基于 Selenium 的网络爬虫基操

有个需求会用到爬虫, 于是使用 Selenium 库学习了一下大致的步骤, 基本实现了要爬取的功能后总结一下如何使用

安装包

首先在 nuget 中安装 `Selenium.WebDriver` 和 `Selenium.WebDriver.GeckoDriver`(火狐) , 咕咕噜用户选择 `Selenium.WebDriver.ChromeDriver`

开启浏览器

先根据自己需要设置浏览器启动参数

```
public static FirefoxOptions SetFirefoxOption()
{
    var options = new FirefoxOptions();
    options.AddArgument("--start-maximized");
    options.AddArgument("--ignore-certificate-errors");
    return options;
}
```

new 一个浏览器出来, IWebDriver 可以用来模拟用户在浏览器中的行为

```
using (IWebDriver driver = new FirefoxDriver(SetFirefoxOption()))
{
    try
    {
        // 浏览器要干的事
    }
    catch (Exception ex)
    {
        Console.WriteLine($"连接时发生错误: {ex.Message}");
    }
    finally
    {
        driver.Quit();
    }
}
```

导航到地址

使用 `driver.Navigate().GoToUrl()` 方法导航到目标地址

```
public static void OpenMainPage(IWebDriver driver, string url)
{
    // 打开主页
    driver.Navigate().GoToUrl(baseUrl);

    // 等待页面加载完成
    wait = new WebDriverWait(driver, TimeSpan.FromSeconds(10));
    wait.Until(d => d.FindElement(By.ClassName("content-box")));

    Console.WriteLine("页面加载完成");
}
```

查找控件

使用 `IWebDriver.FindElement()` 方法来查找页面中的元素, 常用的查找有 `ClassName`, `CssSelector`, `TagName` 等

我习惯的做法是

- 先在浏览器中选中目标, 这时候在 `DOM与样式检测器` 中会显示目标在 `DOM树` 中的层级
- 按照这个层级去 `主控台` 中使用 `querySelector()` 查找
- 不断缩减 `css selector` 中的内容达到最简
- 改为代码

例如

我想要获取一系列图片的地址, 于是先用 `css selector` 定位到了对应的 img

```
document.querySelector("html body.shop-style-02 div.main-box div.main div.main-top
div.major-function-box div.exhibition ul.sm-list li img")
```

缩减后

```
document.querySelectorAll("ul.sm-list li img")
```

然后使用代码查找, 并执行后续的相关逻辑

```
public static void GetProductImages(IWebDriver driver)
{
    try
    {
        var imgElements = driver.FindElements(By.CssSelector("ul.sm-list li img")).ToList();
```

```
        Console.WriteLine("找到的图片 URL:");

        var URLs = imgElements
            .Select(img => img.GetAttribute("src"))
            .Where(src => src != null && src.EndsWith("_s.jpg"))
            .Select(src => src.Replace("_s.jpg", "_n.jpg"))
            .Distinct()
            .ToList();

        URLs.ForEach(Console.WriteLine);

        imageURLs = URLs;
    }
    catch (Exception ex)
    {
        Console.WriteLine($"获取图片链接时发生错误: {ex.Message}");
    }
}
```

切换页面

有时候点击链接会打开一个新的标签页, 这时候 `IWebDriver` 依然会停留在原标签页, 直接使用会导致元素找不到, 需要手动切换

```
// 先获取当前窗口句柄
string currentWindowHandle = driver.CurrentWindowHandle;

// 获取所有打开的窗口句柄
var windowHandles = driver.WindowHandles;

// 切换到新标签页
windowHandles.ToList().ForEach(windowHandle =>
{
    if (windowHandle != currentWindowHandle) driver.SwitchTo().Window(windowHandle);
});
```

我始终保持最多两个页面所以上方代码可行, 如果存在较多页面请尝试将打开标签页前后的 `WindowHandles` 都存下来比较差异

如何在 Swagger 页面展示自定义内容

创建一个新的 ASP.NET Core WebAPI 项目时，会同步使用 Swagger 作为 API 文档，我们可以通过在管线中自定义添加 Swagger 中间件时的实现，来达到对 Swagger 内容或执行逻辑的一些定制化支持

