**C# JSON.NET**

JSON String - 通用型 JSON object - JObject

**JObject :**

**1）**

转换的方法：

JsonConvert.DeserializeObject<JObject> - 此方法可以转换成其他的 C# Class Entity Object, 不仅仅是 JObject

JObject.Parse - 只能返回通用型等的 JObject

string js = "{\"full name\": \"william liu\", age: 99, house: [10,20,30,40]}";

JObject jo1 = JsonConvert.DeserializeObject<JObject>(js);

JObject jo2 = JObject.Parse(js);

MessageBox.Show(jo1.ToString());

MessageBox.Show(jo2.Value<string>("full name") );

也可以如下：

string js = @"{""full name"": ""william liu"", age: 99, house: [10,20,30,40]}";

**2）**

string js = string.Format( "{{\"full name\": \"{0}\", age: {1}, house: [10,20,30,40]}}",

"William Liu",

99);

注意： {{ }} 用于JSON 字符串的开始和结束。 同上面的有点不同

\" \" 表示 "

字符串占位符： {0} {1} 和原来的C# 是一样的

**3）**

string js = string.Format(@"{{""full name"": ""{0}"", age: {1}, house: [10,20,30,40]}}",

"William Liu",

99);

注意： {{ }} 用于JSON 字符串的开始和结束。

"" "" 表示 " , 用两个表示一个

字符串占位符： {0} {1} 和原来的C# 是一样的

**4）**

string js = string.Format(@"{{""full name"": ""{0}"", age: {1},

path: ""C:\\temp\\dir\\lwh.jpg"",

house: [10,20,30,40]}}",

"William Liu", 99);

JObject jo2 = JObject.Parse(js);

MessageBox.Show(jo2.Value<string>("path") ); 输出是： c:\temp\dir\lwh.jpg

注意： 路径使用 \\ 转义字符 \

**5）**

string js = string.Format("{{\"full name\": \"{0}\", age: {1},

path: \"C:\\\\temp\\\\dir\\\\lwh.jpg\",

house: [10,20,30,40]}}",

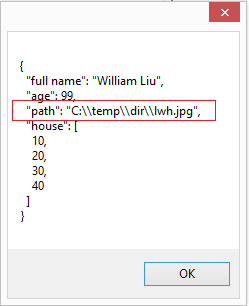
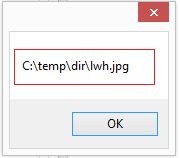
"William Liu", 99);

JObject jo2 = JObject.Parse(js);

MessageBox.Show(jo2.Value<string>("path") ); 输出是： c:\temp\dir\lwh.jpg

注意： 因为没有 @， 字符串里的路径本来就需要 两个 \\ , 这样就得需要4个 \\\\

所以在最终输出时： \\\\ 转义为 \

\\\\ 到 JObject 的path 字符值里是 \\

当我们取 path 字符使用时，则再转义 \

**6）复杂类型的 JObject**

string js = string.Format(@"{{""full name"": ""{0}"",

age: {1},

path: ""C:\\temp\\dir\\lwh.jpg"",

house: [10,20,30,40],

features: [

{{fname:""swin"", score:95}},

{{fname:""ski"", score:86}}

],

date: ""2014-12-25""

}}",

"William Liu", 99);

JObject jo1 = JsonConvert.DeserializeObject<JObject>(js);

JObject jo2 = JObject.Parse(js);

MessageBox.Show(jo1.ToString());

MessageBox.Show(jo2.Value<JArray>("features")[0].Value<string>("fname") );

MessageBox.Show(jo2.Value<DateTime>("date").ToLocalTime().ToString() );

**7）通用型 JObject 类的主要方法：**

JObject.Parse

JObject**.**Value<T>("property\_name") - return T value

jo2.Value<JArray>("features")[0].Value<string>("fname")

jo2.Value<DateTime>("date").ToLocalTime().ToString()

**8）常见错误： 不能把 JArray 当成 JObject 来使用：**

string js = @"[{""first\_name"":""hello""},{""first\_name"":""world""}]";

JObject jo1 = JsonConvert.DeserializeObject<JObject>(js);

JObject jo2 = JObject.Parse(js);

以上将出错

**9）单引号的JSON字符串使用**

string js = "{'full name': 'william liu', age: 99, house: [10,20,30,40]}";

JObject jo1 = JsonConvert.DeserializeObject<JObject>(js);

JObject jo2 = JObject.Parse(js);

MessageBox.Show(jo1.ToString());

MessageBox.Show(jo2.Value<string>("full name"));

string js = @"{'full name': 'william liu', age: 99, house: [10,20,30,40]}";

string js = @"{'full name': 'c:\\temp\\dir\\lwh.jpg', age: 99, house: [10,20,30,40]}";

string js = string.Format(@"{{'full name': '{0}',

age: {1},

path: 'C:\\temp\\dir\\lwh.jpg',

house: [10,20,30,40],

features: [

{{'fname':'swin', score:95}},

{{fname:'skoi', score:86}}

],

date: ""2014-12-25""

}}",

@"c:\\hello\\word\\lwh.jpg", 99);

JObject jo1 = JsonConvert.DeserializeObject<JObject>(js);

JObject jo2 = JObject.Parse(js);

MessageBox.Show(jo1.ToString());

MessageBox.Show(jo2.Value<string>("full name"));

解析一下：

@"c:\\hello\\word\\lwh.jpg" - 为何有 @ 还要使用 \\， 如果只是C#则是没问题，

现在是为了构造 JSON 字符串， 进入字符串是 \\, 等字符串被转化成 C# 的 JObject 后 \\ 变回 \

string js = string.Format("{{'full name': '{0}', age: {1}, path: 'C:\\\\temp\\\\dir\\\\lwh.jpg'}}",

@"c:\\hello\\word\\lwh.jpg", 99);

//string js = "[{\'first\_name\':\'C:\\temp\\dir\\lwh.jpg\'},{\'first\_name\':\'world\'}]";

JObject jo1 = JsonConvert.DeserializeObject<JObject>(js);

JObject jo2 = JObject.Parse(js);

MessageBox.Show(jo1.ToString());

MessageBox.Show(jo2.Value<string>("full name"));

解析一下：

path: 'C:\\\\temp\\\\dir\\\\lwh.jpg' - 为何用四个，而不是两个？ 因为四个变两个（字符转义）， 两个变一个（变成JObject 以后, \\ 再转义成 \

@"c:\\hello\\word\\lwh.jpg" - 有 @ 所以 JSON 字符串里是 \\ , 等变成 JObject 以后转义成 \

**JArray：**

**1）**

string js = @"[{""first\_name"":""hello""},{""first\_name"":""world""}]";

JArray jo1 = JsonConvert.DeserializeObject<JArray>(js);

JArray jo2 = JArray.Parse(js);

MessageBox.Show(jo1.ToString());

MessageBox.Show(jo2.FirstOrDefault().Value<string>("first\_name"));

string js = string.Format(@"{{ name: 'william liu',

**age: 99,**

house: [10,20,30,40],

features: [

{{fname:'{0}', score:95}},

{{fname:'ski', score:86}}

]

}}",

@"C:\\temp\\dir\\lwh.jpg");

JObject jo2 = JObject.Parse(js);

JArray ja = jo2.Value<JArray>("features");

MessageBox.Show(ja[0].Value<string>("fname"));

/////////////////////////////////////////////////////////////

JArray ja = jo2.Value<JArray>("features");

string str = string.Empty;

foreach (JObject el in ja)

{

str += string.Format("\nFname: {0} - score: {1}",

el["fname"].Value<string>(),

el["score"].Value<int>()

);

}

MessageBox.Show(str);

List<student> sts = new List<student> {

new student(){st\_name="William Liu",

st\_id = 10, st\_age = 41,

st\_birth = DateTime.Parse("1983-12-25") },

new student(){st\_name="Susan Sun",

st\_id = 20, st\_age = 26,

st\_birth = DateTime.Parse("1996-6-17")},

new student(){st\_name="Thomas Yang",

st\_id =30, st\_age = 18,

st\_birth = DateTime.Parse("2012-3-8")}

};

string ja1 = JArray.FromObject(sts).ToString();

从 C# Object -> JSON 的方法：

public class student

{

[JsonProperty(PropertyName = "Student Name", Order = 2, Required = Required.Default)]

[DefaultValue("Hello world")]

public string st\_name{ get; set; }

[JsonProperty(PropertyName="Student Age", Order = 1, Required = Required.Default)]

public int? st\_age{get; set;}

public bool st\_gender { get; set; }

public int st\_id { get; set; }

[JsonIgnore]

public string st\_birth { get; set; }

}

[JsonProperty] 是主要用于设置 JSON Object 的映射关系，以及约束关系

PropertyName - 不区分大小写 student\_name , Student Name, STUDENT NAME 是一样的

如果在JSON String 里出现同一个 属性名称多次， 则取最后一个为准，其他忽略。

string ja2 = "{'STUDENT NAME':'Tome11', 'Student Name':'Tome22', 'student name':'Tome33', 'student Age': 38, st\_birth: '2010-12-24', features:[10,20,30,40]}";

student st11 = JsonConvert.DeserializeObject<student>(ja2);

MessageBox.Show(st11.st\_name.ToString());

这里是 输出 "Tome33"

[JsonIgnore] - 指定了这个属性的， 则不参与JSON Serialize 和 Deserialize

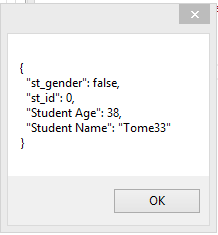
除此之外的其他属性如果有 PropertyName ， JSON string 必须有与之对应的关键字(属性）

如果有其他属性没有设置 PropertyName， 则使用原名

string ja2 = "{'STUDENT NAME':'Tome11', 'Student Name':'Tome22', 'student name':'Tome33', 'student Age': 38, st\_birth: '2010-12-24', features:[10,20,30,40]}";

student st11 = JsonConvert.DeserializeObject<student>(ja2);

MessageBox.Show( JObject.FromObject(st11).ToString(Formatting.Indented) );

 st\_gender 并没有出现在 JSON 里，所以取默认值是 false

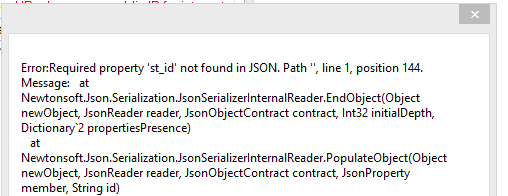
st\_id 也没有出现， 所以默认值是 0

[JsonProperty(Required=Required.Default)] - 同上一样， 可以不出现在JSON string

public int st\_id { get; set; }

[JsonProperty(Required=Required.AllowNull)] - 必须要求出现在 JSON String 里 ， 否则出错

public int st\_id { get; set; }



[JsonProperty(Required=Required.AllowNull)]

- 必须要求出现在 JSON String 里 ， 否则出错

- 虽然允许 null 空值, 但是也要看类型是否支持 null , 如 class, string 支持， int, double, DateTime 不支持。

- string? System.Nullable<string> 是不允许的，因为本身支持 null 赋值

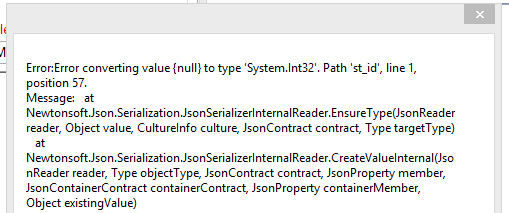
public int st\_id { get; set; }

我们指定 null, 注意是 null , 不能是 Null, NULL 否则出错

string ja2 = "{'student name':'Tome33', 'student Age': 38, st\_id: null, st\_birth: '2010-12-24', features:[10,20,30,40]}";

student st11 = JsonConvert.DeserializeObject<student>(ja2);

MessageBox.Show( JObject.FromObject(st11).ToString(Formatting.Indented) );



解决方案：

1) System.Nullable<type> 或者 Type? 等同的语法

public class student

{

[JsonProperty(PropertyName = "Student Name", Order = 2, Required = Required.Default)]

[DefaultValue("Hello world")]

public string st\_name{ get; set; }

[JsonProperty(PropertyName="Student Age", Order = 1, Required = Required.Default)]

public int? st\_age{get; set;}

public bool st\_gender { get; set; }

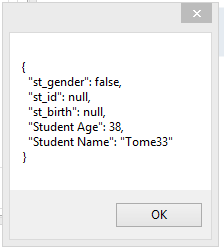
[JsonProperty(Required=Required.AllowNull)]

public int? st\_id { get; set; }

public System.Nullable<DateTime> st\_birth { get; set; }

}

string ja2 = "{'student name':'Tome33', 'student Age': 38, st\_id: null, features:[10,20,30,40]}";

student st11 = JsonConvert.DeserializeObject<student>(ja2);

MessageBox.Show( JObject.FromObject(st11).ToString(Formatting.Indented) );

为预防运行时出错：处理 System.Nullable<Type > 类型，应该先要安全判断：

if (!st11.st\_age.HasValue) MessageBox.Show("St age has no value null");

if (!st11.st\_id.HasValue) MessageBox.Show("St id has no value null");

关于 null 值的几点说明：

string ja2 = "{'student name': '', 'student Age': 38, st\_id: '', features:[10,20,30,40]}";

student st11 = JsonConvert.DeserializeObject<student>(ja2);

MessageBox.Show( JObject.FromObject(st11).ToString(Formatting.Indented) );

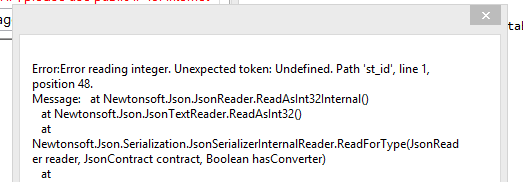
'student name' : '' 注意此值不是 null 值 string.empty 相等

'st\_id': '' 注意此值付给 int, DateTime 就是 null 值

非法的JSON String:

string ja2 = "{'student name': '', 'student Age': 38, st\_id: , st\_birth: , features:[10,20,30,40]}";

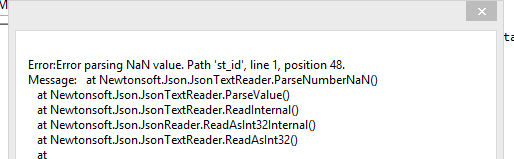
不能 , st\_id: , st\_birth: ,



string ja2 = "{'student name': '', 'student Age': 38, st\_id: Null, st\_birth: NULL, features:[10,20,30,40]}";

