<https://www.evget.com/article/2010/10/26/15416.html>

**概述：**在WinForm控件上我们可以看到很多关于键盘消息处理的方法，比如OnKeyDown， OnKeyPress， ProcessCmdKey， ProcessDialogKey，IsInputKey等等，那么这些方法是如何被组织的，每一个方法的具体含义又是什么哪？Win32的键盘消息又是如何到达控件上的这些方法的，本文将着重阐述这些问题，对.Net WinForm控件的键盘消息处理过程进行剖析。

　　在WinForm控件上我们可以看到很多关于键盘消息处理的方法，比如OnKeyDown， OnKeyPress， ProcessCmdKey， ProcessDialogKey，IsInputKey等等，那么这些方法是如何被组织的，每一个方法的具体含义又是什么哪？Win32的键盘消息又是如何到达控件上的这些方法的，本文将着重阐述这些问题，对.Net WinForm控件的键盘消息处理过程进行剖析。

**1.      WinForm消息循环**

　　大家都知道WinForm也是依赖于底层的消息机制的，通常我们的WinForm应用程序会以如下方式启动：

　　Application.Run(new Form());

　　上面的代码将会在当前线程启动一个消息循环，并且显示指定窗体。

　　反编译Application类的Run方法，我们可以看到这一点：

　　public static void Run(Form mainForm)

　　{

　　 ThreadContext.FromCurrent().RunMessageLoop(-1,new ApplicationContext(mainForm));

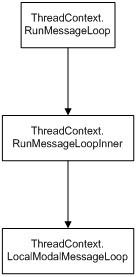
　　}

　　启动消息循环之后，操作系统就会将用户对于当前应用程序的UI输入转换为Windows消息发给当前线程进行处理。本文的重点不在于讲述Windows消息机制，而在于底层消息到达.Net这一层后，WinForm控件是如何处理的。

**2. 消息处理**

　　从上面可以看到通过ThreadContext类型的RunMessageLoop方法，构建了消息循环。那么对于一个特定的Windows消息，ThreadContext又是如何处理的哪？

　　分析ThreadContext的代码可以发现其调用关系如下：



　　在LocalModalMessageLoop方法中我们就可以看到对于Windows消息的处理了：

　　private bool LocalModalMessageLoop()

　　{

　　 // ...

　　 if(!PreTranslateMessage(ref msg))

　　 {

　　 // ...

　　 UnsafeNativeMethods.TranslateMessage(ref msg);

　　 UnsafeNativeMethods.DispatchMessage(ref msg);

　　 }

　　 // ...

　　}

可以发现对于一个特定的Windows消息，分为两个阶段进行处理：

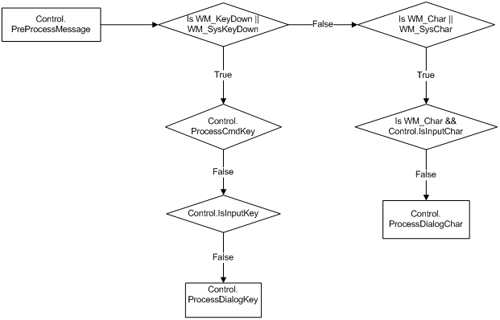
　　PreTranslateMessage   
DispatchMessage   
WinForm控件消息的处理将从这两个地方开始。

　　2.1 PreTranslateMessage

　　PreTranslateMessage提供了一个时机，来决定是否应该Dispatch这个消息，如果返回值为False，这个消息才会派发给WinForm控件。

　　PreTranslateMessage分为两个层次，第一优先调用当前应用程序的IMessageFilter来进行处理，用户可以在这一层进行消息预处理或者消息过滤。如果没有被过滤掉，调用当前控件的PreProcessMessage方法进行消息预处理。

