08 underscore

前面我们已经讲过了，JavaScript是函数式编程语言，支持高阶函数和闭包。函数式编程非常强大，可以写出非常简洁的代码。例如`Array`的`map()`和`filter()`方法：

```
'use strict';
var a1 = [1, 4, 9, 16];
var a2 = a1.map(Math.sqrt); // [1, 2, 3, 4]
var a3 = a2.filter((x) => { return x % 2 === 0; }); // [2,
4]
```

现在问题来了，`Array`有`map()`和`filter()`方法，可是Object没有这些方法。此外，低版本的浏览器例如IE6～8也没有这些方法，怎么办？

方法一，自己把这些方法添加到`Array.prototype`中，然后给`Object.prototype`也加上`mapObject()`等类似的方法。

方法二，直接找一个成熟可靠的第三方开源库，使用统一的函数来实现`map()`、`filter()`这些操作。

我们采用方法二，选择的第三方库就是underscore。

正如jQuery统一了不同浏览器之间的DOM操作的差异，让我们可以简单地对DOM进行操作，underscore则提供了一套完善的函数式编程的接口，让我们更方便地在JavaScript中实现函数式编程。

jQuery在加载时，会把自身绑定到唯一的全局变量`$`上，underscore与其类似，会把自身绑定到唯一的全局变量`_`上，这也是为啥它的名字叫underscore的原因。

用underscore实现`map()`操作如下：

```
'use strict';
_.map([1, 2, 3], (x) => x * x); // [1, 4, 9]
```

咋一看比直接用`Array.map()`要麻烦一点，可是underscore的`map()`还可以作用于Object：

```
'use strict';
_.map({ a: 1, b: 2, c: 3 }, (v, k) => k + '=' + v); //
['a=1', 'b=2', 'c=3']
```

后面我们会详细介绍underscore提供了一系列函数式接口。

Collections

underscore为集合类对象提供了一致的接口。集合类是指Array和Object，暂不支持Map和Set。

map/filter

和`Array`的`map()`与`filter()`类似，但是underscore的`map()`和`filter()`可以作用于Object。当作用于Object时，传入的函数为`function (value, key)`，第一个参数接收value，第二个参数接收key：

```
'use strict';

var obj = {
    name: 'bob',
    school: 'No.1 middle school',
    address: 'xueyuan road'
};
var upper = _.map(obj, function (value, key) {
    return ???;
});
console.log(JSON.stringify(upper));
```

你也许会想，为啥对Object作`map()`操作的返回结果是`Array`？应该是Object才合理啊！把`_.map`换成`_.mapObject`再试试。

every / some

当集合的所有元素都满足条件时，`_.every()`函数返回`true`，当集合的至少一个元素满足条件时，`_.some()`函数返回`true`：

```
'use strict';
// 所有元素都大于0?
_.every([1, 4, 7, -3, -9], (x) => x > 0); // false
// 至少一个元素大于0?
_.some([1, 4, 7, -3, -9], (x) => x > 0); // true
```

当集合是Object时，我们可以同时获得value和key：

```
'use strict';
var obj = {
    name: 'bob',
    school: 'No.1 middle school',
    address: 'xueyuan road'
};
// 判断key和value是否全部是小写:
var r1 = _.every(obj, function (value, key) {
    return ???;
});
var r2 = _.some(obj, function (value, key) {
    return ???;
```

```
});
console.log('every key-value are lowercase: ' + r1 +
'\nsome key-value are lowercase: ' + r2);
```

max / min

这两个函数直接返回集合中最大和最小的数：

```
'use strict';
var arr = [3, 5, 7, 9];
_.max(arr); // 9
_.min(arr); // 3

// 空集合会返回-Infinity和Infinity，所以要先判断集合不为空：
_.max([])
-Infinity
_.min([])
Infinity
```

注意，如果集合是Object，`max()`和`min()`只作用于value，忽略掉key：

```
'use strict';
_.max({ a: 1, b: 2, c: 3 }); // 3
```

groupBy

`groupBy()`把集合的元素按照key归类，key由传入的函数返回：

```
'use strict';

var scores = [20, 81, 75, 40, 91, 59, 77, 66, 72, 88, 99];
var groups = _.groupBy(scores, function (x) {
    if (x < 60) {
        return 'C';
    } else if (x < 80) {
        return 'B';
    } else {
        return 'A';
    }
});
// 结果:
// {
//   A: [81, 91, 88, 99],
//   B: [75, 77, 66, 72],
//   C: [20, 40, 59]
// }
```

