

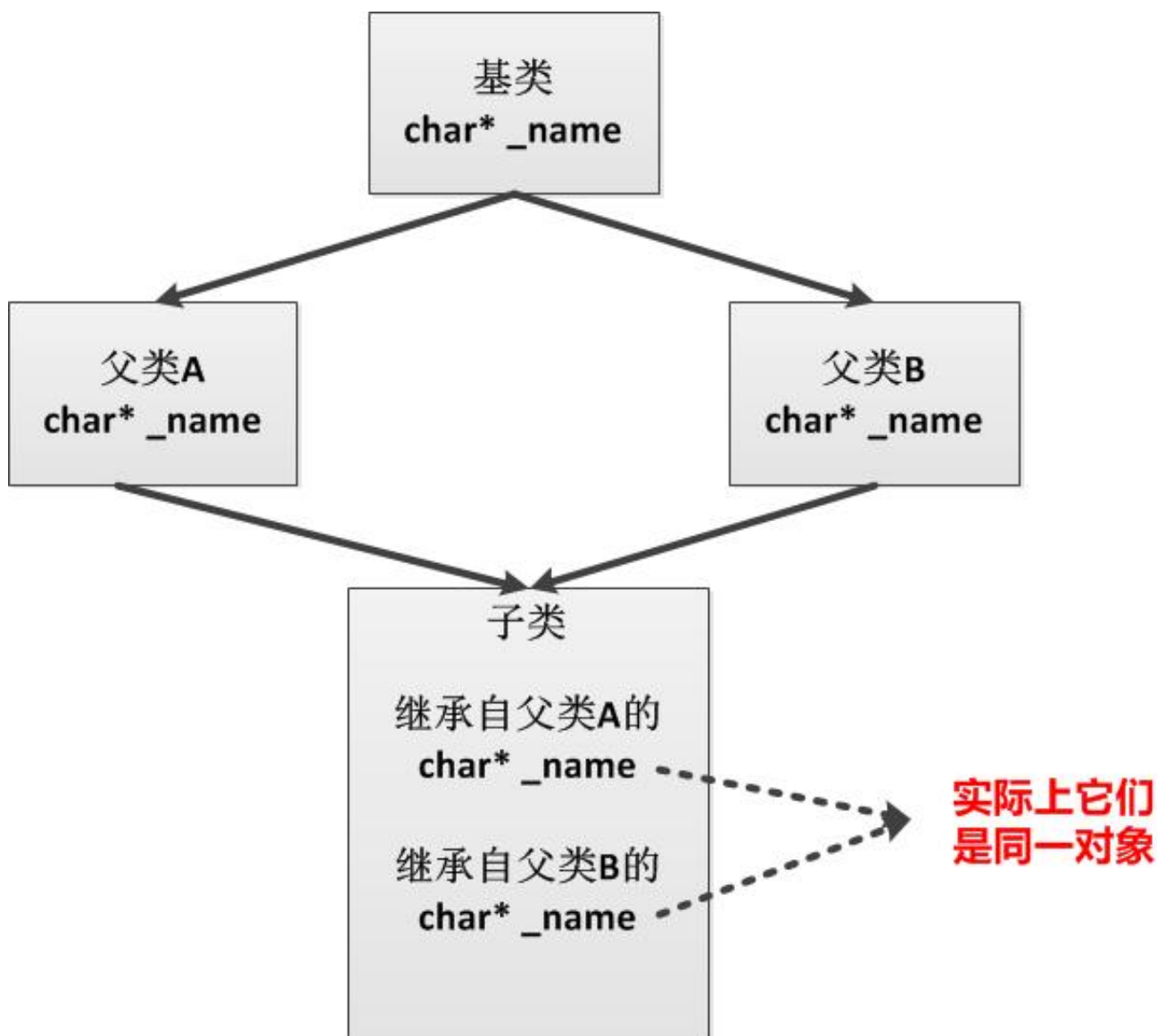
.NET基础拾遗（2）：面向对象的实现和异常的处理基础 - 文章 - 伯乐在线



一、面向对象的实现1.1 C#中的类可以多继承吗？

在C#中申明一个类型时，只支持单继承（即继承一个父类），但支持实现多个接口（Java也是如此）。像C++可能会支持同时继承自多个父类，但.NET的设计小组认为这样的机制会带来一些弊端，并且没有必要。

