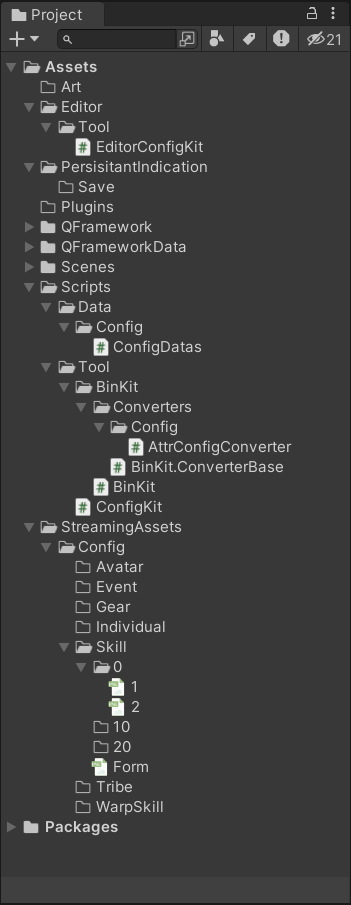
1. 文件夹

文件夹结构如图：  
1.Scripts/Data:  
 其中定义数据结构，当前只有配置数据(Config)，之后会有更多。

2.Scripts/Tool:  
 其中定义的是静态工具类，并非架构中的Utility，而是与QFramework中的各种Kit类似。

ConfigKit是配置文件一键(准确来说是一个方法)读取的工具，是我们数据存储开发要服务的对象之一。  
BinKit和BinKit.ConverterBase和Converters是数据读写开发的重点，后面详细说明。

3.Editor/Tool：

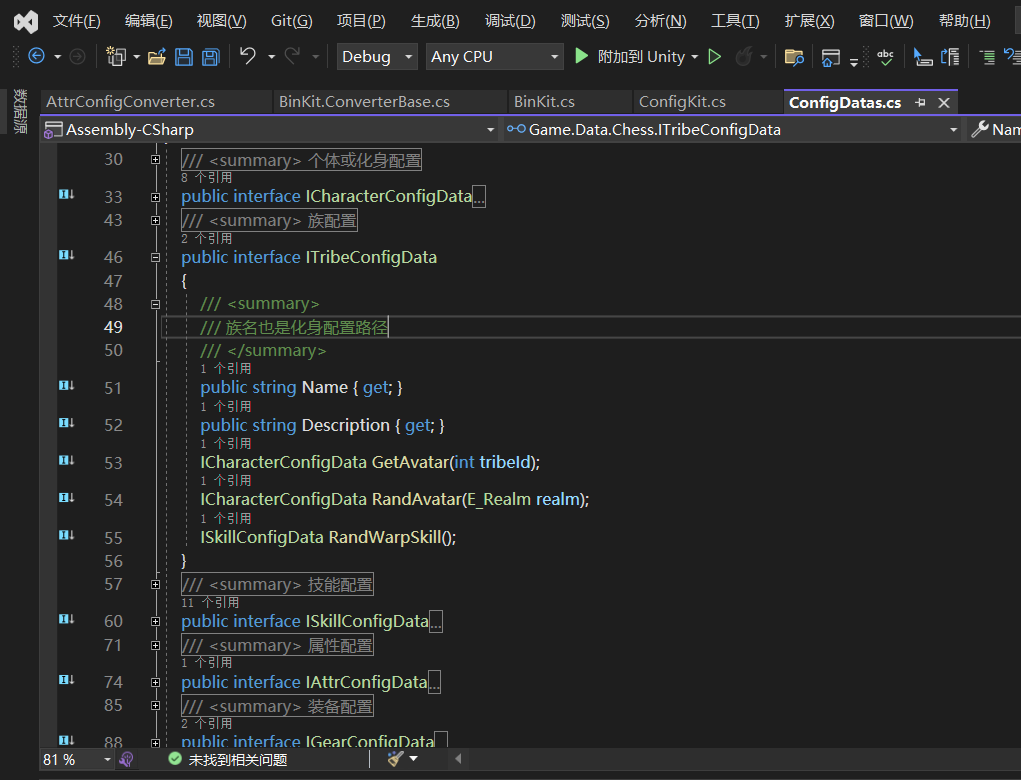
Editor文件夹是游戏编辑器使用的文件夹，其中信息在游戏打包时会被忽略。  
 Editor下的Tool也是静态工具类，只不过是编辑器专用。EditorConfigKit自然是编辑器使用的ConfigKit。他与Scripts中的ConfigKit的区别后面说明，只需要知道他也是数据读写开发要服务的对象之一就行。