　　Control类型的PreProcessMessage处理流程如下：



　　对于WM\_KeyDown消息，其预处理Control类上有三个时机：ProcessCmdKey，IsInputKey以及ProcessDialogKey；对于WM\_KeyChar消息，其预处理有两个时机：IsInputChar和ProcessDialogChar。

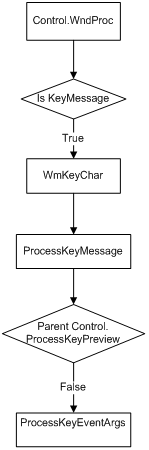
　　ProcessCmdKey默认用来处理快捷键以及菜单快捷键，此方法会递归调用父控件。如果返回值为False，继续调用IsInputKey，决定是否引发KeyDown事件。如果不是InputKey，调用ProcessDialogKey来检查该键是否为导航键，或者进行一些特别的处理，此方法会递归调用父控件的处理。

　　IsInputChar决定输入字符是否为普通字符，如果返回值为True会引发KeyPress事件。返回值为False会调用ProcessDialogChar，ProcessDialogChar默认用来处理Mnemonic键，例如控件的文本为"&OK"， 对于Char"O"的处理。

　　2.2 DispatchMessage

　　如果PreTranslateMessage没有过滤掉该Windows消息的话，该消息将会派发到控件，交由控件的WndProc函数进行处理。

　　下图是控件的处理流程：



　　消息到达WnProc之后，会交由ProcessKeyMessage，ProcessKeyPreview以及ProcessKeyEventArgs处理。每一个方法都会返回一个Boolean值，表明控件是否已经处理了该消息。

　　ProcessKeyMessage会处理所有由WndProc过来的所有键消息，首先会调用父控件的ProcessKeyPreview函数，如果返回True，表明父控件已经处理。否则调用ProcessKeyEventArgs来触发控件的KeyDown，KeyPress，KeyUp事件。

**3. 结语**

　　本文着重讲述了WinForm控件对于键盘消息的处理，分析了消息预处理以及处理两个阶段的各个函数。在进行三方控件的开发中可以根据需要重载这些函数，另外也可从其设计以及实现思路中获得更多启发。

<https://msdn.microsoft.com/zh-cn/library/ms171535(v=vs.110).aspx>

To ensure a control gets a key or combination of keys, you can handle the [PreviewKeyDown](https://msdn.microsoft.com/zh-cn/library/system.windows.forms.control.previewkeydown%28v=vs.110%29.aspx) event and set [IsInputKey](https://msdn.microsoft.com/zh-cn/library/system.windows.forms.previewkeydowneventargs.isinputkey%28v=vs.110%29.aspx) of the [PreviewKeyDownEventArgs](https://msdn.microsoft.com/zh-cn/library/system.windows.forms.previewkeydowneventargs%28v=vs.110%29.aspx) to **true** for the key or keys you want.

# Control.PreviewKeyDown 事件

Occurs before the [KeyDown](https://msdn.microsoft.com/zh-cn/library/system.windows.forms.control.keydown%28v=vs.110%29.aspx) event when a key is pressed while focus is on this control.

The arrow keys are considered navigation keys and pressing these keys typically do not raise the [KeyDown](https://msdn.microsoft.com/zh-cn/library/system.windows.forms.control.keydown%28v=vs.110%29.aspx) event for a [Button](https://msdn.microsoft.com/zh-cn/library/system.windows.controls.button%28v=vs.110%29.aspx). However, pressing the arrow keys for a [Button](https://msdn.microsoft.com/zh-cn/library/system.windows.controls.button%28v=vs.110%29.aspx) does raise the PreviewKeyDown event. By handling the PreviewKeyDown event for a [Button](https://msdn.microsoft.com/zh-cn/library/system.windows.controls.button%28v=vs.110%29.aspx) and setting the [IsInputKey](https://msdn.microsoft.com/zh-cn/library/system.windows.forms.previewkeydowneventargs.isinputkey%28v=vs.110%29.aspx) property to **true**, you can raise the [KeyDown](https://msdn.microsoft.com/zh-cn/library/system.windows.forms.control.keydown%28v=vs.110%29.aspx) event when the arrow keys are pressed.