Swagger 默认会添加在 Program.cs 中，自动生成的相关代码如下

```
builder.Services.AddSwaggerGen();

...

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
        app.UseSwaggerUI();
}
```

通过修改服务的添加来将默认的 Swagger 改为我们想要的样子，首先将原本的 AddSwaggerGen 封装在自己的方法中

```
public static void InitSwagger(this IServiceCollection services)
{
    services.AddSwaggerGen();
}
```

根据 AddSwaggerGen 方法的定义，我们需要添加 SwaggerGenOptions 到方法中

```
public static IServiceCollection AddSwaggerGen(
    this IServiceCollection services,
    Action<SwaggerGenOptions> setupAction = null)
```

接下来，以不同版本的 API 接口为例。创建一个枚举类来标示版本

```
public enum APIVersionEnum
{
    Version_0_0 = 1,
    Version_1_0 = 2,
}
```

在 SwaggerGenOptions 中，需要根据不同的枚举值做出不同的操作

```
services.AddSwaggerGen(options =>
{
        typeof(APIVersionEnum).GetEnumNames().ToList().ForEach(version =>
        {
                // 处理逻辑
        });
});
```

额外地，需要在项目生成属性中勾选输出 API 文档，并在程序中反射到该文档

```
public static void InitSwagger(this IServiceCollection services)
{
    services.AddSwaggerGen(options =>
    {
        typeof(APIVersionEnum).GetEnumNames().ToList().ForEach(version =>
        {
            options.SwaggerDoc(version, version switch
            {
                "Version_0_0" => new OpenApiInfo()
                {
                    Title = "Version_0_0",
                    Version = "v0.0",
                    Description = "description",
                    Contact = new OpenApiContact()
                    {
                        Name = "author",
                        Url = new Uri("http://gz4nna.github.io")
                    }
                },
                "Version_1_0" => new OpenApiInfo()
                {
                    Title = "Version_1_0",
                    Version = "v1.0",
                    Description = "description",
                    Contact = new OpenApiContact()
                    {
                        Name = "author",
                        Url = new Uri("http://gz4nna.github.io")
                    }
                },
                _ => new OpenApiInfo()
                {
                    Title = "title",
                    Version = "v0.0",
                    Description = "description",
```

```
                    Contact = new OpenApiContact()
                    {
                        Name = "author",
                        Url = new Uri("http://gz4nna.github.io")
                    }
                }
            });
        });

        var xmlFileName = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
        options.IncludeXmlComments(Path.Combine(AppContext.BaseDirectory,
xmlFileName), true);
    });
}
```

以上为对服务的修改，SwaggerDoc 定义如下

```
public static void SwaggerDoc(this SwaggerGenOptions swaggerGenOptions, string name,
OpenApiInfo info)
{
    swaggerGenOptions.SwaggerGeneratorOptions.SwaggerDocs.Add(name, info);
}
```

我们用 APIVersionEnum 中的枚举值将原本一整个 Swagger 文档分开，每部分展示的内容除了对 API 本身的手动划分外，还与指定的 OpenApiInfo 有关，常用的属性诸如 `Title`, `Description`, `Version`, `TermsOfService`, `Contact`, `License` 等等

接下来在路由中添加终结点

```
public static void InitSwagger(this WebApplication app)
{
    app.UseSwagger();
    app.UseSwaggerUI(options =>
    {
        typeof(APIVersionEnum).GetEnumNames().ToList().ForEach(version =>
        {
            options.SwaggerEndpoint($"/swagger/{version}/swagger.json", $"{version}");
        });
    });
}
```

最后将两处变更同步到 Program.cs 中，原本的代码变成了

```
builder.Services.InitSwagger();

...

if (app.Environment.IsDevelopment())
{
    app.InitSwagger();
}
```

现在对于一个新的 Controller，可以指定所属版本并在文档中体现出来

```
[ApiController]
[Route("api/[controller]/[action]")]
[ApiExplorerSettings(GroupName = nameof(APIVersionEnum.Version_0_0))]
```

