### JToken和JObject有什么区别

### JToken 是其他类的基类：JObject，JArray，JProperty，JValue，JContainer，JConstructor

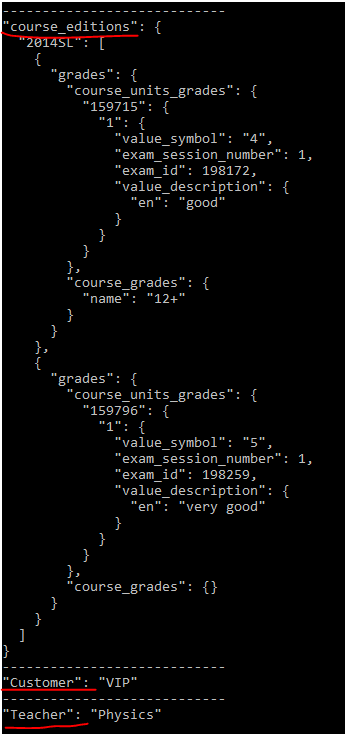
### JToken 也就是说是通用的对象, 而且非常灵活

### public abstract class JToken : IJEnumerable<JToken>, IEnumerable<JToken>, IEnumerable, IJsonLineInfo, ICloneable, IDynamicMetaObjectProvider

### 从JToken的定义和继承可以看到, 甚至可以当做集合来使用：

json = @"{

'course\_editions': {

 '2014SL': [

{

'grades': {

'course\_units\_grades': {

'159715': {

'1': {

'value\_symbol': '4',

'exam\_session\_number': 1,

'exam\_id': 198172,

'value\_description': {'en': 'good'}

}

}

},

'course\_grades': {'name': '12+'}

}

},

{

'grades': {

'course\_units\_grades': {

'159796': {

'1': {

'value\_symbol': '5',

'exam\_session\_number': 1,

'exam\_id': 198259,

'value\_description': {'en': 'very good'}

}

}

},

'course\_grades': {}

}

}

]

},

'Customer': 'VIP'

### }";

JToken jo = JToken.Parse(json);

JEnumerable<JToken> jo99 = jo.Children<JToken>();

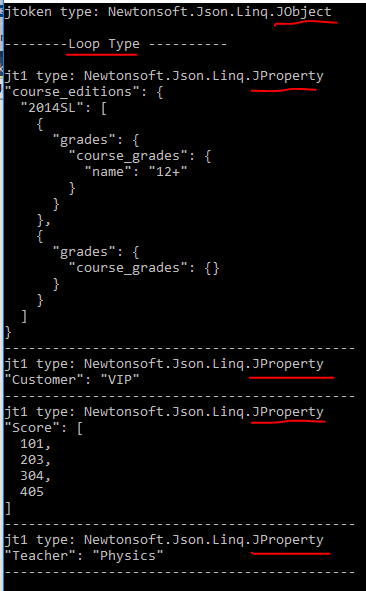
foreach(JToken jjj in jo99)

{

Console.WriteLine("----------------------------");

Console.WriteLine(jjj);

### }

json = @"{

'course\_editions': {

'2014SL': [

{

'grades': {

'course\_grades': {'name': '12+'}

}

},

{

'grades': {

'course\_grades': {}

}

}

]

},

'Customer': 'VIP',

'Score': [101, 203, 304, 405],

'Teacher': 'Physics'

}";

JToken = JToken.Parse(json);

Console.WriteLine("jtoken type: {0}", jtoken.GetType());

Console.WriteLine("\n--------Loop Type ----------\n");

foreach(JToken jt1 in jtoken)

{

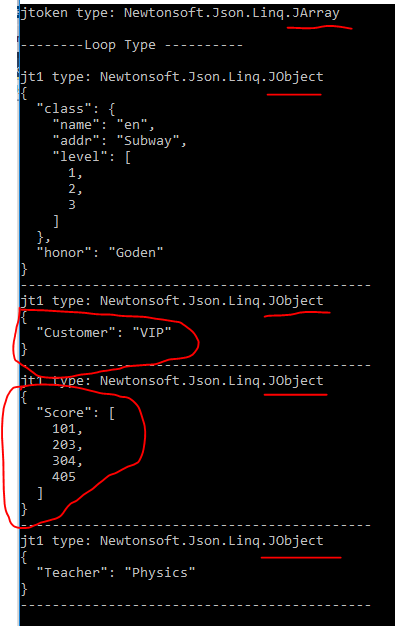
Console.WriteLine("jt1 type: {0}", jt1.GetType());

Console.WriteLine(jt1);

Console.WriteLine("-----------------------------");

### }

### 结构注意：JToken 如果是 JObject 类型，它下属的集合就是 JProperty 集合

json = @"[

{ 'class':

