局程序集缓存(Global Assembly Cache，GAC)；CLR在查找程序集时，会检查这些位置。强名称程序集既可以私有部署，也可以全局部署。

应当仅在需要时才将程序集安装到全局程序集缓存中以进行共享。一般原则是：程序集依赖项保持专用，并在应用程序目录中定位程序集，除非明确要求共享程序集。另外，不必为了使 COM 互操作或非托管代码可以访问程序集而将程序集安装到全局程序集缓存。

因为将程序集安装到GAC中，会破坏我们的一些基本目标，即：简单地安装、备份、还原、移动和卸载应用程序。所以，建议程序员尽量避免全局部署，尽量使用私有部署。

2)         要将程序集安装到全局程序集缓存中的原因有以下几点：

a)         共享位置。

可将应用程序公共使用的程序集放在全局程序集缓存中。

b)         文件安全性。

管理员通常使用访问控制列表 (ACL) 来保护 systemroot 目录，以控制写入和执行访问。因为全局程序集缓存安装在 systemroot 目录中，它继承了该目录的 ACL。建议只允许具有“管理员”权限的用户从全局程序集缓存中删除文件。

c)         并行版本控制。

可在全局程序集缓存中维护程序集的多个副本（名称相同但版本信息不同）。

d)         其他搜索位置。

在探测或使用配置文件中的基本代码信息之前，公共语言运行时会先检查全局程序集缓存中符合程序集请求的程序集。

3)         可采用三种方法将程序集安装到全局程序集缓存中：

a)         使用全局程序集缓存工具 (Gacutil.exe)。

您可以使用 Gacutil.exe 将强名称程序集添加到全局程序集缓存，并查看全局程序集缓存的内容。

在命令提示符处，键入下列命令将强名称程序集安装到全局程序集缓存中：

gacutil –I <assembly name>

注意：Gacutil.exe 只用于开发，不应用于将产品程序集安装到全局程序集缓存中。

b)         使用 Microsoft Windows Installer 。

这是将程序集添加到全局程序集缓存的最常用方法，建议采用。此安装程序可提供全局程序集缓存中程序集的引用计数，还具有其他优点。

c)         使用 Mscorcfg.msc（.NET Framework 配置工具） （.NET 4.0 已删）

Microsoft 管理控制台 (MMC) 单元，可以查看全局程序集缓存并将新的程序集添加到该缓存。

在全局程序集缓存中部署的程序集必须具有强名称。将一个程序集添加到全局程序集缓存时，必须对构成该程序集的所有文件执行完整性检查。缓存执行这些完整性检查以确保程序集未被篡改。

4)         使自己的程序集出现在“.NET”选项卡的列表中

在vs中引用GAC程序集时，通过在 “.NET” 选项卡列表中选择；对于自己的程序集目录若需要他出现在“.NET”选项卡列表中，可在注册表中配置：

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\.NETFramework\AssemblyFolders

(或HKEY\_LOCAL\_MACHINE)

7.         程序集启动并运行

托管模块是一个标准的32位Microsoft Windows可移植执行体(PE32)文件（Portable Executable），或者是一个标准的64位Windows可移植执行体(PE32+)文件，它们都需要CLR才能执行。

1)         一个托管PE文件由4个部分构成：PE32(+)头、CLR头、元数据以及IL。

元数据是一个二进制数据块，由几个表构成。分为三个类别：定义表、引用表和清单表。

通过工具：IL反汇编器可查看元素据。“视图|元数据|显示”（或Ctrl+M）

a)         定义表：ModuleDef、TypeDef、MethodDef、FieldDef、ParamDef、PropertyDef、EventDef。

b)         引用表：AssemblyRef、ModuleRef、TypeRef、MemberRef。

c)         清单也是一组元素据表的集合（AssemblyDef、FileDef、ManifestResourceDef、ExportedTypesDef），表中主要包含了作为程序集的组成部分的那些文件的名称，此外，它们还描述了程序集的版本、语言文化、发布者、公开导出的类型以及构成程序集的所有文件。

CLR总是首先加载包含“清单”元素据表的文件，再根据这个“清单”来获取程序集中的其他文件的名称。

2)         CLR托管代码相对于非托管代码的一个优势：

Windows进程需要使用大量操作系统资源，所以进程数量太多，会损害性能并制约可用的资源。

CLR提供了在一个操作系统进程中执行多个托管应用程序的能力。每个托管的应用程序都在一个AppDomain中执行。默认情况下，每个托管的EXE文件都在它自己的独立空间中运行，这个地址空间只有一个AppDomain。然而，CLR的宿主进程(比如IIS或者Microsoft SQL Server)可决定在单个操作系统进程中运行多个AppDomain。

3)         /platform选项：Any CPU、x86、x64、intel Itanium

-------/platform开关选项对生成的模块的影响以及在运行时的影响

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| /platform开关 | 生成托管托管模块 | X86 Windows | X64 Windows | IA64 Windows |
| anycpu(默认) | PE32/不明确指定 | 作为32位应用程序运行 | 作为64位应用程序运行 | 作为64位应用程序运行 |
| x86 | PE32/x86 | 作为32位应用程序运行 | 作为WoW64位应用程序运行 | 作为WoW64位应用程序运行 |
| x64 | PE32+/x64 | 不运行 | 作为64位应用程序运行 | 不运行 |
| Itanium  （Intel安腾处理器） | PE32+/Itanium | 不运行 | 不运行 | 作为64位应用程序运行 |

Windows的【64位版本】提供一种名为WoW64（Windows on Windowss64）的技术，允许运行32位Windows应用程序。该技术甚至允许使用x86本机代码的32位应用程序在Itanium机器上运行。这是因为WoW64技术能模拟x86指令集，虽然这样做会显著影响性能。

Windows检查好EXE文件头，决定是创建32位、64位还是WoW64进程之后，会在进程的地址空间中加载MSCorEE.dll的x86，x64或IA64版本。然后，进程的主线程调用MSCorEE.dll中定义的一个方法。这个方法初始化CLR，加载EXE程序集，然后调用其入口方法(Main)。

更多C#编译器选项请参加： [《csc.exe(C# 编译器)》](http://www.cnblogs.com/heyuquan/archive/2012/04/01/2427957.html)

4)         方法调用过程：

为了执行一个方法，首先必须把它的IL转换成本地CPU指令，由CLR中的JITCompiler函数负责。由于IL是“即时“(just in time)编译的，所以通常CLR的这个组件称为JITter或JIT编译器。

JIT会在定义程序集的MethodDef元数据表中查找被调用的方法的IL。接着，JIT验证IL代码，并将IL代码编译成本地CPU指令并【存储到动态内存】中，再次调用的方法以本地代码的形式全速运行。

一旦应用程序终止，编译好的代码也会被丢弃。所以将来再次运行应用程序，或者同时启动应用程序的两个实例（实例是使用不同的操作系统进程—eg：x86和x64），JIT编译器必须再次将IL编译为本地指令。

### [强名称程序集与数字证书](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427531.html)

防止程序集被篡改，可以用两种不同但相互补充的方式对程序集进行签名：使用强名称或使用[SignTool.exe（签名工具）](http://www.cnblogs.com/heyuquan/archive/2012/04/01/2427974.html)

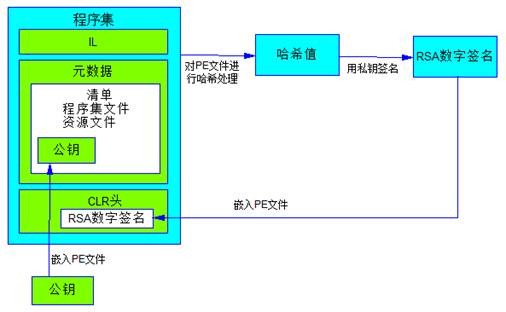
可以将强名称的数字签名和使用SignTool.exe（签名工具）的证书签名一起提供给程序集，或者您可以单独使用其中之一。这两个签名工具一次只能对一个文件进行签名，对于多文件程序集，您可以对包含程序集清单的文件进行签名 (CLR总是首先加载包含“清单”元素据表的文件，再根据这个“清单”来获取程序集中的其他文件的名称) 。

强名称签名的密钥【不必与】签名工具的微软代码签名证书密钥相同。

一、强名称程序集

1.         强名称程序集放篡改原理

强名称程序集图示如下：

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203311919172893.jpg)

检查：加载程序集的检查，系统对包含清单的那个文件的内容进行哈希处理，并将哈希值与PE文件中嵌入的RSA数字签名进行比较。系统还会对程序集的其他文件的内容进行哈希处理，并将哈希值与清单文件的FileDef表中存储的哈希值进行比较。

将一个强命名的程序集安装到GAC时，系统会执行一次检查，核实含有清单的那个文件没有被篡改。这个检查只在安装时执行一次。除此之外，为了增强性能，如果程序集被完全信任，并加载到一个完全信任的AppDomain，CLR不检查强命名的程序集是否被篡改。相反，从非GAC的一个目录加载强命名程序集时，CLR会校验程序集的清单文件，核实文件的内容未被篡改，造成该文件每次加载都会带来额外的性能开销。

2.         实践强名称签名验证

通过使用 Microsoft SDK 工具：ildasm.exe（IL反汇编工具）改变程序集内容。

a)         将程序集用 ildasm.exe 工具打开，执行“文件---转储”保存为 .il 文件(同时还会保存一个.res文件)，用文本工具打开il文件改变文件内容(如多加一个程序集引用)。

b)         执行“运行---cmd”打开命令行程序，输入：ilasm /resource=\*.res \*.IL /dll

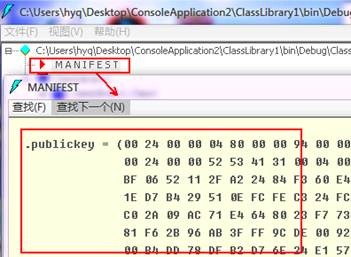
此时强名称程序集本身内容已被更改，在CLR执行程序集加载进行强名称签名检查时报如下错误（若为简单程序集不会检查报错）：

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203311919258131.jpg)

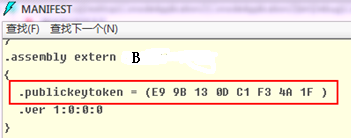
3.         强名称程序集是否安全？

知道强名称签名验证后，我们来看看强名称程序集到底附加了什么信息，比如：强名称程序集A 引用强名称程序集B

-------强名称程序集B本身多出 .publickey 标识

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203311919311374.jpg)

-------强名称程序集A对B的引用处多出 .publickeytoken 标识

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203311919401562.png)

    通过Microsoft SDK 工具：ildasm.exe（IL反汇编工具），我们已经知道可以对程序集内容进行篡改，并且篡改的内容会因为强名称验证而报错。但是：

a)         如果把如上强名称程序集A的 .publickey 内容删除，那么A   
就不再是强名称程序集了，即在加载程序集A时不会进行强名称程序集A的篡改检查。那么这时候可以对A进行任意更改。

b)         A引用B时因为有.publickeytoken强名称标识所以要求B是强名称且生成和A中记录的相同公钥标识。如果仅仅删除A的.publickeytoken那么会报如下错误：

[1111](http://images.cnblogs.com/cnblogs_com/heyuquan/201204/201204020117098114.jpg)

因为找不到指定的强名称标识程序集（ [《(5)CLR 运行时探测程序集引用的步骤》](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427557.html) ），所以我们必须将B的 .publickey内容一并删除。这样可以实现被引用的程序集任意更改。

为了解决这个问题，所以我们必须同时使用 SignTool.exe（签名工具）对程序集进行证书签名。

二、SignTool.exe（签名工具）签名

签名工具要求发行者向第三方证书颁发机构证实其标识并获取证书。然后此证书将嵌入到您的文件中，并且管理员能够使用该证书来决定是否相信这些代码的真实性。

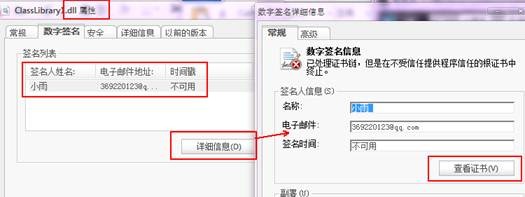
执行如下命令

生成证书 makecert -r -pe -sv MyCert.pvk MyCert.cer

生成.pfx pvk2pfx.exe -pvk MyCert.pvk -spc MyCert.cer -pfx MyCert.pfx

证书签名 signtool sign /f MyCert.pfx D:\ClassLibrary1.dll

这样，在查看被签名的程序集属性时会有“数字签名”页签，

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203311919469822.jpg)

通过查看证书按钮可查看证书具体信息，确保所使用的文件是由指定产家提供，关于证书如何验证请参见 [《数字签名|数字证书》](http://www.cnblogs.com/heyuquan/tag/%E6%95%B0%E5%AD%97%E7%AD%BE%E5%90%8D%7C%E6%95%B0%E5%AD%97%E8%AF%81%E4%B9%A6/)

那么，既然有证书验证为什么还是需要强名称程序集？因为强名称程序集提供了诸如版本控制、能安装在GAC中等简单程序集不能提供的优点

### [程序集加载 Assembly类](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427535.html)

程序集是 .NET Framework 应用程序的构造块；程序集构成了部署、版本控制、重复使用、激活范围控制和安全权限的基本单元。

属性：

CodeBase，EntryPoint，EscapeCodeBase，Evidence，FullName，GlobalAssemblyCache，HostContext，ImageRuntimeVersion，IsDynamic，IsFullTrusted，Location，ManifestModule，PermissionSet，ReflectionOnly，SecurityRuleSet

事件：

http://pic002.cnblogs.com/images/2012/106337/2012040202062066.jpg

实例方法：

CreateInstance()，GetCustomAttributesData()，GetExportedTypes()，GetFile()，GetFiles()，GetLoadedModules()，GetManifestResourceInfo()，GetManifestResourceNames()，GetManifestResourceStream()，GetModule()，GetModules()，GetName()，GetReferenceAssemblies()，GetSatelliteAssembly()，GetType()，GetTypes()，LoadModule()

接口方法

a)         ICustomAttributeProvider：

GeCustomAttributes()，IsDefined()

b)         ISerializable：

http://pic002.cnblogs.com/images/2012/106337/2012040202064568.jpg

静态方法：

CreateQualifiedName()，GetAssembly()，GetCallingAssembly()，GetEntryAssembly()，GeExecutingAssembly()，Load()，LoadFile()，LoadFrom()，LoadModule()，LoadWithPartialName()，ReflectionOnlyLoad()，ReflectionOnlyLoadFrom()，UnsafeLoadFrom()

1.         Assembly.ModuleResolve 事件

公共语言运行时类加载程序不能通过正常方法解析对程序集的内部模块的引用时发生。如使用 LoadFile(Path)、只反射加载等加载程序集，不会加载依赖程序集。

2.         CodeBase 、EscapedCodeBase 和 Location

1)         CodeBase

获取最初指定的程序集的位置，例如，在 AssemblyName 对象中指定的位置。

如果程序集是使用Load(byte[]) 重载作为字节数组加载的，则此属性返回该方法的调用方的位置，而不是已加载程序集的位置。

2)         EscapedCodeBase

获取 URI，包括表示基本代码的转义符。

3)         Location

获取包含清单的已加载文件的路径或 UNC 位置。（绝对路径）

4)         术语：UNC (Universal Naming Convention) 统一命名约定

对于网络服务器上的目标文件，可使用“通用命名约定 (UNC)”地址（UNC：“统一命名约定”地址，用于确定保存在网络服务器上的文件位置）。这些地址以“file:\\”开始并提供服务器名、共享名和文件的完整路径。例如，“file:\\server\share\path\project file.mpp”是绝对 UNC 地址。

3.         IsDynamic

获取一个值，该值指示当前程序集是否是通过使用反射发出在当前进程中动态生成的。动态程序集由派生类 System.Reflection.Emit.AssemblyBuilder 表示；将动态程序集保存到磁盘后，该程序集就不再是动态的。

4.         CreateInstance()

从此程序集中查找指定的类型，并使用系统激活器 ( Activator.CreateInstance ) 创建它的实例。

可传递要搜索的类型的 Type.FullName 、搜索是否区分大小写、搜索成员标识BindingFlags、System.Reflection.Binder对象、类型构造函数的参数数组等参数。

5.         CreateQualifiedName()

创建由类型的程序集的显示名称限定的类型的名称。返回的字符串的格式为： <完整类型名称>，<程序集显示名称> 即： return (typeName + ", " + assemblyName)

6.         GetAssembly()，GetCallingAssembly()，GetEntryAssembly()，GetExecutingAssembly()，GetSatelliteAssembly()

1)         GeAssembly(Type type)

获取在其中定义指定类的当前加载的程序集。

2)         GetCallingAssembly()

返回调用当前正在执行的方法的方法的 Assembly。

如果调用 GetCallingAssembly 方法的方法由实时 (JIT) 编译器内联扩展，或如果其调用方将内联扩展，则由 GetCallingAssembly 返回的程序集可能会发生异常。例如，考虑下列方法和程序集：

程序集 A1 中的方法 M1 调用 GetCallingAssembly。

程序集 A2 中的方法 M2 调用 M1。

程序集 A3 中的方法 M3 调用 M2。

当 M1 未内联时，GetCallingAssembly 返回 A2。当 M1 内联时，GetCallingAssembly 返回 A3。

使用MethodImplOptions.NoInlining可以阻止JIT在编译时把某些方法进行内联处理。EG：[MethodImpl(MethodImplOptions.NoInlining)]

3)         GetEntryAssembly()

获取默认应用程序域中的进程可执行文件。在其他的应用程序域中，这是由 AppDomain.ExecuteAssembly 执行的第一个可执行文件。

如果一个非托管应用程序创建了使用 C# 编写的一个 COM 组件的实例，则从该 C# 组件调用 GetEntryAssembly 方法将返回 null，因为该进程的入口点是非托管代码而不是托管程序集。

4)         GetExecutingAssembly()

                  获取包含当前执行的代码的程序集。

5)         GetSatelliteAssembly( CultureInfo culture,Version version )

获取指定区域性的附属程序集的指定版本。

附属程序集包含本地化的资源，这与主应用程序程序集有所不同，主应用程序程序集包含不可本地化的可执行代码和只用于默认或非特定区域性的资源。

调用 GetSatelliteAssembly(CultureInfo) 重载以使用当前的程序集版本。

7.         Load()，LoadFile()，LoadFrom()，LoadWithPartialName()，ReflectionOnlyLoad()，ReflectionOnlyLoadFrom()，UnsafeLoadFrom()

    请参见 [《(7)动态程序集加载Load()》](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427582.html)

8.         提供实例方法获取|加载模块、获取类型

        GetLoadedModules()、GetModules()、GetModule()、LoadModule()、GetTypes()、GetType()

9.         GetExportedTypes()

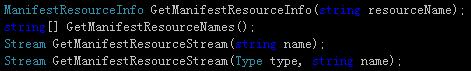
获取此程序集中定义的公共类型，这些公共类型在程序集外可见。只有公共类型以及嵌套在其他公共类型中的公共类型才在程序集外可见。

10.     返回 AssemblyName 对象的方法：GetName()、GetReferencedAssemblies()

11.     GetFile()、GetFiles()

获取此程序集清单的文件表中文件的 System.IO.FileStream。

12.     提供获取程序集清单相关方法



### [绑定程序集配置策略](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427544.html)

[=============C#.Net 篇目录==============](http://www.cnblogs.com/heyuquan/articles/2427494.html)

<runtime> 的 <assemblyBinding> 元素

Xmlns特性，指定程序集绑定所需的 XML 命名空间。使用字符串“urn:schemas-microsoft-com:asm.v1”作为值。

<assemblyBinding>子元素如下：

1.         <dependentAssembly> 元素

封装每个程序集的绑定策略和程序集位置。为每个程序集使用一个 dependentAssembly 元素。

a)         <assemblyIdentity> 元素

b)         <bindingRedirect> 元素 将一个程序集版本重定向到另一个版本。

c)         <codeBase> 元素

确定要使用的程序集版本后，运行时应用确定版本的基本代码设置查找程序集。（可以有多个<codeBase>元素，用于对应不同程序集版本）

计算机配置文件或发布者策略文件中使用 <codeBase> 设置，该文件还必须重定向程序集版本。应用程序配置文件可在不重定向程序集版本的情况下拥有基本代码设置（即可不使用< bindingRedirect >元素）。

如果程序集具有强名称，则基本代码设置可以是本地 Intranet 或 Internet 上的任何地方。如果程序集为私有程序集，则基本代码设置必须是相对于应用程序目录的路径。

注意：在使用<codeBase>，需要提供版本，公钥等长名称程序集标识信息，所以使用部分名称加载程序集不会使用配置策略。

2.         <probing> 元素

指定加载程序集时公共语言运行时要搜索的根位置下用户的子目录列表（必须是应用程序根目录的子目录），多个子目录可用分号隔开。

可以使用应用程序域的 AppendPrivatePath 属性，在应用程序配置文件和托管代码中指定此位置。当在托管代码中指定时，先探测托管代码 privatePath，随后探测应用程序配置文件中指定的路径。

3.         <publisherPolicy> 元素

指定运行时是否采用发行者策略配置信息。

4.         <qualifyAssembly> 元素

配置当动态加载程序集使用程序集的部分名称时应映射到的程序集全名 (名称、版本、区域性，如果适用还有公钥标记)。

要避免对部分程序集名称进行绑定，因为稳定性及兼容性不确定，如下：

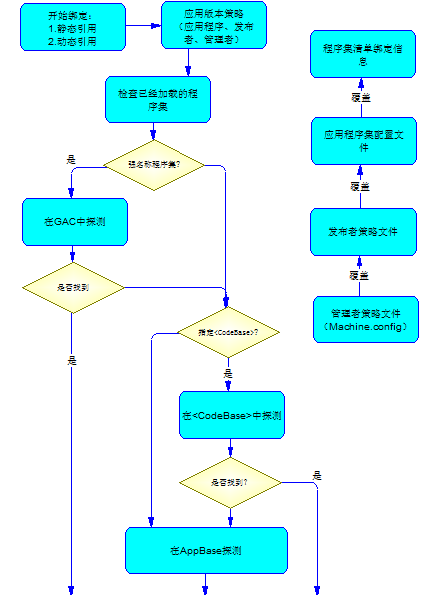
a)         如果没有指定版本，则运行时会尝试用匹配其他指定条件的最高版本号定位程序集。

b)         如果没有指定区域性或公钥标记，则运行时对这两个特性的行为是未定义的。

----------注意：<publisherPolicy>与<codeBase> 应用区别

如果在一个目录中有一个程序集的多个版本，并且要引用该程序集的某个特定版本，则必须使用 <codeBase> 元素而不是 <probing> 元素的 privatePath 特性。如果使用 <probing> 元素，则运行时第一次找到与引用的简单程序集名称匹配的程序集时就会停止探测，而不论这是否是正确的匹配项。如果该程序集是正确的匹配项，则使用它。如果该程序集不是正确的匹配项，则停止探测，绑定失败。

### [CLR 运行时探测程序集引用的步骤](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427557.html)

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/20120331195903848.png)

  开始绑定

当运行时尝试解析对另一个程序集的引用时，就开始进行定位并绑定到程序集的进程。该引用可以是静态的，也可以是动态的。无论引用是对静态程序集的引用还是对动态程序集的引用，运行时均使用相同的解析过程。

a)         静态引用：在生成时，编译器在程序集清单的元数据中记录静态引用。

b)         动态引用：由于调用各种方法而动态构造的，加载方法详细介绍请参见 [《(7)动态程序集加载Load()》](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427582.html) 。

  运行时使用以下步骤来【解析】程序集引用：

第 1 步：检查配置文件，确定版本

可以基于三个 XML 文件在不同的级别下配置程序集绑定行为 ：

a)         应用程序配置文件，可用于启用安全模式。

b)         发行者策略文件，决定升级绑定的新版本。

c)         计算机配置文件，决定绑定的最终版本。

这些文件使用相同的语法，并且给特定程序集提供诸如绑定重定向、代码位置以及绑定模式等信息。

EG：(详细语法请参见 [《(4)绑定程序集配置策略》](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427544.html))

<configuration>

<runtime>

<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">

<dependentAssembly>

<assemblyIdentity name="asm6" publicKeyToken="c0……" />

<bindingRedirect oldVersion="3.0.0.0" newVersion="2.0.0.0"/>

</dependentAssembly>

<dependentAssembly> ……</dependentAssembly>

</assemblyBinding>

</runtime>

</configuration>

1.         应用程序配置文件

第一，在应用程序配置文件中，公共语言运行时检查覆盖调用程序集清单中存储的版本信息的信息。应用程序配置文件可以随应用程序一起部署，但它并不是应用程序执行所必需的。

a)         对于客户端可执行文件，应用程序配置文件与应用程序可执行文件位于同一个目录中，并且两者具有相同的基文件名，但应用程序配置文件的扩展名为 .config。

b)         在基于浏览器的方案中，HTML 文件必须使用 <link> 元素显式指向该配置文件。

2.         发行者策略文件(必须安装在全局程序集缓存)

第二，运行时检查发行者策略文件（如果存在的话）。发行者策略文件是由组件发行者作为共享组件的修正或更新分发的，即针对的是放入全局程序集缓存(GAC)中的程序集。这些文件包含共享组件发行者发布的兼容性信息，这些信息将程序集引用指向新版本，影响使用某个共享组件的所有应用程序。

发行者策略配置文件覆盖来自应用程序的版本信息（即，来自于程序集清单或应用程序配置文件）。

与应用程序配置文件和计算机配置文件不同，发行者策略文件包含在其自己的程序集中，该程序集必须安装在全局程序集缓存中。(al.exe + gacutil.exe)

----------安全模式

安全模式由 <publisherPolicy apply="yes |no"/> 元素决定，该元素仅位于应用程序配置文件中。安全模式指定是否应该从绑定过程中移除发行者策略配置信息。

通常，发行者策略文件是作为 Service Pack 或程序更新的一部分显式安装的。如果升级的共享组件有任何问题，则可以使用安全模式忽略发行者策略文件中的覆盖。（默认为yes）

a)         整个应用程序设置安全模式：设置 <publisherPolicy apply=no/>，并且置于 <assemblyBinding> 元素内不包含依赖程序集元素。

b)         特定的程序集设置安全模式：设置 <publisherPolicy apply=no/> 并放置在 <dependentAssembly> 元素中，使用 <dependentAssembly> 元素指定要影响哪些程序集。

3.         第三，计算机配置文件（管理员策略文件）

运行时检查计算机配置文件Machine.config位于本地计算机上安装【运行时】的根目录的 Config 子目录中。计算机配置文件中的设置优先于所有其他配置设置，确定的版本是最终版本，并且不能被覆盖。 Machine.config 文件中指定的覆盖影响所有的应用程序。

第 2 步：检查已经加载的程序集

在.NET Framework 2.0 以后比如 Assembly.Load 方法有缓存，第一次加载目标程序集的失败或者成功的状态都会被缓存，这样在你下一次加载目标程序集时会直接从缓存里取目标程序集的内容和状态。

当失败后我们可能想要再加载一次或者加载多次直到成功为止，这时需要在app.config 中加入如下配置信息来禁用缓存:

<runtime> <disableCachingBindingFailures enabled="1" /> </runtime>

注意： ReflectionOnlyLoad()\ReflectionOnlyLoadFrom() 只反射加载不会使用缓存。

第 3 步：检查全局程序集缓存

如果被引用的程序集是强命名的（换句话说，AssemblyRef包括了非空的公钥或公钥标记），那么接下来就会在GAC中查找这个程序集。否则，由于弱名称程序集不能安装在GAC中，就会跳过这一步。

第 4 步：通过基本代码或探测定位程序集

在使用调用程序集的引用中的信息和配置文件中的信息确定了正确的程序集【版本】之后，并且在公共语言运行时在全局程序集缓存中进行检查（仅检查具有强名称的程序集）之后，公共语言运行时就会尝试【查找】该程序集。定位程序集的过程包含以下步骤：

1)         如果在配置文件中找到对应的 <codeBase> 元素，则运行时会检查指定的位置。

配置文件的codeBase元素实际标记了一个URL。这个URL可引用用户机器上的任何目录，也可以引用一个Web地址。如果引用的是Web地址，CLR会自动下载文件，并把它存储到用户的下载缓存中（%UserProfile%\Local Settings\Application Data\Assembly下的一个子目录）。将来进行引用时，CLR会将已下载的文件的时间戳与指定URL处的文件的时间戳进行比对。如果URL处的文件具有较新的时间戳，CLR会下载新版本文件并加载它。否则，CLR会加载缓存的文件，不再重复下载（从而增强性能）。

如果找到匹配的程序集，则会使用该程序集，并且不会进行探测。如果CodeBase指定的可执行体与这个引用不匹配，会认为搜索是失败的，不会进行程序集加载。

2)         没有<codeBase>元素则运行时使用指定的规则探测引用的程序集。

  通过探测定位程序集

如果应用程序配置文件中没有 <codeBase> 元素，则运行时使用以下四个条件来探测程序集：

1.         应用程序基，它是指向应用程序位置的URL（即指向应用程序所在的目录）。

a)         对于可执行体而言，它是包括EXE文件的目录。

b)         对于Web应用程序而言，AppBase是由Web服务器定义的应用程序根目录。

2.         区域性，它是被引用的程序集的区域性特性。

3.         名称，它是被引用的程序集的名称。

4.         <probing> 元素的 privatePath 特性，这是根位置下用户定义的子目录列表（必须是应用程序根目录的子目录）。

---------探测应用程序基和区域性目录

运行时始终在应用程序基中开始探测，应用程序基可以是一个 URL，也可以是计算机上的应用程序根目录。如果在应用程序基中没有找到引用的程序集，并且未提供区域性信息，则运行时使用程序集名称搜索任何子目录。探测的目录包括：

a)         [应用程序基] / [程序集名称].dll

b)         [应用程序基] / [程序集名称] / [程序集名称].dll

如果指定了引用的程序集的区域性信息，则只探测以下目录：

a)         [应用程序基] / [区域性] / [程序集名称].dll

b)         [应用程序基] / [区域性] / [程序集名称] / [程序集名称].dll

---------使用 privatePath 特性进行探测

然后，运行时还探测使用 <probing> 元素的 privatePath 特性指定的目录。探测的目录包括：

a)         [应用程序基] / [bin 路径] / [程序集名称].dll

b)         [应用程序基] / [bin 路径] / [程序集名称] / [程序集名称].dll

如果包含区域性，则探测以下目录：

a)         [应用程序基] / [bin 路径] / [区域性] / [程序集名称].dll

b)         [应用程序基] / [bin 路径] / [区域性] / [程序集名称] / [程序集名称].dll

                           <publisherPolicy>与<codeBase> 应用区别请参见： [《(4)绑定程序集配置策略》](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427544.html)

### [程序集加载上下文](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427574.html)

1.         四种程序集加载到上下文及优缺点：

1)         默认加载上下文

加载上下文包含通过探测全局程序集缓存、主机程序集存储区（如果承载运行时）以及应用程序域的 ApplicationBase 和 PrivateBinPath 所找到的程序集。比如Load()使用程序集标识的重载。（探测规则请参见：[《(5)CLR 运行时探测程序集引用的步骤》](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427557.html) ）