可见`groupBy()`用来分组是非常方便的。

shuffle / sample

`shuffle()`用洗牌算法随机打乱一个集合：

```
'use strict';
// 注意每次结果都不一样：
_.shuffle([1, 2, 3, 4, 5, 6]); // [3, 5, 4, 6, 2, 1]
```

`sample()`则是随机选择一个或多个元素：

```
'use strict';
// 注意每次结果都不一样：
// 随机选1个：
_.sample([1, 2, 3, 4, 5, 6]); // 2
// 随机选3个：
_.sample([1, 2, 3, 4, 5, 6], 3); // [6, 1, 4]
```

更多完整的函数请参考underscore的文档：http://underscorejs.org/#collections

Arrays

underscore为`Array`提供了许多工具类方法，可以更方便快捷地操作`Array`。

first / last

顾名思义，这两个函数分别取第一个和最后一个元素：

```
'use strict';
var arr = [2, 4, 6, 8];
_.first(arr); // 2
_.last(arr); // 8
```

flatten

`flatten()`接收一个`Array`，无论这个`Array`里面嵌套了多少个`Array`，`flatten()`最后都把它们变成一个一维数组：

```
'use strict';

_.flatten([1, [2], [3, [[4], [5]]]]); // [1, 2, 3, 4, 5]
```

zip / unzip

`zip()`把两个或多个数组的所有元素按索引对齐，然后按索引合并成新数组。例如，你有一个`Array`保存了名字，另一个`Array`保存了分数，现在，要把名字和分数给对上，用`zip()`轻松实现：

```
'use strict';

var names = ['Adam', 'Lisa', 'Bart'];
var scores = [85, 92, 59];
_.zip(names, scores);
// [['Adam', 85], ['Lisa', 92], ['Bart', 59]]
```

`unzip()`则是反过来：

```
'use strict';
var namesAndScores = [['Adam', 85], ['Lisa', 92], ['Bart',
59]];
_.unzip(namesAndScores);
// [['Adam', 'Lisa', 'Bart'], [85, 92, 59]]
```

object

有时候你会想，与其用`zip()`，为啥不把名字和分数直接对应成Object呢？别急，`object()`函数就是干这个的：

```
'use strict';

var names = ['Adam', 'Lisa', 'Bart'];
var scores = [85, 92, 59];
_.object(names, scores);
// {Adam: 85, Lisa: 92, Bart: 59}
```

注意`_.object()`是一个函数，不是JavaScript的`Object`对象。

range

`range()`让你快速生成一个序列，不再需要用`for`循环实现了：

```
'use strict';

// 从0开始小于10:
_.range(10); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

// 从1开始小于11:
_.range(1, 11); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

// 从0开始小于30，步长5:
_.range(0, 30, 5); // [0, 5, 10, 15, 20, 25]

// 从0开始大于-10，步长-1:
_.range(0, -10, -1); // [0, -1, -2, -3, -4, -5, -6, -7,
-8, -9]
```

更多完整的函数请参考underscore的文档：http://underscorejs.org/#arrays

练习

请根据underscore官方文档，使用`_.uniq`对数组元素进行*不区分大小写*去重：

```
'use strict';

var arr = ['Apple', 'orange', 'banana', 'ORANGE', 'apple',
'PEAR'];
var result = ???
    // 测试
if (result.toString() === ["Apple", "orange", "banana",
"PEAR"].toString()) {
    console.log('测试成功!');
} else {
    console.log('测试失败!');
}
```

Functions

因为underscore本来就是为了充分发挥JavaScript的函数式编程特性，所以也提供了大量JavaScript本身没有的高阶函数。

bind

`bind()`有什么用？我们先看一个常见的错误用法：

```
'use strict';

var s = ' Hello  ';
s.trim();
// 输出'Hello'

var fn = s.trim;
fn();
// Uncaught TypeError: String.prototype.trim called on
null or undefined
```

如果你想用`fn()`取代`s.trim()`，按照上面的做法是不行的，因为直接调用`fn()`传入的`this`指针是`undefined`，必须这么用：

```
'use strict';

var s = ' Hello  ';
var fn = s.trim;
// 调用call并传入s对象作为this:
fn.call(s)
// 输出Hello
```