不能 st\_id: Null, st\_birth: NULL, 只有小写的 null 唯一代表 null

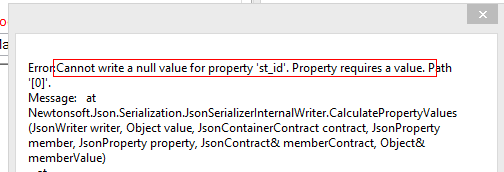
不能st\_id: 'null', st\_birth: 'null', 放到字符串里，也不行



[JsonProperty(Required=Required.Always)] - 虽然 int? 允许赋予 null 值, 但是强制需要赋予值

public int? st\_id { get; set; }

Always - 必须赋予非 null 值。



复合类型的 C# Class

public class student

{

[JsonProperty(PropertyName = "Student Name", Order = 2, Required = Required.Default)]

[DefaultValue("Hello world")]

public string st\_name{ get; set; }

[JsonProperty(PropertyName="Student Age", Order = 1, Required = Required.Default)]

public int? st\_age{get; set;}

public bool st\_gender { get; set; }

[JsonProperty(Required=Required.Always)]

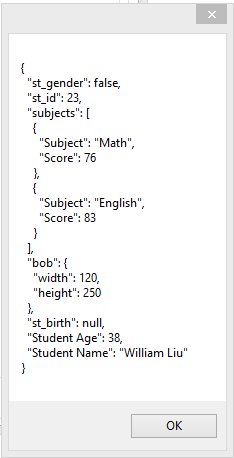
public int? st\_id { get; set; }

[JsonProperty(Required = Required.Always)] - 复合类型不能有 JsonProperty

public IEnumerable<subject> subjects { get; set; }

[JsonProperty(Required = Required.Always)] - 复合类型不能有 JsonProperty

public boby bob { get; set; }

 public System.Nullable<DateTime> st\_birth { get; set; }

}

public class subject

{

[JsonProperty(PropertyName="Subject")]

public string sb\_name { get; set; }

[JsonProperty(PropertyName = "Score")]

public int sb\_score { get; set; }

}

public class boby

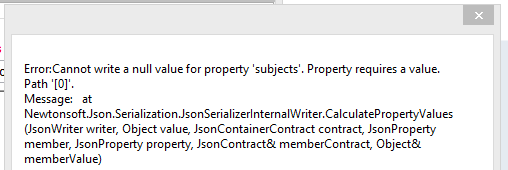
{

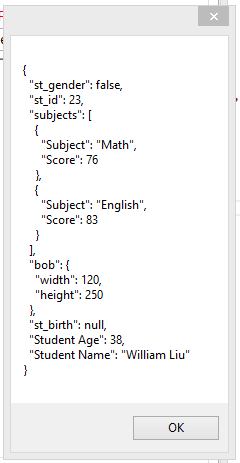
public int width;

public int height;

}

复合类型不能有 JsonProperty 错误信息如下：



JSON String to JSON Object

1. JsonConvert.DeserializeObject<JObject>

string ja2 = "{'student name': 'William Liu', 'student Age':38, st\_id:23, st\_birth:null, subjects:[{subject:'Math',score:76},{subject:'English',score:83}], bob:{width:120, height:250}, features:[10,20,30,40]}";

JObject st11 = JsonConvert.DeserializeObject<JObject>(ja2);

MessageBox.Show(st11.ToString(Formatting.Indented));

* 通用型：

JsonConvert.DeserializeObject<JObject>

string ja2 = "{'student name': 'William Liu', 'student Age':38, st\_id:23, st\_birth:null, subjects:[{subject:'Math',score:76},{subject:'English',score:83}], bob:{width:120, height:250}, features:[10,20,30,40]}";

student st11 = JsonConvert.DeserializeObject<student>(ja2);

MessageBox.Show(JObject.FromObject(st11).ToString());

* 指定具体 C# Object

JsonConvert.DeserializeObject<student>(ja2);

也支持泛型类

1. JsonSerializer: 优点是从 TextReader(StringReader), JsonReader 来获取源串

支持泛型类: T tobj = JsonSerializer.Deserialize<T>( JsonReader )

string ja2 = "{'student name': 'William Liu', 'student Age':38, st\_id:23, st\_birth:null, subjects:[{subject:'Math',score:76},{subject:'English',score:83}], bob:{width:120, height:250}, features:[10,20,30,40]}";

StringReader sr = new StringReader(ja2);

JsonReader jreader = new JsonTextReader(sr);

JsonSerializer jss = new JsonSerializer();

student st22 = jss.Deserialize<student>(jreader);

MessageBox.Show(JObject.FromObject(st22).ToString(Formatting.Indented));

所以可以很方便从文件读取源串：

StreamReader smreader = File.OpenText(@"c:\fm\st.json");

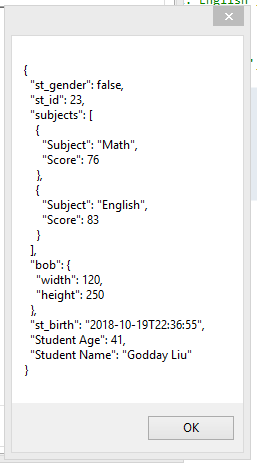
//StringReader sr = new StringReader(ja2);

JsonReader jreader = new JsonTextReader(smreader);

JsonSerializer jss = new JsonSerializer();

student st22 = jss.Deserialize<student>(jreader);

MessageBox.Show(JObject.FromObject(st22).ToString(Formatting.Indented));

文件内容：

{

"st\_gender": false,

"st\_id": 23,

"subjects": [

{

"Subject": "Math",

"Score": 76

},

{

"Subject": "English",

"Score": 83

}

],

"bob": {

"width": 120,

"height": 250

},

"st\_birth":"2018-10-19 10:36:55PM",

"Student Age": 41,

"Student Name": "Godday Liu"

}

写入文件: 使用流文件

StreamWriter sw = File.CreateText(@"c:\fm\stst.json");

JsonWriter jwritter = new JsonTextWriter(sw);

JsonSerializer jss1 = new JsonSerializer();

jwritter.Formatting = Formatting.Indented;

jss1.Serialize(jwritter, st22, typeof(student));

sw.Flush();

sw.Close();

1. JObject.Parse( json\_string ) - 不支持泛型类, 所以只能返回JObject

string js = "{ name:'William Liu',

age:42,

subjects:[

{subject:'Math', score:95},

{subject:'English',score:86}],

profile:{tall:173, salary:65, class:'VIP'}

}";

JObject jo = JObject.Parse(js);

this.richTextBox1.Text = jo.ToString(Formatting.Indented);

1. JObject.Load(JsonReader) – 方便从流文件读取, 不支持泛型类, 所以只能返回JObject

StringReader sreader = new StringReader(js);

JsonReader jreader = new JsonTextReader(sreader);

JObject jo = JObject.Load(jreader);

1. JObject.ReadFrom(JsonReader) - JToken .ToObject<JObject>() 转化JObject

StringReader sreader = new StringReader(js);

JsonReader jreader = new JsonTextReader(sreader);

JObject jo = JObject.ReadFrom(jreader).ToObject<JObject>();

1. JObject.ToObject<T>() – 转化成其他类型;

public class student

{

[JsonProperty(PropertyName="name")]

public string sname;

public int age;

public IList<subject> subjects;

}

public class subject

{

[JsonProperty(PropertyName="subject")]

public string sub;

[JsonProperty(PropertyName = "score")]

public int sco;

}

string js = "{ name:'William Liu',

age:42,

subjects:[{subject:'Math', score:95}, {subject:'English', score:86}], profile:{tall:173, salary:65, class:'VIP'}

}";

StringReader sreader = new StringReader(js);

JsonReader jreader = new JsonTextReader(sreader);

JObject jo = JObject.ReadFrom(jreader).ToObject<JObject>();

student stt = jo.ToObject<student>();

输出：

{

"name": "William Liu",

"age": 42,

"subjects": [

{

"subject": "Math",

"score": 95

},

{

"subject": "English",

"score": 86

}

]

}

1. JArray 的使用： Parse, Load, ReadFrom

string js = "[ {subject:'Math', score:95},

{subject:'English', score:86},

{subject:'Math', score:95},

{subject:'English', score:86}]";

JArray ja = JArray.Parse(js);

string str = string.Empty;

foreach (JObject el in ja)

{

subject el1 = el.ToObject<subject>();

str += string.Format("{0}-{1}\n", el1.sub, el1.sco);

}

this.richTextBox1.Text = str;

string js = "{logs:[{name:'America', time:200}, {name:'China',time:300}]}";

JObject jo = JObject.Parse(js);

JArray ja = jo.Value<JArray>("logs");

string str = string.Empty;

foreach (JObject el in ja) ja[0].values ... ja[n-1].values 实际上是 可以直接转为 JObject

{

str += string.Format("{0}-{1}", el.Value<string>("name"), el.Value<int>("time"));

}

MessageBox.Show(string.Format("{0}", str.ToString()));

JObject j00 = ja[0].Value<JObject>(); - 不指定 key , 则返回的是 JObject

j00.Value<string>("name")

ja[0].Value<string>("name") - 和上面的结果是一样的。

MessageBox.Show("j00 " +

j00.Value<string>("name") + " " +

ja[0].Value<string>("name") );

C# Object to JSON String:

string js = "{ name:'William Liu',

age: 36,

features:[2,3,5,1,0,2],

subjects:{ 'match':89,

'Englis':96}

}";

JObject jo = JObject.Parse(js);

jo.ToString(); 默认是 jo.ToString(Formatting.Indented);

jo.ToString(Formatting.None);

1. JArray.FromObject(object) - JArray.ToString(Formatting.Indented);

List<subject> subs = new List<subject>

{

new subject{ sco = 15, sub="MATH"} ,

new subject{ sco = 19, sub="English"}

};

JArray ja0 = JArray.FromObject(subs);

this.richTextBox1.Text = ja0.ToString(Formatting.Indented);

[

{

"subject": "MATH",

"score": 15

},

{

"subject": "English",

"score": 19

}

]

1. JObject.FromObject(object) - JObject.ToString(Formatting.Indented);

subject sub = new subject { sco = 29, sub = "English" };

JObject jo = JObject.FromObject(sub);

this.richTextBox1.Text = jo.ToString(Formatting.Indented);

1. JsonConvert.SerializeObject(object,Formatting.Indented);

subject sub = new subject { sco = 29, sub = "English" };

string js = JsonConvert.SerializeObject(sub,Formatting.Indented);

this.richTextBox1.Text = js;

1. Create JObject

JObject jo = new JObject

{

new JProperty("name", "William Liu"),

new JProperty("age", 45),

new JProperty("score",new int[]{10,20,30,40}), - 注意这是普通的 int array,

new JProperty("hhh", new JArray( - 注意 JArray 是属于 JProperty 的值, 注意这是JObject array

new JObject{new JProperty("sub","math")},

new JObject{new JProperty("sub","english")}

)

),

new JProperty("hao", new JRaw( new string[] {"hello", "world"} ) )

};

输出:

{

"name": "William Liu",

"age": 45,

"score": [

10,

20,

30,

40

],

"hhh": [

{

"sub": "math"

},

{

"sub": "english"

}

],

"hao": System.String[] -- JRaw 输出引用?

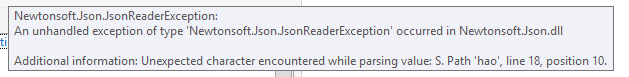
}

this.richTextBox1.Text = jo.Value<string[]>("hao")[0]; 如此访问到 “hello”

使用上面的 JSON string 生成另外一个 JObject 将出错:

因为: "hao": System.String[] 不是合法的 JSON String

JObject jo1 = JObject.Parse(jo.ToString());



JSON 的常见错误：

记住 JSON object 是 { key : value } 键值对

基于此原理：代码生成 JObject 里只包含 JProperty

1. 没有关键字的JSON

string js = "{name:'William Liu', age: 36, features:[2,3,5,1,0,2], {'match':89, 'Englis':96} }";

JObject jo = JObject.Parse(js);

{'match':89, 'Englis':96} - 没有属性关键字以之对应， 有值无键，出错

类似以下代码

JProperty jname = new JProperty("name", "William LIU");

JObject jo = new JObject

{

new JProperty(jname),

new JProperty("features", new JArray(new List<int>{100, 200,300})),

new JObject(new JProperty("Math", 89), new JProperty("English", 87) ) - 有值无键， 出错

};

正确的使用：

JProperty jname = new JProperty("name", "William LIU");

JObject jo = new JObject

{

new JProperty(jname),

new JProperty("features", new JArray(new List<int>{100, 200,300})),

new JProperty("Subjects", new JObject(new JProperty("Math", 89), new JProperty("English", 87) ))

};

string js = "{ name:'William Liu',

age: 36,

features:[2,3,5,1,0,2],

subjects: {'match':89, 'Englis':96}

}";

JObject jo = JObject.Parse(js);

2. null 是必须全小写，Null, NULL , "null" 都不代表 null

Null, NULL 导致parse 出错

3. Object = null

boby bb = null;

JObject jo = new JObject

{

new JProperty(jname),

new JProperty("features", new JArray(new List<int>{100, 200,300})),

new JProperty("Subjects", new JObject(new JProperty("Math", 89), new JProperty("English", 87) )),

new JProperty("object", bb )

};

{

"name": "William LIU",

"features": [

100,

200,

300

],

"Subjects": {

"Math": 89,

"English": 87

},

"object": null

}

4. 特殊类型只有Javascript有： undefined

string js = "{name: , age: null, flag:'null', features:[2,3,5,1,0,2], subjects: {'match':89, 'Englis':96} }";

JObject jo = JObject.Parse(js);

{

"name": undefined,

"age": null,

"flag": "null",

"features": [

2,

3,

5,

1,

0,

2

],

"subjects": {

"match": 89,

"Englis": 96

}

}

public class subject

{

public string name { get; set; }

public int? age { get; set; }

}

subject sbb = jo.ToObject<subject>(); - 无法转化而出错。

如何安全处理：

C# 可以处理 System.Nullable<T> T? 类型， 所以 Javascript 要保证JSON不出现 undefined

string js = "{name: , age: undefined , flag:'null', features:[2,3,5,1,0,2], subjects: {'match':89, 'Englis':96} }";

JObject jo = JObject.Parse(js);

if (!jo.Value<int?>("name").HasValue) MessageBox.Show("Int is Null");

