目录

[浅谈C# 多态的魅力（虚方法，抽象，接口实现） 2](#_Toc224980264)

[1虚函数多态 2](#_Toc1672078328)

[2接口多态 6](#_Toc649454054)

[3使用示例 9](#_Toc1862391524)

[C# 中的多态性 13](#_Toc1640188843)

[1多态的概念 13](#_Toc1553791409)

[1.1重载(overload) 13](#_Toc1171063543)

[1.2重写(override) 13](#_Toc377342446)

[1.3虚方法 14](#_Toc475280331)

[1.4抽象方法 14](#_Toc1544839924)

[1.5隐藏方法 15](#_Toc1047310438)

[2 示例 15](#_Toc1370560654)

[1、重载 17](#_Toc1103314056)

[2、重写和虚方法 17](#_Toc2025530994)

[3、抽象方法 18](#_Toc1188643914)

[4、隐藏方法 18](#_Toc1645378204)

[3深入理解多态性 18](#_Toc724552209)

# 浅谈C# 多态的魅力（虚方法，抽象，接口实现）

前言：我们都知道面向对象的三大特性：封装，继承，多态。封装和继承对于初学者而言比较好理解，但要理解多态，尤其是深入理解，初学者往往存在有很多困惑，为什么这样就可以？有时候感觉很不可思议，由此，面向对象的魅力体现了出来，那就是多态，多态用的好，可以提高程序的扩展性。常用的设计模式，比如简单工厂设计模式，核心就是多态。

其实多态就是：**允许将子类类型的指针赋值给父类类型的指针。**也就是同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。在运行时，可以通过指向基类的指针，来调用实现**派生类**中的方法。如果这边不理解可以先放一放，先看下面的事例，看完之后再来理解这句话，就很容易懂了。  
理解多态之前首先要对面向对象的里氏替换原则和开放封闭原则有所了解。

里氏替换原则（Liskov Substitution Principle）：派生类（子类）对象能够替换其基类（超类）对象被使用。通俗一点的理解就是“子类是父类”，举个例子，“男人是人，人不一定是男人”，当需要一个父类类型的对象的时候可以给一个子类类型的对象；当需要一个子类类型对象的时候给一个父类类型对象是不可以的！

开放封闭原则（Open Closed Principle）：封装变化、降低耦合，软件实体应该是可扩展，而不可修改的。也就是说，对扩展是开放的，而对修改是封闭的。因此，开放封闭原则主要体现在两个方面：对扩展开放，意味着有新的需求或变化时，可以对现有代码进行扩展，以适应新的情况。对修改封闭，意味着类一旦设计完成，就可以独立完成其工作，而不要对类进行任何修改。

对这两个原则有一定了解之后就能更好的理解多态。

## 1虚函数多态

**首先，我们先来看下怎样用虚方法实现多态**

我们都知道，喜鹊（Magpie）、老鹰（Eagle）、企鹅（Penguin）都是属于鸟类，我们可以根据这三者的共有特性提取出鸟类（Bird）做为父类，喜鹊喜欢吃虫子，老鹰喜欢吃肉，企鹅喜欢吃鱼。

创建基类Bird如下，添加一个虚方法Eat():

/// <summary>

/// 鸟类：父类

/// </summary>

public class Bird

{

/// <summary>

/// 吃：虚方法

/// </summary>

public virtual void Eat()

{

Console.WriteLine("我是一只小小鸟，我喜欢吃虫子~");

}

}

创建子类Magpie如下，继承父类Bird，重写父类Bird中的虚方法Eat()：

/// <summary>

/// 喜鹊：子类

/// </summary>

public class Magpie:Bird

{

/// <summary>

/// 重写父类中Eat方法

/// </summary>

public override void Eat()

{

Console.WriteLine("我是一只喜鹊，我喜欢吃虫子~");

}

}

创建一个子类Eagle如下，继承父类Bird，重写父类Bird中的虚方法Eat()：

/// <summary>

/// 老鹰：子类

/// </summary>

public class Eagle:Bird

{

/// <summary>

/// 重写父类中Eat方法

/// </summary>

public override void Eat()

{

Console.WriteLine("我是一只老鹰，我喜欢吃肉~");

