4

第 4 章

Protobuf 在游戏中运用

Protobuf 全称是 Protocol Buffers，它是一种轻便高效的结构化数据存储格式，可以用于结构化数据串行化，很适合做数据存储或 RPC 数据交换格式。Protobuf 可用于通讯协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式，目前提供了 C++、Java、Python、C#等多种语言的 API。Protobuf 是 Google 开源的序列化和反序列化工具，主要用于网络游戏的消息结构体定义上。相对于 XML 文件和 Json 文件，它的性能更好，效率更高。网站 http://code.google.com/p/protobuf/downloads/list 上可以下载 Protobuf 的源代码，它的文件格式以.proto 为扩展名。为了便于大家更好的理解其使用原理，下面就通过结合案例的方式给大家讲解。

4.1 Protobuf 消息结构体定义

使用 Protobuf 定义消息结构体时首先要明白其语法结构。Protobuf 定义的消息由至少一个字段组合而成，类似于 C 语言中的结构，每个字段都有一定的格式，限定修饰 required、optional、repeated 三个修饰符。

- required 修饰符表示一个必须字段。必须相对于发送方，在发送消息之前必须设置该字段的值，对于接收方，必须能够识别该字段的意思。发送之前没有设置 required 字段或者无法识别 required 字段都会引发编解码异常，导致消息被丢弃。
- optional 修饰符表示一个可选字段。可选相对于发送方，在发送消息时，可以有选择性的设置或者不设置该字段的值。对于接收方，如果能够识别可选字段就进行相应的处理，如果无法识别，则忽略该字段，消息中的其他字段正常处理。由于 optional 字段的特性，很多接口在升级版本中都把后来添加的字段统一设置为 optional 字段，这样老的版本无须升级程序也可以与新的软件进行通信，只不过新的字段无法识别而已，因为并不是每个节点都需要新的功能，因此可以做到按需升级和平滑过渡。
- repeated 表示该字段可以包含 0～*N* 个元素。其特性和 optional 一样，但是每一次可以包含多个值。它可以看作是在传递一个数组的值或者 List 列表数值。开发网络游戏时，经常会定义消息结构体，这些消息结构体在客户端和服务器中都会用到，所以只需要定义一套就可以了。现在移动端大部分用户都在使用 Unity 引擎开发，所以这些结构体需要转成 C#语言。下面从结构体定义开始讲起。

4.2 编写 Protobuf 结构体

网络消息的定义通常会使用 Json 文件、二进制文件或者自定义结构体。现在使用 Protobuf 定义消息结构体的公司越来越多，它逐渐成为消息结构体定义的主流，这也要感谢 Google 提供了一个开源的序列化和反序列化工具。本节以实际项目开发的案例给大家介绍一下网络消息结构体的定义，任何大型网络游戏都有角色的定义，因此首先从角色的定义说起。角色消息包括很多的属性，定义结构体如下所示，其中 message 是结构体的修饰符。

```
//角色信息结构
message msgcharinfo
{
    optional uint32 uaid = 1;       //用户 ID
    optional uint32 charid = 2;     //角色 ID
    optional uint32 kind = 3;       //角色种类
    optional string name = 4;       //角色名字
    optional string head = 5;       //头像 ID
    optional uint32 level = 6;      //角色等级
    optional uint32 exp = 7;        //角色经验
```

```
    optional uint32 phypower = 8;   //物理攻击
    optional uint32 leadership = 9; //领导标记
    optional uint32 friendnum = 10; //朋友数量
    optional uint32 gamecoin = 11;  //游戏货币
    optional uint32 diamond = 12;   //钻石数量
};
```

以上是网络游戏中完整的角色定义，包括用户 id、游戏中角色 id 等信息。它存放的文件扩展名为.proto。结构体中各项的修饰都是 optional，也就是可选项，可以不用赋值。protobuf 自身定义的文件是文本文件，如果将该文件直接放到 Unity 工程中，Unity 是不会识别的，这就需要将其转成 Unity 可识别的脚本 C#文件。再举一个枚举定义的例子，代码如下所示。

```
//初始化角色奖励信息
enum enumGetCharRewardResult
{
    Success  = 0;          //成功获取角色
    SystemError = 1;       //系统错误
    NewChar = 2;           //创建新角色奖励信息
};
```

该结构体是以 enum 修饰的枚举类型，里面有三项，枚举定义和 C++或者 Java 定义类似，枚举定义的内部成员不需要任何符号修饰。假设以上内容是 common.proto 文件定义的，下面我们再定义一个 proto 文件，如下所示。

```
package clientmsg;
import "common.proto";
message C2SNameRepetition
{
    optional msgcharinfo charinfo = 1;    //创建角色
    optional uint32 mapid = 2;
    optional uint32 cityid = 3;
};
```

