课题: Unity中的Http网络通信

此文章主要介绍Unity引擎中的Http网络通信。文章将会配合一个Unity项目作为示例，通过Unity封装的WebRequest类对ASP .NET Core WebAPI后端请求最近的天气信息，并使用MVC设计模式搭建一套GUI来进行数据呈现。

涉及内容：

- Mvc设计模式
- Unity网络通信API
- 异步方法
- HTTP通信协议
- JSON格式数据的序列化/反序列化
- 后端服务搭建

Unity Http通信的主要用途

- 与服务器进行数据交互：
  Unity中的HTTP通信可以用于与远程服务器进行数据交互。这可以包括从服务器获取游戏数据、发送玩家分数、保存游戏进度等。通过HTTP通信，可以实现与Web服务或自定义后端服务器的数据交换。
- 下载和上传资源：
  游戏可能需要从服务器下载资源，如纹理、模型、音频文件等。或是接受客户端热更新的补丁。
- 社交媒体集成：
  游戏中集成社交媒体功能时，可能需要使用HTTP通信与社交媒体平台进行交互。这可以包括获取玩家好友列表、分享成就或游戏状态等。
- 登录和身份验证：
  游戏通常需要与服务器进行身份验证，确保玩家是合法用户。
- 获取实时数据：
  某些项目可能需要获取实时数据，如天气信息、股票价格等。通过HTTP通信，可以从相应的API获取这些数据。

基于MVC架构的数据可视化

项目基于MVC架构构建用户界面，定义数据模型，分离界面逻辑与数据处理逻辑。

Mvc架构:

- **Model**: 数据模型的定义

> 数据类型的定义，用于界面层以及控制器层的数据传输，以及客户端与服务端数据的对接。

- **View**: 界面样式，逻辑以及交互的处理

> 用于界面的数据呈现，样式控制，按钮、输入框等交互事件的绑定。

- **Controller**: 数据的请求和处理

> 对服务端的网络请求，数据的获取与处理。

> 文档：Unity中的MVC设计模式。

前后端分离架构与网络请求

客户端通常不在程序中直接访问数据库进行数据存取，流量较大，或是重要的数据一般通过请求服务端来获取。

前后端分离即界面与数据分离，这样的架构有利于项目的分工和解耦，并减少客户端的性能消耗。

HTTP请求方法(Method)

前后端的数据对接通常使用HTTP协议进行传输，并在HTTP报文体中使用JSON或XML的形式存储数据。
客户端通过不同的HTTP请求方法来向后端指示请求的操作，常用的HTTP方法有：

- `GET` 仅向服务器请求检索数据
- `POST` 将数据提交到服务器
- `PUT` 将数据提交到服务器，请求修改并替换指定资源
- `DELETE` 请求服务器删除指定的资源
- ...

> 以上请求方法的描述符合RESTful Api规范，具体的请求操作取决于服务端

Unity WebRequest

`UnityEngine.Networking` 命名空间提供了一些API来进行网络请求。
Unity常用 `UnityWebRequest` 类进行HTTP请求，通过类中的静态方法即可快速对服务端进行请求。

```
// 访问以下方法来创建请求对象
var getReq = UnityWebRequest.Get(string url); // GET
var postReq = UnityWebRequest.Post(string url); // POST
var putReq = UnityWebRequest.Put(string url); // PUT
var deleteReq = UnityWebRequest.Delete(string url); // DELETE

// 调用SendWebRequest方法发送网络请求
// 基于Unity协同程序的异步方法
getReq.SendWebRequest();

// 异步任务需要通过回调函数来获取结果
// 通过监听UnityWebRequest.completed事件来添加回调函数
getReq.completed += OnCompleted;
void OnCompleted(AsyncOperation op)
{
    var webReqOp = op as UnityWebRequestAsyncOperation; // 将AsyncOperation断言为UnityWebR
    var webReq = op.webRequest; // 访问网络请求对象
    if (webReq.isNetworkError || webReq.isHttpError) // 检查是否出现错误
        throw new ApplicationException("请求出现错误");
    // NetworkError:网络相关错误
    // HttpError:Http通信错误，可能是请求的连接或内容有误，或是服务器出现错误(Http 4xx,5xx)
    // PS: 以上两个属性在Unity2020被废弃，请使用 webReq.result枚举来判定错误

    var text = webReq.downloadHandler.text; // 访问下载处理器来获取响应的文字或字节流
    var data = webReq.downloadHandler.data; // 访问下载处理器来获取响应的二进制数据

    print(text);
}
```

