序列化和反序列化利器 Json.NET

我们在写服务的时候，尤其是 api，会经常碰到一些需要将对象序列化处理返回的的服务接口，那么我们通常做法是使用 new JavaScriptSerilizer().Seriliaze(object o)，这种用法，当然还有一些其他的使用形式，比如使用 DataContractJsonSerializer 类进行序列化操作，这两种用法都很简单，相信有很多人用过。不多说，不会的 google 之。

如果说序列化我们理解起来很容易，比如，我们写一个服务，通过 ajax 来访问，ws 返回一个对象(可以是集合数组表或者其他类型的对象)。前台接收的时候，即使你在后台没有显示的进行序列化操作，那么返回的结果是 json，会自动的给序列化后返回。这种情形不多说，很常见。注意，ws 只能返回 xml 和 json 两种数据类型，即使你在后台不显示的序列化，服务在返回时也会自动执行序列化操作，貌似是[ScriptService]这个属性声明的缘故，不可能返回原始对象的。

什么情况下用，假设一种使用场景。我们需要在某个时刻持久化保存某个对象，我们能做什么，将这个对象序列化保存到文件就 ok 了。可以同时保存这个对象的结构和数据。

如果我们需要保存这些对象的结构，也是同样的道理，可以将他们序列化为 xml 保存下来。

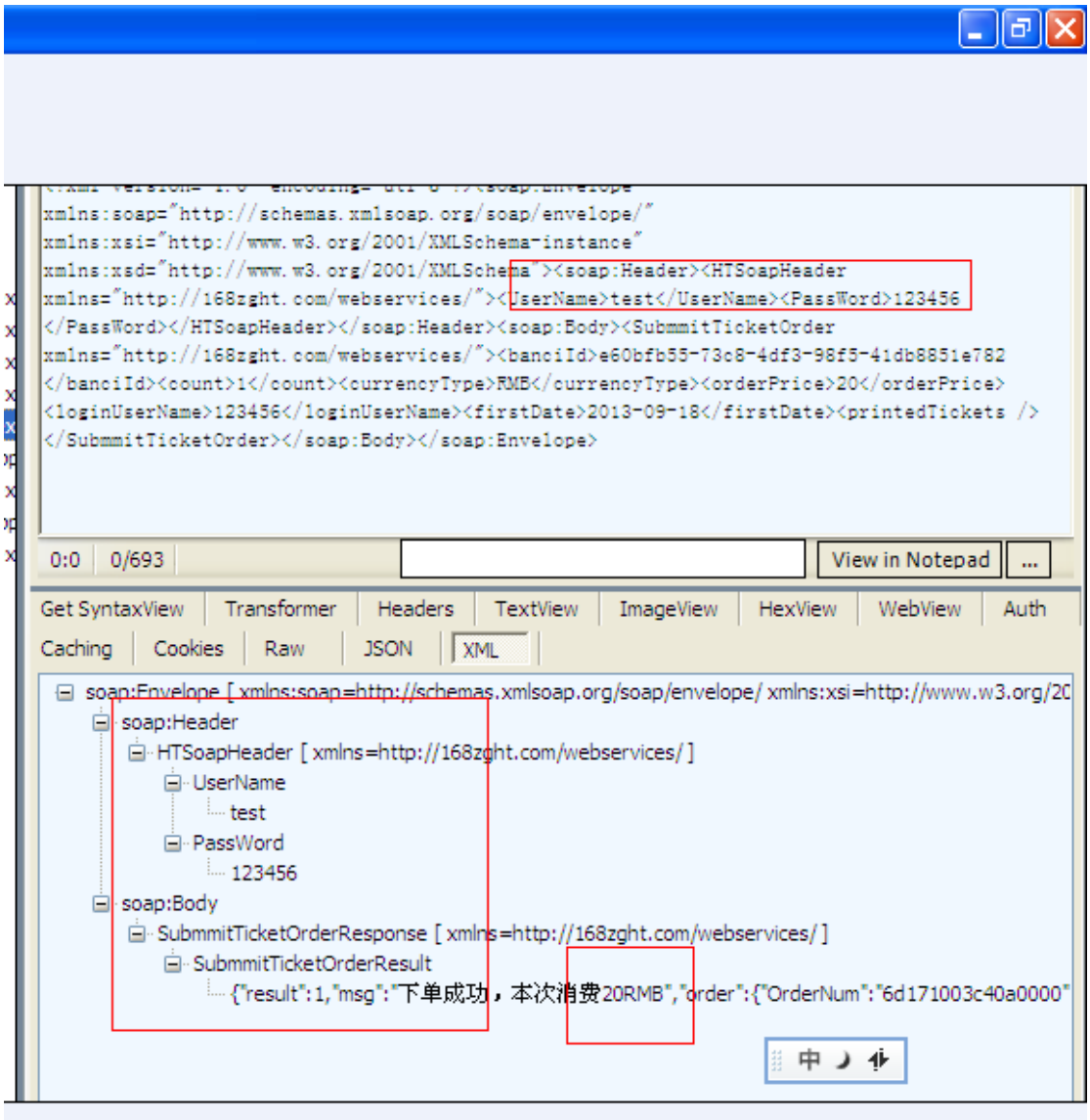
好到这里，序列化的使用场景差不多了。那么我们什么时候需要反序列化呢，很简单。如果我们需要将持久化文件中的数据还原成我们原始的对象，那么我们就得反序列化操作。还原出原始的数据结构和数据值。

