消息数据格式

-虚幻引擎交互设计师班-

火星时代教育

CONTENT

概念分析

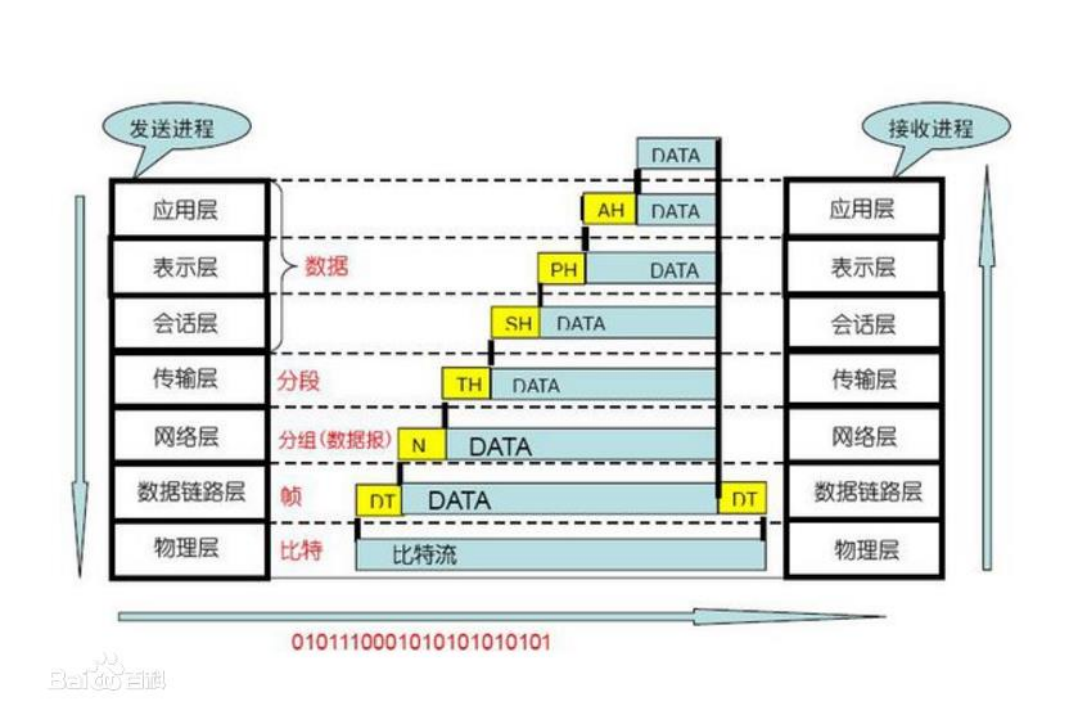
-虚幻引擎交互设计师班-

火星时代教育

数据格式

从参考右图我们可以得知，网络中传输的数据就是数据字节信息。借助物理硬件完成信息传输与接收。在这个过程中，必然会涉及原始数据与传输数据之间的转换，一般如文本，图片，声音，视频均需要完成转换，才能传输。

我们在网络中，需要协定传输数据的格式，数据格式是我们能完成数据交换的根本。也是标记数据读取方式的根本。在计算机中，数据格式也是用来描述数据解析方式的依据。



文本编码

-虚幻引擎交互设计师班-

火星时代教育

为什么要编码?

首先我们需要明白，计算机只认识字节信息，只能存储字节信息，而人类我发直观的阅读字节信息。我们更多使用的是基于图形设计的标识信息。那么现在问题就很明显了，必须有一方妥协，才能够很好的交流，我们无法改变计算机的数据理解方式，人类也不好去改变习惯，但是我们可以设计一种转换规则，将有限的符号（文本）与字节种类一一对应。这种对应方式其实就是编码。

编码是信息从一种形式或格式转换为另一种形式的过程，也称为计算机编程语言的代码简称编码。用预先规定的方法将文字、数字或其它对象编成数码，或将信息、数据转换成规定的电脉冲信号。编码在电子计算机、电视、遥控和通讯等方面广泛使用。编码是信息从一种形式或格式转换为另一种形式的过程。解码，是编码的逆过程。