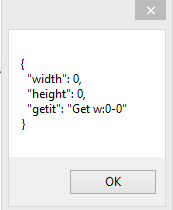
if (string.IsNullOrEmpty(jo.Value<string>("name"))) MessageBox.Show("String is Null");

5. JObject From Ojbect:

并非所有 Object 都支持

DateTime dt1 = MMDate.jstime(1418962526);

JObject jo1 = JObject.FromObject(dt1); - 不支持

public struct boby 属性与字段是键值， 方法不是，也不返回任何值

{

public int width;

public int height;

public string showit()

{

return string.Format("w:{0}-{1}", width, height);

}

public string getit

{

get

{

return string.Format("Get w:{0}-{1}", width, height);

}

}

}

boby dt1 = new boby();

JObject jo1 = JObject.FromObject(dt1);

5. JArray from string 必须是 Array [ ]

string js = "[{name:, age:undefined, flag:'null', features:[2,3,5,1,0,2], subjects: {'match':89, 'Englis':96} }]";

JArray ja = JArray.Parse(js);

ja.ToString(); 注意 undefined

[

{

"name": undefined,

"age": undefined,

"flag": "null",

"features": [

2,

3,

5,

1,

0,

2

],

"subjects": {

"match": 89,

"Englis": 96

}

}

]

public static string ListToString<T>(IList<T> tobj)

{

JArray ja = JArray.FromObject(tobj);

return ja.ToString(Formatting.None);

}

MMJson.ListToString(ja);

放回安全的类型： null

[{"name":null,"age":null,"flag":"null","features":[2,3,5,1,0,2],"subjects":{"match":89,"Englis":96}}]

5. JArray 可以是C# List 类型 必须是 Array [ ]

string js = "[10,20,30]";

JArray ja = JArray.Parse(js);

ja[0].Value<int>().ToString(); - 返回 10

string js = "[10,20,30]";

JArray ja = JArray.Parse(js);

int[] ia = ja.Value<int[]>(); 出错

{"Cannot cast Newtonsoft.Json.Linq.JArray to Newtonsoft.Json.Linq.JToken."}

string js = "[10,20,30]";

JArray ja = JArray.Parse(js);

int[] ia = ja.ToObject<int[]>(); - 正确

ja.Value<int[]>() 与 ja.ToObject<int[]>(); 的区别：

ToObject 用于转化类型：

Value 用于获取值

string js = "[{'name':'Will', age:34, ts:[10,20,30]}, {'name':'Tom', age: 43, tu:[100,220,320]}]";

JArray ja = JArray.Parse(js);

JObject st = ja[0].Value<JObject>(); - 正确

this.rch\_body.Text = string.Join(",", st.Value<string>("name") );

string js = "[{'name':'Will', age:34, ts:[10,20,30]}, {'name':'Tom', age: 43, tu:[100,220,320]}]";

JArray ja = JArray.Parse(js);

JObject st = ja[0].Value<JObject>();

int[] sta = st.Value<int[]>("ts"); - 出错， JArray 则正确

容易犯的错误： [] 不能带 "" 引号括住， 否则成了字符串，而不是 [] Object

string js = "[{'name':'Will', age:34, ts:'[10,20,30]'}, {'name':'Tom', age: 43, tu:[100,220,320]}]";

JArray ja = JArray.Parse(js);

JObject st = ja[0].Value<JObject>();

JArray sta = st.Value<JArray>("ts"); - 出错， 是字符串

JObject sta = st.Value<JObject>("ts"); - 仍然出错， 是字符串，而不是JObject

string sta = st.Value<string>("ts"); - 正确： 输出 字符串 "[10,20,30]"

字符型型的 Array 可以使用JArray 快速转化回 C# 的Object

string js11 = "[10,20,30,40]";

JArray ja11 = JArray.Parse(js11);

int[] iam = ja11.ToObject<int[]>();

转成 List<JObject>

string js = "[{'name':'Will', age:34, ts:'[10,20,30]'}, {'name':'Tom', age: 43, tu:[100,220,320]}]";

JArray ja = JArray.Parse(js);

List<JObject> st = ja.ToObject<List<JObject>>();

string js = "[{'name':'Will', age:34, ts:[10,20,30], ss:{math:95} }, {'name':'Tom', age: 43, tu:[100,220,320]}]";

JArray ja = JArray.Parse(js);

JObject jo = ja[0].Value<JObject>("ss"); - 正确

this.rch\_body.Text = jo.Value<string>("math"); - 正确

string js = "[{'name':'Will', age:34, ts:[10,20,30], ss:{math:95} }, {'name':'Tom', age: 43, tu:[100,220,320]}]";

JArray ja = JArray.Parse(js);

List<subject> jo = ja.ToObject<List<subject>>(); - 可以 JArray 转化成 List<Object>

this.rch\_body.Text = jo[0].name;

System.Nullable<Type> = Type? 使用

public class boby

{

public int width;

public int height;

}

public class test

{

public string name; - string 本身支持null

public int score;

public int? age;

public System.Nullable<DateTime> dt;

public boby bbb;

}

String:

string ss; 此时并非是 null 值

if (ss == null) MessageBox.Show("string is null"); - 未赋值使用， 出现错误

string ss = null;

if (ss == null) MessageBox.Show("string is null"); true

if (string.IsNullOrEmpty(ss)) MessageBox.Show("string is null or empty"); true

if (ss == string.Empty) MessageBox.Show("string is empty"); false

test t1 = new test();

if (t1.name == null) MessageBox.Show("string name is null"); true

if (string.IsNullOrEmpty(t1.name)) MessageBox.Show("string name is null or empty"); true

if (t1.name == string.Empty) MessageBox.Show("string name is empty"); false

if (t1.bbb == null) MessageBox.Show("boby is null"); true

if (!t1.age.HasValue) MessageBox.Show("age.HasValue is false"); true

if (t1.age == null) MessageBox.Show("age is null"); true

if (!t1.dt.HasValue) MessageBox.Show(string.Format("date.HasValue is null: {0}", t1.dt)); true

if (t1.dt == null) MessageBox.Show("datetime is null"); true

MessageBox.Show(string.Format("date: \*{0}\*", t1.dt.Value)); t1.dt.Value 出错

关于运算：

**1） Int = Int? + 100; 出错. 正确做法**： Int = ( Int?.HasValue? Int?.Value : 0 ) + 100;

test t1 = new test();

t1.age = 29; 虽然给 int? 赋值了

t1.score = t1.score + t1.age; int? 无法转化成 int



解决方法：

test t1 = new test() { };

t1.age = 29;

t1.score = t1.score + t1.age.Value; - 有缺陷

t1.score = t1.score + Convert.ToInt32(t1.age); - 此法安全

test t1 = new test() { };

t1.score = t1.score + t1.age.Value; 有缺陷,值为null时，出错 

完美的方案：

t1.score = t1.score + (t1.age.HasValue? t1.age.Value : 100);

**2）Int? = Int? + 100; result int? = (null / Int)**

test t1 = new test() { };

t1.age = 5;

t1.age = t1.age + 30; = 35

test t1 = new test() { };

t1.age = t1.age + 30; = null

**3）Int? = Int +　100; result int? = (Int)**

test t1 = new test() { };

t1.age = t1.score + 30; // t.score default = 0; t1.age = 30;

注意：通常ADO.Net Entity Framework Entity Object 从数据库映射过来，如果数据库的字段 Allow Null. 则自动生成的 ORM Object 的字段对应的类型是 System.Nullable<T> T?

使用时注意 T 和 T? 是不同类型，注意转化。

动态变量也可以转化为 JArray / JObject

var a = new List<object>

{

new {name="AA", age=10},

new {name="BB", age=20},

new {name="CC", age=30}

};

JArray ja = JArray.FromObject(a);

MessageBox.Show(ja.ToString());

var b = new { name = "William", age = 40, edate = DateTime.Now };

MessageBox.Show(b.name + ":" + b.age.ToString() + ":" + b.edate.ToString());

JObject jo = JObject.FromObject(b);

MessageBox.Show(jo.ToString());

**[C#解析json文件的方法](http://www.cnblogs.com/txw1958/archive/2012/08/01/csharp-json.html)**

**C#** **解析 json**

　　JSON(全称为JavaScript Object Notation) 是一种轻量级的数据交换格式。它是基于JavaScript语法标准的一个子集。 JSON采用完全独立于语言的文本格式，可以很容易在各种网络、平台和程序之间传输。JSON的语法很简单，易于人阅读和编写，同时也易于机器解析和生 成。

**JSON与XML的比较**◆可读性  
　　JSON和XML的可读性相比较而言，由于XML提供辅助的标签，更加适合人阅读和理解。  
◆文件大小与传输  
　　XML允许使用方便的标签，所以文件尺寸是要比JSON大的。而且JSON源于Javascript，所以天生的主战场是Javascript与网络，在这里，JSON有着XML无法赶超的优势。

**JSON语法**1. JSON 语法是 JavaScript 对象表示法语法的子集。

* 数据在名称/值对中：名称是字符串，使用双引号表示。值可以是：数字（整数或浮点数），字符串（在双引号中）,数组（在方括号中）,对象（在花括号中），true/false/null。
* 数据由逗号分隔：
* 花括号保存对象：对象可以包含各种数据，包括数组。
* 方括号保存数组：数字可以包含对象。

例如：

[复制代码](javascript:void(0);)

{  
    "employees": [  
        {  
            "firstName": "Bill",  
            "lastName": "Gates"  
        },  
        {  
            "firstName": "George",  
            "lastName": "Bush"  
        }  
    ]  
}

[复制代码](javascript:void(0);)

2. 如果JSON中含有转义字符，则需要转义。例如文件路径中需要使用"\\"而不是"\"。例如：{ "file":"C:\\a.txt"}。

**.NET操作JSON**  
　　JSON文件读入到内存中就是字符串，.NET操作JSON就是生成与解析JSON字符串。操作JSON通常有以下几种方式：  
**1. 原始方式**：自己按照JSON的语法格式，写代码直接操作JSON字符串。如非必要，应该很少人会走这条路，从头再来的。

**2. 通用方式【★★★★★】**：

这种方式是使用开源的类库Newtonsoft.Json(下载地址<http://json.codeplex.com/>)。下载后加入工程就能用。通常可以使用JObject, JsonReader, JsonWriter处理。这种方式最通用，也最灵活，可以随时修改不爽的地方。  
(1)使用JsonReader读Json字符串：

[复制代码](javascript:void(0);)

string jsonText = @"{""input"" : ""value"", ""output"" : ""result""}";  
JsonReader reader = new JsonTextReader(new StringReader(jsonText));  
  
while (reader.Read())  
{  
    Console.WriteLine(reader.TokenType + "\t\t" + reader.ValueType + "\t\t" + reader.Value);  
}

[复制代码](javascript:void(0);)

(2)使用JsonWriter写字符串：

[复制代码](javascript:void(0);)

StringWriter sw = new StringWriter();  
JsonWriter writer = new JsonTextWriter(sw);  
  
writer.WriteStartObject();  
writer.WritePropertyName("input");  
writer.WriteValue("value");  
writer.WritePropertyName("output");  
writer.WriteValue("result");  
writer.WriteEndObject();  
writer.Flush();  
  
string jsonText = sw.GetStringBuilder().ToString();  
Console.WriteLine(jsonText);

[复制代码](javascript:void(0);)

 (3)使用JObject读写字符串：

JObject jo = JObject.Parse(jsonText);  
string[] values = jo.Properties().Select(item => item.Value.ToString()).ToArray();

 (4)使用JsonSerializer读写对象(基于JsonWriter与JsonReader):

数组型数据

[复制代码](javascript:void(0);)

string jsonArrayText1 = "[{'a':'a1','b':'b1'},{'a':'a2','b':'b2'}]";

JArray ja = (JArray)JsonConvert.DeserializeObject(jsonArrayText1);

string ja1a = ja[1]["a"].ToString();

//或者

JObject o = (JObject)ja[1];

string oa = o["a"].ToString();

[复制代码](javascript:void(0);)

嵌套格式

string jsonText = "{\"beijing\":{\"zone\":\"海淀\",\"zone\_en\":\"haidian\"}}";

JObject jo = (JObject)JsonConvert.DeserializeObject(jsonText);

string zone = jo["beijing"]["zone"].ToString();

string zone\_en = jo["beijing"]["zone\_en"].ToString();

自定义类Project

[复制代码](javascript:void(0);)

Project p = new Project() { Input = "stone", Output = "gold" };  
JsonSerializer serializer = new JsonSerializer();  
StringWriter sw = new StringWriter();  
serializer.Serialize(new JsonTextWriter(sw), p);  
Console.WriteLine(sw.GetStringBuilder().ToString());  
  
StringReader sr = new StringReader(@"{""Input"":""stone"", ""Output"":""gold""}");  
Project p1 = (Project)serializer.Deserialize(new JsonTextReader(sr), typeof(Project));  
Console.WriteLine(p1.Input + "=>" + p1.Output);

[复制代码](javascript:void(0);)

　　上面的代码都是基于下面这个Project类定义：

class Project  
{  
    public string Input { get; set; }  
    public string Output { get; set; }  
}

　　此外，如果上面的JsonTextReader等类编译不过的话，说明是我们自己修改过的类，换成你们自己的相关类就可以了，不影响使用。

**3. 内置方式**：使用.NET Framework 3.5/4.0中提供的System.Web.Script.Serialization命名空间下的JavaScriptSerializer类进行对象的序列化与反序列化，很直接。

[复制代码](javascript:void(0);)

 Project p = new Project() { Input = "stone", Output = "gold" };  
 JavaScriptSerializer serializer = new JavaScriptSerializer();  
 var json = serializer.Serialize(p);  
 Console.WriteLine(json);  
  
 var p1 = serializer.Deserialize<Project>(json);  
 Console.WriteLine(p1.Input + "=>" + p1.Output);  
 Console.WriteLine(ReferenceEquals(p,p1));

[复制代码](javascript:void(0);)

**注意：**如果使用的是VS2010，则要求当前的工程的 Target Framework要改成.Net Framework 4，不能使用Client Profile。当然这个System.Web.Extensions.dll主要是Web使用的，直接在Console工程中用感觉有点浪费资源。  
　　此外，从最后一句也可以看到，**序列化与反序列化是深拷贝的一种典型的实现方式**。

**4. 契约方式**：使用System.Runtime.Serialization.dll提供的DataContractJsonSerializer或者 JsonReaderWriterFactory实现。

[复制代码](javascript:void(0);)