使用默认加载上下文具有以下缺点：

a)         加载到其他上下文中的依赖项不可用。

b)         不能将探测路径外部的位置的程序集加载到默认加载上下文中。

2)         加载位置上下文

可以从未位于应用程序路径下（并因此未包含在探测中）的某个路径加载程序集。加载位置上下文允许从该路径查找和加载依赖项，因为路径信息由上下文维护。

此上下文中的程序集可以使用加载到默认加载上下文中的依赖项。

通过使用 Assembly.LoadFrom 方法或按路径加载的其他方法之一来加载程序集具有以下缺点：

a)         如果已加载一个具有相同标识的程序集，则即使指定了不同的路径，LoadFrom 仍返回已加载的程序集。

b)         如果用 LoadFrom 加载一个程序集，随后默认加载上下文中的一个程序集尝试按显示名称加载同一程序集，则加载尝试将失败。对程序集进行反序列化时，可能发生这种情况。

c)         如果用 LoadFrom 加载一个程序集，并且探测路径包括一个具有相同标识但位置不同的程序集，则会发生切换到默认上下文的情况，这样会有损耗性能。

d)         LoadFrom 要求指定路径中包含 FileIOPermissionAccess.Read 和 FileIOPermissionAccess.PathDiscovery 或 WebPermission。

e)         如果存在程序集的本机映像，将不会使用它。

f)          程序集不能以非特定域的方式加载。

3)         只反射上下文

使用 ReflectionOnlyLoad 和 ReflectionOnlyLoadFrom 方法加载的程序集；无法执行这些上下文中的代码。

该上下文通常用于检查无法加载执行的代码，可以

a)         跳过程序集强命名认证。

b)         跳过处理器架构检查规则。

4)         无上下文

在没有上下文的情况下进行加载是将具有同一标识的多个程序集加载到一个应用程序域中的唯一方式。这将省去探测成本。

此上下文中的程序集可以使用加载到默认加载上下文中的依赖项。

a)         使用反射发出生成的瞬态程序集。

b)         从字节数组加载的程序集。（如果加载到的程序集标识（在应用策略后创建的）与全局程序集缓存中的程序集标识匹配，将会从全局程序集缓存加载程序集，这时是加载到“默认加载上下文”中）

c)         LoadFile 方法按照目标程序集的文件路径加载该程序集。

在没有上下文的情况下加载程序集具有以下缺点：

a)         无法将其他程序集绑定到在没有上下文的情况下加载的程序集，除非您处理 AppDomain.AssemblyResolve 事件。

b)         不会自动加载依赖项。您可以在没有上下文的情况下预加载依赖项、将依赖项预加载到默认加载上下文中或通过处理 AppDomain.AssemblyResolve 事件来加载依赖项。

c)         在没有上下文的情况下加载具有同一标识的多个程序集会导致出现类型标识问题。

d)         如果存在程序集的本机映像，将不会使用它。

e)         程序集不能以非特定域的方式加载。

2.         避免将一个程序集加载到多个上下文中

将一个程序集加载到多个上下文中会导致出现类型标识问题。将同一个程序集中的相同类型加载到两个不同的上下文中，就像是加载具有相同名称的两个不同的类型一样。如果您尝试将一个类型强制转换为另一个类型，则将引发 InvalidCastException，并显示一条令人混淆的消息，指示不能将类型 MyType 强制转换为类型 MyType（运行时会将这两个程序集副本中的MyType视为不同的类型）。

另：将一个程序集的多个版本加载到同一种上下文中也会导致出现类型标识问题。

3.         考虑切换到默认加载上下文

检查应用程序的程序集加载和部署模式。是否能够消除从字节数组加载的程序集？是否能够将程序集移动到探测路径中？如果程序集位于全局程序集缓存中或应用程序域的探测路径（即 ApplicationBase 和 PrivateBinPath）中，则可以按照程序集的标识来加载程序集。

如果无法将所有程序集放入探测路径中，请考虑替代方式，例如使用 .NET Framework 外接程序模型、将程序集放置到全局程序集缓存中或创建应用程序域。

### [动态程序集加载Load()](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427582.html)

一、Assembly类提供的多个载入动态程序集方法

1.         Load()，LoadFile()，LoadFrom()，LoadWithPartialName()，ReflectionOnlyLoad()，ReflectionOnlyLoadFrom()，UnsafeLoadFrom()

1)         Load()

可通过程序集的唯一标识符AssemblyName对象或程序集的长|短格式名称字符串或基于通用对象文件格式(COFF，Common Object File Format)的映像byte[] 来加载程序集。

a)         实现在加载目标程序集时不锁定它

byte[] buffer = System.IO.File.ReadAllBytes(ass.Location);

Assembly assembly = Assembly.Load(buffer);

暂时先将目标程序集锁定，然后把程序集内容复制到内存中，读取后将程序集解锁并从内存中加载目标程序集的拷贝。

2)         LoadWithPartialName() 已过时，使用Load()代替。

使用部分名称从应用程序目录或从全局程序集缓存加载程序集。注意：此方法加载不到程序集返回null，不会报错。

3)         LoadFile()

加载指定路径上的程序集文件的内容。

4)         LoadFrom()

可通过给定程序集文件名或路径、哈希值及哈希算法来加载程序集。

5)         ReflectionOnlyLoad()

可用程序集的长|短格式名称字符串或基于通用对象文件格式(COFF)的映像来加载程序集。程序集被加载到调用方的应用程序域的只反射上下文中。

6)         ReflectionOnlyLoadFrom()

将给定路径的程序集加载到只反射上下文中。

7)         UnsafeLoadFrom()

使用包含程序集清单的文件的名称或路径。绕过某些安全检查，将程序集加载到加载源上下文中。

8)         差异

a)         Load(string|AssenblyName)\LoadFrom()\LoadWithPartialName()\ReflectionOnlyLoad()\ReflectionOnlyLoadFrom()\UnsafeLoadFrom()

                                       i.              Load(string|AssemblyName)\LoadWithPartialName()\ReflectionOnlyLoad() 使用程序集(长|短)名称加载。

                                      ii.              LoadFrom()\ReflectionOnlyLoadFrom()\UnsafeLoadFrom() 使用程序集文件名或路径加载。（注意有后缀如 .dll ）

                                    iii.              内部实现会先获取到AssemblyName，调用InternalLoadAssemblyName来加载程序集，InternalLoadAssemblyName若有传入Evidence参数的就进行Evidence凭据验证；然后再检查路径访问权限（除了UnsafeLoadFrom()）：根据程序集标识探测路径或传入路径参数来判断是否为本地文件（file://开头的URI），然后进行相应的本地文件访问权限（FileIOPermission）或Internal资源访问权限（WebPermission）判断，再调用 extern RuntimeAssembly \_nLoad() 进行程序集加载。

（不支持文件传输协议 (FTP)。如果为 assemblyFile 提供的 URI 是一个 FTP 地址，则不会加载程序集。不引发异常。）

                                    iv.              在.NET Framework 2.0 以后比如 Assembly.Load 方法有缓存，第一次加载目标程序集的失败或者成功的状态都会被缓存，这样在你下一次加载目标程序集时会直接从缓存里取目标程序集的内容和状态。

当失败后我们可能想要再加载一次或者加载多次直到成功为止，这时需要在app.config 中加入如下配置信息来禁用缓存:

<runtime> <disableCachingBindingFailures enabled="1" /> </runtime>

注意： ReflectionOnlyLoad()\ReflectionOnlyLoadFrom() 只反射加载不会使用缓存。

                                      v.              ReflectionOnlyLoad()\ReflectionOnlyLoadFrom()

1)         两个方法内部调用同Load()\LoadFrom()，只是在调用加载内部方法时“只反射方法”传入bool参数introspection(检查)为true。

2)         这两个方法是将程序集载入到“只反射上下文”，所以不可以在通过ReflectionOnly 方法加载进来的程序集执行任何方法，包括构造函数，只可以获取程序集的信息和类型。可以使用Assembly实例的ReflectionOnly属性确定该程序集是否加载到了只反射上下文中。

3)         这两个方法在载入程序集时不会将依赖程序集载入，需自行处理ReflectionOnlyAssemblyResolve事件来自行加载依赖程序集；（ReflectionOnlyLoad(byte[]) 重载也一样）

4)         加载到只反射上下文中的代码无法执行。这意味着不能创建自定义特性的实例，因为这将需要执行其构造函数。若要在只反射上下文中加载和检查自定义特性，请使用 CustomAttributeData 类。可以通过使用静态 CustomAttributeData.GetCustomAttributes 方法的相应重载获取此类的实例。

                                    vi.              LoadFrom()

在LoadFrom载入一个程序集时，会先检查前面是否已经载入过具有相同长名称的程序集(AssemblyName.FullName)----注意程序集内容不一样但FullName相同后面进行的载入会直接取前面载入的缓存。

可以通过LoadFile()方法加载不同路径拥有相同AssemblyName.FullName的多个程序集。

另外：ReflectionOnlyLoadFrom()此方法载入相同AssemblyName.FillName程序集时会报错，eg：API 限制: 程序集“file:///C:\新建文件夹\ClassLibrary123.dll”已从其他位置加载。无法从同一个 Appdomain 中的另一新位置加载该程序集。

b)         LoadFile(path)

调用 extern RuntimeAssembly nLoadFile() 进行程序集加载，不会加载程序的依赖项。

c)         Load(byte[])\ReflectionOnlyLoad(byte[])

调用 extern RuntimeAssembly nLoadImage() 进行程序集加载。

2.         动态加载程序集使用证据有两种截然不同的方式：

a)         将输入证据与加载程序所收集的证据合并，以创建用于策略决策的最终证据集。使用传入 Evidence 证据对象的方法包括 Assembly.Load、Assembly.LoadFrom 和 Activator.CreateInstance。

b)         原封不动地使用输入证据作为用于策略决策的最终证据集。使用这种语义的方法包括 Assembly.Load(byte[]) 和 AppDomain.DefineDynamicAssembly()。

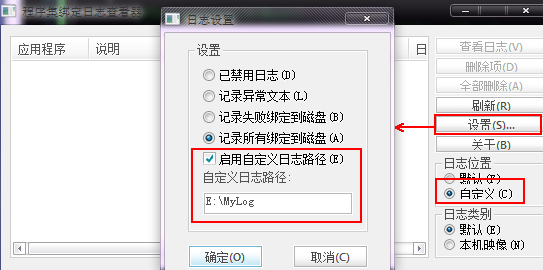
在加载时，程序集的证据用作安全策略的输入。安全策略是由企业和计算机的管理员以及用户策略设置建立的，它在执行时确定向所有托管代码授予的权限组。可以为该程序集（如果该程序集具有签名工具生成的签名）的发行者建立安全策略，或者为该程序集的下载网站和区域（就 Internet Explorer 而言）建立安全策略，也可以为该程序集的强名称建立该策略。例如，一台计算机的管理员可以建立这样一种安全策略：它允许从某一网站下载由指定软件公司签发用以访问计算机上的数据库的所有代码，但不授予对该计算机磁盘的写访问权。

二、使用Fuslogvw.exe程序集绑定日志查看器显示程序集绑定的详细信息，优化性能

启动：  
您必须具有管理员权限运行 fuslogvw.exe。   
1、如果您在计算机上安装的 Visual Studio： 的任务栏上，单击Start，单击All Programs，单击Visual Studio，单击Visual Studio Tools，用鼠标右键单击Visual Studio Command Prompt，然后单击以管理员身份运行在快捷菜单中。  
- 或 -  
2、 如果您必须在计算机上安装 Windows SDK： 的任务栏上，单击Start，单击All Programs，为 Windows SDK 中单击该文件夹，用鼠标右键单击Command Prompt （或CMD Shell），然后单击以管理员身份运行在快捷菜单中。

在管理员命令提示符下键入命令：fuslogvw

首先，按如下设置就可跟踪程序集绑定的详细信息。

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203312026188672.png)

几个跟踪示例：

1.         使用 Load() 长名称加载程序集

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/20120331202621817.jpg)

注意长名称和短名称加载程序集探测中对策略文件的影响

Eg：

使用长名称加载程序集并设置<codeBase>如下图，正确应用了配置策略

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203312026243551.jpg)

因为在使用如< assemblyIdentity >、< codeBase >等元素需要提供版本，公钥等信息，所以使用短名称不会应用配置策略。

2.         通过执行两次失败加载操作，可以验证下面方法在探测程序集时会使用已经加载过的缓存内容和状态

Load(string|AssenblyName) ，LoadWithPartialName()，LoadFrom()，UnsafeLoadFrom()

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203312026362820.jpg)

3.         LoadFile() 加载程序集过程

如果根据指定路径绑定成功，则在“没有上下文”的情况下进行加载程序集；【如果成功加载】，那么CLR还将尝试根据加载到的程序集长名称标识调用 Load() 将此程序集同时加载到“默认加载上下文”中。

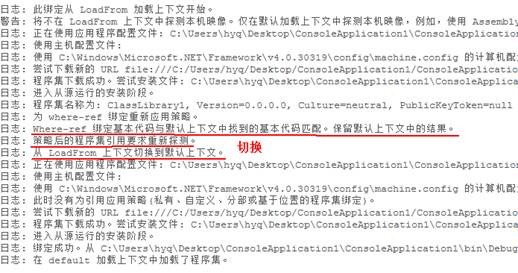
这边的两次绑定不是切换，所以为了避免出现类型标识问题（即：将同一个程序集加载到不同上下文中），尽量避免使用该方法。

4.         Load(byte[]) 加载程序集绑定信息

使用 byte[] 内存进行程序集加载的不会产生程序集绑定日志。（如果加载到的程序集标识（在应用策略后创建的）与全局程序集缓存中的程序集标识匹配，将会从全局程序集缓存加载程序集，产生程序集绑定信息，这时只加载到“默认加载上下文”中）

5.         LoadFrom()\UnsafeLoadFrom()

1)         使用文件名绑定（不是具体路径）

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203312026411768.jpg)

先使用文件名获取到程序集信息，然后根据此程序集的长名称标识调用 Load() 将此加载从“加载位置上下文”切换到“默认加载上下文”。

ReflectionOnlyLoadFrom() 使用文件名绑定时，不会发生上面的切换操作，直接加在到“加载上下文”中；使用具体路径时，不会在GAC和其他位置搜索文件，只探测指定路径位置的程序集。

6.         Load()\LoadWithPartialName() 执行部分绑定信息绑定过程

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203312026472046.jpg)

先使用部分绑定信息获取到程序集信息，然后根据此程序集的长名称标识再调用Load()方法执行加载程序集。

使用ReflectionOnlyLoad()方法不会应用版本策略，所以你指定的是哪个版本，获得的就是哪个版本。如果要自行为一个程序集标识指定版本控制策略，可将字符串传给AppDomain的ApplyPolicy()。

所以在执行程序集加载时：

1)         尽量使用程序集长名称或完整路径进行加载，以避免重复探测程序集开销。

2)         使用 LoadFile() 要注意避免不要将同一程序集加载到不同上下文中。

3)         对于只需获取程序集信息而不通过反射操作成员时，要使用“只反射加载”

### [程序集反射 Type 类](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427596.html)

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203312044478795.jpg)

Type 类

表示类型声明：类类型、接口类型、数组类型、值类型、枚举类型、类型参数、泛型类型定义，以及开放或封闭构造的泛型类型。这个类是线程安全的。

Type 为 System.Reflection 功能的根，也是访问元数据的主要方式。使用 Type 的成员获取关于类型声明的信息，如构造函数、方法、字段、属性和类的事件，以及在其中部署该类的模块和程序集。

Type 是允许多个实现的抽象基类。在反射时，并不直接使用Type类，使用的是公共语言运行时 (CLR) 提供的类型。例如，使用 C# 的 typeof 运算符获取 Type 对象时，该对象实际上是 RuntimeType。 RuntimeType 派生自 Type，并提供所有抽象方法的实现。这些运行时类是 internal。它们的行为由基类文档描述的。

抽象类Type

[clip_image002](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203312044491562.png)

只读静态字段(static readonly)：

Delimiter，EmptyTypes，FilterAttribute，FilterName，FilterNameIgnoreCase，Missing

属性：

Assembly，AssemblyQualifiedName，Attributes，BaseType，ContainsGenericParameters，DeclaringMethod，DeclaringType，FullName，GenericParameterAttributes，GenericParameterPosition，GuId，HasElementType，IsAbstract，IsAnsiClass，IsArray，IsAutoClass，IsAutoLayout，IsByRef，IsClass，IsCOMObject，IsContextful，IsEnum，IsExplicitLayout，IsGenericParameter，IsGenericType，IsGenericTypeDefinition，IsImport，IsInterface，IsLayoutSequential，IsMarshalByBef，IsNested，IsNestedAssembly，IsNestedFamANDAssem，IsNestedFamily，IsNestedFamORAssem，IsNestedPrivate，IsNestedPublic，IsNotPublic，IsPointer，IsPrimitive，IsPublic，IsSealed，IsSecurityCritical，IsSecuritySafeCritical，IsSecurityTransparent，IsSerializable，IsSpecialName，IsUnicodeClass，IsValueType，IsVisible，MemberType，Module，Namespace，ReflectedType，StructLayoutAttribute，TypeHandle，TypeInitializer，UnderlyingSystemType

方法：

Equals()，FindInterfaces()，FindMembers()，GetArrayRank()，GetConstructor()，GetConstrusctors()，GetDefaultMembers()，GetElementType()，GetEnumName()，GetEnumNames()，GetEnumUnderlyingType()，GetEnumValues()，GetEvent()，GetEvents()，GetFIeld()，GetFIelds()，GetGenericArguments()，GetGenericParameterConstraints()，GetGenericTypeDefinition()，GetHashCode()，GetInterface()，GetInterfaceMap()，GetInterfaces()，GetMember()，GetMembers()，GetMethod()，GetMethods()，GetNestedType()，GetNestedTypes()，GetProperty()，GetProperties()，GetType()，InvokeMember()，IsAssignableFrom()，IsEnumDefined()，IsEquivalentTo()，IsInstanceOfType()，IsSubclassOf()，MakeArrayType()，MakeByRefType()，MakeGenericType()，MakePoinerType()

静态属性(static)

DefaultBinder

静态方法(statuc)

GetType()，GetTypeArray()，GetTypeCode()，GetTypeFromCLSID()，GetTypeFromHandle()，GetTypeFromProgID()，GetTypeHandle()，ReflectionOnlyGetType()

1.         从当前Type中按指定条件查找MemberInfo[]的两步：

1)         获取MemberFilter 委托

[clip_image003](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203312044539704.png)

3个返回MemberFilter委托的Type静态只读属性

a)         FilterAttribute 表示用在特性上的成员筛选器。

b)         FilterName 表示用于名称的区分大小写的成员筛选器。

c)         FilterNameIgnoreCase 表示用于名称的不区分大小写的成员筛选器。

MemberFilter 委托，用于Type的实例方法FindMembers，此委托封装的方法使用两个参数：第一个是 MemberInfo 对象，第二个是 Object。此方法确定 MemberInfo 对象是否与 Object 指定的判据相匹配。

2)         使用Type的实例方法FindMembers获取MemberInfo[]

[clip_image005](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203312044585588.jpg)

参数：

MemberTypes [Flags]枚举，要筛选的成员类型；

BindingFlags 由一个或多个指定搜索执行方式的 System.Reflection.BindingFlags 组成。

MemberFilter 执行比较的委托，可以使用Type的3个静态只读属性获取，或则自行new对象，如果当前被检查的成员匹配 filterCriteria（最后一个参数），则返回 true；否则返回 false。

filterCriteria 确定成员是否在 MemberInfo 对象数组中返回的搜索判据。

a)         对于FilterAttribute返回的委托，此参数传枚举：FieldAttributes、MethodAttributes 或 MethodImplAttributes。

b)         对于FilterName和FilterNameIgnoreCase返回的委托，此参数传String，此字符串值可能包含一个【尾部】通配符字符“\*”。（仅支持通配符结尾字符串匹配）

2.         Missing与C#4.0的缺省参数

参考：<http://www.cnblogs.com/artech/archive/2011/01/12/optionalparameter.html>

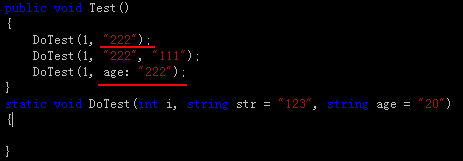
C#4.0加入了缺省参数。（以前要实现默认值只能用重载方法）

1)         缺省参数编译后会转化为标注两个特性（OptionalAttribute--指示参数是可选的)和DefaultParameterValueAttribute ）的参数形式，

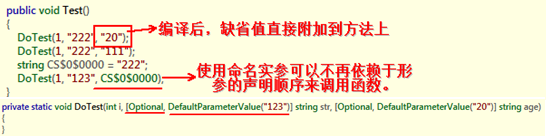
2)         缺省参数的声明放在普通参数之后，数组参数(params)之前。因为参数类型的匹配优先级就为：General Parameter > optional parameter > params parameter。

3)         指定的缺省值必须是一个常量.

示例：

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203312045048615.png)

编译后（Reflector）：

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203312045101019.png)

   在反射上调用Type的实例方法InvokeMember() 时，不能使用命名实参来改变函数声明时的形参顺序，所以需要使用Type.Missing标识缺省值作为参数。

[clip_image008](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203312045168374.png)

3.         Assembly，AssemblyQualifiedName，FullName

1)         Assembly

描述包含当前类型的程序集的 Assembly 实例。

对于泛型类型，该实例描述包含【泛型类型定义】的程序集，而不是创建和使用特定构造类型的程序集。

Assembly.FullName 获取程序集的显示名称。

EG：mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089

2)         AssemblyQualifiedName(程序集限定的类型名称)

获取 Type 的程序集限定名，其中包括从中加载 Type 的程序集的名称。

如果当前 Type 对象表示泛型参数，则此属性返回 Nothing。

EG：System.Array, mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c5

等效于( Type.FullName + “,” + Assembly.FullName )

3)         FullName (类型名称)

Type 的完全限定名，包括 Type 的命名空间，但不包括程序集；

如果当前实例表示泛型类型参数、数组类型、指针类型或基于类型参数的 ref 类型，或表示不属于泛型类型定义但包含无法解析的类型参数的泛型类型，则返回 Nothing。

另外，在使用 GetType() 搜索并加载 Type时：

a)         结合类型名(Type.FullName)时，GetType 将只在调用方的程序集中查找 Type，然后再在系统程序集中查找。

b)         结合程序集限定类型名(Type.AssemblyQualifiedName)时，GetType 将在所有程序集中查找 Type。

4.         IsGenericType，IsGenericTypeDefinition，ContainsGenericParameters

public class G<T> {}

public class Base<T, U> {}

public class Derived<V> : Base<string, V>

{

public G<Derived <V>> F;

public class Nested {}

}

1)         使用 IsGenericType 属性来确定 Type 对象是否表示泛型类型。使用 ContainsGenericParameters 属性来确定某个 Type 对象是表示开放构造的类型（不能实例化）还是封闭构造的类型（可以实例化）----是否包含未明确的类型参数。

例如： Derived<int>[] d = new Derived<int>[0]; 对于 d.GetType() 的类型：

                                       i.              IsGenericType 为 false，因为 d 是一个数组。

                                     ii.              IsGenericTypeDefinition 为 false。

                                    iii.              ContainsGenericParameters 为 false。

2)         当Type和最原始的【泛型声明】一致时IsGenericTypeDefinition才返回true，使用 IsGenericTypeDefinition 属性来确定是否能够从当前类型创建新的【类型】。

例如，Base<String,V> 其 IsGenericTypeDefinition 为 false。

3)         对于泛型的嵌套类

例如，对于typeof(Derived<>.Nested)

a)         在嵌套类型的类型参数列表中包括了封闭类型的类型参数。

EG：public class Outermost<T>

{

public class Inner<U>

{

public class Innermost1<V> {}

public class Innermost2 {}

}

}

如果封闭类型的参数列表有多个类型参数，则所有类型参数都按顺序包括在嵌套类型的类型参数列表中。

                                                     i.              嵌套类 Inner 的类型参数列表有两个类型参数 T 和 U，其中第一个是其封闭类的类型参数。

                                                   ii.              嵌套类 Innermost1 的类型参数列表有三个类型参数 T、U 和 V，其中 T 和 U 来自其封闭类。

                                                  iii.              嵌套类 Innermost2 有两个类型参数 T 和 U，它们都来自其封闭类。

为了从嵌套的类型的泛型类型定义来构造泛型类型，可以使用通过串联所有封闭类型的类型实参数组（从最外层的泛型类型开始，到嵌套的类型本身的类型实参数组结束）形成的数组来调用 MakeGenericType 方法。

b)         IsGenericType 为 true，虽然 Nested 类没有它自己的泛型类型参数，因为它嵌套在一个泛型类型中。这说明了 IsGenericType 属性的递归性质。

c)         **IsGenericTypeDefinition 为 true。**也就是说，您可以调用 MakeGenericType 方法，并提供封闭类型 Derived 的类型参数。

d)         ContainsGenericParameters 为 true，因为封闭类型 Derived 具有泛型类型参数。这一点说明了 ContainsGenericParameters 属性的递归本质。

5.         GetGenericTypeDefinition()、MakeGenericType()

如果具有的开放式泛型类型或方法不是泛型类型定义或泛型方法定义，则不能创建它的实例，也不能提供缺少的类型参数。必须具有泛型类型定义或泛型方法定义。

使用 GetGenericTypeDefinition 方法可获取泛型类型定义，再使用 MakeGenericType 方法创建封闭式泛型类型。

另：使用 MethodInfo.GetGenericMethodDefinition 方法可获取泛型方法定义，再使用 MakeGenericMethod 方法为封闭式泛型方法创建 MethodInfo。

--------------------------------------------------------------------------------

构造泛型类型的实例

1)         获取表示泛型类型的 Type 对象。下面的代码以两种不同方式获取泛型类型 Dictionary<TKey, TValue>：

a)         一种方法使用Type d1 = typeof(Dictionary<,>);

b)         另一种方法调用构造类型 Dictionary<String, Example>的 GetGenericTypeDefinition 方法。

Dictionary<string, Example> d2 = new Dictionary<string, Example>();

Type d3 = d2.GetType();

Type d4 = d3.GetGenericTypeDefinition();

2)         构造一组用于替换类型参数的类型变量。数组必须包含正确数目的 Type 对象，并且顺序和对象在类型参数列表中的顺序相同。

Type[] typeArgs = {typeof(string), typeof(Example)};

3)         调用 MakeGenericType 方法将类型变量绑定到类型参数，然后构造类型。

Type constructed = d1.MakeGenericType(typeArgs);

4)         使用 CreateInstance(Type) 方法重载来创建构造类型的对象。

object o = Activator.CreateInstance(constructed);

6.         DeclaringMethod，DeclaringType，GenericParameterPosition，GenericParameterAttributes

1)         DeclaringMethod

如果当前 Type 表示泛型方法的一个【类型参数】，则返回一个表示声明方法的 MethodBase；否则返回 Nothing。

2)         DeclaringType

如果当前的类型是嵌套类型，则返回表示封闭类型的 Type 对象；

如果当前的类型是泛型类型或泛型类型的类型参数，则返回泛型类型的定义；

如果当前的类型是泛型方法的类型参数，则返回用来声明泛型方法的类型；

否则返回为 Nothing。

3)         GenericParameterPosition

当 Type 对象表示泛型类型或泛型方法的类型参数时，获取类型参数在【声明】它的泛型类型或方法的类型参数列表中的位置。位置编号从 0 开始。

如果当前 Type 不表示类型参数，调用 GenericParameterPosition 属性将导致异常。在检查开放构造类型的类型实参时，应使用 IsGenericParameter 属性来告知哪些是类型参数。

为了给 GenericParameterPosition 属性的值提供正确的上下文，有必要标识类型参数所属的泛型类型或方法。

例如，考虑以下代码中的泛型方法 GetSomething 的返回值：

public class B<T, U> { }

public class A<V>

{

public B<V, X> GetSomething<X>()

{

return new B<V, X>();

}

}

GetSomething 返回的类型取决于提供给类 A 和 GetSomething 自身的类型参数。您可以为 GetSomething 获取 MethodInfo，然后从中获取返回类型。当检查返回类型的类型参数时，GenericParameterPosition 对二者均返回 0。因为 V 是类 A 的类型参数列表中的第一个类型参数；因为 X 是 GetSomething 的类型参数列表中的第一个类型参数。

                                    可以确定泛型类型参数的源，方法如下：

泛型类型参数可能来自要检查的类型、封闭式类型或泛型方法。

首先，使用 DeclaringMethod 属性确定类型参数是否来自泛型方法。如果该属性值不是空引用，则源是泛型方法。

如果源不是泛型方法，则使用 DeclaringType 属性确定泛型类型参数所属的泛型类型。

4)         GenericParameterAttributes

此属性的值包含了描述当前泛型类型参数是否为【协变的标志】以及类型参数约束是否为引用类型、不可为 null 的值类型和具有默认构造函数。[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203312045196092.jpg)

7.         GetGenericParameterConstraints()

返回表示当前泛型类型参数约束的 Type 对象的数组。泛型类型参数上的每个约束都表示为一个 Type 对象。

使用 IsClass 属性确定约束是否为基类约束；如果该属性返回 false，则该约束为接口约束。如果类型参数没有类约束和接口约束，则返回空数组。

8.         IsGenericParameter ，GetGenericArguments()

1)         IsGenericParameter

获取一个值，该值指示当前 Type 是否表示泛型类型或方法的定义中的类型参数。

2)         GetGenericArguments()

返回表示泛型类型的类型实参或泛型类型定义的类型形参的 Type 对象的数组。

数组元素按照它们出现在泛型类型的类型实参列表中的顺序返回。

a)         如果当前类型是封闭构造类型（即 ContainsGenericParameters 属性返回 false），则 GetGenericArguments 方法返回的数组将包含【已赋予】泛型类型定义的泛型类型形参的类型。IsGenericParameter 属性对 GetGenericArguments 方法返回的数组的每个元素返回 false。

b)         如果当前类型是泛型类型定义，该数组将包含类型形参。IsGenericParameter 属性对产生的数组的每个元素返回 true

c)         如果当前类型为开放构造类型（即 ContainsGenericParameters 属性返回 true），也就是说特定类型尚未分配给所有类型参数和封闭泛型类型或方法的类型参数，则数组【同时包含类型和类型参数】。使用 IsGenericParameter 属性来区分它们。

9.         GetDefaultMembers()

默认成员由声明类型上的 DefaultMemberAttribute 特性表示。如果类声明了索引器，请勿手动添加 DefaultMemberAttribute；在该情况下，编译器会自动添加该特性。

下面的示例显示如何通过检索默认成员的自定义特性来检索默认成员。

Type classType = typeof(Class1);

Type attribType = typeof(DefaultMemberAttribute);

DefaultMemberAttribute defMem =

(DefaultMemberAttribute)Attribute.GetCustomAttribute((MemberInfo)classType, attribType);

MemberInfo[] memInfo = classType.GetMember(defMem.MemberName);

使用 Type.GetDefaultMembers 方法会生成相同的结果，但可能更简单一些。但是，如果在类型上定义了多个默认成员，则 GetDefaultMembers 会引发 InvalidOperationException。

还可以使用 Type实例的GetCustomAttributes 方法获得类型的自定义特性，并仅选择 DefaultMemberAttribute。

10.     HasElementType，GetElementType()

HasElementType：获取一个值，通过该值指示当前 Type 是内含还是引用另一类型，即当前 Type 是数组、指针还是通过引用传递。如果 Type 为数组、指针或按引用传递，则为 true；否则为 false。

如果当前 Type 表示泛型类型，或泛型类型或泛型方法的定义中的类型参数，则此属性始终返回 false。

使用 GetElementType 来确定 Type 的元素类型。

示例：

泛型类型的数组本身不是泛型类型。您不能对诸如 C<T>[]的数组类型调用 MakeGenericType。若要基于 C<T>[] 构造封闭泛型类型，需要调用 GetElementType 以获取泛型类型定义 C<T>，对该泛型类型定义调用 MakeGenericType 以【创建构造类型】，最后对该构造类型调用 MakeArrayType 方法以【创建数组类型】。对于指针类型和 ref 类型也是如此。

11.     bool IsAssignableFrom(Type c)，bool IsInstanceOfType(object o)，bool IsSubclassOf(Type c)

1)         bool IsAssignableFrom(Type c)

确定当前的 Type 的实例是否可以从指定 Type 的实例分配。

如果满足下列任一条件，则为 true：**即：父类.IsAssignableFrom(子类)**

a)         c 和当前 Type 表示同一类型；

b)         当前 Type 位于 c 的继承层次结构中；

c)         当前 Type 是 c 实现的接口；

d)         c 是泛型类型参数且当前 Type 表示 c 的约束之一。

如果不满足上述任何一个条件或者 c 为 Nothing，则为 false。

如果 c 参数的类型为 System.Reflection.Emit.TypeBuilder，则结果基于要生成的类型。

2)         bool IsInstanceOfType(object o)

确定指定的对象是否是当前 Type 的实例。

如果满足下列任一条件，则为 true：**即：父类.IsInstanceOfType(子类对象)**

a)         当前 Type 位于由 o 表示的对象的继承层次结构中；

b)         当前 Type 是 o 支持的接口。

如果不属于其中任一种情况，或者 o 为 Nothing，或者当前 Type 为开放式泛型类型（即 ContainsGenericParameters 返回 true），则为 false。