4.StreamingAssets/Config：  
 StreamingAssets是游戏的只读路径，其中的Config中存放的是游玩过程中使用的配置文件的存放位置。

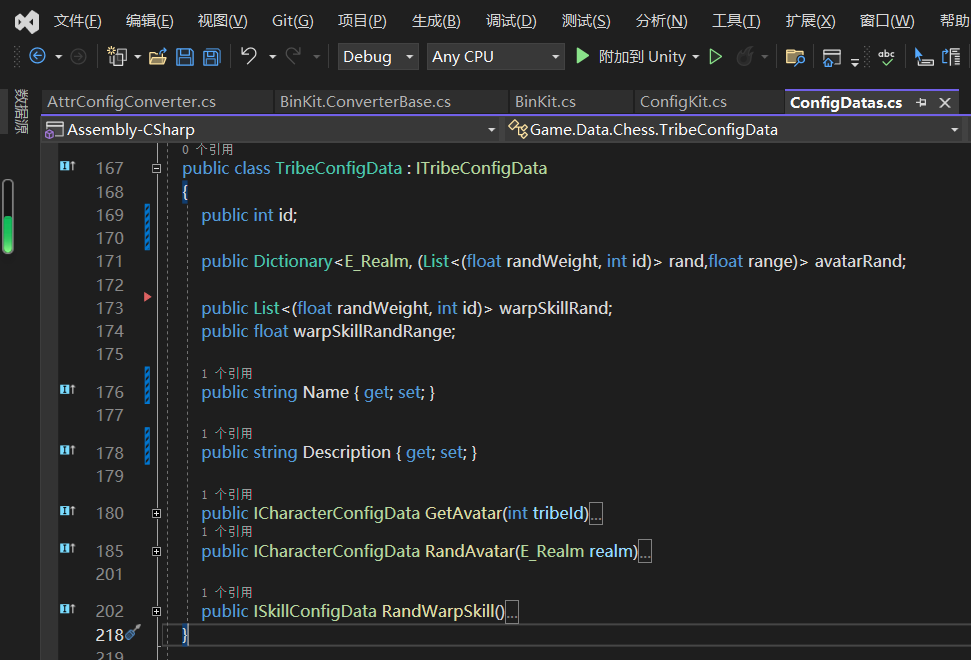
5.PersistantIndication：  
 这只是一个示意文件，游戏的PersistentDataPath是可读可写路径，但不在Assets文件夹中，这里示意我们会在该路径中存放存档信息。  
 存档信息的读写依赖的是设计图中的各种存储器Utility，这些也是数据读写开发要服务的对象之一。

1. 数据结构

我们来看一下Config数据的具体内容，其他数据也会是一样的模式。



首先会发现我们定义了数据的接口，这些接口是在数据已经加载完成后，提供给其他模块使用的。配置文件接口的特征是，所有数据都是只读属性，并且有一些取数据的方法（如随机取出）。



然后来看实现接口的数据结构类。这些是我们真正要读写的东西。其中除了对接口的支持以外，属性都有了写入权限，方便我们从文件中把数据读出来给他赋值。并且还增加了一些用于组织数据的字段，如id和randWeight（随机权重）。有些在接口中看来可以取出的数据其实只存了一个id(如化身信息，跃迁技能信息)

三 序列化和反序列化

在理解BinKit之前，我们需要先了解序列化和反序列化（已经了解的话就跳过吧）。

序列化就是把一个类对象或者结构体对象，总之是以C#中结构组织的数据，转化成一串字符/一堆字节。这串数据很容易就能存储在一个文件里或者进行网络传输（当然我们用不到）。

反序列化就是序列化的反过程，把一串数据转化为类对象或结构体对象。

这个“一串数据”长成什么样没有一定之规，只要反序列化得到的结果和序列化进去的数据一致，就是合格的算法。当然，在此前提下这串数据越短，计算越快越好。

四 认识BinKit

Bin是二进制的意思，所以可知我们序列化和反序列化用的是二进制字节流（可以理解成更高级字节列表）。

BinKit文件中的内容已经写好了，我们做一下介绍：

1.ByteArray：

就是给字节数组加上一个读取位置指针，但是这个指针不会自己动，需要手动设置。

2.IConverter：

有一个IConverter接口，他是IConverter泛型的父类，唯一的目的是应用里氏替换原则：一个IConverter类的列表/字典可以装所有IConverter泛型类。

