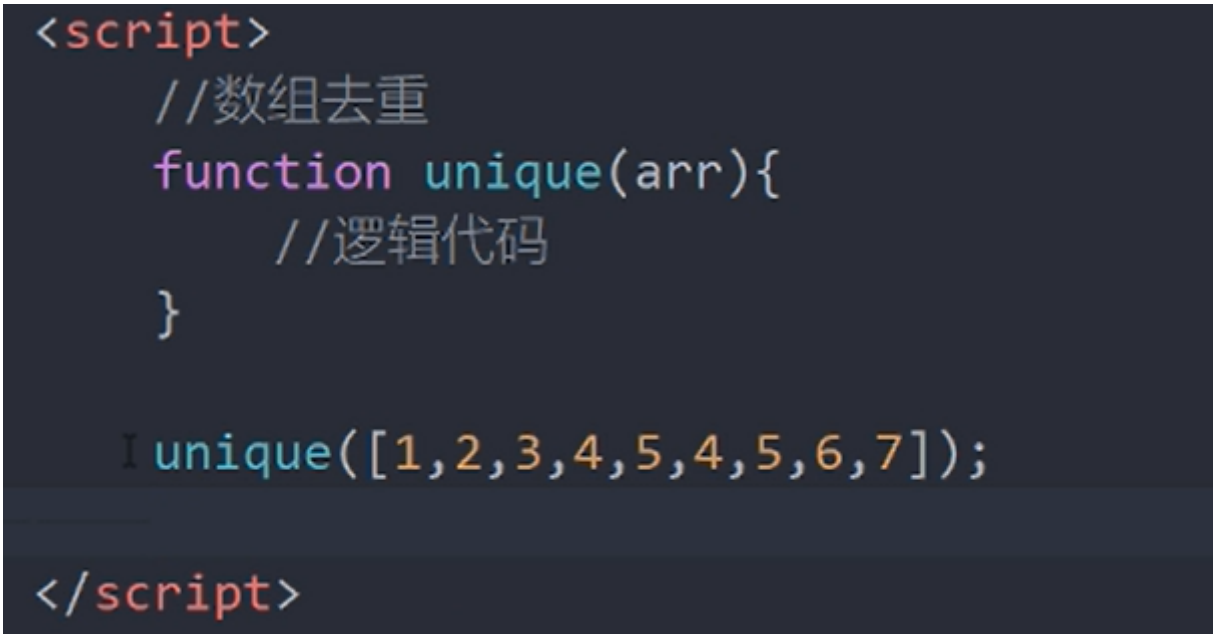
underscore与loadsh非常的类似，是早期比较流行的js库，学习源码主要是为了学习它优秀的架构，而不是它的每一个工具方法是怎么实现的。

https://www.underscorejs.com.cn/
https://www.lodashjs.com/docs/lodash.after

underscore和loadsh这两个库都是对于js对象的增强，原有的js对象在处理数据的过程中提供的方法比较薄弱，处理功能是有限的，如果我们想要增强，需要针对一些特定的业务来进行一些补偿怎么办呢？假设我们想要进行数组去重，通常我们会写一个函数解决，例如：



underscore为我们提供了上百个类似于unique这样的工具函数，它真正比较亮眼的地方在于它接近了类似于java里的Stream流式编程。

流式编程是一种什么样的概念?

它是一种新的解决方案，假设我这里有个数据，拿到这个数据之后先去重，再过滤，再取最大值等等。

```
//数据      先去重   在过滤   在取最大值...

unique([1,2,3,4,5,4,5,6,7]).filter().max();
```

流式编程就是把这里的函数都当做管道，这里的unique、filter、max都是管道，也就是一道工序，而数据就是在管道里进行流通，当上一道工序处理完成后再把数据传给下一道工序，依次执行。

- 流式处理

流是一系列数据，一次只生成一项。程序可以从输入流中一个一个读取数据项，然后以同样的方式将数据项写入输出流。一个程序的输出流可以是另一个程序的输入流。

- 流的特点

元素序列——就像集合一样，流也提供了一个接口，可以访问特定元素类型的一组有序值。集合讲究的是数据，流讲究的是计算。

```
menu.stream().filter(d -> d.getCalories() > 300).map(Dish::getName)
```

这里的menu就是数据，通过menu.stream方法开启流式编程，然后先把数据给到filter，通过它来进行过滤的加工操作，它处理完成之后再把数据给到下一道工序也就是map。

- 数据处理操作

流的数据处理功能支持类似于数据库的操作，以及函数式编程语言中的常用操作，如filter、map、reduce、find、match、sort等操作。流操作可以顺序执行，也可以并行执行。

上面的代码示例中我们想实现的功能就是流式编程，可以看到unique是第一道工序，filter是第二道工序，max是第三道工序，数据先在unique里面加工处理，然后再传给下一道工序依次循环。

这种方式类似于jQuery的链式调用。

```
$("form").css().animate()
```

在underscore中把这种方式成为流式编程，关注的并不是对象而是数据的流通，在链式调用中关注的是对象（每次调用方法后返回一个jQuery实例）。

underscore中为我们提供了一种比较标准的流式编程写法，支持如下两种调用方式

```
_([1,2,3,4,5,4,5,6,7]).unique().filter().max();
```

```
 _.unique([1,2,3,4,5,4,5,6,7]).filter().max();
```

这两种方式是怎么实现的呢?