文本编码种类

计算机最早对于文本的编码只有ASCII编码，他支持128个字符，对于英语来说，是完全够使用了，但是对于亚洲国家来说128个字符是远远不够的！因此，为了解决更多编码问题，人们设计了更多的编码集，例如比较常见的编码集有GBK，ISO，Unicode，UTF-8等。

虚幻引擎支持UTF-8编码

为什么使用UTF-8

在软件设计中，设计者需要更多考虑时间和空间问题，例如有的编码码值较小，转换后可以占用2个字节，或是1个字节，但是编码集却规定一个字符占用4个字节，这其实并不合理！而UTF-8的最大优势在于他是一种可变长度的字符编码。

优点

UTF-8编码可以通过屏蔽位和移位操作快速读写。字符串比较时strcmp()和wcscmp()的返回结果相同，因此使排序变得更加容易。字节FF和FE在UTF-8编码中永远不会出现，因此他们可以用来区分UTF-16或UTF-32文本。

UTF-8是字节顺序无关的。它的字节顺序在所有系统中都是一样的，因此它实际上并不需要BOM。

缺点

你无法从UNICODE字符数判断出UTF-8文本的字节数，因为UTF-8采用的是不定长的编码方式。

虚幻中的文本编码转换

转换宏

虚幻中提供了文本串编码转换宏，如果你的文本串是阿斯克编码可以通过宏转换为UTF-8，或是互相转换，可以参照下图。

```
// Usage of these should be replaced with StringCasts.
#define TCHAR_TO_ANSI(str) (ANSICHAR*)StringCast<ANSICHAR>(static_cast<const TCHAR*>(str)).Get()
#define ANSI_TO_TCHAR(str) (TCHAR*)StringCast<TCHAR>(static_cast<const ANSICHAR*>(str)).Get()
#define TCHAR_TO_UTF8(str) (ANSICHAR*)FTCHARToUTF8((const TCHAR*)str).Get()
#define UTF8_TO_TCHAR(str) (TCHAR*)FUTF8ToTCHAR((const ANSICHAR*)str).Get()
```

应用

虚幻中提供了文本串编码转换宏，如果你的文本串是阿斯克编码可以通过宏转换为UTF-8，或是互相转换，可以参照下图。

```
// Usage of these should be replaced with StringCasts.
#define TCHAR_TO_ANSI(str) (ANSICHAR*)StringCast<ANSICHAR>(static_cast<const TCHAR*>(str)).Get()
#define ANSI_TO_TCHAR(str) (TCHAR*)StringCast<TCHAR>(static_cast<const ANSICHAR*>(str)).Get()
#define TCHAR_TO_UTF8(str) (ANSICHAR*)FTCHARToUTF8((const TCHAR*)str).Get()
#define UTF8_TO_TCHAR(str) (TCHAR*)FUTF8ToTCHAR((const ANSICHAR*)str).Get()
```

JSON&XML

-虚幻引擎交互设计师班-

火星时代教育

JSON

JSON

JSON(JavaScript Object Notation, JS 对象简谱) 是一种轻量级的**数据交换格式**。它基于 ECMAScript (欧洲计算机协会制定的js规范)的一个子集，采用完全独立于编程语言的文本格式来存储和表示数据。简洁和清晰的层次结构使得 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，并有效地提升网络传输效率。

语法规则

JSON是一个序列化的**对象**或**数组**，这表明JSON结构的外围要么是对象结构要么是数组结构。

语法符号：“ {}”“ []”“ ,”“ :”，一般{}圈定为对象结构，[]圈定为数组结构。

构成

JSON语法中，数据是按照KV模式描述的。K是字符串，V是值，值可以是对象，数组，数字，字符串，或是三个字面值(false、null、true)中的一个。例如{“name”：“song”}，其中name为K，song为字符串值。

例子：

对象型JSON

{"name":"song","age":15,"student":true,"friends":["1","2","3"]}

数组型JSON，**数组中只能装填值类型数据，不可以装填KV格式数据**。

["OK",50,true,{"name":"hong"}]

