Udon Script 分析

By ParaN3xus

Outline

I 引言

我想知道一些 VRChat 地图的脚本逻辑, 但是 VRChat 地图的脚本都被编译成了一些神秘的, 无法被 AssetRipper 轻易解析的 MonoBehaviour.

我不太了解 VRChat 的世界创作生态, 但是朋友告诉我这是 Udon Script, 还告诉我他也没法解读这些产物.

既然如此, Challenge Accepted!

II Udon Program

Udon Script 的编译产物.

II.1 资产

使用 AssetRipper 解包地图后, 能得到大量 AssetRipper 无法正确解析的 MonoBehaviour 资产文件. 其中一些 MonoBehaviour 资产包含一个很长的 `serializedProgramCompressedBytes`. 这代表这个资产是一个 Udon Script 的编译产物.

`serializedProgramCompressedBytes` 是一个十六进制字符串, 是 GZip 压缩后的 Udon Program 序列化结果.

II.2 Udon Program 的反序列化

serializedProgramCompressedBytes 经过 GZip 解压后得到的二进制文件是 `UdonProgram` 实例序列化后的结果.

这个序列化过程使用的是一个 VRChat 修改的 `OdinSerializer`. 所以我们可以直接用这个序列化器对应的反序列化器进行反序列化. 一些关键代码如下

cs

```cs
using System.IO;
using VRC.Udon.Common;
using VRC.Udon.Serialization.OdinSerializer;

using var memoryStream = new MemoryStream(fileData);
var context = new DeserializationContext();
var reader = new BinaryDataReader(memoryStream, context);
UdonProgram program =
  VRC.Udon.Serialization.OdinSerializer.SerializationUtility
    .DeserializeValue<UdonProgram>(reader);
```

II.3 `UdonProgram` 类

`UdonProgram` 类中几乎有我们需要的一切. 下面是一个简化[1]的类定义

cs

```cs
public class UdonProgram : IUdonProgram
{
  public string InstructionSetIdentifier { get; }
  public int InstructionSetVersion { get; }
  public byte[] ByteCode { get; }
  public IUdonHeap Heap { get; }
  public IUdonSymbolTable EntryPoints { get; }
  public IUdonSymbolTable SymbolTable { get; }
  public IUdonSyncMetadataTable SyncMetadataTable { get; }
  public int UpdateOrder { get; }
}
```

我们比较关心 `ByteCode`, `Heap`, `EntryPoints`, `SymbolTable` 这几个字段.

II.3.1 Udon 字节码和指令集

是一系列大端序 `u32` 组成的指令的序列.

[1]本小节中出现的类定义只列出了进入序列化后的 Udon Program 二进制的部分.

指令格式为 `OPCODE[OPERAND]`, 两部分各 4 字节, `OPERAND` 是一个大端序 `u32`.

`OPCODE` 包括无参数的 `NOP`, `POP`, `COPY` 和有一个参数的 `PUSH`, `JUMP_IF_FALSE`, `JUMP`, `EXTERN`, `ANNOTATION`, `JUMP_INDIRECT`.

各 `OPCODE` 对应的值为

python

```python
class OpCode(IntEnum):
    NOP = 0
    PUSH = 1
    POP = 2
    JUMP_IF_FALSE = 4
    JUMP = 5
    EXTERN = 6
    ANNOTATION = 7
    JUMP_INDIRECT = 8
    COPY = 9
```

各 OPCODE 和 OPERAND 含义如下:

- `NOP`: 空指令
- `PUSH I`: 将立即数 `I` 压栈
- `POP`: 从栈中弹出一个值并丢弃
- `COPY`: 复制堆中的值
- `JUMP_IF_FALSE ADDR`: 条件跳转到 `ADDR`
- `JUMP ADDR`: 无条件跳转到 `ADDR`
- `EXTERN F`: 调用外部函数, `F` 是堆中的函数签名 `string` 或者函数委托 `UdonExternDelegate` 的地址
- `ANNOTATION`: 注解, 执行时跳过
- `JUMP_INDIRECT IADDR`: 间接跳转到 `IADDR` 作为堆地址指向的值

II.3.2 堆

用于存储 Udon VM 执行该 Udon Program 时堆的初始值, 相当于常量段.

简化的类定义如下

cs

```cs
[Serializable]
public sealed class UdonHeap : IUdonHeap, ISerializable
{
  [NonSerialized]
  private readonly IStrongBox[] _heap;
  [NonSerialized]
  private readonly Dictionary<Type, Type>
    _strongBoxOfTypeCache = new Dictionary<Type, Type>();
  [NonSerialized]
  private readonly Dictionary<Type, Type>
    _strongBoxOfTContainedTypeCache = new Dictionary<Type, Type>();

  public void GetObjectData(
    SerializationInfo info, StreamingContext context
  )
  {
    List<ValueTuple<uint, IStrongBox, Type>> list =
```

```
18        new List<ValueTuple<uint, IStrongBox, Type>>();
19      this.DumpHeapObjects(list);
20      info.AddValue("HeapCapacity", Math.Max(0, this._heap.Length));
21      info.AddValue("HeapDump", list);
22    }
23
24    public void DumpHeapObjects(
25      List<ValueTuple<uint, IStrongBox, Type>> destination
26    )
27    {
28      uint num = 0;
29      while (num < this._heap.Length)
30      {
31        IStrongBox strongBox = this._heap[num];
32        if (strongBox != null)
33        {
34          destination.Add(new ValueTuple<uint, IStrongBox, Type>(
35            num,
36            strongBox,
37            strongBox.GetType().GenericTypeArguments[0]
38          ));
39        }
40        num += 1;
41      }
42    }
43  }
```

我们感兴趣的就是其中的 `HeapDump`, 这是一个 `(Addr, Value, Type)` 三元组的列表.