{ name: 'en', addr:'Subway', level:[1,2,3] },

honor: 'Goden'

},

{ 'Customer': 'VIP' },

{ 'Score': [101, 203, 304] },

{ 'Teacher': 'Physics' }

]";

JToken = JToken.Parse(json);

Console.WriteLine("jtoken type: {0}", jtoken.GetType());

Console.WriteLine("\n--------Loop Type ----------\n");

foreach(JToken jt1 in jtoken)

{

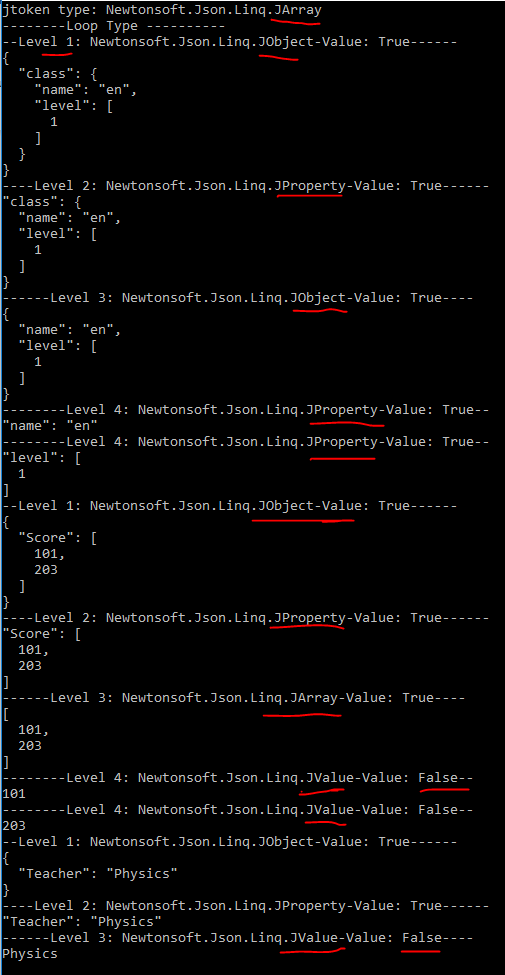
Console.WriteLine("jt1 type: {0}", jt1.GetType());

Console.WriteLine(jt1);

Console.WriteLine("-----------------------------");

### }

### 结构注意：JToken 如果是 JARRAY 类型，它下属的集合就是 JObject 集合

json = @"[

{ 'class':{ name: 'en', level:[1] } },

{ 'Score': [101, 203] },

{ 'Teacher': 'Physics' }

]";

JToken jtoken = JToken.Parse(json);

Console.WriteLine("jtoken type: {0}", jtoken.GetType());

Console.WriteLine("\n--------Loop Type ----------\n");

foreach (JToken jt1 in jtoken)

{

Console.WriteLine($"\n\n---Level 1: {jt1.GetType()}

-Value: {jt1.HasValues}------------");

Console.WriteLine(jt1);

foreach (JToken jt2 in jt1)

{

Console.WriteLine($"---Level 2-----------");

Console.WriteLine(jt2);

foreach(JToken jt3 in jt2)

{

Console.WriteLine($"---Level 3-------");

Console.WriteLine(jt3);

foreach (JToken jt4 in jt3)

{

Console.WriteLine($"---Level 4---");

Console.WriteLine(jt4);

}

}

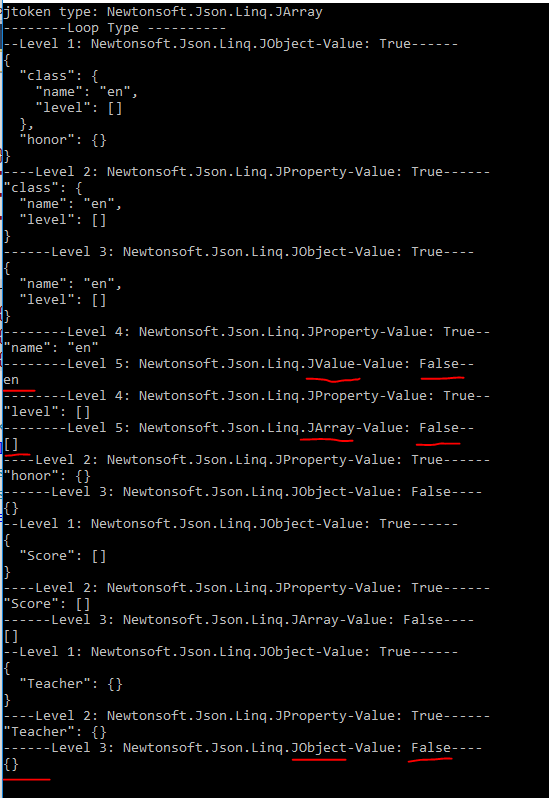
}

}

结构：

foreach ([JObject, JValue] in JArray)

foreach (JProperty in JObject)