}

}

创建一个子类Penguin如下，继承父类Bird，重写父类Bird中的虚方法Eat()：

/// <summary>

/// 企鹅：子类

/// </summary>

public class Penguin:Bird

{

/// <summary>

/// 重写父类中Eat方法

/// </summary>

public override void Eat()

{

Console.WriteLine("我是一只小企鹅，我喜欢吃鱼~");

}

}

到此，一个基类，三个子类已经创建完毕，接下来我们在主函数中来看下多态是怎样体现的。

static void Main(string[] args)

{

//创建一个Bird基类数组，添加基类Bird对象，Magpie对象，Eagle对象，Penguin对象

Bird[] birds = {

new Bird(),

new Magpie(),

new Eagle(),

new Penguin()

};

//遍历一下birds数组

foreach (Bird bird in birds)

{

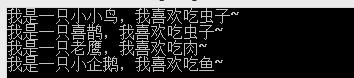
bird.Eat();

}

Console.ReadKey();

}

运行结果：



由此可见，子类Magpie，Eagle，Penguin对象可以赋值给父类对象，也就是说父类类型指针可以指向子类类型对象，这里体现了里氏替换原则。

父类对象调用自己的Eat()方法，实际上显示的是父类类型指针指向的子类类型对象重写父类Eat后的方法。这就是多态。

多态的作用到底是什么呢？

其实多态的作用就是把不同的子类对象都当作父类来看，可以屏蔽不同子类对象之间的差异，写出通用的代码，做出通用的编程，以适应需求的不断变化。  
以上程序也体现了开放封闭原则，如果后面的同事需要扩展我这个程序，还想再添加一个猫头鹰（Owl），很容易，只需要添加一个Owl类文件，继承Bird，重写Eat()方法，添加给父类对象就可以了。至此，该程序的扩展性得到了提升，而又不需要查看源代码是如何实现的就可以扩展新功能。这就是多态带来的好处。

**我们再来看下利用抽象如何来实现多态**

还是刚才的例子，我们发现Bird这个父类，我们根本不需要使用它创建的对象，它存在的意义就是供子类来继承。所以我们可以用抽象类来优化它。  
我们把Bird父类改成抽象类，Eat()方法改成抽象方法。代码如下：

/// <summary>

/// 鸟类：基类

/// </summary>

public abstract class Bird

{

/// <summary>

/// 吃：抽象方法

/// </summary>

public abstract void Eat();

}

抽象类Bird内添加一个Eat()抽象方法，没有方法体。也不能实例化。  
其他类Magpie，Eagle，Penguin代码不变，子类也是用override关键字来重写父类中抽象方法。  
Main主函数中Bird就不能创建对象了，代码稍微修改如下：

static void Main(string[] args)

{

//创建一个Bird基类数组，添加 Magpie，Eagle，Penguin对象

Bird[] birds = {

new Magpie(),

new Eagle(),

new Penguin()

};

//遍历一下birds数组

foreach (Bird bird in birds)

{

bird.Eat();

}

Console.ReadKey();

}

执行结果：

IMG_274

由此可见，我们选择使用虚方法实现多态还是抽象类抽象方法实现多态，取决于我们是否需要使用基类实例化的对象.

比如说 现在有一个Employee类作为基类，ProjectManager类继承自Employee，这个时候我们就需要使用虚方法来实现多态了，因为我们要使用Employee创建的对象，这些对象就是普通员工对象。

再比如说 现在有一个Person类作为基类，Student，Teacher 类继承Person，我们需要使用的是Student和Teacher创建的对象，根本不需要使用Person创建的对象，

所以在这里Person完全可以写成抽象类。

总而言之，是使用虚方法，或者抽象类抽象方法实现多态，视情况而定，什么情况？以上我说的两点~

接下来~~~~

我要问一个问题，喜鹊和老鹰都可以飞，这个飞的能力，我怎么来实现呢？

XXX答：“在父类Bird中添加一个Fly方法不就好了~~”

我再问：“好的，照你说的，企鹅继承父类Bird，但是不能企鹅不能飞啊，这样在父类Bird中添加Fly方法是不是不合适呢？”

XXX答：“那就在能飞的鸟类中分别添加Fly方法不就可以了吗？”