在现在这个技术更替如此之快的时代。我们迫切的需要一些高效更好用效率更高性能更优的工具。在一些序列化和反序列化使用频繁的方案里，如果你还在用一些什么百度来的 xmlHelper 什么 JsonHelper，此种云云。果断的 out，如果你是一个停留在铁器时代的 coder。，如果你对工具类有顽强的偏爱。你可以关掉有右上角的哪个×了。

我说个具体使用场景，目前华通项目中用到的。如果采用分布式的服务方案，将服务和设计剥离开来，核心服务通过 api 接口提供，client 和 Service 分开部署。那么我们访问 api 时，返回的结果肯定是 xml 或者 json。那么这个时候，client 在拿到服务处理结果后，需要做进一步的处理。我们迫切的需要将 result 还原成便于我们 client 平台能够处理的对象，这样将返回的结果强类型化，便于我们使用和操作，也会大大的减小出错的可能性。

以上的这种服务需求，微软有一套技术方案出来，那就是 wcf，一套完整的跨平台分布式服务方案。感兴趣的可以多了解学习。

多说一点，目前我们的 api 仅仅是通过 ws 来提供，通信协议通过 Soap。但是存在不少问题。我发现的最严重的是授权的安全性问题，见图



有很多工具能抓取通信报文、这个是我用 fiddler 抓取的 api 访问的通信报文，观察发现报文头中用户授权的 username 和 Password 居然是明文。。。。。。。。。。。。如果我们给别人提供的产品”很受欢迎”，相信我，访问令牌是明文，这种访问权限控制迟早会悲剧。

WCF 是建立在传统的 webService 架构上的一个全新的通信平台。我理解它是 donet 平台下的全新的 ws 方案。所以我个人认为，未来 api 的提供方案需要从传统的 ws 过渡到 WCF。因为他是为跨平台而生，安全性更高，提供的通信方式更多。这是需要的并且也是迫切的，而且从现在各个项目的情况来看，移动 app 的兴起，越发的表现出对 api 接口的可靠性安全性和平台无关性有要求。目前 ht 在开发 winform 版本的时候，传统的 api 引用形式，已经碰到了一些问题。当我苦思不得其解的时候，我改用 wcf 的服务引用形式。问题迎刃而解、同时我发现 wcf 在使用上完全兼容的传统的 web 引用这种形式，让我越加坚信。Wcf 会替代传统 ws。分布式技术和 SOA(面向服务的体系结构)浪潮早已经掀起。所以，wcf 替代传统的 ws 是必然，但是目前的一些方案，如同去年 wgc 和小魏在参加完世界开发者大会回来的分享说的一样。目前我们的有些技术方案真的落后很多很多

我们需要决心也需要勇气，完成一些方案的过渡，以适应日渐复杂多样的产品要求。否则，再过一年，甚至几年，我们依旧再采用历年项目沿用下来的方案，最后的结果我相信不说都能懂。

回到我们本次的主题上来，序列化和反序列的使用场景都说了下，那么接下来我具体介绍我要分享的核心组件 Json.net。这个组件是一个私人的开源项目，而且被很多 it 服务商广泛采用。VS10 在创建 Mvc 项目的时候，会自动引入这个组件，和 Jquery 一样，可见这个组件的流行程度。