样例

```
{
    "person": [
        {
            "teachers": [
                {
                    "zhang": {
                        "name": "zhangq",
                        "hobby": [
                            "running",
                            "reading"
                        ]
                    }
                },
                {
                    "liu": {
                        "name": "lium",
                        "hobby": [
                            "codeing",
                            "sleep"
                        ]
                    }
                }
            ]
        },
        {
            "students": [
                {
                    "mao": {
                        "name": "maon",
                        "hobby": [
                            "playgame",
                            "reading"
                        ]
                    }
                },
                {
                    "wang": {
                        "name": "wangx",
                        "hobby": [
                            "walking",
                            "sleep"
                        ]
                    }
                }
            ]
        }
    ]
}
```

虚幻引擎中的JSON

前言

如果希望在虚幻引擎中使用JSON必须借助C++完成，并且需要在项目的模块构建文件中引入**JSON模块**。

```
public class UEAnt : ModuleRules
{
    Zery Zhang *
    public UEAnt(ReadOnlyTargetRules Target) : base(Target)
    {
        PCHUsage = PCHUsageMode.UseExplicitOrSharedPCHs;

        PublicDependencyModuleNames.AddRange( collection: new string[] { "Core", "CoreUObject", "Engine", "InputCore", "NsAnt", "Json" });

        PrivateDependencyModuleNames.AddRange( collection: new string[] {  });

        // Uncomment if you are using Slate UI
        // PrivateDependencyModuleNames.AddRange(new string[] { "Slate", "SlateCore" });

        // Uncomment if you are using online features
        // PrivateDependencyModuleNames.Add("OnlineSubsystem");

        // To include OnlineSubsystemSteam, add it to the plugins section in your uproject file with the Enabled attribute set to true
    }
}
```

解析JSON数据

解析JSON数据，首先需要了解JSON层级结构，能够区分值类型。一般JSON解析分两种情况

- 根是对象型数据
- 根是数组型数据

解析对象型JSON

解析时需要注意层级结构，以及读取API函数的应用。切记操作过程中使用共享指针完成。

```
//读取对象型JSON数据
FString JsonStr = TEXT("{\"name\":\"xiaoming\"}");
//创建JsonReader
TSharedRef<TJsonReader<TCHAR>> JsonReader = TJsonReaderFactory<TCHAR>::Create(JsonStr);
//借助解析器解析到JsonObject(因为跟是Object结构)
TSharedPtr<FJsonObject> RootObj;
//反向序列化json文本结构到内存中，以对象结构操作
if (FJsonSerializer::Deserialize(JsonReader, [&]RootObj))
{
    //Value值为 xiaoming
    FString Value = RootObj->GetStringField(TEXT("name"));
}
```

读取数组型JSON

数组型Json需要注意数组中的元素会被转换为FJsonValue类型。通过As函数可以转到到对应类型。

```
//读取对象型JSON数据
FString JsonStr = TEXT("[\"str\", 60, {\"name\":\"xiaoming\"}]");
//创建JsonReader
TSharedRef<TJsonReader<TCHAR>> JsonReader = TJsonReaderFactory<TCHAR>::Create(JsonStr);
//借助解析器解析到JsonObject(因为跟是Object结构)
TArray<TSharedPtr<FJsonValue>> JsonArray;
//反向序列化json文本结构到内存中，以对象结构操作
if (FJsonSerializer::Deserialize(JsonReader, [&]JsonArray))
{
    //读取文本串 直接使用FJsonValue操作函数
    JsonArray[0]->AsString();//返回 str
    JsonArray[1]->AsNumber();
    TSharedPtr<FJsonObject> JsonObj = JsonArray[2]->AsObject();//返回 TSharePtr<FJsonObject>
    JsonObj->GetStringField(TEXT("name"));//返回 xiaoming
}
```

