**委托与事件**

**16.1 委托**

事件处理程序是基于“委托”机制运行的。有时我们需要将一个函数作为另一个函数的参数，这时就要用到委托（Delegate）机制。委托类似于C++中的指针函数的作用，让委托函数名指向不同的函数，来实现不同的功能。

class Circus

{

public delegate void Display(string strMsg);

public static void DogDisplay(string strMsg)

{

*Console*.*WriteLine*("我是Snoopy，{0}！", strMsg);

}

public static void CatDisplay(string strMsg)

{

*Console*.*WriteLine*("我是Hellocaty，{0}！", strMsg);

}

public static void CircusStart(Display display,string msg)

{

display(msg);

}

}

class Program

{

static void Main(string[] args)

{

Circus.Display Dogdisplay = new Circus.Display(Circus.DogDisplay);

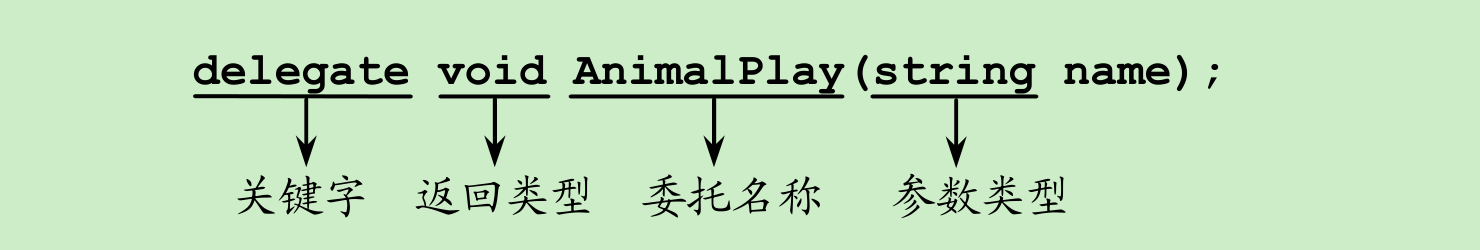
Circus.CircusStart(Dogdisplay,"good even!");

*Console*.*ReadKey*();

}

}

委托用关键字 delegate 声明：



在新版本的.NET 中，可以直接传递函数名称，省去把函数转换为委托实例的过程。

Circus.CircusStart(Circus.DogDisplay,"GOOD Morning!");

以下为一个委托在求定积分中的使用：

class DefiniteIntegral

{

//被积函数 F1(x)=2x+1

public static double F1(double d) { return 2 \* d + 1; }

public delegate double DelegateFunc(double x);

// 积分函数

public static double Definite(DelegateFunc delegateFunc, double dLow, double dUp)

{

double dRet = 0;

const int nCount = 1000;

double dDerta = (dUp - dLow) / nCount;

for (int i = 0; i < nCount; i++)

{

dRet += (delegateFunc(dLow + i \* dDerta)) \* dDerta;

}

return dRet;

}

}

class Program

{

static void Main(string[] args)

{

double dRet = DefiniteIntegral.Definite(DefiniteIntegral.F1, 1, 8);

*Console*.*WriteLine*("函数[2x+1]积分范围[{0},{1}],面积{2}",1,8, dRet);

*Console*.*ReadKey*();

}

}

**16.2 多播委托**

我们可以向一个委托中注册多个函数，这种包含多个函数的委托称为多播委托

（ Multicast Delegate ）。那些被委托的函数的引用都存储在多播委托的调用列表中，当调用多播委托时，会按一定顺序依次调用列表中的所有函数。

static void Main(string[] args)

{

Circus.Display play = new Circus.Display(Circus.DogDisplay);

play += new Circus.Display(Circus.CatDisplay);

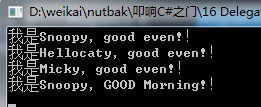
play += Circus.MiseDisplay;

Circus.CircusStart(play,"good even!");

Circus.CircusStart(Circus.DogDisplay,"GOOD Morning!");

*Console*.*ReadKey*();

}



从多播委托中注销函数的语法为：

通过 -=运算符从多播委托注销函数

多播委托的返回值一般为 void ，如果为非 void 类型，多播委托可能有多个返回值（因

为会调用多个函数），这时多播委托将返回最后一个方法的返回值。但实际中不推荐这样

应用，因为这些 方法的调用顺序未正式定义，因此应避免编写依赖于特定调用顺序的代码 。

**16.3 匿名函数**

创建委托实例时不仅可以使用已有的函数，而且可以直接使用匿名函数（ Anonymous

Function ）。

double dRet2 = DefiniteIntegral.Definite(delegate(double x){ return x \* x; }, 1, 8);

**16.4 Lam bda表达式**

Lambda 表达式是一种简洁的定义匿名函数的方法，用 Lambda 表达式构造的匿名函数看上去就像一个计算表达式，它使用“ => ”符号来连接参数和事件处理代码，极大的增强

了匿名函数的可读性。

DefiniteIntegral.DelegateFunc f3 = (double x) => { return x \* x; };

double dRet3 = DefiniteIntegral.Definite(f3, 1, 8);

**16.5 宽松委托**

一般情况下我们要求委托和被委托函数的签名完全一样，但从 C#3.0 开始，出现了更

为宽松的委托，被委托函数的参数类型可以比委托要求的更大、更宽泛，被委托函数的返

回值类型可以比委托要求的更小、更精确。

**总而言之，被委托函数的参数类型可以是委托参数的基类（逆变），被委托函数的返**

**回值类型可以是委托返回值的派生类（协变）。**

**16.6 事件处理机制**