说了些废话。继续介绍组件，官网上的介绍

1. Json.net 可以在.NET 对象和 Json 之间实现灵活的转换
2. Linq to Json 技术可以更为方便的对进行进行读写操作
3. 高性能，远远比 Mincrosoft 提供的 Json 序列化器性能要高
4. Json 数据交错式写入，数据可读性更好
5. 可实现 Json 到 xml 的双向转换
6. 多平台的支持，NET 2, .NET 3.5, .NET 4, Silverlight and Windows Phone
7. 最后，我自己加上一点，他的这个组件，开源，并且在持续更新修复 bug

在 donet 平台下。微软的 Linq 技术大行其道，其优雅性和灵活性我非常喜欢。可喜的看见看见国内外很多开源组件和相关的中间件在与 Linq 技术无缝结合。这也是特别喜欢的一条。用过 Nhibernate 或者 Hibernate 的应该知道。新的 NH 框架也在吸取借鉴 LInq 的一些优秀特性，比如新版的 Nh 提供了流式配置，流式查询等。给众多使用者带来的极大的便利。
Java 平台下的一些优秀组件也吸收 Linq 的一些特点
，比如 Linq4J。

好了，不多说，讲使用把，
1.反序列化普通类型

```
//创建服务实例建议加上命名空间
using (var ws = new PrintTicketWebService())
{
    try
    {
        ws.HTSoapHeaderValue = htSoapHeader;
        var tempData = ws.GetTicketsList("8ababe02cc1f7d14", null);
        //注意，如果添加本地服务示例(localhost)，那么将代理类中将生成对应的返回类型(原因尚不明确)
        //服务调用结束后，请使用bo层中的bo类型来接收反序列化得到的实体，如下使用
        APIReturnBO bo = JsonConvert.DeserializeObject<APIReturnBO>(tempData);
        //装箱拆箱性能影响待考证
        SummitOrderReturnBO a = ((JObject)(bo.order)).ToObject<SummitOrderReturnBO>();
    }
    catch (SoapException exception)
    {
        LogHelper.WriteLog(string.Format("GetTicketsList({0})远程服务调用出错", "8ababe02cc1f7d14"), exception);
    }
}
```

1 处拿到 json
2.使用泛型反序列化方法得到强类型 bo 实体
3.第 3 处强类型访问 order，强制转换为 JObject，再调用 toObject<T>的 泛型方法，注意 order 的实际类型是 JObject 类型

2.反序列化对象后获取值类型数据

```
//服务创建
//获取用户账户余额示例
using (var ws=new VipUser.LoginWebService.LoginWebService())
{
    try
    {
        ws.HTSoapHeaderValue = htSoapHeader;
        var reliable= ws.GetUserCountReliableMoney("w23002");
        //使用JObject中间类型过度
        var moneyTemp = (JObject)JsonConvert.DeserializeObject(reliable);
        var rmb = moneyTemp["BalanceRMB"]==null?0L:moneyTemp["BalanceRMB"].Value<long>();
        var hkd = moneyTemp["BalanceHKD"]==null?0L:moneyTemp["BalanceHKD"].Value<long>();
    }
    catch (SoapException e2)//注意第一级异常捕获类型为SoapException,
    {
        LogHelper.WriteLog(string.Format("GetUserCountReliableMoney({0})远程服务调用出错", "w23002"), e2);
    }
    catch (Exception exception)
    {
        LogHelper.WriteLog(string.Format("GetUserCountReliableMoney({0})远程服务调用出错", "w23002"), exception);
    }
}
```

1. 2 处反序列化数据为中间类型 JObject
2. 1 处是使用注意判空，
3. 3 处使用 Value<T>泛型方法获取值

4. 反序列化集合类型

```
using (var ws=new VipUser.TicketInfoService.TicketInfoService())
{
    ws.HTSoapHeaderValue = htSoapHeader;
    //注意，api服务可靠性不是绝对的，内部api调用需要添加异常处理
    //防止调用抛异常导致form挂掉
    try
    {
        var result= ws.GetAllSoldTicketsInfo("w23002", "2013-9-2", null);
        //这里需要对result进行判空指处理，否则反序列化组件抛异常
        if(result != null)
        {
            //使用Jobject中间类型过度，正确用法如下
            JObject tempInfo = (JObject)JsonConvert.DeserializeObject(result);
            var tempCout=tempInfo.Count ;
            var list = tempInfo["list"];
            var baduse=JsonConvert.DeserializeObject<List<SoldTicketCountSumBO>>(list.ToString());
            var ticketInfos=list.ToObject<List<SoldTicketCountSumBO>>();//得到所售车票情况
            //弱引用类型访问取值前注意判空和空指
            var rmb = tempInfo["RMB"]==null?0L:tempInfo["RMB"].Value<long>();//人民币售票总额
            var hkd = tempInfo["HKD"]==null?0L:tempInfo["HKD"].Value<long>();//港币售票总额
        }
    }
```

