第 3 章

CHAPTER 3

Unity3D 插件高级功能实现

3.1 插件的通信与协作

当 Unity 插件的需求实现起来比较复杂，或者模块之间存在依赖，抑或插件和插件之间存在引用等现象时，这时为了降低代码耦合度，开发出高质量、易扩展和易维护的插件，就可以采用通信与协作技术来解决上述问题。

3.1.1 共享数据

共享数据的本质是想办法让不同模块或者插件能从同一个地方读写数据，比较常见的实现方式有以下几种。

采用静态和全局变量的方式，这种方式是内存实现极其简单，只需提供一个静态类或者在全局声明一个数据读写对象，但是如果过度地使用这种方式，则可能会导致代码难以维护和理解。

采用本机缓存的方式主要是解决本机插件不同功能模块之间共享数据的问题。可以参考表 2-1 进行选择，即以文件形式存入持久化目录还是采用 PlayerPrefs 形式进行缓存。本节内容在共享数据时如果以文件的方式，则需要考虑文件的大小，太大的文件必然会影响共享的速度。另外，在采用 PlayerPrefs 形式时，在不同平台存储位置也是不一样的，如表 3-1 所示。

表 3-1 PlayerPrefs 不同平台存储位置

平　台	存 储 位 置
Windows	存储在注册表中，可以通过注册表编辑器访问，路径是：HKCU/Software/[公司名称]/[产品名称]/键名
macOS	存储在/Users/用户名/Library/Preferences/文件夹中，其中包含一个名为“unity.[公司名称].[产品名称].plist”的文件
Linux	存储在用户的主目录中的“~/.config/unity3d/[公司名称]/[产品名称]/prefs”文件中

续表

平　台	存储位置
Android	存储在应用的数据目录中，通常是“/data/包名/shared_prefs/”文件夹中的一个 XML 文件
iOS	存储在应用的沙盒中的“Library/Preferences”目录中，文件名是“unity.[公司名称].[产品名称].plist”
WebGL	存储在浏览器的本地存储中，可以通过浏览器开发工具进行访问

采用数据库的方式，这种方式主要是解决存储数据比较复杂的情况，可以在插件的不同模块之间或者插件与插件之间共享数据时使用，有离线数据库和在线数据库的区别，可以参考表 3-1 的数据库适用场景选择不同的数据库。

采用服务器桥接方式，这种方式需要后端开发者的参与，不直接访问数据库，而是通过网络协议向后端发送读写请求，通过后端程序来完成数据的读写操作。后端程序通常会通过搭建服务器，在服务器搭建数据库进行存储。此方式可以让客户端读写简单，通常一个 API 就可以完成，但是需要后端程序的支持。

3.1.2　事件系统

事件系统是一种非常实用的通信机制，它允许不同的组件或模块之间不必直接引用对方，仅通过事件系统就可以进行松耦合通信。也可以通过在不影响其他部分代码的情况下，只通过添加新的事件处理逻辑就可以完成扩展操作。由于事件系统的核心思想是发布-订阅(Publish-Subscribe)模式，即当某个事件发生时，所有订阅了该事件的对象都会被通知，这种模式让通信关系变得比较清晰，也有助于代码的维护和调试。以上几点让插件自身的模块之间或者插件与插件之间的通信都变得可靠。

Unity 自身提供了一个事件系统，可以通过 C＃的事件(Events)和委托(Delegate)实现，但是，对于更复杂的需求开发，可能需要一个更加灵活和强大的事件管理器。这时，开发者可以创建自己的事件系统，或者使用现成的事件管理库，如 UniRx、EventSystem 等。

由于本书针对 Unity 插件开发，因此一般不会引用第三方的事件管理系统，本节将提供一个简易的事件管理器案例，并以此进行讲解。

【案例 3-1】 简易事件管理器。

首先，创建一个基础的事件类，用来作为所有特定事件的基类，这个类可以包含用来通信的数据字段定义，这里简单定义为用事件名称来描述事件，代码如下：

```
//第 3 章 //MsgData.cs

//< summary >
//事件消息数据基类
//</summary >
public class MsgData : EventArgs
{
}
```