如何封装一个 NuGet 包

创建一个类库,包含接口和实现,将需要的功能在方法中暴露,将使用的工具隐藏在方法的实现中 在 .csproj 文件中添加 NuGet 包的元数据

```
<PropertyGroup>
        <!-- 编译目标框架 -->
        <TargetFramework>net6.0</TargetFramework>

        <!-- 包唯一标识符 -->
        <PackageId>MyCustomLibrary</PackageId>

        <!-- 包版本 -->
        <Version>1.0.0</Version>

        <!-- 包作者 -->
        <Authors>YourNameOrCompany</Authors>

        <!-- 包描述 -->
        <Description>This is a library.</Description>

        <!-- 包版权信息 -->
        <Copyright>Copyright © 2024 YourCompany</Copyright>

        <!-- 包标签（关键词） -->
        <PackageTags>EFCore;Redis;Caching;Repository</PackageTags>

        <!-- 许可证表达式或自定义许可证文件 -->
        <PackageLicenseExpression>MIT</PackageLicenseExpression>
        <!-- <PackageLicenseFile>LICENSE.txt</PackageLicenseFile> -->
        <!-- 可选，使用自定义文件代替许可证表达式 -->

        <!-- 代码仓库 URL -->
        <RepositoryUrl>https://url</RepositoryUrl>

        <!-- 包图标（需添加到项目并设置“在输出目录中复制”） -->
        <Icon>icon.png</Icon>

        <!-- 发布日志 -->
        <ReleaseNotes>Basic repository and caching support.</ReleaseNotes>

        <!-- 项目主页或文档 URL -->
        <ProjectUrl>https://yourprojecthomepage.com</ProjectUrl>

        <!-- 是否要求用户在安装时接受许可证 -->
```

```
        <RequireLicenseAcceptance>true</RequireLicenseAcceptance>

        <!-- 简短描述 -->
        <Summary>A lightweight library for caching and data access.</Summary>

        <!-- 主要编程语言 -->
        <Language>en-US</Language>

        <!-- 包显示名称 -->
        <Title>My Custom Library</Title>

        <!-- 构建时生成 NuGet 包 -->
        <GeneratePackageOnBuild>true</GeneratePackageOnBuild>
</PropertyGroup>
```

如何将项目做成 Docker 镜像

在项目目录下制作 Dockerfile 文件,注意检查使用的端口

```
# 使用官方 .NET 8 SDK 镜像来构建应用程序
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

# 使用官方 .NET 8 SDK 镜像来进行构建
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src

# 将项目文件复制到 Docker 容器中
COPY ["YourProjectName.csproj", "./"]

# 还原依赖项
RUN dotnet restore "./YourProjectName.csproj"

# 将项目的所有文件复制到容器中并进行构建
COPY . .
RUN dotnet publish "./YourProjectName.csproj" -c Release -o /app/publish

# 创建最终镜像并从生成的发布输出运行应用程序
FROM base AS final
WORKDIR /app
COPY --from=build /app/publish .
ENTRYPOINT ["dotnet", "YourProjectName.dll"]
```

曾经碰到情况为引用了类库,但是不在同一目录下,解决方法

1. 放弃使用类库,转为在该项目中实现
2. 在父级目录下创建 Dockerfile,拷贝使用的所有目录,并使用 .sln 文件来还原依赖

```
# 使用官方 .NET 8 SDK 镜像来构建应用程序
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src

# 将整个解决方案文件夹复制到容器中
COPY . .

# 还原整个解决方案的依赖项，以便处理所有项目的引用
```

```
RUN dotnet restore "SolutionFile.sln"

# 使用解决方案文件来构建项目并发布
RUN dotnet publish "MainProject/MainProject.csproj" -c Release -o /app/publish

# 使用运行时镜像来执行已发布的应用
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS final
WORKDIR /app
COPY --from=build /app/publish .
ENTRYPOINT ["dotnet", "MainProject.dll"]
```

在当前目录下运行 cmd 命令

```
docker build -t yourimagename .
```

最后创建容器,注意检查端口是否冲突

```
docker run -d -p 8080:80 --name yourcontainername yourimagename
```

前言

T-SQL 主要分爲三種類型的陳述式

- DDL
- DML
- DCL

DML

DML(Data Manipulation Language)