对，这样是可以，功能完全可以实现，可是这样违背了面向对象开放封闭原则，下次我要再扩展一个鸟类比如猫头鹰（Owl），我还要去源代码中看下Fly是怎么实现的，然后在Owl中再次添加Fly方法，相同的功能，重复的代码，这样是不合理的，程序也不便于扩展；

其次，如果我还要添加一个飞机类（Plane），我继承Bird父类，合适吗？

很显然，不合适！所以我们需要一种规则，那就是接口了，喜鹊，老鹰，飞机，我都实现这个接口，那就可以飞了，而企鹅我不实现这个接口，它就不能飞~~

## 2接口多态

**好，接下来介绍一下接口如何实现多态~**

添加一个接口IFlyable，代码如下：

/// <summary>

/// 飞 接口

/// </summary>

public interface IFlyable

{

void Fly();

}

喜鹊Magpie实现IFlyable接口，代码如下：

/// <summary>

/// 喜鹊：子类，实现IFlyable接口

/// </summary>

public class Magpie:Bird,IFlyable

{

/// <summary>

/// 重写父类Bird中Eat方法

/// </summary>

public override void Eat()

{

Console.WriteLine("我是一只喜鹊，我喜欢吃虫子~");

}

/// <summary>

/// 实现 IFlyable接口方法

/// </summary>

public void Fly()

{

Console.WriteLine("我是一只喜鹊，我可以飞哦~~");

}

}

老鹰Eagle实现IFlyable接口，代码如下：

/// <summary>

/// 老鹰：子类实现飞接口

/// </summary>

public class Eagle:Bird,IFlyable

{

/// <summary>

/// 重写父类Bird中Eat方法

/// </summary>

public override void Eat()

{

Console.WriteLine("我是一只老鹰，我喜欢吃肉~");

}

/// <summary>

/// 实现 IFlyable接口方法

/// </summary>

public void Fly()

{

Console.WriteLine("我是一只老鹰，我可以飞哦~~");

}

}

在Main主函数中，创建一个IFlyable接口数组，代码实现如下：

static void Main(string[] args)

{

//创建一个IFlyable接口数组，添加 Magpie对象，Eagle对象

IFlyable[] flys = {

new Magpie(),

new Eagle()

};

//遍历一下flys数组

foreach (IFlyable fly in flys)

{

fly.Fly();

}

Console.ReadKey();

}

执行结果：

IMG_283

由于企鹅Penguin没有实现IFlyable接口，所以企鹅不能对象不能赋值给IFlyable接口对象，所以企鹅，不能飞~

好了，刚才我提到了飞机也能飞，继承Bird不合适的问题，现在有了接口，这个问题也可以解决了。如下，我添加一个飞机Plane类，实现IFlyable接口，代码如下：

/// <summary>

/// 飞机类，实现IFlyable接口

/// </summary>

public class Plane:IFlyable

{

/// <summary>

/// 实现接口方法

/// </summary>

public void Fly()

{

Console.WriteLine("我是一架飞机，我也能飞~~");

}

}

在Main主函数中，接口IFlyable数组，添加Plane对象：

class Program

{

static void Main(string[] args)

{

//创建一个IFlyable接口数组，添加 Magpie，Eagle，Plane对象

IFlyable[] flys = {

new Magpie(),

new Eagle(),

new Plane()

};

//遍历一下flys数组

foreach (IFlyable fly in flys)

{

fly.Fly();

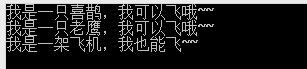
}

Console.ReadKey();

}

}

执行结果：



由此，可以看出用接口实现多态程序的扩展性得到了大大提升，以后不管是再扩展一个蝴蝶（Butterfly），还是鸟人（Birder）创建一个类，实现这个接口，在主函数中添加该对象就可以了。  
也不需要查看源代码是如何实现的，体现了开放封闭原则！

接口充分体现了多态的魅力~~

## 3使用示例

以上通过一些小的事例，给大家介绍了面向对象中三种实现多态的方式，或许有人会问，在项目中怎么使用多态呢？多态的魅力在项目中如何体现？  
那么接下来我做一个面向对象的简单计算器，来Show一下多态在项目中使用吧！

加减乘除运算，我们可以根据共性提取出一个计算类，里面包含两个属性 Number1和Number2，还有一个抽象方法Compute();代码如下：

/// <summary>

/// 计算父类

/// </summary>

public abstract class Calculate

{

public int Number1

{

get;

set;