然后创建一个事件管理类，用来存储订阅的事件和提供订阅事件、取消订阅和发布事件3种方法，这里使用委托Action实现通信功能（读者可修改代码基于事件实现一个版本），代码如下：

```
//第3章 //EventManager.cs

//<summary>
//事件管理器
//</summary>
public class EventManager : MonoSingleton<EventManager>
{
    //存储订阅的事件
    private Dictionary<string, List<Action<MsgData>>> eventDict = new Dictionary
<string, List<Action<MsgData>>>();

    //<summary>
    //订阅事件
    //</summary>
    //<param name="eventName"></param>
    //<param name="listener"></param>
    public void Subscribe(string eventName, Action<MsgData> listener)
    {
        if (!eventDict.ContainsKey(eventName))
        {
            eventDict[eventName] = new List<Action<MsgData>>();
        }
        eventDict[eventName].Add(listener);
    }

    //<summary>
    //取消订阅
    //</summary>
    //<param name="eventName"></param>
    //<param name="listener"></param>
    public void Unsubscribe(string eventName, Action<MsgData> listener)
    {
        if (eventDict.ContainsKey(eventName))
        {
            eventDict[eventName].Remove(listener);
        }
    }

    //<summary>
    //发布事件
```

```
        //</summary>
        //< param name = "eventName"></param >
        //< param name = "gameEvent"></param >
        public void FireEvent(string eventName, MsgData gameEvent)
        {
            if (eventDict.ContainsKey(eventName))
            {
                foreach (var listener in eventDict[eventName])
                {
                    listener?.Invoke(gameEvent);
                }
            }
        }
    }
```

以上完成了一个简易的事件系统，使用方式也很简单。首先需要继承 MsgData 类定义一个具体的事件，然后事件接收方订阅此事件，事件发起方发布此事件即可完成，代码如下：

```
    //第 3 章 //Script_3_1_2.cs

    //定义事件
    public class StartEventData : MsgData
    {
        //要传输的内容
        public string StartupParams { get; private set; }

        public StartEventData(string startupParams)
        {
            StartupParams = startupParams;
        }
    }

    void Start()
    {
        //订阅事件
        EventManager.Instance.Subscribe("StartEvent", OnStartEvent);
    }

    //< summary >
    //订阅事件的处理逻辑
    //</summary >
    //< param name = "msg"></param >
    private void OnStartEvent(MsgData msg)
    {
```

```
        StartEventData data = (StartEventData)msg;
        if (data != null)
        {
            Debug.Log( $ "已收到事件,内容是: {data.StartupParams}");
        }
    }

    private void OnGUI()
    {
        //事件发起方
        if (GUILayout.Button("模拟事件发起方"))
        {
            EventManager.Instance.FireEvent("StartEvent", new StartEventData("我是启动参数数
据流"));
        }
    }
```

当单击按钮时,StartEvent 事件便会被触发,然后 OnStartEvent 函数便会被执行。

3.1.3 消息队列

消息队列也是一种使用频率较高的通信机制,有同步消息和异步消息之分,同步消息因为消息的处理逻辑在主线程上执行,当消息处理逻辑较为复杂或者耗时较长时就会阻塞主线程,这会导致程序的更新和渲染等关键操作无法及时执行,进而引起画面卡顿或者帧率下降,而异步消息将消息处理逻辑放在多线程或者协程中实现,可以有效地避免这种阻塞现象。本节将通过协程实现异步消息队列。这里需要阐述协程的工作方式,协程是一种用户级别的轻量级线程,但是它不是真正操作系统的线程,而是在 Unity 的执行循环中被调度和管理的一种机制。该机制是通过 yield 关键字来挂起和恢复执行的,当协程执行到 yield 语句时,它会暂停当前的执行,然后将控制权交还给 Unity 的事件循环,再在每帧结束后去检查 yield 条件是否满足,如果满足就会继续执行协程; 如果不满足则在下一帧后再检测 yield 条件。它的这个过程是协作式的挂起,不是抢占式的中断,因此协程可以在不阻塞主线程的情况下执行长时间的操作。