Project p = new Project() { Input = "stone", Output = "gold" };  
DataContractJsonSerializer serializer = new DataContractJsonSerializer(p.GetType());  
string jsonText;  
  
using (MemoryStream stream = new MemoryStream())  
{  
    serializer.WriteObject(stream, p);  
    jsonText = Encoding.UTF8.GetString(stream.ToArray());  
    Console.WriteLine(jsonText);  
}  
  
using (MemoryStream ms = new MemoryStream(Encoding.UTF8.GetBytes(jsonText)))  
{  
    DataContractJsonSerializer serializer1 = new DataContractJsonSerializer(typeof(Project));  
    Project p1 = (Project)serializer1.ReadObject(ms);  
    Console.WriteLine(p1.Input + "=>" + p1.Output);  
}

[复制代码](javascript:void(0);)

　　这里要注意，这里的Project类和成员要加相关的Attribute：

[复制代码](javascript:void(0);)

[DataContract]  
class Project  
{  
    [DataMember]  
    public string Input { get; set; }  
    [DataMember]  
    public string Output { get; set; }  
}

   最近在做微信开发时用到了一些json的问题，就是把微信返回回来的一些json数据做一些处理，但是之前json掌握的不好，浪费了好多时间在查找一些 json有关的转换问题，我所知道的方法只有把json序列化和反序列化一下，但是太麻烦了我觉得，所以就在找一些更简单又方便使用的方法。也许这个会有 用吧，所以先放到这以后能用到的。

原文出处：<http://www.cnblogs.com/mcgrady/archive/2013/06/08/3127781.html>

JSON的全称是”JavaScript Object Notation”，意思是JavaScript对象表示法，它是一种基于文本，独立于语言的轻量级数据交换格式。XML也是一种数据交换格式，为什么没 有选择XML呢？因为XML虽然可以作为跨平台的数据交换格式，但是在JS(JavaScript的简写)中处理XML非常不方便，同时XML标记比数据 多，增加了交换产生的流量，而JSON没有附加的任何标记，在JS中可作为对象处理，所以我们更倾向于选择JSON来交换数据。这篇文章主要从以下几个方 面来说明JSON。

1，JSON的两种结构  
2，认识JSON字符串  
3，在JS中如何使用JSON  
4，在.NET中如何使用JSON  
5，总结

## JSON的两种结构

JSON有两种表示结构，对象和数组。  
对象结构以”{”大括号开始，以”}”大括号结束。中间部分由0或多个以”，”分隔的”key(关键字)/value(值)”对构成，关键字和值之间以”：”分隔，语法结构如代码。

{

key1:value1,

key2:value2,

...

}

其中关键字是字符串，而值可以是字符串，数值，true,false,null,对象或数组

数组结构以”[”开始，”]”结束。中间由0或多个以”，”分隔的值列表组成，语法结构如代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | [      {          key1:value1,          key2:value2      },      {           key3:value3,           key4:value4      }  ] |

## 认识JSON字符串

之前我一直有个困惑，分不清普通字符串，json字符串和json对象的区别。经过一番研究终于给弄明白了。比如在js中。

字符串：这个很好解释，指使用“”双引号或’’单引号包括的字符。例如：var comStr = 'this is string';  
json字符串：指的是符合json格式要求的js字符串。例如：var jsonStr = "{StudentID:'100',Name:'tmac',Hometown:'usa'}";  
json对象：指符合json格式要求的js对象。例如：var jsonObj = { StudentID: "100", Name: "tmac", Hometown: "usa" };

## 在JS中如何使用JSON

JSON是JS的一个子集，所以可以在JS中轻松地读，写JSON。读和写JSON都有两种方法，分别是利用”.”操作符和“[key]”的方式。  
我们首先定义一个JSON对象，代码如下。

[复制代码](javascript:void(0);)

var obj = {

"1": "value1",

"2": "value2",

count: 3,

person: [ //数组结构JSON对象，可以嵌套使用

{

id: 1,

name: "张倩"

},

{

id: 2,

name: "张帅"

}

],

object: { //对象结构JSON对象

id: 1,

msg: "对象里的对象"

}

};

[复制代码](javascript:void(0);)

1，从JSON中读数据

[复制代码](javascript:void(0);)

function ReadJSON() {

alert(obj.1); //会报语法错误，可以用alert(obj["1"]);说明数字最好不要做关键字

alert(obj.2); //同上

alert(obj.person[0].name); //或者alert(obj.person[0]["name"])

alert(obj.object.msg); //或者alert(obj.object["msg"])

}

[复制代码](javascript:void(0);)

2，向JSON中写数据

比如要往JSON中增加一条数据，代码如下：

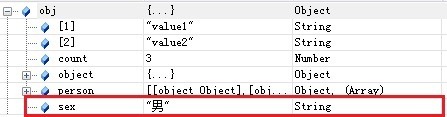
function Add() {

//往JSON对象中增加了一条记录

obj.sex= "男" //或者obj["sex"]="男"

}

增加数据后的JSON对象如图：

[](http://images.cnitblog.com/blog/311549/201306/08225647-1f3c6b7829074c529ee3365315ccc6b0.jpg)

3，修改JSON中的数据

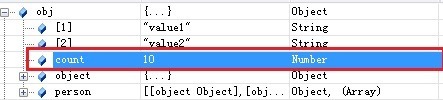
我们现在要修改JSON中count的值，代码如下：

function Update() {

obj.count = 10; //或obj["count"]=10

}

修改后的JSON如图。

[](http://images.cnitblog.com/blog/311549/201306/08225654-7373ca8222ad4b999701ce3f988d5d68.jpg)

4，删除JSON中的数据

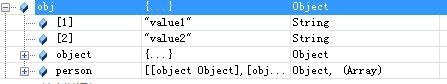
我们现在实现从JSON中删除count这条数据，代码如下：

function Delete() {

delete obj.count;

}

删除后的JSON如图

[](http://images.cnitblog.com/blog/311549/201306/08225700-1fc21b4018054d0b8d23477a035eb15f.jpg)

可以看到count已经从JSON对象中被删除了。

5，遍历JSON对象

可以使用for…in…循环来遍历JSON对象中的数据，比如我们要遍历输出obj对象的值，代码如下：

function Traversal() {

for (var c in obj) {

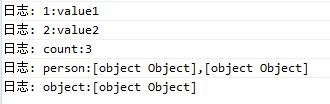
console.log(c + ":", obj[c]);

}

}

程序输出结果为：

程序输出结果为：

[](http://images.cnitblog.com/blog/311549/201306/08225705-188a9dd782ca4f42934fcba606e2b162.jpg)

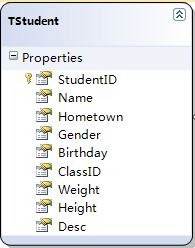
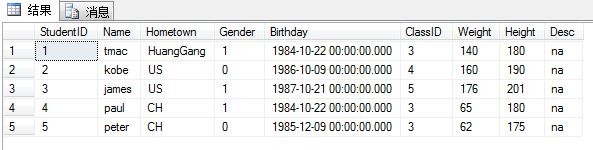
## 在.NET中如何使用JSON

说到在.net中使用JSON，就不得不提到JSON.NET，它是一个非常著名的在.net中处理JSON的工具，我们最常用的是下面两个功能。

**1，通过序列化将.net对象转换为JSON字符串**

在web开发过程中，我们经常需要将从数据库中查询到的数据(一般为一个集合，列表或数组等)转换为JSON格式字符串传回客户端，这就需要进行序 列化，这里用到的是JsonConvert对象的SerializeObject方法。其语法格式 为：JsonConvert.SerializeObject(object)，代码中的”object”就是要序列化的.net对象，序列化后返回的是 json字符串。

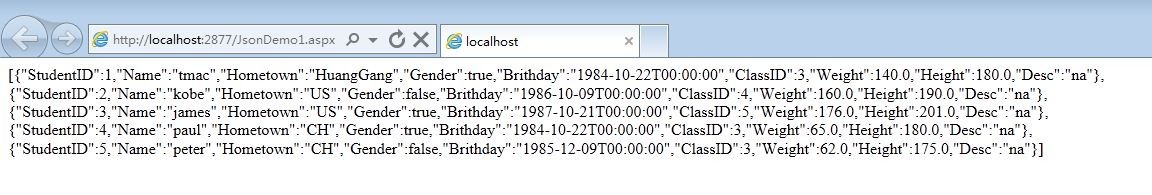
比如，现在我们有一个TStudent的学生表，表中的字段和已有数据如图所示

[](http://images.cnitblog.com/blog/311549/201306/10165638-39a0521daa1a49b5b94ee8b764699ce4.jpg)[](http://images.cnitblog.com/blog/311549/201306/10165646-be2e8e7ef4454ba0a12befa763b90b16.jpg)

从表中我们可以看到一共有五条数据，现在我们要从数据库中取出这些数据，然后利用JSON.NET的JsonConvert对象序列化它们为json字符串，并显示在页面上。C#代码如下

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifView Code

输出结果

[](http://images.cnitblog.com/blog/311549/201306/10165700-09eab47cea0d4d61a62e93d63937a40c.jpg)

从图中我们可以看到，数据库中的5条记录全部取出来并转化为json字符串了。

**2，使用LINQ to JSON定制JSON数据**

使用JsonConvert对象的SerializeObject只是简单地将一个list或集合转换为json字符串。但是，有的时候我们的前端 框架比如ExtJs对服务端返回的数据格式是有一定要求的，比如下面的数据格式，这时就需要用到JSON.NET的LINQ to JSON，LINQ to JSON的作用就是根据需要的格式来定制json数据。

比如经常用在分页的json格式如代码：

{

"total": 5, //记录总数

"rows":[

//json格式的数据列表

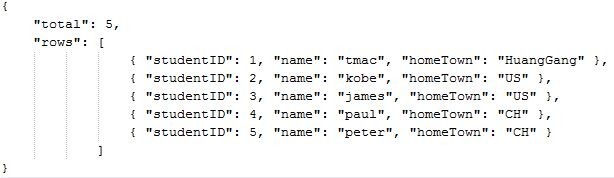
]

}

使用LINQ to JSON前，需要引用Newtonsoft.Json的dll和using Newtonsoft.Json.Linq的命名空间。LINQ to JSON主要使用到JObject, JArray, JProperty和JValue这四个对象，JObject用来生成一个JSON对象，简单来说就是生成”{}”，JArray用来生成一个JSON数 组，也就是”[]”，JProperty用来生成一个JSON数据，格式为key/value的值，而JValue则直接生成一个JSON值。下面我们就 用LINQ to JSON返回上面分页格式的数据。代码如下：

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifView Code

输出结果为：

[](http://images.cnitblog.com/blog/311549/201306/10165716-effccc4be3054f5ab8ad12692ad7e613.jpg)

**3，处理客户端提交的JSON数据**

客户端提交过来的数据一般都是json字符串，有了更好地进行操作（面向对象的方式），所以我们一般都会想办法将json字符串转换为json对象。例如客户端提交了以下数组格式json字符串。

[

{StudentID:"100",Name:"aaa",Hometown:"china"},

{StudentID:"101",Name:"bbb",Hometown:"us"},

{StudentID:"102",Name:"ccc",Hometown:"england"}

]

在服务端就可以使用JObject或JArray的Parse方法轻松地将json字符串转换为json对象，然后通过对象的方式提取数据。下面是服务端代码。

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 protected void Page\_Load(object sender, EventArgs e)

2 {

3 string inputJsonString = @"

4 [

5 {StudentID:'100',Name:'aaa',Hometown:'china'},

6 {StudentID:'101',Name:'bbb',Hometown:'us'},

7 {StudentID:'102',Name:'ccc',Hometown:'england'}

8 ]";

9 JArray jsonObj = JArray.Parse(inputJsonString);

10 string message = @"<table border='1'>

11 <tr><td width='80'>StudentID</td><td width='100'>Name</td><td width='100'>Hometown</td></tr>";

12 string tpl = "<tr><td>{0}</td><td>{1}</td><td>{2}</td></tr>";

13 foreach (JObject jObject in jsonObj)

14 {

15 message += String.Format(tpl, jObject["StudentID"], jObject["Name"],jObject["Hometown"]);

16 }

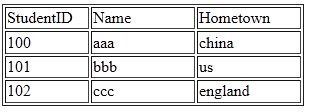
17 message += "</table>";

18 lbMsg.InnerHtml = message;

19 }

[复制代码](javascript:void(0);)

输出结果：

[](http://images.cnitblog.com/blog/311549/201306/10165723-b51ce912fc4a436195a738d0dad0d324.jpg)

**当然，服务端除了使用LINQ to JSON来转换json字符串外，也可以使用JsonConvert的DeserializeObject方法。**如下面代码实现上面同样的功能。

List<Student> studentList = JsonConvert.DeserializeObject<List<Student>>(inputJsonString);//注意这里必须为List<Student>类型,因为客户端提交的是一个数组json

foreach (Student student in studentList)

{

message += String.Format(tpl, student.StudentID, student.Name,student.Hometown);

}

## 总结

在客户端，读写json对象可以使用”.”操作符或”["key”]”，json字符串转换为json对象使用eval()函数。  
在服务 端，由.net对象转换json字符串优先使用JsonConvert对象的SerializeObject方法，定制输出json字符串使用LINQ to JSON。由json字符串转换为.net对象优先使用JsonConvert对象的DeserializeObject方法，然后也可以使用LINQ to JSON。

根据所需调用方法就行。不过也可以用Newtonsoft.Json这个dll文件，如果转换数组的话就用

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

1 JObject json = (JObject)JsonConvert.DeserializeObject(str);

2 JArray array = (JArray)json["article"];

3 foreach (var jObject in array)

4 {

5 //赋值属性

6 }

**Json.Net 学习笔记(一)**

使用Newtonsoft.Json这是一个开源的Json.Net库。

下载地址：http://json.codeplex.com/releases/view/50552。当前版本为 Release 8

从下载到的源代码中获取Newtonsoft.Json.Net20.dll,添加到自己的工程中。

using Newtonsoft.Json;

定义类:

 public class Message  
    {  
        public string Address { get; set; }  
        [JsonProperty(TypeNameHandling = TypeNameHandling.All)]  
        public object Body { get; set; }  
    }  
    public class SearchDetails  
    {  
        public string Query { get; set; }  
        public string Language { get; set; }  
    }

测试:

           Message message = new Message  
            {  
                Address = "http://google.com",  
                Body = new SearchDetails { Query = "Json.Net", Language = "en-us" }  
            };  
            string jsonMsg = JsonConvert.SerializeObject(message, Formatting.Indented);//Indented表示以缩进形式显示结果  
            System.Diagnostics.Debug.Write(jsonMsg);  
            Message deserialized = JsonConvert.DeserializeObject<Message>(jsonMsg);  
            SearchDetails searchDetails = (SearchDetails)deserialized.Body;  
            Response.Write(searchDetails.Query + "," + searchDetails.Language + "<br/>");

Debug输出结果格式：

{  
  "Address": "http://google.com",  
  "Body": {  
    "$type": "TestJsonSerialization.SearchDetails, TestJsonSerialization",  
    "Query": "Json.Net",  
    "Language": "en-us"  
  }  
}

注：1.JsonProperty标记字段或属性，用来控制它作为一个Json对象的属性序列化。

       2.TypeNameHandling 用来为Json序列化指定类型名。它有几个枚举值：

|  |  |
| --- | --- |
| **Member** | **Description** |
| None | Do not include the .NET type name when serializing types. |
| Objects | Include the .NET type name when serializing into a JSON object structure. |
| Arrays | Include the .NET type name when serializing into a JSON array structure. |
| Auto | Include the .NET type name when the type of the object being serialized is not the same as its declared type. |
| All | Always include the .NET type name when serializing. |

**Json.Net 学习笔记(二) Linq to Json**

using Newtonsoft.Json.Linq;

定义类：

    public class Product  
    {  
        public string Name { get; set; }  
        public DateTime Expiry { get; set; }  
        public decimal Price { get; set; }  
        public string[] Sizes { get; set; }  
    }

测试:

            Product product = new Product  
            {  
                Name = "Apple",  
                Expiry = new DateTime(2010, 12, 18),  
                Price = 3.99M,  
                Sizes = new string[] { "Small", "Medium", "Large" }  
            };

            string serializedJson = JsonConvert.SerializeObject(product);

            JObject o = JObject.Parse(serializedJson);  
            string name = (string)o["Name"];  
            //Apple  
            JArray sizes = (JArray)o["Sizes"];  
            string smallest = (string)sizes[0];  
            Response.Write(name + "," + smallest + "<br/>");//输出Small  
            //SelectToken  
            smallest = (string)o.SelectToken("Sizes[0]");  
            Response.Write(smallest + "<br/>");//输出Small  
            //SelectToken with Linq  
            var sizeLen5 = o["Sizes"].Select(i => (string)i).Where(i => i.Length == 5).ToList<string>();  
           foreach (var size in sizeLen5)  
            {  
                Response.Write((string)size+ " <br/>");  
            };//输出Small和Large

注:JArray表示一个Json集合，JObject表示一个Json对象。

**Json.Net学习笔记(三) Json与Xml之间的转换**

Json.Net 支持 Json 与Xml之间的相互转换.

转换规则：

 1.元素保持不变  
 2.属性添加前缀@  
 3.单个子文本结点直接作为元素结点的值，否则通过#text访问  
 4.XML声明和处理指令以?为前缀  
 5. 字符数据(Character data)、注释、whitespace 和significate whitespace结点分别通过#cdata-section,#comment,#whitespace and #significate-whitespace 访问。  
 6.同一级多个相同名字的结点作为一组，放到同一个集合中  
 7.空元素为null

using Newtonsoft.Json;

测试：

string xml = <?xml version=""1.0"" standalone=""no""?>  
                            <root>  
                            <person id=""1"">  
                            <name>Alan</name>  
                            <url>http://www.google.com</url>  
                            </person>  
                            <person id=""2"">  
                            <name>Louis</name>  
                            <url>http://www.yahoo.com</url>  
                            </person>  
                            Hello World  
                            </root>";

            XmlDocument doc = new XmlDocument();  
            doc.LoadXml(xml);

           //XML to JSON  
            string jsonText = JsonConvert.SerializeXmlNode(doc, Newtonsoft.Json.Formatting.Indented);  
            Console.WriteLine(jsonText);

输出结果：

{  
  "?xml": {  
    "@version": "1.0",  
    "@standalone": "no"  
  },  
  "root": {  
    "person": [  
      {  
        "@id": "1",  
        "name": "Alan",  
        "url": "http://www.google.com"  
      },  
      {  
        "@id": "2",  
        "name": "Louis",  
        "url": "http://www.yahoo.com"  
      }  
    ],  
    "#text": "\r\n                            Hello World\r\n                            "  
  }  
}

            //JSON to XML

            XmlDocument docJson = (XmlDocument)JsonConvert.DeserializeXmlNode(jsonText);  
            Console.WriteLine(docJson.InnerXml);

posted @ 2011-06-11 15:04 资源收集 阅读(68) 评论(0) 编辑

**Json.Net学习笔记(四) Json对象的读写**

为了手动读写Json对象，Json.Net提供了JsonReader 和 JsonWriter这两个抽象类及其相应的派生类:

1.JsonTextReader 和 JsonTextWriter

用来读写Json对象的文本，JsonTextWriter 有大量设置去控制Json对象的格式。

测试：

            //写操作

            StringBuilder sb = new StringBuilder();  
            StringWriter sw = new StringWriter(sb);  
            using (JsonWriter jsonWriter = new JsonTextWriter(sw))  
            {

                jsonWriter.Formatting = Formatting.Indented;  
                jsonWriter.WriteStartObject();  
                  
                jsonWriter.WritePropertyName("CPU");  
                jsonWriter.WriteValue("Intel");  
                jsonWriter.WritePropertyName("PSU");  
                jsonWriter.WriteValue("500W");  
                jsonWriter.WritePropertyName("Drives");

                jsonWriter.WriteStartArray();  
                jsonWriter.WriteValue("DVD read/writer");  
                jsonWriter.WriteComment("(broken)");  
                jsonWriter.WriteValue("500 gigabyte hard drive");  
                jsonWriter.WriteValue("200 gigabype hard drive");

                jsonWriter.WriteEndArray();  
                jsonWriter.WriteEndObject();

            }

            Console.WriteLine(sb.ToString());

输出结果：

{  
  "CPU": "Intel",  
  "PSU": "500W",  
  "Drives": [  
    "DVD read/writer"  
    /\*(broken)\*/,  
    "500 gigabyte hard drive",  
    "200 gigabype hard drive"  
  ]  
}

      //读操作

     TextReader txtReader = new StringReader(sb.ToString());  
            using (JsonReader jsonReader = new JsonTextReader(txtReader))  
            {  
                string readResult = "{\n";  
                while (jsonReader.Read())  
                {  
                    if (jsonReader.TokenType == JsonToken.PropertyName)  
                    {  
                        readResult += (string)jsonReader.Value + ":";  
                    }  
                    else if (jsonReader.TokenType == JsonToken.String)  
                    {  
                        readResult += (string)jsonReader.Value + "\n";  
                    }  
                    else if (jsonReader.TokenType == JsonToken.StartArray)  
                    {  
                        readResult += "[\n";  
                        while (jsonReader.Read())  
                        {  
                            if (jsonReader.TokenType == JsonToken.EndArray)  
                            {  
                                readResult = readResult.Remove(readResult.Length - 2, 1);  
                                readResult += "]\n";  
                                break;  
                            }  
                            if (jsonReader.TokenType == JsonToken.String)  
                                readResult += (string)jsonReader.Value + ",\n";  
                            else if (jsonReader.TokenType == JsonToken.Comment)  
                                readResult += "/\*" + (string)jsonReader.Value + "\*/,\n";  
                        }  
                    }  
                }  
                readResult += "}\n";  
                Console.WriteLine(readResult);

             }

2.JTokenReader and JTokenWriter

JTokenReader and JTokenWriter是读写Linq to Json对象的，他们存在于Newtonsoft.Json.Linq名字空间中

         JObject o = new JObject(  
                        new JProperty("Name", "ZhangSan"),  
                        new JProperty("BirthDay", new DateTime(1985, 11, 27)));

            JsonSerializer serializer = new JsonSerializer();  
            Person person = (Person)serializer.Deserialize(new JTokenReader(o), typeof(Person));  
            Console.WriteLine(person.Name);//输出ZhangSan

**Json.Net学习笔记(五) 自定义Json序列化特性**

Attributes 可以用来控制Json.Net如何序列化和反序列化.Net对象。

>JsonObjectAttribute--标记在类上，用于控制该类如何被序列化为一个Json对象(JSON Object)

>JsonArrayAttribute--标记在集合上，用于控制该集合如何被序列化为一个Json集合(JSON Array)

>JsonPropertyAttribute--标记在字段和属性上，用于控制它如何被序列化为一个Json对象中的属性

>JsonConverterAttribute--标记在类或者字段和属性上，用于序列化时指定Json转换器(JsonConverter)

 和使用内置的Json.Net attributes一样，当确定Json如何被序列化和反序列化时，Json.Net也查找DataContract和DataMember attributes。如果都存在，Json.Net 序列化特性将优先使用。

示例：

   [JsonObject(MemberSerialization.OptOut)]  
    public class Person  
    {  
        // "John Smith"  
        [JsonProperty]  
        public string Name { get; set; }  
        // "2000-12-15T22:11:03"  
        [JsonProperty]  
        [JsonConverter(typeof(IsoDateTimeConverter))]  
        public DateTime BirthDate { get; set; }  
        // new Date(976918263055)  
        [JsonProperty]  
        [JsonConverter(typeof(JavaScriptDateTimeConverter))]  
        public DateTime LastModified { get; set; }  
        // not serialized  
        public string Department { get; set; }  
    }  
->JsonObjectAttribute  
  这 里的MemberSerialization标记指定成员序列化为opt-in(每一个成员都必须标记JsonProperty或者DataMember attribute才被序列化)或者opt-out(默认情况下，除了被标记为JsonIgnoreAttribute的所有成员都将被序列化,这是 Json.Net 的默认行为)  
  Json.Net把实现了IEnumerable接口的.Net类序列化为一个Json集合，用IEnumerable 的值来填充该集合。标记JsonPropertyAttribute来重写此行为，并强制序列化器序列化该类的字段和属性。  
->JsonPropertyAttribute  
  JsonPropertyAttribute有很多用途  
    1.默认情况下，Json 属性与.Net属性有相同的名字，这个attribute允许自定义属性名  
    2.当member serialization 被设置成opt-in时，指定属性应该被序列化  
    3.在序列化和反序列化时包含非public属性  
->JsonIgnoreAttribute  
  从序列化中排除指定的字段或者属性  
->JsonConverterAttribute  
  JsonConverterAttribute指定Json序列化器(JsonSerializer)用来转换一个对象  
  这个特性可以用来标记类或者成员，当它用来标记类的时候，由attribute指定的JsonConverter将为类提供默认的序列化方法.当它用来标记字段或者属性的时候，指定的JsonConverter将一直用于序列化该值。  
  使用JsonConverter的优先级顺序为：先成员，后类，最后为传给JsonSerializer的任何转换器.  
示例:  
    public class MemberConverterClass  
    {  
        public DateTime DefaultConverter { get; set; }  
        [JsonConverter(typeof(IsoDateTimeConverter))]  
        public DateTime MemberConverter { get; set; }  
    }//注：这是一个JsonConverterAttribute 应用于属性的例子  
测试:  
 DateTime date = Convert.ToDateTime("1970-01-01T00:00:00Z").ToUniversalTime();  
            MemberConverterClass c = new MemberConverterClass  
            {  
                DefaultConverter = date,  
                MemberConverter = date  
            };  
            string json = JsonConvert.SerializeObject(c, Formatting.Indented);  
            Console.WriteLine(json);  
输出结果:  
{  
  "DefaultConverter": "\/Date(0)\/",  
  "MemberConverter": "1970-01-01T00:00:00Z"  
}

posted @ 2011-06-11 15:08 资源收集 阅读(133) 评论(0) 编辑

**Json.Net学习笔记(六) Json.Net中关于时间的序列化**

DateTime类型在Json中表示是很难的。

问题来源于JSON spec (Json规范)自身，没有精确的语法。这个规范包括objects, arrays, strings, integers 和 floats,但是对于date是什么样子的没有定义准确的标准。

关于时间，Json.Net使用的默认格式与微软:"\/Date(1198908717056)\/"相同，您可以从此外获得更多信息.

DateTime JsonConverters

由于在Json中没有关于时间的准确标准，当与其他系统进行互操作时就会有大量不同的时间格式。幸运的是，Json.Net为了处理读写自定义时间提供了一个解决方案:JsonConverters. JsonConverter是用来重写一个类型如何被序列化的。

示例：

public class LogEntry  
    {  
        public string Details { get; set; }  
        public DateTime LogDate { get; set; }  
    }

测试1：

       LogEntry logEntry = new LogEntry()  
            {  
                LogDate = new DateTime(2010, 12, 19, 0, 0, 0, DateTimeKind.Utc),  
                Details = "Application started."  
            };  
            string defaultJson = JsonConvert.SerializeObject(logEntry, Formatting.Indented);  
            Console.WriteLine(defaultJson);

输出结果：

{  
  "Details": "Application started.",  
  "LogDate": "\/Date(1292716800000)\/"  
}

测试2:

    string javascriptJson = JsonConvert.SerializeObject(logEntry, Formatting.Indented, new JavaScriptDateTimeConverter());  
     Console.WriteLine(javascriptJson);  
输出结果：

{  
  "Details": "Application started.",  
  "LogDate": new Date(  
    1292716800000  
  )  
}

测试3:

            string isoJson = JsonConvert.SerializeObject(logEntry, Formatting.Indented, new IsoDateTimeConverter());  
            Console.WriteLine(isoJson);

输出结果：

{  
  "Details": "Application started.",  
  "LogDate": "2010-12-19T00:00:00Z"  
}

 简单地，通过JsonConverter您可以使用Json.Net的序列化器

>JavaScriptDateTimeConverter

  JavaScriptDateTimeConverter类是来自于Json.Net的两个DataTime Json转换器之一,这个转换器序列化一个DateTime类型为一个JavaScipt日期对象 (https://developer.mozilla.org/en/Core\_JavaScript\_1.5\_Reference/Global\_Objects/Date)

  new Date(1292716800000)

从技术上讲，根据规范这是无效的数据格式，但是所有的浏览器、和一些Json的框架包括Json.Net都支持它。

>IsoDateTimeConverter

  IsoDateTimeConverter序列一个DateTime类型为一个ISO 8601格式的字符串.