}

public int Number2

{

get;

set;

}

public abstract int Compute();

}

接下来，我们添加一个加法器，继承计算Calculate父类：

/// <summary>

/// 加法器

/// </summary>

public class Addition : Calculate

{

/// <summary>

/// 实现父类计算方法

/// </summary>

/// <returns>加法计算结果</returns>

public override int Compute()

{

return Number1 + Number2;

}

}

再添加一个减法器，继承计算Calculate父类：

/// <summary>

/// 减法器

/// </summary>

public class Subtraction : Calculate

{

/// <summary>

/// 实现父类计算方法

/// </summary>

/// <returns>减法计算结果</returns>

public override int Compute()

{

return Number1 - Number2;

}

}

在主窗体FormMain中，编写计算事件btn\_Compute\_Click，代码如下：

private void btn\_Compute\_Click(object sender, EventArgs e)

{

//获取两个参数

int number1 = Convert.ToInt32(this.txt\_Number1.Text.Trim());

int number2 = Convert.ToInt32(this.txt\_Number2.Text.Trim());

//获取运算符

string operation = cbb\_Operator.Text.Trim();

//通过运算符，返回父类类型

Calculate calculate = GetCalculateResult(operation);

calculate.Number1 = number1;

calculate.Number2 = number2;

//利用多态，返回运算结果

string result = calculate.Compute().ToString();

this.lab\_Result.Text = result;

}

/// <summary>

/// 通过运算符，返回父类类型

/// </summary>

/// <param name="operation"></param>

/// <returns></returns>

private Calculate GetCalculateResult(string operation)

{

Calculate calculate = null;

switch (operation)

{

case "+":

calculate = new Addition();

break;

case "-":

calculate = new Subtraction();

break;

}

return calculate;

}

在该事件中主要调用GetCalculateResult方法，通过运算符，创建一个对应的加减乘除计算器子类，然后赋值给父类，其实这就是设计模式中的简单工厂设计模式，我给你一个运算符你给我生产一个对应的加减乘除计算器子类，返回给我。。其实大多数的设计模式的核心就是多态，掌握好多态，设计模式看起来也很轻松。

现阶段工作已经完成，但是过了一段时间，又添加新的需求了，我还要扩展一个乘法了，那好，很简单只要创建一个乘法计算器继承Calculate父类即可，看代码：

/// <summary>

/// 乘法计算器

/// </summary>

public class Multiplication:Calculate

{

public override int Compute()

{

return Number1\*Number2;

}

}

然后在GetCalculateResult函数中添加一个case 就好了：

switch (operation)

{

case "+":

calculate = new Addition();

break;

case "-":

calculate = new Subtraction();

break;

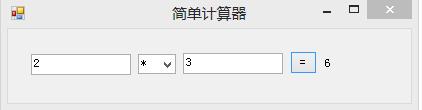
case "\*":

calculate = new Multiplication();

break;

}

执行结果：



好了，就这么方便，一个新的功能就扩展完毕了，我根本不需要查看源代码是如何实现的，这就是多态的好处！

# C# 中的多态性

相信大家都对面向对象的三个特征封装、继承、多态很熟悉，每个人都能说上一两句，但是大多数都仅仅是知道这些是什么，不知道 CLR 内部是如何实现的，所以本篇文章主要说说多态性中的一些概念已经内部实现的机理。

## 1多态的概念

首先解释下什么叫多态：同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果，这就是多态性。换句话说，实际上就是同一个类型的实例调用"相同"的方法，产生的结果是不同的。这里的"相同"打上双引号是因为这里的相同的方法仅仅是看上去相同的方法，实际上它们调用的方法是不同的。

说到多态，我们不能免俗的提到下面几个概念:重载、重写、虚方法、抽象方法以及隐藏方法。下面就来一一介绍他们的概念。

### 1.1重载(overload)

在同一个作用域(一般指一个类)的两个或多个方法函数名相同，参数列表不同的方法叫做重载，它们有三个特点(俗称两必须一可以):