序列化JSON

序列化操作与解析操作一样，也分为两种情况即

- 根是数组型数据
- 根是对象型数据

序列化对象型JSON

```cpp
//序列化对象型Json
TSharedPtr<FJsonObject> JsonRoot = MakeShareable(new FJsonObject);
//设置数据内容
JsonRoot->SetStringField(TEXT("name"), StringValue:TEXT("xiaoming"));
JsonRoot->SetNumberField(TEXT("age"), Number:20);

//创建用于接收转换Json后的字符串对象
FString JsonStr;
//构建JsonWriter
TSharedRef<TJsonWriter<TCHAR>> JsonWriter = TJsonWriterFactory<TCHAR>::Create(&JsonStr);
//序列化操作 注意传入FJsonObject共享引用
if (FJsonSerializer::Serialize(Object:JsonRoot.ToSharedRef(), JsonWriter))
{
    //转换成功后 JsonStr值为 {"name":"xiaoming", "age":20}
}
```

序列化数组型JSON

```cpp
//序列化数组型Json
//注意数组中装填的元素是JsonValue
TArray<TSharedPtr<FJsonValue>> JsonArray;
//向数组中添加元素 添加整数型值
JsonArray.Add( Item: MakeShareable(new FJsonValueNumber(50)));
JsonArray.Add( Item: MakeShareable(new FJsonValueString(TEXT("Hello"))));
//向数组中添加对象型json结构
TSharedPtr<FJsonObject> JsonObj = MakeShareable(new FJsonObject);
JsonObj->SetStringField(TEXT("name"), StringValue: TEXT("xiaoming"));
JsonObj->SetNumberField(TEXT("age"), Number: 20);
JsonArray.Add( Item: MakeShareable(new FJsonValueObject(JsonObj)));
//构建JsonWriter
FString JsonStr;
TSharedRef<TJsonWriter<TCHAR>> JsonWriter = TJsonWriterFactory<TCHAR>::Create(&JsonStr);
//序列化Json
if (FJsonSerializer::Serialize(JsonArray, JsonWriter))
{
    //JsonStr值为 "[50,"Hello",{"name":"xiaoming", "age":20}]"
}
```

XML

XML

可扩展标记语言，标准通用标记语言的子集，简称XML。是一种用于标记电子文件使其具有结构性的标记语言。XML是纯文本文件，最大的特点就是具有**自我描述性**。它可以简单的用在任何应用程序中读/写数据。是一种常用的**数据交换公共语言**。XML可以可以更容易的与Windows、Mac OS、Linux以及其他平台下产生的信息结合，然后可以很容易加载XML数据到程序中并分析它，并以XML格式输出结果。

样例

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
    <to>George</to>
    <from>John</from>
    <heading>Reminder</heading>
    <body>Don't forget the meeting!</body>
</note>
```

格式规则

1、必须有声明语句

XML声明是XML文档的第一句，如：<?xml version="1.0" encoding="utf-8"?>

2、区分大小写

3、XML文档只有一个根元素

4、属性值使用引号

5、所有的标记必须有相应的结束标记

6、所有标记均需要被关闭

样例

```
<?xml version="1.0" encoding="utf-8"?>
<persons>
  <AX graduate="2010" skill="A" working="5">
    <address>A01</address>
    <wife age="35">X</wife>
    <son age="8">K</son>
  </AX>
  <AT graduate="2005" skill="B" working="9">
    <address>A20</address>
    <wife age="29">T</wife>
    <mom age="68">H</mom>
  </AT>
</persons>
```

虚幻引擎中的XML

虚幻引擎中的XML

在虚幻引擎中，如果需要解析XML，需要借助模块XmlParser。

```
public class LegoGame : ModuleRules
{
    Zery Zhang *
    public LegoGame(ReadOnlyTargetRules Target) : base(Target)
    {
        PCHUsage = PCHUsageMode.UseExplicitOrSharedPCHs;

        PublicDependencyModuleNames.AddRange(collection:new string[] { "Core", "CoreUObject", "Engine", "InputCore", "Slate",
            "SlateCore", "UMG", "AIModule", "GamePlayTasks", "XmlParser"
        , "NavigationSystem"});