# Control.ProcessKeyPreview 方法 (Message)

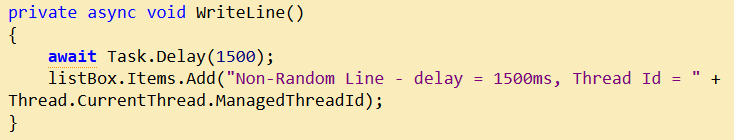
This method is called by a child control when the child control receives a keyboard message. The child control calls this method before generating any keyboard events for the message. If this method returns **true**, the child control considers the message processed and does not generate any keyboard events.

# async / await demo

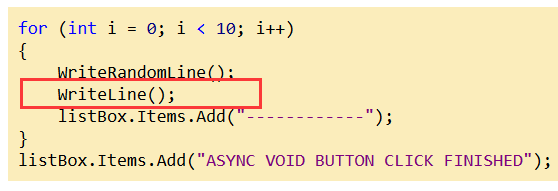
<https://www.codeproject.com/Tips/791286/async-await-demo>

The async and await keywords don't cause additional threads to be created. Async methods don't require multithreading because an async method doesn't run on its own thread. The method runs on the current synchronization context and uses time on the thread only when the method is active. You can use Task.Run to move CPU-bound work to a background thread, but a background thread doesn't help with a process that's just waiting for results to become available.

返回Task的方法被调用时都要加await，Task.Delay返回task



WriteLine前面不用加await。碰到await都要返回到调用者



# Application.Run Method (ApplicationContext)

Begins running a standard application message loop on the current thread, with an [ApplicationContext](https://msdn.microsoft.com/en-us/library/system.windows.forms.applicationcontext.aspx).

An [ApplicationContext](https://msdn.microsoft.com/en-us/library/system.windows.forms.applicationcontext%28v=vs.110%29.aspx) in which the application is run.

# Application.Run Method (Form)

Begins running a standard application message loop on the current thread, and makes the specified form visible.

Typically, the main function of an application calls this method and passes to it the main window of the application.

This method adds an event handler to the mainForm parameter for the Closed event. The event handler calls ExitThread to clean up the application.

# Application.Run Method ()

Begins running a standard application message loop on the current thread, without a form.

When the main message loop is closed, the application exits. In Windows Forms, this loop is closed when the [Exit](https://msdn.microsoft.com/en-us/library/ms157894%28v=vs.110%29.aspx) method is called, or when the [ExitThread](https://msdn.microsoft.com/en-us/library/system.windows.forms.application.exitthread%28v=vs.110%29.aspx) method is called on the thread that is running the main message loop.

Most Windows Forms developers will not need to use this version of the method. You should use the [Run(Form)](https://msdn.microsoft.com/en-us/library/ms157902%28v=vs.110%29.aspx) overload to start an application with a main form, so that the application terminates when the main form is closed. For all other situations, use the [Run(ApplicationContext)](https://msdn.microsoft.com/en-us/library/ms157901%28v=vs.110%29.aspx) overload, which supports supplying an [ApplicationContext](https://msdn.microsoft.com/en-us/library/system.windows.forms.applicationcontext%28v=vs.110%29.aspx) object for better control over the lifetime of the application.

# ApplicationContext Class

https://msdn.microsoft.com/en-us/library/system.windows.forms.applicationcontext(v=vs.110).aspx

You can use the ApplicationContext class to redefine the circumstances that cause a message loop to exit. By default, the ApplicationContext listens to the [Closed](https://msdn.microsoft.com/en-us/library/system.windows.forms.form.closed%28v=vs.110%29.aspx) event on the application's main [Form](https://msdn.microsoft.com/en-us/library/system.windows.forms.form%28v=vs.110%29.aspx), then exits the thread's message loop.