JProperty **– 只有唯一的一个子元素**

foreach ([JObject, JArray, JValue] in JProperty) – 只循环一次

JArray: 可能包括 JObject 集合，或者包括 JValue 集合

JObject: 可能包括 JProperty集合，或者包括 {} 空集合

JProperty: 只能包括一个JObject对象，

或者一个 {} 空对象，

或者一个JArray对象，

或者一个JValue对象

json = @"[

{ 'class':{ name: 'en', level:[] }, honor:{} },

{ 'Score': [] },

{ 'Teacher': {} }

]";

bool hasValue = jtoken.HasValues;

JValue.HasValues = false ( always )

JArray = [] HasValues = false

JObject = {} HasValues = false

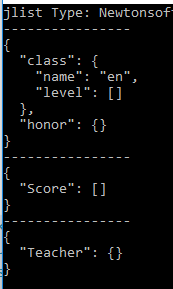
JToken.**Value<>(key) - key 大小写敏感**

JToken = JToken.Parse(json);

JObject jo = jtoken.Value<JObject>(); - 对 JToken 本身进行类型强制转化，

JArray ja = jtoken.Value<JArray>(); - 如果类型不符合则抛出错误

Console.WriteLine("jo Type: {0}", jo.GetType());

----------------------------------------------------------------

json = @"[

{ 'class':{ name: 'en', level:[] }, honor:{} },

{ 'Score': [] },

{ 'Teacher': {} }

]";

JToken = JToken.Parse(json);

IEnumerable<JObject> jlist = jtoken.Values<JObject>();

Console.WriteLine("jlist Type: {0}", jlist.GetType());

foreach(JObject jel in jlist)

{

Console.WriteLine(jel);

}

----------------------------------------------------------------

JArray ja = jtoken.Value<JArray>(); - 这样也可以

----------------------------------------------------------------------------------------------

json = @"{

'course\_editions': {

'2014SL': [],

'2015SL': { year: 2015, month: 12 }

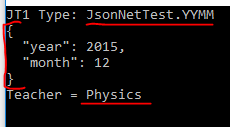
},

'Customer': 'VIP',

'Score': [101, 203, 304, 405],

'Teacher': 'Physics'

}";



YYMM jt1 = jtoken.Value<JToken>("course\_editions")

.Value<JObject>("2015SL")

.ToObject<YYMM>();

Console.WriteLine("JT1 Type: {0}", jt1.GetType());

Console.WriteLine(JToken.FromObject(jt1));

string teacher = jtoken.Value<string>("Teacher");

Console.WriteLine("Teacher = {0}", teacher);

JToken.Value<key> 只能强制转化系统的类型如：

string, int, DateTime, JArray, JObject

不能转化为自定义的数据模型类，如果需要可以使用 ToObject<T>()

{'Teacher': '2018-12-23 13:30'}

DateTime teacher = jtoken.Value<DateTime>("Teacher"); - 和上面的效果是一样的

DateTime teacher = jtoken.Value<JValue>("Teacher").ToObject<DateTime>(); - 和上面的效果是一样的

string json = "{'admin':'this is good', 'hello':'world', 'sdate':'2018-10-15 12:30:35', 'male':'FaLSE',

score:[21,23,35]}";

JObject jobj = JObject.Parse(json);

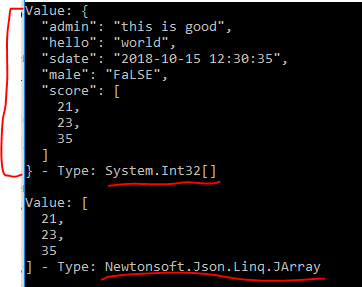
Console.WriteLine("Value: {0} - Type: {1}", jobj.Value<JObject>(),

jobj.Value<JArray>("score").ToObject<int[]>().GetType());

Console.WriteLine();

Console.WriteLine("Value: {0} - Type: {1}", jobj.GetValue("score"), jobj.GetValue("score").GetType());

Console.WriteLine();



JProperty.Value – 返回JPropety 的值，可能是JToken, JObject, JArray, JValue

string json = "{'admin':'this is good', 'hello':'world', 'sdate':'2018-10-15 12:30:35', 'male':'FaLSE', score:[21,23,35]}";