首先，看看多继承有啥好处？多继承的好处是更加贴近地设计类型。例如，当为一个图形编辑器设计带文本框的矩形类型时，最方便的方法可能是这个类型既继承自文本框类型，又继承自矩形类型，这样它就天生地具有输入文本和绘画矩形的功能。But，自从C++使用多继承依赖，就一直存在一些弊端，其中最为严重的还是所谓的“砖石继承”带来的问题，下图解释了砖石继承问题。



如上图所示，砖石继承问题根源在于最终的子类从不同的父类中继承到了在它看来完全不同的两个成员，而事实上，这两个成员又来自同一个基类。鉴于此，在C#/Java中，多继承的机制已经被彻底抛弃，取而代之的是单继承和多接口实现的机制。众所周知，接口并不做任何实际的工作，但是却制定了接口和规范，它定义了特定的类型都需要“做什么”，而把“怎么做”留给实现它的具体类型去考虑。也正是因为接口具有很大的灵活性和抽象性，因此它在面向对象的程序设计中更加出色地完成了抽象的工作。

1.2 C#中重写、重载和隐藏是什么鬼？

在C#或其他面向对象语言中，重写、重载和隐藏的机制，是设计高可扩展性的面向对象程序的基础。

（1）重写和隐藏

重写（Override）是指子类用Override关键字重新实现定义在基类中的虚方法，并且在实际运行时根据对象类型来调用相应的方法。

隐藏则是指子类用new关键字重新实现定义在基类中的方法，但在实际运行时只能根据引用来调用相应的方法。

以下的代码说明了重写和隐藏的机制以及它们的区别：

C#

```
public class Program
{
    public static void Main(string[] args)
    {
        // 测试二者的功能
        OverrideBase ob = new OverrideBase();
        NewBase nb = new NewBase();
        Console.WriteLine(ob.ToString() + ":" + ob.GetString());
        Console.WriteLine(nb.ToString() + ":" + nb.GetString());
        Console.WriteLine();
        // 测试二者的区别
        BaseClass obc = ob as BaseClass;
        BaseClass nbc = nb as BaseClass;
        Console.WriteLine(obc.ToString() + ":" + obc.GetString());
        Console.WriteLine(nbc.ToString() + ":" + nbc.GetString());
        Console.ReadKey();
    }
}

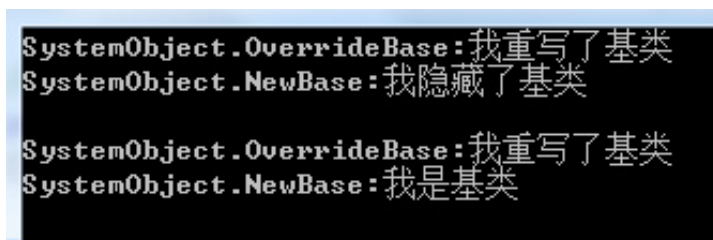
// Base class
public class BaseClass
{
    public virtual string GetString()
    {
```

```
        return "我是基类";
    }
}

// Override
public class OverrideBase : BaseClass
{
    public override string GetString()
    {
        return "我重写了基类";
    }
}

// Hide
public class NewBase : BaseClass
{
    public new virtual string GetString()
    {
        return "我隐藏了基类";
    }
}
```

以上代码的运行结果如下图所示：



```
SystemObject.OverrideBase: 我重写了基类
SystemObject.NewBase: 我隐藏了基类

SystemObject.OverrideBase: 我重写了基类
SystemObject.NewBase: 我是基类
```

我们可以看到：当通过基类的引用去调用对象内的方法时，重写仍然能够找到定义在对象真正类型中的GetString方法，而隐藏则只调用了基类中的GetString方法。

（2）重载

重载（Overload）是拥有相同名字和返回值的方法却拥有不同的参数列表，它是实现多态的立项方案，在实际开发中也是应用得最为广泛的。常见的重载应用包括：构造方法、ToString()方法等等；

以下代码是一个简单的重载示例：

```
C#
public class OverLoad
{
    private string text = "我是一个字符串";
    // 无参数版本
    public string PrintText()
    {
        return this.text;
    }
}
```