给大家解释一下代码。第一行 package clientmsg;表示的是模块的封装，其含义类似 C++或者 C#的 namespace 命名空间。第二行 import "common.proto";表示的是引用该文件，目的是需要用到该文件已定义的结构体，例如，message C2SNameRepetition 定义的内容中的语句 optional msgcharinfo charinfo = 1;引用的是 common.proto 文件中已定义的 msgcharinfo 的结构体，Protobuf 支持这种引用关系，从中可以看出 Protobuf 语言也是比较灵活的，文件与文件之间是可以互相引用的，接下来开始介绍转换工具的制作。

4.3 Protobuf 转换工具制作

定义好了 proto 文件后，如果直接放到 Unity 中，它只能被作为文本文件，这不是开发者想要的，因为要在程序中使用定义好的结构体，需要一个能将其转换成 C#脚本文件的工具。下面告诉大家制作该工具需要做哪些工作。制作工具需要的库文件可以在网上下载到，就是已编译好的库工程，主要内容如图 4-1 所示。

名称	修改日期	类型	大小
common.xslt	2013/5/7 21:06	XSLT Stylesheet	6 KB
csharp.xslt	2013/5/7 21:06	XSLT Stylesheet	36 KB
descriptor.proto	2013/5/7 21:06	PROTO 文件	16 KB
Licence.txt	2013/5/7 21:06	文本文档	1 KB
protobuf-net.dll	2013/5/7 21:06	应用程序扩展	137 KB
protobuf-net.Extensions.dll	2013/5/7 21:06	应用程序扩展	14 KB
protobuf-net.Extensions.pdb	2013/5/7 21:06	Program Debug...	28 KB
protobuf-net.Extensions.XML	2013/5/7 21:06	BaiduBrowser H...	5 KB
protobuf-net.pdb	2013/5/7 21:06	Program Debug...	426 KB
protobuf-net.xml	2013/5/7 21:06	BaiduBrowser H...	87 KB
protoc.exe	2013/5/7 21:06	应用程序	1,573 KB
protoc-license.txt	2013/5/7 21:06	文本文档	2 KB
protogen.exe	2013/5/7 21:06	应用程序	1,743 KB
protogen.pdb	2013/5/7 21:06	Program Debug...	82 KB
vb.xslt	2013/5/7 21:06	XSLT Stylesheet	42 KB
xml.xslt	2013/5/7 21:06	XSLT Stylesheet	1 KB

图 4-1　工具库目录

接下来需要写一个批处理文件执行 proto 批量转换操作。假设上述库文件是在文件路径：3Party\protobuf-net\net 下面，制作的工具文件的扩展名为.bat，批处理文件完整内容如下所示。

```
@echo off
set tool=..\3Party\protobuf-net\net

rem ================================================
rem  Support
set proto=common.proto
%tool%\protogen.exe -i:%proto% -o:%proto%.cs -q

set proto=login.proto
%tool%\protogen.exe -i:%proto% -o:%proto%.cs -q

set proto=begingame.proto
```

```
%tool%\protogen.exe -i:%proto% -o:%proto%.cs -q
pause
```

其中语句 set tool=..\3Party\protobuf-net\net 表示的是库文件所在的目录；set proto=common.proto 表示的是要转换的 proto 文件名字；%tool%\protogen.exe-i:%proto%-o:%proto%.cs-q 表示的是调用上述目录下的库，将 common.proto 转化成 common.proto.cs 文件，依此类推。因为我们定义的 commo.cs 是公用的文件，下面的文件都会引用到该文件的内容，同时把 proto 文件复制到与扩展名为.bat 相同的文件夹下面。其执行的效果如图 4-2 所示。

名称	修改日期	类型	大小
begingame.proto	2014/11/5 14:49	PROTO 文件	3 KB
begingame.proto.cs	2016/3/26 10:37	Visual C# Sourc...	28 KB
common.proto	2014/11/5 14:49	PROTO 文件	19 KB
common.proto.cs	2016/3/26 10:28	Visual C# Sourc...	132 KB
login.proto	2014/9/13 14:41	PROTO 文件	4 KB
login.proto.cs	2016/3/26 10:34	Visual C# Sourc...	32 KB
make-protobuf.bat	2016/3/26 10:28	Windows 批处理...	1 KB

图 4-2　转换文件示意图

make-protobuf.bat 就是执行的批处理程序，执行的结果就是生成对应的 cs 文件，然后将 cs 文件拖放到 Unity 的目录下面。以 login.proto.cs 文件为例，生成的 cs 文件内容如图 4-3 所示。

```
using common;

namespace clientmsg
{
  [global::System.Serializable, global::ProtoBuf.ProtoContract(Name=@"c2s_login")]
  public partial class c2s_login : global::ProtoBuf.IExtensible
  {
    public c2s_login() {}


    private string _name = "";
    [global::ProtoBuf.ProtoMember(1, IsRequired = false, Name=@"name", DataFormat = global::ProtoBuf.DataFormat.Default)]
    [global::System.ComponentModel.DefaultValue("")]
    public string name
    {
      get { return _name; }
      set { _name = value; }
    }

    private string _pwd = "";
    [global::ProtoBuf.ProtoMember(2, IsRequired = false, Name=@"pwd", DataFormat = global::ProtoBuf.DataFormat.Default)]
    [global::System.ComponentModel.DefaultValue("")]
    public string pwd
    {
```