* 方法名必须相同
* 参数列表必须不相同
* 返回值类型可以不相同

例如：

public void Sleep(){

Console.WriteLine("Animal睡觉");}public int Sleep(int time){

Console.WriteLine("Animal{0}点睡觉", time);

return time;}

### 1.2重写(override)

子类中为满足自己的需要来重复定义某个方法的不同实现，需要用 override 关键字，被重写的方法必须是虚方法，用的是 virtual 关键字。它的特点是(三个相同):

* 相同的方法名
* 相同的参数列表
* 相同的返回值

如：父类中的定义：

public virtual void EatFood(){

Console.WriteLine("Animal吃东西");}

子类中的定义：

public override void EatFood(){

Console.WriteLine("Cat吃东西");

//base.EatFood();}

**小提示：**经常有童鞋问重载和重写的区别,而且网络上把这两个的区别作为 C# 做常考的面试题之一。实际上这两个概念完全没有关系，仅仅都带有一个"重"字。他们没有在一起比较的意义，仅仅分辨它们不同的定义就好了。

### 1.3虚方法

即为基类中定义的允许在派生类中重写的方法，使用virtual关键字定义。如：

public virtual void EatFood(){

    Console.WriteLine("Animal吃东西");

}

注意：虚方法也可以被直接调用。如：

Animal a = new Animal(); a.EatFood();

执行输出结果为：Animal吃东西

### 1.4抽象方法

在基类中定义的并且必须在派生类中重写的方法，使用 abstract 关键字定义。如：

public abstract class Biology{

public abstract void Live();}public class Animal : Biology{

public override void Live()

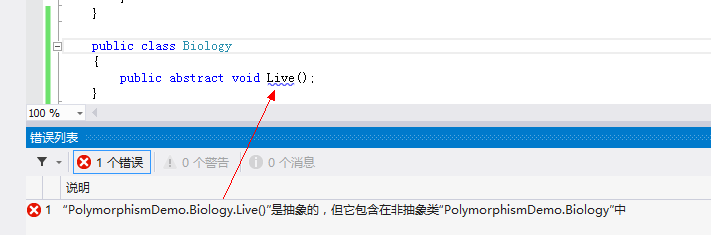
{

Console.WriteLine("Animal重写的抽象方法");

//throw new NotImplementedException();

} }

**注意：**抽象方法只能在抽象类中定义，如果不在抽象类中定义，则会报出如下错误：



**虚方法和抽象方法的区别是**：因为抽象类无法实例化，所以抽象方法没有办法被调用，也就是说抽象方法永远不可能被实现。

### 1.5隐藏方法

**在派生类中定义的和基类中的某个方法同名的方法，使用 new 关键字定义。如在基类 Animal 中有一方法 Sleep():**

public void Sleep(){

Console.WriteLine("Animal Sleep");}

则在派生类 Cat 中定义隐藏方法的代码为：

new public void Sleep(){

Console.WriteLine("Cat Sleep");}

或者为：

public new void Sleep(){

Console.WriteLine("Cat Sleep");}

**注意：**

（1）隐藏方法不但可以隐藏基类中的虚方法，而且也可以隐藏基类中的非虚方法。

（2）隐藏方法中父类的实例调用父类的方法，子类的实例调用子类的方法。

（3）和上一条对比：重写方法中子类的变量调用子类重写的方法，父类的变量要看这个父类引用的是子类的实例还是本身的实例，如果引用的是父类的实例那么调用基类的方法，如果引用的是派生类的实例则调用派生类的方法。

## 2 示例

好了，基本概念讲完了，下面来看一个例子，首先我们新建几个类：

public abstract class Biology{

public abstract void Live();}public class Animal : Biology{

public override void Live()

{

Console.WriteLine("Animal重写的Live");

//throw new NotImplementedException();

}

public void Sleep()

{

Console.WriteLine("Animal Sleep");

}

public int Sleep(int time)

{

Console.WriteLine("Animal在{0}点Sleep", time);

return time;

}

public virtual void EatFood()

{

Console.WriteLine("Animal EatFood");

}}public class Cat : Animal{

public override void EatFood()

{

Console.WriteLine("Cat EatFood");

//base.EatFood();

}

new public void Sleep()

{

Console.WriteLine("Cat Sleep");

}

//public new void Sleep()

//{

// Console.WriteLine("Cat Sleep");

//}}public class Dog : Animal{

public override void EatFood()

{

Console.WriteLine("Dog EatFood");