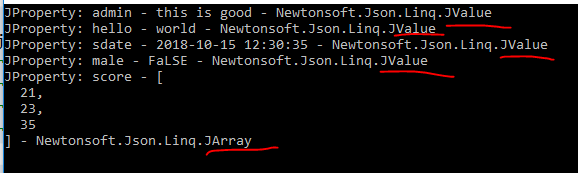
JObject jobj = JObject.Parse(json);

foreach(JProperty jp in jobj.Properties())

{

Console.WriteLine("JProperty: {0} - {1} - {2}", jp.Name, jp.Value, jp.Value.GetType());

}



Console.WriteLine("JProperty: {0} - {1} - {2}", jp.Name, jp.Value.ToObject<string>(), jp.Value.GetType());

JToken.**Values<>(key) – 的困惑, key 应该如此理解**

IEnumerable<int> scores = jtoken.Values<int>("Score"); - 这里虽然不抛出错误

Console.WriteLine("Scores: {0}", scores.GetType());

foreach (var el in scores)

{

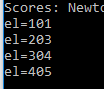
Console.WriteLine("el={0}", el); - 遍历元素时则抛出错误

}



IEnumerable<int> scores = jtoken.Value<JToken>("Score").Values<int>(); - 相当于JProperty.Values<int>()

IEnumerable<int> scores = jtoken.Value<JArray>("Score").Values<int>(); - JArray也是可以的

Console.WriteLine("Scores: {0}", scores.GetType());

foreach (var el in scores)

{

Console.WriteLine("el={0}", el);

}

**key 应该如此理解：**

json = @"{

'course\_editions': {

'2014SL': [20, 22, 33],

'2015SL': { year: 2015, month: 12 }

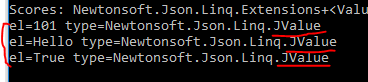
},

'Customer': 'VIP',

'Score': [{id: 101},{id: 'Hello'}, {ID: 304}, {NO: 405}, {id: true}],

'Teacher': '2018-10-15'

}";



IEnumerable<object> scores = jtoken.Value<JArray>("Score").Values<object>("id");

Console.WriteLine("Scores: {0}", scores.GetType());

foreach (var el in scores)

{

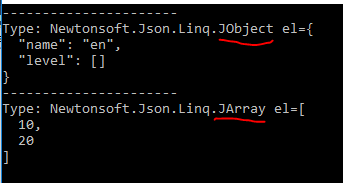
Console.WriteLine("el={0} type={1}", el, el.GetType());

}

//如果指定类型，出现不能转化的情况，则抛出错误

IEnumerable<int> scores = jtoken.Value<JArray>("Score").Values<int>("id");

--- 另外一个案例 --------------------------------

json = @"[

{ 'class':{ name: 'en', level:[] }, honor:{} },

{ 'class': [10,20] },

{ 'Score': [] },

{ 'Teacher': {} }

]";

var jos = jtoken.Values<JToken>("class");

foreach (JToken el in jos) {

Console.WriteLine("----------------------");

Console.WriteLine("Type: {0} el={1}",el.GetType(), el);

}

总结：

JToken.Value<T>(Key) – 当前 JProperty(Key) 所对应的值，可能是 JObject, JArray, JValue 或者 null

var jo = jtoken.Value<JToken>("fdddf"); - key 不存在，则抛出错误

Error: Object reference not set to an instance of an object.

------------------------------------------------------------------------

json = @"{

'course\_editions': {

'2014SL': [20, 22, 33],

'2015SL': { year: 2015, month: 12 }

},

'Customer': 'VIP',

'Score': [{id: 101},{id: 300}, {ID: 304}, {NO: 405}, {id: 308}],

'Teacher': {}

}";

JToken = JToken.Parse(json);

var jo = jtoken.Value<JToken>("Teacher");

Console.WriteLine("el={0} type={1} hasValue:{2}",jo, jo.GetType(), jo.HasValues);



string jo = jtoken.Value<JToken>("Customer").Value<string>();

string jo1 = jtoken.Value<string>("Customer");

Console.WriteLine("el={0} type={1} hasValue:{2}",jo, jo.GetType(), jo1);

int? jo1 = jtoken.Value<int?>("Customer"); - 'Customer': null, 'Customer': ,

JToken.Values<T>(Key) -应用于JArray, 如果是JObject集合，Key是JPropertys[Key] 返回对应值

Values<T>(Key) 很容易跟Value<T>(Key) 搞混淆

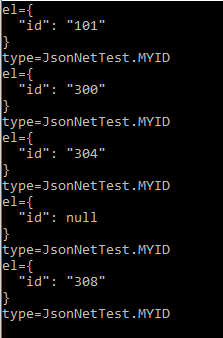
看懂下面的例子，就明白区别了

'Score': [101, 203, 304, 405]

IEnumerable<int> scores = jtoken.Value<JToken>("Score").Values<int>(); - Values<T> 返回IEnumerable<T>