图 4-3　转成 cs 文件示意图

由于篇幅所限，只截取一部分内容，第一行表示的是引用 common，下面是命名空间以及类声明。眼尖的读者可能注意到了一个细节就是类前面的修饰符 partial，它属于一个局部类型，

局部类型允许我们将一个类、结构或接口分成几个部分，分别实现在几个不同的 cs 文件中。给大家普及一下 partial 的基础知识，一是类型特别大，不宜放在一个文件中实现；二是一个类型中的一部分代码为自动化工具生成的代码，不宜与我们自己编写的代码混合在一起；三是需要多人合作编写一个类。这几种情况适用于 partial 修饰。接下来介绍一下如何在 Unity 中使用。

4.4 Protobuf 文件在 Unity 中的运用

在 Unity 中使用定义的 Protobuf 文件，首先需要把 protobuf-net 的源文件放到 Unity 目录下，源文件的下载地址是：https://github.com/mgravell/protobuf-net/tree/master/protobuf-net，使用源文件的目的是为了实现 Protobuf 在 Android 和 iOS 平台同时使用，代码可以直接在 Google 提供的官网上下载，拖放到 Unity 中的效果如图 4-4 所示。

这些前期工作完成后就可以直接调用 Protobuf 源代码中的库函数进行序列化和反序列化。下面将生成的 cs 脚本文件拖放到 Unity 中，效果如图 4-5 所示。

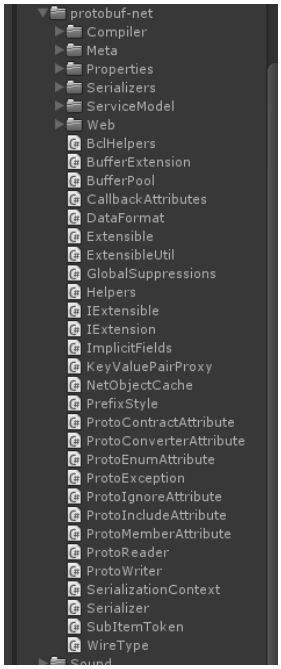

图 4-4　Protobuf 库源文件

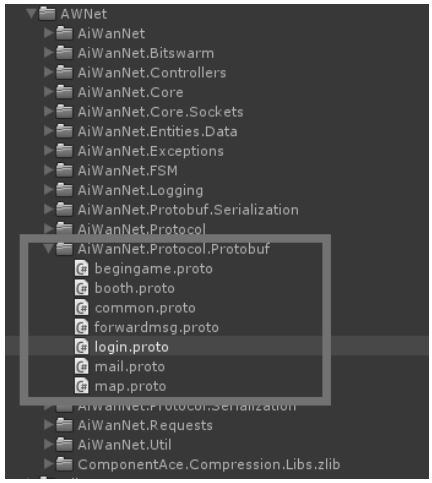

图 4-5　生成的 Protobuf 的 cs 代码

接下来介绍一下如何在 Unity 中使用它。在使用定义好的结构体时，需要在文件中加入引用头文件的 using clientmsg，然后在函数中首先 new 一个对象，如下所示。

```
clientmsg.c2s_login msg = new clientmsg.c2s_login();
```

然后对结构体填充数值，如下所示。

```
    msg.name = Global.userInputName;
    msg.pwd = Global.password;
    messageContentLen += msg.name.Length;
    messageContentLen += msg.pwd.Length;
```

最后可以将结构体发送到服务器 SendProtoBufMsg(msg, awnet);上，这样整个 Protobuf 文件的使用就结束了，希望对大家有所帮助。

4.5　小结

Protobuf 是 Google 开源的，它被用于序列化和反序列化。以前定义消息结构时，采用的是自定义结构体，扩展起来非常麻烦，有时服务器改动了，客户端没改动过，经常会导致各种问题。后期使用了 Protobuf 彻底解决了这个问题，服务器只需要定义一份消息文件即可。Protobuf 提供了转化成不同语言的工具，比如可以转化成 C++、C#、Java 等。Protobuf 底层已经实现了序列化反序列化接口，使用时只需调用，无须自己实现。

5

第 5 章

游戏中的文本文件加密

所有的游戏开发都离不开文本文件的加载读取，文本文件主要是方便策划随时根据游戏调整数值。首先策划要根据游戏设计基本的数值，游戏都是采用数值驱动的，策划的数值体现在文本文件上。游戏的文本文件格式非常多，游戏开发使用的文件类型有：XML 文件、Json 文件、二进制文件、csv 文件以及自定义格式文件等。文本文件的作用是方便策划配置，随时修改游戏数值。本章介绍的是对 csv 文件的加密和读取，告诉读者如何使用程序代码封装加载和读取文本文件的接口。文本文件的加载流程如图 5-1 所示。