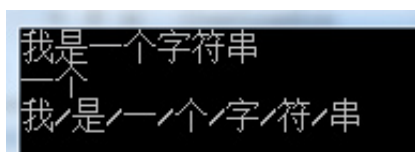
```

    }
    // 两个int参数的重载版本
    public string PrintText(int start, int end)
    {
        return this.text.Substring(start, end - start);
    }
    // 一个char参数的重载版本
    public string PrintText(char fill)
    {
        StringBuilder sb = new StringBuilder();
        foreach (var c in text)
        {
            sb.Append(c);
            sb.Append(fill);
        }
        sb.Remove(sb.Length - 1, 1);
        return sb.ToString();
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        OverLoad ol = new OverLoad();
        // 传入不同参数，PrintText的不同重载版本被调用
        Console.WriteLine(ol.PrintText());
        Console.WriteLine(ol.PrintText(2, 4));
        Console.WriteLine(ol.PrintText('/'));
        Console.ReadKey();
    }
}

```

运行结果如下图所示：



1.3 为什么不能在构造方法中调用虚方法？

在C#程序中，构造方法调用虚方法是一个需要避免的禁忌，这样做到底会导致什么异常？我们不妨通过下面一段代码来看看：

C#// 基类

```
public class A
```

```
{

    protected Ref my;
    public A()
    {
        my = new Ref();
        // 构造方法
        Console.WriteLine(ToString());
    }
    // 虚方法
    public override string ToString()
    {
        // 这里使用了内部成员my.str
        return my.str;
    }
}

// 子类
public class B : A
{
    private Ref my2;
    public B()
        : base()
    {
        my2 = new Ref();
    }
    // 重写虚方法
    public override string ToString()
    {
        // 这里使用了内部成员my2.str
        return my2.str;
    }
}

// 一个简单的引用类型
public class Ref
{
    public string str = "我是一个对象";
}

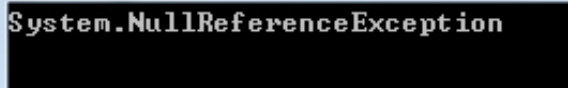
public class Program
{
    public static void Main(string[] args)
    {
        try
        {
```

```

        B b = new B();
    }
    catch (Exception ex)
    {
        // 输出异常信息
        Console.WriteLine(ex.GetType().ToString());
    }
    Console.ReadKey();
}
}

```

下面是运行结果，异常信息是空指针异常？



（1）要解释这个问题产生的原因，我们需要详细地了解一个带有基类的类型（事实上是 System.Object，所有的内建类型都有基类）被构造时，所有构造方法被调用的顺序。