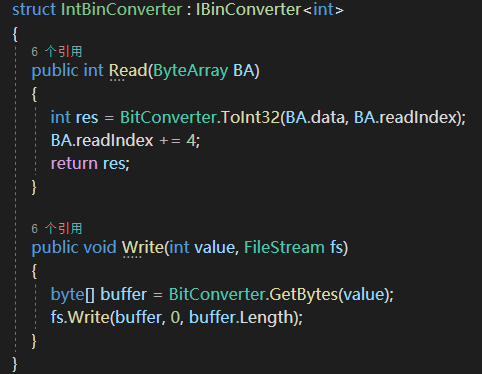
3.IConverter泛型接口：

有Write（序列化并写入流）和Read（从ByteArray中读取数据，进行反序列化后返回）

4.Converter实例：

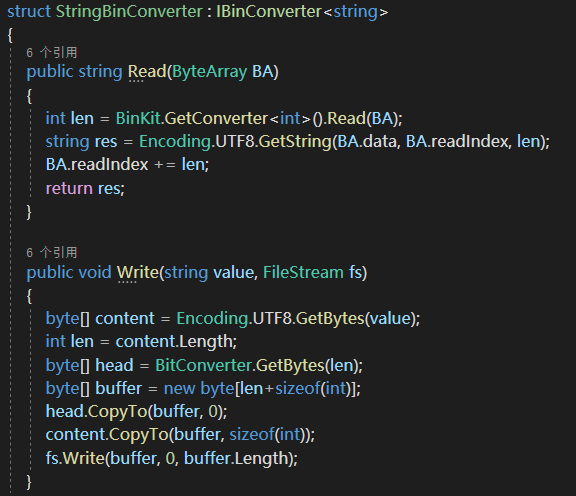
Converter基础实例都写在了BinKit.ConverterBase中

我们来看已经写好的几个Converter



int类型的Read，使用BitConverter从BA中读出一个int，占4字节，所以让读指针挪了sizeof（int），即4字节。

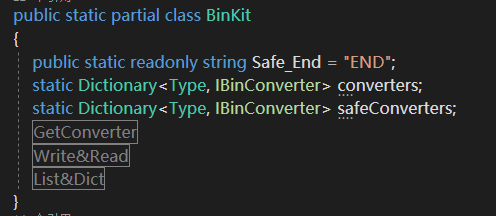
int类型的Write，使用BitConverter把int转成字节数组，写入流中。



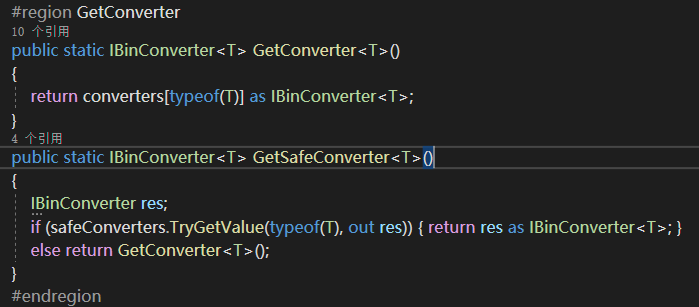
string类型的Write，因为他是不定长的，对这类数据的标准作法就是在数据前加一个int（4字节）来标识他的长度（注意是字节长度不是字符串长度）。

string类型的Read，从BinKit中获取了int类型的Converter，读取字节长度，然后用Encoding反序列化字符串，然后修改读指针。这里需要特别注意，所有Converter的Read都会保证正确更新读指针，因此我们调用别的Converter进行反序列化后，不能再多更新一次读指针，所以在读取int后我们没有让读指针+4。

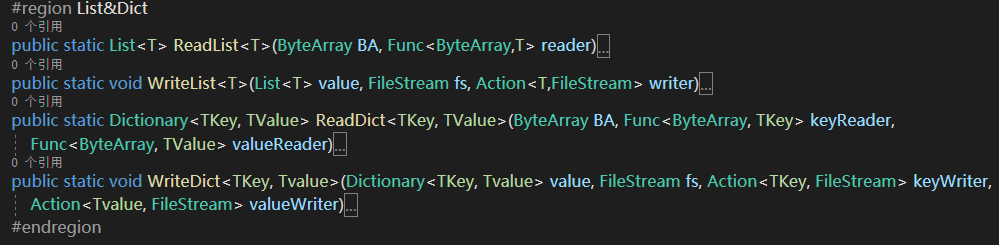
1. BinKit：



有一个converters存储各种Converter，可以通过类型获取，方法如下:

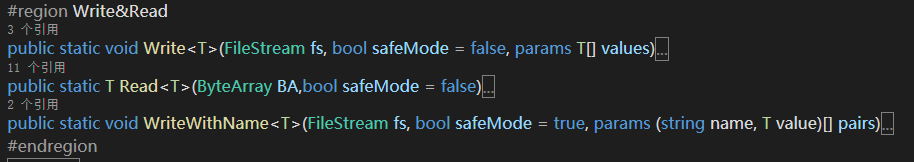


有读写列表和字典的方法，因为这些逻辑比较特殊，需要传入读取元素的方法。如下:



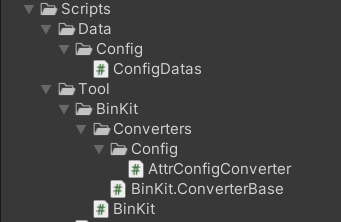
有一些快捷的读写方法。还有一个safeConverters。这里存的是SafeConverter，还有一个常量Safe\_End，都是给编辑器用的，这些内容在后面实例中都会提及。

快捷读写:

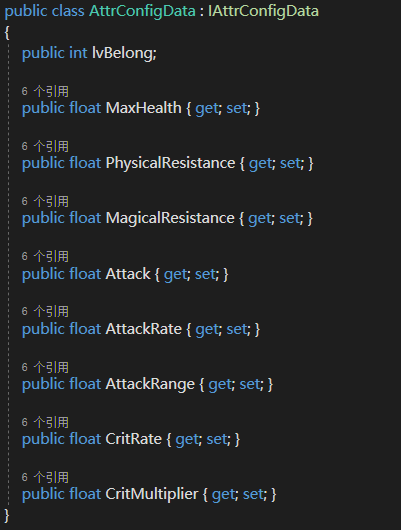


6.实例：

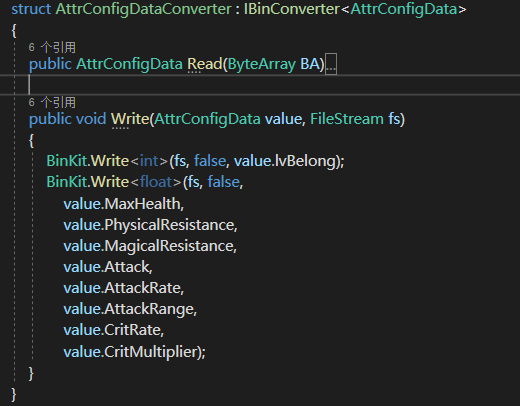
我们来开发一些Converter(下称转换器)看看，开发一个属性配置的转换器，放在Script/Tool/BinKit/Converters/Config下