II.3.3 入口点表

实际上是函数表.

简化的类定义如下

cs

```
1  [Serializable]
2  public sealed class UdonSymbolTable : IUdonSymbolTable, ISerializable
3  {
4    private readonly ImmutableArray<string> _exportedSymbols;
5    private readonly ImmutableDictionary<string, IUdonSymbol> _nameToSymbol;
6
7    void ISerializable.GetObjectData(
8      SerializationInfo info, StreamingContext context
9    )
10   {
11     info.AddValue(
12       "Symbols",
13       this._nameToSymbol.Values.ToList<IUdonSymbol>()
14     );
15     info.AddValue(
16       "ExportedSymbols",
17       this._exportedSymbols.ToList<string>()
18     );
19   }
```

```
20 }
21
22 [Serializable]
23 public sealed class UdonSymbol : IUdonSymbol, ISerializable
24 {
25   public string Name { get; }
26   public Type Type { get; }
27   public uint Address { get; }
28
29   void ISerializable.GetObjectData(
30     SerializationInfo info, StreamingContext context
31   )
32   {
33     info.AddValue("Name", this.Name);
34     info.AddValue("Type", this.Type);
35     info.AddValue("Address", this.Address);
36   }
37 }
```

这里每个 `UdonSymbol` 里的

- `Name` 是函数名
- `Address` 是该函数的首条指令在 `UdonProgram.ByteCode` 中的索引
- `Type` 无意义

这给我们带来了很多方便.

II.3.4 符号表

类定义和入口点表相同, 其中每个 `UdonSymbol` 里的

- `Name` 是符号名
- `Address` 是该符号在堆中的地址
- `Type` 是符号类型

III Udon VM

是一个简单的栈式虚拟机.

III.1 堆, 栈和寄存器

- 堆: 是一个 `IStrongBox[]`, 地址就是数组索引, 使用程序中的常量段初始化
- 栈: 一个 `u32` 栈
- PC: 单位是字节

III.2 外部函数

Udon VM 的外部函数委托是 `UdonExternDelegate`, 具体定义为

cs
```cs
delegate void UdonExternDelegate(IUdonHeap heap, Span<uint>
parameterAddresses);
```

也即传入

- 堆用于获取参数和写入结果
- 一系列参数地址(在堆中的)用于获取参数

在此基础上封装了 `CachedUdonExternDelegate`, 具体定义为

cs
```cs
class CachedUdonExternDelegate
{
  public readonly string externSignature;
  public readonly UdonExternDelegate externDelegate;
  public readonly int parameterCount;
}
```

`CachedUdonExternDelegate` 可以完全通过一个 `string` 获取, 也即 `externSignature`.

这个 `externSignature` 其实就是简单的函数签名, 如

cs
```cs
ExternVRCEconomyIProduct.__Equals__VRCEconomyIProduct__SystemBoolean
ExternVRCEconomyIProduct.__get_Buyer__VRCSDKBaseVRCPlayerApi
ExternVRCEconomyIProduct.__get_Description__SystemString
ExternVRCEconomyIProduct.__get_ID__SystemString
ExternVRCEconomyIProduct.__get_Name__SystemString
```

这个签名由两部分组成, 分别是 `ModuleName` 和 `FuncSignature`. 类(也即 `Module`)通过实现 `IUdonWrapperModule`, 将自己的 `ModuleName` 和所有 `FuncSignature` 及其对应的参数数量注册到 `UdonWrapper` 中, 供其使用完整的 `externSignature` 获取.

III.3 执行过程

读取当前 PC 处的指令

- `NOP`: PC 步进 4 字节
- `PUSH`: 把 `OPERAND` 作为立即数压栈, PC 步进 8 字节
- `POP`: 弹栈, 丢弃栈顶值, PC 步进 4 字节
- `JUMP_IF_FALSE`: 栈顶是堆地址, 弹栈, 读该地址对应的堆元素(`bool`)的值
  - 若为 `true`, PC 步进 8 字节
  - 若为 `false`, 设置 PC 为 `OPERAND`
- `JUMP`: 设置 PC 为 `OPERAND`

- `EXTERN`: 调用外部函数. 尝试读取 `OPERAND` 作为堆地址指向的对象
  - 若为 `string`, 通过 UdonWrapper 获取该 `string` 对应的 `CachedUdonExternDelegate`
  - 若为 `CachedUdonExternDelegate`, 也得到了 `CachedUdonExternDelegate`

  从栈中连续弹出 `CachedUdonExternDelegate.parameterCount` 个参数地址, 按与弹栈相反的顺序(也即最初的栈顶为最后一个地址)组装成 `Span<uint> parameterAddresses`, 并调用 `UdonExternDelegate`. PC 步进 8 字节
- `ANNOTATION`: PC 步进 8 字节
- `JUMP_INDIRECT`: 设置 PC 为 `OPERAND` 作为堆地址指向的 `u32` 值
- `COPY`: 从栈中先后弹出 `TARGET` 和 `SOURCE` 两个地址, 然后把堆中 `TARGET` 地址指向的值使用 `SOURCE` 地址指向的值覆盖. 所在 PC 步进 4 字节

IV 反编译

在网络上找到了开源的反汇编器和反编译器. 但是这两个实现都过于初级, 能做的明显还有很多. 如果没有意外, 我会在最近(或许不是)实现一个更好的反编译器.