> - 详见UntiyWebRequest官方文档

关于同步和异步方法

- 同步：
  即刻实行，函数结束则任务结束，目标数据可以立刻获取到，但是会阻塞主线程

```
var text = File.ReadAllText("a.txt");
Console.WriteLine(text);
```

- 异步： 延迟执行，函数执行仅代表异步任务的创建，结束的时机不可知，目标数据无法立刻获取到，需要借助回调函数来间接获取

```
var task = File.ReadAllTextAsync("a.txt");
task.ContinueWith(t => Console.WriteLine(t.Result)); // 绑定回调函数
```

> - 带有"*Async(Asynchronous)*"字样的函数或类一般都与异步相关
> - .NET SDK中所有异步函数均以"*Async*"结尾，并且返回 `Task` 类型
> - 低版本的Untiy或是.NET框架可能缺少某些异步函数

异步任务的特点：
异步任务通常使使用多线程或是协同程序实现，不影响主线程/UI线程工作。
异步任务创建后，需要手动管理任务的生命周期，如开始，中止，完成。
异步任务执行成功后，其获取到的数据需要以回调的形式间接地进行使用。

关于JSON序列化方法

常见的Web API大多数的请求/响应格式都是Json格式的字符串(部分可能为XML)，因此Unity/前端每次请求时都需要对数据模型进行序列化或是对Json字符串进行反序列化。

Unity中有多种方案进行Json(反)序列化：

- 使用 `JsonUtility` (Unity自带，性能好，但支持类型较少)

```
using UnityEngine;

Model obj;
string json = JsonUtility.ToJson(obj); // 序列化
Model model = JsonUtility.FromJson<Model>(json); // 反序列化
```

- 使用 `System.Text.Json` (C#框架自带，早期.NET版本性能和兼容性不佳)

```
using System.Text.Json;

Model obj;
string json = JsonSerializer.Serialize(obj);
Model model = JsonSerializer.Deserialize<Model>(json);
```

- 使用 `Newtonsoft.Json` (第三方Nuget包，性能一般，拥有完善的类型适配以及强大的反序列化功能)

```
using Newtonsoft.Json;

Model obj;
string json = JsonConvert.SerializeObject(obj);
Model model = JsonConvert.DeserializeObject<Model>(json);
```

> - `System.Text.Json` 早期版本性能和兼容性一般，而在.NET Core 3.1版本之后逐渐好转
> - `Newtonsoft.Json` 是C#包管理器下载量排行第一的包，从这里可以下载
> - `Newtonsoft.Json` 在新版本Unity中可以通过包管理器引入，2018版本似乎不支持，但可以通过载入

关于后端服务的搭建

后端服务可使用各类Web框架来搭建，如 `Flask` ， `Node.js ( Express.js` ， `Sails.js` 等)， `ASP .NET Core` ， `Spring` 等各类框架。

使用 `Express.js` 搭建简易API的案例：

```
const express = require('express') // 引入express.js包
const app = express() //创建web应用实例
const port = 7890 //端口号为7890

// 映射'/Hello'路径，请求时返回"Hello World!"字符串
app.get('/Hello', (req, res) => {
  res.send('Hello World!')
})

// 监听配置的端口(7890)并输出日志
app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

使用 `Node.js` 环境安装 `Express.js` 并运行以上代码，即可在本机的 `http://localhost:7890` 路径上部署一个Web服务器，请求 `http://localhost:7890/Hello`路径即可得到 "Hello World!"` 文本。

> - 该示例项目的后端使用 `.NET 8` 搭建，具体代码请参考 `WeatherApiDemo` 文件夹。
> - 项目源码，文档以及发布文件已经上传至github仓库：
>   https://github.com/AzusaArchive/UnityWebRequestDemo