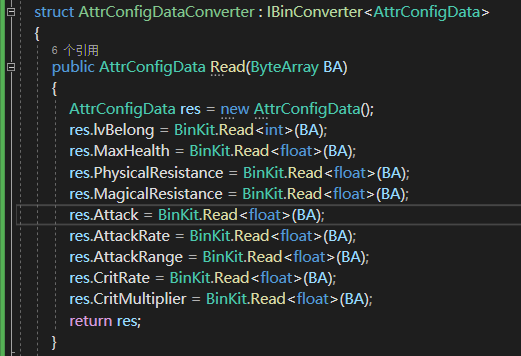
先来看看我们要读写的对象，注意是实现类不是接口(见下页)。

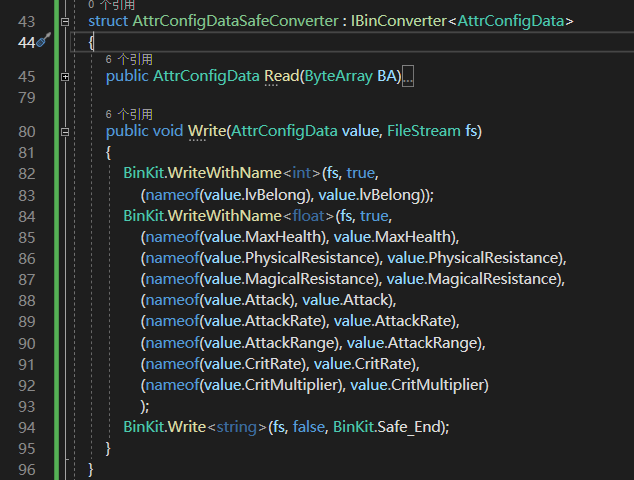


我建议你们写的时候也想这样截一张图对着看，ctrl+滚轮可以缩放编辑器视图。



先把序列化方法写出来，涉及到BinKit.Write()，第一个参数是文件流直接传fs，第二个是safeMode直接传方法提示里的默认值，之后就是存的值，只要类型相同就可以传任意多个，会按传入顺序依次存储。

然后可以写反序列化方法了，BinKit.Read()方法从BA中读取值，与序列化顺序一致即可。



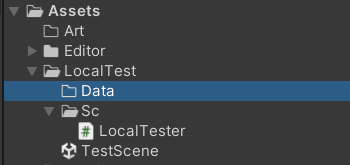
然后写Safe模式的，思路就是在存每个变量的值之前把名字存进去，因为开发阶段可能会给类增加新的变量，这种方式可以在增加新变量后仍然正常读取之前配置过的数据。

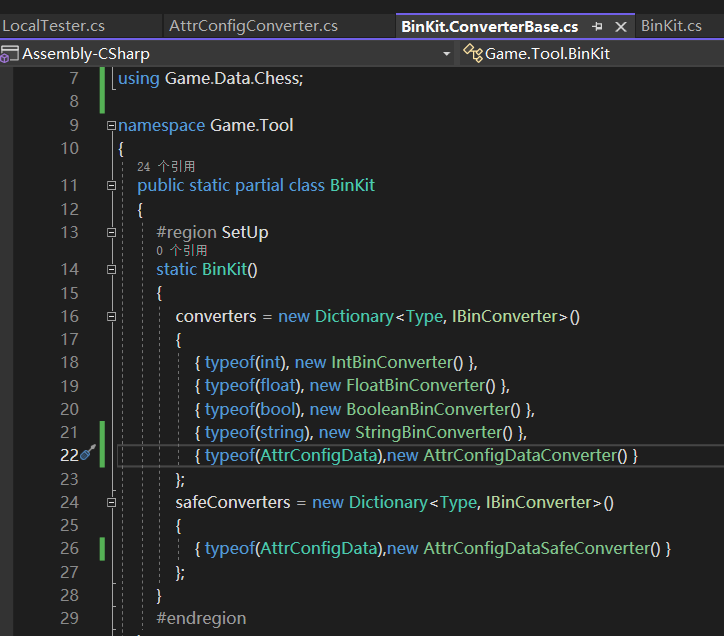
BinKit.WriteWithName()唯一的变化就是传入的不只是值，而是由名称和值组成的元组。

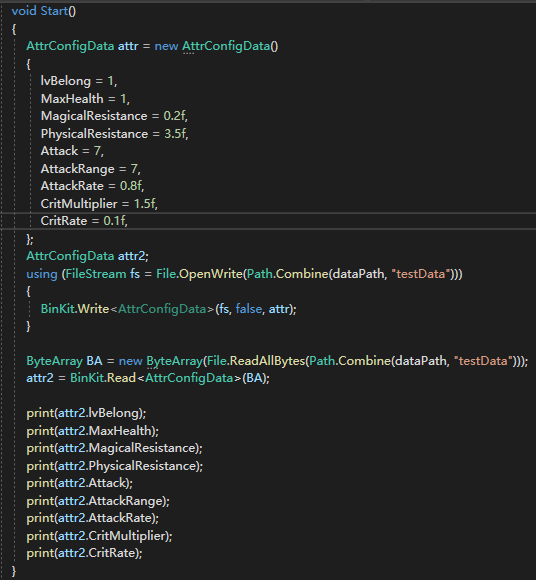


安全模式的反序列化需要用到C#的“反射”特性，在学会反射之后，这部分应该不难理解。

五 测试

在Assets下创建LocalTest文件夹，已经让GitHub忽略该目录下的所有内容。

记得登记你写的转换器。

编写测试脚本，新建一个对象，存进去再取出来。

运行结果如下：



六 任务：

目前任务就是给ConfigData都写转换器。