要支持这两种方式的调用，第一种直接把underscore当做一个函数，通过underscore函数的调用，把数据传递给后面的工序来开启流式编程。第二种方式直接把underscore当做一个对象，通过对象的unique方法把数据传递给第一道工序开启流式编程。

```
_([1,2,3,4,5,4,5,6,7])          函数
_.unique([1,2,3,4,5,4,5,6,7])   对象
```

根据上面的分析我们编写如下代码：

```
(function(root) {
 // 创建underscore
 var _ = {};

 // 将underscore挂载到全局对象下
 root._ = _;
})(this)
```

上面的代码中，如果将underscore赋值为一个对象，那么第二种调用方式就实现不了。好在js中函数也是对象，所以我们改写代码如下：

```
(function(root) {
 // 创建underscore
 var _ = function() {

```

```
  };

  // 将underscore挂载到全局对象下
  root._ = _;
})(this)
```

接下来我们需要考虑的事情是，在调用_()的时候也需要跟调用jQuery的方法一样，需要创建它的实例，也就意味着如果接下来我们需要扩展一个数组去重的方法时，需要找到underscore的原型对象，给原型对象扩展一个unique方法给它的实例进行调用。

```
(function(root) {
  // 创建underscore
  var _ = function() {

  };

  _.prototype.unique = function() {

  }

   // 将underscore挂载到全局对象下
   root._ = _;
})(this)
```

问题在于在jQuery中，当我们调用$()时会创建它的实例对象，这种方式是通过一种叫做无new化构建的技术实现的，也就是说我们不需要通过new这个关键字。

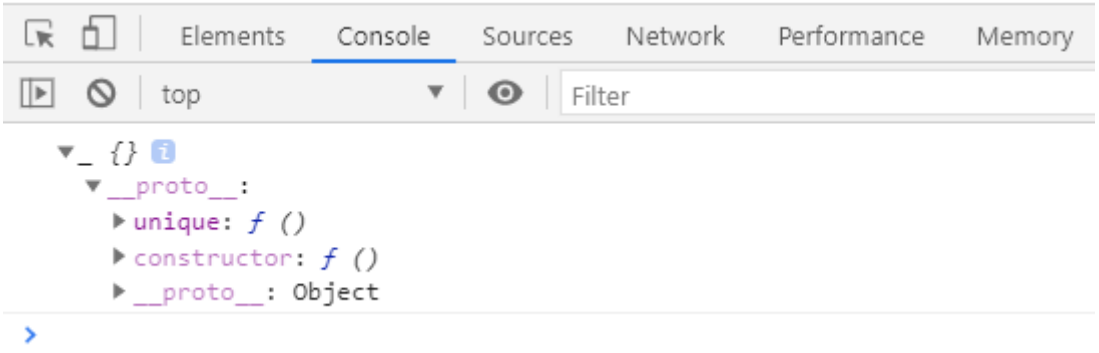
我们也要完成这个功能，当我们在调用underscore的时候，也就类似于我们需要通过new去创建一个underscore的实例，示例如下：

```
(function(root) {
  // 创建underscore
  var _ = function() {
  // 在调用underscore构造函数时，内部的this通过
  // instanceof来判断是否指向underscore的实例
  // 如果不是的话，通过return返回underscore的实例
  if(!(this instanceof _)) {
  return new _();
  }
   };

```

```
12  _.prototype.unique = function() {
13
14  }
15
16  // 将underscore挂载到全局对象下
17  root._ = _;
18 })(this)
```