數據操作語言,用於操作和處理數據庫中的數據,主要涉及增刪改查等操作

常見用法

select

```
SELECT
        *
FROM
        Employees
```

如果有 `where` 條件,放在 `group by` 或 `order by` 之前

如果有複雜的查詢條件,在使用 EF Core 時,我傾向與分步執行后在內存中整合結果,在程序層處理數據

但是在sql中更加常用的方式是,將結果的查詢放在最外層,然後通過子查詢或公共表達式(Common Table Expressions)來不斷降低每一步的條件複雜度,也就是說將數據處理放在了數據庫層

子查詢示例

```
SELECT
        *
FROM
        Employees
WHERE
        EmployeeID IN (
            SELECT
                    EmployeeID
            FROM
                    Orders
            WHERE
```

```
                OrderDate > '2023-01-01'
        )
```

公共表達式示例

```
WITH RecentOrders AS (
        SELECT
                EmployeeID,
                COUNT(*) AS OrderCount
        FROM
                Orders
        WHERE
                OrderDate > '2023-01-01'
        GROUP BY
                EmployeeID
)

SELECT
        e.FirstName,
        e.LastName,
        r.OrderCount
FROM
        Employees e
        JOIN
        RecentOrders r
        ON
                e.EmployeeID = r.EmployeeID
```

有關 select 的更多使用範例可以查看微軟文檔

insert

```
INSERT INTO
        Employees (FirstName, Department)
VALUES
        ('Jane', 'Marketing');
```

插入表的時候可以選擇只插入部分列

例如上面例子中,Employees 表中可以有不只兩個列的屬性,但是在顯式指定了插入數據是屬於具體的哪些列后,完全少插入一些數據(在不違反完整性唯一性等等條件下)

注意

儘管有以下定義

```
-- Syntax for Azure Synapse Analytics and Parallel Data Warehouse and Microsoft Fabric

INSERT [INTO] { database_name.schema_name.table_name | schema_name.table_name | table_name }
    [ ( column_name [ ,...n ] ) ]
    {
      VALUES ( { NULL | expression } )
      | SELECT <select_criteria>
    }
    [ OPTION ( <query_option> [ ,...n ] ) ]
[;]
```

但事實上,不同的數據庫產品可能會有自己的 SQL 方言,所以强制添加而不是省略引數 into 是更好的做法.
後續示例也都基於這個規則

有關 insert 的更多使用範例可以查看微軟文檔

update

```
UPDATE
        Employees
SET
        Department = 'Marketing',
        FirstName = 'Jane'
WHERE
        LastName = 'Doe'
```

delete

```
DELETE FROM
        Employees
WHERE
        LastName = 'Doe'
```

DDL

DDL(Data Definition Language)

數據定義語言,用於定義和管理數據庫中的結構,主要在修改表結構而不是內容時使用

常見用法

create

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName NVARCHAR(50),
    LastName NVARCHAR(50),
    Department NVARCHAR(50)
)
```

alter

```
ALTER TABLE
        Employees
ADD
        Email NVARCHAR(100)


ALTER TABLE
        Employees
ALTER COLUMN
        LastName NVARCHAR(100)


ALTER TABLE
        Employees
DROP COLUMN
        Department
```

drop

```
DROP TABLE
        Employees
```

DCL

DCL(Data Control Language) 數據控製語言,用於控制對數據庫的訪問權限,主要涉及用戶權限等

其他操作

若非一般的操作,基本上我偏向於在程序代碼中完成,但是也有必須使用SQL語句的情況

merge

merge 是用於將數據插入,更新或刪除到目標表中的一種語句

常常會涉及源表和目標表的概念,并且可以在一個操作中完成複數個操作的功能

```
MERGE
INTO
        Employees AS Target
USING
        NewEmployees AS Source
ON
        Target.EmployeeID = Source.EmployeeID
WHEN
MATCHED
THEN
    UPDATE
    SET
            Target.FirstName = Source.FirstName,
            Target.LastName = Source.LastName,
            Target.Department = Source.Department
WHEN
NOT MATCHED
THEN
    INSERT
            (EmployeeID, FirstName, LastName, Department)
    VALUES
            (Source.EmployeeID, Source.FirstName, Source.LastName, Source.Department)
WHEN
NOT MATCHED BY SOURCE
THEN
DELETE
```

而在 EF Core 中,類似的功能實現如下