在C#中，当一个类型被构造时，它的构造顺序是这样的：

执行变量的初始化表达式 → 执行父类的构造方法（需要的话）→ 调用类型自己的构造方法

我们可以通过以下代码示例来看看上面的构造顺序是如何体现的：

C#

```

public class Program
{
    public static void Main(string[] args)
    {
        // 构造了一个最底层的子类类型实例
        C newObj = new C();
        Console.ReadKey();
    }
}
// 基类类型
public class Base
{
    public Ref baseString = new Ref("Base 初始化表达式");
    public Base()
    {
        Console.WriteLine("Base 构造方法");
    }
}
// 继承基类

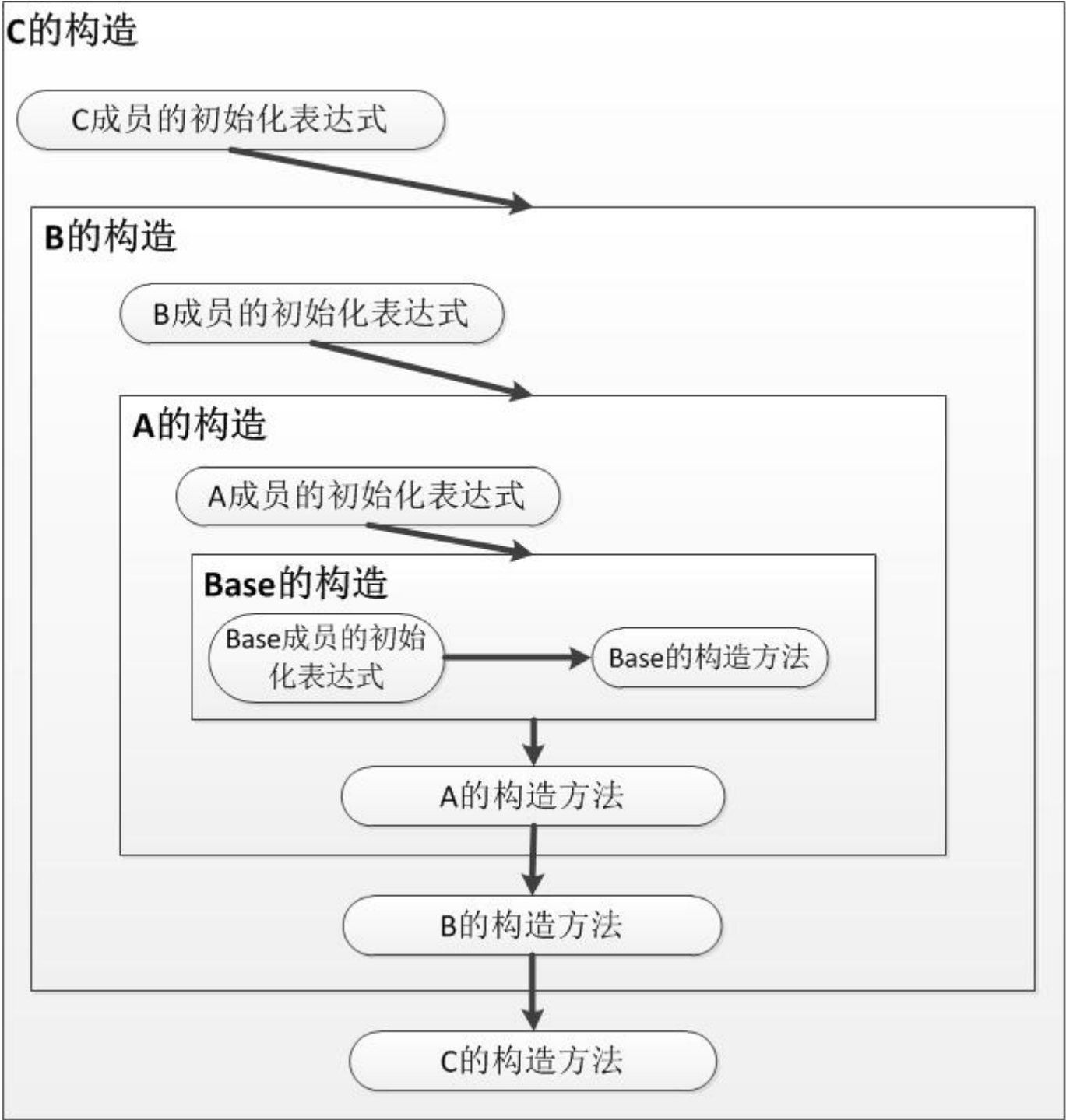
```

```
public class A : Base
{
    public Ref aString = new Ref("A 初始化表达式");
    public A()
        : base()
    {
        Console.WriteLine("A 构造方法");
    }
}
// 继承A
public class B : A
{
    public Ref bString = new Ref("B 初始化表达式");
    public B()
        : base()
    {
        Console.WriteLine("B 构造方法");
    }
}
// 继承B
public class C : B
{
    public Ref cString = new Ref("C 初始化表达式");
    public C()
        : base()
    {
        Console.WriteLine("C 构造方法");
    }
}
// 一个简单的引用类型
public class Ref
{
    public Ref(string str)
    {
        Console.WriteLine(str);
    }
}
```

调试运行，可以看到派生顺序是:Base → A → B → C，也验证了刚刚我们所提到的构造顺序。

```
C 初始化表达式
B 初始化表达式
A 初始化表达式
Base 初始化表达式
Base 构造方法
A 构造方法
B 构造方法
C 构造方法
```

上述代码的整个构造顺序如下图所示：



（2）了解完产生本问题的根本原因，反观虚方法的概念，当一个虚方法被调用时，CLR总是根据对象的实际类型来找到应该被调用的方法定义。换句话说，当虚方法在基类的构造方法中被调用时，它的类型让然保持的是子类，子类的虚方法将被执行，但是这时子类的构造方法却还没有完成，任何对子类未构造成员的访问都将产生异常。

如何避免这类问题呢？其根本方法就在于：永远不要在非叶子类的构造方法中调用虚方法。

1.4 C#如何声明一个类不能被继承？

这是一个被问烂的问题，在C#中可以通过sealed关键字来申明一个不可被继承的类，C#将在编译阶段保证这一机制。但是，继承式OO思想中最重要的一环，但是否想过继承也存在一些问题呢？在设计一个会被继承的类型时，往往需要考虑再三，下面例举了常见的一些类型被继承时容易产生问题：