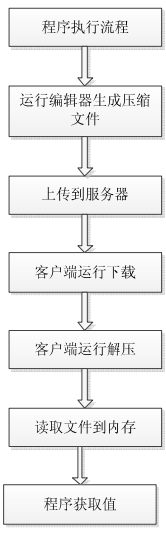

图 5-1　文件加密流程

为了防止游戏开发使用的文本文件被破解，要将其加密压缩。操作步骤：首先运行编辑器把所有的配置文件压缩到 zip 文件中，再将压缩后的文件上传到资源服务器，客户端运行时通过文本文件的对比把版本高的加密压缩文件下载到本地，客户端在程序中进行解压缩，读取文本文件到内存中存储，程序根据提供的接口从内存中取出数据使用。以上就是整个文本文件加载的思路，接下来看看配置文件的文本格式。该技术已经在游戏产品开发中运用并且已上线运营，下面给大家看看 csv 文件格式。

5.1　配置文件格式

csv 文件是 excel 表格转化过来的文本文件，操作方式非常简单，将 excel 另存为 csv 格式就完成了从 excel 到 csv 的转化，这就是程序需要的文件。如果将其直接拖到 Unity 工程中是不可以的，需要将其扩展名 csv 改成 txt 格式。最终的 csv 文件格式如下所示。

```
Id,Category,PropId,PropName,CurrencyType,Price,UpgradePrice
1_1,1,1,蛮族,1,3300,0
1_2,1,2,泰坦,2,2500,0
1_3,1,3,兽王,2,396000,0
1_4,1,4,浪人,1,2640,0
1_5,1,5,零式,2,369600,0
1_6,1,6,侍魂,1,2970,0
1_7,1,7,执政官,1,3300,0
1_8,1,8,天启,1,3300,0
```

文件第一行是表格的结构属性，结构属性下面是对应的各个字段值，各个字段值之间用逗号分隔开，便于文件加载时区分不同的字段，非常方便。字段了解清楚了，接下来我们封装文件加载接口。

5.2　文件加载接口

程序需要封装接口实现 csv 文件加载读取。在这里由于文本文件的内容是常驻内存的，所以采用的是 static 静态类的实现方式，网上也有很多关于文本文件的加载，在这里我封装了一套适用于游戏的接口。完整的代码如下所示。

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System;
using System.IO;
using System.Reflection;
using System.Linq;
```

```
///作用：文件读取、加载
///Author:Jxw
///Time:2015/10/22

#if UNITY_EDITOR
#pragma warning disable 0649 //检测到无法访问的代码
#endif
namespace BIEFramework.Manager {

    public static class FileManager {
        private static bool m_bInitAssetBundle;

        //静态数据
        private static List<ArmorInfo> armorInfoList = new List<ArmorInfo>();
        private static List<BumperInfo> bumperInfoList = new List<BumperInfo>();
        private static List<CarInfo> carInfoList = new List<CarInfo>();
        private static List<CarSkinInfo> carskinInfoList = new List<CarSkinInfo>();
        private static List<NitroInfo> nitroInfoList = new List<NitroInfo>();
        private static List<PerformanceInfo> performanceInfoList = new List<PerformanceI
nfo>();
        private static List<WeaponInfo> weaponInfoList = new List<WeaponInfo>();
        private static List<GuideGirlInfo> guideGirlInfoList = new List<GuideGirlInfo>();
        private static List<SkillBaseInfo> skillBaseInfoList = new List<SkillBaseInfo>();
        private static List<SkillModel> skillModelList=new List<SkillModel>();
        private static List<SkillInfo> skillInfoList = new List<SkillInfo>();
        private static List<NPCInfo> npcInfoList = new List<NPCInfo>();
        private static List<DefaultPlayerInfo> defaultPlayerInfoList = new List<DefaultP
layerInfo>();
        private static List<MapInfo> mapInfoList = new List<MapInfo>();
        private static List<TaskInfo> taskInfoList = new List<TaskInfo>();
        private static List<PropInfo> propInfoList = new List<PropInfo>();
        private static List<DailyReward> dailyRewardList = new List<DailyReward>();
        private static List<GoodInfo> goodList = new List<GoodInfo>();
        private static List<DailyTask> dailyTaskList = new List<DailyTask>();
        private static List<PopupInfo> popupInfolList = new List<PopupInfo>();
        //数据文件位置
        public const string dataFolder = "/data/";
private static bool isDataInit = false;
        public static void Init() {
            if (isDataInit) {
                return;
```

```
            }
            isDataInit = true;
            InitBundle();
        }

        public static void InitBundle() {
            if (!m_bInitAssetBundle) {
                m_bInitAssetBundle = true;
     //解释文件函数调用
                ParserFromTxtFile<ArmorInfo>(armorInfoList);
                ParserFromTxtFile<BumperInfo>(bumperInfoList);
                ParserFromTxtFile<CarInfo>(carInfoList);
                ParserFromTxtFile<CarSkinInfo>(carskinInfoList);
                ParserFromTxtFile<NitroInfo>(nitroInfoList);
                ParserFromTxtFile<PerformanceInfo>(performanceInfoList);
                ParserFromTxtFile<WeaponInfo>(weaponInfoList);
                ParserFromTxtFile<GuideGirlInfo>(guideGirlInfoList);
                ParserFromTxtFile<SkillModel>(skillModelList);