        PrivateDependencyModuleNames.AddRange(collection:new string[] {  });

        // Uncomment if you are using Slate UI
        // PrivateDependencyModuleNames.AddRange(new string[] { "Slate", "SlateCore" });

        // Uncomment if you are using online features
        // PrivateDependencyModuleNames.Add("OnlineSubsystem");

        // To include OnlineSubsystemSteam, add it to the plugins section in your uproject file with the Enabled attribute set to true
    }
}
```

解析XML

解析标签需要注意使用FXmlNode类，以下部分代码

```
//解析XML
TSharedPtr<FXmlFile> XmlFile = MakeShareable(new FXmlFile);
//加载磁盘中的XML文件
if (XmlFile->LoadFile(Path:TEXT("d:\\a.xml")))
{
    //获取XML根
    FXmlNode* RootNode = XmlFile->GetRootNode();
    if (RootNode)
    {
        RootNode->GetTag();//获取标签名称
        RootNode->GetAttribute(InTag:TEXT("Name"));//获取标签属性Name值
        TArray<FXmlNode*> Nodes = RootNode->GetChildrenNodes();//获取子标签内容
    }
}
```

序列化XML

虚幻引擎对于序列化标签的API支持比较简单，在操作过程中，**无法直接向标签中添加标签描述**，故序列化功能过于单播。

```
//标签头文本
FString Xml = TEXT("<Person></Person>");
//创建标签文件数据，注意创建方法为从数据流中创建
TSharedPtr<FXmlFile> XmlFile = MakeShareable(new FXmlFile(Xml, EConstructMethod::ConstructFromBuffer));
//追加标签内容
XmlFile->GetRootNode()->AppendChildNode(InTag:TEXT("Age"), InContent:TEXT("50"));
//修改标签Age内容，由于Age是在标签首，故可以直接通过0访问
//XmlFile->GetRootNode()->GetFirstChildNode();//获取首标签（只读）Age
//XmlFile->GetRootNode()->GetChildrenNodes()[0];//获取首标签（读写）Age
//修改Age节点中的50到60
FXmlNode* Node = XmlFile->GetRootNode()->GetChildrenNodes()[0];
Node->SetContent(TEXT("60"));
//将标签存储到本地
XmlFile->Save(Path:TEXT("E:\\a.xml"));
```

Protocol Buffers

-虚幻引擎交互设计师班-

火星时代教育

Protocol Buffers

Protocol Buffers，是Google公司开发的一种数据描述语言，类似于XML能够将结构化数据序列化，可用于数据存储、通信协议等方面。它是一种**独立的数据交换格式**，可用于多种领域。**它独立于语言，独立于平台**。谷歌提供了多种编程语言下的序列化和反序列化方案。由于它是一种二进制格式，比使用XML进行数据交换包体更小（小3-10倍），解析速度更快（快20-100倍）。对于PB来说，它的使用大致分为三步：

1. 定制消息体message（运行前）
2. 编译消息体message，生成对应的平台代码（运行前）
3. 应用代码读写结构化数据（运行中）

定义消息体

基本格式

首先需要注意Protocol Buffers需要先定义消息体。它按照给定的格式完成制定，大致如下：

message 名称{

}

消息体最后需要被保存为.proto的文件，用于生成二进制消息文件。

我们目前针对的是Proto3进行学习，所有内容针对Proto3说明。

样例格式