如果我们在页面中引入这段代码，调用_()时，this指向的是window，这时候就会通过我们的判断代码返回underscore的实例了。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
<script src="_.js"></script>
<script>
console.log(
  _()
)
</script>
</body>
</html>
```

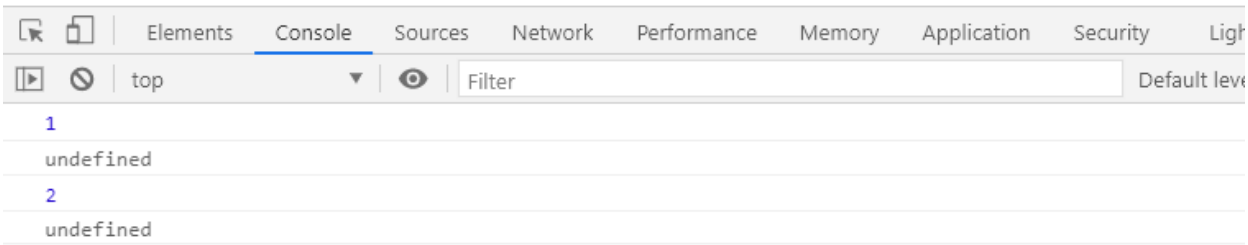


可以看到underscore的原型对象上找到了unique方法，通过_().unique()就可以调用了，要实现第二种方式的调用，就需要在underscore对象本身上扩展unique方法，代码如下：

```
1 (function(root) {
2  // 创建underscore
```

```
var _ = function() {
// 在调用underscore构造函数时，内部的this通过
// instanceof来判断是否指向underscore的实例
// 如果不是的话，通过return返回underscore的实例
if(!(this instanceof _)) {
return new _();
}
 };

 _.unique = function() {
 console.log(2);
 }
 _.prototype.unique = function() {
 console.log(1);
 }

 // 将underscore挂载到全局对象下
 root._ = _;
})(this)
```

```
<script src="_.js"></script>
<script>
console.log(
  _().unique()
)

console.log(
  _.unique()
)
</script>
```



上面的代码示例中，如果我们要实现两种不同方式的调用来开启流式编程，要么用时间换空间，要么用空间换时间，但是上面的示例中什么都没有，非常的傻。不管通过哪种方式调

用，最终要实现的功能是数组去重，它们内部的逻辑代码是一样的，如果每一个工具函数都需要写两份，underscore中有上百个工具函数，代码的冗余程度就非常高。

这个问题要怎么解决呢？underscore中告诉我们有一个方法叫做mixin。它是怎么做的呢？

1. 找到underscore对象上有哪些可枚举的属性
2. 找到可枚举的属性，存储在数组中
3. 遍历数组，找到数组中对应的成员，这些成员就是给underscore对象上扩展的属性名称，再给原型对象扩展

那么我们给原型对象添加的方法就不需要手动去写了，直接改成下面的代码即可

```
_.prototype[key] = function() {
 // 逻辑代码
}
```

至于mixin的实现如下：

```
(function(root) {
 // 创建underscore
 var _ = function() {
 // instanceof来判断this是否指向underscore的实例
 // 如果不是的话，通过return返回underscore的实例
 if(!(this instanceof _)) {
 return new _();
 }
 };

 _.unique = function () {
 console.log(2);
 }
 //_.prototype[key] = function() {
 // 逻辑代码
 //}

 // mixin
 _.mixin = function (target) {

 }

```

```
 // 调用mixin，把underscore对象传入
 _.mixin(_);
 // 将underscore挂载到全局对象下
 root._ = _;
})(this)
```

在mixin的内部我们需要调用一个beforeHook方法，给这个方法传入一个数组，这个数组用于存储给underscore扩展的一些属性和方法

```
(function(root) {
 // 创建underscore
 var _ = function() {
 // instanceof来判断this是否指向underscore的实例
 // 如果不是的话，通过return返回underscore的实例
 if(!(this instanceof _)) {
 return new _();
 }
 };

 _.unique = function () {
 console.log(2);
 }
 //_.prototype[key] = function() {
 // 逻辑代码
 //}

 var beforeHook = function (arr) {

 }

 // mixin
 _.mixin = function (target) {
 beforeHook(['unique', 'mixin', ......]);
 }

 // 调用mixin，把underscore对象传入
 _.mixin(_);
 // 将underscore挂载到全局对象下
 root._ = _;
})(this)
```

在上面的代码中我们可以看到underscore下有unique、mixin两个方法，在扩展的过程中会增加很多方法，因此这个数组我们需要动态获取，我们需要在beforeHook中调用anderscore的process方法，这个方法接收underscore实例对象，再通过for in循环遍历对象拿到underscore上所有的可枚举属性组成数组返回，代码如下：

```
(function(root) {
 // 创建underscore
 var _ = function() {
 // instanceof来判断this是否指向underscore的实例
 // 如果不是的话，通过return返回underscore的实例
 if(!(this instanceof _)) {
 return new _();
 }
 };

 _.unique = function () {
 console.log(2);
 }
 //_.prototype[key] = function() {
 // 逻辑代码
 //}