【案例 3-2】 基于同步,异步线程和异步协程分别实现消息队列通信。

首先,定义消息对象,用来描述消息类型和消息内容,也可以自定义添加其他的属性字段,代码如下:

```
//第 3 章 //Message.cs

//< summary >
//消息
//</summary >
public class Message
```

```
{
    //消息类型
    public string Type { get; private set; }
    //消息内容
    public string Content { get; private set; }

    public Message(string type, string content)
    {
        Type = type;
        Content = content;
    }
}
```

然后以同步机制为例实现消息管理器。需要定义一个消息队列，用来存储消息，再定义一个字典，用来对存储的消息进行事件绑定，然后对外提供发送消息、订阅消息和取消订阅3种方法。最后，实现如何分发消息，这也是同步和异步的区别所在，基于同步机制实现的消息管理，代码如下：

```
//第 3 章 //MessageManager_Normal.cs

//同步机制消息管理器
public class MessageManager_Normal : MonoSingleton<MessageManager_Normal>
{
    //消息队列
    private Queue<Message> messageQueue = new Queue<Message>();
    //为消息绑定事件
    private Dictionary<string, Action<Message>> listeners = new Dictionary<string,
Action<Message>>();

    //<summary>
    //发送消息
    //</summary>
    //<param name="message"></param>
    public void Send(Message message)
    {
        messageQueue.Enqueue(message);
    }

    //<summary>
    //订阅消息
    //</summary>
    //<param name="messageType"></param>
    //<param name="listener"></param>
    public void Subscribe(string messageType, Action<Message> listener)
```

```
        {
            if (!listeners.ContainsKey(messageType))
            {
                listeners[messageType] = listener;
            }
            else
            {
                listeners[messageType] += listener;
            }
        }

        //< summary >
        //取消订阅
        //</summary >
        //< param name = "messageType"></param >
        //< param name = "listener"></param >
        public void Unsubscribe(string messageType, Action < Message > listener)
        {
            if (listeners.ContainsKey(messageType))
            {
                listeners[messageType] -= listener;
            }
        }

        void Update()
        {
            //消息出队,并执行消息绑定的事件
            if(messageQueue.Count > 0)
            {
                var message = messageQueue.Dequeue();
                if (message != null && listeners.ContainsKey(message.Type))
                {
                    listeners[message.Type]?.Invoke(message);
                }
            }

        }
    }
```

然后实现基于多线程的消息管理类,它也是一个单例类,但不用继承自 MonoBehaviour,只需将以上代码修改两处。第 1 处是将 Update 的方法换成异步线程执行方法。第 2 处是为发送消息函数体增加一个调用异步线程方法,核心代码如下:

```
//第 3 章 //MessageManager_Task.cs

//< summary >
//发送消息
//</summary >
//< param name = "message"></param >
public void Send(Message message)
{
    messageQueue.Enqueue(message);
    ProcessMessagesAsync();
}

//< summary >
//消息出队,并执行消息绑定的事件
//</summary >
private async void ProcessMessagesAsync()
{
    await Task.Run(() =>
    {
        var message = messageQueue.Dequeue();
        if (message != null && listeners.ContainsKey(message.Type))
        {
            Task.Run(() => listeners[message.Type]?.Invoke(message));
        }
    });
}
```