3)         bool IsSubclassOf(Type c) **即：子类.IsSubClassOf(父类)**

确定当前 Type 表示的类是否是从指定的 Type 表示的类派生的。

如果 Type 由 c 参数表示并且当前的 Type 表示类，并且当前的 Type 所表示的类是从 c 所表示的类派生的，则为 true；否则为 false。

如果 c 和当前的 Type 表示相同的类，则此方法还返回 false。

IsSubclassOf 方法不能用于确定某个接口是否派生自另一个接口或某个类是否实现了某个接口。对于此目的，应该使用 GetInterface 方法。

12.     IsArray

获取一个值，通过该值指示 Type 是否为数组。Eg：int[] array = {1,2,3,4};

对于 Array 类，IsArray 属性返回 false。要检查数组，请使用诸如 typeof(Array).IsAssignableFrom(type) 之类的代码。

13.     IsEnum，IsClass，IsValueType

1)         IsEnum

获取一个值，该值指示当前的 Type 是否表示枚举。IsEnum属性对枚举返回 true，但是对 Enum 类型本身则不然(Enum 类型本身是class)。

2)         IsValueType

获取一个值，通过该值指示 Type 是否为值类型。

IsValueType属性对 ValueType 类返回 false，因为 ValueType 自身不是值类型。它是所有值类型的基类，因此可以将任何值类型分配给它。在将值类型分配给类型为 ValueType 的字段时，会将值类型装箱。如果 ValueType 自身是值类型，则不能这样做。

IsValueType属性对枚举返回 true，但是对 Enum 类型本身则不然。

3)         IsClass

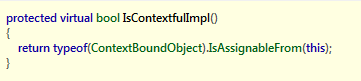
获取一个值，通过该值指示 Type 是否是一个类；即，不是值类型或接口。注意：对于表示 Enum 和 ValueType 的 Type，此属性返回 true。

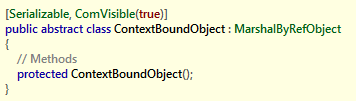
14.     IsContextful、IsMarshalByRef、IsPrimitive

检查给定类型可否在当前上下文中承载，该类型可否按引用来进行封送，以及该类型是否是基元数据类型。

基元类型为 Boolean、Byte、SByte、Int16、UInt16、Int32、UInt32、Int64、UInt64、IntPtr、UIntPtr、Char、Double 和 Single。

IsContextfil在Type中的默认实现如下：

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203312045225238.png)

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/20120331204525514.png)

ContextBoundObject

上下文可以看作应用程序域中一个包含对象和消息接收器的区域。对上下文里的对象的调用会转换成可以被MessageSink(消息接收器)拦截和处理的消息。我们知道要把调用转换成消息，必须通过透明代理这个中介。而且，仅当对象是MarshalByRefObject的子类的实例并被其所在的应用程序域以外的实体调用时，CLR才会为它创建透明代理。

ContextBoundObject的子类的实例被视为上下文绑定的(context-bound)。没有继承自ContextBoundObject的类的实例则被视为上下文灵活的(context-agile)。

a)         上下文绑定的对象永远在其上下文中执行。

b)         只要不是远程对象，上下文灵活的对象总是在执行这个调用的上下文中执行。

15.     IsSecurityCritical，IsSecuritySafeCritical，IsSecurityTransparent

1)         IsSecurityCritical [SecurityCriticalAttribute]

如果当前类型在当前信任级别上是安全关键的或安全可靠关键的，则为 true；如果它是透明的，则为 false。

2)         IsSecuritySafeCritical [SecuritySafeCriticalAttribute]

如果当前类型在当前信任级别上是安全可靠关键的，则为 true；如果它是安全关键的或透明的，则为 false。

3)         IsSecurityTransparent [SecurityTransparentAttribute]

如果该类型在当前信任级别上是安全透明的，则为 true；否则为 false。

|  |  |  |  |
| --- | --- | --- | --- |
| 安全级别 | IsSecurityCritical | IsSecuritySafeCritical | IsSecurityTransparent |
| Critical(关键的) | true | false | false |
| 安全关键的 | true | true | false |
| 透明 | false | false | true |

对于部分信任的程序集，这些属性的值取决于该程序集的当前信任级别。

a)         如果程序集被加载到一个部分信任的应用程序域（例如，加载到沙盒中的应用程序域），则运行时将忽略程序集的安全注释。将程序集及其所有类型都当作透明。

b)         仅当该程序集被加载到一个完全受信任的应用程序域时（例如，加载到桌面应用程序的默认应用程序域中），运行库才会注意部分受信任的程序集的安全注释。

c)         受信任的程序集（即，安装在全局程序集缓存中的强名称程序集）始终以完全信任的方式加载，而不管应用程序域的信任级别，因此其当前信任级别始终是完全受信任。

使用 Assembly.IsFullyTrusted 和 AppDomain.IsFullyTrusted 属性，可确定程序集和应用程序域的当前信任级别。

16.     IsVisible

获取一个指示 Type 是否可由程序集之外的代码访问的值。如果当前 Type 是公共类型或公共嵌套类型从而使所有封闭类型都是公共类型，则为 true；否则为 false.

IsVisible属性可以确定类型是否为组件程序集的公共接口的一部分。

17.     TypeHandle

获取当前 Type 的句柄。

TypeHandle 封装指向表示类型的内部数据结构的指针。此句柄在进程的生存期内是唯一的。该句柄仅在从中获取它的应用程序域中有效。

18.     在某个类型上进行反射时，Get 系列方法返回的【基类】的成员。

|  |  |  |
| --- | --- | --- |
| 成员类型 | Static | 非静态 |
| 构造函数 | 否 | 否 |
| 嵌套类型 |
| Event | 不适用 | 常规类型系统规则是：该继承与实现属性的方法的继承相同。反射将属性视为按名称和签名隐藏。 |
| Property |
| 字段 | 否 | 是。字段总是按名称和签名隐藏。 |
| 方法 | 否 | 是。方法（虚方法和非虚方法）可按名称隐藏或按名称和签名隐藏。 |

a)         【按名称和签名隐藏】会考虑签名的所有部分，包括自定义修饰符、返回类型、参数类型、标记和非托管调用约定。这是二进制比较。

b)         【属性和事件按名称和签名隐藏】如果基类中有同时带 get 访问器和 set 访问器的属性，但派生类中仅有 get 访问器，则派生类属性隐藏基类属性，并且您将无法访问基类的设置程序。

c)         自定义特性不是常规类型系统的组成部分。

19.     （静态）ReflectionOnlyGetType(string typeName, bool throwIfNotFound, bool ignoreCase)

获取具有指定名称的 Type，指定是否执行区分大小写的搜索，以及在找不到类型时是否引发异常。该类型只为反射加载，而不为执行加载。

a)         如果包含该类型的程序集尚未加载到仅反射上下文中，且该类型是程序集限定类型，则使用 ReflectionOnlyGetType 方法等效于首先使用 ReflectionOnlyLoad 方法仅为反射加载程序集，然后通过调用程序集的 Assembly.GetType 方法加载该类型。

b)         如果程序集已为执行而加载，则另一个副本加载到仅“反射上下文”中。

20.     Type MakeByRefType()，Type MakePointerType()

MakeByRefType() 返回表示作为 ref 参数传递时的当前类型的 Type 对象。使用 Microsoft 中间语言 (MSIL) 的语法，如果当前 Type 对象表示 Int32，则此方法将返回表示 Int32& 的 Type 对象。

MakePointerType() 返回表示指向当前类型的指针的 Type 对象。使用 Microsoft 中间语言 (MSIL) 的语法，如果当前 Type 对象表示 Int32，则此方法将返回表示 Int32\* 的 Type 对象。

EG：public void Test(ref Example e){ }

Type t = typeof(Example).MakeByRefType(); // ToString() 输出“Example&”

Type t = typeof(Example).MakePointerType(); // ToString() 输出” Example\*”

21.     GetInterfaceMap(Type interfaceType)

我们通过 FindInterfaces()，GetInterfaces()，GetInterface() 方法可以获取指定类型及其所有基类型上定义的所有接口。但是，我们不能得到接口与实现该接口的实际方法之间的映射。这时我们就需要调用 GetInterfaceMap() 返回映射关系类InterfaceMapping类。

InterfaceMapping 类型定义的4个公共字段

|  |  |  |
| --- | --- | --- |
| 字段名称 | 数据类型 | 说明 |
| TargetType | Type | 这是用于调用GetInterfaceMapping的类型 |
| InterfaceType | Type | 传给GetInterfaceMapping的接口类型 |
| InterfaceMethods | MethodInfo[] | 一个数组，它的每个元素都暴露了与一个接口的方法有关的信息 |
| TargetMethods | MethodInfo[] | 一个数组，它的每个元素都暴露了由类型定义的一个方法的信息，这个方法实现了接口中对应的方法 |

InterfaceMethods 和 TargetMethods数组是互为对应的

### [程序集的加载和反射](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427617.html)

一、程序集的加载

程序集是 .NET Framework 应用程序的构造块；程序集构成了部署、版本控制、重复使用、激活范围控制和安全权限的基本单元。

绑定是查找与唯一指定的类型相对应的声明(即实现)的过程。根据此过程是发生在编译时还是运行时分为：

a)         静态绑定：在生成时，编译器在程序集清单的元数据中记录静态引用。

b)         动态绑定：由于调用各种方法而动态构造的，EG： Assembly.Load 方法。

程序集如何加载请参见下面链接：

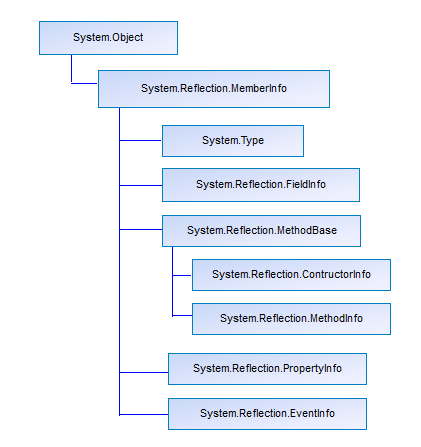
1、 [《(5)CLR 运行时探测程序集引用的步骤》](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427557.html)

2、 [《(6)程序集加载上下文》](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427574.html)

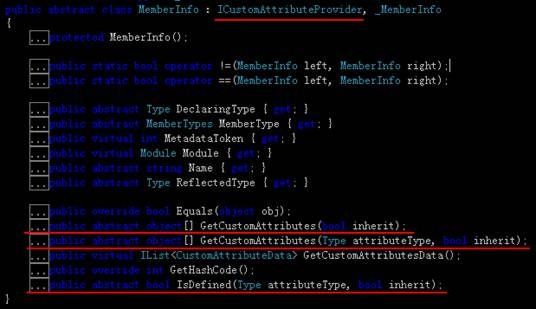
3、 [《(7)动态程序集加载Load()》](http://www.cnblogs.com/heyuquan/archive/2012/03/31/2427582.html)

二、程序集的反射

字段、构造器、方法、属性、事件和嵌套类型都可以被定义成一个类型的成员。FCL(Framework Class Library—框架类库)定义了一个抽象基类 MemberInfo ，封装了一组所有类型成员都通用的属性。

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203312055251818.png)

对“动态绑定”的程序集进行成员绑定的规则可通过 Binder 类重写，作为搜索成员方法的参数传入；如果此参数传null，则使用默认绑定规则进行成员绑定同“静态绑定”。

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/20120331205602986.jpg)

1.         区分DeclaringType 和ReflectedType

a)         ReflectedType 获取用于获取 MemberInfo 的此实例的类对象。返回的类对象总是执行反射的Type起点。

b)         DeclaringType 获取声明该成员的类。

[](http://images.cnblogs.com/cnblogs_com/heyuquan/201203/201203312056153187.jpg)

2.         GetCustomAttributes()

运行时自定义特性反射模型

-------System.DLL

public class DescriptionAttribute : Attribute

{ ……}

-------System.Web.DLL

internal class MyDescriptionAttribute : DescriptionAttribute

{ ……}

public class LocalizationExtenderProvider

{

[MyDescriptionAttribute(...)]

public CultureInfo GetLanguage(...)

{ ……}

}

如果运行时尝试使用GetCustomAttributes(Type type) 为附加到 GetLanguage 方法的公共自定义特性类型 DescriptionAttribute 检索自定义特性，则该运行时将执行下列操作：

a)         运行时检查 GetCustomAttributes(Type type) 的 DescriptionAttribute 类型参数是否为公共的，判断是否可见和访问。

b)         运行时检查从 DescriptionAttribute 派生的用户定义类型 MyDescriptionAttribute 在 System.Web.DLL 程序集（它在该程序集中附加到 GetLanguage() 方法）内是否可见和可以访问。

c)         运行时检查 MyDescriptionAttribute 的【构造函数】是否在 System.Web.DLL 程序集中可见和可以访问。

a)        运行时调用带有自定义特性参数的 MyDescriptionAttribute 的构造函数，然后将新对象返回给调用方。（“只反射上下问”因为不能执行代码，所以需使用 CustomAttributeData 类访问自定义特性）

三、获取Type对象的引用

1)         C# typeof 操作符获得某个类型的 Type 对象，typeof操作符相对其他获取type方式能获得更高的性能，操作符用于获取“早期绑定”的类型信息。

2)         Object.GetType 方法返回表示【实例类型】的 Type 对象。可以直接用 o1.GetType()==o2.GetType() 来判断对象是否属于同一个类型。

3)         System.Reflection.Assembly 对象可先Load尚未加载的程序集，再获取 Type 对象，包括 Assembly.GetType()、Assembly.GetTypes() 和 Assembly.GetExportedTypes()。

4)         Module.GetTypes、Module.GetType 和 Module.FindTypes 方法返回 Type 对象，这些对象表示在某个模块中定义的类型。第一个方法可用于获得模块中定义的所有公共类型和私有类型的 Type 对象的数组。（可通过 Assembly.GetModule()、Assembly.GetModules ()或 Type.Module 属性来获得 Module 的实例。）

5)         Type类方法：

a)         静态 Type.GetType 方法可从已加载的程序集中获取 Type 对象，该对象表示由其完全限定名指定的类型。

b)         FindInterfaces 方法返回某个类型所支持的接口类型的筛选后的列表。

c)         GetElementType 方法返回表示元素的 Type 对象。

d)         GetInterfaces 和 GetInterface 方法返回表示某个类型所支持的接口类型的 Type 对象。

e)         GetTypeArray 方法返回表示任意一组对象所指定的类型的 Type 对象数组。这些对象用 Object 类型的数组指定。

f)          GetTypeFromProgID 和 GetTypeFromCLSID 方法是为 COM 互操作而提供的。这些方法返回表示 ProgID 或 CLSID 所指定的类型的 Type 对象。

g)         GetTypeFromHandle 方法是为COM互操作而提供的。此方法返回表示类句柄所指定的类型的 Type 对象。

h)         MakeGenericType 方法返回 Type 对象，该对象表示构造泛型类型，如果该对象的 ContainsGenericParameters 属性返回 true，则该类型为开放构造类型，否则为封闭构造类型。只能实例化封闭的泛型类型。

i)           MakeArrayType、MakePointerType 和 MakeByRefType 方法返回 Type 对象，这些对象分别表示指定类型的数组、指向指定类型的指针以及引用参数的类型（在 C# 中为 ref）。

四、构造类型的实例

1)         System.Activator 的 CreateInstance() 静态方法

注意CLR不要求值类型定义任何构造函数，所以想在不调用构造器的情况下创建值类型的一个实例，必须调用 Activator.CreateInstance() 并且是传入Type 或传入Type和Boolean参数的重载。

2)         System.Activator 的 CreateInstanceFrom() 静态方法

通过指定程序集文件名、类型名等参数，内部会根据传入的程序集文件名调用Assembly的LoadFrom()加载程序集。返回值不是对新对象的一个引用，而是ObjectHandle对象引用（派生自MarshalByRefObject），必须调用ObjectHandle的Unwarp()方法解包对象。

3)         System.AppDomain的实例方法

CreateInstance()，CreateInstanceAndUnwrap()，CreateInstanceFrom()，CreateInstanceFromAndUnwrap()。同Activator类方法行为类似。

4)         System.Type的 InvokeMember() 实例方法

5)         System.Reflection.ConstructorInfo类的Invoke() 实例方法

6)         Array的CreateInstance() 静态方法，创建System.Array及其派生类的实例

7)         Delegate的 CreateDelegate() 静态方法，创建委托（System.MulticastDelegate派生类型）的实例

五、反射的性能

反射机制在提供运行时发现并使用编译时还不了解的类型及其成员功能时，也存有其缺点：

1.反射会造成编译时无法保证类型安全性

由于反射要严重依赖字符串，所以会丧失编译时的类型安全性（即：编译成功，运行报错）。

2.反射速度慢

a)搜索：使用 System.Reflection 命名空间中的类型扫描程序集的元数据时，反射要不断的执行字符串的搜索。通常，搜索时不区分大小写的比较，这会更进一步影响性能。

b)调用：使用反射调用一个成员时。比如调用方法，首先必须将实参打包(pack)成一个数组；在内部，反射必须将这些实参解包(unpack)到线程栈上。此外，在调用方法前，CLR必须检查实参具有正确的数据类型。最后，CLR必须确保调用者有正确的安全权限来访问被调用的成员。

六、调用类型的成员

1.         Type 的 InvokeMember() 实例方法  （出于性能和扩展考虑，推荐直接使用第二种缓存方式）

通过指定：成员名称、BindingFlags、要在其上调用成员的Object、调用成员的参数数组等参数进行成员调用

该方法会执行两个操作。

a)         绑定成员：选择要调用的一个恰当的成员

b)         调用：实际调用成员

2.         一次绑定，多次调用

a)         绑定成员：先通过 Type 的如 GetFields()，GetConstructors()，GetMethods()，GetProperties()，GetEvents() 等方法获取并缓存这个对象引用，该对象的类型提供了直接访问特定成员的方法。

b)         调用：

|  |  |
| --- | --- |
| 成员类型 | 用于调用成员的方法 |
| FieldInfo | 调用GetValue获取字段的值  调用SetValue设置字段的值 |
| PropertyInfo | 调用GetValue调用属性的get访问器方法  调用SetValue调用属性的set访问器方法 |
| EventInfo | 调用AddEventHandler调用事件的add访问器方法  调用RemoveEventHandler调用事件的remove访问器方法 |
| ConstructorInfo | 调用Invoke构造类型的一个实例，并调用一个构造器 |
| MethodInfo | 调用Invoke调用类型的一个方法 |

## 资源回收机制

### 垃圾回收（GC）Gabage Collector

https://www.cnblogs.com/wonderKK/p/4133122.html

JAVA 和 .net等动态语言

### JAVa回收机制

Java采用VM（Virtual Machine）机制，由VM来管理程序的运行当然也包括对GC管理。

.net采用了和Java类似的方法由CLR(Common Language Runtime)来管理。这两大阵营的出现将人们引入了以虚拟平台为基础的开发时代，GC也在这个时候越来越得到大众的关注。

GC如其名，就是垃圾收集，当然这里仅就内存而言。Garbage Collector（垃圾收集器，在不至于混淆的情况下也成为GC）以应用程序的root为基础，遍历应用程序在Heap上动态分配的所有对象[2]，通过识别它们是否被引用来确定哪些对象是已经死亡的哪些仍需要被使用。已经不再被应用程序的root或者别的对象所引用的对象就是已经死亡的对象，即所谓的垃圾，需要被回收。这就是GC工作的原理。为了实现这个原理，GC有多种算法。比较常见的算法有Reference Counting，Mark Sweep，Copy Collection等等。目前主流的虚拟系统.net CLR，Java VM和Rotor都是采用的Mark Sweep算法。

### C++回收机制

C++的智能指针调用，进行资源回收。

<http://blog.csdn.net/yeahhook/article/details/6796242>

### C/C++中几种经典的垃圾回收算法

<http://blog.csdn.net/wallwind/article/details/6889917>

## 对象的引用和销毁

1、一个对象中的委托，挂载了另外一个对象的方法，如果另外一个对象new了一个新的实例或者为null时，挂载的委托方法仍然执行！！！！！

这个对象还是存在的！！！！！！！！！

https://www.cnblogs.com/yang\_sy/p/3784151.html

https://www.cnblogs.com/solan/archive/2012/08/24/CSharp11.html

C# 垃圾回收机制。

https://www.cnblogs.com/wonderKK/p/4133122.html

C# 垃圾回收机制 (Garbage Collector（垃圾收集器，在不至于混淆的情况下也成为GC）

只针对对象，只针对引用型。

如果对象不在被引用，就会被清理，

清理的算法不同。

## 为什么重写equals()方法必须要重写hashCode()方法？

<https://blog.csdn.net/cnd2449294059/article/details/102733301>

# **数据库相关**

## ORM

对象关系映射（Object Relational Mapping，简称ORM）是一种为了解决[面向对象](http://baike.baidu.com/view/125370.htm)与关系数据库存在的互不匹配的现象的技术。

对象关系映射（Object Relational Mapping，简称ORM）是通过使用描述对象和数据库之间映射的[元数据](http://baike.baidu.com/subview/107838/107838.htm)，将java程序中的对象自动持久化到关系数据库中。

实际应用中即在关系型数据库和[业务实体](http://baike.baidu.com/subview/4907952/4908984.htm)对象之间作一个映射，这样，我们在具体的操作[业务对象](http://baike.baidu.com/subview/478351/478351.htm)的时候，就不需要再去和复杂的[SQL语句](http://baike.baidu.com/subview/1040165/1040165.htm)打交道，只要像平时操作对象一样操作它就可以了。  
　　ORM框架就是用于实现ORM技术的程序。  
　　常见的ORM框架有：[Hibernate](http://baike.baidu.com/subview/7291/7291.htm)、[iBATIS](http://baike.baidu.com/subview/628102/628102.htm)、[TopLink](http://baike.baidu.com/subview/1415795/1415795.htm)、Castor JDO、Apache OJB等。

## E-R图

E-R图也称实体-联系图(Entity Relationship Diagram)，提供了表示实体类型、属性和联系的方法，用来描述现实世界的[概念模型](http://baike.baidu.com/view/176360.htm)。

·**实体型**(Entity)：具有相同属性的实体具有相同的特征和性质，用实体名及其属性名集合来抽象和刻画同类实体**;**在E-R图中用**矩形**表示，矩形框内写明实体名；比如学生[张三](http://baike.baidu.com/view/283918.htm)、学生李四都是实体。如果是[弱实体](http://baike.baidu.com/view/1966692.htm)的话，在矩形外面再套实线矩形。

·**属性**(Attribute)：实体所具有的某一特性，一个实体可由若干个属性来刻画。在E-R图中用**椭圆形**表示，并用无向边将其与相应的实体连接起来；比如学生的姓名、学号、性别、都是属性。如果是多值属性的话，在椭圆形外面再套实线椭圆。如果是派生属性则用虚线椭圆表示。

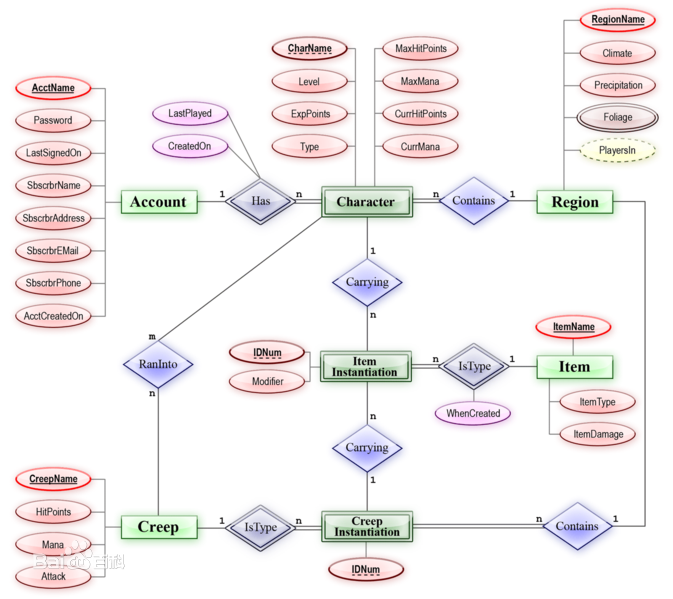
·**联系**(Relationship)：联系也称关系，信息世界中反映实体内部或实体之间的联系。实体内部的联系通常是指组成实体的各属性之间的联系；实体之间的联系通常是指不同[实体集](http://baike.baidu.com/view/176381.htm)之间的联系。在E-R图中用**菱形**表示，菱形框内写明联系名，并用无向边分别与有关实体连接起来，同时在无向边旁标上联系的类型（1 : 1，1 : n或m : n）。比如老师给学生授课存在授课关系，学生选课存在选课关系。如果是弱实体的联系则在菱形外面再套菱形。

矩形框：表示实体，在框中记入实体名。

[菱形](http://baike.baidu.com/view/133728.htm)框：表示联系，在框中记入联系名。

椭圆形框：表示实体或联系的属性，将属性名记入框中。对于[主属性](http://baike.baidu.com/view/703562.htm)名，则在其名称下划一下划线。

连线：实体与属性之间；实体与联系之间；联系与属性之间用直线相连，并在直线上标注联系的类型。（对于一对一联系，要在两个实体连线方向各写1； 对于一对多联系，要在一的一方写1，多的一方写N；对于多对多关系，则要在两个实体连线方向各写N,M。)[1]



# **异步调用，并行开发**

（1）普通方法调用（直接调用）与Invoke（）方法调用方法 使用的线程Id是一样的 即属于同步。

（2）BeginInvoke(<输入和输出变量>，AsyncCallback callback,object asyncState)方法调用方法 则是启用了新的线程Id       属于异步

可通过 Thread.CurrentThread.ManagedThreadId    获取当前线程的Id

\*委托类型的BeginInvoke(<输入和输出变量>，AsyncCallback callback,object asyncState)方法

异步调用的核心， BeginInvoke（多线程能执行的原因）。

输入和输出变量：表式委托对应的实参。

第二个参数（AsyncCallback callback）：回调函数，表示异步调用后自动调用的函数 ,共用一个线程id.  AsyncCallback是一个委托   有一个参数（asyncState ）

第三个参数（object asyncState）：用于向回调函数提供参数信息。返回值：IasyncResult:异步操作状态接口，封装了异步执行中的参数。

EndInvoke监视BeginInvoke。委托类型的EndInvoke（）方法：借助IasyncResult接口对象，不断查询异步调用是否结束。该方法知道异步调用的方法所有参数，所以，异步调用完毕后，取出异步调用的结果作为返回值。

对于同步和异步的总结：**异步三大特点**

（1）同步方法会出现“假死现象”即卡住界面，异步则不会 原因：异步启动了子线程执行任务，主线程得到释放

（2）同步速度慢，异步速度快  原因：异步启动了多个线程执行任务，占用更多的资源（异步时cpu瞬间上升）

（3）异步是无序的 原因：线程的启动和执行是由操作系统决定的，是无序的。可能每个子线程里面耗时不一样

http://www.cnblogs.com/durow/p/4826653.html

http://www.makaidong.com/%E5%8D%9A%E5%AE%A2%E5%9B%AD%E7%B2%BE/20150925/42013.html

Task.Factory.StartNew

<https://www.cnblogs.com/duanbiflying/p/6400605.html>

## [C#中Func与Action的理解](https://www.cnblogs.com/ultimateWorld/p/5608122.html)

<https://www.cnblogs.com/ultimateWorld/p/5608122.html>

https://www.cnblogs.com/fish-li/archive/2011/10/23/2222013.html

<https://www.cnblogs.com/lsxqw2004/p/4922374.html>

异步调用详细

介绍非常详细，各个版本的C#和对应的C++

fun

action

https://www.cnblogs.com/springsnow/p/9409205.html

并行编程

delegate：需要预先使用delegate关键词定义好委托对象才能使用，参数任意、返回值任意

　　event：无需定义委托，直接使用event关键词来定义好即可，参数限定为两个，第一个object类型的sender（一般传事件源），第二个继承EventArgs的泛型参数（事件参数），返回值任意

　　Action：无需任何定义，直接把它当做一种类型来使用即可，一般作为方法参数，进行回调操作，例如导入进度回调通知，只有一个泛型参数，无返回值。

　　Func：无需任何定义，和Action类似，只是它具有返回值

task

ansy await

https://www.cnblogs.com/neverc/p/4653539.html

异步并行编程，比较详细。

task.run

task.Task.Factory.StartNew(

//===========================================================

//===========================================================

关键字 params,可以实现函数，数组为参数的，多个并列的输入

private void testParamsMethod()

{

testmethond(100, 200);

}

private void testmethond(params int[] used)

{

}

## 异步和线程并行方面讲解得很详细

回调，类似于中断，通常会中断正在执行的代码，并按照回调顺序执行。

代理，事件基本类似

https://www.cnblogs.com/zhangweizhong/p/6628442.html

[AutoResetEvent的使用介绍（用AutoResetEvent实现同步）](http://www.cnblogs.com/zhangweizhong/p/6628442.html)

C# 中的$

c# $ = string.format

线程间常用的

线程

锁

互斥量

原子操作

信号量

机会异常

线程安全的

C#

Begininvoke、Invoke、Ansy 、Wait、Task

Lock、mutex、ManualResetEvent

线程池

[C#使用Interlocked进行原子操作](http://www.cnblogs.com/Scl891004X/p/6122405.html)

http://www.cnblogs.com/Scl891004X/archive/2016/12/01/6122405.html

使用Interlocked进行原子操作：

原子操作的机制

http://www.cnblogs.com/kissdodog/archive/2013/04/07/3003822.html

<http://www.cnblogs.com/apsnet/archive/2012/07/08/2581475.html>

<http://www.cnblogs.com/xinhuawei/p/5398193.html#header04>

<http://www.cnblogs.com/kissdodog/archive/2013/04/07/3003822.html>

## 异步处理参考

<http://www.cnblogs.com/zhaopei/p/async_two.html>

（1）普通方法调用（直接调用）与Invoke（）方法调用方法 使用的线程Id是一样的 即属于同步。

（2）BeginInvoke(<输入和输出变量>，AsyncCallback callback,object asyncState)方法调用方法 则是启用了新的线程Id       属于异步

可通过 Thread.CurrentThread.ManagedThreadId    获取当前线程的Id

\*委托类型的BeginInvoke(<输入和输出变量>，AsyncCallback callback,object asyncState)方法

异步调用的核心， BeginInvoke（多线程能执行的原因）。

输入和输出变量：表式委托对应的实参。

第二个参数（AsyncCallback callback）：回调函数，表示异步调用后自动调用的函数 ,共用一个线程id.  AsyncCallback是一个委托   有一个参数（asyncState ）

第三个参数（object asyncState）：用于向回调函数提供参数信息。返回值：IasyncResult:异步操作状态接口，封装了异步执行中的参数。

EndInvoke监视BeginInvoke。委托类型的EndInvoke（）方法：借助IasyncResult接口对象，不断查询异步调用是否结束。该方法知道异步调用的方法所有参数，所以，异步调用完毕后，取出异步调用的结果作为返回值。

对于同步和异步的总结：**异步三大特点**

（1）同步方法会出现“假死现象”即卡住界面，异步则不会 原因：异步启动了子线程执行任务，主线程得到释放

（2）同步速度慢，异步速度快  原因：异步启动了多个线程执行任务，占用更多的资源（异步时cpu瞬间上升）

（3）异步是无序的 原因：线程的启动和执行是由操作系统决定的，是无序的。可能每个子线程里面耗时不一样

http://www.cnblogs.com/durow/p/4826653.html

## lock相关

1、如果一个类的实例是public的，最好不要lock(this)。因为使用你的类的人也许不知道你用了lock，如果他new了一个实例，并且对这个实例上锁，就很容易造成死锁。

2、如果MyType是public的，不要lock(typeof(MyType))。

3、永远也不要lock一个字符串。

somePrivateStaticObject

1.避免锁定public类型对象。

　　如果实例可以被公共访问，将出现lock(this)问题。

　　如有一个类MyClass，该类有一个Method方法通过lock(this)来实现互斥：

复制代码

1 public class MyClass

2 {

3 public void Method()

4 {

5 lock(this)

6 {

7 ......