 _.process = function (target) {
 var ref = [];
 for (var name in target) {
 ref.push(name);
 }
 return ref;
 }

 var beforeHook = function (arr) {

 }

 // mixin
 _.mixin = function (target) {
 beforeHook(_.process(target));
 }

 // 调用mixin，把underscore对象传入
 _.mixin(_);
```

```
// 将underscore挂载到全局对象下
root._ = _;
})(this)
```

beforeHook方法除了接收一个数组之外，还需要接收一个回调函数

```
var beforeHook = function (arr, callback) {
 console.log(arr);
}
```

我们先输出arr看一下，已经拿到了这些可枚举的属性

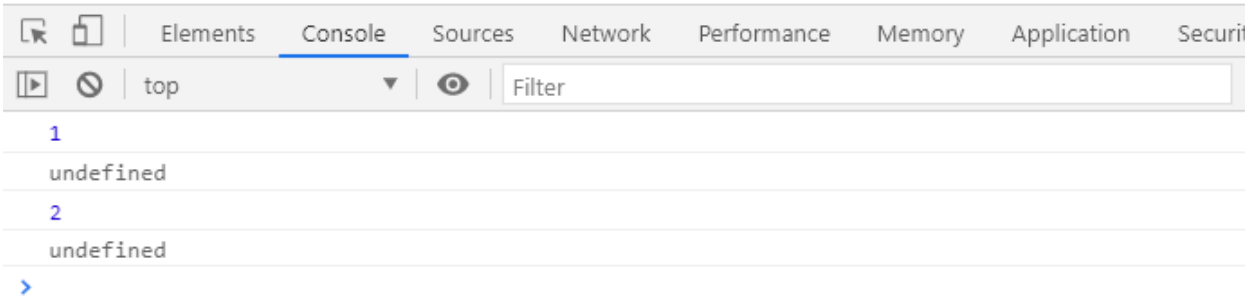
```
▶ (3) ["unique", "process", "mixin"]
```

接下来我们需要遍历这个arr，将数组的每一个成员传递给回调函数，在回调函数中给underscore的原型对象扩展属性。

```
(function(root) {
 // 创建underscore
 var _ = function() {
 // instanceof来判断this是否指向underscore的实例
 // 如果不是的话，通过return返回underscore的实例
 if(!(this instanceof _)) {
 return new _();
 }
 };

 _.unique = function () {
 console.log(2);
 }

 _.process = function (target) {
 var ref = [];
 for (var name in target) {
 ref.push(name);
 }
 return ref;
 }

 var beforeHook = function (arr, callback) {
 for (var i = 0; i < arr.length; i++) {
 callback[arr[i]];
```

```
26  }
27  }
28
29  // mixin
30  _.mixin = function (target) {
31  beforeHook(_.process(target), function (key) {
32  target.prototype[key] = function () {
33  // 逻辑代码
34  console.log(1);
35  }
36  });
37  }
38
39  // 调用mixin，把underscore对象传入
40  _.mixin(_);
41  // 将underscore挂载到全局对象下
42  root._ = _;
43 })(this)
```

这个时候我们就可以通过两种不同的方式调用underscore了，回到页面运行代码可以看到输出了



但这并不是我们想要的结果，我们希望同一个方法的输出结果是一致的，最终执行的是一个逻辑代码，所以我们需要改写代码，先找到underscore上面对应的key的属性值，假设key为unique，接着就需要找到unique对应的函数，把这个函数赋值给变量func存储起来，接着在原型对象的方法体中调用func就可以了。

```
1 (function(root) {
2  // 创建underscore
3  var _ = function() {
4  // instanceof来判断this是否指向underscore的实例
5  // 如果不是的话，通过return返回underscore的实例
6  if(!(this instanceof _)) {
7  return new _();
```

```
}
};

_.unique = function () {
console.log(2);
}
//_.prototype[key] = function() {
// 逻辑代码
//}

_.process = function (target) {
var ref = [];
for (var name in target) {
ref.push(name);
}
return ref;
}

var beforeHook = function (arr, callback) {
for (var i = 0; i < arr.length; i++) {
callback(arr[i]);
}
}

// mixin
_.mixin = function (target) {
beforeHook(_.process(target), function (key) {
var func = target[key];
target.prototype[key] = function () {
// 逻辑代码
func();
}
});
}

