首先，我觉得，一个概念，如果不理解也不影响使用的话，那么，就没必要去理解它、去学习它。闭包就是这样一个概念，你不理解它也能很好的用它。俺这两年写as3程序，是天天在和它打交道，甚至有过一个function套一个，一个方法中套了20多个function的极端例子，但从未深究过它是怎么实现的，它就像水和空气一样，我们不需要知道水是H2O，空气是氧气氮气二氧化碳等的混合物，也活的好好的。

其次，我觉得，网上对闭包概念的解释都太狭隘了，看得人蛋疼，就像回到了i++,++i时代一样。如果非要去理解这个概念，像那样去理解，则收获太小，不值得。

维基百科上对闭包的解释就很经典：

在计算机科学中，闭包（Closure）是词法闭包（Lexical Closure）的简称，是引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。所以，有另一种说法认为闭包是由函数和与其相关的引用环境组合而成的实体。

Peter J. Landin 在1964年将术语闭包定义为一种包含环境成分和控制成分的实体。

下面是我理解的闭包概念。

先看看数学上的闭包。

(1,5) 是一个区间，但对这个区间做分析、计算什么的，经常会用到1和5这两个不属于这个区间的值，[1,5]就是(1,5)的闭包。

在生活上，我们办事情，找A部门，A部门说，你先得找B部门盖个章，B部门说，你先得找C部门盖个章，C部门说，这个东西不是我们的职权范围…… 踢皮球，这就是非闭包。闭包就是负责到底，你找到A部门，A部门接待的那个人负责到底，他/她去协调B部门和C部门。

在工程上，闭包就是项目经理，负责调度项目所需要的资源。老板、客户有什么事情，直接找项目经理即可，不用再去找其它的人。

在程序语言中，闭包就是一种语法糖，它以很自然的形式，把我们的目的和我们的目的所涉及的资源全给自动打包在一起，以某种自然、尽量不让人误解的方式让人来使用。至于其具体实现，我个人意见，在不影响使用的情况下，不求甚解即可。在很多情况下，需要在一段代码里去访问外部的局部变量，不提供这种语法糖，需要写非常多的代码，有了闭包这个语法糖，就不用写这么多代码，自然而然的就用了。

这样一来，可以把闭包从一个语法机制提升为一种设计原则：

闭包是从用户角度考虑的一种设计概念，它基于对上下文的分析，把龌龊的事情、复杂的事情和外部环境交互的事情都自己做了，留给用户一个很自然的接口。

在这个原则下，函数式语言中，那种所谓的闭包只是一种“闭包”，还有大量的其它类型的“闭包”等待发现和实现。

下面举出一些闭包设计原则的正例和反例。

正例：

Flex中的数据绑定语法就是一种“闭包”。x="{b.num + c.num}"，对于这个语法，编译器自动去上下文中寻找叫 b 和 c 的变量，然后再找他们内部 num 变量，如果他们都是可绑定的话，则自动给它们添加上绑定链，当 b, c, num 等有任一变动时，更新 x 的值。

反例：

Winform 中的设计就违反了闭包原则，当不是在该UI线程中，更新某些控件的值时，会抛出异常。只能去invoke调用，而invoke的接口很难用，相信很多人对这东东极其反感。

闭包不一定是语法糖。当我们不能直接扩展编译器时，我们就无法增加语法糖来实现闭包机制，这时，就要用现有的语言机制来实现了。

下面，我们来对winform的invoke方法进行改造，使它满足闭包原则。下面是代码：

    public class ControlFuncContext   
    {   
        public Control Control { get; private set; }   
        public Delegate Delegate { get; private set; }

        public ControlFuncContext(Control ctl, Delegate d)   
        {   
            this.Control = ctl;   
            this.Delegate = d;   
        }

        public void Invoke0()   
        {   
            if (Control.IsHandleCreated == true)   
            {   
                try   
                {   
                    Delegate.DynamicInvoke();   
                }   
                catch(ObjectDisposedException ex)   
                {   
                }   
            }   
        }