//base.EatFood();

}}

下面来看看需要执行的代码：

class Program{

static void Main(string[] args)

{

//Animal的实例

Animal a = new Animal();

//Animal的实例，引用派生类Cat对象

Animal ac = new Cat();

//Animal的实例，引用派生类Dog对象

Animal ad = new Dog();

//Cat的实例

Cat c = new Cat();

//Dog的实例

Dog d = new Dog();

//重载

a.Sleep();

a.Sleep(23);

//重写和虚方法

a.EatFood();

ac.EatFood();

ad.EatFood();

//抽象方法

a.Live();

//隐藏方法

a.Sleep();

ac.Sleep();

c.Sleep();

Console.ReadKey();

}}

首先，我们定义了几个我们需要使用的类的实例，需要注意的是:

(1)Biology类是抽象类，无法实例化；

(2)变量ac是Animal的实例，但是指向一个Cat的对象。因为Cat类型是Animal类型的派生类，所以这种转换没有问题。这也是多态性的重点。

下面我们来一步一步的分析：

### 1、重载

a.Sleep();

a.Sleep(23);

很明显，Animal 的变量 a 调用的两个 Sleep 方法是重载的方法，第一句调用的是无参数的 Sleep() 方法，第二句调用的是有一个 int 参数的 Sleep 方法。注意两个 Sleep 方法的返回值不一样，这也说明了重写的三个特征中的最后一个特征——返回值可以不相同。

运行的结果如下：

Animal SleepAnimal在23点Sleep

### 2、重写和虚方法

a.EatFood();

ac.EatFood();

ad.EatFood();

在这一段中，a、ac 以及 ad 都是 Animal 的实例，但是他们引用的对象不同，a 引用的是 Animal 对象，ac 引用的是 Cat 对象，ad 引用的是 Dog 对象，这个差别会造成执行结果的什么差别呢，请看执行结果：

Animal EatFoodCat EatFoodDog EatFood

第一句 Animal 实例，直接调用 Animal 的虚方法 EatFood，没有任何问题。

在第二、三句中，虽然同样是 Animal 的实例，但是他们分别指向 Cat 和 Dog 对象，所以调用的 Cat 类和 Dog 类中各自重写的 EatFood 方法，就像是 Cat 实例和 Dog 实例直接调用 EatFood 方法一样。这个也就是多态性的体现：同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。

### 3、抽象方法

a.Live();

这个比较简单，就是直接重写父类 Biology 中的 Live 方法，执行结果如下:

Animal重写的Live

### 4、隐藏方法

a.Sleep();

ac.Sleep();

c.Sleep();

在分析隐藏方法时要和虚方法、重写相互比较。变量 a 调用 Animal 类的 Sleep 方法以及变量 c 调用 Cat 类的 Sleep 方法没有异议，但是变量 ac 引用的是一个 Cat 类型的对象，它应该调用 Animal 类型的 EatFood 方法呢，还是 Cat 类型的 EatFood 方法呢？答案是调用父类即 Animal 的 EatFood 方法。执行结果如下：