                ParserFromTxtFile<SkillInfo>(skillInfoList);
                ParserFromTxtFile<MapInfo>(mapInfoList);
                ParserFromTxtFile<TaskInfo>(taskInfoList);
                ParserFromTxtFile<NPCInfo>(npcInfoList);
                ParserFromTxtFile<DefaultPlayerInfo>(defaultPlayerInfoList);
                ParserFromTxtFile<PropInfo>(propInfoList);
                ParserFromTxtFile<GoodInfo>(goodList);
                ParserFromTxtFile<DailyReward>(dailyRewardList);
                ParserFromTxtFile<DailyTask>(dailyTaskList);
                ParserFromTxtFile<PopupInfo>(popupInfolList);
}

   public static void ParserFromTxtFile<T>(List<T> list, bool bRefResource = false) {
            string asset = null;

            //获取文件路径
            string file = ((DataPathAttribute)Attribute.GetCustomAttribute(typeof(T), ty
peof(DataPathAttribute))).fiePath;

            if (bRefResource) {
                asset = ((TextAsset)Resources.Load(file, typeof(TextAsset))).text;
            } else {
```

```
                asset = File.ReadAllText(Util.DataPath + file+".txt");
            }

            StringReader reader = null;

            try {
                bool isHeadLine = true;
                string[] headLine = null;
                string stext = string.Empty;
                reader = new StringReader(asset);
                while ((stext = reader.ReadLine()) != null) {
                    if (isHeadLine) {
                        headLine = stext.Split(',');
                        isHeadLine = false;
                    } else {
                        string[] data = stext.Split(',');
                        list.Add(CreateDataModule<T>(headLine.ToList(), data));
                    }
                }
            } catch (Exception exception) {
                Debug.Log("file:" + file + ",msg:" + exception.Message);
            } finally {
                if (reader != null) {
                    reader.Close();
                }

            }
        }

        private static T CreateDataModule<T>(List<string> headLine, string[] data) {
            T result = Activator.CreateInstance<T>();
            FieldInfo[] fis = typeof(T).GetFields(BindingFlags.Public | BindingFlags.Ins
tance);
            foreach (FieldInfo fi in fis) {
                string column = headLine.Where(tempstr => tempstr == fi.Name).FirstOrDef
ault();
                if (!string.IsNullOrEmpty(column)) {
                    string baseValue = data[headLine.IndexOf(column)];
                    object setValueObj = null;
                    Type setValueType = fi.FieldType;
                    if (setValueType.Equals(typeof(short))) {
```

```
                        setValueObj = string.IsNullOrEmpty(baseValue.Trim()) ? (short)0
:  Convert.ToInt16(baseValue);
                    } else if (setValueType.Equals(typeof(int))) {
                        setValueObj = string.IsNullOrEmpty(baseValue.Trim()) ? 0 : Conve
rt.ToInt32(baseValue);
                    } else if (setValueType.Equals(typeof(long))) {
                        setValueObj = string.IsNullOrEmpty(baseValue.Trim()) ? 0 : Conve
rt.ToInt64(baseValue);
                    } else if (setValueType.Equals(typeof(float))) {
                        setValueObj = string.IsNullOrEmpty(baseValue.Trim()) ? 0 : Conve
rt.ToSingle(baseValue);
                    } else if (setValueType.Equals(typeof(double))) {
                        setValueObj = string.IsNullOrEmpty(baseValue.Trim()) ? 0 : Conve
rt.ToDouble(baseValue);
                    } else if (setValueType.Equals(typeof(bool))) {
                        setValueObj = string.IsNullOrEmpty(baseValue.Trim()) ? false : C
onvert.ToBoolean(baseValue);
                    } else if (setValueType.Equals(typeof(byte))) {
                        setValueObj = Convert.ToByte(baseValue);
                    } else {
                        setValueObj = baseValue;
                    }
                    fi.SetValue(result, setValueObj);
                }
            }
            return result;
        }
       public static ArmorInfo FindArmorInfoFromId(int id) {
            ArmorInfo data = null;
            data = armorInfoList.Find(x => x.Id == id);
            if (data == null) {
                Debugger.Log("Error : Not Found In ArmorInfo. ID :" + id);
            }
            return data;
        }

        public static List<ArmorInfo> FindarmorInfoList() {
            return armorInfoList;
        }
///<summary>
///注释，各个数据对应的文件
```

```
///</summary>
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited = false)]
public class DataPathAttribute : Attribute {
    public string fiePath { get; set; }
    public DataPathAttribute(string _fiePath) {
        fiePath = _fiePath;
    }
}
```

下面把代码的编写思路给读者理顺一下。加载文本文件，首先声明关于文本文件的结构体列表，文本文件的行表示的就是一个结构体，可以根据行的属性定义数据文件的结构体，根据表的结构体定义用于存储文本文件 List 表，列表中的结构体会在下面给大家介绍，文件中声明的语句如下：