// 调用mixin，把underscore对象传入
_.mixin(_);
// 将underscore挂载到全局对象下
root._ = _;
})(this)
```

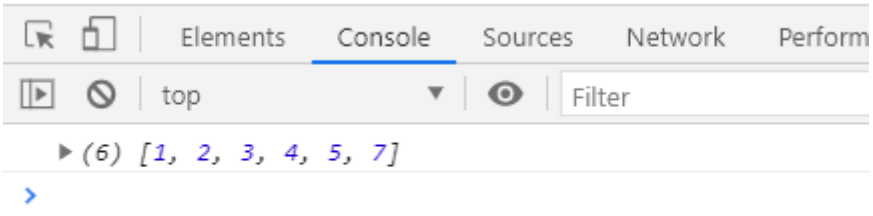
这样意味着后续的逻辑代码直接在unique函数当中编写就可以了，不用再同时关心两个地方的代码是怎么写的。这就是mixin架构，未来整个工具库的增强都不用关心，只需要关注怎么给underscore对象扩展属性，函数中的逻辑该怎么写。

上面的代码存在一些问题，它并没有一个真实的接口进行测试，不知道代码的可用性是怎样的，接下来我们测试一下，先把unique方法写完，通过unique这个api来测试一下这段代码的可用性。

```
(function(root) {
 // 创建underscore
 var _ = function() {
 // instanceof来判断this是否指向underscore的实例
 // 如果不是的话，通过return返回underscore的实例
 if(!(this instanceof _)) {
 return new _();
 }
 };

 _.unique = function (source) {
 var ref = [];
 for (var i = 0; i < source.length; i++ ) {
 var target = source[i];
 if (ref.indexOf(target) === -1) {
 ref.push(target);
 }
 }
 return ref;
 }

 _.process = function (target) {
 var ref = [];
 for (var name in target) {
 ref.push(name);
 }
 return ref;
 }

 var beforeHook = function (arr, callback) {
```

```
31  for (var i = 0; i < arr.length; i++) {
32  callback(arr[i]);
33  }
34  }
35
36  // mixin
37  _.mixin = function (target) {
38  beforeHook(_.process(target), function (key) {
39  var func = target[key];
40  target.prototype[key] = function () {
41  // 逻辑代码
42  func();
43  }
44  });
45  }
46
47  // 调用mixin，把underscore对象传入
48  _.mixin(_);
49  // 将underscore挂载到全局对象下
50  root._ = _;
51 })(this)
```

```
14  console.log(
15    _.unique([1,2,3,4,5,4,5,7])
16  )
```



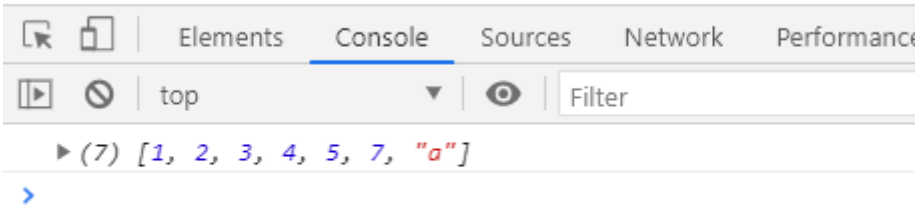
上面的数组已经去重了，此时我们再扩展一下，在数组里面加入大写的A和小写的a，我们需要让它不区分大小写去重，那么我们可以在unique中传入一个回调函数，通过这个回调函数专门做这个事情。

```
1 (function(root) {
2  // 创建underscore
3  var _ = function() {
```

```
// instanceof来判断this是否指向underscore的实例
// 如果不是的话，通过return返回underscore的实例
if(!(this instanceof _)) {
return new _();
}
};

 _.unique = function (source, callback) {
 var ref = [];
 for (var i = 0; i < source.length; i++ ) {
 // 判断callback是否存在
 // 存在则将数组成员传递给callback
 var target = callback ? callback(source[i]) : source[i];
 if (ref.indexOf(target) === -1) {
 ref.push(target);
 }
 }
 return ref;
 }

 _.process = function (target) {
 var ref = [];
 for (var name in target) {
 ref.push(name);
 }
 return ref;
 }

 var beforeHook = function (arr, callback) {
 for (var i = 0; i < arr.length; i++) {
 callback(arr[i]);
 }
 }

 // mixin
 _.mixin = function (target) {
 beforeHook(_.process(target), function (key) {
 var func = target[key];
 target.prototype[key] = function () {
 // 逻辑代码
```

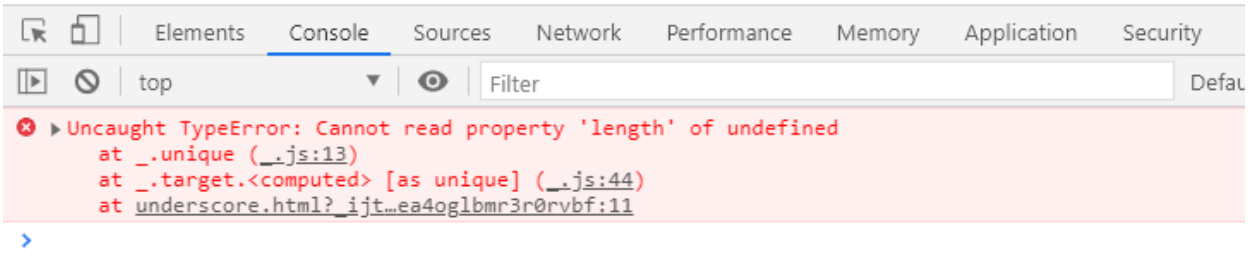
```
    func();
    }
    });
    }