8 }

9 }

10 }

复制代码

　　如果一个MyClass的实例在不同线程中执行Method方法，可以实现互斥。但如果多个MyClass的实例分别在不同的线程中执行Method方法，互斥将失效，因为此处的lock(this)仅对当前的实例对象进行了加锁。

2.禁止锁定类型

　　lock(typeof(ClassName))锁定范围更为广泛，由于一个类的所有实例都只有一个类型对象（该对象是typeof的返回结果），锁定它，就锁定了该对象的所有实例。微软现在建议（原文请参考：http://www.microsoft.com/china/MSDN/library/enterprisedevelopment/softwaredev/SDaskgui06032003.mspx?mfr=true）不要使用lock(typeof(ClassName))。以下的微软的原文描述：

　　首先锁定类型对象是个很缓慢的过程，并且类中的其他线程、甚至在同一个应用程序域中运行的其他程序都可以访问该类型对象，因此，它们就有可能代替您锁定类型对象，完全阻止您的执行，从而导致你自己的代码的挂起。

　　这里的基本问题是，您并未拥有该类型对象，并且您不知道还有谁可以访问它。总的来说，依靠锁定不是由您创建、并且您不知道还有谁可以访问的对象是一种很不好的做法。这样做很容易导致死锁。

3.禁止锁定字符串

　　锁住一个字符串更为神奇，只要字符串内容相同，就能引起程序挂起。

　　在.NET中，字符串会被暂时存放，如果两个变量的字符串内容相同的话，.NET会把暂存的字符串对象分配给该变量。所以如果有两个地方都在使用lock("my lock")的话，它们实际锁住的是同一个对象。

如何正确使用lock呢？

　　微软给出的建议是：只锁定私有对象。

　　示例代码：

复制代码

1 public class MyClass

2 {

3 private static Object somePrivateStaticObject = new Object();

4

5 // methods of class go here--can lock somePrivateStaticObject

6 public void Method()

7 {

8 lock(somePrivateStaticObject)

9 {

10 ......

11 }

12 }

13 }

复制代码

锁定私有对象的好处：

　　首先，类以外的任何代码都无法锁定MyClass.somePrivateStaticObject，因此避免了许多死锁的可能。由于死锁属于那种最难找到根源的问题，因此，避免发生死锁的可能是一件很好的事情。

其次，应用程序中只有一份MyClass.somePrivateStaticObject的副本，并且系统上运行的其他每个应用程序也只有一个副本。因此，在同一个应用程序域中的应用程序之间没有相互影响。

## 同步/异步， 阻塞于非阻塞

本来是想介绍回调函数在非阻塞模式的地位，但是在这之前需要知道这些基本知识。然后再验证回调函数的作用

一、同步与异步

同步/异步, 它们是消息的通知机制   
1. 概念解释   
A. 同步   
所谓 同步，就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回。   
按照这个定义，其实绝大多数函数都是同步调用（例如sin isdigit等）。   
但是一般而言，我们在说同步、异步的时候，特指那些需要其他部件协作或者需要一定时间完成的任务。   
最常见的例子就是 SendMessage。   
该函数发送一个消息给某个窗口，在对方处理完消息之前，这个函数不返回。   
当对方处理完毕以后，该函数才把消息处理函数所返回的值返回给调用者。   
B. 异步   
异步的概念和同步相对。   
当一个异步过程调用发出后，调用者不会立刻得到结果。   
实际处理这个调用的部件是在调用发出后，   
通过状态、通知来通知调用者，或 通过回调函数处理这个调用。   
以 Socket为例，   
当一个客户端通过调用 Connect函数发出一个连接请求后，调用者线程不用等待结果，可立刻继续向下运行。   
当连接真正建立起来以后，socket底层会发送一个消息通知该对象。   
C. 三种返回结果途径    
执行部件和调用者可以通过三种途径返回结果：   
a.    状态、   
b.   通知、   
c.    回调函数。   
可以使用哪一种依赖于执行部件的实现，除非执行部件提供多种选择，否则不受调用者控制。   
a. 如果执行部件用状态来通知，   
    那么调用者就需要每隔一定时间检查一次，效率就很低   
    有些初学多线程编程的人，总喜欢用一个循环去检查某个变量的值，这其实是一种很严重的错误。   
b. 如果是使用通知的方式，   
    效率则很高，因为执行部件几乎不需要做额外的操作。   
  
c. 至于回调函数，   
    和通知没太多区别。   
2. 举例说明   
理解这两个概念，可以用去银行办理业务(可以取钱，也可以存钱)来比喻:   
当到银行后,   
.可以去ATM机前排队等候                                  -- (排队等候)就是同步等待消息   
.可以去大厅拿号,等到排到我的号时，   
 柜台的人会通知我轮到我去办理业务.               -- (等待别人通知)就是异步等待消息.   
在异步消息通知机制中,   
等待消息者(在这个例子中就是等待办理业务的人)往往注册一个回调机制,   
在所等待的事件被触发时由触发机制(在这里是柜台的人)通过某种机制(在这里是写在小纸条上的号码)   
找到等待该事件的人.   
在select/poll 等IO 多路复用机制中就是fd,   
当消息被触发时,触发机制通过fd 找到处理该fd的处理函数.   
3. 在实际的程序中,   
同步消息通知机制：就好比简单的read/write 操作,它们需要等待这两个操作成功才能返回;   
                  同步, 是由处理消息者自己去等待消息是否被触发;   
异步消息通知机制：类似于select/poll 之类的多路复用IO 操作,   
                  当所关注的消息被触发时,由消息触发机制通知触发对消息的处理.   
                  异步, 由触发机制来通知处理消息者;   
还是回到上面的例子,   
轮到你办理业务, 这个就是你关注的消息,   
而办理什么业务, 就是对这个消息的处理,   
两者是有区别的.   
而在真实的IO 操作时: 所关注的消息就是     该fd是否可读写,   
                     而对消息的处理是     对这个fd 进行读写.   
同步/异步仅仅关注的是如何通知消息,它们对如何处理消息并不关心,   
好比说,银行的人仅仅通知你轮到你办理业务了,   
而办理业务什么业务(存钱还是取钱)他们是不知道的.   
二、阻塞与非阻塞   
阻塞/非阻塞, 它们是程序在等待消息(无所谓同步或者异步)时的状态.   
1. 概念解释   
A. 阻塞   
阻塞调用是指调用结果返回之前，当前线程会被挂起。函数只有在得到结果之后才会返回。   
有人也许会把阻塞调用和同步调用等同起来，实际上他是不同的。   
对于同步调用来说，很多时候当前线程还是激活的，只是从逻辑上当前函数没有返回而已。   
socket接收数据函数recv是一个阻塞调用的例子。   
当socket工作在阻塞模式的时候， 如果没有数据的情况下调用该函数，则当前线程就会被挂起，直到有数据为止。   
?   
B. 非阻塞   
非阻塞和阻塞的概念相对应， 指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回。   
C. 对象的阻塞模式和阻塞函数调用   
对象是否处于阻塞模式和函数是不是阻塞调用有很强的相关性，但是并不是一一对应的。   
阻塞对象上可以有非阻塞的调用方式，我们可以通过一定的API去轮询状态，   
在适当的时候调用阻塞函数，就可以避免阻塞。   
而对于非阻塞对象，调用特殊的函数也可以进入阻塞调用。函数select就是这样的一个例子。   
2. 举例说明   
继续上面的那个例子,   
不论是排队等待，还是使用号码等待通知,   
如果在这个等待的过程中,   
. 等待者除了等待消息之外不能做其它的事情,那么该机制就是阻塞的,   
  表现在程序中,也就是该程序一直阻塞在该函数调用处不能继续往下执行.   
. 相反,有的人喜欢在银行办理这些业务的时候一边打打电话发发短信一边等待,这样的状态就是非阻塞的,   
  因为他(等待者)没有阻塞在这个消息通知上,而是一边做自己的事情一边等待.   
三、易混淆的点   
很多人也会把异步和非阻塞混淆,   
因为异步操作一般都不会在真正的IO 操作处被阻塞,   
比如如果用select 函数,当select 返回可读时再去read 一般都不会被阻塞   
就好比当你的号码排到时一般都是在你之前已经没有人了,所以你再去柜台办理业务就不会被阻塞.   
可见,同步/异步与阻塞/非阻塞是两组不同的概念,它们可以共存组合,   
而很多人之所以把同步和阻塞混淆,我想也是因为没有区分这两个概念,   
比如阻塞的read/write 操作中,其实是把消息通知和处理消息结合在了一起,   
在这里所关注的消息就是fd 是否可读/写,而处理消息则是对fd 读/写.   
当我们将这个fd 设置为非阻塞的时候,read/write 操作就不会在等待消息通知这里阻塞,   
如果fd 不可读/写则操作立即返回.   
四、同步/异步与阻塞/非阻塞的组合分析   
\_\_\_\_\_\_\_阻塞\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_非阻塞\_\_\_\_\_   
同步 | 同步阻塞              同步非阻塞   
异步 | 异步阻塞              异步非阻塞   
同步阻塞形式:   
  效率是最低的，   
  拿上面的例子来说，就是你专心排队，什么别的事都不做。   
  实际程序中   
  就是未对fd 设置O\_NONBLOCK 标志位的read/write 操作,   
异步阻塞形式:   
  如果在银行等待办理业务的人采用的是异步的方式去等待消息被触发,也就是领了一张小纸条,   
  假如在这段时间里他不能离开银行做其它的事情,那么很显然,这个人被阻塞在了这个等待的操作上面;   
  异步操作是可以被阻塞住的,只不过它不是在处理消息时阻塞,而是在等待消息被触发时被阻塞.   
  比如select 函数,   
  假如传入的最后一个timeout 参数为NULL,那么如果所关注的事件没有一个被触发,   
  程序就会一直阻塞在这个select 调用处.   
同步非阻塞形式:   
  实际上是效率低下的,   
  想象一下你一边打着电话一边还需要抬头看到底队伍排到你了没有,   
  如果把打电话和观察排队的位置看成是程序的两个操作的话,   
  这个程序需要在这两种不同的行为之间来回的切换,效率可想而知是低下的;   
  很多人会写阻塞的read/write 操作,   
  但是别忘了可以对fd 设置O\_NONBLOCK 标志位,这样就可以将同步操作变成非阻塞的了;   
异步非阻塞形式:   
  效率更高,   
  因为打电话是你(等待者)的事情,而通知你则是柜台(消息触发机制)的事情,   
  程序没有在两种不同的操作中来回切换.   
  比如说，这个人突然发觉自己烟瘾犯了,需要出去抽根烟,   
  于是他告诉大堂经理说,排到我这个号码的时候麻烦到外面通知我一下(注册一个回调函数),   
  那么他就没有被阻塞在这个等待的操作上面,自然这个就是异步+非阻塞的方式了.   
?   
  如果使用异步非阻塞的情况,   
  比如aio\_\*组的操作,当发起一个aio\_read 操作时,函数会马上返回不会被阻塞,

  当所关注的事件被触发时会调用之前注册的回调函数进行处理,

还有相关的文章:   
首先来解释同步和异步的概念,这两个概念与 **消息的通知机制** 有关.   
举个例子,比如我去银行办理业务,可能选择排队等候,也可能取一个小纸条上面有我的号码,等到排到我这一号时由柜台的人通知我轮到我去办理业务了.   
前者(排队等候)就是同步等待消息,而后者(等待别人通知)就是异步等待消息.在异步消息处理中,等待消息者(在这个例子中就是等待办理业务的人)往往注 册一个回调机制,在所等待的事件被触发时由触发机制(在这里是柜台的人)通过某种机制(在这里是写在小纸条上的号码)找到等待该事件的人.   
而在实际的程序中,同步消息处理就好比简单的read/write操作,它们需要等待这两个操作成功才能返回;而异步处理机制就是类似于select /poll之类的多路复用IO操作,当所关注的消息被触发时,由消息触发机制通知触发对消息的处理.   
其次再来解释一下阻塞和非阻塞,这两个概念与 **程序等待消息(无所谓同步或者异步)时的状态** 有关.   
继续上面的那个例子,不论是排队还是使用号码等待通知,如果在这个等待的过程中,等待者除了等待消息之外不能做其它的事情,那么该机制就是阻塞的,表现在 程序中,也就是该程序一直阻塞在该函数调用处不能继续往下执行.相反,有的人喜欢在银行办理这些业务的时候一边打打电话发发短信一边等待,这样的状态就是 非阻塞的,因为他(等待者)没有阻塞在这个消息通知上,而是一边做自己的事情一边等待.但是需要注意了,第一种同步非阻塞形式实际上是效率低下的,想象一 下你一边打着电话一边还需要抬头看到底队伍排到你了没有,如果把打电话和观察排队的位置看成是程序的两个操作的话,这个程序需要在这两种不同的行为之间来 回的切换,效率可想而知是低下的;而后者,异步非阻塞形式却没有这样的问题,因为打电话是你(等待者)的事情,而通知你则是柜台(消息触发机制)的事情, 程序没有在两种不同的操作中来回切换.   
很多人会把同步和阻塞混淆,我想是因为很多时候同步操作会以阻塞的形式表现出来,比如很多人会写阻塞的read/write操作,但是别忘了可以对fd设 置O\_NONBLOCK标志位,这样就可以将同步操作变成非阻塞的了;同样的,很多人也会把异步和非阻塞混淆,因为异步操作一般都不会在真正的IO操作处 被阻塞,比如如果用select函数,当select返回可读时再去read一般都不会被阻塞,就好比当你的号码排到时一般都是在你之前已经没有人了,所 以你再去柜台办理业务就不会被阻塞.   
----------------------------------------- 分割线 ------------------------------------------------------   
昨晚写完这篇文章之后,今早来看了看反馈,同时再自己阅读了几遍,发现还是有一些地方解释的不够清楚,在这里继续补充完善一下我的说法,但愿没有越说越糊涂.   
同步和异步:上面提到过, **同步和异步仅仅是关于所关注的消息如何通知的机制,而不是处理消息的机制** .也就是说, **同步的情况下,是由处理消息者自己去等待消息是否被触发,而异步的情况下是由触发机制来通知处理消息者** ,所以在异步机制中,处理消息者和触发机制之间就需要一个连接的桥梁,在我们举的例子中这个桥梁就是小纸条上面的号码,而在select/poll等IO 多路复用机制中就是fd,当消息被触发时,触发机制通过fd找到处理该fd的处理函数.   
请注意理解 **消息通知和处理消息** 这两个概念,这是理解这个问题的关键所在.还是回到上面的例子,轮到你办理业务这个就是你关注的消息,而去办理业务就是对这个消息的处理,两者是有区别 的.而在真实的IO操作时,所关注的消息就是该fd是否可读写,而对消息的处理就是对这个fd进行读写.同步/异步仅仅关注的是如何通知消息,它们对如何 处理消息并不关心,好比说,银行的人仅仅通知你轮到你办理业务了,而如何办理业务他们是不知道的.   
而很多人之所以把同步和阻塞混淆,我想也是因为没有区分这两个概念,比如阻塞的read/write操作中,其实是把消息通知和处理消息结合在了一起,在 这里所关注的消息就是fd是否可读/写,而处理消息则是对fd读/写.当我们将这个fd设置为非阻塞的时候,read/write操作就不会在等待消息通 知这里阻塞,如果fd不可读/写则操作立即返回.   
很多人又会问了,异步操作不会是阻塞的吧?已经通知了有消息可以处理了就一定不是阻塞的了吧?   
其实 **异步操作是可以被阻塞住的,只不过通常不是在处理消息时阻塞,而是在等待消息被触发时被阻塞** .比如select函数,假如传入的最后一个timeout参数为NULL,那么如果所关注的事件没有一个被触发,程序就会一直阻塞在这个select调 用处.而如果使用异步非阻塞的情况,比如aio\_\*组的操作,当我发起一个aio\_read操作时,函数会马上返回不会被阻塞,当所关注的事件被触发时会 调用之前注册的回调函数进行处理,具体可以参见我上面的连接给出的那篇文章.回到上面的例子中,如果在银行等待办理业务的人采用的是异步的方式去等待消息 被触发,也就是领了一张小纸条,假如在这段时间里他不能离开银行做其它的事情,那么很显然,这个人被阻塞在了这个等待的操作上面;但是呢,这个人突然发觉 自己烟瘾犯了,需要出去抽根烟,于是他告诉大堂经理说,排到我这个号码的时候麻烦到外面通知我一下(注册一个回调函数),那么他就没有被阻塞在这个等待的 操作上面,自然这个就是异步+非阻塞的方式了.

## Invoke beginivoke

（1）普通方法调用（直接调用）与Invoke（）方法调用方法 使用的线程Id是一样的 即属于同步。

（2）BeginInvoke(<输入和输出变量>，AsyncCallback callback,object asyncState)方法调用方法 则是启用了新的线程Id       属于异步

可通过 Thread.CurrentThread.ManagedThreadId    获取当前线程的Id

\*委托类型的BeginInvoke(<输入和输出变量>，AsyncCallback callback,object asyncState)方法

异步调用的核心， BeginInvoke（多线程能执行的原因）。

输入和输出变量：表式委托对应的实参。

第二个参数（AsyncCallback callback）：回调函数，表示异步调用后自动调用的函数 ,共用一个线程id.  AsyncCallback是一个委托   有一个参数（asyncState ）

第三个参数（object asyncState）：用于向回调函数提供参数信息。返回值：IasyncResult:异步操作状态接口，封装了异步执行中的参数。

EndInvoke监视BeginInvoke。委托类型的EndInvoke（）方法：借助IasyncResult接口对象，不断查询异步调用是否结束。该方法知道异步调用的方法所有参数，所以，异步调用完毕后，取出异步调用的结果作为返回值。

对于同步和异步的总结：**异步三大特点**

（1）同步方法会出现“假死现象”即卡住界面，异步则不会 原因：异步启动了子线程执行任务，主线程得到释放

（2）同步速度慢，异步速度快  原因：异步启动了多个线程执行任务，占用更多的资源（异步时cpu瞬间上升）

（3）异步是无序的 原因：线程的启动和执行是由操作系统决定的，是无序的。可能每个子线程里面耗时不一样

http://www.cnblogs.com/durow/p/4826653.html

## 线程的信号机制

有时候你需要一个线程在接收到某个信号时，才开始执行，否则处于等待状态，这是一种基于信号的事件机制。.NET框架提供一个ManualResetEvent类来处理这类事件，它的 WaiOne 实例方法可使当前线程一直处于等待状态，直到接收到某个信号。它的Set方法用于打开发送信号。下面是一个信号机制的使用示例：

Lamda

C#并行编程高级教程(中文版).pdf

http://www.cnblogs.com/xinhuawei/p/5398193.html#header04

注意 ：因为在C# 中所有的类型都是继承与object类，所以，以前常用的bool 等变量做为线程间的原子变量的思想是错误的。

## 泛型委托

C#委托的介绍(delegate、Action、Func、predicate)

https://www.cnblogs.com/akwwl/p/3232679.html

委托是一个类，它定义了方法的类型，使得可以将方法当作另一个方法的参数来进行传递。事件是一种特殊的委托。

　　1.委托的声明

　　(1). delegate

delegate我们常用到的一种声明

　　 Delegate至少0个参数，至多32个参数，可以无返回值，也可以指定返回值类型。

　　 例：public delegate int MethodtDelegate(int x, int y);表示有两个参数，并返回int型。

　　(2). Action

Action是无返回值的泛型委托。

　　 Action 表示无参，无返回值的委托

　　 Action<int,string> 表示有传入参数int,string无返回值的委托

　　Action<int,string,bool> 表示有传入参数int,string,bool无返回值的委托

Action<int,int,int,int> 表示有传入4个int型参数，无返回值的委托

　　 Action至少0个参数，至多16个参数，无返回值。

　　 例：

public void Test<T>(Action<T> action,T p)

{

action(p);

}

　 (3). Func

　　 Func是有返回值的泛型委托

　　 Func<int> 表示无参，返回值为int的委托

　　 Func<object,string,int> 表示传入参数为object, string 返回值为int的委托

　　 Func<object,string,int> 表示传入参数为object, string 返回值为int的委托

　　 Func<T1,T2,,T3,int> 表示传入参数为T1,T2,,T3(泛型)返回值为int的委托

　　 Func至少0个参数，至多16个参数，根据返回值泛型返回。必须有返回值，不可void

例：

public int Test<T1,T2>(Func<T1,T2,int>func,T1 a,T2 b)

{

return func(a, b);

}

(4) .predicate

　　 predicate 是返回bool型的泛型委托

　　 predicate<int> 表示传入参数为int 返回bool的委托

　　 Predicate有且只有一个参数，返回值固定为bool

　　 例：public delegate bool Predicate<T> (T obj)

　　2.委托的使用

　　(1).Delegate的使用

复制代码

public delegate int MethodDelegate(int x, int y);

private static MethodDelegate method;

static void Main(string[] args)

{

method = new MethodDelegate(Add);

Console.WriteLine(method(10,20));

Console.ReadKey();

}

private static int Add(int x, int y)

{

return x + y;

}

复制代码

　　(2).Action的使用

复制代码

static void Main(string[] args)

{

Test<string>(Action,"Hello World!");

Test<int>(Action, 1000);

Test<string>(p => { Console.WriteLine("{0}", p); }, "Hello World");//使用Lambda表达式定义委托

Console.ReadKey();

}

public static void Test<T>(Action<T> action, T p)

{

action(p);

}

private static void Action(string s)

{

Console.WriteLine(s);

}

private static void Action(int s)

{

Console.WriteLine(s);

}

复制代码

　　可以使用 Action<T1, T2, T3, T4> 委托以参数形式传递方法，而不用显式声明自定义的委托。 封装的方法必须与此委托定义的方法签名相对应。 也就是说，封装的方法必须具有四个均通过值传递给它的参数，并且不能返回值。 （在 C# 中，该方法必须返回 void）通常，这种方法用于执行某个操作。

　　(3).Func的使用

复制代码

static void Main(string[] args)

{

Console.WriteLine(Test<int,int>(Fun,100,200));

Console.ReadKey();

}

public static int Test<T1, T2>(Func<T1, T2, int> func, T1 a, T2 b)

{

return func(a, b);

}

private static int Fun(int a, int b)

{

return a + b;

}

复制代码

　　(4). predicate的使用

　　泛型委托：表示定义一组条件并确定指定对象是否符合这些条件的方法。此委托由 Array 和 List 类的几种方法使用，用于在集合中搜索元素。

复制代码

static void Main(string[] args)

{

Point[] points = { new Point(100, 200),

new Point(150, 250), new Point(250, 375),

new Point(275, 395), new Point(295, 450) };

Point first = Array.Find(points, ProductGT10);

Console.WriteLine("Found: X = {0}, Y = {1}", first.X, first.Y);

Console.ReadKey();

}

private static bool ProductGT10(Point p)

{

if (p.X \* p.Y > 100000)

{

return true;

}

else

{

return false;

}

}

复制代码

　　使用带有 Array.Find 方法的 Predicate 委托搜索 Point 结构的数组。如果 X 和 Y 字段的乘积大于 100,000，此委托表示的方法 ProductGT10 将返回 true。Find 方法为数组的每个元素调用此委托，在符合测试条件的第一个点处停止。

　　3.委托的清空

　　(1).在类中申明清空委托方法，依次循环去除委托引用。

方法如下：

复制代码

　　　 public MethodDelegate OnDelegate;

public void ClearDelegate()

{

while (this.OnDelegate != null)

{

this.OnDelegate -= this.OnDelegate;

}

}

复制代码

　　(2).如果在类中没有申明清空委托的方法，我们可以利用GetInvocationList查询出委托引用，然后进行去除。

　　方法如下：

复制代码

public MethodDelegate OnDelegate;

　　　　 static void Main(string[] args)

{

Program test = new Program();

if (test.OnDelegate != null)

{

System.Delegate[] dels = test.OnDelegate.GetInvocationList();

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

{

test.OnDelegate -= dels[i] as MethodDelegate;

}

}

}

复制代码

　　4.委托的特点

　　委托类似于 C++ 函数指针，但它们是类型安全的。

　　委托允许将方法作为参数进行传递。

　　委托可用于定义回调方法。

　　委托可以链接在一起；例如，可以对一个事件调用多个方法。

　　方法不必与委托签名完全匹配。

　　5.总结：

　 Delegate至少0个参数，至多32个参数，可以无返回值，也可以指定返回值类型

　　Func可以接受0个至16个传入参数，必须具有返回值

　　Action可以接受0个至16个传入参数，无返回值

Predicate只能接受一个传入参数，返回值为bool类型

## Task.run, task.factory.stratnew

Task.Run 是在 dotnet framework 4.5 之后才可以使用， Task.Factory.StartNew 可以使用比 Task.Run 更多的参数，可以做到更多的定制。

可以认为 Task.Run 是简化的 Task.Factory.StartNew 的使用，除了需要指定一个线程是长时间占用的，否则就使用 Task.Run

### 创建新线程

下面来告诉大家使用两个函数创建新的线程

Task.Run(() =>

{

var foo = 2;

});

这时 foo 的创建就在另一个线程，需要知道 Task.Run 用的是线程池，也就是不是调用这个函数就会一定创建一个新的线程，但是会在另一个线程运行。

Task.Factory.StartNew(() =>

{

ar foo = 2;

});

可以看到，两个方法实际上是没有差别，但是Task.Run比较好看，所以推荐使用Task.Run。

### 等待线程

创建的线程，如果需要等待线程执行完成在继续，那么可以使用 await 等待

复制代码

private static async void SeenereKousa()

{

Console.WriteLine("开始 线程"+Thread.CurrentThread.ManagedThreadId);

await Task.Run(() =>

{

Console.WriteLine("进入 线程" + Thread.CurrentThread.ManagedThreadId);

});

Console.WriteLine("退出 线程"+Thread.CurrentThread.ManagedThreadId);

}

复制代码

但是需要说的是这里使用 await 主要是给函数调用的外面使用，上面代码在函数里面使用 await 函数是 void 那么和把代码放在 task 里面是相同

复制代码

private static async void SeenereKousa()

{

Console.WriteLine("开始 线程"+Thread.CurrentThread.ManagedThreadId);

await Task.Run(() =>

{

Console.WriteLine("进入 线程" + Thread.CurrentThread.ManagedThreadId);

Console.WriteLine("退出 线程"+Thread.CurrentThread.ManagedThreadId);

});

}

复制代码

但是如果把 void 修改为 Task ，那么等待线程才有用

除了使用 await 等待，还可以使用 WaitAll 等待

复制代码

Console.WriteLine("开始 线程" + Thread.CurrentThread.ManagedThreadId);

var t = Task.Run(() =>

{

Console.WriteLine("进入 线程" + Thread.CurrentThread.ManagedThreadId);

});

Task.WaitAll(t);

Console.WriteLine("退出 线程" + Thread.CurrentThread.ManagedThreadId);

复制代码

使用 WaitAll 是在调用 WaitAll 的线程等待，也就是先在线程 1 运行，然后异步到 线程2 运行，这时线程1 等待线程2运行完成再继续，所以输出

1. 开始 线程1
2. 进入 线程2
3. 退出 线程1

### 长时间运行

两个函数最大的不同在于 Task.Factory.StartNew 可以设置线程是长时间运行，这时线程池就不会等待这个线程回收

复制代码

Task.Factory.StartNew(() =>

{

for (int i = 0; i < 100; i++)

{

var foo = 2;

}

Console.WriteLine("进行 线程" + Thread.CurrentThread.ManagedThreadId);

}, TaskCreationOptions.LongRunning);

复制代码

**所以在需要设置线程是长时间运行的才需要使用 Task.Factory.StartNew 不然就使用 Task.Run**

调用 Task.Run(foo) 就和使用下面代码一样

Task.Factory.StartNew(foo, CancellationToken.None, TaskCreationOptions.DenyChildAttach, TaskScheduler.Default);

## Thread带参数和使用

1.ThreadStart

线程执行带参数的方法，new Thread(new ThreadStart(delegate { ThreadTask(firstPage, lastPage); }));

2.System.Threading.ParameterizedThreadStart  
其实没有必要，ThreadStart本来就是个委托，里面又来了个匿名方法。多了一次方法的调用，虽然性能不会影响什么，但看着很别扭。c#有一个委托System.Threading.ParameterizedThreadStart，就是来做这事的，从名字上也可又看出来是带参数的.

下面是原型：

// 表示在 System.Threading.Thread 上执行的方法。  
// 参数: obj: 包含该线程过程的数据的对象。  
[ComVisible(false)]  
public delegate void ParameterizedThreadStart(object obj);

下面是代码实例：

static void Test(object arg)  
  
{  
  
int n = (int)arg;//如果是数组等，可又做相应转换  
  
long r = 0;  
  
for (int i = 0; i < n; i++)  
  
{  
  
r += i;  
  
}  
  
Console.WriteLine(r);  
  
}  
  
static void Main(string[] args)  
  
{  
int n = 1000 \* 10000;  
Thread t = new Thread(new ParameterizedThreadStart(Test));  
t.Start(n);//多参数可又组成数组传递进去。  
}

## 线程开销

采用c\_timer封装形式

测试次数: 0发送指令统计：1 耗时: 25.93ms [0:0:0:25], 速率: 38.6 指令/s count 36

浪费在36-26=10；

直接启异步 thread

测试次数: 0 发送指令统计： 1 耗时: 25.93ms [0:0:0:25], 速率: 38.6 指令/s count 34

浪费在34-26=8；

Task.Factory.StartNew(cmdProce.SendCommand);的形式

测试次数: 0 发送指令统计： 1 耗时: 27.93ms [0:0:0:27], 速率: 35.8 指令/s count 29

消耗 29-28=1；

结论： C#采用task的形式，要比thread节省开销，同样节省5ms时间。

对于单个任务开销比较大，线程创建和使用时间长

## 线程池

### 多个并发的线程，采用线程池的方式。

<http://blog.csdn.net/kavensu/article/details/8093756>

### 同种线程，大并发情况。

# **集合，容器，数组操作**

## [**C# List<T>排序总结**](https://www.cnblogs.com/mq0036/p/11386984.html)

### **对基础类型排序**

#### **Sort,** Reverse**,orderby,** OrderByDescending

#### **基本方法**

调用sort方法，如果需要降序，进行反转：

List<int> list = new List<int>();

list.Sort();// 升序排序

list.Reverse();// 反转顺序

#### lambda****：****

使用lambda表达式，在前面加个负号就是降序了

List<int> list= new List<int>(){5,1,22,11,4};

list.Sort((x, y) => x.CompareTo(y));//升序

list.Sort((x, y) => -x.CompareTo(y));//降序

### **接下来是对非基本类型排序**

#### **无法直接采用list.Sort，会给出元素无法排序的异常。**

首先写一个类用于排序，里面有两个属性，一个构造方法，重写了ToString方法：

class People

{

public People(int id, string name)

{

this.Id = id;

this.Name = name;

}

public int Id { get; set; }

public string Name { get; set; }

//重写ToString

public override string ToString()

{

return "ID:" + Id + " Name:" + Name;

}

}

然后添加一些随机数据，仍希望用Sort排序

List<People> list = new List<People>();

Random r = new Random();

//添加数据

for(int i = 0; i < 10; i++)

{

int j = r.Next(0, 10);

list.Add(new People(j, "name" + j));

}

Console.WriteLine("排序前：");

foreach(var p in list)

{

Console.WriteLine(p);

}

list.Sort();//排序

Console.WriteLine("排序后：");

foreach (var p in list)

{

Console.WriteLine(p);

}

### **实现IComparable接口**

class People: IComparable<People>

{

public People(int id, string name)

{

this.Id = id;

this.Name = name;

}

public int Id { get; set; }

public string Name { get; set; }

//重写的CompareTo方法，根据Id排序

public int CompareTo(People other)

{

if (null == other)

{

return 1;//空值比较大，返回1

}

//return this.Id.CompareTo(other.Id);//升序

return other.Id.CompareTo(this.Id);//降序

}

//重写ToString

public override string ToString()

{

return "ID:" + Id + " Name:" + Name;

}

}

### **实现IComparer接口**

我们首先来看看这个接口：

public interface IComparer<in T>

{

// Parameters:

// x:

// The first object to compare.