```
        private static List<ArmorInfo> armorInfoList = new List<ArmorInfo>();
        private static List<BumperInfo> bumperInfoList = new List<BumperInfo>();
        private static List<CarInfo> carInfoList = new List<CarInfo>();
        private static List<CarSkinInfo> carskinInfoList = new List<CarSkinInfo>();
```

文件存储的列表定义好了，接下来就是读取文件，解释文件，调用函数。

```
  public static void InitBundle()
```

在 InitBundle()函数中调用了解释文本文件的函数。

```
   ParserFromTxtFile<ArmorInfo>(armorInfoList);
   ParserFromTxtFile 函数的作用是将加载到的配置文件的内容存储到已定义好的列表中。
   ParserFromTxtFile 函数采用了模版的定义方式内容如下所示。
   public static void ParserFromTxtFile<T>(List<T> list, bool bRefResource = false) {
            string asset = null;

            //获取文件路径
            string file = ((DataPathAttribute)Attribute.GetCustomAttribute(typeof(T), ty
peof(DataPathAttribute))).fiePath;

            if (bRefResource) {
                asset = ((TextAsset)Resources.Load(file, typeof(TextAsset))).text;
            } else {
                asset = File.ReadAllText(Util.DataPath + file+".txt");
            }

            StringReader reader = null;
```

```
            try {
                bool isHeadLine = true;
                string[] headLine = null;
                string stext = string.Empty;
                reader = new StringReader(asset);
                while ((stext = reader.ReadLine()) != null) {
                    if (isHeadLine) {
                        headLine = stext.Split(',');
                        isHeadLine = false;
                    } else {
                        string[] data = stext.Split(',');
                        list.Add(CreateDataModule<T>(headLine.ToList(), data));
                    }
                }
            } catch (Exception exception) {
                Debug.Log("file:" + file + ",msg:" + exception.Message);
            } finally {
                if (reader != null) {
                    reader.Close();
                }

            }
        }
```

这个函数的作用就是加载并解释 csv 文件的代码，采用的也是一行一行的读取方式。

下面是文件结构体的定义。

```
public class CombatInfo {
#region 武器数据模型
[System.Serializable]
[DataPath(FileManager.dataFolder + "weapon")]
public class WeaponInfo {
    public int Id;
    public int Type;
    public string Name;
    public string Description;
    public string Resources;
    public string Quality;
    public int Attack;
    public int UpgradeAttack;
```

```
    public int Magazine;
    public int Range;
    public int Speed;
    public float Rateoffire;
}
#endregion

#region 保险杠数据模型
[System.Serializable]
[DataPath(FileManager.dataFolder + "bumper")]
public class BumperInfo {
    public int Id;
    public string Name;
    public string Description;
    public string Resources;
    public int Damage;
    public int UpgradeDamage;
    public int Defence;
    public int UpgradeDefence;
    public int ParryHurt;
}

#endregion
#region 装甲数据模型
[System.Serializable]
[DataPath(FileManager.dataFolder + "armor")]
public class ArmorInfo {
    public int Id;
    public string Name;
    public string Description;
    public string Resources;
    public int Defence;
    public int UpgradeDefence;
}

#endregion

#region shop 数据模型
[DataPath(FileManager.dataFolder+"shop")]
public class GoodInfo{
    public string Id;
```

```
    public string Category;
    public string PropId;
    public string PropName;
    public int CurrencyType;
    public string Price;
    public float UpgradePrice;
}
#endregion

#region 汽车数据模型
[DataPath(FileManager.dataFolder + "car")]
public class CarInfo {
    public int Id;
    public string Name;
    public string Description;
    public string Templat;
    public string Quality;
    public string Resources;
    public int Control;
    public int MaxRpm;
    public int Tire;
    public float TireRadius;
    public int Engine;
    public int Turbine;
    public int Drivetrain;
    public float DrivetrainRatio;
    public int Nitrous;
    public int SkinId;
    public string CarSkin;

    public int[] CarSkinIds {
        get {
            string[] strs = CarSkin.Split(';');
            int[] result = new int[strs.Length];
            for (int i = 0; i < strs.Length; i++) {
                result[i] = int.Parse(strs[i]);
            }
            return result;
        }
    }
    public int WheelSkin;
```

```
    public int HP;
    public int NPCHP;
}
#endregion
[DataPath(FileManager.dataFolder + "performance")]
public class PerformanceInfo {
    public int Id;
    public int Type;
    public string Name;
    public string Description;
    public string Resources;
    public int AddMaxRpm;
    public string Resources2;
    public string Resources3;
}
```

语句 [DataPath(FileManager.dataFolder + "weapon")] 表示的是文本文件所在路径，FileManager.dataFolder 可以修改成手机端的存储地址，路径下面是结构体的声明，在使用时可以直接通过 FileManager 调用已经封装好的接口即可实现，比如 List<ArmorInfo>=FileManager.FindarmorInfoList()可以获取整个文件的列表，也可以通过其 Id 获取到某一列的值，再比如语句 ArmorInfo info=FileManager.FindArmorInfoFromId(id)可以获取到某一行的值，逻辑程序调用非常方便，文本文件加载读取完成。补充一下，DataPath 的路径定义是根据已定义的函数实现的，如下所示。

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited = false)]
public class DataPathAttribute : Attribute {
    public string fiePath { get; set; }
    public DataPathAttribute(string _fiePath) {
        fiePath = _fiePath;
    }
```

