

# 数据存储与读取

游戏中通常要实现“保存进度”“读取进度”之类功能，那么在开发中，就需要对数据进行存储和读取等工作。本章讲述用PlayerPrefs实现对简单数据的处理以及用JSON实现对对象结构的数据的处理。

## 1 PlayerPrefs

PlayerPrefs是Unity自带的数据结构，位于UnityEngine命名空间下。它可以对整数，浮点数，字符串3种类型的数据进行存取。它是持久存储于设备上的。例如安卓，只要用户没有删除应用或者手动去清除应用数据，PlayerPrefs的数据就会一直保留。

### 1.1 整数的存取

一般脚本都会使用UnityEngine命名空间，所以可以在脚本里直接使用PlayerPrefs.

**PlayerPrefs使用“键/值”配对的规则**，如下：

```
int num = 10; // 定义一个整型变量num
```

```
PlayerPrefs.SetInt("Number", num); // 存储该变量，第一个参数是键，第二个参数是值
```

如何读取呢？

```
int num = PlayerPrefs.GetInt("Number");
```

### 1.2 浮点数，字符串的存取

```
float PI = 3.14f; // 定义一个浮点数

PlayerPrefs.SetFloat("PI", PI); // 存储

PI = PlayerPrefs.GetFloat("PI"); // 读取

String str = "neworigin"; // 定义一个字符串

PlayerPrefs.SetString("Str", str); // 存储

Str5 = PlayerPrefs.GetFloat("Str"); // 读取
```

练习：

```
void Start () {

int num = 10;

    // 存

    PlayerPrefs.SetInt ("Number", num);


    // 取

    int res = PlayerPrefs.GetInt ("Number");

    Debug.Log (res);

}
```

### 1.3 C#特性之属性（复习）

```
private string name;
```

```

public string Name

{

    get{ return name; }

    set{ name = value; }

}

void Start () {

    Name = "Neworigin";

    Debug.Log (Name);

}

```

#### 1.4 属性与**PlayerPrefs**结合

通过上面的知识点，我们知道，**PlayerPrefs**包含读和写两部分，刚好跟**get**和**set**对应。所以我们用**get**和**set**包装一下，**PlayerPrefs**不就可以当作普通变量使用了吗？

```

public string Name

{

    get{

        return PlayerPrefs.GetString("Name");

    }

    set{

        PlayerPrefs.SetString("Name", value);

    }

}

```

```
        PlayerPrefs.Save();

    }

}

// Use this for initialization

void Start () {

    Name = "Neworigin";

    Debug.Log (Name);

}
```

## 2 JSON: JSON实现对对象结构的数据的处理

JSON是一种轻量级的数据交换和存储格式，可以用于对数据的设备（如手机的本地存储）和向Web服务器上传，并且符合面向对象编程的思想。JSON采用完全独立于语言的文本格式，但也使用了类似于C语言家族的习惯（包括C，C++，C#，JAVA等）。这些特性使JSON成为理想的数据交换语言，易于人阅读和编写，同时也易于机器解析和生成。

### 2.1 JSON数据格式

JSON基本数据书写格式是：名称/值，如“name”:”张三”。JSON基本结构主要有以下两种。

### 2.1.1 对象

用{}包裹，用名称/值来表示对象中的一个属性，如下所示：

```
public class Person
{
    public string name;
    public int age;
    public Person(string _name, int _age)
    {
        name = _name;
        age = _age;
    }
}
```

对象Person xx = new Person("Neworigin", 20);

JSON表示就是：{"name":"Neworigin", "age":19}

### 2.1.2 数组

用[]包裹，用“,”表示并列关系，如：

```
{"people":[{"name":"Neworigin", "age":20},
{"name":"Test", "age":"23"}]}
```

```
{
```

```
  "name": jjjew
```

```
“Qinqi”:[
```

```
{},{}
```

```
]
```

```
}
```

## 2.2 序列化与反序列化

序列化就是把一个对象保存到一个文件或数据库字段中去

反序列化就是在适当的时候把这个文件再转化成原来的对象使用

JsonFx是一个类对象和JSON数据相互转换的动态链接库，下面来说它怎么使用。

注：JsonFx需要从网下下载，然后导入到Unity中才能使用。

```
using UnityEngine;
```

```
using System.Collections;
```

```
using JsonFx.Json;
```

```
public class Person
```

```
{
```

```
    public string name;
```

```
    public int age;
```

```
    public Person(string _name, int _age)
```

```
    {
```

```

        name = _name;

        age = _age;
    }
}

public class Test : MonoBehaviour {

    // Use this for initialization

    void Start () {

        Person per = new Person ("Tom", 20);

        // 将对象序列化成JSON字符串

        string strPer = JsonWriter.Serialize (per);

        Debug.Log (strPer);

        // 将JSON字符串反序列化成对象

    }

    // Update is called once per frame

    void Update () {

    }
}