'Score': [{id: 101},{id: 300}, {ID: 304}, {NO: 405}, {id: 308}]

IEnumerable<object> scores = jtoken.Value<JArray>("Score").Values<object>("id");

JToken.ToObject<>() – 比较好理解，转化为强类型

'Score': [{id: 101},{id: 300}, {ID: 304}, {NO: 405}, {id: 308}]

public class MYID

{

public string id { get; set; }

}

JToken = JToken.Parse(json);

IList<MYID> jos = jtoken.Value<JToken>("Score").ToObject<IList<MYID>>();

foreach (MYID jo in jos)

{

Console.WriteLine("el={0}\ntype={1}",

JObject.FromObject(jo).ToString(),

jo.GetType());

}

JToken.SelectToken(jpath)

JToken.SelectTokens(jpath)

json = @"{

'course\_editions': {

'2014SL': [20, 22, 33],

'2015SL': { year: 2015, month: 12 }

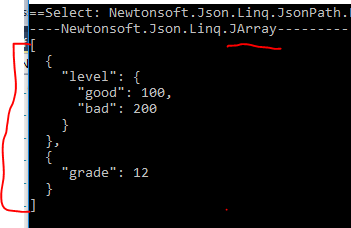
},

'Customer': 'VIP User',

'Score': [{id: 101},{id: 300}, {ID: 304}, {NO: 405}, {id: 308}],

'Teacher': [{ level: {good:100, bad:200}},{ grade:12}]

}";

------------------------------------------------------------

JObject jtoken = JObject.Parse(json);

IEnumerable<JToken> jts = jtoken.SelectTokens("Teacher");

Console.WriteLine($"==Select: {jts.GetType()} ===");

foreach(JToken jt in jts)

{

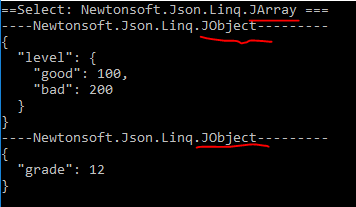
Console.WriteLine($"----{jt.GetType()}---------");

Console.WriteLine(jt);

}

--------------------------------------------------------------

JToken jts = jtoken.SelectToken("Teacher");



---------------------------------------------------------------

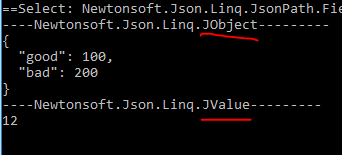
IEnumerable<JToken> jts = jtoken.SelectTokens("Teacher.\*"); - 没有返回具体的对象



----------------------------------------------------------------------------------

'Teacher': { level: {good:100, bad:200}, grade:12 }

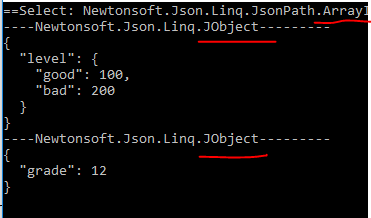
IEnumerable<JToken> jts = jtoken.SelectTokens("Teacher.\*");



----------------------------------------------------------------

'Teacher': [{ level: {good:100, bad:200}},{ grade:12}]

IEnumerable<JToken> jts = jtoken.SelectTokens("Teacher[\*]");



-----------------------------------------------------------------

json = @"{

'course\_editions': {

'2014SL': [20, 22, 33],

'2015SL': { year: 2015, month: 12, price:20 }

},

'Customer': 'VIP User',

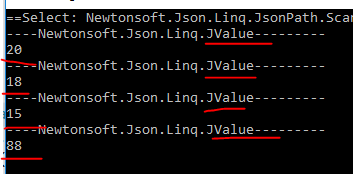
'price': 18,

'Score': [{id: 101},{id: 300}, {ID: 304}, {NO: 405}, {id: 308}, {price:15}],

'Teacher': [{ level: {good:100, bad:200}, price: 88},{ grade:12}]

}";

IEnumerable<JToken> jts = jtoken.SelectTokens("$..price"); - 扁平化获取 price 列表



JToken jts = jtoken.SelectToken("$..price"); - 因为返回多个JToken, 所以会抛出错误

Error: Path returned multiple tokens.