- （1）为了让派生类型可以顺利地序列化，非叶子类需要实现恰当的序列化方法；
- （2）当非叶子类实现了ICloneable等接口时，意味着所有的子类都被迫需要实现接口中定义的方法；
- （3）非叶子类的构造方法不能调用虚方法，而且更容易产生不能预计的问题；

鉴于以上问题，在某些时候没有派生需要的类型都应该被显式地添加sealed关键字，这是避免继承带来不可预计问题的最有效办法。

二、异常的处理

2.1 如何针对不同的异常进行捕捉？

相信阅读本文的园友都已经养成了try-catch的习惯，但对于异常的捕捉和处理可能并不在意。确实，直接捕捉所有异常的基类：Exception 使得程序方便易懂，但有时这样的捕捉对于业务处理没有任何帮助，对于特殊异常应该采用特殊处理能够更好地引导规划程序流程。

下面的代码演示了一个对于不同异常进行处理的示例：

C#

```
public class Program
{
    public static void Main(string[] args)
    {
        Program p = new Program();
        p.RiskWork();
        Console.ReadKey();
    }
    public void RiskWork()
    {
        try
        {
            // 一些可能会出现异常的代码
        }
        catch (NullReferenceException ex)
        {
            HandleExpectedException(ex);
        }
        catch (ArgumentException ex)
        {
            HandleExpectedException(ex);
        }
    }
}
```

```
    }  
    catch (FileNotFoundException ex)  
    {  
        HandlerError(ex);  
    }  
    catch (Exception ex)  
    {  
        HandleCrash(ex);  
    }  
}  
// 这里处理预计可能会发生的，不属于错误范畴的异常  
private void HandleExpectedException(Exception ex)  
{  
    // 这里可以借助log4net写入日志  
    Console.WriteLine(ex.Message);  
}  
// 这里处理在系统出错时可能会发生的，比较严重的异常  
private void HandlerError(Exception ex)  
{  
    // 这里可以借助log4net写入日志  
    Console.WriteLine(ex.Message);  
    // 严重的异常需要抛到上层处理  
    throw ex;  
}  
// 这里处理可能会导致系统崩溃时的异常  
private void HandleCrash(Exception ex)  
{  
    // 这里可以借助log4net写入日志  
    Console.WriteLine(ex.Message);  
    // 关闭当前程序  
    System.Threading.Thread.CurrentThread.Abort();  
}  
}
```

(1) 如代码所示，针对特定的异常进行不同的捕捉通常很有意义，真正的系统往往要针对不同异常进行复杂的处理。异常的分别处理是一种好的编码习惯，这要求程序员在编写代码的时候充分估计到所有可能出现异常的情况，当然，无论考虑得如何周到，最后都需要对异常的基类Exception进行捕捉，这样才能保证所有的异常都不会被随意地抛出。

(2) 除此之外，除了在必要的时候写try-catch，很多园友更推荐使用框架层面提供的异常捕捉方案，以.NET为例：

- WinForm，可以这样写：`AppDomain.CurrentDomain.UnhandledException += new UnhandledExceptionHandler(UnhandledExceptionHandler);`

- ASP.NET WebForm, 可以在Application_Error()方法里捕获异常
- ASP.NET MVC, 可以写ExceptionHandler
- ASP.NET WebAPI, 可以写ExceptionHandler

2.2 如何使用Conditional特性？

大家都知道，通常在编译程序时可以选择Debug版本还是Release版本，编译器将会根据”调试“和”发布“两个不同的出发点去编译程序。在Debug版本中，所有Debug类的断言（Assert）语句都会得到保留，相反在Release版本中，则会被通通删除。这样的机制有助于我们编写出方便调试同时又不影响正式发布的程序代码。

But，单纯的诊断和断言可能并不能完全满足测试的需求，有时可能会需要大批的代码和方法去支持调试和测试，这个时候就需要用到Conditional特性。Conditional特性用于编写在某个特定版本中运行的方法，通常它编写一些在Debug版本中支持测试的方法。当版本不匹配时，编译器会把Conditional特性的方法内容置为空。