接下来我们考虑一下安全问题，也就是文本文件加密。

5.3 文本文件加密算法及应用

移动端游戏经常被一些玩家破解成白包，但是为了安全性，开发者还是需要使用算法对文

本文件加密。加密的算法非常多，比如通常使用的是 MD5 算法、OBFS 算法和 SHA512 算法等。由于 MD5 算法经常使用，网上也有现成的代码，所以本节略过，直接讲 OBFS 算法和 SHA512 算法。为了便于大家理解，先把加密算法代码奉上。

```
//OBFS 加密算法
private static string OBFS(string str)
{
   int length = str.Length;
   var array = new char[length];
   for (int i = 0; i < array.Length; i++)
   {
      char c = str[i];
      var b = (byte)(c ^ length - i);
      var b2 = (byte)((c >> 8) ^ i);
      array[i] = (char)(b2 << 8 | b);
   }
   return new string(array);
}

 //SHA512 加密算法
 public static string GetSHA512Password(string password)
 {
     byte[] bytes = Encoding.UTF7.GetBytes(password);
     byte[] result;
     SHA512 shaM = new SHA512Managed();
     result = shaM.ComputeHash(bytes);
     StringBuilder sb = new StringBuilder();
     foreach (byte num in result)
     {
         sb.AppendFormat("{0:x2}", num);
     }
     return sb.ToString();
 }
```

以上两个算法实现了文本文件的加密，函数的参数是开发者自己定义的字符串，然后在该字符串的基础上通过算法加密生成新的字符串用于压缩文件加密。下面列出配置文件，并且调用 OBFS 函数或者 SHA512 函数对文件进行加密，返回的是经过加密的字符串，同时调用函数 SaveConfigXMLTOZip 对文件进行压缩加密。对应的函数语句如下所示。

```
private const string configurationFile = "config.txt";
```

```
    private const string localizationFile = "translations.txt";
     private /*const*/ string configurationZipPwd = OBFS("~Ũ□;o▯ۇ▯ □ϖ □▯▯");
private /*const*/ string configurationZipPwd = GetSHA512Password("ۇ※█ ぷ┤▓ちЖ ╒ Ψ≮≯ゆű
ň ξ ζ");
#if UNITY_EDITOR
    protected void SaveConfigXMLToZip()
    {
        using (ZipFile zipFile = new ZipFile(Encoding.UTF8))
        {
            zipFile.Password = configurationZipPwd;
            zipFile.AddEntry(configurationFile, configuration.bytes);
            zipFile.AddEntry(localizationFile, localization.bytes);
            string zipPath = Path.Combine(Application.persistentDataPath,
configurationZipFile);
            LogTool.Log("Saving configuration in \"" + zipPath + "\"");
            zipFile.Save(zipPath);
        }
    }
#endif
```

文件的压缩是在编辑模式下，程序运行时会将文本文件压缩，同时把加密的密码赋值给它。在程序启动时，先把资源服务器加载文件的版本号与本地的版本号对比，决定下载需要的文本文件。

```
#region Coroutines
    IEnumerator DownloadVersionFile()
    {
        Asserts.Assert(!downloadingVersionFile);
        downloadingVersionFile = true;

        WWW versionLoader = new WWW(configurationZipURL + versionFile + "?nocache=" +
Environment.TickCount);

        while (!versionLoader.isDone)
        {
            yield return new WaitForEndOfFrame();
        }

        if (versionLoader.isDone && string.IsNullOrEmpty(versionLoader.error))
        {
            versionString = versionLoader.text;
```

```
      }
      else
         versionString = version.text;

      versionLoader.Dispose();

      LogTool.Log("VERSION NUMBER: " + versionString);

         downloadingVersionFile = false;

      PlayerPrefs.SetInt("last_vn", lastVersionNumber);
   }
```