-----------------------------------------------------------------

JArray jts = jtoken.SelectToken("$.Score").Value<JArray>();

Console.WriteLine($"==Select: {jts.GetType()} ===");

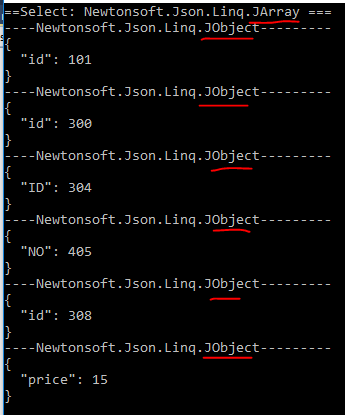
foreach(JToken jt in jts)

{

Console.WriteLine($"----{jt.GetType()}---------");

Console.WriteLine(jt);

}



JObject，JArray - 用法和JToken的使用原理一样

json = @"[

{ 'class':{ name: 'en', level:[] }, honor:{} },

{ 'Score': [] },

{ 'Teacher': {} }

]";

JObject jobj = JObject.Parse(json);

如果不是JSON Object 则会抛出错误

类名 说明  
JObject  用于操作JSON对象  
JArray    用语操作JSON数组  
JValue   表示数组中的值  
JProperty 表示对象中的属性,以"key/value"形式  
JToken  用于存放Linq to JSON查询后的结果

JObject 是 JContainer 的子类，而 JContainer 又是 JToken 的子类。这就好像说“人”是动物的子类，而动物是生物的子类。显然，使用 Jtoken 的地方，不仅仅要能让 JObject 对象实例使用，还要兼容 JArray 等等其它类型的对象。此时用 JObject 就太“小气”了，不够准确。

**JSONPath详解**

<https://support.smartbear.com/alertsite/docs/monitors/api/endpoint/jsonpath.html>

**JSONPath Syntax**

 Last modified on November 1, 2018

JSONPath is a query language for JSON, similar to XPath for XML. AlertSite [API endpoint monitors](https://support.smartbear.com/alertsite/docs/monitors/api/endpoint/create.html) let you use JSONPath in [assertions](https://support.smartbear.com/alertsite/docs/monitors/api/endpoint/assertions.html) to specify the JSON fields that need to be verified.

JSONPath Notation

A JSONPath expression specifies a path to an element (or a set of elements) in a JSON structure. Paths can use the dot notation:

$.store.book[0].title

or the bracket notation:

$['store']['book'][0]['title']

The leading $ represents the root object or array and can be omitted. For example, $.foo.bar and foo.bar are the same, and so are $[0].status and [0].status.

Other syntax elements are described below.

| **Expression** | **Description** |
| --- | --- |
| $ | The root object or array. |
| .*property* | Selects the specified property in a parent object. |
| ['*property*'] | Selects the specified property in a parent object. Be sure to put single quotes around the property name.  **Tip:**Use this notation if the property name contains special characters such as spaces, or begins with a character other than A..Za..z\_. |
| [*n*] | Selects the *n*-th element from an array. Indexes are 0-based. |
| [*index1*,*index2*,*…*] | Selects array elements with the specified indexes. Returns a [list](https://support.smartbear.com/alertsite/docs/monitors/api/endpoint/jsonpath.html#multiple). |
| ..*property* | Recursive descent: Searches for the specified property name recursively and returns an array of all values with this property name. Always returns a [list](https://support.smartbear.com/alertsite/docs/monitors/api/endpoint/jsonpath.html#multiple), even if just one property is found. |
| \* | Wildcard selects all elements in an object or an array, regardless of their names or indexes. For example, address.\* means all properties of the addressobject, and book[\*] means all items of the book array. |
| [*start*:*end*] [*start*:] | Selects array elements from the *start* index and up to, but not including, *end*index. If *end* is omitted, selects all elements from *start* until the end of the array. Returns a [list](https://support.smartbear.com/alertsite/docs/monitors/api/endpoint/jsonpath.html#multiple). |
| [:*n*] | Selects the first *n* elements of the array. Returns a [list](https://support.smartbear.com/alertsite/docs/monitors/api/endpoint/jsonpath.html#multiple). |
| [*-n*:] | Selects the last *n* elements of the array. Returns a [list](https://support.smartbear.com/alertsite/docs/monitors/api/endpoint/jsonpath.html#multiple). |
| [?(*expression*)] | [Filter expression](https://support.smartbear.com/alertsite/docs/monitors/api/endpoint/jsonpath.html#filters). Selects all elements in an object or array that match the specified filter. Returns a [list](https://support.smartbear.com/alertsite/docs/monitors/api/endpoint/jsonpath.html#multiple). |
| [(*expression*)] | Script expressions can be used instead of explicit property names or indexes. An example is [(@.length-1)] which selects the last item in an array. Here, length refers to the length of the current array rather than a JSON field named length. |
| @ | Used in filter expressions to refer to the current node being processed. |

Notes:

* JSONPath expressions, including property names and values, are **case-sensitive**.
* Unlike XPath, JSONPath does not have operations for accessing parent or sibling nodes from the given node.