```

```
}
```

## 2.3 Json数据的存储

Unity及其使用的Mono是跨平台的，符合.NET框架。我们完全可以使用System.IO下的File.ReadAllText()和File.WriteAllText()这两个函数来实现数据的存取。

```
using UnityEngine;
```

```
using System.Collections;
```

```
using JsonFx.Json;
```

```
using System.IO;
```

```
public class Person
```

```
{
```

```
    public string name;
```

```
    public int age;
```

```
    public Person(){
```

```
    }
```

```
    public Person(string _name, int _age)
```

```
    {
```

```
        name = _name;
```



```

        age = _age;
    }
}

public class Test : MonoBehaviour {

    string path = "/data.txt";

    void OnGUI () {

        if(GUILayout.Button("保存"))
        {

            Write();

        }

        if(GUILayout.Button("读取"))
        {

            Read();

        }

    }

    void Write()

    {

        Person john = new Person("John",19);

        string Json_Text = JsonWriter.Serialize(john);

```

```

        File.WriteAllText (GetDataPath () +path, Json_Text);
    }

    void Read()
    {
        string Json_Text = File.ReadAllText (GetDataPath ()
+path);

        Person john =
JsonReader.Deserialize<Person>(Json_Text);

        Debug.Log(john.name + "'s age is " +john.age);
    }

    public static string GetDataPath ()
    {
        if (Application.platform ==
RuntimePlatform.IPhonePlayer) {
            //iphone路径

            string path = Application.dataPath.Substring (0,
Application.dataPath.Length - 5);

            path = path.Substring (0, path.LastIndexOf ('/'));

            return path + "/Documents";

        } else if (Application.platform ==
RuntimePlatform.Android) {

```

```

        //安卓路径

        return Application.persistentDataPath + "/";

    } else

    {

        //其他路径

        return Application.dataPath;

    }

}
}

```

## 2.4 数据加密

通过上面数据存储后，打开data.txt文件是可以直接看到文本内容的，十分不安全，需要对文件内容进行加密。加密算法有很多种，如RC2， RC4等。这里使用的是Rijindael算法。

Rijindael是.NET里包含的一个对称加密接口，在加密和解密时都使用相同的的密钥。位于System.Security.Cryptography命名空间下。Rijindael算法符合AES堆成密码标准，密钥长度为128,192,256位之一。

加密步骤如下：

- (1) 设置字符串密钥并转化为byte数组。这里使用32的字符串转化长度为32的byte数组，也就是256位密钥。

```
static string key = "12348578902223367877723456789012";
```

```
byte[] keyArray = UTF8Encoding.UTF8.GetBytes (key);
```

(2) 创建Rijndael对象并设置参数。

```
RijndaelManaged rDel = new RijndaelManaged ();
```

```
rDel.Key = keyArray;
```

```
rDel.Mode = CipherMode.ECB;
```

```
rDel.Padding = PaddingMode.PKCS7;
```

```
ICryptoTransform cTransform = rDel.CreateEncryptor ();
```

(3) 加密。

```
// 将原始字符串转化成byte数组
```

```
byte[] toEncryptArray = UTF8Encoding.UTF8.GetBytes  
(toE);
```

```
// 加密
```

```
byte[] resultArray = cTransform.TransformFinalBlock  
(toEncryptArray, 0, toEncryptArray.Length);
```

```
// 转换回字符串并返回
```

```
return Convert.ToBase64String (resultArray, 0,  
resultArray.Length);
```

解密步骤如下：

(1) 和加密共用同样的密钥

(2) 创建RijndaelManaged对象并设置参数，和加密的第

(2) 步一致。

(3) 解密。

```
// 将加密后的字符串转化成byte数组  
  
byte[] toEncryptArray = Convert.FromBase64String (toD);  
  
// 解密  
  
byte[] resultArray = cTransform.TransformFinalBlock  
(toEncryptArray, 0, toEncryptArray.Length);  
  
// 转换回字符串并返回  
  
return UTF8Encoding.UTF8.GetString (resultArray);
```

### 3 xml(plist)文件

所谓xml文件，它是可扩展标记语言，标准通用标记语言的子集，是一种用于标记电子文件使其具有结构性的标记语言。

可扩展的标记语言是一种很像超文本标记语言的标记语言，它的设计宗旨是传输数据，而不是显示数据，它的标签没有被预定义，需要自行定义标签，它被设计为具有自我描述性，它是W3C的推荐标准。