1 处，访问 api 获取 Json 数据

2 处，反序列化为为 JOject 中间类型

3 处，弱引用访问，我很不喜欢这种访问方式，但是目前我还没有找到能强引用访问方式。

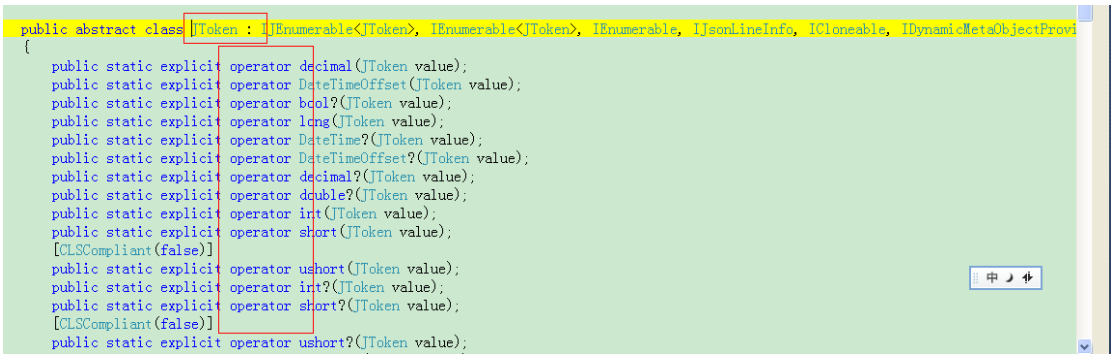
4 处，这里尤其注意，这是不好的用法。会影响性能。为什么呢，大家看下 2 处。list 得到 Jobect 类型以后再次 tostring，意味着会再次解析，会 string->xml->JObject->object(非 CLR 中的 Object，指我们自己定义的 Poco 类型)的重复过程,影响性能。Jobect 是一种介于 xml 和 object 中间类型，得到 JObject 的时候意味着已经完成了 string->xml 的解析。为什么要拿解析过的数据再次 tostring，仅仅是为了重新调用 deseriliazerObject<T>方法。相对更优的办法是，拿到 JObject 以后，使用 ToObject<T>（）方法得到我们需要的对象。这样用不会有二次解析的问题,如 5 处使用。

6 处，同样的，我们要获取值，那么我们需要在取值的时候进行判空处理，再调用 Value<T>

5. Linq to Json 的使用

在讲他的 Linq 用法之前，我先说下该组件中的几个对象类型

JToken，定义了 Json.NET 中的值对象，基元类型。注意是值对象，Json 是键值对的形式。JToken 定义的是键值对中的值对象。这是 Json.NET 中最基本的也是最重要的类型。在 LInq to Json 中使用的对象，都派生于 JToken，也就是是 Jobject，Jproperty，JArray 都可以通过转换来表示为 Jtoken 的表示形式



```
public abstract class JToken : IJEnumerable<JToken>, IEnumerable<JToken>, IEnumerable, IJsonLineInfo, ICloneable, IDynamicMetaObjectProvi
{
    public static explicit operator decimal(JToken value);
    public static explicit operator DateTimeOffset(JToken value);
    public static explicit operator bool?(JToken value);
    public static explicit operator long(JToken value);
    public static explicit operator DateTime?(JToken value);
    public static explicit operator DateTimeOffset?(JToken value);
    public static explicit operator decimal?(JToken value);
    public static explicit operator double?(JToken value);
    public static explicit operator int(JToken value);
    public static explicit operator short(JToken value);
    [CLSCompliant(false)]
    public static explicit operator ushort(JToken value);
    public static explicit operator int?(JToken value);
    public static explicit operator short?(JToken value);
    [CLSCompliant(false)]
    public static explicit operator ushort?(JToken value);
```