Filters

Filters are logical expressions used to filter arrays. An example of a JSONPath expression with a filter is

$.store.book[?(@.price < 10)]

where @ represents the current array item or object being processed. Filters can also use $ to refer to the properties outside of the current object:

$.store.book[?(@.price < $.expensive)]

An expression that specifies just a property name, such as [?(@.isbn)], matches all items that have this property, regardless of the value.

Additionally, filters support the following operators:

| **Operator** | **Description** |
| --- | --- |
| == | Equals to. 1 and '1' are considered equal. String values must be enclosed in single quotes (not double quotes): [?(@.color=='red')]. |
| != | Not equal to. String values must be enclosed in single quotes. |
| > | Greater than. |
| >= | Greater than or equal to. |
| < | Less than. |
| <= | Less than or equal to. |
| =~ | Match a [JavaScript regular expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions). For example, [?(@.description =~ /cat.\*/i)] matches items whose description starts with *cat* (case-insensitive).  **Note:**Not supported at [locations that use Ready! API 1.1](https://support.smartbear.com/alertsite/docs/monitors/api/readyapi-versions.html). |
| ! | Use to negate a filter: [?(!@.isbn)] matches items that do not have the isbn property.  **Note:**Not supported at [locations that use Ready! API 1.1](https://support.smartbear.com/alertsite/docs/monitors/api/readyapi-versions.html). |
| && | Logical AND, used to combine multiple filter expressions:  [?(@.category=='fiction' && @.price < 10)] |
| || | Logical OR, used to combine multiple filter expressions:  [?(@.category=='fiction' || @.price < 10)]  **Note:**Not supported at [locations that use Ready! API 1.1](https://support.smartbear.com/alertsite/docs/monitors/api/readyapi-versions.html). |

Examples

For these examples, we will use a modified version of JSON from <http://goessner.net/articles/JsonPath/index.html#e3>:

{  
  "store": {  
    "book": [  
      {  
        "category": "reference",  
        "author": "Nigel Rees",  
        "title": "Sayings of the Century",  
        "price": 8.95  
      },  
      {  
        "category": "fiction",  
        "author": "Herman Melville",  
        "title": "Moby Dick",  
        "isbn": "0-553-21311-3",  
        "price": 8.99  
      },  
      {  
        "category": "fiction",  
        "author": "J.R.R. Tolkien",  
        "title": "The Lord of the Rings",  
        "isbn": "0-395-19395-8",  
        "price": 22.99  
      }  
    ],  
    "bicycle": {  
      "color": "red",  
      "price": 19.95  
    }  
  },  
  "expensive": 10  
}

In all these examples, the leading $. is optional and can be omitted.

| **Expression** | **Meaning** |
| --- | --- |
| $.store.\* | All direct properties of store (not recursive). |
| $.store.bicycle.color | The color of the bicycle in the store.  Result: red |
| $.store..price $..price | The prices of all items in the store.  Result: [8.95, 8.99, 22.99, 19.95] |
| $.store.book[\*] $..book[\*] | All books in the store. |
| $..book[\*].title | The titles of all books in the store.  Result: [Sayings of the Century, Moby Dick, The Lord of the Rings] |
| $..book[0] | The first book.  Result: [{"category":"reference","author":"Nigel Rees","title":"Sayings of the Century","price":8.95}] |
| $..book[0].title | The title of the first book.  Result: Sayings of the Century |
| $..book[0,1].title $..book[:2].title | The titles of the first two books.  Result: [Sayings of the Century, Moby Dick] |
| $..book[-1:].title $..book[(@.length-1)].title | The title of the last book.  Result: [The Lord of the Rings]  The result is a [list](https://support.smartbear.com/alertsite/docs/monitors/api/endpoint/jsonpath.html#multiple), because [*-n*:] always returns lists. |
| $..book[?(@.author=='J.R.R. Tolkien')].title | The titles of all books by *J.R.R. Tolkien* (exact match, case-sensitive).  Result: [The Lord of the Rings]  The result is a list, because filters always return lists. |
| $..book[?(@.isbn)] | All books that have the isbn property. |
| $..book[?(!@.isbn)] | All books without the isbn property. |
| $..book[?(@.price < 10)] | All books cheaper than 10. |
| $..book[?(@.price > $.expensive)] | All expensive books. |
| $..book[?(@.author =~ /.\*Tolkien/i)] | All books whose author name ends with *Tolkien* (case-insensitive). |
| $..book[?(@.category == 'fiction' || @.category == 'reference')] | All fiction and reference books. |
| $..\* | All members of the JSON structure beneath the root (child objects, individual property values, array items), combined into an array. |

Considerations for JSONPath Expressions That Return Multiple Elements

JSONPath queries can return not just a single element, but also a list of matching elements. For example, given this JSON:

{  
  "name": "Rose Kolodny",  
  "phoneNumbers": [  
    {  
      "type": "home",  
      "number": "954-555-1234"  
    },  
    {  
      "type": "work",  
      "number": "754-555-5678"  
    }  
  ]  
}

the JSONPath expression

phoneNumbers[\*].number

returns a list containing two phone numbers:

[954-555-1234, 754-555-5678]

Note that this is not a JSON array, it is just a comma-separated list of items where [ ] indicates the beginning and end of the list.

