C#的委托，匿名方法，Lambda表达式
参考书籍：《C#图解教程》

委托就像C++的函数指针一样，可以指向一个函数，但委托更牛逼，指的其实是一个函数列表，依次执行函数。

在C++中，匿名方法和Lambda表达式其实是一回事，C#里就是两回事了。

委托

委托的初始化

```
delegate void MyDel(int value);
//下面两句等价，SomeClass.SomeFunction是某个相同返回值，参数列表的函数。
MyDel del = new MyDel(SomeClass.SomeFunction);
MyDel del = SomeClass.SomeFunction;
```

组合委托 & 为委托添加方法 & 删除方法 & 委托调用

```
//可以直接使用 + 操作符将方法或委托组合在一起。委托允许重复，及执行多次。
MyDel del = delA + delA + delB + SomeClass.SomeFunction;

//或者使用 += 操作符添加方法或委托。
del += delC + SomeClass.SomeFunction;

//当要注意使用 += 操作符时，左值要先初始化。
MyDel del2;
del2 += del; //错误，del2没有初始化。

//删除用 -= 操作符。
del -= delA;
del = del - SomeClass.SomeFunction;

//调用,相当于给列表中所有方法给了666的参数。
del(666);
```

在使用 += 运算符时，实际发生的是创建一个新的委托，把左边的委托加上右边方法的组合，再赋值个左边的委托。

使用 -= 运算符时，如果匹配有多个相同的方法，会从列表最后向前搜索，删掉第一个匹配的方法的实例。

空委托用 -= 运算符会报异常。

匿名方法

匿名方法是要带delegate关键字的。

```
//正常的画风应该是这样的。
MyDel del = delegate(int value)
{
return value + 100;
}

//省略圆括号，但必须满足以下两个条件
//1.委托的参数列表里面不包含任何out参数。
//2.匿名方法不使用任何参数。
MyDel del = delegate
{
SomeFunction();
SomeFunction2(1,2,3,4);
}
del(666); //传进去的666对上面没有参数的匿名函数没有影响。

//params参数，如果委托有params参数，匿名方法的参数列表将忽略params关键字。
delegate void anotherDel(int X,params int[] Y);
anotherDel aDel = delegate(int X,int Y) { }; //省略了params
```

Lambda表达式

相比匿名方法，Lambda表达式看起来简单许多，直接上图。

```
MyDel del = delegate(int x)    { return x + 1; } ;     // 匿名方法
MyDel le1 =         (int x) => { return x + 1; } ;     // Lambda表达式
MyDel le2 =             (x) => { return x + 1; } ;     // Lambda表达式
MyDel le3 =              x  => { return x + 1; } ;     // Lambda表达式
MyDel le4 =              x  =>          x + 1    ;     // Lambda表达式
```