//

// y:

// The second object to compare.

//

// Returns:

// A signed integer that indicates the relative values of x and y, as shown in the

// following table.Value Meaning Less than zerox is less than y.Zerox equals y.Greater

// than zerox is greater than y.

int Compare(T x, T y);

}

**重点就看返回值，小于0代表x < y,等于0代表x=y，大于0代表x > y.**

class People:IComparer<People>

{

public People(int id, string name)

{

this.Id = id;

this.Name = name;

}

public int Id { get; set; }

public string Name { get; set; }

//Compare函数

public int Compare(People x, People y)

{

return x.Id.CompareTo(y.Id);//升序

}

//重写ToString

public override string ToString()

{

return "ID:" + Id + " Name:" + Name;

}

}

但是还没完，我们其实是用了**第2点**说的第一个重载方法，所以List还需要参数：

IComparer<People> comparer = new People();

list.Sort(comparer);

### **lamda更简单的**

虽然想实现排序上面的接口代码也不多，但有时候只是偶尔排序，并不像修改类，怎么办呢？当然有更简单的方法，委托和lambda表达式：

所以就有了下面的代码，不需要修改类，只需要用委托构造重载而已：

list.Sort(

delegate(People p1,People p2)

{

return p1.Id.CompareTo(p2.Id);//升序

}

);

当然，lambda表达式实现更简单：

list.Sort((x,y)=> { return x.Id.CompareTo(y.Id); })

### **OrderBy方法**

此方法不会改变原始对象中的数据，而是会返回一个新的IOrderedEnumerable<T>对象，可以将排序好的list再赋给原来的list，也可以给其他的对象。

list = list.OrderBy(o => o.Id).ToList();//升序

var listTmp = list.OrderByDescending(o => o.Id).ToList();//降序

### **多权重排序**

先比较id，再比较Name, 或者按照其他的比较规则，进行综合比较，最后返回 int 值，进行判断。

**重点就看返回值，进行两个物进行比较判断，从而让整个集合有序.**

public int CompareTo(People other)

{

if (null == other)

{

return 1;//空值比较大，返回1

}

//等于返回0

int re = this.Id.CompareTo(other.Id);

if (0 == re)

{

//id相同再比较Name

return this.Name.CompareTo(other.Name);

}

return re;

}

IComparer和delegate还有lambda里可以这样：

public int Compare(People x, People y)

{

int re = x.Id.CompareTo(y.Id);

if (0 == re)

{

return x.Name.CompareTo(y.Name);

}

return re;

}

OrderBy方法有点不同：

list = list.OrderBy(o => o.Id).ThenBy(o=>o.Name).ToList();

list = list.OrderByDescending(o => o.Id).ThenByDescending(o=>o.Name).ToList();//降序

。

## C# LIST查找

### find

**public T Find(Predicate<T> match)**

搜索与制定谓词所定义的条件相匹配的元素，并返回整个List中的第一个匹配元素

判断返回真假的方法，进行过滤集合中的每一个元素

定义user:

public class User

{

public int UserID;

public string UserName;

public User(int i, string n)

{

UserID = i;

UserName = n;

}

}

查找方法

List<User> userCollection = new List<User>();

userCollection.Add(new User(1, "testOne"));

userCollection.Add(new User(2, "testTwo"));

userCollection.Add(new User(3, "testThree"));

User resultUser = userCollection.Find(

delegate(User user)

{

return user.UserID == 1 && user.UserName.Equals("testOne");

});

User resultUser2 = userCollection.Find((User user) =>user.UserID == 1 && user.UserName.Equals("testOne"));

### FindAll, findlast, findindex, findlastindex,

List<university> result=list.FindAll(t=>t.Place.Equals(qq));

## Linq及其介绍

http://download.csdn.net/download/chehaojie/4659459

### [C#编程 LINQ查询](https://www.cnblogs.com/lmx282110xxx/p/10798682.html)

https://www.cnblogs.com/lmx282110xxx/p/10798682.html

## 不同数组间的copyP

### Marshal.Copy，存在数组到IntPtr，IntPtr到数组的2次拷贝

public static T2[] Arr2Arr<T1, T2>(T1[] from)

where T1: struct

where T2 :struct

{

int byteNum = from.Length \* from[0].Sizeof();

T2 testByte = new T2();

dynamic dFrom = from;

dynamic dTo = new T2[byteNum / testByte.Sizeof()];

IntPtr ptr = Marshal.AllocHGlobal(byteNum);

Marshal.Copy(dFrom, 0, ptr, from.Length);

Marshal.Copy(ptr, dTo, 0, dTo.Length);

return dTo;

}

### unsafe的方法，通过指针获得IntPtr，减少了一次复制

public unsafe static T2[] Arr2Arr<T1, T2>(T1[] from, void \* pFrom)

where T1 : struct

where T2 : struct

{

int byteNum = from.Length \* from[0].Sizeof();

T2 testByte = new T2();

dynamic dTo = new T2[byteNum / testByte.Sizeof()];

IntPtr ptr = new IntPtr(pFrom);

Marshal.Copy(ptr, dTo, 0, dTo.Length);

return dTo;

}

### Array.Copy的方法，原生的数组复制方法

public static T2[] Arr2Arr3<T1, T2>(T1[] from)

where T1 : struct

where T2 : struct

{

int byteNum = from.Length \* from[0].Sizeof();

T2 testByte = new T2();

T2[] dTo = new T2[byteNum / testByte.Sizeof()];

Array.Copy(from, dTo, dTo.Length);

return dTo;

}

### 通过Buffer.BlockCopy拷贝数组，速度最快

public static T2[] Arr2Arr4<T1, T2>(T1[] from)

where T1 : struct

where T2 : struct

{

int byteNum = from.Length \* from[0].Sizeof();

T2 testByte = new T2();

T2[] dTo = new T2[byteNum / testByte.Sizeof()];

Buffer.BlockCopy(from, 0, dTo, 0, byteNum);

return dTo;

}

## dictionary的升序降序排列，list dictionary

https://blog.csdn.net/weixin\_39885282/article/details/79501815

### Dictionary排序

Dictionary<int ,int> test=new Dictionary<int, int>();

test.Add(1,1);

test.Add(2,3);

test.Add(3,-1);

var dicNew = test.OrderBy(s1 => s1.Value).ToDictionary(o => o.Key, p => p.Value);;

var dicNewtt= test.OrderByDescending(s1 => s1.Value).ToDictionary(o => o.Key, p => p.Value);

### dictionary, list 间转化 ,array

// 声明Dictionary并初始化

Dictionary<string, string> dic = new Dictionary<string, string>() {

{"1", "one"},

{"2", "two"},

{"3", "three"}

};

// 获得KeyList

List<string> keyList = dic.Keys.ToList();

// 获得ValueList

List<string> valueList = dic.Values.ToList();

// 获得KeyValuePairList

List<KeyValuePair<string, string>> kvList = dic.ToList();

// KeyValuePairList -> Dictionary

Dictionary<string, string> dic2 = kvList.ToDictionary(item => item.Key, item => item.Value);

// 在KeyValuePairList中添加一个重复Key，转换为Dictionary时异常

kvList.Add(new KeyValuePair<string, string>("1", "1"));

Dictionary<string, string> dic3 = kvList.ToDictionary(item => item.Key, item => item.Value); // 发生异常

## Array 数组

Array 类是 C# 中所有数组的基类，它是在 System 命名空间中定义。Array 类提供了各种用于数组的属性和方法,可看作扩充了功能的数组(但不等同数组)，可以使用Array类的属性来对数组进行各种操作。

## Marshal 类

https://msdn.microsoft.com/zh-cn/library/system.runtime.interopservices.marshal(VS.80).aspx  
.NET Framework 2.0 其他版本   
提供了一个方法集，这些方法用于分配非托管内存、复制非托管内存块、将托管类型转换为非托管类型

## 术语

Predicate 谓语部分;谓语;谓词;断言

# **问题排查**

## 直接挂掉

查看windows系统，应用软件日志。按照时间点查找问题的发生点。

## CLR20R3

你的应用程序出现了异常，而.net没有或者无法捕捉该异常，所以导致出现错误clr20r3情况，

实例问题：

发生时间202017 1.03

Problem signature:

Problem Event Name: CLR20r3

Problem Signature 01: EURO.ECLIBHandler.exe

Problem Signature 02: 1.0.12.856

Problem Signature 03: 5f0faeab

Problem Signature 04: PresentationCore

Problem Signature 05: 4.6.1055.0

Problem Signature 06: 563c1e85

Problem Signature 07: 548c

Problem Signature 08: 0

Problem Signature 09: System.OutOfMemoryException

OS Version: 6.1.7601.2.1.0.256.48

Locale ID: 2052

Additional Information 1: 947c

Additional Information 2: 947c43737277b564d0d60899b2dbd370

Additional Information 3: 30ae

Additional Information 4: 30aecc81042385e9be11471ad463a2d7

Read our privacy statement online:

http://go.microsoft.com/fwlink/?linkid=104288&clcid=0x0409

If the online privacy statement is not available, please read our privacy statement offline:

C:\Windows\system32\en-US\erofflps.txt

NET RunTime

Application: EURO.ECLIBHandler.exe

Framework Version: v4.0.30319

Description: The process was terminated due to an unhandled exception.

Exception Info: System.OutOfMemoryException

at System.Windows.Media.Composition.DUCE+Channel.BeginCommand(Byte\*, Int32, Int32)

at System.Windows.Media.GlyphRun.CreateOnChannel(Channel)

at System.Windows.Media.GlyphRun.System.Windows.Media.Composition.DUCE.IResource.AddRefOnChannel(Channel)

at System.Windows.Media.RenderData.System.Windows.Media.Composition.DUCE.IResource.AddRefOnChannel(Channel)

at System.Windows.Media.DrawingVisual.RenderContent(System.Windows.Media.RenderContext, Boolean)

at System.Windows.Media.Visual.RenderRecursive(System.Windows.Media.RenderContext)

at System.Windows.Media.Visual.UpdateChildren(System.Windows.Media.RenderContext, ResourceHandle)

at System.Windows.Media.Visual.RenderRecursive(System.Windows.Media.RenderContext)

at System.Windows.Media.Visual.UpdateChildren(System.Windows.Media.RenderContext, ResourceHandle)

at System.Windows.Media.Visual.RenderRecursive(System.Windows.Media.RenderContext)

at System.Windows.Media.Visual.UpdateChildren(System.Windows.Media.RenderContext, ResourceHandle)

at System.Windows.Media.Visual.RenderRecursive(System.Windows.Media.RenderContext)

at System.Windows.Media.Visual.UpdateChildren(System.Windows.Media.RenderContext, ResourceHandle)

at System.Windows.Media.Visual.RenderRecursive(System.Windows.Media.RenderContext)

at System.Windows.Media.Visual.UpdateChildren(System.Windows.Media.RenderContext, ResourceHandle)

## 引发类型为“System.OutOfMemoryException”的异常

什么时候会引发System.OutOfMemoryException：操作系统无法满足GC对连续内存块的请求，则会发生System.OutOfMemoryException

可能原因  
1：内存真的不够了，连虚拟内存都用完了。  
2：内存还有，但碎片化严重，无法找到合适的连续内存块。

一些常见的原因  
1、系统里缓存了大量的数据，没有及时释放。应该控制缓存的数据大小，缓存失效的时间。  
2、操作的大数据的文件或者DataTable。应该分块读取。  
3、本地资源泄漏。  
解决方案：  
3.1 应该及时释放本地资源  
3.2 使用IDisposable模式

4、大量的对象被固定，不能被压缩移动，导致内存产生大量碎片。  
解决方案：

4.1 如果固定对象大约在同一时间分配，则每两个对象之间的碎片更小  
4.2 优先初始化，因为较旧的对象位于堆的底部，但大多数可用空间都是在堆顶部生成的。  
4.3 固定对象的时间越短，GC就越容易压缩堆

<https://blog.csdn.net/Iron_Ye/article/details/82913025?utm_medium=distribute.pc_relevant_right.none-task-blog-BlogCommendFromMachineLearnPai2-2.nonecase&depth_1-utm_source=distribute.pc_relevant_right.none-task-blog-BlogCommendFromMachineLearnPai2-2.nonecase>

## WPF UnhandledException

在 WPF 程序中，通常可以通过 Application.DispatcherUnhandledException 或 AppDomain.UnhandledException 事件来处理全局 未处理异常，其中前者是由 WPF 框架提供的，后者是由 .NET Framework 提供的，后者能够捕获更多的未处理异常。对于 Task 中的未处理异常，这两种事件都不会触发，仅能通过 TaskScheduler.UnobservedTaskException 事件来捕获。另外，还有个 AppDomain.FirstChanceException 事件，每个异常都会引发该事件，即使该异常已被 try...catch 处理，此事件不在本文的讨论范围内。

### Application.DispatcherUnhandledException 事件

能够捕获 UI 线程抛出的未处理异常

可通过事件参数 e.Handled = true 来阻止程序崩溃

### AppDomain.UnhandledException 事件

能捕获 所有线程（Task 除外） 抛出的未处理异常

默认情况无法阻止程序崩溃（可通过 legacyUnhandledExceptionPolicy 配置异常策略 ）

### TaskScheduler.UnobservedTaskException

仅能捕获 Task 中抛出的未处理异常

事件的触发有延时，依赖垃圾回收

### 顽固的异常 托管异常

private void ButtonBase\_OnClick(object sender, RoutedEventArgs e)

{

Task.Run(() => Add(1, 2));

}

将调用的函数，放到task里面

### WPF app 捕获异常

/// <summary>

/// App.xaml 的交互逻辑

/// </summary>

public partial class App : Application

{

protected override void OnStartup(StartupEventArgs e)

{

base.OnStartup(e);

//UI线程的异常捕捉

this.DispatcherUnhandledException += App\_DispatcherUnhandledException;

}

private void App\_DispatcherUnhandledException(object sender, System.Windows.Threading.DispatcherUnhandledExceptionEventArgs e)

{

try

{

e.Handled = true;

if (e.Exception.InnerException == null)

{

MessageBox.Show("（1）发生了一个错误！请联系开发人员！" + Environment.NewLine

+ "（2）错误源：" + e.Exception.Source + Environment.NewLine

+ "（3）详细信息：" + e.Exception.Message + Environment.NewLine);

//+ "（4）报错区域：" + e.Exception.StackTrace);

}

else

{

MessageBox.Show("（1）发生了一个错误！请联系开发人员！" + Environment.NewLine

+ "（2）错误源：" + e.Exception.InnerException.Source + Environment.NewLine

+ "（3）错误信息：" + e.Exception.Message + Environment.NewLine

+ "（4）详细信息：" + e.Exception.InnerException.Message + Environment.NewLine

+ "（5）报错区域：" + e.Exception.InnerException.StackTrace);

}

}

catch (Exception e2)

{

//此时程序出现严重异常，将强制结束退出

MessageBox.Show("程序发生致命错误，将终止，请联系运营商！");

}

}

## c#异常处理

##### 系统异常

系统抛出的异常对编程人员来说是透明的，也就是我们不需要关心系统是如何得知出错了的，系统类库一旦出错就会将异常对象抛给我们调用它的方法，因为系统本身并不知道要如何处理这个错误。

##### 用户抛出异常

用户抛出异常通常在自己写类库，定义一套API给别人使用的时候用到，这个时候我们并不知道要如何处理这些逻辑错误，所以就需要交给知道如何处理逻辑错误的方法去处理。

在C#里通过定义了throw关键字来抛出异常对象，这里除了抛出新创建的异常对象，也可以通过将已经catch到的异常重新抛出。如果不知道如何处理这个异常，重新抛出异常是很好的习惯，因为既然已经抛出了异常，那么性能已经损失了，所以也不在乎多这么一丁点来更好的挽回错误的结果。

#### 处理异常

所有的异常必须得到妥善的处理，也就是说你必须处理所有的异常。因为系统出错就抛异常，所以这个是c#骨子里的东西，无法避免，所以，系统里到处都会充斥着try{}catch{}。

但是try{}catch{}本身并不会影响系统的性能，所以在没有异常发生的时候try{}catch{}是不会让你系统变慢的，而一旦发生系统异常，你不处理系统就崩溃掉了，你到底是愿意系统慢一点点然后处理掉这个错误呢还是愿意系统崩溃掉呢？

对于处理异常这一点其实我觉得java那种比较严格的，要求严格声明并处理所有异常的方式比较好，能够强制让你重视起异常这回事来，免得系统崩溃掉才想起自己没处理这个异常。

### 如何处理

如何正确的处理异常？这是个很多c#程序员都没最终搞明白的话题，很多人所了解的也无非就是不要用异常处理当业务逻辑，但是何谓用异常来处理逻辑就不得而知了。这里我来详细说明一下，什么地方要用异常，什么地方不要用。

首先有一个前提，异常机制是一个由底向上的冒泡的过程，所以正常的逻辑是，异常由底层抛出，由高层来处理。

#### 处理异常正确的例子

要编写健壮的引用程序，首先要保证必须处理所有的系统异常，也就是调用.NET类库的方法的时候，这个方法可能抛出的异常

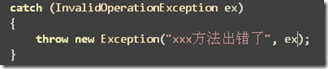
例：



我们可以看到，GetResponse方法会抛出两个异常 InvalidOperationException和WebException。那么我们就需要将调用的代码所在的catch单独处理这两个异常



如果你在这个方法中不知道应该如何处理这个异常，比如这个方法不上不下，和表现层离了好几层，而又需要在表现层通知用户或者由上层业务来决定是通知用户还是悄悄的进村，又或者是悄悄的重新尝试一次，那么自己不能决定的事情就要抛给上层去处理。



有人可能会说.NET可以统一处理异常的，不过我不推荐那种大而全的处理方式，不够细致，很多时候会被奇怪的错误搞得你很追查错误的本源。且比如说遇到网络问题重试就没法处理。

异常必须就近处理，这样才能方便追查，而且注意那个“xxx方法出错了”那个地方，很重要，这个描述可以让你在系统排错的时候少走很多弯路，尽量写详细点。

如果你的代码是给别的程序员使用，而不是和最终用户直接交互，那么除了处理所有系统抛出的异常以外，还需要用异常来验证过滤入口参数。比如，假设你要给其他程序员提供一个将用户对象插入数据库的方法：

public void InsertUser(User user)

{

if(user==null)

{

throw new ArgumentNullException("参数user为null");

}

//调用Orm }

这里我们验证了入口参数，并尽早的抛出了异常，因为这里抛出的异常和不处理等db操作抛出异常，肯定是这里手动抛出的开销更小。这里有一个原则就 是，如果这个参数会造成底层代码直接出错，那么就就近处理它，而不要放任其在底层造成系统异常的抛出。当然还有一个原则就是不要在这里判断业务逻辑，比如 上面的例子就不要在这里验证User的属性的数值是不是合法之类的。

#### 处理异常错误的例子

**1：用异常验证用户输入**

用户输入的合法性验证是属于业务逻辑的一部分，绝对不要用异常去处理，注意，是用户输入，所以这个经验仅限于表现层逻辑

**典型的错误1：**

try

{

int i=int.Parse(textBox1.Text);

}

catch(Exception ex)

{

alert(“不要输入非数字”);

}

**典型错误2：**

void ValidateInput(int i)

{

if(i<0&&i>100)

{

throw new Exception("输入数据范围错误");

}

}

以上两种错误都是错误使用异常的典型，

**2：将异常延迟到底层**

这一点我们在正确的例子里提到过

**典型错误3：**

try

{

string name=Request.QueryString["xx"];

List<User> userls=User.QueryUserByName(name);

}

catch(SqlException ex)

{

}

这个错误在于完全不验证用户输入而直接把数据的验证抛向数据库，等待数据库报错来判断用户输入的正确性，这个是非常致命的错误，很多注入漏洞都是由此产生的。

**3：完全不用异常机制**

产生这个错误肯定是一个非常脑残的决定造成的。不过很多时候某些不了解异常机制的人，由于对异常的性能开销的恐惧感，经常会做出这么脑残的决定。

**典型错误4：**

public bool InsertUser(User user,ref int errcode)

{

if(user==null)

{

errcode=110;//参数为空错误的代码 return false;

}

//调用Orm }

感觉就是一夜回到了解放前，性能倒是高了，但是系统异常怎么办呢？一旦数据库出错就只等着系统崩溃了。某些有经验的说我会把下面的 try{}catch{}起来，不过那不是脱了裤子放屁么，异常都抛出来了，开销已经产生了，结果换来的是牺牲了异常对象的丰富信息而换来了畸形的系统逻 辑。性能也没得到提高。

### 异常对性能的影响

异常机制是C#的特征，因此决定了你不可能逃避，所以讨论异常给你带来了多大开销都是扯淡，没有必要的研究，从根本上无法解决问题。我们应该弄清楚的是，异常的抛出给系统带来什么样的影响，如何在保证系统健壮性的基础上减小不必要的性能消耗。

1.异常的性能开销随着调用栈的深度增加而增大。

对比测试：

测试代码1

Stopwatch sw = new Stopwatch();

sw.Start();

for (int i = 0; i < 10; i++)

{

try

{

throw new Exception("test");

}

catch { }

}

sw.Stop();

MessageBox.Show(sw.ElapsedMilliseconds + "ms");

测试结果：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 调用次数 | 第一次 | 第二次 | 第三次 | 第四次 |
| 时间 | bubuko.com,布布扣 | bubuko.com,布布扣 | bubuko.com,布布扣 | bubuko.com,布布扣 |

测试代码2

Stopwatch sw = new Stopwatch();

sw.Start();

for (int i = 0; i < 10; i++)

{

try

{

System.IO.File.OpenRead("c:\\不存在的txt.txt");

}

catch { }

}

sw.Stop();

MessageBox.Show(sw.ElapsedMilliseconds + "ms");

测试结果

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 调用次数 | 第一次 | 第二次 | 第三次 | 第四次 |
| 时间 | bubuko.com,布布扣 | bubuko.com,布布扣 | bubuko.com,布布扣 | bubuko.com,布布扣 |

小结：由此我们可以发现随着调用栈的深入，性能开销也越大，所以异常应该尽早抛出。

2.对输入的数据应该在业务逻辑中严格检查。

3.**try{}catch{}不会造成任何的系统开销，造成系统开销的是throw 抛出异常**，这是再三强调的了

诸如：

bubuko.com,布布扣

这种言论就纯属脑残了。

### 结论

错误码和异常，这两者在程序的表达能力上是等价的。它们都可以向调用者传达“与常规情况不一样的状态”。考虑使用哪一种，更多地是从API的设计、系统的性能指标、新旧代码的一致性这3个角度来考虑的(本文主要从API设计的角度着手)。

接下来，看一个使用返回错误码的例子：

using namespace std;

int strlen(char \*string) {   
    if (string == NULL) {   
        return -1;   
    }

    int len = 0;

    while(\*string++ != \'\\0\') {   
        len += 1;   
    }

int main(void) {   
    int rc;

    char input[] = {0};   
    rc = strlen(input);   
    if (rc == -1) {   
        cout << \"Error input!\" << endl;   
        return -1;   
    }   
    cout << \"strlen: \" << rc << endl;

    char \*input2 = NULL;   
    rc = strlen(input2);   
    if (rc == -1) {   
        cout << \"Error input!\" << endl;   
        return -2;   
    }   
    cout << \"strlen: \" << rc << endl;

与之等价的使用异常的程序是：

using namespace std;

int strlen(char \*string) {   
    if (string == NULL) {   
        throw \"Invalid input!\";   
    }

    int len = 0;

    while(\*string++ != \'\\0\') {   
        len += 1;   
    }

int main(void) {   
    char input[] = {0};   
    cout << \"strlen: \" << strlen(input) << endl;

char \*input2 = NULL;   
    cout << \"strlen: \" << strlen(input2) << endl;

从以上两个程序片段的对比中，不难看出使用异常的程序更为简洁易懂。为什么？

原因是：返回错误码的方式，使得调用方必须对返回值进行判断，并作相应的处理。这里的处理行为，大部份情况下只是打一下日志，然后返回，如此这 般一直传递到最上层的调用方，由它终止本次的调用行为。这里强调的是，“必须要处理错误码“，否则会有两个问题：1)程序接下来的行为都是基于不确定的状 态，继续往下执行的话就有可能隐藏BUG；2)自下而上传递的过程实际上是语言系统出栈的过程，我们必须在每一层都记下日志以形成日志栈，这样才便于追查 问题。

而采用异常的方式，只管写出常规情况下的逻辑就可以了，一旦出现异常情况，语言系统会接管自下而上传递信息的过程。我们不用在每一层调用都进行 判断处理(不明确处理，语言系统自动向上传播)。最上层的调用方很容易就可以获得本次的调用栈，把该调用栈记录下来就可以了。因此，使用异常能够提供更为 简洁的API。

上述的例子还不是最绝的，因为错误码和常规输出值并没有交集，那最绝的情况是什么呢？错误码侵入了或者说污染了常规输出值的值域了，这时只能通过其它的渠道返回常规输出了。如：

using namespace std;

int get\_avg\_temperature(int day, int \*result) {   
    if (day < 0) {   
        return -1;   
    }

    \*result = day;

int main(void) {   
    int rc;   
    int result;

    rc = get\_avg\_temperature(1, &result);   
    if (rc == -1) {   
        cout << \"Error input!\" << endl;   
        return -1;   
    }   
    cout << \"avg temperature: \" << result << endl;

    rc = get\_avg\_temperature(-1, &result);   
    if (rc == -1) {   
        cout << \"Error input!\" << endl;   
        return -2;   
    }   
    cout << \"avg temperature: \" << result << endl;

当然，如果能忍受低效率，也可以把错误码和常规输出捆到一个结构里再返回，如下：

using namespace std;

typedef struct {   
    int rc;   
    int result;   
} box\_t;

box\_t get\_avg\_temperature(int day) {   
    box\_t b;

    if (day < 0) {   
        b.rc = -1;   
        b.result = 0;   
        return b;   
    }

    b.rc = day;   
    b.result = 0;

int main(void) {   
    box\_t b;

    b = get\_avg\_temperature(1);   
    if (b.rc == -1) {   
        cout << \"Error input!\" << endl;   
        return -1;   
    }   
    cout << \"avg temperature: \" << b.result << endl;

    b = get\_avg\_temperature(-1);   
    if (b.rc == -1) {   
        cout << \"Error input!\" << endl;   
        return -2;   
    }   
    cout << \"avg temperature: \" << b.result << endl;

与之等价的使用异常的程序是：

using namespace std;

int get\_avg\_temperature(int day) {   
    if (day < 0) {   
        throw \"Invalid day!\";   
    }

int main(void) {   
    cout << \"avg temperature: \" << get\_avg\_temperature(1) << endl;   
    cout << \"avg temperature: \" << get\_avg\_temperature(-1) << endl;   
    return 0;   
}

哪一个丑陋，哪一个优雅，我想应该不用我多说了。既然使用异常这么好，那我们是不是干脆全部使用异常算了？当然也不是。以下这个例子也使用了异常，但我们可以看到程序仍然比较冗长：

#include   
#include   
#include

using namespace std;

class database {   
private:   
    map store;   
public:   
    database() {   
        store[\"a\"] = 100;   
        store[\"b\"] = 99;   
        store[\"c\"] = 98;   
    }   
    int get(string key) {   
        map ::iterator iter = store.find(key);   
        if (iter == store.end()) {   
            throw \"No such user exception!\";   
        }   
        return iter->second;   
    }   
};

int main(void) {   
    database db;   
    try {   
        cout << \"score: \" << db.get(\"a\") << endl;   
    } catch (char const \*&e) {   
        cout << e << endl;   
    }   
    try {   
        cout << \"score: \" << db.get(\"d\") << endl;   
    } catch (char const \*&e) {   
        cout << e << endl;   
    }   
    return 0;   
}

与之等价的使用错误码的程序如下：

#include   
#include   
#include

using namespace std;

class database {   
private:   
    map store;   
public:   
    database() {   
        store[\"a\"] = 100;   
        store[\"b\"] = 99;   
        store[\"c\"] = 98;   
    }   
    map ::iterator get(string key) {   
        return store.find(key);   
    }   
    inline map ::iterator not\_exist() {   
        return store.end();   
    }   
};

int main(void) {   
    database db;   
    map ::iterator iter;

    iter = db.get(\"a\");   
    if (iter == db.not\_exist()) {   
        cout << \"no such user!\" << endl;   
    } else {   
        cout << \"score: \" << iter->second << endl;   
    }

    iter = db.get(\"d\");   
    if (iter == db.not\_exist()) {   
        cout << \"no such user!\" << endl;   
    } else {   
        cout << \"score: \" << iter->second << endl;   
    }

在这个例子当中，使用异常并没有带来明显的好处，该做的事情还得做。这种情况下，是选择错误码还是选择异常要结合系统自身的需求和现有代码的情况

使用错误码的例子：   
1、检索数据时，对应某一键不存在相应的记录的情况。这时应使用错误码。这种情况要明确告诉调用方，并不是系统出异常了，而是数据确实不存在，请作出相应的处理。   
2、查找用户时，输入的用户名并不存在的情况。这时应使用错误码。这种情况要明确告诉调用方，并不是系统出异常了，而是用户确实不存在，请作出相应的处理。

使用异常的例子：   
1、读取文件时，文件不存在的情况。(我要读取文件，我认为它是存在的(可能已经进行了相应的判断)，但现在文件不存在了(可能中途被人删除了)，OK，抛出异常，调用方不明确处理的话，系统异常终止)。   
2、修改用户资料时，用户不存在的情况。(我要修改用户资料，我认为用户是存在的(可能已经进行了相应的判断)，但现在用户不存在了(可能中途被人删除了)，OK，抛出异常，调用方不明确处理的话，系统异常终止)。

。

综上所述，是返回错误码还是抛出异常，有以下3条规则：

规则1：本次发生的异常现象是不是真的非常罕见？非常罕见意味着发生该现象的机率非常低，调用方每次都处理将浪费大量的精力，但如果出现，系统应该明确终止，而不是继续往下执行，所以使用异常；其它情况，参考规则2。

规则2：本次发生的异常现象是不是出乎意料之外？出乎意料之外意味着系统已经出问题了(状态和预期不一致)，被调用方没办法再往下执行了，而调用方如果不明确处理的话，系统也应该终止(因为系统状态出问题了)，所以使用异常；其它情况，参考规则3。

规则3：遇到模凌两可的情况，则根据系统的性能需求(异常使用不当可能会造成系统抖动)、现有代码的情况(要保持一致的程序风格)。

# **专题**

## SVN同步

程序集的版本，同SVN上的一致。

步骤：

### 拷贝

Properties目录下赋值拷贝AssemblyInfo.cs



### 修改文件内容

[assembly: AssemblyVersion("2.8.0.$WCREV$")]

[assembly: AssemblyFileVersion("2.8.0.$WCREV$")]

### 属性生成事件，预先生成命令

SubWCRev.exe $(SolutionDir) $(ProjectDir)Properties\AssemblyInfo.template.cs $(ProjectDir)Properties\AssemblyInfo.cs

## 可伸缩性编程

Scalability/extensibility

http://www.xuebuyuan.com/1063103.html

### 可伸缩性（scalability）常见问题

有时由于硬件设备的限制（硬件设备的能力成为性能瓶颈），单纯软件上的性能优化已经无法提升系统性能？

这时需要改造系统的架构体系，提升系统的扩展能力，组合多个只具备低处理能力的硬件设备，从而达到一个具有高处理能力的系统。

当用户数和数据量增加时，软件系统维持高服务质量的能力。例如当业务量小的时候，软件系统运行在一台服务器上，当业务量大的时候，可以通过增加服务器，或增加单台服务器上锁运行的软件系统的个数了来提高其性能，从而达到不需要对软件系统本身进行编程级的修改或变更。

### 可扩展性（extensibility）：

为适应新需求或需求变化为软件增加功能的能力。说得具体一点就是，设计良好的代码允许更多的功能在必要的时候可以被很容易的插入到适当的位置，而不需要修改很多的程序。理论上来说，可扩展性越好，程序员为增加新需求或需求变化所做的处理越容易，做的工作越少，成本也就越低。特别是对于做产品的公司来说，因为自己要维护产品，而且产品一般要不断更新，不断添加新功能，因此可扩展性的设计尤其重要。