最后基于协程的消息管理类大同小异,因为用协程,因此也是基于 MonoBehaviour 的单例类,并且也只需修改以上两部分。第 1 处将 Update 换成协程方法。第 2 处是为发送消息函数体增加一个调用协程的方法,核心代码如下:

```
//第 3 章 //MessageManager_Coroutine.cs

//< summary >
//发送消息
//</summary >
//< param name = "message"></param >
public void Send(Message message)
{
    messageQueue.Enqueue(message);
    StartCoroutine(ProcessMessagesCoroutine());
}

//< summary >
```

```
    //消息出队,并执行消息绑定的事件
    //</summary>
    //< returns ></returns >
    private IEnumerator ProcessMessagesCoroutine()
    {
        while (messageQueue.Count > 0)
        {
            var message = messageQueue.Dequeue();
            if (message != null && listeners.ContainsKey(message.Type))
            {
                listeners[message.Type]?.Invoke(message);
            }
            yield return null;
        }
    }
```

如此,便完成了3种方式实现的消息队列通信机制。使用方法也非常简单,这里使用OnGUI的按钮来模拟3种方式的消息发送,然后分别监听这3种方式的消息即可,代码如下:

```
    //第3章 //Script_3_1_3.cs

    void Start()
    {
        //订阅3种机制实现的消息
        MessageManager_Normal.Instance.Subscribe("Greeting", OnHandleGreetingMessage_Normal);
        MessageManager_Task.Instance.Subscribe("Greeting", OnHandleGreetingMessage_Task);
        MessageManager_Coroutine.Instance.Subscribe("Greeting", OnHandleGreetingMessage_Coroutine);
    }
    private void OnDestroy()
    {
        //摧毁时取消订阅3种机制实现的消息
        MessageManager_Normal.Instance.Unsubscribe("Greeting", OnHandleGreetingMessage_Normal);
        MessageManager_Task.Instance.Unsubscribe("Greeting", OnHandleGreetingMessage_Task);
        MessageManager_Coroutine.Instance.Unsubscribe("Greeting", OnHandleGreetingMessage_Coroutine);
    }

    //基于同步消息的接收方
    private void OnHandleGreetingMessage_Normal(Message message)
    {
        Debug.Log( $ "收到打招呼消息[同步]: {message.Content}");
```

```
}

//基于异步多线程的接收方
private void OnHandleGreetingMessage_Task(Message message)
{
    Debug.Log( $ "收到打招呼消息[异步 - 多线程]: {message.Content}");
}

//基于异步协程的接收方
private void OnHandleGreetingMessage_Coroutine(Message message)
{
    Debug.Log( $ "收到打招呼消息[异步 - 协程]: {message.Content}");
}

private void OnGUI()
{
    //模拟同步消息的发起方
    if (GUILayout.Button("模拟发送消息[同步]: 打招呼"))
    {
        Message message = new Message("Greeting", "【同步】Hello World!");
        MessageManager_Normal.Instance.Send(message);
    }
    //模拟异步多线程的发起方
    if (GUILayout.Button("模拟发送消息[异步 - 多线程]: 打招呼"))
    {
        Message message = new Message("Greeting", "【异步 - 多线程】Hello World!");
        MessageManager_Task.Instance.Send(message);
    }
    //模拟异步协程的发起方
    if (GUILayout.Button("模拟发送消息[异步 - 协程]: 打招呼"))
    {
        Message message = new Message("Greeting", "【异步 - 协程】Hello World!");
        MessageManager_Coroutine.Instance.Send(message);
    }
}
```

3.1.4　接口和抽象类

接口(Interface)和抽象类(Abstract Class)是两种在面向对象编程中用于实现抽象和多态性的主要方法。它们都提供了一种可以在不同对象之间共享的约定方法,但是不具体实现这种约定,具体的实现是在子类中实现的。这种方法有助于提高代码的可维护性、可扩展性和模块化设计。