  "2010-12-19T00:00:00Z"

IsoDateTimeConverter 类有一个属性DateTimeFormat,用来更进一步的自定义字符串格式.

最后要注意的一件事是，所有被Json.Net返回的日期值都是UTC Time

**Json.Net学习笔记(七) 序列化集合**

Json序列化器为序列化及反序列化集合对象提供了良好的支持.

->Serializing

   为了序列化一个集合---一个泛型的list,array,dictionary,或者自定义集合---简单地调用序列化器，使用您想要进行序列化的集合对象作为参数，Json.Net会序列化集合以及所有它包含的值。

示例：

public class Product  
    {  
        public string Name { get; set; }  
        public decimal Price { get; set; }  
        [JsonConverter(typeof(IsoDateTimeConverter))]  
        public DateTime ExpiryDate { get; set; }  
    }

测试：

            Product p1 = new Product()  
            {  
                Name = "Product 1",  
                Price = 99.95m,  
                ExpiryDate = new DateTime(2010, 12, 19, 0, 0, 0, DateTimeKind.Utc)  
            };  
            Product p2 = new Product  
            {  
                Name = "Product 2",  
                Price = 12.50m,  
                ExpiryDate = new DateTime(2011, 1, 1, 0, 0, 0, DateTimeKind.Utc)  
            };  
            List<Product> products = new List<Product>();  
            products.Add(p1);  
            products.Add(p2);  
            string json = JsonConvert.SerializeObject(products, Formatting.Indented);  
            Console.WriteLine(json);

输出结果：

[  
  {  
    "Name": "Product 1",  
    "Price": 99.95,  
    "ExpiryDate": "2010-12-19T00:00:00Z"  
  },  
  {  
    "Name": "Product 2",  
    "Price": 12.50,  
    "ExpiryDate": "2011-01-01T00:00:00Z"  
  }  
]

->Deserializing

  为了反序列化Json到一个.Net集合中，只要指定一个您想要反序列化的集合类型就可以了，Json.Net支持多种类型的集合.

示例:

    string json2 = @"[  
                              {  
                                ""Name"": ""Product 1"",  
                                ""ExpiryDate"": ""2010-12-19T00:00:00Z"",  
                                ""Price"": 99.95,  
                                ""Sizes"": null  
                              },  
                              {  
                                ""Name"": ""Product 2"",  
                                ""ExpiryDate"": ""2011-01-01T00:00:00Z"",  
                                ""Price"": 12.50,  
                                ""Sizes"": null  
                              }  
                            ]";  
            List<Product> productList = JsonConvert.DeserializeObject<List<Product>>(json2);  
            Console.WriteLine(productList.Count);//2  
            Product product1 = productList[0];  
            Console.WriteLine(product1.Name);//Product 1

->Deserializing Dictionaries

  使用Json.Net你也可以反序列化一个Json对象到一个.Net的泛型Dictionary中.Json对象的属性名和属性值将会被添加到Dictionary中.

示例:

            string jsonDictionary = @"{""key1"":""value1"",""key2"":""value2""}";  
            Dictionary<string, string> dic = JsonConvert.DeserializeObject<Dictionary<string, string>>(jsonDictionary);  
            Console.WriteLine(dic.Count);//2  
            Console.WriteLine(dic["key1"]);//value1

posted @ 2011-06-11 15:09 资源收集 阅读(73) 评论(0) 编辑

**Json.Net学习笔记(八) 序列化的回调方法**

Json.Net支持序列化回调方法，回调方法通过Json序列化器(JsonSerializer)可以用来操作一个对象，在它被序列化和反序列化之前或者之后.

OnSerializing   
OnSerialized   
OnDeserializing   
OnDeserialized

为了告诉序列化器在对象 的序列化生命周期中哪个方法应该被调用，需要用适当的 attribute(OnSerializingAttribute, OnSerializedAttribute, OnDeserializingAttribute, OnDeserializedAttribute) 来标记方法.

例如对象序列化的回调方法:

using Newtonsoft.Json;  
using System.Runtime.Serialization;

 public class SerializationEventTestObject  
    {  
        // This member is serialized and deserialized with no change.  
        public int Member1 { get; set; }  
        // The value of this field is set and reset during and   
        // after serialization.  
        public string Member2 { get; set; }  
        // This field is not serialized. The OnDeserializedAttribute   
        // is used to set the member value after serialization.  
        [JsonIgnore]  
        public string Member3 { get; set; }  
        // This field is set to null, but populated after deserialization.  
        public string Member4 { get; set; }  
        public SerializationEventTestObject()  
        {  
            Member1 = 11;  
            Member2 = "Hello World!";  
            Member3 = "This is a nonserialized value";  
            Member4 = null;  
        }  
        [OnSerializing]  
        internal void OnSerializingMethod(StreamingContext context)  
        {  
            Member2 = "This value went into the data file during serialization.";  
        }  
        [OnSerialized]  
        internal void OnSerializedMethod(StreamingContext context)  
        {  
            Member2 = "This value was reset after serialization.";  
        }  
        [OnDeserializing]  
        internal void OnDeserializingMethod(StreamingContext context)  
        {  
            Member3 = "This value was set during deserialization";  
        }  
        [OnDeserialized]  
        internal void OnDeserializedMethod(StreamingContext context)  
        {  
            Member4 = "This value was set after deserialization.";  
        }  
    }

示例对象被序列化和反序列化:

 SerializationEventTestObject obj = new SerializationEventTestObject();  
            Console.WriteLine(obj.Member1);// 11  
            Console.WriteLine(obj.Member2);// Hello World!  
            Console.WriteLine(obj.Member3);// This is a nonserialized value  
            Console.WriteLine(obj.Member4);// null  
            string json = JsonConvert.SerializeObject(obj, Formatting.Indented);  
            Console.WriteLine(json);

            //{  
            //  "Member1": 11,  
            //  "Member2": "This value went into the data file during serialization.",  
            //  "Member4": null  
            //}

            Console.WriteLine(obj.Member1);//11  
            Console.WriteLine(obj.Member2);// This value was reset after serialization.  
            Console.WriteLine(obj.Member3);// This is a nonserialized value  
            Console.WriteLine(obj.Member4);// null

            obj = JsonConvert.DeserializeObject<SerializationEventTestObject>(json);  
            Console.WriteLine(obj.Member1);//11  
            Console.WriteLine(obj.Member2);// This value went into the data file during serialization.  
            Console.WriteLine(obj.Member3);// This value was set during deserialization  
            Console.WriteLine(obj.Member4);// This value was set after deserialization

**Json.Net学习笔记(九) 异常处理**

Json.Net支持在序列化和反序列化的过程中进行异常处理。异常处理让您捕获一个异常，您可以选择是否处理它，继续序列化或者让异常抛给上一层，在你的应用程序中被抛出。

异常处理通过两个方法来定义：the Error event on JsonSerializer 和 OnErrorAttribute

>Error Event

  error event是一个建立在JsonSerializer 上的异常处理.当序列化或者反序列化JSON时，任何有异常抛出的情况error event都会被触发.就像建立在JsonSerializer上的所有设置一样，它也可以在JsonSerializerSettings 上进行设置从而传递给JsonConvert的序列化方法.

示例：

 List<string> errors = new List<string>();  
            List<DateTime> c = JsonConvert.DeserializeObject<List<DateTime>>(@"[  
                  ""2010-12-19T00:00:00Z"",  
                  ""I am not a date and will error!"",  
                  [  
                    1  
                  ],  
                  ""2011-01-01T00:00:00Z"",  
                  null,  
                  ""2010-12-25T00:00:00Z""  
                  ]", new JsonSerializerSettings()  
                        {  
                            Error = delegate(object sender, Newtonsoft.Json.Serialization.ErrorEventArgs e)  
                            {  
                                errors.Add(e.ErrorContext.Error.Message);  
                                e.ErrorContext.Handled = true;  
                            },  
                            Converters = { new IsoDateTimeConverter() }  
                        });  
            foreach (DateTime t in c)  
            {  
                Console.WriteLine(t.ToString());  
            }  
            //2010-12-19 00:00:00  
            //2011-01-01 00:00:00  
            //2010-12-25 00:00:00  
            foreach (string err in errors)  
            {  
                Console.WriteLine(err);  
            }  
            //The string was not recognized as a valid DateTime. There is a unknown word starting at index 0.  
            //Unexpected token parsing date. Expected String, got StartArray.  
            //Cannot convert null value to System.DateTime.

  在这个例子中我们把一个Json数组反序列化为一个DateTime的集合，在JsonSerializerSettings中有一个hander被赋值 成了error event ,它用来记录error message，并标记这个error为已处理(handled).

反序列化JSON的结果 是三个被成功反序列化的日期和三个error messages：一个是不正确的格式,"I am not a date and will error!",一个是嵌套了JSON数组，最后一个是null值，因为定义的list不允许有可空类型的DateTime.这个事件处理已经记录了这些 信息，Json.Net在序列化时继续进行(没有因为异常而停止)，因为这些错误已经被标记为已处理。

   值得注意的是，在Json.Net进行异常处理时，没有处理的异常将会被抛到上一层，并在它的每个parent触发事件，例如：在序列化若干对象的集合 时，一个未处理的异常将被触发两次，首先在对象上，然后在集合上。这样就会让您在处理异常的时候，选择在它发生的地方，或者是它的一个parent上。

 JsonSerializer serializer = new JsonSerializer();  
            serializer.Error += delegate(object sender, Newtonsoft.Json.Serialization.ErrorEventArgs e)  
            {  
                // only log an error once  
                if (e.CurrentObject == e.ErrorContext.OriginalObject)  
                    errors.Add(e.ErrorContext.Error.Message);  
            };

如果您不是立即处理一个 异常，仅仅是想针对它完成一次操作，您可以验证一下ErrorEventArg's CurrentObject是否等于OriginalObject.OriginalObject是抛出异常的对象，CurrentObject是事件被 触发的对象.他们只会在第一次(事件被OriginalObject触发时)相等.

>OnErrorAttribute

 OnErrorAttribute 的工作方式非常像其他Json.Net支持的.NET serialization attributes ,简单地把它标记在带有正确参数(一个StreamingContext和一个ErrorContext)的方法上就可以使用了，与方法的名字没有关系。

示例：

  public class PersonError  
    {  
        private List<string> \_roles;  
        public string Name { get; set; }  
        public int Age { get; set; }  
        public List<string> Roles  
        {  
            get  
            {  
                if (\_roles == null)  
                    throw new Exception("Roles not loaded!");  
                return \_roles;  
            }  
            set { \_roles = value; }  
        }  
        public string Title { get; set; }  
        [OnError]  
        internal void OnError(StreamingContext context, ErrorContext errorContext)  
        {  
            errorContext.Handled = true;  
        }  
    }

在这个例子中，当\_roles没有被设置值时访问Roles属性将会抛出一个异常.在序列化Roles属性时，异常处理的方法将设置error为handled,从而允许Json.Net继续序列化这个类。

测试：

 PersonError person = new PersonError  
            {  
                Name = "George Michael Bluth",  
                Age = 16,  
                Roles = null,  
                Title = "Mister Manager"  
            };  
            string json = JsonConvert.SerializeObject(person, Formatting.Indented);  
            Console.WriteLine(json);

输出：

            {  
              "Name": "George Michael Bluth",  
              "Age": 16,  
              "Title": "Mister Manager"  
            }

posted @ 2011-06-11 15:10 资源收集 阅读(57) 评论(0) 编辑

**Json.Net学习笔记(十) 保持对象引用**

默认情况下，Json.Net将通过对象的值来序列化它遇到的所有对象。如果工个list包含两个Person引用，这两个引用都指向同一个对象，Json序列化器将输出每一个引用的所有名称和值。

定义类：

  public class Person  
    {  
        public DateTime BirthDate { get; set; }  
        public DateTime LastModified { get; set; }  
        public string Name { get; set; }  
    }

测试：

         Person p = new Person()  
            {  
                BirthDate = new DateTime(1985, 11, 27, 0, 0, 0, DateTimeKind.Utc),  
                LastModified = new DateTime(2010, 12, 20, 0, 0, 0, DateTimeKind.Utc),  
                Name = "James"  
            };  
            List<Person> people = new List<Person>();  
            people.Add(p);  
            people.Add(p);  
            string json = JsonConvert.SerializeObject(people, Formatting.Indented);  
            Console.WriteLine(json);

输出结果：

[  
  {  
    "BirthDate": "\/Date(501897600000)\/",  
    "LastModified": "\/Date(1292803200000)\/",  
    "Name": "James"  
  },  
  {  
    "BirthDate": "\/Date(501897600000)\/",  
    "LastModified": "\/Date(1292803200000)\/",  
    "Name": "James"  
  }  
]

在大多数情况下这是期望 的结果，但是在某些场景下，将list中的第二项作为第一项的一个引用来输出会是一个更好的解决方案。如果上面的Json现在被反序列化，返回的list 会包含两个完全分离的对象，它们具有相同的值。通过值来输出引用也会在对象上导致循环引用的发生。

>PreserveReferencesHandling

   string json2 = JsonConvert.SerializeObject(people, Formatting.Indented,  
                new JsonSerializerSettings() { PreserveReferencesHandling = PreserveReferencesHandling.Objects });  
            Console.WriteLine(json2);

输出结果：

          [  
              {  
                "$id": "1",  
                "BirthDate": "\/Date(501897600000)\/",  
                "LastModified": "\/Date(1292803200000)\/",  
                "Name": "James"  
              },  
              {  
                "$ref": "1"  
              }  
            ]

          List<Person> deserializedPeople = JsonConvert.DeserializeObject<List<Person>>(json2,  
                new JsonSerializerSettings { PreserveReferencesHandling = PreserveReferencesHandling.Objects });

            Console.WriteLine(deserializedPeople.Count);// 2

            Person p1 = deserializedPeople[0];  
            Person p2 = deserializedPeople[1];  
            Console.WriteLine(p1.Name);// James  
            Console.WriteLine(p2.Name);// James  
            bool equal = Object.ReferenceEquals(p1, p2);// true

在list中的第一个Person被序列化时增加了一个额外的对象Id,现在第二个Person对象仅仅是第一个的引用。

现在使用PreserveReferencesHandling后，在序列化时只创建了一个Person对象，list中包含它的两个引用，原来我们叫作反射(mirroring) 。

>IsReference on JsonObjectAttribute, JsonArrayAttribute and JsonPropertyAttribute

  在对象序列化器上设置PreserveReferencesHandling，将会改变所有对象被序列化和反序列化的方式。为了更加细致地控制对象和成员 被序列化为一个引用，可以在JsonObjectAttribute, JsonArrayAttribute 和 JsonPropertyAttribute上使用IsReference 属性.