```
// 1. 查询所有匹配的记录
var matchedEmployees = context.Employees
    .Where(e => context.NewEmployees.Any(ne => ne.EmployeeID == e.EmployeeID))
    .ToList();

// 2. 更新匹配的记录
foreach (var employee in matchedEmployees)
{
    var newEmployee = context.NewEmployees
        .FirstOrDefault(ne => ne.EmployeeID == employee.EmployeeID);

    if (newEmployee != null)
```

```
    {
        employee.FirstName = newEmployee.FirstName;
        employee.LastName = newEmployee.LastName;
        employee.Department = newEmployee.Department;
    }
}

context.SaveChanges();

// 3. 插入未匹配的记录
var unmatchedNewEmployees = context.NewEmployees
    .Where(ne => !context.Employees.Any(e => e.EmployeeID == ne.EmployeeID))
    .ToList();

context.Employees.AddRange(unmatchedNewEmployees);
context.SaveChanges();

// 4. 删除不再匹配的记录
var employeesToDelete = context.Employees
    .Where(e => !context.NewEmployees.Any(ne => ne.EmployeeID == e.EmployeeID))
    .ToList();

context.Employees.RemoveRange(employeesToDelete);
context.SaveChanges();
```

索引

若數據表的規模較大,在查詢的時候需要花費的時間可能會漫長到無法等待,在列上建立索引可以加快查詢到速度,並有效避免遍歷全部記錄才能夠找到目標

數據表中的索引就相當於在原有的一列記錄上疊加了一個值從而達到優化目的

聚集索引

在表中設置主鍵后,將會自動創建一個聚集索引

聚集索引將會決定表中數據的物流存儲順序,并且只能存在一個聚集索引

除了主鍵,範圍查詢也適合使用聚集索引

```
CREATE CLUSTERED INDEX
        IX_Employees_EmployeeID
ON
        Employees(EmployeeID)
```

非聚集索引

非聚集索引像是對列數據創建了一張跳轉表,適合排序和分組

```
CREATE NONCLUSTERED INDEX
        IX_Employees_LastName
ON
        Employees(LastName)
```

唯一索引

唯一索引僅確保數據唯一,可以是聚集索引也可以是非聚集索引

```
CREATE UNIQUE INDEX
        IX_Employees_Email
ON
        Employees(Email);
```

全文索引

XML索引

空間索引

過濾索引

事務

函數

安裝使用

項目中也許會使用不同的數據庫,甚至可能同時涉及多個不同的數據庫.不管是開發還是維護,都有必要瞭解常見產品的基本使用

- SQL server
-

<折光编年史:reflaction>开发总结

十月底 TapTap 举办了 聚光灯GameJam 活动,我应小新邀请,在活动中作为一名程序参与制作了 <折光编年史:reflaction>

<折光> 是一款卡通风塔防游戏,玩家需要放置防御塔来保护基地不受敌人的攻击.防御塔主要使用光线对敌人攻击,另外设有辅助性质的塔,可提供反射和折射等功能

前期我主要负责了编写敌人相关的逻辑,后期负责整个项目的代码维护和功能添加.现在活动告一段落,遂回想一下这次经历所带来的收获

程序方面

从自己代码中总结

优雅致死,追求完美可能会损失效率

举个例子,在刚开始的阶段,我负责的是所有敌人的相关逻辑编写.出于 cs 的编码思维,我很自然地一开始就使用一个接口来描述所有敌人的基本属性和基本行为,然后继承自该接口,实现了一个敌人的基本类,再着手开始各种敌人的开发

在整个开发的后期,我这样的做法确实带来了很大的便利性.得益于我在设计阶段就考虑得很充分,后续不管增加什么样的敌人,大部分都只需要继承已有的类,然后重写几行特定的行为就可以了.甚至有些敌人可以在不修改代码而只是调整预制体参数设置的情况下完成

但是在开发前期,由于我需要把精力放在思考"如何设计才是更合理的"这个问题上面,导致另一位程序完成许多功能时我还依旧在各种方案之间取舍,结果最后被评价了"过于模式化"┳┳﹏┳┳但其实我完全可以每个类单独开发,重复的代码 CV 一下也花不了几秒钟,对性能也没有什么影响,至于优化完全可以在 PostJam 的阶段完成,那么为什么不呢?完全只是因为我觉得不优雅😭

上面的情况反映出来的**起步阶段的效率丢失**是一方面,另一方面是,GameJam 本身是一种生命周期极短的活动,一个游戏需要在几天之内完成,完成后也不一定会继续开发.就比如这次的<折光>,在策划案中的内容明显比实际开发的内容更加丰富,并且直到发布在 TapTap 上的前夕,成员们都还有继续开发后续内容的打算.然而现实是,直到现在也没有着手开始对游戏进行完善.于是乎,<折光>彻底变成了一上线就"死掉"的项目,我彻彻底底地做了一次**过度设计**

我明显地感觉到,自己学习的东西越多,越有一种想要在一开始就达成"完美设计"的强烈冲动,但是必须得考虑是否值得.短期且不需要重写的东西还是适当将重心放在追求速度上吧~

使用 StartCoroutine 开启协程达到延时效果

在编码时碰到一个需求:我要在指定时间间隔后执行一些逻辑

在 .NET 程序中,一般会采用 Task.Delay 方法结合 await/async 来实现这个功能,但是 Task 实际上是采用线程实现的,而在 Unity 中,尽管存在对 Task 的支持,但是在非主线程上获取物体是会出问题的,所以这时候需

要换用协程来处理

例如以下代码:

```
public bool Attackable
{
    get => attackable;
    set
    {
        attackable = value;
        if(attackable == false) StartCoroutine(ResetAttackableAfterDelay(AttackInterval));
    }
}

private IEnumerator ResetAttackableAfterDelay(float delay)
{
    yield return new WaitForSeconds(delay);
    attackable = true;
}
```

使用 **`StartCoroutine` 方法开启一个协程**,在 `Unity` 运行到这个地方时,会在每一帧检查迭代器 `ResetAttackableAfterDelay` 中的 `WaitForSeconds` 方法是否完成,完成后才会执行语句 `attackable = true;`, 从而达到本处延时但不影响其他部分的正常运行(不会阻塞主线程)

对应的 `.NET` 版本应该是这样

```
private Task resetTask;

public bool Attackable
{
    get => attackable;
    set
    {
        attackable = value;
        if (attackable == false) resetTask = ResetAttackableAfterDelay(AttackInterval);
    }
}

private async Task ResetAttackableAfterDelay(int delay)
{
    await Task.Delay(delay);
    attackable = true;
}
```

使用 PlayerPrefs 存储临时数据

PlayerPrefs 是 Unity 提供的一个轻量级数据存储系统,适合用来**保存简单的数据**.

我一开始并不知道有它的存在,于是在单个场景中使用单例实现数据的临时存放.但是后续出现了要在多个场景之间进行变量传递的需求,此时我发现 PlayerPrefs 是一个很好的工具,就像一个随时可用的字典

- 存储数据

```
PlayerPrefs.SetString(string key, string value);
PlayerPrefs.Save();
```

- 获取数据

```
PlayerPrefs.GetString(string key, string defaultValue = "");
```

- 删除数据

```
PlayerPrefs.DeleteKey(string key);
```

甚至它可以在应用退出的时候自动保存数据,所以条件有限的时候还可以勉强用来做持久化.更多内容参考官方文档的描述

使用 JSON 来保存数据

和"困觉"的小伙伴们一起开发的时候,曾经尝试过将数据存放在 csv 中,而这次我选择了使用 JSON 来存放关卡信息.这样的选择主要考虑了几方面:

- 希望可以在游戏发布后方便地修改数据(自带老金)
- 没有尝试过在 Unity 中使用 JSON

首先,我将关卡信息放在 Unity 默认存放数据用的 Assets/Resources 文件夹下,这样可以确保使用 Resources.Load 方法获取文件

接下来,将持久化文件放在各个平台对应的默认存放位置,这个位置可以使用 Application.persistentDataPath 获取到

至此,完成了数据在目标平台的存放,读取也很简单

首先使用 File.ReadAllText 方法读取文件,再使用 JsonUtility.FromJson<T> 方法转换成相应的类型,这部分的操作和 .NET 程序中是完全一样的

总体上来说,在 Unity 中使用 JSON 来存放信息的**操作没有特殊的变化**.需要注意的是,虽然我在使用的时候一切正常,但是 Resources.Load 这个方法被描述为"不支持直接加载 JSON"(我使用的泛型类型为 TextAsset,估计把它当作 txt 去读了)

更加推荐的方式是将文件放入 Assets/StreamingAssets 文件夹,然后这样使用