接口作为约定方法,只能定义一组没有实现细节的方法或属性,不能被直接实例化,但

它允许被多重继承，也就是说一个类可以实现多个接口。继承接口的类需要分别实现每个接口的定义细节，但调用方却不用关心实现细节，直接调用接口方法即可完成通信和协作。

抽象类与接口类似，也不能被直接实例化。不同的是，抽象类可以包含实现代码、抽象方法和属性，但需要注意的是，在C#中不支持多重继承。

可以发现，接口和抽象类与前面提到的消息和事件等通信协作方法不一样，它更适用于在不同模块和插件之间定义一种通信协作协议，然后由各自的子类去实现具体的内容。在Unity插件开发中，这种方式除了用在插件模块高度抽象实现上，也用在插件提供一些可扩展的预留功能上，在插件定义一个接口或者抽象类，然后支持用户对此插件额外地进行扩展。

本节通过一个简单案例讲解接口和抽象类的简单使用。

【案例 3-3】 使用接口和抽象类实现不同模块之间打招呼。

分析这个简单案例的需求可以知道，不同的模块之间都有类似的打招呼行为，只是打招呼的内容可能是不一样的，因此可以通过接口和抽象类分别约束一个打招呼的行为。接口代码如下：

```
//第 3 章 //IGreeting.cs

//打招呼接口
public interface IGreeting
{
    //打招呼次数
    int GreetingCount {get; set;}
    //定义打招呼的行为
    void Greeting();
}
```

抽象类代码如下：

```
//第 3 章 //AbstractGreeting.cs

//打招呼抽象类
public abstract class AbstractGreeting
{
    //打招呼次数
    public abstract int GreetingCount {get;}

    //定义打招呼行为
    public abstract void Greeting();

    //带有具体实现的方法
    public string GetTime()
    {
```

```
            return DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss");
        }

    }
```

然后实现具体的实现类。继承自接口类 IGreeting 的第 1 个派生类 ChineseGreeting，代码如下：

```
    //第 3 章 //ChineseGreeting.cs

    //打招呼对象
    public class ChineseGreeting : IGreeting
    {
        //打招呼次数
        public int GreetingCount {get; set;} = 1;

        //< summary >
        //打招呼的具体实现
        //</summary >
        public void Greeting()
        {
            for(int i = 0; i < GreetingCount; i++)
            {
                Debug.Log("您好,欢迎来到中国!");
            }
        }
    }
```

继承自接口类 IGreeting 的第 2 个派生类 EnglishGreeting，代码如下：

```
    //第 3 章 //EnglishGreeting.cs

    //打招呼类对象
    public class EnglishGreeting : IGreeting
    {
        //设定打招呼次数
        public int GreetingCount {get; set;} = 2;
        //< summary >
        //打招呼的具体实现
        //</summary >
        public void Greeting()
        {
            for (int i = 0; i < GreetingCount; i++)
            {
                Debug.Log("Hello,China!");
            }
        }
    }
```

继承自抽象类 AbstractGreeting 的第 1 个派生类 BoyGreeting，代码如下：

```
//第 3 章 //BoyGreeting.cs

//打招呼对象
public class BoyGreeting : AbstractGreeting
{
    //打招呼次数
    public override int GreetingCount => 2;

    //< summary >
    //打招呼具体实现
    //</summary >
    public override void Greeting()
    {
        for (int i = 0; i < GreetingCount; i++)
        {
            Debug.Log( $ "时间:{GetTime()}] 你好,大美女～～");
        }
    }
}
```

继承自抽象类 AbstractGreeting 的第 2 个派生类 GirlGreeting,代码如下:

```
//第 3 章 //GirlGreeting.cs

//打招呼对象
public class GirlGreeting : AbstractGreeting
{
    //打招呼次数
    public override int GreetingCount => 1;

    //< summary >
    //具体打招呼细节
    //</summary >
    public override void Greeting()
    {
        for (int i = 0; i < GreetingCount; i++)
        {
            Debug.Log( $ "时间:{GetTime()}] 你好,大帅哥～～");
        }
    }
}
```