可扩展性可以通过软件框架来实现：动态加载的插件（如osgi技术）、顶端有抽象接口的认真设计的类层次结构、有用的回调函数构造以及功能很有逻辑并且可塑性很强的代码结构，面向接口编程，模块时间通过分离关注点，进行低耦合的设计，这些措施都可以实现。

但是任何事物都有两面性，由于考虑可扩展性设计，有时为了满足未来可能的需求变化和需求增加，可能造成软件工程的过度设计开发，从而成本变高，也因此可能导致项目延期，从而错过市场。

### 横向伸缩scale out

1. 定义：

横向伸缩scale out，是指系统向外扩展，也就是通过增加系统的处理节点的方式来提高系统的整体处理能力，特别是提高其性能。具体说就是增加机器来提升系统的性能。

1. scale out的优点

  服务器陈本低，仅仅通过价格便宜的PC Server搭建出一个处理能力非常强大的计算集群。

  基本上不存在用户量或数据量增加就再次遇到系统性能增加的瓶颈，因为理论上来说，通过增加PC Server或服务器主机的个数时没有限制的。

  单个服务器节点出现故障对系统整体的影响也比较小。

1. scale out的缺点：

  因为服务器节点增多，造成系统架构整体复杂度提高，比如会有很多架构方面的问题需要解决，比如session复制的问题等。

  集群维护难度增大，维护成本更高。

### 纵向伸缩scale up

1. 定义：

纵向伸缩scale up，是指向上扩展，也就是通过增加当前服务器节点的处理能力来提升系统的整体处理能力。具体说来，包括通过升级现有服务器的配置，如增加内存，增加CPU,增加存储系统的硬件配置，或者更换更强的服务器和更高端的存储系统。

1. scale up的优点：

  处理的节点少，维护很简单，维护成本低。

  系统集中在一台服务器，架构设计简单，开发容易。

1. scale up的缺点

  整个系统集中在一台服务器，为满足应用系统的需求，对服务器的硬件配置必然很高，而高端设备的服务器价格都很贵，成本自然很高。

  应用系统业务需求量可能无限增加，但是单台服务器的处理能力总是很有限，所以很容易遇到无法解决的性能瓶颈。

  整个系统集中在一台服务器，发生故障后影响是很大的。

### 横向伸缩和纵向伸缩的综合分析

从横向伸缩和纵向伸缩的优缺点来看，scale up更适合短期处理问题，因为这样成本低，架构简单，技术上要求也简单，风险也小，特别是系统上线要求比较急，项目进度要求比较紧的情况下，scale up还是不错的选择。

但是对于一个需要长期运维的系统来说，scale up不能从根本上解决问题（除非业务量和数据量一直不上去，呵呵，这估计是大家都不希望看到的），因为业务量和数据量上去了，迟早会有性能瓶颈出现的，硬件技术的发展是有限的，很多时候都难以赶上业务的发展，如果数据量持续增加，可能隔不了多少时间，又要升级服务器，经常会更换硬件设备。然而scale out在这方面却有优势，服务器成本低，理论上是可无限扩展，当然技术上的复杂度是会不少，当时值得研究，解决起来也问题不大。

# **界面展现技术**

## 控件和窗口的常用方法调用

1. 控件或窗口加入VIEW

如窗口或控件加入view

1. 构造函数，哪里构造，哪里调用， 在C#里，new的时候，就需要调用

构造函数是在实例化的时候。

1. Load函数

load,是窗口或控件要被展现的时候，需要重新调用！！！！！！所以窗口要被重新展现的时候，唯一初始化定义就不能放在Load 里面

## 窗口覆盖最前面的问题

1、一个程序多个窗口，就有窗口覆盖的问题。

一个窗口最大化会遮盖其他窗口

## 常用Ui技术

Canvas +

Grid 行列划分

界面工程库 ，前台引用方式

xmlns:xctk="<http://schemas.xceed.com/wpf/xaml/toolkit>"

### 辅助程序代码插件。

在字段和属性上，打上标记，然后用辅助程序进行生成相应的展现界面和程序代码，将数据交换，展现形式的代码自动完成，这段代码是有规律的。

也类似于代码生成器。

## WPF的绑定技术

https://www.cnblogs.com/zlgcool/archive/2008/10/22/1316605.html

## 样式

<https://www.cnblogs.com/lovezhangyu/p/10229250.html>

样式讲解

<https://www.cnblogs.com/tsliwei/p/6138412.html>

bootstrap样式使用

## 模板

<https://www.cnblogs.com/lonelyxmas/p/10167821.html>

## Datagrid编辑

<DataGrid x:Name="dgTubes" AutoGenerateColumns="False" Margin="10,10,10,81">

<DataGrid.Columns>

<DataGridTextColumn Binding="{Binding name}" Header="name" Width="50"/>

<DataGridTemplateColumn Header="nStatus" Width="\*">

<DataGridTemplateColumn.CellEditingTemplate>

<DataTemplate >

<TextBox Text="{Binding nStatus}" VerticalAlignment="Center"/>

</DataTemplate>

</DataGridTemplateColumn.CellEditingTemplate>

</DataGridTemplateColumn>

</DataGrid.Columns>

</DataGrid>

要点：

1. 采用cell的时候，必须要用<DataGridTemplateColumn.CellEditingTemplate>
2. 实现绑定必须要用公用的观察者模型
3. Model必须实现InotifyPropertyChanged
4. 属性，必须要实现

属性，必须要实现

set

{

\_nStatus = value;

this.RaisePropertyChanged("nStatus");

}

5、在按键的事件处理函数中，进行轮训查询的时候，只有增加线程才能查询回调，否则无法退出轮训查询。

## 布局面板

<https://www.cnblogs.com/weivyuan/articles/2601903.html>

<https://www.cnblogs.com/zh7791/p/9012880.html>

1. Canvas的简要分析

    Canvas是一种比较简单的布局容器，采用绝对定位，精确地控制每个元素的位置。实际情况是Canvas不做任何布局，它只是简单地把元素放到指定的位 置就OK了，而且它默认不会改变元素的大小来填充可用的布局控件，也就意味着其子元素是根据元素的内容来调整尺寸大小。

1. DockPanel的简要分析

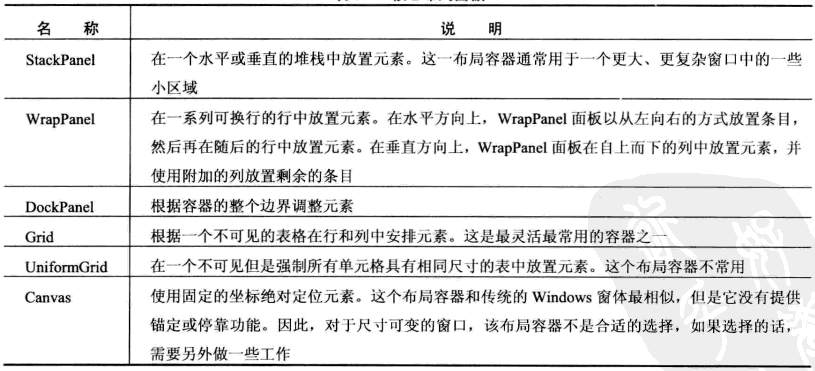
    DockPanel适合于进行简单的整体布局，可以用来划分窗体的基本结构。它总是会对每个元素进行排列，默认情况下，最后一个子元素负责填充剩余的布局控件。

1. StackPanel的简要分析

    StackPanel也是一种简单的布局面板，它将子元素按照行或列进行排列。但是它的功能不足进行整个面板的布局，而是进行一些的模块布局。每个子元素 与面板一样宽或一样高，当子元素超过面板空间，会自动截断。剩余的空间将由元素的HorizontalAlignment和 VerticalAlignment属性决定。

1. WrapPanel的简要分析

    WrapPanel与StackPanel基本没有区别，不同的是，当子元素超过面板宽度的时候，WrapPanel会在下一行进行排列。



## 资源字典

1. 单独封装成资源字典，。

<https://www.cnblogs.com/tsliwei/p/6138412.html>

1. 如果在app中引用，则所有的xmal都会默认应用样式
2. 资源字典在一个窗口或空间内，要么使用资源字典，要么用key

### DynamicResource与StaticResource的区别

动态资源，静态资源

https://blog.csdn.net/johnsuna/article/details/2500723

#### 定义资源字典

定义资源字典文件 buttonTest.xaml

创建style, 设定样式和绑定目标

<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

<!--定义按钮样式·通用按钮 ,如果没有 x:Key="ButtonGeneral"，则所有button都适应-->

<!--<Style TargetType="Button">-->

<Style x:Key="ButtonGeneralbutton" TargetType="Button">

…….

</ResourceDictionary>

#### 使用资源字典

1. 在使用的窗口或控件中，建立资源字典的引用

<Window.Resources>

<ResourceDictionary>

<ResourceDictionary.MergedDictionaries>

<ResourceDictionary Source="pack://application:,,,/EURO.Common.Controls;component/ComRes/DicButton.xaml"/>

</ResourceDictionary.MergedDictionaries>

</ResourceDictionary>

</Window.Resources>

///Source="pack://application:,,,/EURO.Common.Controls;component/ComRes/DicButton.xaml"/>

pack://application:,,,/EURO.Common.Controls; 固定控件名称

component/ComRes/ 组件定义/在控件内的目录

1. //控件使用时，绑定相应控件

<Button Content="Button" Style="{StaticResource ButtonGeneralbutton}" HorizontalAlignment="Left" Margin="245,355,0,0" VerticalAlignment="Top" Width="175" Height="41"/>

1. 前后台绑定资源

<https://blog.csdn.net/lisenyang/article/details/18231195>

1、xmlns

Mapping URI的格式是

clr-namespace:<clr namespace>[;assembly=<assembly name>]

（1）如果自定义类和XAML处在同一个Assembly之中，只还需要提供clr-namespace值。

**[html]** [view plaincopy](http://blog.csdn.net/wushang923/article/details/7382709)

1. xmlns:converter="clr-namespace:Pansoft.SCV.Workflows.OpenAccount.ValueConverter"

（2）如果自定义类和XAML处在不同的Assembly之中。除了clr-namespace值外，还必须提供assembly的值。

**[html]** [view plaincopy](http://blog.csdn.net/wushang923/article/details/7382709)

1. xmlns:converter="clr-namespace:Pansoft.SCV.UIFramework.ValueConverter;assembly=Pansoft.SCV.UIFramework"

clr-namespace和assembly的拼写必须都是小写。

这样在XAML中就可以通过namespace prefix和类名使用自定义的元素了。举例：

**[html]** [view plaincopy](http://blog.csdn.net/wushang923/article/details/7382709)

1. **<converter:ImageSourceConverter** x:Key="ImageConverter"**/>**

2、Resource

前台：

**[html]** [view plaincopy](http://blog.csdn.net/wushang923/article/details/7382709)

1. **<Page.Resources>**
2. **<ResourceDictionary>**
3. **<ResourceDictionary.MergedDictionaries>**
4. **<ResourceDictionary** Source="/Pansoft.SCV.UIFramework;component/Style/GlobaStyle.xaml"**/>**
5. **</ResourceDictionary.MergedDictionaries>**
6. **<converter:ImageSourceConverter** x:Key="ImageConverter"**/>**
7. **<Style** TargetType="{x:Type TextBox}"**>**
8. **</Style>**
9. **</ResourceDictionary>**
10. **</Page.Resources>**

后台：

**[csharp]** [view plaincopy](http://blog.csdn.net/wushang923/article/details/7382709)

1. Resources.MergedDictionaries.Add(**new** ResourceDictionary()
2. {
3. Source = **new** Uri("../Pansoft.SCV.Workflows.OpenAccount;component/Style/GlobaStyle.xaml", UriKind.Relative)
4. });

或者

**[csharp]** [view plaincopy](http://blog.csdn.net/wushang923/article/details/7382709)

1. Resources.MergedDictionaries.Add(**new** ResourceDictionary()
2. {
3. Source = **new** Uri("pack://application:,,,/Pansoft.SCV.Workflows.OpenAccount;component/Style/GlobaStyle.xaml")
4. });

3、Binding

前台：

**[html]** [view plaincopy](http://blog.csdn.net/wushang923/article/details/7382709)

1. **<trans:OpenAccountTranscation** x:Key="WorkflowNode"**/>**

或者

**[csharp]** [view plaincopy](http://blog.csdn.net/wushang923/article/details/7382709)

1. Resources.Add("WorkflowNode", node.Owner.Transcation);

调用：

**[html]** [view plaincopy](http://blog.csdn.net/wushang923/article/details/7382709)

1. **<Page.DataContext>**
2. **<Binding** Source="{StaticResource WorkflowNode}"**/>**
3. **</Page.DataContext>**

**[html]** [view plaincopy](http://blog.csdn.net/wushang923/article/details/7382709)



**[html]** [view plaincopy](http://blog.csdn.net/wushang923/article/details/7382709)

1. **<TextBlock** Text="{Binding Name}"  FontSize="18" Margin="20,0"**/>**

或者直接写：

**[csharp]** [view plaincopy](http://blog.csdn.net/wushang923/article/details/7382709)

1. DataContext = node.Owner.Transcation;

后台：

**[csharp]** [view plaincopy](http://blog.csdn.net/wushang923/article/details/7382709)

1. Binding MyBinding = **new** Binding();
2. MyBinding.Path = **new** PropertyPath("Name");
3. MyBinding.Mode = BindingMode.OneWay;
4. MyBinding.Source = node.Owner.Transcation;
5. MyTextBlock.DataContext = node.Owner.Transcation;
6. MyTextBlock.SetBinding(TextBlock.TextProperty, MyBinding);

4、后台重新绑定

xaml：

**[html]** [view plaincopy](http://blog.csdn.net/wushang923/article/details/7382709)

1. **<Button** x:Name="BtnSwitchLangs" Content="{DynamicResource Execute}" Width="200" Height="60" Click="Button\_Click\_2" Margin="0,5"**/>**

后台（重新绑定）：

**[csharp]** [view plaincopy](http://blog.csdn.net/wushang923/article/details/7382709)

1. BtnSwitchLangs.SetResourceReference(ContentProperty, "ReExecute");//为内容绑定新的源

### WPF 命名空间

https://blog.csdn.net/catshitone/article/details/71213371?locationNum=11&fps=1

wpf中的xmlns命名空间为什么是一个网址，代表了什么意思

## auto规则

采用GRID

* Auto 表示自动适应显示内容的宽度, 如自动适应文本的宽度,文本有多长,控件就显示多长.
* \*则表示按比例来分配宽度.

控件指定了尺寸，优先按照尺寸来。

并且控件需要指定为scretch

按照比例分配 ，第一列占3/10

* <ColumnDefinition Width="3\*" />
* <ColumnDefinition Width="7\*" />

第一行占 3/10

* <RowDefinition Height="3\*" />
* <RowDefinition Height="7\*" />

public MainWindow()

{

InitializeComponent();

var image1 = new System.Windows.Media.Imaging.BitmapImage();

image1.BeginInit();

image1.UriSource = new Uri(@"F:\测试图片\test\IMG10201.jpg", UriKind.Absolute);

image1.EndInit();

Imagebox.Source = image1;

}

## 日志模块

### 封装的日志模块

|  |  |  |
| --- | --- | --- |
| BaseLoggingService | 日志服务基类 |  |
| ConsoleLogService | 控制台输出信息 |  |
| DebugLoggingService | Vs command t调试输出 |  |
| NLogLoggingService | 采用NLOG作为日志文件接口 |  |
| WFileLoggingService | 自定义的文件IO操作形式的日志接口 |  |
| ComLogService | 重定向输出 |  |

LoggingProvider 各层级模块可以定义自己的Loggingprovider,增加模块的各项头信息和说明信息。

GlobalLoggingProvider

全局的日志模块，可以直接使用。

### 创建独立日志

public class HandlerServiceProvider

{

public static LibVersion makeVersion()

{

var libv=new LibVersion

{

AssemblyLanguage = "C Sharp",

AssemblyProduct = "EURO COMMUNICATION LIBRARY ",

AssemblyVersion = "1.0.0.7",

AssemblyCopyright = "Copyrights(C) Euroimmun China"

};

return libv;

}

public static void PrintHead()

{

LibVersion lversion = makeVersion();

log.PrintMessageInfo( "");

log.PrintMessageInfo("================================================================================");

log.PrintMessageInfo( " " +"Name: " + lversion.AssemblyProduct);

log.PrintMessageInfo( " " + lversion.AssemblyCopyright);

log.PrintMessageInfo(" " + "ProtocolVersion: " + lversion.ComVersion);

log.PrintMessageInfo(" " + "Language: " + lversion.AssemblyLanguage);

log.PrintMessageInfo(" " + "ProgramVersion :" + lversion.AssemblyVersion);

log.PrintMessageInfo( " " + "Date: " + lversion.Data );

log.PrintMessageInfo( "================================================================================");

log.PrintMessageInfo( "");

}

private static LoggingProvider log = new LoggingProvider(EnumLogLevel.ALL,"TestCase:");

public static LoggingProvider Log

{

get

{

return log;

}

set { log = value; }

}

public ushort OwnerSSmid { get; set; }

public void PrintComLog(string extLogMessage, EnumMessageType messageType = EnumMessageType.Undefine)

{

if (messageType == ECLIBHandler.Library.Service.EnumMessageType.Undefine)

{

Log.PrintMessageInfo(string.Format("Moudle {0} {1}", OwnerSSmid, extLogMessage));

}

else

{

Log.PrintMessageInfo(string.Format("Moudle {0} {1} {2}", OwnerSSmid, extLogMessage, messageType));

}

}

public void PrintComLog(ushort senderSmid, ushort targetSmid, byte[] data, ECLIBHandler.Library.Service.EnumCharType type = ECLIBHandler.Library.Service.EnumCharType.Decimal, ECLIBHandler.Library.Service.EnumDirect direct = ECLIBHandler.Library.Service.EnumDirect.Send, ECLIBHandler.Library.Service.EnumMessageType messageType = ECLIBHandler.Library.Service.EnumMessageType.Undefine)

{

PrintComLog(LogFormat.GetLogMsg(senderSmid, targetSmid, data, type, direct), messageType);

}

}

public struct LibVersion

{

public string AssemblyProduct;

public string AssemblyVersion;

public string ComVersion;

public string AssemblyLanguage;

public string Data;

public string AssemblyCopyright;

};

### 使用日志服务

HandlerServiceProvider.Log.LoggingService=new NLogLoggingService(); //使用具体日志服务输出对象

HandlerServiceProvider.Log.PrintMessageInfo("this is the first program!");

HandlerServiceProvider.Log.LoggingService.InfoIsEnable = true; //日志等级为info

### 使用第三方接口

HandlerServiceProvider.Log.LoggingService=new ComLogService(){InPrintTextMethod = printmessageMethod }

HandlerServiceProvider.Log.PrintMessageInfo("this is the first program!");

HandlerServiceProvider.Log.LoggingService.InfoIsEnable = true;

Void printmessageMethod(string str)

{

Textbox.text+=str;  
}

### NLOG配置

#### LOG NLOG

<https://github.com/nlog/nlog/wiki/File-target#size-based-file-archival>

**https://www.cnblogs.com/sorex/archive/2013/01/31/2887174.html**

NLOG记录进数据库。

NLOG记录日志超过一定程度后如何处理。

<?xml version="1.0" encoding="utf-8" ?>

<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<targets>

<!--此部分中的所有目标将自动异步-->

<target name="asyncFile" xsi:type="AsyncWrapper">

<!--项目日志保存文件路径说明fileName="${basedir}/保存目录，以年月日的格式创建/${shortdate}/${记录器名称}-${单级记录}-${shortdate}.txt"-->

<target name="log\_file" xsi:type="File"

fileName="${basedir}/ProjectLogs/${shortdate}/${logger}-${level}-${shortdate}.txt"

layout="${longdate} | ${message} ${onexception:${exception:format=message} ${newline} ${stacktrace} ${newline}"

archiveFileName="${basedir}/archives/${logger}-${level}-${shortdate}-{#####}.txt"

archiveAboveSize="102400"

archiveNumbering="Sequence"

concurrentWrites="true"

keepFileOpen="false" />

</target>

<!--使用可自定义的着色将日志消息写入控制台-->

<target name="colorConsole" xsi:type="ColoredConsole" layout="[${date:format=HH\:mm\:ss}]:${message} ${exception:format=message}" />

</targets>

<!--规则配置,final - 最终规则匹配后不处理任何规则-->

<rules>

<logger name="Microsoft.\*" minlevel="Info" writeTo="" final="true" />

<logger name="\*" minlevel="Info" writeTo="asyncFile" />

<logger name="\*" minlevel="Warn" writeTo="colorConsole" />

</rules>

</nlog>

#### rules【规则】说明：

　　规则是logger具有以下属性的元素：  
name - 记录器名称过滤器 - 可能包含通配符（\*和？）  
minlevel - 记录的最低级别  
maxlevel - 记录的最高级别  
level - 单级记录  
levels - 逗号分隔的记录级别列表  
writeTo - 逗号分隔的要写入的目标列表  
final - 最终规则匹配后不处理任何规则  
enabled- 设置为false禁用规则而不删除它  
ruleName- 规则标识符，允许使用Configuration.FindRuleByName和进行规则查找Configuration.RemoveRuleByName。在NLog 4.6.4中引入  
注意：虽然命名了规则logger，但它没有定义记录器。它引用一个或多个记录器。  
通过将规则name模式与记录器名称匹配，将规则映射到记录器。规则name属性可以包括通配符（\*和？），以通过通配符匹配来匹配记录器名称。  
\* - 匹配0个或更多字符  
? - 恰好匹配1个字符

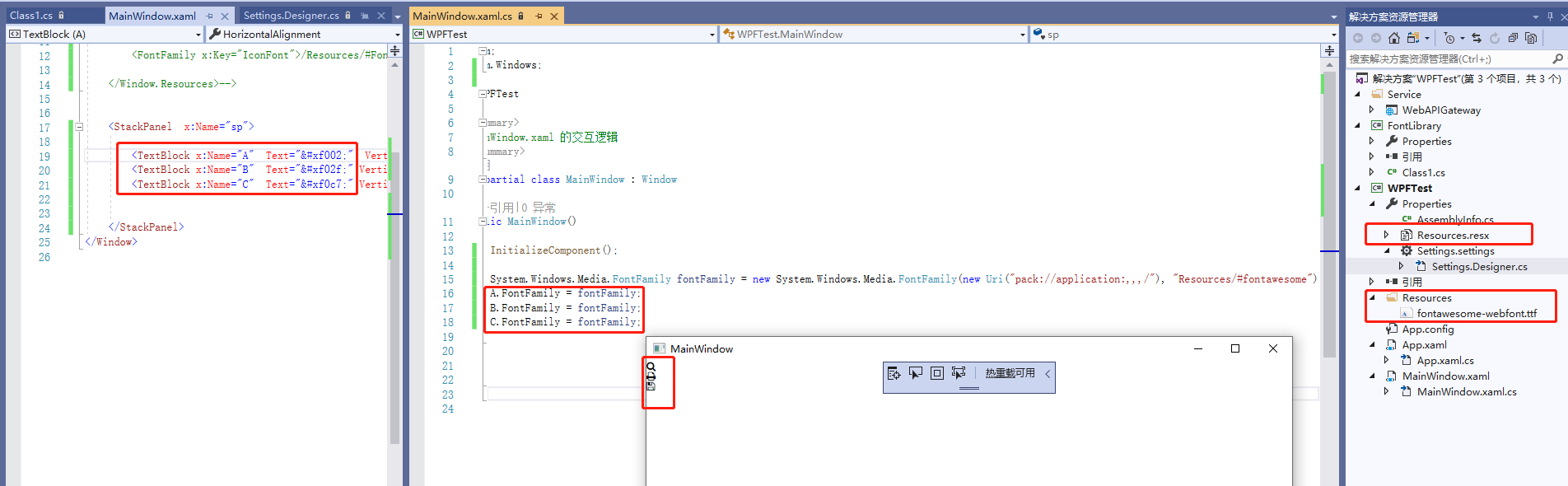
#### targets【目标】：

　　每个目标由一个target元素表示。每个目标需要两个属性：  
name - 目标名称  
type - 目标类型 - 例如“文件”，“数据库”，“邮件”。使用名称空间时，此属性已命名xsi:type。  
除了这些属性之外，目标通常还接受其他参数，这些参数会影响诊断跟踪的写入方式。每个目标都有一组不同的参数，它们在项目主页上有详细描述，并且它们是上下文相关的。Intellisense也可以在Visual Studio中使用。  
例如，File目标接受fileName定义输出文件名的参数，Console目标具有error参数，该参数确定诊断跟踪是否写入标准错误（stderr）而不是进程的标准输出（stdout）。  
此示例演示了一个targets定义多个目标的部分：两个文件，一个网络目标和OutputDebugString目标：

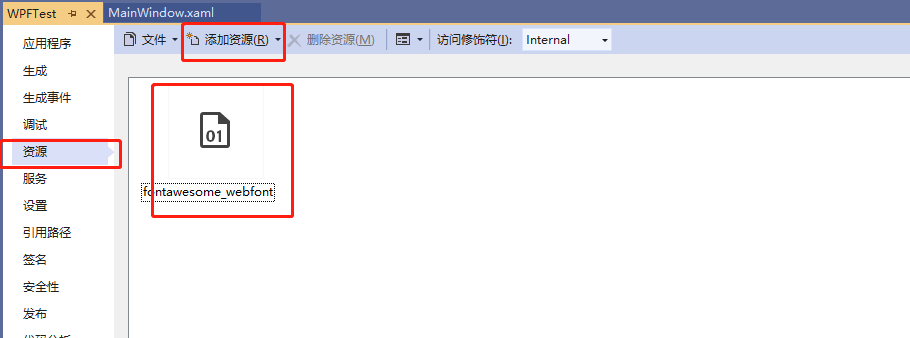
## [WPF 将字体文件 添加到 资源文件，并在后台代码使用](https://www.cnblogs.com/zerommc/p/11607696.html)

<https://www.cnblogs.com/zerommc/p/11607696.html>

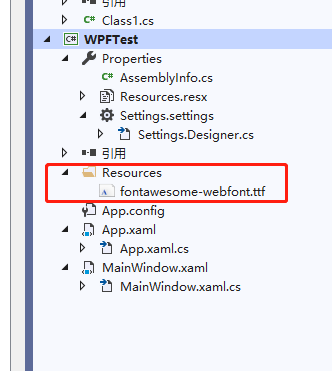
先看结果



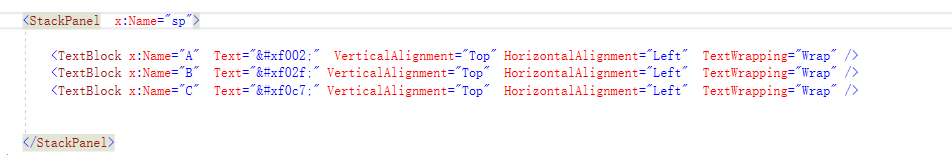
1、将字体文件，导入到资源文件，如：



 添加后，自动生成



2、在窗体xaml中添加如：



 3、在xaml窗体对应的cs文件中，为TextBlock指定字段



 创建一个字段对象，并指定字体文件的所在路径 Resources 指自动生成的资源文件夹，#fontawesome是指自体名称 ，而非字体文件名称，如何查看字体名称：”双击“打开字体文件，如图



## 报表

## 数据库

## 结果展示

## 多语言

## Font 类

第一个是 new Font( 字体名称 , 字号 ) ，例如，label1.Font=new Font("黑体",9)。

第二个是 new Font( 字体名称 , 字号，字体风格 )，其中第三个参数是枚举类型，具体定义如下：

enum FontStyle{  
Regular =0,// 正常字体  
Bold =1,// 黑体  
Italic =2,// 斜体  
BoldItalic =3,// 黑斜体  
Underline =4,// 下划线， 5=黑体下划线， 6=斜体下划线， 7=黑斜体下划线  
Strikeout =8}// 删除线， 9=黑体删除线， 10=斜体删除线，依此类推。

例如修改标签控件字体为斜体：  
label1.Font=new Font("黑体",9,label1.Font.Style|FontStyle.Italic);  
或者： label1.Font=new Font("黑体",9,label1.Font.Style|(FontStyle)2);

修改标签控件字体不为斜体：  
label1.Font=new Font("黑体",9,label1.Font.Style&~FontStyle.Italic);  
或者： label1.Font=new Font(" 黑体",9,label1.Font.Style&(FontStyle)(~2));

# **Vs调试**

## Vshost

宿主进程是VS2005/2008/2010 中的一项功能，能提高调试性能，支持部分信任调试并支持设计时表达式计算。宿主进程文件的文件名中包含 vshost，并位于项目的输出文件夹中。

　　●提高的调试性能

　　宿主进程创建一个应用程序域并将调试器与应用程序关联起来。执行这些任务会导致在开始调试和开始运行应用程序之间有很明显的延迟。通过在后台创建应用程序域和关联调试器，并在运行应用程序之间保存应用程序域和调试器状态，宿主进程使性能得到提高。

　　●部分信任调试

　　在“项目设计器”的安全页中，可将应用程序指定为部分信任应用程序。调试部分信任应用程序需要对应用程序域进行特殊的初始行。此初始化由宿主进程处理。

　　●设计时表达式计算

　　设计时表达式计算使您可以从“即时”窗口进行代码测试，而不必运行应用程序。宿主进程在设计时表达式计算期间执行此代码。

# **软件工程**

## 规范化编程

### XML注释规范

https://www.cnblogs.com/mq0036/p/3544264.html

NET中的规范标准注释(一) -- XML注释标签讲解

http://www.cnblogs.com/mq0036/p/3544268.html

创建帮助文档

?C#?注释（Comment）规范?

注释规范包括：

模块（类）注释规范、类的属性、方法注释规范、代码间注释?

1.?模块（类）注释规范?

模块开始必须以以下形式书写模块注释：?

///<summary>?

///模块编号：<模块编号，可以引用系统设计中的模块编号>???

///作用：<对此类的描述，可以引用系统设计中的描述>???

///作者：作者中文名?

??

///编写日期：<模块创建日期，格式：YYYY-MM-DD>??

?

///</summary>?

如果模块有修改，则每次修改必须添加以下注释：?

??

///<summary>???

///Log

编号：<Log编号,从1开始一次增加>?

??

///修改描述：<对此修改的描述>???

///作者：修改者中文名?

??

///修改日期：<模块修改日期，格式：YYYY-MM-DD>??

?

///</summary>?

?

2.?类属性注释规范?

在类的属性必须以以下格式编写属性注释：?

??

///?<summary>??

///属性说明?

///?</summary>?

3.?方法注释规范?

在类的方法声明前必须以以下格式编写注释?

??

///?<summary>?

?

?

///?说明：<对该方法的说明>?