Animal SleepAnimal SleepCat Sleep

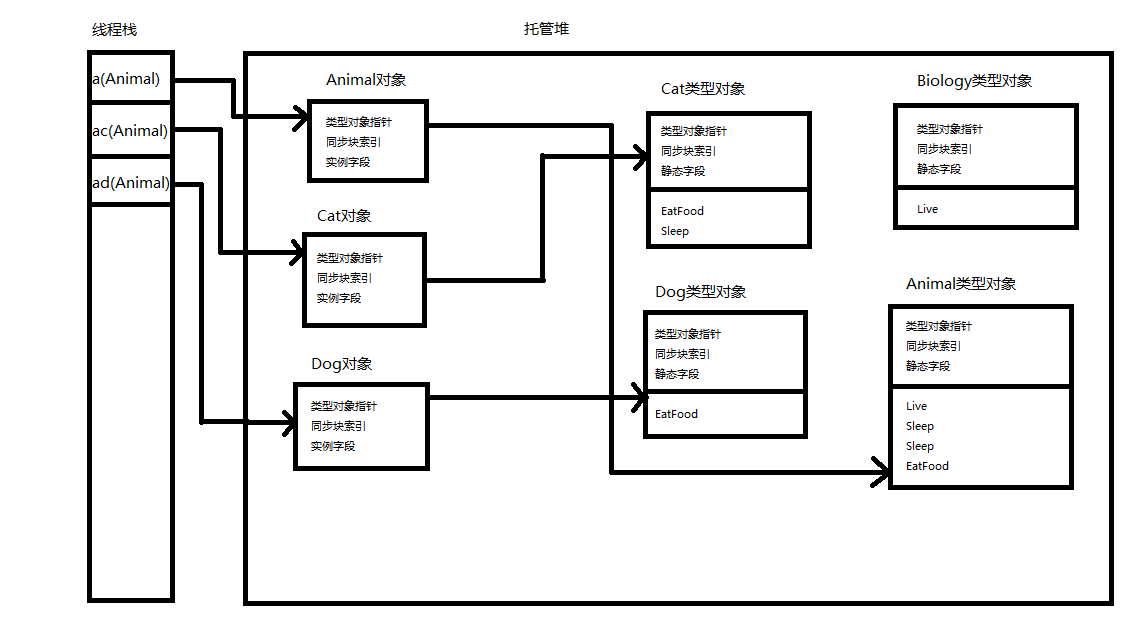
大多数的文章都是介绍到这里为止，仅仅是让我们知道这些概念以及调用的方法，而没有说明为什么会这样。下面我们就来深入一点，谈谈多态背后的机理。

## 3深入理解多态性

要深入理解多态性，就要先从值类型和引用类型说起。我们都知道值类型是保存在线程栈上的，而引用类型是保存在托管堆中的。因为所有的类都是引用类型，所以我们仅仅看引用类型。

现在回到刚才的例子，Main 函数时程序的入口，在 JIT 编译器将 Main 函数编译为本地CPU指定时，发现该方法引用了 Biology、Animal、Cat、Dog这几个类，所以 CLR 会创建几个实例来表示这几个类型本身，我们把它称之为"类型对象"。该对象包含了类中的静态字段，以及包含类中所有方法的方法表，还包含了托管堆中所有对象都要有的两个额外的成员——类型对象指针(Type Object Point)和同步块索引(sync Block Index)。

可能上面这段对于有些没有看过相关CLR书籍的童鞋没有看懂，所以我们画个图来描述一下：



上面的这个图是在执行 Main 函数之前 CLR 所做的事情，下面开始执行 Main 函数（方便起见，简化一下 Main 函数）：

//Animal的实例

Animal a = new Animal();

//Animal的实例，引用派生类Cat对象

Animal ac = new Cat();

//Animal的实例，引用派生类Dog对象

Animal ad = new Dog();

a.Sleep();

a.EatFood();

ac.EatFood();

ad.EatFood();

下面实例化三个 Animal 实例，但是他们实际上指向的分别是 Animal 对象、Cat 对象和 Dog 对象，如上图。

请注意，变量 ac 和 ad 虽然都是 Animal 类型，但是指向的分别是 Cat 对象和 Dog 对象，**这里是关键。**

当执行 a.Sleep() 时，由于 Sleep 是非虚实例方法，JIT 编译器会找到发出调用的那个变量（a）的类型（Animal）对应的类型对象（Animal 类型对象）。然后调用该类型对象中的 Sleep 方法，如果该类型对象没有 Sleep 方法，JIT 编译器会回溯类的基类（一直到 Object）中查找 Sleep 方法。

当执行 ac.EatFood 时，由于 EatFood 是**虚实例方法**，JIT 编译器调用时会在方法中生成一些额外的代码，这些代码会首先检查发出调用的变量（ac），然后跟随变量的引用地址找到发出调用的对象（Cat 对象），找到发出调用的对象对应的类型对象（Cat 类型对象），最后在该类型对象中查找 EatFood 方法。同样的，如果在该类型对象中没有查找到 EatFood 方法，JIT 编译器会回溯到该类型对象的基类中查找。

上面描述的就是 JIT 编译器在遇到调用类型的非虚实例方法以及虚实例方法时的不同执行方式，也这是处理这两类方法的不同方式造成了表面上我们看到的面向对象的三个特征之一——多态性。

原文地址：<https://www.cnblogs.com/zhangkai2237/archive/2012/12/20/2826734.html>