JQuery工具方法

工具方法与实例方法差别

实例方法：$('div').html() 以一个实例去调用jquery上的方法进行操作的方法称为实例方法
工具方法：$.type(arr) 无需以一个实例去调用的方法，实例只作为一个被处理的对象
jQuery上的工具方法是定义在jQuery函数里的属性(方法)，因为函数也是对象，只要是对象，就是属性和方法的集合，而实例方法都是定义在jQuery原型上的方法

$.type()

$.type()可以判断参数的数据类型，且所返回的类型字符串是唯一的，函数即function，数组即array

```
var arr = [1,2,3];  console.log($.type(123))  输出结果：array
```

$.trim()

$.trim()可以消除字符串两边的空格

```
var str = '  dg  ';  console.log('|' + $.trim(str) + '|')  输出结果 |dg|
```

$.proxy()

$.proxy()可以改变函数内部的this指向，改变后返回一个新的函数，该新的函数的this指向，为我们设定的对象

```
function show(){console.log(this)}
var obj = {age: 25}
var fn = $.proxy(show,obj)
fn()                                    输出结果：{age: 25}
```

既然$.proxy()方法会返回一个新的函数，那么代表可以往这个新返回的函数内部传参

```
function show(a,b){console.log(this); console.log(a + b)}
var obj = {age: 25}
$.proxy(show,obj)(1,2)
输出结果：{age: 25}    3
```

$.proxy()是一个实现了柯里化的方法，也就意味着我们可以在一定程度上决定每次传参的个数，需要注意的是仅仅是一定程度上，$.proxy()并不是完全自由化的柯里化，在传参的格式上依旧有一定的限制
$.proxy()可接受两次传参，但第一个参数与第二个参数也就是目标函数与this指向不能分开传入，而剩下的参与实际运算的，不涉及结构的参数可以分开传入

```
$.proxy(show,obj,1)(2)
$.proxy(show,obj)(1,2)
$.proxy(show,obj,1,2)()
```

$.proxy()在单变量开发中的应用

```
var obj = {
    init: function(){
        this.bindevent()
    },
    bindevent:function(){
        $('div').click(this.cb)
    },
    cb:function(a,b){
        console.log(this)
        //希望this为obj，实际this为div元素
    }
}
obj.init()
```

```
var obj = {
    init: function(){
        this.bindevent()
    },
    bindevent:function(){
        $('div').click($.proxy(this.cb,this,1,2))
    },
    cb:function(a,b){
        console.log(this)  //this为obj
    }
}
obj.init()
```

$.noConflict()

$.noConflict() 可以移交$权限，防止变量名冲突，假如html页面同时存在两个JS库(其中一个为jQuery)，同时另外一个JS库中也有变量$，并且jQuery作为最后一个引入的库，那么它的$必然覆盖前一个JS库中的$，在这种情况下，假想执行第一个库中的$方法的话是不可能实现的，此时我们便可以通过$.noConflict()的方法把jQuery中对变量$的绑定让出，那么前一个JS库便重新恢复了对自己库中的$的使用，其次假如用一个新的变量(a)去接收$.noConflict()的返回值的话，此时jQuery中的原$方法的变量名将变为该变量(a)，那么当再次调用jQuery时便可以通过a('div')的方式执行

在$.noConflict()不传参或参数为false时，$需要已经被移交，但依然可以通过jQuery.()的方式进行操作，
假如$.noConflict(true)的话，$将连同jQuery被一同移交，那么此时我们便可以调用上一个不同版本的jQuery中的方法
需要注意的是在执行$.noConflict()务必用一个新的变量去接收它的返回值，否则当前jQuery将无法再被调用。

```
<script>
    function $ (a){console.log(a + 1)}
</script>
<script src="./jquery.js"></script>
<script>
    var wjquery = $.noConflict(false);
    console.log(wjquery('div'))    // w.fn.init  [div, prevObject: w.fn.init(1)]

    console.log(jQuery('div'))     // w.fn.init  [div, prevObject: w.fn.init(1)]

    console.log($('div'))           // div1

    wjquery.noConflict(true);
    console.log(jQuery('div'))     // TypeError: jQuery is not a function

</script>
```

$.parseJSON()

$.parseJSON()可以把字符串格式的JSON转换为JSON对象

```
var obj = {"aa":123,"bb":"dg"}
var str = JSON.stringify(obj);  返回结果："{"aa":123,"bb":"dg"}"
console.log($.parseJSON(str))  打印结果：  {"aa":123,"bb":"dg"}
```