/// </summary>

/// <param name="<参数名称>"><参数说明></param>

///<returns>

///<对方法返回值的说明，该说明必须明确说明返回的值代表什么含义>

///</returns>

4.代码间注释规范

代码间注释分为单行注释和多行注释：

单行注释：

//<单行注释>

多行注释：

/\*

多行注释

1多行注释

2多行注释

3\*/

代码中遇到语句块时必须添加注释（if,for,foreach,……）,添加的注释必须能够说明此语句块的作用和实现手段（所用算法等等）。

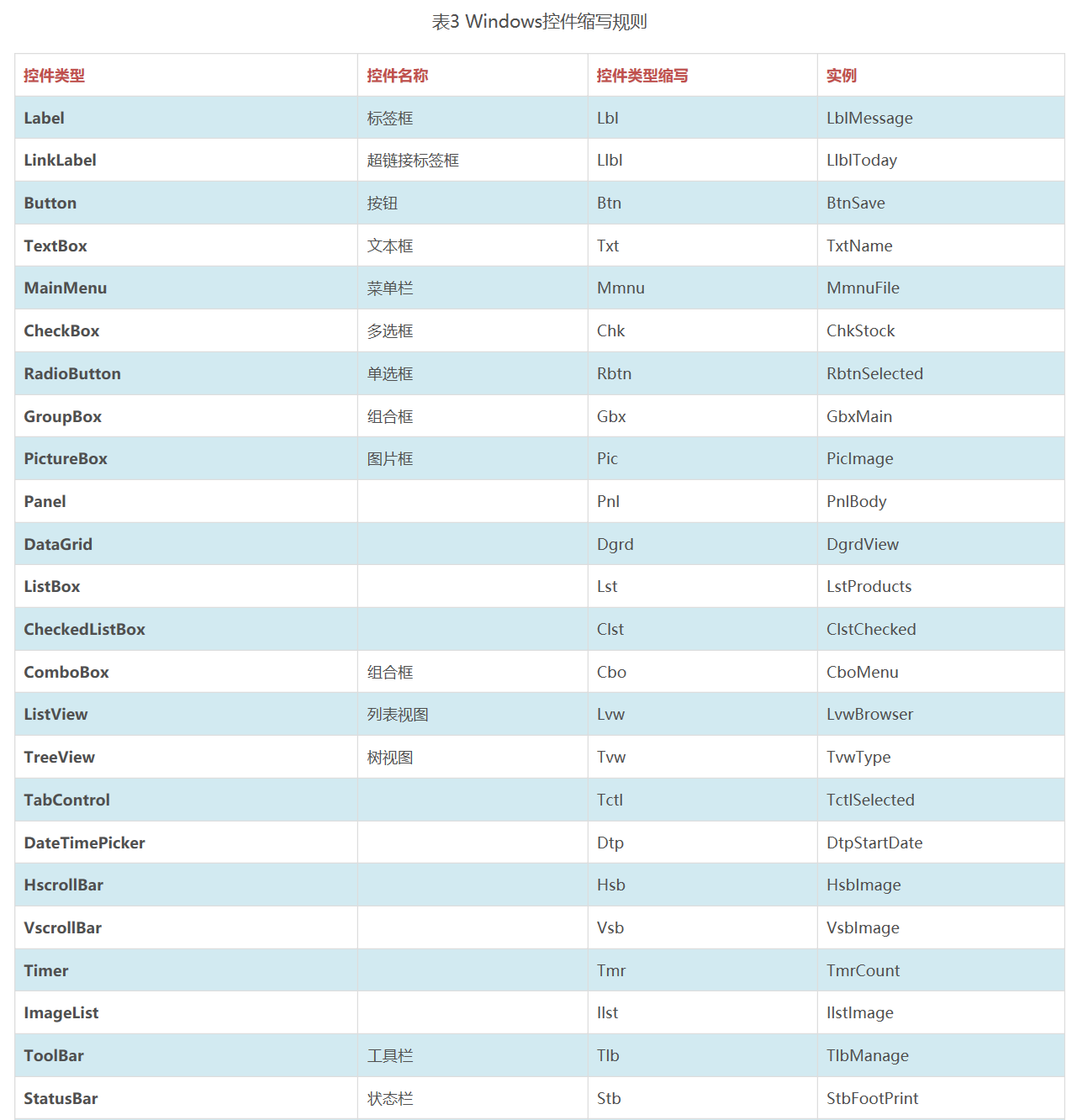
### 命名规范

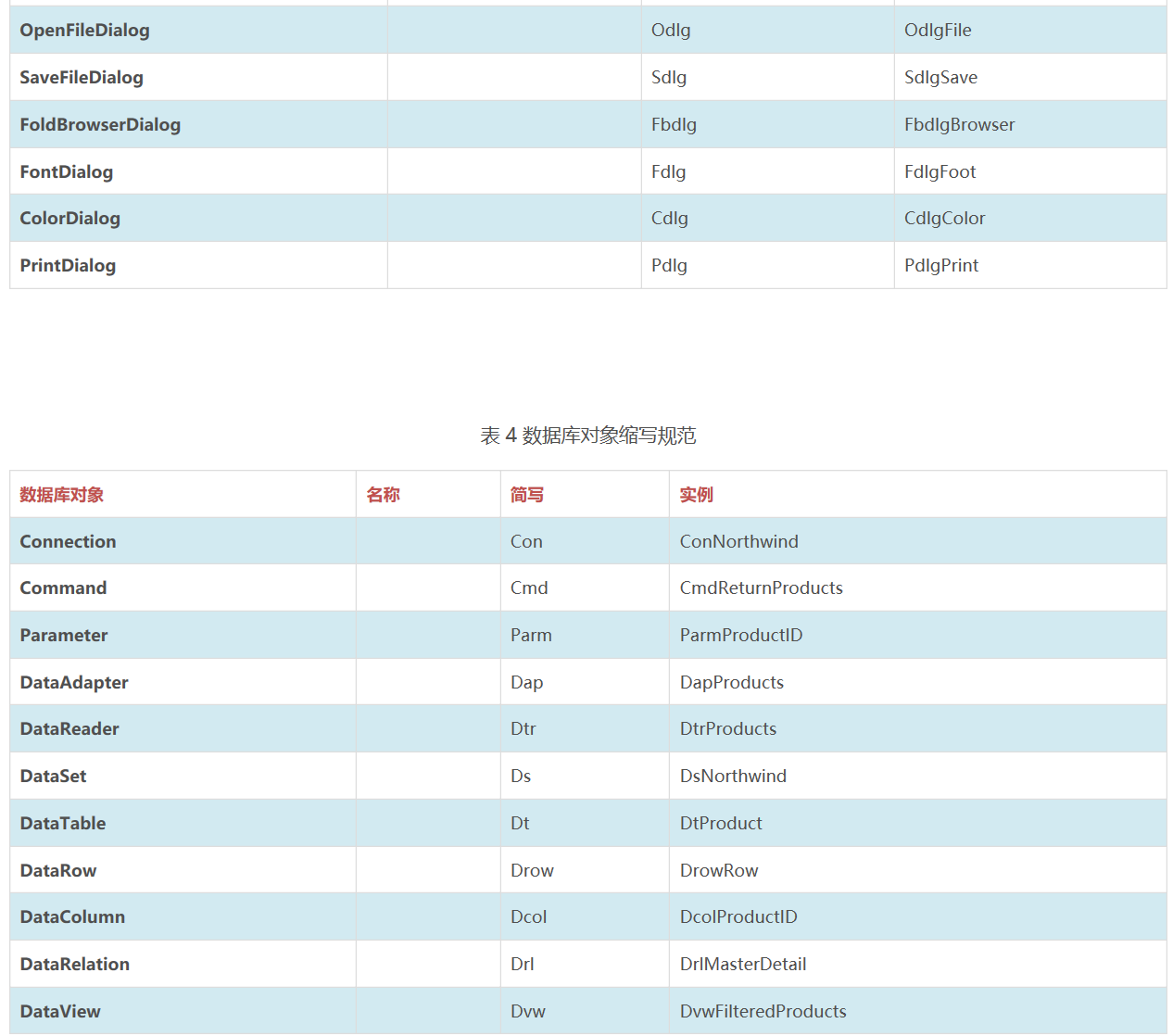
<https://www.cnblogs.com/zhaoshujie/p/9594688.html>

<https://www.cnblogs.com/cg919/p/10512749.html>

1.  命名规范  
a) 类  
【规则1-1】使用Pascal规则命名类名，即首字母要大写。  
eg:  
Class Test  
{  
    ...  
}  
【规则1-2】使用能够反映类功能的名词或名词短语命名类。  
【规则1-3】不要使用“I”、“C”、“\_”等特定含义前缀。  
【规则1-4】自定义异常类应以Exception结尾。  
eg:  
Class TestException  
{  
    ...  
}  
【规则1-5】文件名要能反映类的内容，最好是和类同名。  
   
b) 类字段(类成员)  
【规则2-1】用camel规则来命名类成员变量名称，即首单词（或单词缩写）小写。  
【规则2-2】类字段变量名前可加“\_”前缀。  
【规则2-3】坚决禁止在普通变量前加“m\_”（这是VC老命名规则）。  
eg:  
Class Test  
{  
    privatestring myName;  
    privatestring \_myCoCo;  
    ...  
}  
c）方法  
   【规则3-1】方法名采用Pascal规则，第一个字符要大写。  
【规则3-2】方法名应使用动词或动词短语。  
【规则3-3】类中访问修饰符或功能相同的方法应该放在一起， 且公共或实现接口的方法在前。  
eg:  
Class Test  
{  
      ...  
      publicvoid GetData(...)  
      {  
         ...  
      }  
   
      privatevoid GetName(...)  
      {  
         ...  
      }  
        
      privatestatic void GetCount(...)  
      {  
         ...  
      }  
   
   
}  
   d）属性  
      【规则4-1】使用名词定义属性，属性使用Pascal规则，首字符大写  
      【规则4-2】属性和相应字段名称要关联， 可以使用“重构”菜单来生成属性。  
eg:  
Class Test  
{  
   private string myName;  
   public string MyName  
   {  
      set  
      {  
         myName = Value;  
      }  
      get  
      {  
         return myName;  
      }  
   }  
}  
   e）参数  
      【规则5-1】参数采用camel规则命名，且首字符小写。  
      【规则5-2】使用描述性参数名称，参数名称应当具有最够的说明性。  
      【规则5-3】不要给参数加匈牙利语类型表示法的前缀。  
      【规则5-4】检查方法所有输入参数的有效性。  
      eg:  
      ClassTest  
      {  
         public void GetData(string   
name,string strFlag)  
         {  
            ...  
         }  
      }  
   f）常量  
      【规则6-1】只读常量使用Pascal命名规则，即首字母大写。  
   【规则6-2】枚举名使用Pascal规则命名，枚举成员本质属于常量，命名规则同上。  
      【规则6-3】枚举值从小到大顺序定义。  
      【规则6-4】静态字段或属性采用Pascal规则，即首字符大写。  
      eg:  
      ClassTest  
      {  
         public const double Pi = 3.14159365753;  
         private readonly double   
Pai = 3.14159365753;  
      }  
   g）接口  
      【规则7-1】接口定义使用Pascal规则，且必须以大写“I”开头。  
      【规则7-2】接口名称要有意义，中间不要有下划线“\_”等字符。  
   【规则7-3】如果类实现了接口，名称尽量和接口相同， 只是省掉“I”字符。  
   eg:注意这里的接口修饰符只能用public和internal  
   interfaceITest  
   {  
      ...  
   }  
   h）事件  
      【规则8-1】委托名称采用Pascal规则，即首字符大写。  
   【规则8-2】定义事件的委托要使用EventHandler后缀，且包括sender和e两个参数。  
   【规则8-3】事件用到的参数，名称要带EventArgs后缀。  
   eg:  
   ClassTest  
   {  
      private delegate void DoTask();  
      private event DoTask DoTaskEventHandler;  
      private event DoTask DoTaskEventHandler(Object sender,EventArgse);  
      private event DoTaskDoTaskEventHandler2(string     
strEventArgs)  
      {  
         ...  
      }  
      ...  
   }  
   
i) 命名空间  
【规则9-1】命名空间名称采用Pascal规则，且首字符大写。  
【规则9-2】命名空间名称尽量反映其内容所提供的整体功能。  
eg:  
namespace MyTest  
{  
      ClassTest  
    {  
       ...  
    }  
    ...  
}  
2.  注释规范  
a）文件头部注释  
【规则1-1】文件都包含文件头， 要说明文件名、作者、创建时间、变更记录。  
      【规则1-2】推荐采用.NET形式书写头部注释。(待考查)  
   b）类及其成员注释  
      【规则2-1】对方法和类使用“///”三斜线注释。  
   【规则2-2】代码行文注释采用“//”和“/\*\*/”进行，应该尽量说明问题。  
3.行文规范  
   a）缩写规范  
      【规则1-1】标识符应当直观可望文知意，不提倡使用任何缩写。  
   【规则1-2】字符串变量推荐是用“str”或“s”开头，采用string.Empty来初始化。  
   【规则1-3】普通对象可以以“obj”开头。  
   【规则1-4】缩写可自行定义，一般取单词的前/后字符组成，以含义直观为准则。  
   【规则1-5】一般情况下不要让缩写破坏标识符的含义。  
b）排版  
   【规则2-1】每行语句至少占一行，如果语句过长（超过一屏），则该语句断为两行显示。  
   【规则2-2】把相似的内容放在一起，比如字段、属性、方法、事件等，使用“#region--#endregion”命令分组。  
   【规则2-3】多个程序元素进行对等操作时， 操作符之前、之后或者前后都要加空格。  
   【规则2-4】每个方法的源程序行数原则上应该少于200行。(如果超过过多，则需要另写一方法)  
   【规则2-5】语句嵌套层次不得超过3层。  
   【规则2-6】避免相同的代码段在多个地方出现。(尽量避免代码重复，能复用则复用)  
c）语句结构  
   【规则3-1】如果使用了异常结构，一定要处理异常， 一般是要写日志文件。  
   【规则3-2】分支语句不应该使用复杂长条件， 应该将长条件封装成方法。  
   【规则3-3】switch语句，case后面必须接break。  
   【规则3-4】禁止使用goto语句进行跳转。  
   【规则3-5】行文中严禁出现“魔数”，特定含义的常数必须定义成枚举或常量。  
   【规则3-6】不同类型的操作符混合使用时，使用括号给出优先级。  
   【规则3-7】不允许使用复杂的操作符组合等。  
   【规则3-8】循环、判断语句的程序块部分用花括号括起来， 即使只有一条语句。(return;待议)  
   【规则3-9】在switch语句中总是要有default字句，建议使用断言。  
   【规则3-10】每个类和方法完成单一的功能，不设计多用途面面俱到的类或方法。  
   【规则3-11】严禁使用未经初始化的变量，变量通常使用构造方法来初始。  
d）代码缩进  
   【规则4-1】碰到大括号要换行。  
   【规则4-2】不允许使用Java中的括号换行规范。  
e）大小写  
   【规则5-1】不要创建名称相同，但大小写区别的任何元素。  
   【规则5-2】应当大写仅有两个字符的缩写。(如果只有两个字符，则都大写)  
   【规则5-3】不要把易混淆的数字和字符放在一起。  
   【规则5-4】使用英文命名标识符。  
f）重名规范  
   【规则6-1】不允许变量名、类名、属性名、 方法名等与系统标识符重名。（系统标识符见附表）  
g）SQL编码规范  
   【规则7-1】SQL语句全部大写。(本人的习惯是小写，为了性能，最好是大写，我自己也得把习惯改改咯，哈哈哈)  
   【规则7-2】对较为复杂的SQL语句加上注释，说明其功能。  
   【规则7-3】连接符OR、IN、AND、以及=、<=、>=等前后加空格。  
   【规则7-4】使用明确的列代替 SELECT \*。  
h）软件架构  
   【规则8-1】数据库中每一张表对应一个实体类/数据传输对象（DTO）。  
   【规则8-2】实体类名称使用表名，也可带有Dto后缀。  
   【规则8-3】三层架构应当合理使用，不应生搬硬套。  
   【规则8-4】三层架构元素推荐使用后缀：  
      数据传输对象         XxxxDto  
      DAO工厂            XxxDAOFactory  
      DAO接口            IXxxxDAO  
      服务接口           IxxxxService  
      DAO的数据库实现     XxxxDAOOracle/XxxxDAOInfomix  
      业务逻辑           XxxxManager  
i）系统  
   【规则9-1】在我国内不建议随便使用设计模式等代码模式，因为并不流行。  
   【规则9-2】系统输入、资源操作（如内存分配、文件及目录操作）、网络操作（如通信、调用等）、任务间的操作（如通信、调用等）时必须进行错误、超时、或则异常处理。  
   【规则9-3】模块的编写应有完善的测试方面的考虑。  
   
 







## 过度设计和设计不足

   如今天你打算开发一个知乎的客户端，那么一个过度设计的＂你＂就开始琢磨了，嗯，我要面向对象，我要可扩展，我要牛逼！开始设计，首先业务和界面要分开， 先用库的形式搞一个知乎sdk，然后你就搞了一大堆接口，可是接口还没实现呢，你又想了，我要通用啊，现在的社区网站有一定的共同点，我要把他们的sdk 封装到一起！对，这样牛逼，回头我一开源说不定大家还会表扬我呢。然后你就开始阅读各种api文档，微博，腾讯，天涯，豆瓣，人人。。。你甚至把 email都放进去了，作了很多看起来特别伟大的抽象，终于把sdk设计好了。这时候你又想，接口通用还不够，我的实现也要通用，对，把 ios，android和桌面系统的底层都封装起来。。。这事有点难，你搞了好久差不多可以了。然后，不出意外的，你又想了，所谓mvc，v也要牛逼啊， 我也要做成统一的，最好我这个软件同时支持各大网站，以后升级的时候把库加进去，一配置就支持新的网站，对，插件机制！这个好，eclips就是这样嘛！ 跟大师学没错。。。于是你开始设计抽象ui部分，等这些都做完了，你还不过瘾，你还要内嵌脚本，外置css，各种跨平台，各种松耦合，你甚至要搞一个 objc到java的代码生成器，这样就横跨ios和android了。。。

        设计过度其实就是用户想要杀鸡刀，但你设计出了杀牛刀甚至是电锯。

        稍加留意，可以发现当你在设计的时候有了下述想法，就要警惕是否在过度设计了。

* 设计决策以是否面向扩展为首要衡量标准。
* 解决问题的第一思路是设计模式。
* 相信软件质量会随着使用的框架、库的增加而提升。
* 追求每一个类、层的可替换性。
* 如果没有类图等外在因素的辅助，很难看懂你的代码。
* 大量零散接口的引入并强调是面向接口编程。

        另一个极端就是设计不足，设计出来的系统复用性差，扩展性不强，不能灵活的应对变化，简言之，设计没到位。这个的特征不太好总结，很容易退化成代码问题，我只总结了这样三个危险现象：

* 头脑中没有设计观念，写代码时写到哪想到哪。
* 代码随意堆砌，可见性、生存期使用混乱。
* 设计方案无法应对合理的需求变更，维护成本提高。

        那么要怎样才能减少或避免设计过度与不足呢？

* 先设计后编码。老生常谈了，可惜知易行难。
* 先以设计过度为目标，当能意识到已经过度的时候再来纠正，从中提高对‘适度的设计’的理解。如果不具备设计过度的能力，往往也不太清楚什么是‘适度的设计’，从过度往下降会更容易，但从不足往上升却很难。所以还是要大胆的去设计，然后才能对设计进行评价。
* 提高理论认识，多与他人讨论设计方案以减少设计不足。多交流，把思维用语言表达出来，在此过程中会发现自我的不足，别人也会指出一些没考虑的漏洞，以此来弥补不足之处。
* 在迭代中使用演进式设计以限制设计过度。以满足需求为出发点，不追求完美的永恒的架构，只追求演进式设计过程，通过迭代不断的调整和优化。

## 代码混淆保密

https://www.cnblogs.com/wifi/articles/2453594.html

代码混淆加密

https://jingyan.baidu.com/article/47a29f2457cc76c0142399a5.html

dotNET\_Reactor混淆器的使用。

一、工程经验

1、接口不要局限于interface这种方式。

库，API，xml描述，基类，第三方库，外部文件，数据库，都可以看做接口。

2、编写一次，经常使用

3、泛型与模板

二、软件工程和架构开发方式

通用开发架构

1、内核插件式

2、快速迭代开发

3、类库模块封装单元测试

4、最终优化适用

适应的需求。

1、需求不明确，需要改进变化。

2、市场响应强烈，功能需求随市场变化强烈。

3、产品定义没有最终限制，产品的功能在不断扩展。

软件生命期模型

分阶段模型

瀑布模型

螺旋模型

软件工程方法

三、C#遵守的几个原则

1、内部变量定义采用 \_小写名称

2、属性都采用 大写开头的同类名称

## UML图例

UML关系(泛化,实现,依赖,关联(聚合,组合))

http://justsee.iteye.com/blog/808799

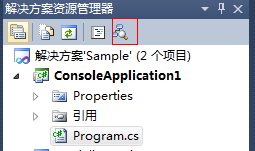
C#中的类图描述

<http://www.cnblogs.com/huangxincheng/archive/2012/10/17/2728736.html>

一：用途

     用于描述系统的静态结构，或许在所有的uml图中，类图是我们最熟悉不过的，在我们没有接触uml的时候，可能都看过

类图，早在vs2005里面“解决方案资源管理器”的下边有一个“查看类图”的小图标，并且还能支持“正向“和”反向“工程。



<1>反向工程

  首先我们定义两个类：User和Product

[复制代码](javascript:void(0);)

1 using System;

2 using System.Collections.Generic;

3 using System.Linq;

4 using System.Text;

5

6 namespace ConsoleApplication1

7 {

8 class Program

9 {

10 static void Main(string[] args)

11 {

12

13 }

14 }

15

16 /// <summary>

17 /// 用户类

18 /// </summary>

19 public class User

20 {

21 public string Name { get; set; }

22

23 public int Age { get; set; }

24

25 public string Sex { get; set; }

26 }

27

28 /// <summary>

29 /// 产品类

30 /// </summary>

31 public class Product

32 {

33 public string Name { get; set; }

34

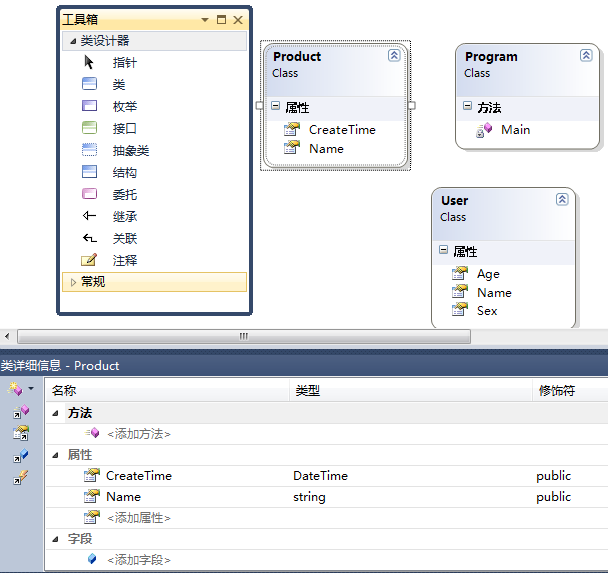
35 public DateTime CreateTime { get; set; }

36 }

37 }

[复制代码](javascript:void(0);)

然后我们点击”查看类图“，看看给我们生成的类图是咋样的。



<2> 正向工程

      既然是正向工程，那我们就可以在”类设计器“上面随便拖一些元素看看效果，细节的大家可以自己玩一玩。

二：基本元素

1：类图，枚举，接口，抽象类，结构，委托

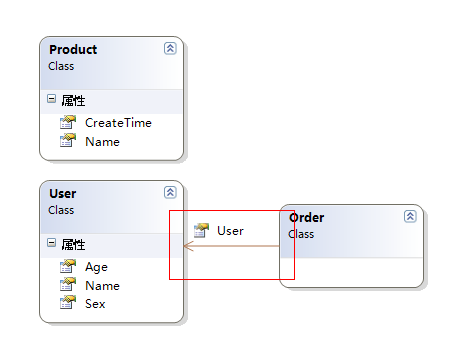
    这几个元素我想学OO的都已经烂熟于心了，也没有什么好解释的。

2：关联关系

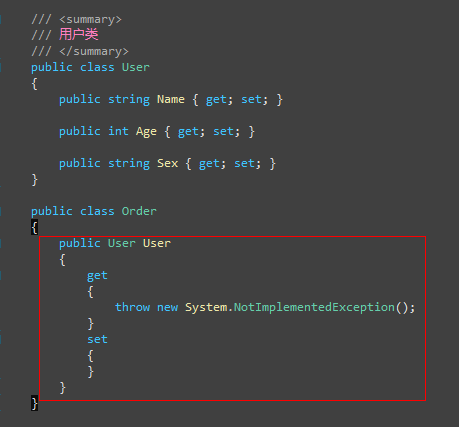
    关联关系一般作为类与类之间的一种强依赖关系，这种关系具有稳定和长期性，比如在C#中的代码实现为：将一个类作为

另一个类中的属性，比如这里我新建一个Order类，将User类作为Order类的一个属性。

先看类图：



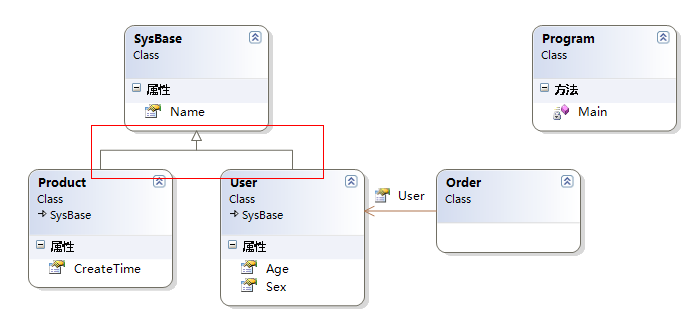
然后看下是否为我们需要的代码：



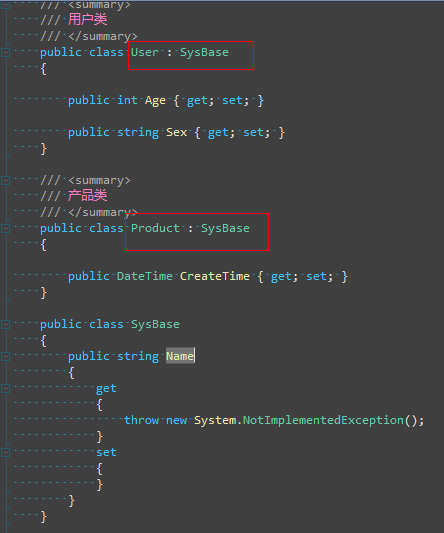
3：继承关系

   在OO的三大特性中就有继承，我们都知道继承这么个概念，那么在类图中该如何展现呢？我们发现在User和Product

中都有一个Name属性，根据OO原则我们需要将Name属性单独提出来，然后让其他类继承。



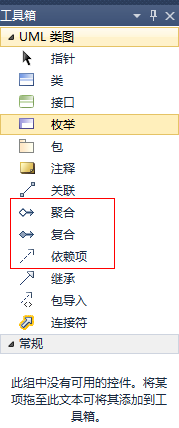
下面我们看下生成代码，是否真如类图描述一样，（双击）任意类图即可，嘿嘿，是不是有点意思。



好，到现在为止，在类图这一块，我们已经掌握了20%，只要多练习练习即可，当然你可能觉得这些代码比较死，是的，

实际开发中，我们常会用CodeSmith来解决这些枯燥无味的代码。

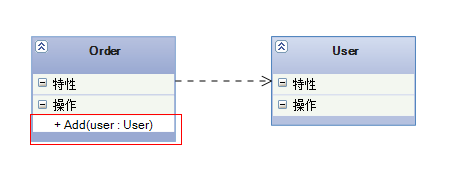
在uml的类图中还有几个关系需要表达一下，只不过实际应用比较少而已，好，下面我们看看”建模项目“里面的UML类图



4：依赖关系

    同样它也是类与类之间的关系，只不过这种关系比较弱，具有临时性和特定环境下的偶然性，可能大家不是很容易理解，

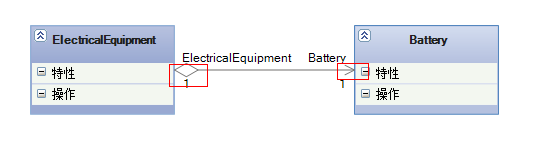
如果用C#解释就一目了然了，在代码中一般是一个类作为另一个类中方法的参数，既然是参数，那么它的生命周期你懂的。



5：聚合关系

    在官方文档中它描述的是一种”has-a“的关系，也就是整体与局部的关系，整体挂了，局部不见得就挂了，比如：你的大功率

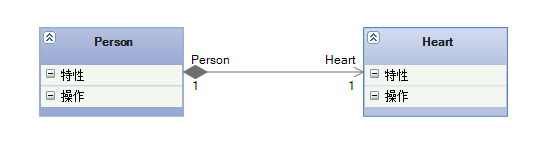
电器挂了，不见得里面的电池就挂了，其次我们要注意”空心菱形“是整体端，“箭头”端是局部端。



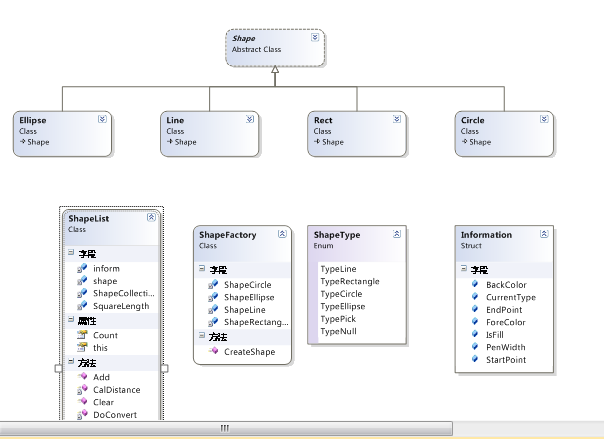
6：组合关系

    在官方文档中它描述的是一种”contains-a“的关系，同样也是表示整体和局部的关系，整体挂了，局部也挂了，因为他们享有

共同的生命周期，比如：你挂了，你的心脏肯定挂了，同样"实心菱形“是整体端。



下面我们看看稍微复杂一点的”画图软件“的类图设计，大家也可以看看自己手头的项目类图，是否符合OO规范。



## 接口和抽象类设计

### 概念

什么是**接口**？

接口是包含一组虚方法的抽象类型，其中每一种方法都有其名称、参数和返回值。

接口方法不能包含任何实现，CLR允许接口可以包含事件、属性、索引器、静态方法、静态字段、静态构造函数以及常数。但是注意：C#中不能包含任何静态成员。

一个类可以实现多个接口，当一个类继承某个接口时，它不仅要实现该接口定义的所有方法，还要实现该接口从其他接口中继承的所有方法。

什么是**抽象类**？

抽象类提供多个派生类共享基类的公共定义，它既可以提供抽象方法，也可以提供非抽象方法。抽象类不能实例化，必须通过继承由派生类实现其抽象方法，因此对抽象类不能使用new关键字，也不能被密封。如果派生类没有实现所有的抽象方法，则该派生类也必须声明为抽象类。另外，实现抽象方法由overriding方法来实现。

#### 抽象类

(1) 抽象方法只作声明，而不包含实现，可以看成是没有实现体的虚方法  
 (2) 抽象类不能被实例化  
 (3) 抽象类可以但不是必须有抽象属性和抽象方法，但是一旦有了抽象方法，就一定要把这个类声明为抽象类  
 (4) 具体派生类必须覆盖基类的抽象方法  
 (5) 抽象派生类可以覆盖基类的抽象方法，也可以不覆盖。如果不覆盖，则其具体派生类必须覆盖它们。如：  
  
using System;  
  
public abstract class A //抽象类A   
{   
 private int num=0;  
  
 public int Num //抽象类包含属性   
 {   
 get   
 {   
 return num;   
 }   
 set   
 {   
 num = value;   
 }   
   
 }  
  
 public virtual int getNum() //抽象类包含虚方法   
 {   
 return num;   
 }  
  
 public void setNum(int n) // //抽象类包含普通方法   
 {   
 this.num = n;   
 }  
  
 public abstract void E(); //类A中的抽象方法E   
   
}  
  
public abstract class B : A //由于类B继承了类A中的抽象方法E,所以类B也变成了抽象类   
{   
   
}  
  
public class C : B   
{   
 public override void E() //重写从类A继承的抽象方法。如果类B自己还定义了抽象方法，也必须重写   
 {   
 //throw new Exception("The method or operation is not implemented.");   
 }   
}  
  
public class Test   
{   
 static void Main()   
 {   
 C c = new C();   
 c.E();   
 }   
}

#### 接 口

(1) 接口不能被实例化  
 (2) 接口只能包含方法声明  
 (3) 接口的成员包括方法、属性、索引器、事件  
 (4) 接口中不能包含常量、字段(域)、[构造函数](https://www.baidu.com/s?wd=%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3P1FhP1IhuHTLuHczPhR40ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En10zPHTYPHbv)、析构函数、[静态成员](https://www.baidu.com/s?wd=%E9%9D%99%E6%80%81%E6%88%90%E5%91%98&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3P1FhP1IhuHTLuHczPhR40ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En10zPHTYPHbv)。如：  
  
public delegate void EventHandler(object sender, Event e);  
  
public interface ITest   
{   
 //int x = 0;  
  
 int A   
 {   
 get;   
 set;   
 }  
  
 void Test();  
  
 event EventHandler Event;   
   
 int this[int index]   
 {   
 get;  
  
 set;   
 }   
}  
  
(5) 接口中的所有成员默认为public，因此接口中不能有private修饰符  
 (6) 派生类必须实现接口的所有成员  
 (7) 一个类可以直接实现多个接口，接口之间用逗号隔开  
 (8) 一个接口可以有多个父接口，实现该接口的类必须实现所有父接口中的所有成员