它是通过 WWW 下载的，先从资源服务器下载文本文件，接下来下载文件的压缩包，函数代码如下所示。

```
   IEnumerator DownloadZip()
   {
      Asserts.Assert(!downloadingZip);
      downloadingZip = true;
      WWW zipLoader = new WWW(configurationZipURL + configurationZipFile + "?nocache=" +
Environment.TickCount);

      while (!zipLoader.isDone)
      {
         if (stopDownloading)
         {
            downloadingZip = false;
            stopDownloading = false;

            LogTool.Log("Download configuration STOPPED!");
         }

         yield return new WaitForEndOfFrame();
      }

      if (zipLoader.isDone && string.IsNullOrEmpty(zipLoader.error))
      {
         LogTool.Log("**** PELLE: DOWNLOADING ZIP COMPLETED! Duration: " +
(Time.realtimeSinceStartup - startTime));
```

```
         using (FileStream fs = new FileStream(Path.Combine(Application.persistentDataPath,
configurationZipFile), FileMode.Create))
         {
            fs.Seek(0, SeekOrigin.Begin);
            fs.Write(zipLoader.bytes, 0, zipLoader.bytes.Length);
            fs.Flush();
         }

         zipLoader.Dispose();

         if (!downloadingZip)
         {
            LogTool.Log("Download configuration OK!");
            yield break;
         }
         else
            LogTool.Log("Download configuration OK, configurations will be loaded from new
zip!");
      }
      else
      {
         zipLoader.Dispose();
         downloadingZip = false;
         stopDownloading = false;

         yield break;
      }

      if (!dataLoaded && !stopDownloading)
         this.TryLoadingXMLsFromZip();

      downloadingZip = false;
      stopDownloading = false;
   }
```

最后一步就是解压缩文件并且解释文本文件，函数代码如下所示。

```
 protected void TryLoadingXMLsFromZip()
   {
      string zipPath = Path.Combine(Application.persistentDataPath, configurationZipFile);
```

```
      if (!File.Exists(zipPath))
      {
         LogTool.Log("Configuration not found!");
         this.ParseConfigXML(configuration.text, false);
         this.ParseLocalizationXML(localization.text, false);
         return;
      }

      using (ZipFile zipFile = new ZipFile(zipPath, Encoding.UTF8))
      {
         zipFile.Password = configurationZipPwd;

         ZipEntry xmlConfEntry  = zipFile[configurationFile],
                xmlLocaleEntry = zipFile[localizationFile];
         if (null == xmlConfEntry || null == xmlLocaleEntry)
         {
            LogTool.Log("Downloaded configuration INVALID!");
            this.ParseConfigXML(configuration.text, false);
            this.ParseLocalizationXML(localization.text, false);
            return;
         }

         using (MemoryStream ms = new MemoryStream())
         {
            xmlConfEntry.Extract(ms);

            string xmlText = Encoding.UTF8.GetString(ms.GetBuffer(), 0,
ms.GetBuffer().Length);
            this.ParseConfigXML(xmlText, true);

            ms.Seek(0, SeekOrigin.Begin);
            xmlLocaleEntry.Extract(ms);

            xmlText = Encoding.UTF8.GetString(ms.GetBuffer(), 0, ms.GetBuffer().Length);
            this.ParseLocalizationXML(xmlText, true);
         }
      }
   }
```

到这里文本文件加密算法就写完了，文本文件挂接如图 5-2 所示。

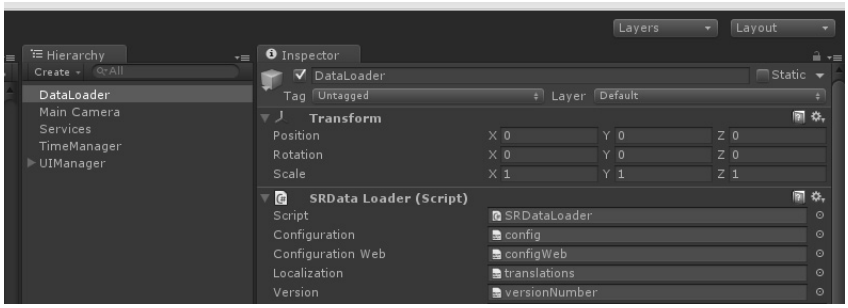

图 5-2　文本文件挂接

先把加载的脚本文件挂接到对象上，SRData Loader 就是程序的加载脚本，这样程序在启动时会从资源服务器下载，然后将其加载到内存中，函数代码如下所示。

```
   new void Awake()
   {
      SRDataLoader.Instance = this;

#if UNITY_EDITOR && !UNITY_WEBPLAYER
      this.SaveConfigXMLToZip();
#endif
      dataLoaded = false;
      downloadingZip = false;
      stopDownloading = false;

      startTime = Time.realtimeSinceStartup;
      lastVersionNumber = PlayerPrefs.GetInt("last_vn", 0);

      this.TryLoadingXMLsFromZip();

         #elif !UNITY_EDITOR
      if (Application.internetReachability != NetworkReachability.NotReachable)
      {
         this.StartCoroutine(this.DownloadVersionFile());
         this.StartCoroutine(this.DownloadZip());
      }
#endif

      DontDestroyOnLoad(gameObject);
   }
```

我们在 Awake 函数中完成了版本号和版本文件的下载，然后按照程序流程执行解压，读取文本文件操作。

5.4　小结

配置文件对于游戏来说是必备的，数据驱动已经成为游戏开发必备的条件。策划会根据游戏的玩法调整游戏中的数据表现，比如玩家与怪物战斗的血量配置、背包物品的属性数据配置、玩家自身的属性数据配置、关卡的难度配置，等等。配置文件的格式非常多，本章主要是介绍了 csv 的加载读取方式。策划数据表的配置大部分都是 excel 表格，将它们转成 csv 格式非常方便。