    // 调用mixin，把underscore对象传入
    _.mixin(_);
    // 将underscore挂载到全局对象下
    root._ = _;
})(this)
```

```
console.log(
  _.unique([1,2,3,4,5,4,5,7,'a','A'], function (item) {
    return typeof item === 'string' ? item.toLowerCase() : item;
  })
)
```



上面的代码演示了underscore其中一种调用方式，那么使用第二种调用方式是否能达到我们的预期效果呢?

```
console.log(
  _([1,2,3,4,5,4,5,7,'a','A']).unique(function (item) {
    return typeof item === 'string' ? item.toLowerCase() : item;
  })
)
```



运行报错，原因是当我们通过实例去调用unique方法的时候并没有给它传参，所以unique方法的sourse接收的是一个undefined。那么，我该怎么把数据跟回调正确的推送给unique方法呢?

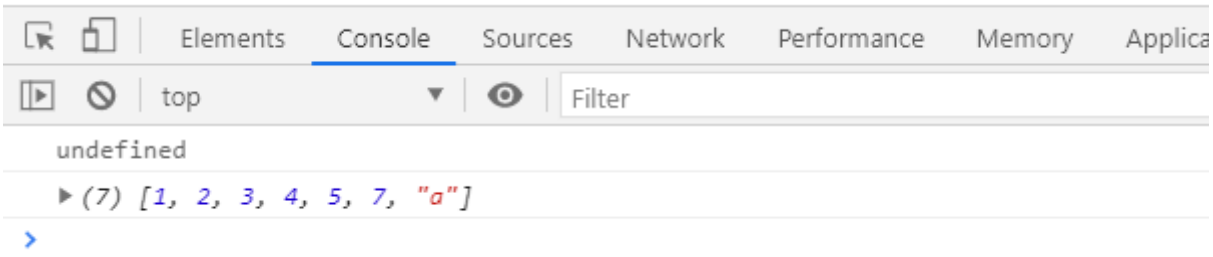
先看数据，数据会先传递给underscore的构造函数，所以我们在构造函数中接收传入的数据

```
// 接收数据
var _ = function(data) {
 if(!(this instanceof _)) {
 // 将数据传递给underscore的实例
 return new _(data);
 }
 // 通过this.wrapper把数据保存起来
 this.wrapper = data;
};
```

然后回到mixin这块，如果我们在调用unique的时候想要获取这个数据，通过this.wrapper就可以获取到了，我们需要将数据作为第一个参数传递给func，在unique的sourse中就能接收它。

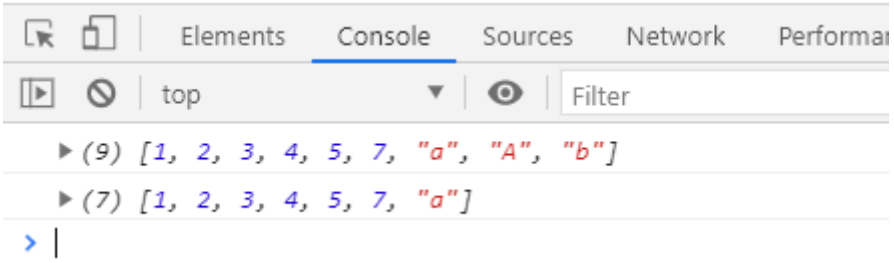
```
// mixin
_.mixin = function (target) {
 beforeHook(_.process(target), function (key) {
 var func = target[key];
 target.prototype[key] = function () {
 // 逻辑代码
 func(this.wrapper);
 }
 });
}
```