在JsonObjectAttribute, JsonArrayAttribute 上设置IsReference 为true，意味着Json序列化器总是会序列这个类型为一个引用。在JsonPropertyAttribute上设置IsReference为 true将只序列化这个属性为一个引用。

 [JsonObject(IsReference = true)]  
    public class EmployeeReference  
    {  
        public string Name { get; set; }  
        public EmployeeReference Manager { get; set; }  
    }

测试：

           List<EmployeeReference> empList = new List<EmployeeReference>();  
            empList.Add(empRef);  
            empList.Add(empRef);  
            string empRefJson = JsonConvert.SerializeObject(empList, Formatting.Indented);  
            Console.WriteLine(empRefJson);

输出结果:

[  
  {  
    "$id": "1",  
    "Name": "IsReference",  
    "Manager": null  
  },  
  {  
    "$ref": "1"  
  }  
]

>IReferenceResolver

要想定制引用的生成方式，可以继承自IReferenceResolver接口来使用Json序列化器。

**Json.Net学习笔记(十一) CustomCreationConverter**

CustomCreationConverter是一个在序列化过程中提供自定方式去创建一个对象的Json转换器，一旦对象被创建，它将被序列化器填充值。

 public interface IPerson  
    {  
        string FirstName { get; set; }  
        string LastName { get; set; }  
        DateTime BirthDate { get; set; }  
    }

    public class Employee : IPerson  
    {  
        public string FirstName { get; set; }  
        public string LastName { get; set; }  
        public DateTime BirthDate { get; set; }

        public string Department { get; set; }  
        public string JobTitle { get; set; }  
    }  
    public class PersonConverter : CustomCreationConverter<IPerson>  
    {  
        public override IPerson Create(Type objectType)  
        {  
            return new Employee();  
        }  
    }

这是一个非常简单的例子。更复杂的场景可能包含一个对象工厂或者服务定位器(service locator)用来在运行时解析这个对象。

测试:

      string json = @"[  
              {  
                ""FirstName"": ""Maurice"",  
                ""LastName"": ""Moss"",  
                ""BirthDate"": ""\/Date(252291661000)\/"",  
                ""Department"": ""IT"",  
                ""JobTitle"": ""Support""  
              },  
              {  
                ""FirstName"": ""Jen"",  
                ""LastName"": ""Barber"",  
                ""BirthDate"": ""\/Date(258771661000)\/"",  
                ""Department"": ""IT"",  
                ""JobTitle"": ""Manager""  
              }  
            ]";

            List<IPerson> people = JsonConvert.DeserializeObject<List<IPerson>>(json, new PersonConverter());  
            IPerson person = people[0];  
            Console.WriteLine(person.GetType());// CustomCreationConverterTest.Employee            
            Console.WriteLine(person.FirstName);// Maurice  
            Employee employee = (Employee)person;  
            Console.WriteLine(employee.JobTitle);// Support

**Json.Net学习笔记(十二) 协议解析**

IContractResolver接口提供了一种方法去定制序列化器如何去序列化和反序列化.Net对象为Json对象

实现IContractResolver接口，然后为Json序列化器分配一个实例对象，这样你就可以控制对象是否被序列化为Json对象或者Json数组、对象的成员应该被序列化成什么、如何被序列化、及他们被谁调用等。

->DefaultContractResolver

   DefaultContractResolver是序列化器默认的解析器，它以虚函数的形式提供了很多扩展功能，可以被覆写(overriden).

->CamelCasePropertyNamesContractResolver

   CamelCasePropertyNamesContractResolver继承自DefaultContractResolver，并且简单的覆写了Json属性名为camel命名规则的形式(首字母小写)

  public class Product  
    {  
        public string Name { get; set; }  
        public decimal Price { get; set; }  
        public DateTime ExpiryDate { get; set; }  
        public string[] Sizes { get; set; }

    }

测试：

 Product product = new Product  
            {  
                ExpiryDate = new DateTime(2010, 12, 20, 18, 1, 0, DateTimeKind.Utc),  
                Name = "Widget",  
                Price = 9.99m,  
                Sizes = new[] { "Small", "Medium", "Large" }  
            };  
            string json = JsonConvert.SerializeObject(  
                            product,  
                            Formatting.Indented,  
                            new JsonSerializerSettings { ContractResolver = new CamelCasePropertyNamesContractResolver() }  
                             );  
            Console.WriteLine(json);

输出结果:

{  
  "name": "Widget",  
  "price": 9.99,  
  "expiryDate": "\/Date(1292868060000)\/",  
  "sizes": [  
    "Small",  
    "Medium",  
    "Large"  
  ]

**Json.Net学习笔记(十三) 缩小序列化Json的大小**

当序列化.Net对象为Json对象时经常会遇到的一个问题是：最终的Json包含大量多余的属性和属性值。这个问题在返回Json到客户端时显得特别重要。Json越大意味着需要更大的带宽，使网速变得更慢。

为了解决多余的Json这个问题，Json.Net有一系列内置的选项可以进行调整。

->JsonIgnoreAttribute and DataMemberAttribute

默认情况下，在Json创建的时候Json.Net会包含所有类级别的public属性和字段。添加JsonIgnoreAttribute到属性上，告诉序列化器序列化时跳过它。

 public class Car  
    {  
        // included in JSON  
        public string Model { get; set; }  
        public DateTime Year { get; set; }  
        public List<string> Features { get; set; }

        // ignored  
        [JsonIgnore]  
        public DateTime LastModified { get; set; }  
    }

如果类有很多属性，你只 想序列化它的一小部分，添加JsonIgore到所有其他的属性上会比较冗余，也比较容易出错。有一种用来处理这种情况的方法，添加 DataContractAttribute到类上，添加DataMemberAttribute到需要被序列化的属性上。与使用 JsonIgnoreAttribute的opt-out序列化相比，opt-in序列化仅仅你标记的属性被需列化。

using System.Runtime.Serialization;

  [DataContract]  
    public class Computer  
    {  
        // included in JSON  
        [DataMember]  
        public string Name { get; set; }  
        [DataMember]  
        public decimal SalePrice { get; set; }

        // ignored  
        public string Manufacture { get; set; }  
        public int StockCount { get; set; }  
        public decimal WholeSalePrice { get; set; }  
        public DateTime NextShipmentDate { get; set; }  
    }

->Formatting

Json序列化时，用可选参数Formatting.Indented生成良好的显示格式，可读性更好。另一方面，Formatting.None会跳过不必要的空格和换行符，让Json的结果更小。生成的显示格式更加紧凑，也许效率更高。

->NullValueHandling

   在序列化器中NullVlaueHandling是可选的。它控制序列化器如何处理值为null的属性。通过设置NullValueHandling.Ignore值，序列化器会跳过值为null的属性。

  public class Movie  
    {  
        public string Name { get; set; }  
        public string Description { get; set; }  
        public string Classification { get; set; }  
        public string Studio { get; set; }  
        public DateTime? ReleaseDate { get; set; }  
        public List<string> ReleaseCountries { get; set; }  
    }

测试1：

           Movie movie = new Movie();  
            movie.Name = "Bad Boys III";  
            movie.Description = "It's no Bad Boys";

            string included = JsonConvert.SerializeObject(movie,  
              Formatting.Indented,  
              new JsonSerializerSettings { });  
            Console.WriteLine(included);

输出结果：

{  
  "Name": "Bad Boys III",  
  "Description": "It's no Bad Boys",  
  "Classification": null,  
  "Studio": null,  
  "ReleaseDate": null,  
  "ReleaseCountries": null  
}

测试2:

           string ignored = JsonConvert.SerializeObject(movie,  
              Formatting.Indented,  
              new JsonSerializerSettings { NullValueHandling = NullValueHandling.Ignore });  
            Console.WriteLine(ignored);

输出结果：

{  
  "Name": "Bad Boys III",  
  "Description": "It's no Bad Boys"  
}

NullValueHandling也可以在单独的属性上用JsonPropertyAttribute进行定制。使用JsonPropertyAttribute时设置的NullValueHandling值会在Json序列化器中为该属性进行重写。

->DefaultValuehandling

   在序列化器中DefaultValuehandling是可选的。它控制序列化器如何处理带有默认值的属性。通过设置 DefaultValuehandling.Ignore值，序列化器会跳过带有默认值的属性。对于引用类型为null，对于值类型，如int和 DateTime，序列化器将跳过默认未初使化值的值类型。

  Json.Net也允许你通过DefaultValueAttribute自定义默认值，例如：如果一个字符串类型的属性Department在它的默认 状态下总是返回一个空字符，但是在你的Json对象中你不想要那个空字符串，你可以在Department上使用空字符串参数标记 DefaultValueAttribute，这意味着Department在序列化时不再写入Json中，除非它有非空值。

  public class Invoice  
    {  
        public string Company { get; set; }  
        public decimal Amount { get; set; }

        // false is default value of bool  
        public bool Paid { get; set; }  
        // null is default value of nullable  
        public DateTime? PaidDate { get; set; }

        // customize default values  
        [DefaultValue(30)]  
        public int FollowUpDays { get; set; }  
        [DefaultValue("")]  
        public string FollowUpEmailAddress { get; set; }  
    }

测试1：

           Invoice invoice = new Invoice  
            {  
                Company = "Acme Ltd.",  
                Amount = 50.0m,  
                Paid = false,  
                FollowUpDays = 30,  
                FollowUpEmailAddress = string.Empty,  
                PaidDate = null  
            };

            string includedDefaultValue = JsonConvert.SerializeObject(invoice,  
              Formatting.Indented,  
              new JsonSerializerSettings { });  
            Console.WriteLine(includedDefaultValue);

输出结果：

{  
  "Company": "Acme Ltd.",  
  "Amount": 50.0,  
  "Paid": false,  
  "PaidDate": null,  
  "FollowUpDays": 30,  
  "FollowUpEmailAddress": ""  
}

测试2:

            string ignoredDefaultValue = JsonConvert.SerializeObject(invoice,  
              Formatting.Indented,  
              new JsonSerializerSettings { DefaultValueHandling = DefaultValueHandling.Ignore });  
            Console.WriteLine(ignoredDefaultValue);

输出结果:

{  
  "Company": "Acme Ltd.",  
  "Amount": 50.0,  
  "Paid": false//??  
}

DefaultValueHandling也可以在单独的属性上用JsonPropertyAttribute进行定制。使用JsonPropertyAttribute时设置的DefaultValueHandling值会在Json序列化器中为该属性进行重写。

->IContractResolver

  为了获得更多的灵活性，IContractResolver接口提供了定制.Net对象序列化为Json的每一个方面，包括在运行时改变序列化的行为。

 public class DynamicContractResolver : DefaultContractResolver  
    {  
        private readonly char \_startingWithChar;  
        public DynamicContractResolver(char startingWithChar)  
        {  
            \_startingWithChar = startingWithChar;  
        }

        protected override IList<JsonProperty> CreateProperties(JsonObjectContract contract)  
        {  
            IList<JsonProperty> properties = base.CreateProperties(contract);

            // only serializer properties that start with the specified character  
            properties =  
              properties.Where(p => p.PropertyName.StartsWith(\_startingWithChar.ToString())).ToList();

            return properties;  
        }  
    }

    public class Book  
    {  
        public string BookName { get; set; }  
        public decimal BookPrice { get; set; }  
        public string AuthorName { get; set; }  
        public int AuthorAge { get; set; }  
        public string AuthorCountry { get; set; }  
    }

测试:

           Book book = new Book  
            {  
                BookName = "The Gathering Storm",  
                BookPrice = 16.19m,  
                AuthorName = "Brandon Sanderson",  
                AuthorAge = 34,  
                AuthorCountry = "United States of America"  
            };

            string startingWithA = JsonConvert.SerializeObject(book, Formatting.Indented,  
              new JsonSerializerSettings { ContractResolver = new DynamicContractResolver('A') });  
            Console.WriteLine(startingWithA);  
            // {  
            //   "AuthorName": "Brandon Sanderson",  
            //   "AuthorAge": 34,  
            //   "AuthorCountry": "United States of America"  
            // }

            string startingWithB = JsonConvert.SerializeObject(book, Formatting.Indented,  
              new JsonSerializerSettings { ContractResolver = new DynamicContractResolver('B') });  
            Console.WriteLine(startingWithA);  
            // {  
            //   "BookName": "The Gathering Storm",  
            //   "BookPrice": 16.19  
            // }

**Json.Net学习笔记(十四) JSON的部分序列化**

通常当用到大的Json文档的时候，你可能只对其中的一小个片段信息感兴趣。这种情况下你想把Json.Net转换为.Net 对象就会让人很困扰，因为你必须为Json的整个结果定义一个.Net的类。

使用Json.Net很容易避开这个问题。在把它们传递到Json.Net序列化器之前，你可以使用Linq to Json 提取Json中你想要序列化的一些片段。

 string googleSearchText = @"{  
                  ""responseData"": {  
                    ""results"": [  
                      {  
                        ""GsearchResultClass"": ""GwebSearch"",  
                        ""unescapedUrl"": ""http://en.wikipedia.org/wiki/Paris\_Hilton"",  
                        ""url"": ""http://en.wikipedia.org/wiki/Paris\_Hilton"",  
                        ""visibleUrl"": ""en.wikipedia.org"",  
                        ""cacheUrl"": ""http://www.google.com/search?q=cache:TwrPfhd22hYJ:en.wikipedia.org"",  
                        ""title"": ""<b>Paris Hilton</b> - Wikipedia, the free encyclopedia"",  
                        ""titleNoFormatting"": ""Paris Hilton - Wikipedia, the free encyclopedia"",  
                        ""content"": ""[1] In 2006, she released her debut album...""  
                      },  
                      {  
                        ""GsearchResultClass"": ""GwebSearch"",  
                        ""unescapedUrl"": ""http://www.imdb.com/name/nm0385296/"",  
                        ""url"": ""http://www.imdb.com/name/nm0385296/"",  
                        ""visibleUrl"": ""www.imdb.com"",  
                        ""cacheUrl"": ""http://www.google.com/search?q=cache:1i34KkqnsooJ:www.imdb.com"",  
                        ""title"": ""<b>Paris Hilton</b>"",  
                        ""titleNoFormatting"": ""Paris Hilton"",  
                        ""content"": ""Self: Zoolander. Socialite <b>Paris Hilton</b>...""  
                      }  
                    ],  
                    ""cursor"": {  
                      ""pages"": [  
                        {  
                          ""start"": ""0"",  
                          ""label"": 1  
                        },  
                        {  
                          ""start"": ""4"",  
                          ""label"": 2  
                        },  
                        {  
                          ""start"": ""8"",  
                          ""label"": 3  
                        },  
                        {  
                          ""start"": ""12"",  
                          ""label"": 4  
                        }  
                      ],  
                      ""estimatedResultCount"": ""59600000"",  
                      ""currentPageIndex"": 0,  
                      ""moreResultsUrl"": ""http://www.google.com/search?oe=utf8&ie=utf8...""  
                    }  
                  },  
                  ""responseDetails"": null,  
                  ""responseStatus"": 200  
                }";  
            JObject googleSearch = JObject.Parse(googleSearchText);  
            // get JSON result objects into a list  
            IList<JToken> results = googleSearch["responseData"]["results"].Children().ToList();

