第一章　网络编程

http://www.cnblogs.com/mushroom/p/5079964.html

1.1　概述

作为一个 Universal Windows Platform (UWP) 开发者，如果你尝试使用 http 与 web 服务或其他服务端通讯时，有多个 API 可以选择。UWP 中最常见并推荐使用的 HTTP 客户端 API 实现是 `System.Net.Http.HttpClient` 和 `Windows.Web.Http.HttpClient` 。

关于这些 APIs 不同之处，从功能上来说两组 APIs 是上相等的，那在不同场景下选择哪一个呢，诸如此类的问题。在这篇文章中，我们会去尝试定位这些问题，理清楚这两组 APIs 的用途及使用场景。

第一个推荐 AIP 是 `System.Net.Http.HttpClient`，相比旧的 HttpWebRequest API，这个 API 的目标是提供一个简单的，干净的抽象层，比较灵活的实现 http 客户端功能。比如，它允许链接自定义处理器，开发者可以拦截每个 request 和 response，去实现自定义逻辑。在 windows8.1 之后，所有功能都在.NET 下面实现。在 windows10 UWP 中这个 API 实现移到 `Windows.Web.Http`和 `WinINet Http`层上。

另外一个推荐 API 是 `Windows.Web.Http.HttpClient`，增加这个 API 的主要目是，把不同 windows 应用开发语言 (C#, VB, C++, JavaScript) 下，不同 Http APIs 合成一个，它支持上述 APIs 的所有特性。大多数基础 API 都是从 System.Net.Http 派生的，在 Windows HTTP 基础上实现。

1.2　如何选择

在 UWP 中这些 HTTP API 都是可以使用的，对于开发者来说最大的问题是在 APP 中应该使用哪一个。其答案取决去几个因素：

- 是否需要结合本地 UI 收集用户证书，控制 HTTP 缓存读和写，或者通过指定的 ssl 客户端证书去做认证？如果需要认证，那是应使用 Windows.Web.Http.HttpClient。在现在的 UWP 中，Windows.Web.Http 提供 HTTP 设置，它比 System.Net.Http API 更好的控制这些。在未来的版本，也会加强支持 System.Net.Http 在 UWP 中的特性。

- 是否考虑写**跨平台**的.NET 代码 (跨 UWP/ASP.NET 5/IOS 和 Android)?如果需要,那使用 System.Net.Http API。它可以让你写的代码复用在其他.Net 平台上，比如 ASP.Net 5 和.NET 桌面平台应用。通过使用 Xamarin，这些 API 在 IOS 和 Android 中也得到支持。

现在就比较好理解为什么会有两个相似 APIs 了，也了解怎么在二者之间进行选择，下面进一步了解这两个对象模型。

1.3 System.Net.Http

其 HttpClient 对象是最顶端的抽象模型，在 HTTP 协议 client-server 模型中它表示 client 这部分。其 client 能发出多个 request 请求 (用 HttpRequestMessage 表示) 到服务端上，从服务端接收响应 (用 HttpResponseMessage 表示)。用 HttpContent 基类和它派生出的类，表示对象 body 和每个 request 或 response 的 content 头部，比如 StreamContent，MultipartContent 和 StringContent。它们表示各种 http 实体 body 内容。这些类都会提供 ReadAs 开头的一组方法，它能从请求或响应实体 body 中，以字符串形式、字节数组、流形式读取内容。

每一个 HttpClient 对象下都有一个处理者对象，它表示 client 下所有与 HTTP 相关的配置。从概念上来说，可以认为它是 client 部分下 HTTP 协议栈的代表。在客户端发送 HTTP 请求到服务端和传输数据到客户端上，它是非常可靠的。

在 System.Net.Http API 中默认处理者是 HttpClientHandler。当你创建 HttpClient 对象实例时，会使用默认 HTTP stack 设置，自动帮你创建一个 HttpClientHandler。如果你想修改默认一些设置，比如缓存行为，自动压缩，证书或代理，可以直接创建一个 HttpClientHandler 实例，修改它的属性，把它当做 HttpClient 构造函数的参数传入。这样 HttpClient 对象就会使用我们自定义的处理器，如下:

```
HttpClientHandler myHandler = new HttpClientHandler();
myHandler.AllowAutoRedirect = false;
HttpClient myClient = new HttpClient(myHandler);
```