```
console.log(
  _([1,2,3,4,5,4,5,7,'a','A','b']).unique(function (item) {
    return typeof item === 'string' ? item.toLowerCase() : item;
  })
)

console.log(
  _.unique([1,2,3,4,5,4,5,7,'a','A'], function (item) {
    return typeof item === 'string' ? item.toLowerCase() : item;
  })
)
```



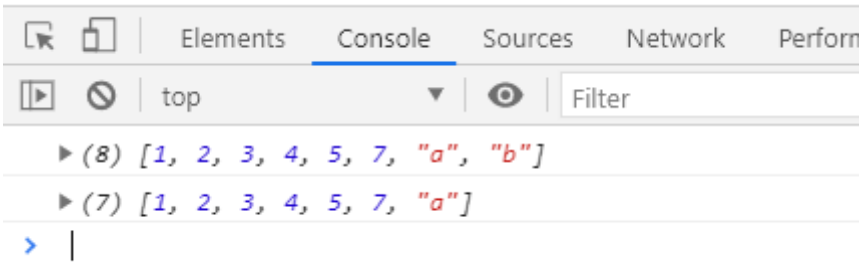
此时运行就不会报错了，但是可以看到上面的调用是一个undefined，原因是调用原型对象扩展的方法时并没有设置返回值，所以我们需要在调用原型对象的方法时设置返回值，它的返回值就是underscore的这个unique函数(func)。

```
// mixin
_.mixin = function (target) {
 beforeHook(_.process(target), function (key) {
 var func = target[key];
 target.prototype[key] = function () {
 // 逻辑代码
 return func(this.wrapper);
 }
 });
}
```



但又有一个问题，它并没有按照我们传入的处理函数区分大小写，原因是由于处理函数我们传给了实例的unique方法，实例的unique函数指向的是target.prototype[key] = function这个函数，我们可以通过argument获取传入的处理函数，将处理函数传递给unique函数。

```
// mixin
_.mixin = function (target) {
 beforeHook(_.process(target), function (key) {
 var func = target[key];
 target.prototype[key] = function () {
 // 逻辑代码
 return func(this.wrapper, arguments[0]);
 }
 });
}
```

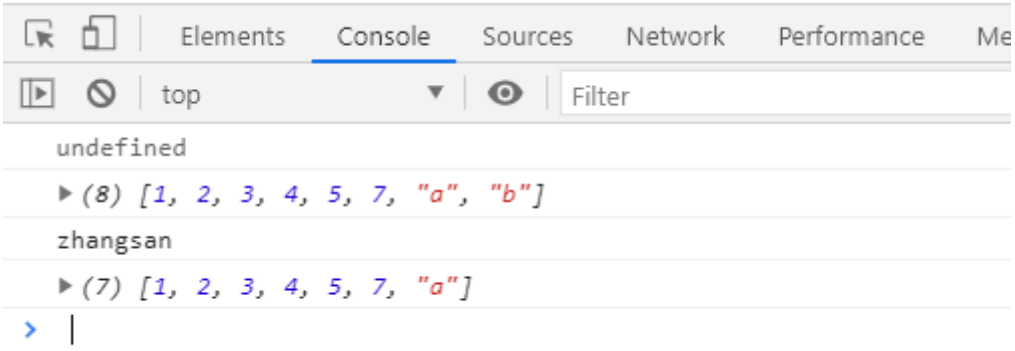


通过目前的unique接口，测试了整体代码的可行性，我们发现代码可以跑起来了，但这样的代码没有通用性，我们一切的接口都是围绕unique方法来进行的，在underscore中会有很多的工具方法，如果接口里面我们需要的参数是不固定的，可能有多个，例如：

```
console.log(
  _([1,2,3,4,5,4,5,7,'a','A','b']).unique(function (item) {
    return typeof item === 'string' ? item.toLowerCase() : item;
  }, 'zhangsan')
)

console.log(
  _.unique([1,2,3,4,5,4,5,7,'a','A'], function (item) {
    return typeof item === 'string' ? item.toLowerCase() : item;
  }, 'zhangsan')
)
```

```
_.unique = function (source, callback, n) {
 console.log(n)
```

```
var ref = [];
for (var i = 0; i < source.length; i++ ) {
// 判断callback是否存在
// 存在则将数组成员传递给callback
var target = callback ? callback(source[i]) : source[i];
if (ref.indexOf(target) === -1) {
ref.push(target);
 }
 }
 return ref;
}
```



上面的调用方式就会输出undefined，原因是调用func方法时没有传参，此时我们不可能通过argument将参数一个一个写进去，那么我们要如何解决呢?

我们可以通过数组合并的方式解决这个问题

1. 将this.wrapper存入数组中