这样搞多麻烦！还不如直接用`s.trim()`。但是，`bind()`可以帮我们把`s`对象直接绑定在`fn()`的`this`指针上，以后调用`fn()`就可以直接正常调用了：

```
'use strict';

var s = ' Hello  ';
var fn = _.bind(s.trim, s);
fn();
// 输出Hello
```

结论：当用一个变量`fn`指向一个对象的方法时，直接调用`fn()`是不行的，因为丢失了`this`对象的引用。用`bind`可以修复这个问题。

partial

`partial()`就是为一个函数创建偏函数。偏函数是什么东东？看例子：

假设我们要计算xy，这时只需要调用`Math.pow(x, y)`就可以了。

假设我们经常计算2y，每次都写`Math.pow(2, y)`就比较麻烦，如果创建一个新的函数能直接这样写`pow2N(y)`就好了，这个新函数`pow2N(y)`就是根据`Math.pow(x, y)`创建出来的偏函数，它固定住了原函数的第一个参数（始终为2）：

```
'use strict';

var pow2N = _.partial(Math.pow, 2);
pow2N(3); // 8
pow2N(5); // 32
pow2N(10); // 1024
```

如果我们不想固定第一个参数，想固定第二个参数怎么办？比如，希望创建一个偏函数`cube(x)`，计算x3，可以用`_`作占位符，固定住第二个参数：

```
'use strict';

var cube = _.partial(Math.pow, _, 3);
cube(3); // 27
cube(5); // 125
cube(10); // 1000
```

可见，创建偏函数的目的是将原函数的某些参数固定住，可以降低新函数调用的难度。

memoize

如果一个函数调用开销很大，我们就可能希望能把结果缓存下来，以便后续调用时直接获得结果。举个例子，计算阶乘就比较耗时：

```
'use strict';

function factorial(n) {
```

```
    console.log('start calculate ' + n + '!...');
    var s = 1, i = n;
    while (i > 1) {
        s = s * i;
        i --;
    }
    console.log(n + '! = ' + s);
    return s;
}

factorial(10); // 3628800
// 注意控制台输出:
// start calculate 10!...
// 10! = 3628800
```

用`memoize()`就可以自动缓存函数计算的结果：

```
'use strict';

var factorial = _.memoize(function(n) {
    console.log('start calculate ' + n + '!...');
    var s = 1, i = n;
    while (i > 1) {
        s = s * i;
        i --;
    }
    console.log(n + '! = ' + s);
    return s;
});

// 第一次调用:
factorial(10); // 3628800
// 注意控制台输出:
// start calculate 10!...
// 10! = 3628800

// 第二次调用:
factorial(10); // 3628800
// 控制台没有输出
```

对于相同的调用，比如连续两次调用`factorial(10)`，第二次调用并没有计算，而是直接返回上次计算后缓存的结果。不过，当你计算`factorial(9)`的时候，仍然会重新计算。

可以对`factorial()`进行改进，让其递归调用：

```
'use strict';

var factorial = _.memoize(function(n) {
    console.log('start calculate ' + n + '!...');
```

```
    if (n < 2) {
        return 1;
    }
    return n * factorial(n - 1);
});

factorial(10); // 3628800
// 输出结果说明factorial(1)~factorial(10)都已经缓存了:
// start calculate 10!...
// start calculate 9!...
// start calculate 8!...
// start calculate 7!...
// start calculate 6!...
// start calculate 5!...
// start calculate 4!...
// start calculate 3!...
// start calculate 2!...
// start calculate 1!...

factorial(9); // 362880
// console无输出
```

once

顾名思义，`once()`保证某个函数执行且仅执行一次。如果你有一个方法叫`register()`，用户在页面上点两个按钮的任何一个都可以执行的话，就可以用`once()`保证函数仅调用一次，无论用户点击多少次：

```
'use strict';
var register = _.once(function () {
    alert('Register ok!');
});
// 测试效果:
register();
register();
register();
```

delay

`delay()`可以让一个函数延迟执行，效果和`setTimeout()`是一样的，但是代码明显简单了：

```
'use strict';

// 2秒后调用alert():
_.delay(alert, 2000);
```

如果要延迟调用的函数有参数，把参数也传进去：

```
'use strict';

var log = _.bind(console.log, console);
_.delay(log, 2000, 'Hello,', 'world!');
// 2秒后打印'Hello, world!':
```