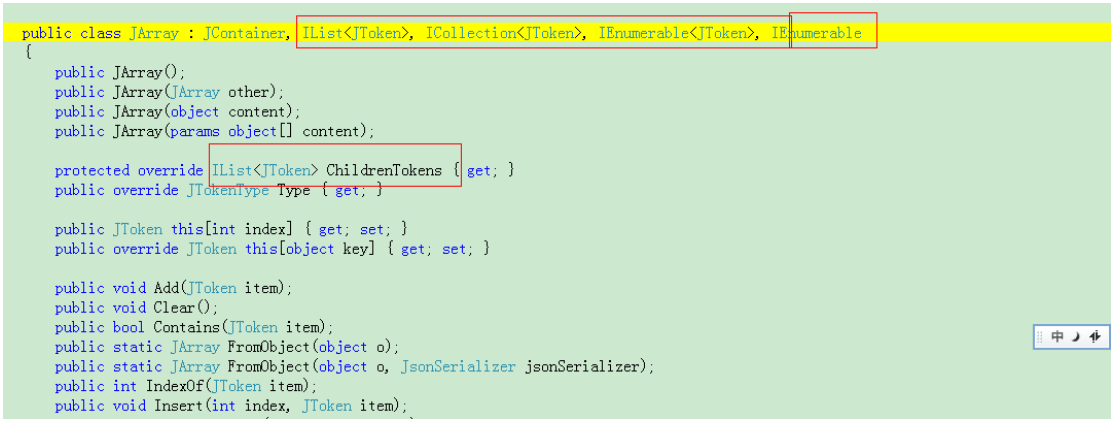
声明了很多 JToken 类型和 C#基本类型转换的操作符，直接支持隐式转换。这里可以很明显的看出 Jtokent 和其他几种类型的派生关系(支持隐式转化)

JObject。上面以前提到过，我理解的是介于 xml 和 Poco 对象类型的的“弱”基础对象，他就相当与 Js 中的 Json 对象，只不过现在他成了在 C#语言中的 Json 对象，服务器端的 Json 对象的存在形式。看 dll 中的定义

```
namespace Newtonsoft.Json.Linq
{
    public class JObject : JContainer, IDictionary<string, JToken>, ICollection<KeyValuePair<string, JToken>>, IEnumerable<KeyValuePair<strir
    {
        public JObject();
        public JObject(JObject other);
        public JObject(object content);
        public JObject(params object[] content);

        protected override IList<JToken> ChildrenTokens { get; }
        public override JTokenType Type { get; }

        public override JToken this[object key] { get; set; }
        public JToken this[string propertyName] { get; set; }

        public event PropertyChangedEventHandler PropertyChanged;
        public event PropertyChangingEventHandler PropertyChanging;

        public void Add(string propertyName, JToken value);
        public static JObject FromObject(object o);
        public static JObject FromObject(object o, JsonSerializer jsonSerializer);
        public IEnumerator<KeyValuePair<string, JToken>> GetEnumerator();
        protected override DynamicMetaObject GetMetaObject(Expression parameter);
```

很明显的看到，他实现了键为 stirng，值 Jtoken 类型字典接口，并实现了键为 stirng，值 Jtoken 类型 ICollection 接口，并且也同时实现了枚举器接口。为 Linq 的遍历访问提供了支持，为弱引用类型访问提供支持。还记得上面的 tempInfo[“RMB”]这种访问形式么，看了 JObject 的定义再明白不过了。JObject 定义的是一个键值对对象，包括键和值，内部对象可嵌套。你看下 ChildrenToken 对象，这个是集合类型。这也很好理解了，前端的 Json 对象是可以嵌套的，那么在服务端，JObject 形式存在的 Json 对象中仍然是可以嵌套的