```
Syntax = “proto3” //协议版本
message Person{//消息名称
    int32 age = 1;//字段规则（暂无） 字段类型，字段名称，字段序号
    string name = 2;
    repeated string friends = 3;//repeated重复子段，类似数组
}
```

注意事项

- 协议版本不能为空
- 协议名称，非数字开头，敏感大小写
- 字段规则中，单一数据（singular）无需指定关键字，数组型需要指定标记（repeated）
- 字段类型（后表）
- 字段名称，非数字开头即可（禁止中文）敏感大小写
- 字段序号，每个字段的序号必须是唯一的并且为正整数，最小为1最大为(2^29)-1，且禁止是19000~19999的任何一个值

字段类型

.proto Type	Notes	C++ Type	Java/Kotlin Type[1]	Python Type[3]	Go Type	Ruby Type	C# Type	PHP Type
double		double	double	float	float64	Float	double	float
float		float	float	float	float32	Float	float	float
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32	int	int	int32	Fixnum or Bignum (as required)	int	integer
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	int64	long	int/long[4]	int64	Bignum	long	integer/string
uint32	Uses variable-length encoding.	uint32	int[2]	int/long[4]	uint32	Fixnum or Bignum (as required)	uint	integer
uint64	Uses variable-length encoding.	uint64	long[2]	int/long[4]	uint64	Bignum	ulong	integer/string
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int	int	int32	Fixnum or Bignum (as required)	int	integer
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	int64	long	int/long[4]	int64	Bignum	long	integer/string
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^{28}.	uint32	int[2]	int/long[4]	uint32	Fixnum or Bignum (as required)	uint	integer
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^{56}.	uint64	long[2]	int/long[4]	uint64	Bignum	ulong	integer/string
sfixed32	Always four bytes.	int32	int	int	int32	Fixnum or Bignum (as required)	int	integer
sfixed64	Always eight bytes.	int64	long	int/long[4]	int64	Bignum	long	integer/string
bool		bool	boolean	bool	bool	TrueClass/FalseClass	bool	boolean
string	A string must always contain UTF-8 encoded or 7-bit ASCII text, and cannot be longer than 2^{32}.	string	String	str/unicode[5]	string	String (UTF-8)	string	string
bytes	May contain any arbitrary sequence of bytes no longer than 2^{32}.	string	ByteString	str (Python 2) bytes (Python 3)	[]byte	String (ASCII-8BIT)	ByteString	string

默认值

- string类型，默认值为 ""
- bytes类型，默认值为 空字节
- bool类型，默认值为false
- numeric类型，默认值为0
- enum类型，默认值为第一个值，且必须为0
- message类型，默认值未设置，值由语言特性确定

枚举类型

```
syntax = "proto3";
message Person {
    enum Gender {
        NONE = 0;
        MALE = 1;
        FEMALE = 2;
    }
    Gender gender = 1;
}
```

内部引用

```
syntax = "proto3";
message Person {
    enum Gender {
        NONE = 0;
        MALE = 1;
        FEMALE = 2;
    }
    Gender gender = 1;
}

message Team {
    repeated Person persons = 1;
}
```

外部引用

```
syntax = "proto3";
message Person {
    enum Gender {
        NONE = 0;
        MALE = 1;
        FEMALE = 2;
    }
    Gender gender = 1;
}

//Person是在当前消息同文件夹下的proto文件
syntax = "proto3";
import "Person.proto";
message Team {
    repeated Person persons = 1;
}
```

使用import关键字可以引入已经编写好的proto文件（注意路径）

嵌套

```
syntax = "proto3";
message Person {
    enum Gender {
        NONE = 0;
        MALE = 1;
        FEMALE = 2;
    }
    Gender gender = 1;
}

//Person是在当前消息同文件夹下的proto文件
syntax = "proto3";
import "Person.proto";
message Team {
    repeated Person persons = 1;
}
```

使用import关键字可以引入已经编写好的proto文件（注意路径）

更新规则

由于业务需求，对于已经编写好的消息可能会涉及修改调整，那么就需要更新消息。更新消息需要遵守以下规则。

- 不要修改任何已经存在的项的ID
- 如果增加了新的项(使用了新的ID)，原有的旧的Message生成的代码进行序列化的信息，依然能被你新的Message生成的代码解析
- 项可以删除，可以修改项名称的方式，或者直接把对应的tagId标记为删除
- int32，uint32，int64，uint64，bool 是彼此兼容的，也就是说这些类型的项可以任何变更到彼此中的任何一个
- sint32，sint64 是彼此兼容的
- string 和 bytes 是彼此兼容的
- fixed32 与 sfixed32，fixed64，sfixed64是兼容的

Thanks

-火星时代游戏设计学院-