#### 3.1 xml(plist)文件读取

有时候游戏中需要有公告栏，在公告栏中经常会出现一些文字信息。或者在游戏加载的时候，会在进度条上出现一些小提示，如下所示。



这些文字往往会放在一个xml文件中，那么这时候就需要我们去解读这个xml文件。如下有一个xml文件。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>tips</key>
  <array>
    <string>重新挑战已经通过的关卡，也可以获得金币</string>
    <string>某一关卡无法通过？您缺少的仅仅是英雄和士兵的等级罢了！</string>
    <string>优先升级城堡等级、兵力恢复速度，会带来意想不到效果</string>
    <string>选择合适的时机使用英雄技能或城防道具，可以提高获胜几率</string>
    <string>尽可能一次性出多个士兵，能够最大化提高您的军团战斗力</string>
    <string>合适的时机使用英雄技能或城防道具，可以提高您的获胜几率</string>
    <string>获得成就可以获得大量钻石，当您缺少钻石时不妨看看成就列表</string>
    <string>每个兵种的特殊各不相同，不仅仅是高级兵种才有展示的空间</string>
    <string>当连续闯关失败时，换一个战术或出兵的方式就能有所转机</string>
    <string>让士兵与敌人的等级保持一致是一个不错的主意</string>
    <string>获得成就可以获得大量钻石，缺少钻石的时候不妨看看成就列表</string>
    <string>战斗中想查看地图上的状态，只需要左右滑动屏幕就行啦</string>
  </array>
</dict>
</plist>
```

我们就对以上文件进行读取，Unity工程在这里就不多说了，请自行创建。以下是解析的代码。

```
using UnityEngine;
using System.Collections;
using System.IO;
using System.Xml; //Xml解析的库

public class XmlParserDemo : MonoBehaviour {

    string srcP = "/test.xml";

    public UILabel myL;

    string []_tips;

    void Start () {

        loadXml();

        showWenzi();

    }

    void loadXml(){
```

```

string filePath =Application.dataPath+srcP;
if(!File.Exists(filePath))return;

XmlDocument xmlDoc = new XmlDocument();//xml解析工厂
xmlDoc.Load(filePath);//加载xml文件

XmlNodeList node =
xmlDoc.SelectSingleNode("plist").ChildNodes;//SelectSingleNode获取根
节点 根据根节点获取所有的子节点

foreach(XmlElement nodeList in node){//遍历根节点下的所有
子节点(虽然只有dict)

    foreach(XmlElement xe in nodeList ){//遍历dict下的所有
子节点(key ,array)

        if(xe.Name.Equals("array")){//如果节点名称是array

            int i = 0;

            _tips = new string[xe.ChildNodes.Count];

            foreach(XmlElement wenZi in xe)//遍历array
下的子节点(既string)

                {

                    Debug.Log(wenZi.InnerText);//获取节点
的值

                    _tips[i] = wenZi.InnerText;

                    i++;

                }

            break;

        }

    }

}

}

```

```

void showWenzi()
{
    if(_tips == null || _tips.Length == 0) return;
    int idx = Random.Range(0, _tips.Length - 1); //取随机数
    myL.text = _tips[idx];
}
}

```

不过需要注意的是UILabel对象没法显示中文字体，因此需要我们创建一个中文字体提供给Label使用。关于中文字体的制作，我们在学习NGUI的时候已经讲过，这里不再说明。以下就是程序运行后的结果。



上面的xml解析可以归属于xml文件的查找操作，那么还有xml文件的增加、删除、以及修改操作，具体看如下代码。

//增删改操作

```

void createEle()
{
    XmlElement ele = _xmlDoc.CreateElement("string");
    ele.InnerText = "遍历根节点下的所有子节点(虽然只有dict)";
    XmlNodeList node =
    _xmlDoc.SelectSingleNode("plist").ChildNodes;
    foreach(XmlElement nodeList in node){//遍历根节点下的所有
子节点(虽然只有dict)
        foreach(XmlElement xe in nodeList ){//遍历dict下的所有
子节点(key ,array)
            if(xe.Name.Equals("array")){//如果节点名称是array

```



```
//          xe.AppendChild(ele);
//          xe.RemoveChild(xe.LastChild);
//
//          xe.ReplaceChild();
//      }
//  }
//  }
//  //保存
//  _xmlDoc.Save(_filePath);
//  }
```