JArray，定义了值集合对象，JToken 对象的数组形式，看定义，他也实现了枚举器接口还实现了 IList 容器接口



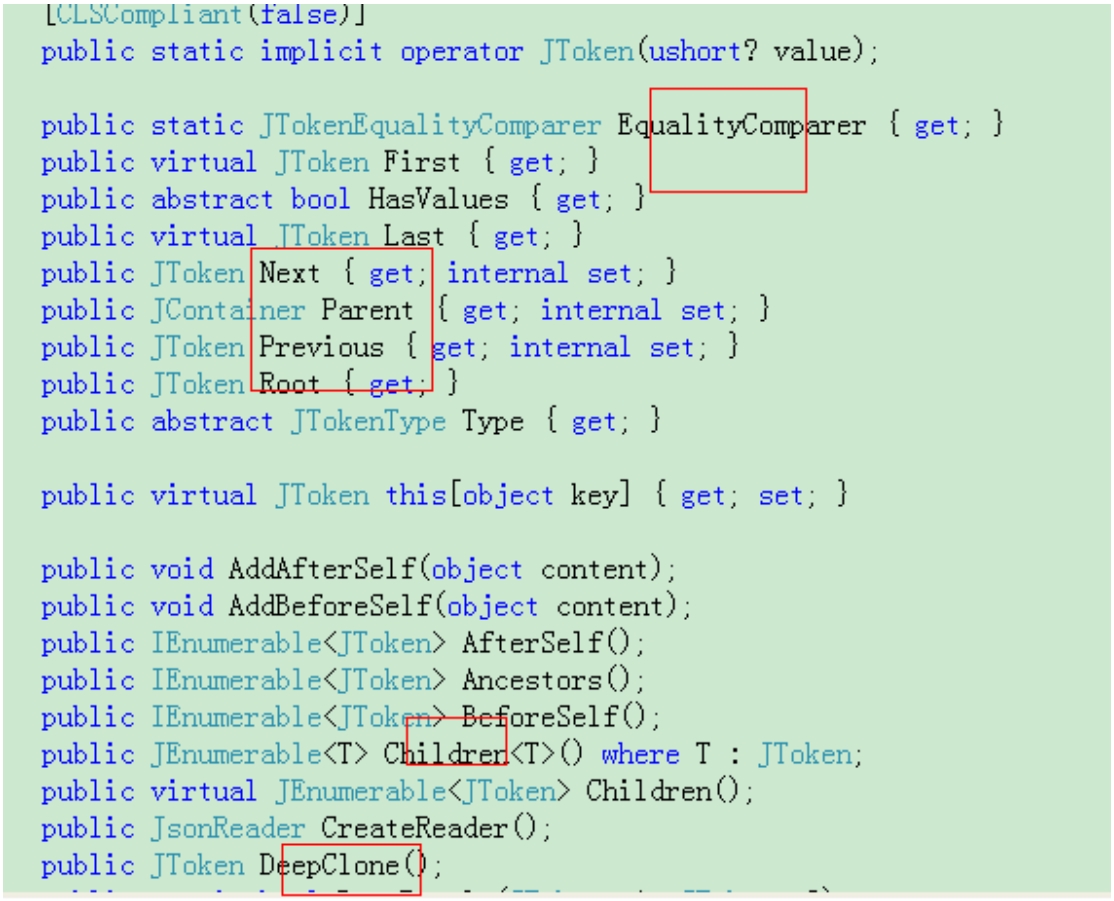
```
public class JArray : JContainer, IList<JToken>, ICollection<JToken>, IEnumerable<JToken>, IEnumerable
{
    public JArray();
    public JArray(JArray other);
    public JArray(object content);
    public JArray(params object[] content);

    protected override IList<JToken> ChildrenTokens { get; }
    public override JTokenType Type { get; }

    public JToken this[int index] { get; set; }
    public override JToken this[object key] { get; set; }

    public void Add(JToken item);
    public void Clear();
    public bool Contains(JToken item);
    public static JArray FromObject(object o);
    public static JArray FromObject(object o, JsonSerializer jsonSerializer);
    public int IndexOf(JToken item);
    public void Insert(int index, JToken item);
```

大家看下，有我们 C#数组中常用的一些方法，比如 indexof，add,remove,contains.等等

以上三种类型定义，尤其是相关接口的实现，直接为 Linq to Json 的实现提供了支持。另外做额外说明

JToken 类型内部的数据结构

是一个链式树，看 dll 中定义如下



```
[CLSCompliant(false)]
public static implicit operator JToken(ushort? value);

public static JTokenEqualityComparer EqualityComparer { get; }
public virtual JToken First { get; }
public abstract bool HasValues { get; }
public virtual JToken Last { get; }
public JToken Next { get; internal set; }
public JContainer Parent { get; internal set; }
public JToken Previous { get; internal set; }
public JToken Root { get; }
public abstract JTokenType Type { get; }

public virtual JToken this[object key] { get; set; }

public void AddAfterSelf(object content);
public void AddBeforeSelf(object content);
public IEnumerable<JToken> AfterSelf();
public IEnumerable<JToken> Ancestors();
public IEnumerable<JToken> BeforeSelf();
public IEnumerable<T> Children<T>() where T : JToken;
public virtual IEnumerable<JToken> Children();
public JsonReader CreateReader();
public JToken DeepClone();
```