When using “equals” assertions against a list of matches, specify a list of expected values enclosed in [ ] and separated by a comma and one space:

[apples, 15, false, ["foo","bar"], {"status":"ok"}]

Standalone strings (like apples) should not have enclosing quotes, unless the quotes are part of the value.

[[https://support.smartbear.com/alertsite/docs/_assets/commonImages/plus-btn.gif](javascript:ShowExample(%22IDKM5NPAZ00R45OMSDAJ5HR3YDOHYNTCRMLPPMOZNB0QBO3T3ZEEKK_id%22,%20%22IDKM5NPAZ00R45OMSDAJ5HR3YDOHYNTCRMLPPMOZNB0QBO3T3ZEEKK_div%22))Example](javascript:ShowExample(%22IDKM5NPAZ00R45OMSDAJ5HR3YDOHYNTCRMLPPMOZNB0QBO3T3ZEEKK_id%22,%20%22IDKM5NPAZ00R45OMSDAJ5HR3YDOHYNTCRMLPPMOZNB0QBO3T3ZEEKK_div%22))

Values that are JSON arrays and objects keep inner quotes, but are minified with no spaces between their items: ["foo","bar"], not [ "foo" , "bar" ].

**C# switch … case**

switch(jt3.GetType())

{

case typeof(JArray):

break;

case typeof(JObject):

break;

}

无法直接使用类型作为条件值，因为 case 需要一个常量值

switch(jt3.GetType().ToString())

{

case (1+30).ToString(): - 这样会出错，因为不是常量， ToString() 需要运算

break;

case "J" + "Object": - 这样虽然可以，但是无法准确获取到类型的字符串

Console.WriteLine("This is JObject");

break;

case "JArray": - 这样虽然可以，但是无法准确获取到类型的字符串

Console.WriteLine("This is JArray");

break;

}

根据上面的缺陷：可以加以改进而达到目的

public static class DHelper

{

public static string GetTypeString(this IDictionary<Type, string> dict, Type type)

{

if (dict.ContainsKey(type))

return dict[type];

else

return string.Empty;

}

}

IDictionary<Type, string> jtype = new Dictionary<Type, string>() - 使用IDictionary作为映射

{

{typeof(JArray), "JArray"},

{typeof(JObject), "JObject"},

};

switch (jtype.GetTypeString(jt3.GetType()))

{

case "JObject":

Console.WriteLine("This is JObject");

break;

case "JArray":

Console.WriteLine("This is JArray");

break;

}

最直接的方法是：

if (jt3.GetType() == typeof(JArray)) Console.WriteLine("This is JArray");

string json = "{'admin':'this is good', 'hello': {vv:'2018-12-25'}, 'sdate':'2018-10-15 12:30:35', 'male':'FaLSE', score:[21,23,35]}";

JObject jobj = JObject.Parse(json);

Console.WriteLine(

jobj.Value<string>("admin") + " : " + 效果一样

jobj.Value<JToken>("admin").Value<string>()

);

Console.WriteLine(

jobj.Value<JToken>("hello").Value<DateTime>("vv") + " : " + 效果一样

jobj.SelectToken("$.hello").Value<DateTime>("vv")

);

string json = "{'return': { 'case':

[{id:{value:800}, name:'Good'}, {id:{value:900}, name:'better'}]

}}";

JObject jobj = JObject.Parse(json);

var elist = jobj.SelectToken("$.return.case").Values<JObject>("id").Values<int>("value");

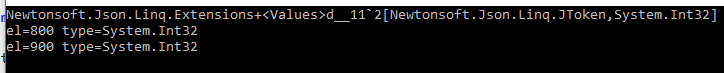
Console.WriteLine(elist.GetType());

foreach (var el in elist)

{

Console.WriteLine($"el={el} type={el.GetType()}");

}



string json =

"{'return':

{'case':[

{id:{value:800}, name:'Good'},

{id:{value:900}, name:'better'}

]

}

}";

JObject jobj = JObject.Parse(json);

var elist = jobj.SelectTokens("$.return.case[\*].id.value").Values<int>();

Console.WriteLine(elist);

foreach (var el in elist)

{

Console.WriteLine($"el={el} type={el.GetType()}");

}