更多完整的函数请参考underscore的文档：http://underscorejs.org/#functions

Objects

和`Array`类似，underscore也提供了大量针对Object的函数。

keys / allKeys

`keys()`可以非常方便地返回一个object自身所有的key，但不包含从原型链继承下来的：

```
'use strict';

function Student(name, age) {
    this.name = name;
    this.age = age;
}

var xiaoming = new Student('小明', 20);
_.keys(xiaoming); // ['name', 'age']
```

`allKeys()`除了object自身的key，还包含从原型链继承下来的：

```
'use strict';

function Student(name, age) {
    this.name = name;
    this.age = age;
}
Student.prototype.school = 'No.1 Middle School';
var xiaoming = new Student('小明', 20);
_.allKeys(xiaoming); // ['name', 'age', 'school']
```

values

和`keys()`类似，`values()`返回object自身但不包含原型链继承的所有值：

```
'use strict';

var obj = {
    name: '小明',
    age: 20
};

_.values(obj); // ['小明', 20]
```

注意，没有`allvalues()`，原因我也不知道。

mapObject

`mapObject()`就是针对object的map版本：

```
'use strict';

var obj = { a: 1, b: 2, c: 3 };
// 注意传入的函数签名，value在前，key在后:
_.mapObject(obj, (v, k) => 100 + v); // { a: 101, b: 102,
c: 103 }
```

invert

`invert()`把object的每个key-value来个交换，key变成value，value变成key：

```
'use strict';

var obj = {
    Adam: 90,
    Lisa: 85,
    Bart: 59
};
_.invert(obj); // { '59': 'Bart', '85': 'Lisa', '90':
'Adam' }
```

extend / extendOwn

`extend()`把多个object的key-value合并到第一个object并返回：

```
'use strict';

var a = {name: 'Bob', age: 20};
_.extend(a, {age: 15}, {age: 88, city: 'Beijing'}); //
{name: 'Bob', age: 88, city: 'Beijing'}
// 变量a的内容也改变了：
a; // {name: 'Bob', age: 88, city: 'Beijing'}
```

注意：如果有相同的key，后面的object的value将覆盖前面的object的value。

`extendOwn()`和`extend()`类似，但获取属性时忽略从原型链继承下来的属性。

clone

如果我们要复制一个object对象，就可以用`clone()`方法，它会把原有对象的所有属性都复制到新的对象中：

```
'use strict';
var source = {
    name: '小明',
    age: 20,
    skills: ['JavaScript', 'CSS', 'HTML']
};
var copied = _.clone(source);
console.log(JSON.stringify(copied, null, '  '));
```

注意，`clone()`是“浅复制”。所谓“浅复制”就是说，两个对象相同的key所引用的value其实是同一对象：

```
source.skills === copied.skills; // true
```

也就是说，修改`source.skills`会影响`copied.skills`。

isEqual

`isEqual()`对两个object进行深度比较，如果内容完全相同，则返回`true`：

```
'use strict';

var o1 = { name: 'Bob', skills: { Java: 90, JavaScript: 99
}};
var o2 = { name: 'Bob', skills: { JavaScript: 99, Java: 90
}};

o1 === o2; // false
_.isEqual(o1, o2); // true
```

`isEqual()`其实对`Array`也可以比较：

```
'use strict';

var o1 = ['Bob', { skills: ['Java', 'JavaScript'] }];
var o2 = ['Bob', { skills: ['Java', 'JavaScript'] }];

o1 === o2; // false
_.isEqual(o1, o2); // true
```

更多完整的函数请参考underscore的文档：http://underscorejs.org/#objects

Chaining

还记得jQuery支持链式调用吗？

```
$('a').attr('target', '_blank')
      .append(' <i class="uk-icon-external-link"></i>')
      .click(function () {});
```

如果我们有一组操作，用underscore提供的函数，写出来像这样：

```
_.filter(_.map([1, 4, 9, 16, 25], Math.sqrt), x => x % 2
=== 1);
// [1, 3, 5]
```

能不能写成链式调用？

能！

underscore提供了把对象包装成能进行链式调用的方法，就是`chain()`函数：

```
var r = _.chain([1, 4, 9, 16, 25])
         .map(Math.sqrt)
         .filter(x => x % 2 === 1)
         .value();
console.log(r); // [1, 3, 5]
```

因为每一步返回的都是包装对象，所以最后一步的结果需要调用`value()`获得最终结果。

小结

通过学习underscore，是不是对JavaScript的函数式编程又有了进一步的认识？