```
string path = System.IO.Path.Combine(Application.streamingAssetsPath, "文件名");

UnityWebRequest request = UnityWebRequest.Get(path);
yield return request.SendWebRequest();

string jsonContent = request.downloadHandler.text;
// 解析
```

UnityWebRequest 是异步操作,需要用上文提到的方式去开启协程来调用.这种方式效率较低,一般用于**需要跨平台并确保发布时不易更改的静态文件**,比如这次的关卡信息就很合适

从他人代码中学习

使用 LineRender 制作射线

防御塔的主要攻击手段是发射光线到敌人身上,这部分的实现小伙伴采用了 LineRender 这个类型来绘制光线,这恰好是我不了解的知识

LineRender 作为组件挂载到 GameObject 上,简单的绘制激光主要需要设置:

- 线条宽度

```
lineRenderer.startWidth = 0.1f;
lineRenderer.endWidth = 0.1f;
```

- 线条颜色

```
lineRenderer.material = new Material(Shader.Find("Unlit/Color"));
lineRenderer.material.color = laserColor;
```

- 起点和终点

```
lineRenderer.SetPosition(0, laserOrigin.position);
lineRenderer.SetPosition(1, hit.point);
```

可以参考官方文档中对此的相关描述

安装 Aseprite

`Aseprite` 支持用户自行编译使用, 仓库位于这里

INSTALL.md 中有基本的安装指南, 以下是在win10上的实践

平台

我使用的是 Windows 10, 编译的 `Aseprite` 版本为 1.3.9

依赖

使用 Windows 编译 `Aseprite` 需要:

- Visual Studio VisualStudio 不是必须的, 使用 VisualStudio 是为了选择 `Visual Studio installer` 中的工作负荷 使用c++的桌面开发, 这个工作负荷会帮助安装好
  - 3.16以上版本的 CMake
  - Ninja

  这两个才是必要的, 可以自行安装, 如果使用 VS 的话, 可以在安装目录的 `Common7\IDE\CommonExtensions\Microsoft\CMake` 这个路径下找到这两个工具
- 编译后的 Skia 库 去看仓库中的 README 文件里推荐使用哪个 branch, 我在安装时使用了 m102

  下载发行版可以省去编译的环节,注意需要和平台对应, 我选择的是x64

确保拥有以上内容后, 可以在系统环境变量中将 cmake 和 ninja 添加到 Path 中, 遇到添加后不生效的情况可以关闭所有 cmd 后使用以下命令来刷新验证是否成功添加(或者重启来刷新):

```
echo %PATH%
```

编译

`Visual Studio` 的编译器可能无法直接被找到, 因此, 需要使用 `Developer Command Prompt for VS` 运行编译的指令

> **注意**

在开始菜单中的 `Developer Command Prompt for VS` 默认是32位, 如果编译目标为64位, 需要使用 `Common7\Tools\VsDevCmd.bat` 这个批处理

```
call "VsDevCmd.bat" -arch=x64
```

在 cmd 中运行以上命令后, 便进入了 `Developer Command Prompt for VS` 工具

由于 `Aseprite` 中 CMakeList 文件使用的编译器名称和 VS 中不同, 此时还需要进一步设定来帮助 CMake 指定编译器:

```
set CC=cl
set CXX=cl
```

现在, 可以在想要的输出目录下运行以下命令(根据自己的变量名替换):

```
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo -DLAF_BACKEND=skia -DSKIA_DIR=C:/deps/skia -
DSKIA_LIBRARY_DIR=C:/deps/skia/out/Release-x64 -DSKIA_LIBRARY=C:/deps/skia/out/Release-
x64/skia.lib -G Ninja ..
```

成功后, 运行:

```
ninja aseprite
```

生成的可执行文件在输出目录的 `\bin` 下