            // serialize JSON results into .NET objects  
            IList<SearchResult> searchResults = new List<SearchResult>();  
            foreach (JToken result in results)  
            {  
                SearchResult searchResult = JsonConvert.DeserializeObject<SearchResult>(result.ToString());  
                searchResults.Add(searchResult);  
            }

            // Title = <b>Paris Hilton</b> - Wikipedia, the free encyclopedia  
            // Content = [1] In 2006, she released her debut album...  
            // Url = http://en.wikipedia.org/wiki/Paris\_Hilton

            // Title = <b>Paris Hilton</b>  
            // Content = Self: Zoolander. Socialite <b>Paris Hilton</b>...  
            // Url = http://www.imdb.com/name/nm0385296/

**C#操作JSON**

1. .NET对JSON的支持介绍....................................................... 1

(1) 操作Json的DLL介绍..................................................... 1

(2) 使用System.Web.Extensions.dll的限制............... 1

2. JSON序列化和反序列化.......................................................... 1

(1) 使用Newtonsoft.Json.dll实现................................. 1

(2) 使用System.Web.Extensions.dll实现.................... 2

(3) 使用System.Runtime.Serialization.dll实现.... 3

(4) 小结..................................................................................... 5

3. 解析JSON字符串(使用Newtonsoft.Json.dll).............. 5

(1) 使用JsonTextReader类进行只进读取(不常用)...... 5

(2) 使用JArray、JObject、JToken进行读取(常用)... 5

(3) Json时间字符串的处理.................................................. 6

4. 读取Json字符串的技巧......................................................... 6

(1) 使用匿名类........................................................................ 6

(2) 使用索引器........................................................................ 6

**1..NET对JSON的支持介绍**

(1)操作Json的DLL介绍

.NET自身有System.Runtime.Serialization.dll与System.Web.Extensions.dll，使用这两个DLL可以把对象序列化和反序列化成Json数据。

也可以使用第三方的Newtonsoft.Json.dll来操作Json数据，使用它会更方便的操作Json数据，其功能也跟强一些。下载地址：<http://json.codeplex.com/>

(2)使用System.Web.Extensions.dll的限制

要使用System.Web.Extensions.dll必须是在Web项目中，只有在Web项目中才能引用此DLL。

**2.JSON序列化和反序列化**

(1)使用Newtonsoft.Json.dll实现

**代码示例：**

    public class Person

    {

        public string Name;//姓名

        public bool Sex;//性别，是否是男

        public Person(string name, bool sex)

        {

            this.Name = name;

            this.Sex = sex;

        }

        public override string ToString()

        {

            return "姓名：" + this.Name + "\t性别：" + (this.Sex ? "男" : "女");

        }

    }

    public class Programmer : Person

    {

        public List<string> Languages;//编程语言

        public Programmer(string name, bool sex, List<string> languages) : base(name, sex)

        {

            this.Languages = languages;

        }

        public override string ToString()

        {

            StringBuilder str = new StringBuilder();

            str.Append(base.ToString() + "\t编程语言：");

            foreach (string l in this.Languages)

            {

                str.Append(l + " ");

            }

            return str.ToString();

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            string filePath = @"D:\users\lizw\桌面\ObjectJson.txt";

            List<string> languages = null;

            List<Programmer> list = new List<Programmer>();

            languages = new List<string>(new string[] { "C#", "Java" });

            list.Add(new Programmer("李志伟", true, languages));

            languages = new List<string>(new string[] { "C#", "C++" });

            list.Add(new Programmer("Coder2", false, languages));

            languages = new List<string>(new string[] { "C#", "C++", "C", "Java" });

            list.Add(new Programmer("Coder3", true, languages));

            //序列化对象

            string jsonStr = JsonConvert.SerializeObject(list);//将对象转换成json存储

            File.WriteAllText(filePath, jsonStr);

            list.Clear();

            //反序列化对象

            list = JsonConvert.DeserializeObject<List<Programmer>>(jsonStr);

            foreach (Programmer p in list)

            {

                Console.WriteLine(p);

            }

            Console.WriteLine("OK!");

            Console.Read();

        }

    }

**得到的Json文本：**

[{"Languages":["C#","Java"],"Name":"李志伟","Sex":true},

{"Languages":["C#","C++"],"Name":"Coder2","Sex":false},

{"Languages":["C#","C++","C","Java"],"Name":"Coder3","Sex":true}]

**注意：**此种方式Json序列化对象时，只能序列化对象的公有成员。

(2)使用System.Web.Extensions.dll实现

**代码示例：**

    public class Person

    {

        public string Name;//姓名

        public bool Sex;//性别，是否是男

        public Person() { }//必须有此构造方法，否者无法反序列化

        public Person(string name, bool sex)

        {

            this.Name = name;

            this.Sex = sex;

        }

        public override string ToString()

        {

            return "姓名：" + this.Name + "\t性别：" + (this.Sex ? "男" : "女");

        }

    }

    public class Programmer : Person

    {

        public List<string> Languages;//编程语言

        public Programmer() { }//必须有此构造方法，否者无法反序列化

        public Programmer(string name, bool sex, List<string> languages)

            : base(name, sex)

        {

            this.Languages = languages;

        }

        public override string ToString()

        {

            StringBuilder str = new StringBuilder();

            str.Append(base.ToString() + "\t编程语言：");

            foreach (string l in this.Languages)

            {

                str.Append(l + " ");

            }

            return str.ToString();

        }

    }

    public partial class WebForm1 : System.Web.UI.Page

    {

        protected void Page\_Load(object sender, EventArgs e)//Web页面的加载事件

        {

            List<string> languages = null;

            List<Programmer> list = new List<Programmer>();

            languages = new List<string>(new string[] { "C#", "Java" });

            list.Add(new Programmer("李志伟", true, languages));

            languages = new List<string>(new string[] { "C#", "C++" });

            list.Add(new Programmer("Coder2", false, languages));

            languages = new List<string>(new string[] { "C#", "C++", "C", "Java" });

            list.Add(new Programmer("Coder3", true, languages));

            //序列化对象

            JavaScriptSerializer jsonSerialize = new JavaScriptSerializer();

            string jsonStr = jsonSerialize.Serialize(list);//将对象转换成json存储

            Response.Write(jsonStr + "<br>");//前台输出

            //反序列化对象

            list.Clear();

            list = jsonSerialize.Deserialize<List<Programmer>>(jsonStr);

            foreach (Programmer p in list)

            {

                Response.Write(p + "<br>");//前台输出

            }

        }

    }

**得到的Json文本：**

[{"Languages":["C#","Java"],"Name":"李志伟","Sex":true},

{"Languages":["C#","C++"],"Name":"Coder2","Sex":false},

{"Languages":["C#","C++","C","Java"],"Name":"Coder3","Sex":true}]

**注意：**此种方式Json序列化对象时，除了只能序列化对象的公有成员外，被序列化的对象还必须有无参的构造方法才能被反序列化！System.Web.Extensions.dll只能在Web项目中引用，在控制台和WinFrom项目中无法引用！

(3)使用System.Runtime.Serialization.dll实现

**代码示例：**

    [DataContract]//必须申明，否则无法序列化

    public class Person

    {

        [DataMember(Name = "姓名")]//必须申明，否则无法序列化

        private string Name;

        [DataMember(Name = "性别")]//必须申明，否则无法序列化

        private bool Sex;

        public Person(string name, bool sex)

        {

            this.Name = name;

            this.Sex = sex;

        }

        public override string ToString()

        {

            return "姓名：" + this.Name + "\t性别：" + (this.Sex ? "男" : "女");

        }

    }

    [DataContract]

    public class Programmer : Person

    {

        [DataMember(Name = "编程语言")]

        private List<string> Languages;

        public Programmer(string name, bool sex, List<string> languages)

            : base(name, sex)

        {

            this.Languages = languages;

        }

        public override string ToString()

        {

            StringBuilder str = new StringBuilder();

            str.Append(base.ToString() + "\t编程语言：");

            foreach (string l in this.Languages)

            {

                str.Append(l + " ");

            }

            return str.ToString();

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            string filePath = @"D:\users\lizw\桌面\ObjectJson.txt";

            List<string> languages = null;

            List<Programmer> list = new List<Programmer>();

            languages = new List<string>(new string[] { "C#", "Java" });

            list.Add(new Programmer("李志伟", true, languages));

            languages = new List<string>(new string[] { "C#", "C++" });

            list.Add(new Programmer("Coder2", false, languages));

            languages = new List<string>(new string[] { "C#", "C++", "C", "Java" });

            list.Add(new Programmer("Coder3", true, languages));

            //序列化对象

            DataContractJsonSerializer ser =

new DataContractJsonSerializer(typeof(List<Programmer>));

            MemoryStream ms = new MemoryStream();

            ser.WriteObject(ms, list);//将对象转换成json存储

            string jsonStr = Encoding.UTF8.GetString(ms.ToArray());

            File.WriteAllText(filePath, jsonStr);

            //反序列化对象

            list.Clear();

            ms.Position = 0;

            list = (List<Programmer>)ser.ReadObject(ms);

            foreach (Programmer p in list)

            {

                Console.WriteLine(p);

            }

            ms.Dispose();//关闭内存流

            Console.WriteLine("OK!");

            Console.Read();

        }

    }

**得到的Json文本：**

[{"姓名":"李志伟","性别":true,"编程语言":["C#","Java"]},

{"姓名":"Coder2","性别":false,"编程语言":["C#","C++"]},

{"姓名":"Coder3","性别":true,"编程语言":["C#","C++","C","Java"]}]

**注意：**此种方式Json序列化对象时，可以序列化对象的任意成员(包括私有成员)，但是使用此方式必须要在类和成员的定义处加上相应的特性(具体请参考代码)。

(4)小结

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **使用的DLL** | **序列化范围** | **项目范围** | **依赖构造器** | **依赖特性** |
| Newtonsoft.Json.dll | 公有成员 | 任意项目 | 否 | 否 |
| System.Web.Extensions.dll | 公有成员 | Web项目 | 无参构造器 | 否 |
| System.Runtime.Serialization.dll | 所有成员 | 任意项目 | 否 | 是 |

**3.解析JSON字符串(使用Newtonsoft.Json.dll)**

(1)使用JsonTextReader类进行只进读取(不常用)

    class Program

    {

        static void Main(string[] args)

        {

            //Json字符串

            string jsonStr = @"

                [{'Languages':['C#','Java'],'Name':'李志伟','Sex':true},

                {'Languages':['C#','C++'],'Name':'Coder2','Sex':false},

                {'Languages':['C#','C++','C','Java'],'Name':'Coder3','Sex':true}]";

            JsonTextReader json = new JsonTextReader(new StringReader(jsonStr));

            while (json.Read())

            {

                Console.WriteLine(json.Value + "--" + json.TokenType + "--" + json.ValueType);

            }

            Console.WriteLine("OK!");

            Console.Read();

        }

    }

(2)使用JArray、JObject、JToken进行读取(常用)

    class Program

    {

        static void Main(string[] args)

        {

            //Json字符串

            string jsonStr = @"

                [{'Languages':['C#','Java'],'Name':'李志伟','Sex':true},

                {'Languages':['C#','C++'],'Name':'Coder2','Sex':false},

                {'Languages':['C#','C++','C','Java'],'Name':'Coder3','Sex':true}]";

            JArray ja = (JArray)JsonConvert.DeserializeObject(jsonStr);

            foreach (JToken jt in ja)

            {

                JObject jo = (JObject)jt;

                JArray temp = (JArray)jo["Languages"];

                foreach (JToken token in temp)

                {

                    Console.Write(token+" ");

                }

                Console.WriteLine("\t" + jo["Name"] + "\t" + jo["Sex"]);

            }

            Console.WriteLine("OK!");

            Console.Read();

        }

    }

(3)Json时间字符串的处理

    class Program

    {

        static void Main(string[] args)

        {

            DateTime time = DateTime.Now;

            //这里使用自定义日期格式

            IsoDateTimeConverter timeConverter = new IsoDateTimeConverter();

            timeConverter.DateTimeFormat = "北京时间：yyyy-MM-dd HH:mm:ss";

            //序列化时间

            string JsonStr = JsonConvert.SerializeObject(time, timeConverter);

            Console.WriteLine(JsonStr);

            //反序列化时间

            DateTime time2 = JsonConvert.DeserializeObject<DateTime>(JsonStr, timeConverter);

            Console.WriteLine(time2);

            Console.WriteLine("OK!");

            Console.Read();

        }

    }

**4.读取Json字符串的技巧**

(1)使用匿名类

    class Program

    {

        static void Main(string[] args)

        {

            //Json字符串

            string jsonStr = @"{'Languages':['C#','Java'],'Name':'李志伟','Sex':true}";

            //创建匿名类

            var TempClass = new { Languages = new string[0], Name = string.Empty, Sex = false };

            //反序列化

            var O = JsonConvert.DeserializeAnonymousType(jsonStr, TempClass);

            Console.WriteLine(O.Name+"\t"+O.Sex+"\t"+O.Languages[1]);

            Console.Read();

        }

    }

(2)使用索引器

    class Program

    {

        static void Main(string[] args)

        {

            //Json字符串

            string jsonStr = @"{'Languages':['C#','Java'],'Name':'李志伟','Sex':true}";

            //反序列化

            JObject O = JsonConvert.DeserializeObject(jsonStr) as JObject;

            //使用索引器访问

            Console.WriteLine(O["Name"] + "\t" + O["Sex"] + "\t" + O["Languages"][1]);

            Console.Read();

        }

    }