下面的一段代码演示了Conditional特性的使用：

C#//含有两个成员，生日和身份证

```
//身份证的第6位到第14位必须是生日
//身份证必须是18位
public class People
{
    private DateTime _birthday;
    private String _id;
    public DateTime Birthday
    {
        set
        {
            _birthday = value;
            if (!Check())
                throw new ArgumentException();
        }
        get
        {
            Debug();
            return _birthday;
        }
    }
    public String ID
    {
        set
        {
            _id = value;
        }
    }
}
```

```
        if (!Check())
            throw new ArgumentException();
    }

    get
    {
        Debug();
        return _id;
    }
}

public People(String id, DateTime birthday)
{
    _id = id;
    _birthday = birthday;
    Check();
    Debug();
    Console.WriteLine("People实例被构造了...");
}

// 只希望在DEBUG版本中出现
[Conditional("DEBUG")]
protected void Debug()
{
    Console.WriteLine(_birthday.ToString("yyyy-MM-dd"));
    Console.WriteLine(_id);
}

//检查是否符合业务逻辑
//在所有版本中都需要
protected bool Check()
{
    if (_id.Length != 18 ||
        _id.Substring(6, 8) != _birthday.ToString("yyyyMMdd"))
        return false;

    return true;
}

}

public class Program
{
    public static void Main(string[] args)
    {
        try
        {
            People p = new People("513001198811290215", new
DateTime(1988, 11, 29));
```

```
        p.ID = "513001198811290215";  
    }  
    catch (ArgumentException ex)  
    {  
        Console.WriteLine(ex.GetType().ToString());  
    }  
    Console.ReadKey();  
}  
}
```

下图则展示了上述代码在Debug版本和Release版本中的输出结果：

①Debug版本：

A screenshot of a console window with a black background and white text. The text is displayed on three lines: "1988-11-29", "513001198811290215", and "People实例被构造了...".

②Release版本：

A screenshot of a console window with a black background and white text. The text is displayed on one line: "People实例被构造了...".

Conditional机制很简单，在编译的时候编译器会查看编译状态和Conditional特性的参数，如果两者匹配，则正常编译。否则，编译器将简单地移除方法内的所有内容。

2.3 如何避免类型转换时的异常？

我们经常会面临一些类型转换的工作，其中有些是确定可以转换的（比如将一个子类类型转为父类类型），而有些则是尝试性的（比如将基类引用的对象转换成子类）。当执行常识性转换时，我们就应该做好捕捉异常的准备。

当一个不正确的类型转换发生时，会产生InvalidCastException异常，有时我们会用try-catch块做一些尝试性的类型转换，这样的代码没有任何错误，但是性能却相当糟糕，为什么呢？异常是一种耗费资源的机制，每当异常被抛出时，异常堆栈将会被建立，异常信息将被加载，而通常这些工作的成本相对较高，并且在尝试性类型转换时，这些信息都没有意义。

So，在.NET中提供了另外一种语法来进行尝试性的类型转换，那就是关键字 `is` 和 `as` 所做的工作。

（1）`is` 只负责检查类型的兼容性，并返回结果：`true` 和 `false`。→ 进行类型判断

C#

```
public static void Main(string[] args)  
{  
    object o = new object();  
    // 执行类型兼容性检查  
    ISample sample = o as ISample;
```

```
        if(sample != null)
        {
            sample.SampleShow();
        }
        Console.ReadKey();
    }
```

两者的共同之处都在于：不会抛出异常！综上所述，`as` 较 `is` 在执行效率上会好一些，在实际开发中应该量才而用，在只进行类型判断的应用场景时，应该多使用 `is` 而不是 `as`。

参考资料

- (1) 朱毅，《进入IT企业必读的200个.NET面试题》
- (2) 张子阳，《.NET之美：.NET关键技术深入解析》
- (3) 王涛，《你必须知道的.NET》

加入伯乐在线专栏作者。扩大知名度，还能得赞赏！详见《[招募专栏作者](#)》

1 赞 2 收藏 [评论](#)



合作联系

Email: bd@jobbole.com

QQ: 2302462408 （加好友请注明来意）

更多频道

- [小组](#) - 好的话题、有启发的回复、值得信赖的圈子
- [头条](#) - 分享和发现有价值的内容与观点
- [相亲](#) - 为IT单身男女服务的征婚传播平台
- [资源](#) - 优秀的工具资源导航
- [翻译](#) - 翻译传播优秀的外文文章
- [文章](#) - 国内外的精选文章
- [设计](#) - UI, 网页, 交互和用户体验
- [iOS](#) - 专注iOS技术分享
- [安卓](#) - 专注Android技术分享
- [前端](#) - JavaScript, HTML5, CSS
- [Java](#) - 专注Java技术分享
- [Python](#) - 专注Python技术分享