从以上代码可以看出,两者的使用非常类似,主要通过约定基类方法进行模块间的通信和协作。对它们的使用只需在适当的模块实例化出派生类对象,然后统一调用 Greeting 方法,具体的实现都在派生类中实现了,代码如下:

```
//第 3 章 //Script_3_1_4.cs

private void OnGUI()
{
    if(GUILayout.Button("功能模块 1 - 中国人打招呼【接口】"))
    {
        IGreeting chinese = new ChineseGreeting();
        chinese.Greeting();
    }

    if (GUILayout.Button("功能模块 2 - 英国人打招呼【接口】"))
    {
        IGreeting english = new EnglishGreeting();
        english.Greeting();
    }

    if (GUILayout.Button("功能模块 3 - 男孩打招呼【抽象类】"))
    {
        AbstractGreeting boys = new BoysGreeting();
        boys.Greeting();
    }

    if (GUILayout.Button("功能模块 4 - 女孩打招呼【抽象类】"))
    {
        AbstractGreeting girls = new GirlGreeting();
        girls.Greeting();
    }
}
```

3.2　插件与 Unity3D 编辑器的集成

当插件本身功能或者局部功能已经完善时，可以再结合 Unity3D 编辑器可扩展的特性，用来快速配置插件模块参数或者测试插件功能等，这样便可以进一步提升插件的使用效率。鉴于本书 1.2 节已经讲解了如何对编辑器进行扩展，本节仅进行补充和完善。

3.2.1　自定义编辑器窗口

编辑器窗口可以给使用者提供一个更聚焦的使用体验，例如 Unity 的 Package Manager 窗口和 Addressables 插件的编辑窗口，但如果一个插件仅通过简单配置和调度 API 就可以提供完整服务，那就没必要再开发一个编辑器窗口了，不提倡过度设计和开发。

但如果要自定义编辑器窗口，这里对主要步骤进行总结，首先新建的类要继承自

EditorWindow 类，其次需要通过 MenuItem 属性添加菜单项。最后，当选中这个菜单项时调用 EditorWindow.GetWindow 方法打开这个窗口。至于窗口上的各种控件布局和行为则通过重写 OnGUI 方法实现。自定义编辑器窗口可以参考本书案例 1-6 中的编辑器登录窗口实现部分，但本节也提供一个示例代码，用于展示一些与之不同的使用方法。

【案例 3-4】 更多的自定义编辑器窗口示例。

本案例将展示搜索组件、对象选择组件、滚动条组件、按钮组件和选择组件按钮的使用，以及如何使用 GUI.skin.FindStyle 获取内置样式的方法（内置样式工具在 1.2.6 节已实现，路径为 PluginDev→获取样式），代码如下：

```
//第 3 章 //PluginDevWindow.cs

public class PluginDevWindow : EditorWindow
{
    //搜索文本
    private string searchString = "";
    //滚动坐标
    private Vector2 scrollPosition = Vector2.zero;
    //选择的对象
    private Object selectedObject;

    private bool[] boolArray = new bool[10];

    [MenuItem("PluginDev/扩展编辑器窗口示例")]
    public static void ShowWindow()
    {
        var window = GetWindow<PluginDevWindow>("编辑器窗口示例");
        window.maxSize = new Vector2(400, 300);
        window.Show();
    }

    void OnGUI()
    {
        //水平布局
        GUILayout.BeginHorizontal("Box");
        GUILayout.Label("这是文本标签", EditorStyles.boldLabel);
        if (GUILayout.Button("这是按钮", GUILayout.Width(100)))
        {
            Debug.Log("单击按钮...");
        }
        GUILayout.EndHorizontal();

        GUILayout.Space(10);                    //设置空行
        GUILayout.BeginHorizontal(GUI.skin.FindStyle("Toolbar"));
```

```
        searchString = GUILayout.TextField(searchString, GUI.skin.FindStyle
("ToolbarSearchTextField"));
        //搜索输入框
        if (GUILayout.Button("这是搜索", GUI.skin.FindStyle("ToolbarSearchCancelButton")))
        {
            searchString = "";
            GUI.FocusControl(null);
        }
        GUILayout.EndHorizontal();

        GUILayout.Space(5);
        GUILayout.BeginHorizontal();
        GUILayout.Label("这是选择对象", GUILayout.Width(50));
        //对象选择框
        selectedObject = EditorGUILayout.ObjectField(selectedObject, typeof(Object), false);
        GUILayout.EndHorizontal();

        GUILayout.Space(5);
        //滚动条视图
        scrollPosition = GUILayout.BeginScrollView(scrollPosition);
        //布局 Toggle
        for (int i = 0; i < 10; i++)
        {
            GUILayout.BeginHorizontal();
            boolArray[i] = GUILayout.Toggle(boolArray[i], $"这是一个单选按钮{i + 1}",
"Radio");
            GUILayout.EndHorizontal();
        }
        GUILayout.EndScrollView();

        GUILayout.Space(10);
        GUILayout.BeginHorizontal();
        if (GUILayout.Button("这是另一个按钮", GUILayout.Height(40)))
        {
            Debug.Log("又单击一个按钮...");
        }
        GUILayout.EndHorizontal();
    }
}
```