#### 抽象类和接口

相同点：  
 (1) 都可以被继承  
 (2) 都不能被实例化  
 (3) 都可以包含方法声明  
 (4) 派生类必须实现未实现的方法  
 区 别：  
 (1) 抽象基类可以定义字段、属性、方法实现。接口只能定义属性、索引器、事件、和方法声明，不能包含字段。  
 (2) 抽象类是一个不完整的类，需要进一步细化，而接口是一个行为规范。微软的自定义接口总是后带able字段，证明其是表述一类“我能做。。。”  
 (3) 接口可以被多重实现，抽象类只能被单一继承  
 (4) 抽象类更多的是定义在一系列紧密相关的类间，而接口大多数是关系疏松但都实现某一功能的类中  
 (5) 抽象类是从一系列相关对象中抽象出来的概念， 因此反映的是事物的内部共性；接口是为了满足外部调用而定义的一个功能约定， 因此反映的是事物的外部特性  
 (6) 接口基本上不具备继承的任何具体特点,它仅仅承诺了能够调用的方法   
 (7) 接口可以用于支持回调,而继承并不具备这个特点  
 (8) 抽象类实现的具体方法默认为虚的，但实现接口的类中的接口方法却默认为非虚的，当然您也可以声明为虚的   
 (9) 如果抽象类实现接口，则可以把接口中方法映射到抽象类中作为抽象方法而不必实现，而在抽象类的子类中实现接口中方法

抽象类中

抽象方法必须重写 不必声明主体 只能重写一次。

虚方法可以重写，也可以不重写。 必须声明 可以被深入的继承重写多次。

普通方法不可以重写 必须声明 不能被重写。

子类可以覆盖被继承类的基类函数（抽象和虚函数）。不管基类和抽象类都只能单向继承，接口可以多个实现，只要有一个类实现就可以。

### 相同点和不同点

#### 相同点

1. 都不能被直接实例化，都可以通过继承实现其抽象方法。

2. 都是面向抽象编程的技术基础，实现了诸多的设计模式。

#### 不同点

1. 接口支持多继承；抽象类不能实现多继承。

2. 接口只能定义抽象规则；抽象类既可以定义规则，还可能提供已实现的成员。

3. 接口是一组行为规范；抽象类是一个不完全的类，着重族的概念。

4. 接口可以用于支持回调；抽象类不能实现回调，因为继承不支持。

5. 接口只包含方法、属性、索引器、事件的签名，但不能定义字段和包含实现的方法；抽象类可以定义字段、属性、包含有实现的方法。

6. 接口可以作用于值类型和引用类型；抽象类只能作用于引用类型。例如，Struct就可以继承接口，而不能继承类。

通过相同与不同的比较，我们 只能说接口和抽象类，各有所长，但无优略。在实际的编程实践中，我们要视具体情况来酌情量才，但是以下的经验和积累，或许能给大家一些启示，除了我的一些 积累之外，很多都来源于经典，我相信经得起考验。所以在规则与场合中，我们学习这些经典，最重要的是学以致用，当然我将以一家之言博大家之笑，看官请继 续。

#### 规则与场合

1.         请记住，面向对象思想的一个最重要的原则就是：**面向接口编程**。

2.         借助接口和抽象类，23个设计模式中的很多思想被巧妙的实现了，我认为其精髓简单说来就是：面向抽象编程。

3.         抽象类应主要用于关系密切的对象，而接口最适合为不相关的类提供通用功能。

4.         接口着重于CAN-DO关系类型，而抽象类则偏重于IS-A式的关系；

5.         接口多定义对象的行为；抽象类多定义对象的属性；

6.         接口定义可以使用public、protected、internal 和private修饰符，但是几乎所有的接口都定义为public，原因就不必多说了。

7.         在接口中，所有的方法都默认为public。

8.         “接口不变”，是应该考虑的重要因素。所以，在由接口增加扩展时，应该增加新的接口，而不能更改现有接口。

9.         尽量将接口设计成功能单一的功能块，以.NET Framework为例，IDisposable、IDisposable、IComparable、IEquatable、IEnumerable等都只包含一个公共方法。

10.     接口名称前面的大写字母“I”是一个约定，正如字段名以下划线开头一样，请坚持这些原则。

11.     如果预计会出现版本问题，可以创建“抽象类”。例如，创建了狗（Dog）、鸡（Chicken）和鸭（Duck），那么应该考虑抽象出动物（Animal）来应对以后可能出现风马牛的事情。而向接口中添加新成员则会强制要求修改所有派生类，并重新编译，所以版本式的问题最好以抽象类来实现。

12.     从抽象类派生的非抽象类必须包括继承的所有抽象方法和抽象访问器的实实现。

13.     对抽象类不能使用new关键字，也不能被密封，原因是抽象类不能被实例化。

14.     在抽象方法声明中不能使用 static 或 virtual 修饰符。

**MSDN这样建议**：

1.         **如果预计要创建组件的多个版本，则创建抽象类**。抽象类提供简单易行的方法来控制组件版本。通过更新基类，所有继承类都随更改自动更新。另一方面，接口一旦创建就不能更改。如果需要接口的新版本，必须创建一个全新的接口。

**2.         如果创建的功能将在大范围的全异对象间使用，则使用接口。抽象类应主要用于关系密切的对象，而接口最适合为不相关的类提供通用功能。**

3.         **如果要设计小而简练的功能块，则使用接口。如果要设计大的功能单元，则使用抽象类**。

4.         **如果要在组件的所有实现间提供通用的已实现功能，则使用抽象类**。抽象类允许部分实现类，而接口不包含任何成员的实现。

### 经典示例

.NET Framework中关于接口和抽象类的使用：

**集合类使用了基于接口的设计**，请关注System.Collections中关于接口的设计实现；

**数据流相关类使用了基于抽象类的设计**，请关注System.IO.Stream类的抽象类设计机制。

### 相关概念

1.         面向对象及其三大特性

2.         继承性(Inheritance)

3.         多态性(Polymorphism)

4.         封装性(Encapsulation)

5.         设计模式

### 附录

**1. 考虑在逐步抽象过程中下列三者各自的特性变化**(可以作一总结)：

**1.类 ————→ 2.抽象类 ————→ 3.接口**

**|————————抽象———————→|**

#### 接口和抽象类总结一览表

|  |  |  |  |
| --- | --- | --- | --- |
|  | **关键字** | **接口** | **抽象类** |
| (本表只为个人查阅之用，表中可能存在的错误、不准确之处谢谢指正)  最后更新：2011-10-26    作者：Ronli  更新链接：<http://www.cnblogs.com/ronli/archive/2011/10/26/2224654.html> | | | |
| **不同点** | | | |
| 1 | 继承 | 接口支持多继承 | 抽象类不能实现多继承 |
| 2 | 抽象 | 接口只能定义抽象规则 | 抽象类既可以定义规则，还可能提供已实现的成员。但不能使用new关键字，也不能被密封，原因是抽象类不能被实例化 |
| 3 | 概念 | 接口是一组行为规范 | 抽象类是一个不完全的类，着重族的概念 |
| 4 | 回调 | 接口可以用于支持回调 | 抽象类不能实现回调，因为继承不支持 |
| 5 | 成员 | 接口只包含方法、属性、索引器、事件的签名，但不能定义字段和包含实现的方法 | 抽象类可以定义字段、属性、包含有实现的方法 |
| 6 | 数据类型 | 接口可以作用于值类型和引用类型。例如，Struct就可以继承接口，而不能继承类 | 抽象类只能作用于引用类型 |
| **规则与场合** | | | |
| **1** | 面向对象思想最重要的原则 | 面向接口编程 | |
| **2** | 设计模式精髓 | 面向抽象编程 | |
| 3 | 主要应用 | 为不相关的类提供通用功能 | 关系密切的对象 |
| 4 | 关系类型 | CAN-DO关系类型 | IS-A关系类型 |
| 5 | 定义对象 | 定义对象的行为 | 定义对象的属性 |
| 6 | 可用修饰符 | 都可以使用public、protected、internal 和private修饰符 | |
| 7 | 默认修饰符 | public | private |
| 8 | 更新 | 接口不变。接口一旦创建就不能更改，如果需要接口的新版本，必须创建一个全新的接口 | 易控制组件版本，更新基类，所有继承类都随更改自动更新 |
| 9 | 功能设计 | 设计小而简练的功能块 | 设计大的功能单元 |
| 10 | 命名约定 | 大写字母“I”开头 |  |
| 11 | 组件版本问题 | 接口中添加新成员则会强制要求修改所有派生类，并重新编译 | 更新基类，所有继承类都随更改自动更新 |
| 12 | 继承要求 | 必须实现所有接口方法 | 从抽象类派生的非抽象类必须包括继承的所有抽象方法和抽象访问器的实现 |
| 13 | 定义要求 | 只能定义抽象规则 | 既可以定义规则，还可能提供已实现的成员。但不能使用new关键字，也不能被密封，原因是抽象类不能被实例化 |
| 14 | 抽象方法声明 | 默认为public | 不能使用 static 或 virtual 修饰符 |

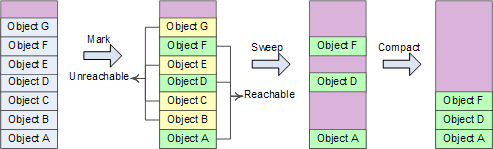
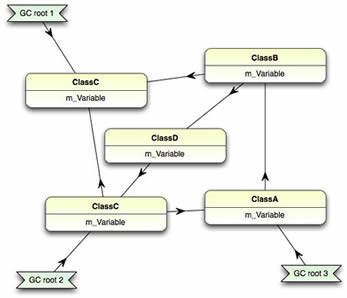
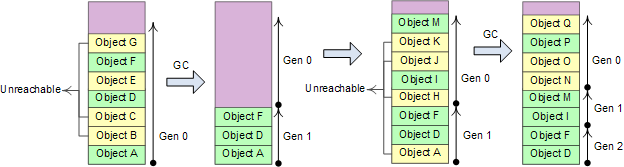
## 回收机制

**GC的前世与今生**

虽然本文是以.net作为目标来讲述GC，但是GC的概念并非才诞生不久。早在1958年，由鼎鼎大名的图林奖得主John McCarthy所实现的Lisp语言就已经提供了GC的功能，这是GC的第一次出现。Lisp的程序员认为内存管理太重要了，所以不能由程序员自己来管理。但后来的日子里Lisp却没有成气候，采用内存手动管理的语言占据了上风，以C为代表。出于同样的理由，不同的人却又不同的看法，C程序员认为内存管理太重要了，所以不能由系统来管理，并且讥笑Lisp程序慢如乌龟的运行速度。的确，在那个对每一个Byte都要精心计算的年代GC的速度和对系统资源的大量占用使很多人的无法接受。而后，1984年由Dave Ungar开发的Small talk语言第一次采用了Generational garbage collection的技术（这个技术在下文中会谈到），但是Small talk也没有得到十分广泛的应用。   
直到20世纪90年代中期GC才以主角的身份登上了历史的舞台，这不得不归功于Java的进步，今日的GC已非吴下阿蒙。Java采用VM（Virtual Machine）机制，由VM来管理程序的运行当然也包括对GC管理。90年代末期.net出现了，.net采用了和Java类似的方法由CLR(Common Language Runtime)来管理。这两大阵营的出现将人们引入了以虚拟平台为基础的开发时代，GC也在这个时候越来越得到大众的关注。   
为什么要使用GC呢？也可以说是为什么要使用内存自动管理？有下面的几个原因：   
1、提高了软件开发的抽象度；   
2、程序员可以将精力集中在实际的问题上而不用分心来管理内存的问题；   
3、可以使模块的接口更加的清晰，减小模块间的偶合；   
4、大大减少了内存人为管理不当所带来的Bug；   
5、使内存管理更加高效。   
总的说来就是GC可以使程序员可以从复杂的内存问题中摆脱出来，从而提高了软件开发的速度、质量和安全性。 

**什么是GC**

GC如其名，就是垃圾收集，当然这里仅就内存而言。Garbage Collector（垃圾收集器，在不至于混淆的情况下也成为GC）以应用程序的root为基础，遍历应用程序在Heap上动态分配的所有对象[2]，通过识别它们是否被引用来确定哪些对象是已经死亡的哪些仍需要被使用。已经不再被应用程序的root或者别的对象所引用的对象就是已经死亡的对象，即所谓的垃圾，需要被回收。这就是GC工作的原理。为了实现这个原理，GC有多种算法。比较常见的算法有Reference Counting，Mark Sweep，Copy Collection等等。目前主流的虚拟系统.net CLR，Java VM和Rotor都是采用的Mark Sweep算法。

**一、Mark-Compact 标记压缩算法**简单把.NET的GC算法看作Mark-Compact算法   
阶段1: Mark-Sweep 标记清除阶段   
先假设heap中所有对象都可以回收，然后找出不能回收的对象，给这些对象打上标记，最后heap中没有打标记的对象都是可以被回收的   
阶段2: Compact 压缩阶段   
对象回收之后heap内存空间变得不连续，在heap中移动这些对象，使他们重新从heap基地址开始连续排列，类似于磁盘空间的碎片整理   
  
Heap内存经过回收、压缩之后，可以继续采用前面的heap内存分配方法，即仅用一个指针记录heap分配的起始地址就可以   
主要处理步骤：将线程挂起=>确定roots=>创建reachable objectsgraph=>对象回收=>heap压缩=>指针修复   
可以这样理解roots：heap中对象的引用关系错综复杂（交叉引用、循环引用），形成复杂的graph，roots是CLR在heap之外可以找到的各种入口点。GC搜索roots的地方包括全局对象、静态变量、局部对象、函数调用参数、当前CPU寄存器中的对象指针（还有finalizationqueue）等。主要可以归为2种类型：已经初始化了的静态变量、线程仍在使用的对象（stack+CPU register）   
Reachable objects：指根据对象引用关系，从roots出发可以到达的对象。例如当前执行函数的局部变量对象A是一个rootobject，他的成员变量引用了对象B，则B是一个reachable object。从roots出发可以创建reachable objectsgraph，剩余对象即为unreachable，可以被回收   
  
指针修复是因为compact过程移动了heap对象，对象地址发生变化，需要修复所有引用指针，包括stack、CPUregister中的指针以及heap中其他对象的引用指针   
Debug和release执行模式之间稍有区别，release模式下后续代码没有引用的对象是unreachable的，而debug模式下需要等到当前函数执行完毕，这些对象才会成为unreachable，目的是为了调试时跟踪局部对象的内容   
传给了COM+的托管对象也会成为root，并且具有一个引用计数器以兼容COM+的内存管理机制，引用计数器为0时这些对象才可能成为被回收对象   
Pinnedobjects指分配之后不能移动位置的对象，例如传递给非托管代码的对象（或者使用了fixed关键字），GC在指针修复时无法修改非托管代码中的引用指针，因此将这些对象移动将发生异常。pinnedobjects会导致heap出现碎片，但大部分情况来说传给非托管代码的对象应当在GC时能够被回收掉   
**二、 Generational 分代算法**程序可能使用几百M、几G的内存，对这样的内存区域进行GC操作成本很高，分代算法具备一定统计学基础，对GC的性能改善效果比较明显   
将对象按照生命周期分成新的、老的，根据统计分布规律所反映的结果，可以对新、老区域采用不同的回收策略和算法，加强对新区域的回收处理力度，争取在较短时间间隔、较小的内存区域内，以较低成本将执行路径上大量新近抛弃不再使用的局部对象及时回收掉   
分代算法的假设前提条件：   
1、大量新创建的对象生命周期都比较短，而较老的对象生命周期会更长   
2、对部分内存进行回收比基于全部内存的回收操作要快   
3、新创建的对象之间关联程度通常较强。heap分配的对象是连续的，关联度较强有利于提高CPU cache的命中率   
.NET将heap分成3个代龄区域: Gen 0、Gen 1、Gen 2   
  
Heap分为3个代龄区域，相应的GC有3种方式: # Gen 0 collections, # Gen 1 collections, #Gen 2 collections。如果Gen 0 heap内存达到阀值，则触发0代GC，0代GC后Gen 0中幸存的对象进入Gen1。如果Gen 1的内存达到阀值，则进行1代GC，1代GC将Gen 0 heap和Gen 1 heap一起进行回收，幸存的对象进入Gen2。2代GC将Gen 0 heap、Gen 1 heap和Gen 2 heap一起回收   
Gen 0和Gen 1比较小，这两个代龄加起来总是保持在16M左右；Gen2的大小由应用程序确定，可能达到几G，因此0代和1代GC的成本非常低，2代GC称为fullGC，通常成本很高。粗略的计算0代和1代GC应当能在几毫秒到几十毫秒之间完成，Gen 2 heap比较大时fullGC可能需要花费几秒时间。大致上来讲.NET应用运行期间2代、1代和0代GC的频率应当大致为1:10:100。

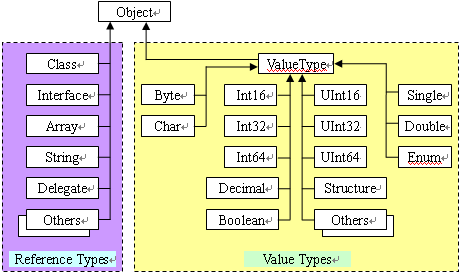
**三、Finalization Queue和Freachable Queue**

这两个队列和.net对象所提供的Finalize方法有关。这两个队列并不用于存储真正的对象，而是存储一组指向对象的指针。当程序中使用了new操作符在Managed Heap上分配空间时，GC会对其进行分析，如果该对象含有Finalize方法则在Finalization Queue中添加一个指向该对象的指针。在GC被启动以后，经过Mark阶段分辨出哪些是垃圾。再在垃圾中搜索，如果发现垃圾中有被Finalization Queue中的指针所指向的对象，则将这个对象从垃圾中分离出来，并将指向它的指针移动到Freachable Queue中。这个过程被称为是对象的复生（Resurrection），本来死去的对象就这样被救活了。为什么要救活它呢？因为这个对象的Finalize方法还没有被执行，所以不能让它死去。Freachable Queue平时不做什么事，但是一旦里面被添加了指针之后，它就会去触发所指对象的Finalize方法执行，之后将这个指针从队列中剔除，这是对象就可以安静的死去了。.net framework的System.GC类提供了控制Finalize的两个方法，ReRegisterForFinalize和SuppressFinalize。前者是请求系统完成对象的Finalize方法，后者是请求系统不要完成对象的Finalize方法。ReRegisterForFinalize方法其实就是将指向对象的指针重新添加到Finalization Queue中。这就出现了一个很有趣的现象，因为在Finalization Queue中的对象可以复生，如果在对象的Finalize方法中调用ReRegisterForFinalize方法，这样就形成了一个在堆上永远不会死去的对象，像凤凰涅槃一样每次死的时候都可以复生。

**托管资源：**

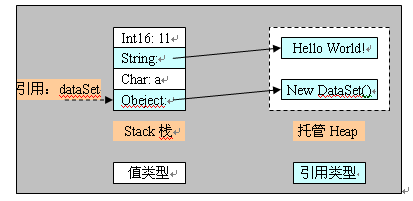
Net中的所有类型都是（直接或间接）从System.Object类型派生的。

CTS中的类型被分成两大类——引用类型（reference type，又叫托管类型[managed type]），分配在内存堆上，值类型（value type）。值类型分配在堆栈上。如图



值类型在栈里，先进后出，值类型变量的生命有先后顺序，这个确保了值类型变量在推出作用域以前会释放资源。比引用类型更简单和高效。堆栈是从高地址往低地址分配内存。

引用类型分配在托管堆(Managed Heap)上，声明一个变量在栈上保存，当使用new创建对象时，会把对象的地址存储在这个变量里。托管堆相反，从低地址往高地址分配内存，如图



.net中超过80%的资源都是托管资源。

**非托管资源：**

ApplicationContext,Brush,Component,ComponentDesigner,Container,Context,Cursor,FileStream,Font,Icon,Image,Matrix,Object,OdbcDataReader,OleDBDataReader,Pen,Regex,Socket,StreamWriter,Timer,Tooltip ,文件句柄,GDI资源,数据库连接等等资源。可能在使用的时候很多都没有注意到！

**.NET的GC机制有这样两个问题：**

首先，GC并不是能释放所有的资源。它不能自动释放非托管资源。

第二，GC并不是实时性的，这将会造成系统性能上的瓶颈和不确定性。

GC并不是实时性的，这会造成系统性能上的瓶颈和不确定性。所以有了IDisposable接口，IDisposable接口定义了Dispose方法，这个方法用来供程序员显式调用以释放非托管资源。使用using 语句可以简化资源管理。

**示例**

当你用Dispose方法释放未托管对象的时候，应该调用GC.SuppressFinalize。如果对象正在终结队列(finalization queue),GC.SuppressFinalize会阻止GC调用Finalize方法。因为Finalize方法的调用会牺牲部分性能。如果你的Dispose方法已经对委托管资源作了清理，就没必要让GC再调用对象的Finalize方法(MSDN)。附上MSDN的代码，大家可以参考.

public class BaseResource : IDisposable   
{   
// 指向外部非托管资源   
private IntPtr handle;   
// 此类使用的其它托管资源.   
private Component Components;   
// 跟踪是否调用.Dispose方法，标识位，控制垃圾收集器的行为   
private bool disposed = false;   
  
// 构造函数   
public BaseResource()   
{   
// Insert appropriate constructor code here.   
}   
  
// 实现接口IDisposable.   
// 不能声明为虚方法virtual.   
// 子类不能重写这个方法.   
public void Dispose()   
{   
Dispose(true);   
// 离开终结队列Finalization queue   
// 设置对象的阻止终结器代码   
//   
GC.SuppressFinalize(this);   
}   
  
// Dispose(bool disposing) 执行分两种不同的情况.   
// 如果disposing 等于 true, 方法已经被调用   
// 或者间接被用户代码调用. 托管和非托管的代码都能被释放   
// 如果disposing 等于false, 方法已经被终结器 finalizer 从内部调用过，   
//你就不能在引用其他对象，只有非托管资源可以被释放。   
protected virtual void Dispose(bool disposing)   
{   
// 检查Dispose 是否被调用过.   
if (!this.disposed)   
{   
// 如果等于true, 释放所有托管和非托管资源   
if (disposing)   
{   
// 释放托管资源.   
Components.Dispose();   
}   
// 释放非托管资源，如果disposing为 false,   
// 只会执行下面的代码.   
CloseHandle(handle);   
handle = IntPtr.Zero;   
// 注意这里是非线程安全的.   
// 在托管资源释放以后可以启动其它线程销毁对象，   
// 但是在disposed标记设置为true前   
// 如果线程安全是必须的，客户端必须实现。   
  
}   
disposed = true;   
}   
// 使用interop 调用方法   
// 清除非托管资源.   
[System.Runtime.InteropServices.DllImport("Kernel32")]   
private extern static Boolean CloseHandle(IntPtr handle);   
  
// 使用C# 析构函数来实现终结器代码   
// 这个只在Dispose方法没被调用的前提下，才能调用执行。   
// 如果你给基类终结的机会.   
// 不要给子类提供析构函数.   
~BaseResource()   
{   
// 不要重复创建清理的代码.   
// 基于可靠性和可维护性考虑，调用Dispose(false) 是最佳的方式   
Dispose(false);   
}   
  
// 允许你多次调用Dispose方法,   
// 但是会抛出异常如果对象已经释放。   
// 不论你什么时间处理对象都会核查对象的是否释放,   
// check to see if it has been disposed.   
public void DoSomething()   
{   
if (this.disposed)   
{   
throw new ObjectDisposedException();   
}   
}   
  
  
// 不要设置方法为virtual.   
// 继承类不允许重写这个方法   
public void Close()   
{   
// 无参数调用Dispose参数.   
Dispose();   
}   
  
public static void Main()   
{   
// Insert code here to create   
// and use a BaseResource object.   
}   
}

GC.Collect() 方法

作用：强制进行垃圾回收。

GC的方法：

|  |  |
| --- | --- |
| 名称 | 说明 |
| Collect() | 强制对所有代进行即时垃圾回收。 |
| Collect(Int32) | 强制对零代到指定代进行即时垃圾回收。 |
| Collect(Int32, GCCollectionMode) | 强制在 GCCollectionMode 值所指定的时间对零代到指定代进行垃圾回收。 |

**GC注意事项：**

1、只管理内存，非托管资源，如文件句柄，GDI资源，数据库连接等还需要用户去管理

2、循环引用，网状结构等的实现会变得简单。GC的标志也压缩算法能有效的检测这些关系，并将不再被引用的网状结构整体删除。

3、GC通过从程序的根对象开始遍历来检测一个对象是否可被其他对象访问，而不是用类似于COM中的引用计数方法。

4、GC在一个独立的线程中运行来删除不再被引用的内存

5、GC每次运行时会压缩托管堆

6、你必须对非托管资源的释放负责。可以通过在类型中定义Finalizer来保证资源得到释放。

7、对象的Finalizer被执行的时间是在对象不再被引用后的某个不确定的时间。注意并非和C++中一样在对象超出声明周期时立即执行析构函数

8、Finalizer的使用有性能上的代价。需要Finalization的对象不会立即被清除，而需要先执行Finalizer.Finalizer不是在GC执行的线程被调用。GC把每一个需要执行Finalizer的对象放到一个队列中去，然后启动另一个线程来执行所有这些Finalizer.而GC线程继续去删除其他待回收的对象。在下一个GC周期，这些执行完Finalizer的对象的内存才会被回收。

9、.NET GC使用"代"(generations)的概念来优化性能。代帮助GC更迅速的识别那些最可能成为垃圾的对象。在上次执行完垃圾回收后新创建的对象为第0代对象。经历了一次GC周期的对象为第1代对象。经历了两次或更多的GC周期的对象为第2代对象。代的作用是为了区分局部变量和需要在应用程序生存周期中一直存活的对象。大部分第0代对象是局部变量。成员变量和全局变量很快变成第1代对象并最终成为第2代对象。

10、GC对不同代的对象执行不同的检查策略以优化性能。每个GC周期都会检查第0代对象。大约1/10的GC周期检查第0代和第1代对象。大约1/100的GC周期检查所有的对象。重新思考Finalization的代价：需要Finalization的对象可能比不需要Finalization在内存中停留额外9个GC周期。如果此时它还没有被Finalize,就变成第2代对象，从而在内存中停留更长时间。

# 新观点

## 接口和抽象类区别

1、接口可以看做是后续继承的特殊类。

2、所有共性的东西，采用基类；后续群体中的一部分有特殊功能的，抽出接口类

3、接口的功能封装成一个具体的类，然后在继承类中实现一个字段？

如果要这样的话，那么就必须要先实现封装类的方法，同时所有继承类的实现的方法又一致了。

所以抽象类和继承类的本质就在于整体和局部，如果大部分都有这样的特性，就来抽象类；如果群体中的小范围有同样的方法，就用接口类

## [静态类，静态方法，静态变量与非静态之前的区别](http://blog.csdn.net/wangsen2235068/article/details/6603249)

所有static 字段都有一共公共存储空间，所有方法都有共同空间。静态变量可以直接引用。静态方法需要静态变量。静态类里面所有的变量和方法都要是静态的。

http://blog.csdn.net/wangsen2235068/article/details/6603249

**静态类：**

1：仅包含静态变量和静态方法。  
2：无法实例化。  
3：不能包含实例构造函数。

4：是密封的。

**非静态类：**

1.可以包含非静态成员，也可以包含静态成员。

2.每次实例化都是一个新的对象。

3.可以不显示实例化，因为内部有一个默认的静态构造函数,不可重载，当创建类实例或引用任何静态成员之前，静态构造函数被自动执行，并且只执行一次。

4.也可以通过 类名.方法名或变量名 访问静态成员

**静态方法：**

1.静态方法不能引用非静态变量

4：静态方法只能被重载，而不能被重写，因为静态方法不属于类的实例成员；当然也不能是Virtual和abstract类型的

**非静态方法：**

1.可以同时引用 非静态变量和静态变量

**静态变量**

1.只有一个副本，实例化类，不会初始化静态变量的值。属于类所有，生命周期和网站运用程序一样长  
2.C# 不支持静态局部变量(在方法内部定义静态变量)。

**非静态变量**

1.每次实例化都是一个新的变量。自然要初始化值

**静态构造函数**

1：静态类可以有静态构造函数，静态构造函数不可继承；

2：静态构造函数可以用于静态类，也可用于非静态类；

3：静态构造函数无访问修饰符、无参数，只有一个 static 标志；

4：静态构造函数不可被直接调用，当创建类实例或引用任何静态成员之前，静态构造函数被自动执行，并且

   只执行一次。

问题：

1.静态类可以实例化吗？静态类可以有非静态方法和变量吗?静态类的方法和变量怎么调用？

A:静态类不能实例化；不能有非静态方法和变量；对象名.方法名或变量名调用。

2.非静态类可以包含静态变量和方法吗？怎么调用？

A:非静态类可以包含静态变量和方法；类名.方法名或变量名调用

3.静态方法可以引用非静态变量吗？

4.实例类中的静态变量和非静态变量有什么区别？

A:静态变量只有一个副本，实例变量每实例化话一次都是一个新的变量，变量值都会初始化。 而静态变量变量的生存周期和网站运用程序生命周期长度一样。值会累加。

类都有默认构造函数。

## C++ 与C#对比学习

http://blog.csdn.net/weiwenhp/article/details/7967449

常见接口

IComparable, IFormattable, IConvertible, IComparable<long>, IEquatable<long>

## Const readonly

const 的概念就是一个包含不能修改的值的变量。  
常数表达式是在编译时可被完全计算的表达式。因此不能从一个变量中提取的值来初始化常量。  
如果 const int a = b+1;b是一个变量，显然不能再编译时就计算出结果，所以常量是不可以用变量来初始化的。  
  
readonly 允许把一个字段设置成常量，但可以执行一些运算，可以确定它的初始值。  
因为 readonly 是在计算时执行的，当然它可以用某些变量初始化。  
readonly 是实例成员，所以不同的实例可以有不同的常量值，这使readonly更灵活。  
  
readonly 关键字与 const 关键字不同。  
  
1. const 字段只能在该字段的声明中初始化。  
 readonly 字段可以在声明或[构造函数](https://www.baidu.com/s?wd=%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YzPWRYrycdP1wbmHPBPjfd0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnW6vP1D4rHmk)中初始化。因此，根据所使用的[构造函数](https://www.baidu.com/s?wd=%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YzPWRYrycdP1wbmHPBPjfd0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnW6vP1D4rHmk)，readonly 字段可能具有不同的值。  
2. const 字段是编译时常数，而 readonly 字段可用于运行时常数。  
3. const 默认就是静态的，而 readonly 如果设置成静态的就必须显示声明。  
4．const 对于引用类型的常数，可能的值只能是 string 和 null。  
 readonly可以是任何类型  
  
总结  
const只能在初期就使用常量初始化好。对于每一次编译后的结果，const的值是固定的，而readonly的值是可以在运行的时候才确定值的~~

## sealed 修饰符表示密封

用于类时，表示该类不能再被继承，不能和 abstract 同时使用，因为这两个修饰符在含义上互相排斥

用于方法和属性时，表示该方法或属性不能再被重写，必须和 override 关键字一起使用，因为使用 sealed 修饰符的方法或属性肯定是基类中相应的虚成员

通常用于实现第三方类库时不想被客户端继承，或用于没有必要再继承的类以防止滥用继承造成层次结构体系混乱

恰当的利用 sealed 修饰符也可以提高一定的运行效率，因为不用考虑继承类会重写该成员.

11月3日：

1. 类的实际存储静态为公用的地址，动态的成员创建新的地址。
2. 类中的方法仍可以看作是单独的方法，有访问空间限制。
3. 类可以看作是特殊的结构体，可以用初始化函数列表来实现。
4. 多接口可以实现多重继承，接口中的方法可以重写，并可以实现一个。要实现不同有相应的处理。

# **技术网站**