1.3.1 链式处理器

System.Net.Http.HttpClient API 设计中一个重要优势是：能够插入自定义处理器、在 HttpClient 对象下创建一连串的处理器。例如：构建一个 app，它从 web 服务中请求一些数据。这时就可以自定义逻辑去处理 HTTP 服务端响应的 4xx (客户端错误) 和 5xx (服务端错误)，使用具体的重试步骤，比如尝试不同的端口请求或添加一个用户认证。还可能会想从业务逻辑部分分离出 HTTP 相关的工作，它只关心 web 服务的数据返回。

这就可以使用自定义处理器类来完成，它从 DelegatingHandler 派生出，例如 CustomHandler1，然后创建一个新实例，把它传入 HttpClient 构造函数。DelegatingHandler 类的 InnerHandler 属性被用指定下一个处理器，比如，可以添加个新的自定处理器 (例 CustomHandler2) 到处理链上。处理链上最后一个处理者的 InnerHandler，可以设置成 HttpClientHandler 的实例，它将传递请求到系统的 HTTP 协议栈上。从概念上来看如下图：

下面是完成这部分的例子代码：

```
public class CustomHandler1 : DelegatingHandler
{
    protected async override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
            Debug.WriteLine("Processing request in Custom Handler 1");
            HttpResponseMessage response = await base.SendAsync(request,
                cancellationToken);
            Debug.WriteLine("Processing response in Custom Handler 1");
            return response;
        }
}

public class CustomHandler2 : DelegatingHandler
{
    // Similar code as CustomHandler1.
}

public class Foo
{
    public void CreateHttpClientWithChain()
    {
        HttpClientHandler systemHandler = new HttpClientHandler();
        CustomHandler1 myHandler1 = new CustomHandler1();
        CustomHandler2 myHandler2 = new CustomHandler2();

        // Chain the handlers together.
```

```
        myHandler1.InnerHandler = myHandler2;
        myHandler2.InnerHandler = systemHandler;

        // Create the client object with the topmost handler in the chain.
        HttpClient myClient = new HttpClient(myHandler1);
    }
}
```

说明：

如果你试图发送一个请求到远程服务器端口上，其链上最后的处理器通常是 HttpClientHandler，它实际是从系统 HTTP 协议栈层面发送这个请求或接收这个响应。作为一种选择，可以使用一个模拟处理器，模拟发送请求到服务器上，返回一个伪造的响应，这可以用来单元测试。

在传递请求到内部处理器之前或响应处理器之上，添加一个处理逻辑，能减少性能消耗。这个处理器场景下，最好能避免使用耗时的同步操作。

关于链式处理概念的详细信息，可以看 Henrik Nielsen 的这篇博客,(注意文章参考的是 ASP.NET Web API 的 API 版本。它和本文讨论的.NET framework 有一些细微的不同，但在链式处理器上的概念是一样的)

1.3.2 Windows.Web.Http

Windows.Web.Http API 的对象模型跟上面描述的 System.Net.Http 版本非常 ，它也有 client entity 的概念，一个处理器 (在这叫“filter”过滤器)，及在 client 和系统默认过滤器之间选择是否插入自定义逻辑。

其大多数类型是直接类似于 System.Net.Http 的类型的，如下:

在上面关于 System.Net.Http API 的链式处理器讨论，也可应用于 Windows.Web.Http API，这里你可以创建自定义链式过滤器，传递它们到 HttpClient 对象的构造函数中。

第二章　数据库的使用

2.1　使用 SQLite

SQLite 是开源的独立数据库，其优势在于可以完全独立工作，不需要太多设置。自 Windows 10 Anniversary Update (Build 14393) 开始，Windows 10 SDK 内置了 SQLite。

在开发时，出于以下几方面的原因，建议直接采用系统内置的 SQLite SDK：

- 可以减小程序的安装包的大小。
- 可以加快程序的启动时间，因为 SDK 版的 SQLite 很有可能已经在内存中了。
- 无需关心 SQLite 的更新，因为 Windows 系统会完成该工作。