3.2.2 自定义快捷键

在Unity编辑器中定义快捷键需要通过一种方法来执行逻辑，并且使用属性MenuItem来为这种方法定义一个菜单命令及对应的快捷键，但没这么简单，快捷键的设置需要遵循特定的格式才会生效，其中%代表Ctrl(在macOS系统中为Cmd)；#代表Shift；&代表Alt，而“_”后面直接跟着的字符则表示一个按键。例如，_F1表示F1。本节通过一个案例进行讲解。

【案例3-5】 自定义快捷键操作。

本案例主要说明4种快捷键的定义方法，代码如下：

```
//第3章 //Script_3_2_2.cs

[MenuItem("PluginDev/快捷键测试[Ctrl + E] % e")]
public static void CtrlAndE()
{
    Debug.Log("Ctrl + E");
}

[MenuItem("PluginDev/快捷键测试[Shift + E] # e")]
public static void ShiftAndE()
{
    Debug.Log("Shift + E");
}

[MenuItem("PluginDev/快捷键测试[Alt + E] &e")]
public static void AltAndE()
{
    Debug.Log("Alt + E");
}

[MenuItem("PluginDev/快捷键测试[E] _e")]
public static void e()
{
    Debug.Log("E");
}
```

3.2.3 自定义回调事件

在1.2.7节已经讲解过编辑器回调的一些函数，本节将补充一个可以在代码编译完成后自定义的回调事件。它就是UnityEditor.Callbacks.DidReloadScripts属性，此属性允许在Unity脚本编译完成后自动执行特定的函数。这个功能特别适合在需要自动刷新资源、

更新脚本变量或者初始化脚本时使用。本节通过一个案例讲解如何使用此属性。

【案例 3-6】 在 Unity 编辑器中实现等代码编译完成后查询场景里节点名为 Target GameObject 的对象，并为此对象查找或添加一个脚本，然后初始化此脚本的变量值。

本案例需要在类里面定义一个静态方法，用来表示待执行的逻辑，并且用属性 UnityEditor. Callbacks. DidReloadScripts 进行约束，代码如下：

```
//第 3 章 //Script_3_2_3.cs

[DidReloadScripts]
private static void OnScriptsReloaded()
{
    GameObject[] allObjects = Object.FindObjectsOfType<GameObject>();
    foreach (var obj in allObjects)
    {
        //查找目标节点
        if (obj.name == "TargetGameObject")
        {
            Script_3_2_3_Instance instance = obj.GetComponent<Script_3_2_3_Instance>();
            if (instance == null)
            {
                //自动添加组件
                instance = obj.AddComponent<Script_3_2_3_Instance>();
                Debug.Log( $"已为{obj.name}自动添加了组件 Script_3_2_3_Instance");
            }
            //设置组件的变量的初始值
            instance.Id = "设置为初始化值";
        }
    }
}
```