好了，几种对象都说完了，该说 Linq to Json 的使用了，其实很简单。跟我们日常用 Linq

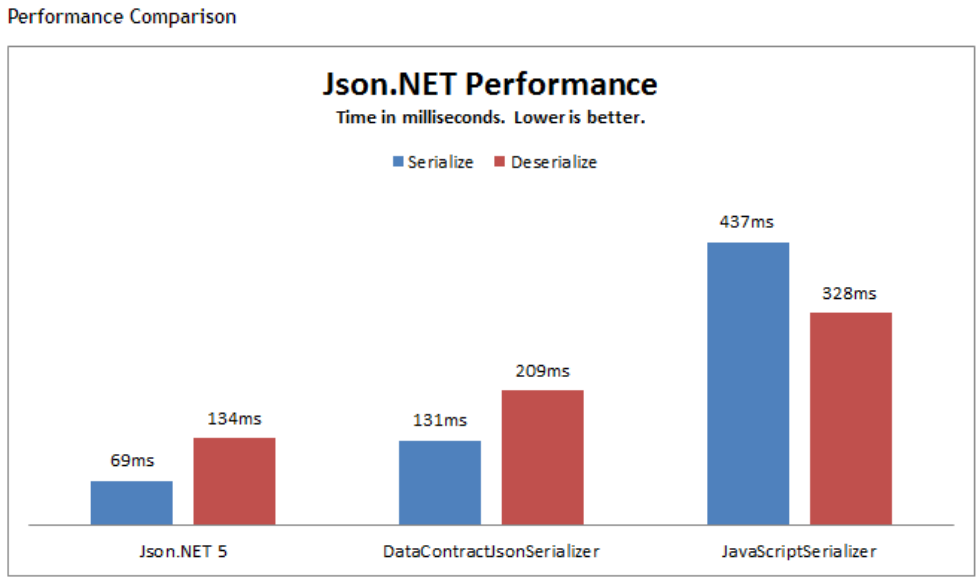
一样，只不过，我们这里的使用。遍历的单元类型变成 JToken 类型，看代码

```
/// <summary>
/// 获取包车线路的分页数据
/// Linq to Json 获取分页数据
/// by ys on 2013-9-10
/// </summary>
/// <param name="totalDatas">首次请求时加载的所有线路数据</param>
/// <param name="pagedSize"></param>
/// <param name="pagedIndex"></param>
/// <returns></returns>
private JToken[] GetPageDatas(JArray totalDatas, int pagedSize, int pagedIndex)
{
    if (pagedIndex <= 0)
    {
        pageIndex = 1;
        pagedIndex = 1;
    }
    int skipDatasCount = (pagedIndex - 1) * pagedSize;
    var pageDatas = totalDatas.Skip(skipDatasCount).Take(pagedSize).ToArray();
    return pageDatas;
}
```

使用 Linq 前提是实现了 IEnumerable<T>枚举器接口，所以我们使用 Linq to json 的时候，确保遍历的数据集合是 JArray（JToken）类型

大家看见我这个。直接 Linq to json 获取分页数据，别的什么 first(),select ,where 等等 ，都跟我们日常中用的一样。不多说。

到末尾了，貌似忘了什么东西，对，那就就是 json.net 性能优势。光说不管用，得有一些图或者说来来说明。好，看下图



这是微软的两种序列化和反序列化工具类和 Json.net 性能 比较结果图。Json.NET 有绝对的优势。

其实想来也是必然的。Json.net 的基元素类型是 JToken，数据结构是链式树，看样子是二叉链式树。哇擦。。。二叉链式树，这种树结构，在检索和添加元素的时候，比普通线

性数组或者线性表结构快很多很多。性能是很高的。兴许这就是微软也引入他的组件的原因把

好了，到这里要说的内容差不多说完了，没多少东西。
Json.NET 官网自己去百度。如果你连百度都懒得去搞，那真是浪费你的时间看这篇分享了。
周末愉快。
2013-9-15