        public void Invoke1<T>(T obj)   
        {   
            if (Control.IsHandleCreated == true)   
            {   
                try   
                {   
                    Delegate.DynamicInvoke(obj);   
                }   
                catch (ObjectDisposedException ex)   
                {   
                }   
            }   
        }

        public void Invoke2<T0,T1>(T0 obj0, T1 obj1)   
        {   
            if (Control.IsHandleCreated == true)   
            {   
                try   
                {   
                    Delegate.DynamicInvoke(obj0, obj1);   
                }   
                catch (ObjectDisposedException ex)   
                {   
                }   
            }   
        }   
    }

    public static class FormClassHelper   
    {

        public static void InvokeAction(this Control ctl, Action action)   
        {   
            if (ctl.IsHandleCreated == true)   
            {   
                ControlFuncContext fc = new ControlFuncContext(ctl, action);   
                ctl.Invoke(new Action(fc.Invoke0));   
            }   
        }

        public static void InvokeAction<T>(this Control ctl, Action<T> action, T obj)   
        {   
            if (ctl.IsHandleCreated == true)   
            {   
                ControlFuncContext fc = new ControlFuncContext(ctl, action);   
                ctl.Invoke(new Action<T>(fc.Invoke1<T>), obj);   
            }   
        }

        public static void InvokeAction<T0, T1>(this Control ctl, Action<T0, T1> action, T0 obj0, T1 obj1)   
        {   
            if (ctl.IsHandleCreated == true)   
            {   
                ControlFuncContext fc = new ControlFuncContext(ctl, action);   
                ctl.Invoke(new Action<T0, T1>(fc.Invoke2<T0, T1>), obj0, obj1);   
            }   
        }   
    }

使用起来很简单，直接调用扩展方法 InvokeAction 即可，不必去考虑跨线程还是不跨线程这些“环境因素”，跨线程调用，我们已经通过用户不必知晓的方式，把它封装起来了。

再举个例子，写程序经常需要这样一个功能：打开一个图像文件，然后进行处理。正常写法很麻烦，比如，那个filter格式就很容易忘记，那么，我们就把它闭包化，把不该让用户知道，不该让用户敲键盘的都给它封装起来：

public static void OpenFile(this Form element, Action<String> callbackOnFilePath, String filter = "所有文件|\*.\*")

        {

            String filePath;

            OpenFileDialog dlg = new OpenFileDialog();

            dlg.Filter = filter;

            dlg.FileOk += (object sender, CancelEventArgs e) =>

            {

                filePath = dlg.FileName;

if (callbackOnFilePath != null)

                    callbackOnFilePath(filePath);

            };

            dlg.ShowDialog();

        }

public static void OpenImageFile(this Form element, Action<String> callbackOnFilePath, String filter = "图像文件|\*.bmp;\*.jpg;\*.gif;\*.png")

        {

            OpenFile(element, callbackOnFilePath, filter);

        }

再举一个例子，这个例子是as3中的。在Flex中，控件有一个callLater 方法，在下一帧时进行调用。这个方法非常有用，很多时候，非Flex项目也需要这样的一个方法。下面，我们进行模拟：

package orc.utils   
{   
    import flash.display.Stage;   
    import flash.events.Event;

    public class CallLaterHelper   
    {   
        public function CallLaterHelper(stage:Stage, callback:Function)   
        {   
            this.callback = callback;   
            this.stage = stage;

            stage.addEventListener(Event.ENTER\_FRAME, onStageEnterFrame);   
        }   
        private var stage:Stage;   
        private var callback:Function;   
        private function onStageEnterFrame(event:Event):void   
        {   
            stage.removeEventListener(Event.ENTER\_FRAME, onStageEnterFrame);   
            if(callback != null)   
            {   
                callback();   
            }   
        }   
    }   
}

然后在基础控件中，提供callLater方法：

public function callLater(callback:Function):void   
{   
    new CallLaterHelper(this.stage,callback);   
}

总结：

（1）闭包是一种设计原则，它通过分析上下文，来简化用户的调用，让用户在不知晓的情况下，达到他的目的；

（2）网上主流的对闭包剖析的文章实际上是和闭包原则反向而驰的，如果需要知道闭包细节才能用好的话，这个闭包是设计失败的；

（3）尽量少学习。