$.makeArray()

$.makeArray() 可以把一个类数组转化为数组

```
var obj = {0:123,1:"dg",length:2}
var arr = $.makeArray(obj)
```

$.extend() $.fn.extend()

$.extend() 方法扩展，提供新的jQuery工具方法
$.fn.extend() 方法扩展，提供新的jQuery实例方法 记住是实例方法 不是工具方法

区别：$.extend() 与 $.fn.extend() 的区别在于$.extend()是把新的方法添加至jQuery函数上的方法，而 $.fn.extend()是把新的方法添加至jQuery原型上的方法，该方法的调用需要通过一个dom元素或变量实行，如$('div')

```
$.extend({
    jx: function(){
        return 'yangjunxing'
    }
})
console.log($.jx())
```

```
$.fn.extend({
    jx: function(){
        return 'yangjunxing'
    }
})
console.log($('div').jx())
```

既然可以可以通过extend为jQuery函数或jQuery原型身上添加方法，并且方法必须以对象的形式传入，那么也就是说extend方法具有一个添加合并的功能，既然如此，同样我们可以通过extend()将两个对象合并

```
var obj1 = { person1: 'yjx', number: 1 }
var obj2 = { person2: 'yhj', number: 2 }
$.fn.extend(obj1,obj2)
console.log(obj1)       //{person1: "yjx", number: 2, person2: "yhj"}
```

需要注意的是假如obj2的属性中有引用值的话，那么拷贝至obj1中的该引用值类型的属性将与obj2中的同一名字的属性指向同一个房间，也就是说$.fn.extend(obj1,obj2)不过是浅拷贝，实现深拷贝的方法是$.fn.extend(true,obj1,obj2)

$.Callbacks()

$.Callbacks()能返回一个函数对象，但此时的对象没有实际内容，我们必须通过add()并且以回调函数的形式往里传入方法，函数对象的内部才能拥有方法，同时可以通过fire()的方式执行函数对象内部的所有函数
$.Callbacks()能接收一个字符串作为参数，并只有四种形式once memory unique stopOnFalse

$.Callbacks('once') 只执行一次fire()
$.Callbacks('unique') 相同的函数只执行一次
$.Callbacks('memory') 即使add()执行在fire()之后，新add()的函数也会被执行
$.Callbacks('stopOnFalse') 此时fire()的话，将会只执行至有return false的函数

```
function fn1(){ console.log('fn1')}
function fn2(){ console.log('fn2')}
function fn3(){ console.log('fn3')}
var cb = $.Callbacks('once memory');
cb.add(fn1,fn2); cb.fire(); cb.add(fn3); cb.fire();
//fn1 fn2 fn3
```

$.Deferred()

$.Deferred()主要用于发送网络请求时，是一个具有延迟特性的对象，同时它也相当于有状态的$.Callbacks(),
通过$.Deffered()后会返回一个函数对象该函数对象内部存有三种函数方法，分别是notity代表数据发送中，reject代表数据发送失败，resolve代表数据发送成功，其次这三个函数方法对应着三个回调函数(progress, fail, done)，以用于执行当不同状态时应该执行怎样的代码

```
function aa() {
    var dtd = $.Deferred();
    setInterval(function(){dtd.notify()},500)
    setInterval(function(){dtd.reject()},1000)
    setInterval(function(){dtd.resolve()},2000)
    return dtd.promise()
}
var cb = aa();
cb
.done(function(){console.log('ok')})
.fail(function(){console.log('error')})
.progress(function(){console.log('doing')})
```

reject() 和resolve()只会被执行其一, 因为执行成功与执行失败只能有一个，而假如执行了notify的话，Deferred对象会继续等待被执行，直至执行到reject或resolve为止
避免恶意修改Deferred对象的状态，应该return dtd.promise()那么所return出来的deferred对象，将是只读的对象，在函数外部进行 notify, reject, resolve操作将报错

$.when()

$.when()只有返回的全部Defferred对象都为resolve状态时，才会执行done回调函数，否则都会执行fail回调函数

```
function aa() {
    var dtd = $.Deferred();
    dtd.resolve();
    return dtd.promise()
}
function bb() {
    var dtd = $.Deferred();
    dtd.resolve();
    return dtd.promise()
}
var cb = aa(); var cb2 = bb()
$.when(cb,cb2)
.done(function(){console.log('ok')})
.fail(function(){console.log('error')})
.progress(function(){console.log('doing')})
```