关于 SQLite 更多的知识，可以参考：`http://sqlite.org/`

2.1.1　通过 c++ 使用 SQLite

在 UWP 开发的过程中，通过 c++ 使用 SQLite，目前由三种方法（针对于使用 c++ 的程序，非我的重点！）：

1. 使用 SDK 中的 SQLite。
   (a) 首先需要引入库：
   ```
   #include <winsqlite/winsqlite3.h>
   ```
   (b) 设置项目链接到“winsqlite3.lib”。
2. 通过 App Package 来包含 SQLite。
3. 在 Visual Studio 中直接编译 SQLite 源代码。

2.1.2 通过 C# 使用 SQLite

安装 SQLite 的 C# 包装器

SQLite 采用 C 语言开发,在通过 C# 使用 SQLite 时需要安装 C# 包装器。推荐使用“**Microsoft.Data.Sqlite**”:

1. 在“Solution Explorer”中右键单击“Reference”，选择“Manage NuGet Packages”。
2. 在“Installed”选项卡中，确保“Microsoft.NETCore.UniversalWindowsPlatform ”的版本大于 5.2.2。
3. 在“Browse”选项卡中，搜索“Microsoft.Data.Sqlite”，点击“Install”。

当然，更简单的安装方法时：

```
Install-Package Microsoft.Data.SQLite
```

引入 SQLite 命名空间

在需要使用 SQLite 的 C# 代码中需要引入命名空间：

```
using Microsoft.Data.Sqlite;
using Microsoft.Data.Sqlite.Internal;
```

在 App 的 Constructor 中，使用 SQLite 之前，需要使用：

```
SqliteEngine.UseWinSqlite3();
```

来确保程序使用 SDK 版的 SQLite。

创建 SQLite 数据库

示例代码如下：

```
string conn_string = "Filename=sqliteSample.db"
string create_table_sql = @"
    CREATE TABLE IF NOT EXISTS book (
        id    INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
        title TEXT    NOT NULL,
        isbn  TEXT    NOT NULL UNIQUE);
    ";
using (SqliteConnection db_conn = new SqliteConnection(conn_string))
{
    db.Open();
    SqliteCommand createTable = new SqliteCommand(create_table_sql, db_conn);
    try
    {
        createTable.ExecuteReader();
    }
    catch (SqliteException e)
    {
        //Do nothing
    }
    db.Close()
}
```

插入数据

```
string conn_string = "Filename=sqliteSample.db"
string book_title = "Molecular Cloning: A Laboratory Manual"
string book_isbn = "978-1-936113-42-2"

using (SqliteConnection db_conn = new SqliteConnection(conn_string))
{
    db.Open();
    SqliteCommand insert_cmd = new SqliteCommand(
        @"INSERT INTO TABLE book (title, isbn) VALUES (@title, @isbn)",
        db_conn);
    insert_cmd.Parameters.AddWithValue("@title", book_title);
    insert_cmd.Parameters.AddWithValue("@isbn", book_isbn);
    try
```

```
    {
        insert_cmd.ExecuteReader();
    }
    catch (SqliteException e)
    {
        //Do nothing
    }
    db.Close()
}
```

查询数据

```
public class Book {
    public int Id;
    public string Title;
    public string Isbn;
}

List<Book> books = new List<Book>();

string conn_string = "Filename=sqliteSample.db"
string book_title = "Molecular Cloning: A Laboratory Manual"

using (SqliteConnection db_conn = new SqliteConnection(conn_string))
{
    db.Open();
    SqliteCommand select_cmd = new SqliteCommand(
        @"SELECT id, title, isbn FROM book WHERE title = @title",
        db_conn);
    select_cmd.Parameters.AddWithValue("@title", book_title);
    SqliteDataReader query;
    try
    {
        query = select_cmd.ExecuteReader();
```

```
    }
    catch (SqliteException e)
    {
        //Do nothing
    }
    while (query.Read())
    {
        books.Add(new Book(
            Id = query.GetInt(0),
            Title = query.GetString(1),
            Isbn = query.GetString(2)
        ));
    }
    db.Close()
}
```

2.2 Entity Framework Core

https://docs.microsoft.com/en-us/ef/core/get-started/uwp/getting-started

2.3 MySQL