```
// mixin
_.mixin = function (target) {
 beforeHook(_.process(target), function (key) {
 var func = target[key];
 target.prototype[key] = function () {
 var decon = [this.wrapper];
 // 逻辑代码
 return func(this.wrapper, arguments[0]);
 }
  });
}
```

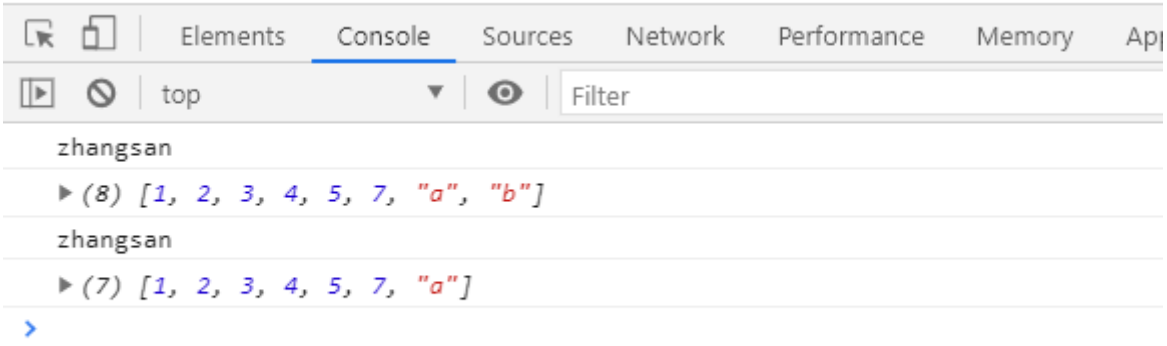
2. 将argument解构并与数组合并

```
// mixin
_.mixin = function (target) {
 beforeHook(_.process(target), function (key) {
 var func = target[key];
 target.prototype[key] = function () {
 var decon = [this.wrapper];
 // 通过apply将类数组参数解构并与decon合并
 // 结构类似于['数据', '参数1', '参数2', '...', '参数n']
 Array.prototype.push.apply(decon, arguments);
  // 逻辑代码
  return func(this.wrapper, arguments[0]);
  }
  });
}
```

3. 通过apply调用func，让this指向调用这个函数的实例，再将decon数组传入，数组的每一项成员都会作为func的参数推送给unique函数

```
// mixin
_.mixin = function (target) {
 beforeHook(_.process(target), function (key) {
 var func = target[key];
 target.prototype[key] = function () {
 var decon = [this.wrapper];
 Array.prototype.push.apply(decon, arguments);
 // 通过apply调用func，数组的每一项成员都会作为func的参数传入
 return func.apply(this, decon);
  }
  });
}
```

此时运行结果就正确了



代码如下：

```
(function(root) {
 // 创建underscore
 // 接收数据
 var _ = function(data) {
 // instanceof来判断this是否指向underscore的实例
 // 如果不是的话，通过return返回underscore的实例
 if(!(this instanceof _)) {
 // 将数据传递给underscore的实例
 return new _(data);
 }
 // 通过this.wrapper把数据保存起来
 this.wrapper = data;
 };

 _.unique = function (source, callback) {
 var ref = [];
 for (var i = 0; i < source.length; i++ ) {
 // 判断callback是否存在
 // 存在则将数组成员传递给callback
 var target = callback ? callback(source[i]) : source[i];
 if (ref.indexOf(target) === -1) {
 ref.push(target);
 }
 }
 return ref;
 }

 _.process = function (target) {
 var ref = [];
```

```
30  for (var name in target) {
31  ref.push(name);
32  }
33  return ref;
34  }
35
36  var beforeHook = function (arr, callback) {
37  for (var i = 0; i < arr.length; i++) {
38  callback(arr[i]);
39  }
40  }
41
42  // mixin
43  _.mixin = function (target) {
44  beforeHook(_.process(target), function (key) {
45  var func = target[key];
46  target.prototype[key] = function () {
47  var decon = [this.wrapper];
48  Array.prototype.push.apply(decon, arguments);
49  // 逻辑代码
50  return func.apply(this, decon);
51  }
52  });
53  }
54
55  // 调用mixin，把underscore对象传入
56  _.mixin(_);
57  // 将underscore挂载到全局对象下
58  root._ = _;
59 })(this)
```

而我们最终的目的是要实现流式编程，要把函数当做管道，把数据当做液体在管道中进行流通，想要搭建流式编程我们关注的是数据，但实际上我们还要通过对象的方式来桥接链式调用，也就是说，这种流式编程实际上在js中最底层的原理还是在不断的调用对象，这里直接返回的是数据显然是不行的，我们还是要返回对象，这个对象里面裹挟着在上一道工序处理的数据的结果。

要完成这个目标，我们需要做一些约定，例如：

```
console.log(
  // 如果想要开启流式编程，必须先调用chain方法
  _([1,2,3,4,5,4,5,7,'a','A','b']).chain().unique(function (item) {
    return typeof item === 'string' ? item.toLowerCase() : item;
  })
)
```

调用chain方法则开启流式编程，如果没有调用，则直接返回数据的处理结果如[1, 2, 3, 4, 5, 7, "a", "b"]

我们先定义一个chain方法，通过underscore的实例调用chain时，执行的是target.prototype[key] = function这个方法，这个方法我们没有传值，因此argument就是空的类数组。调用func时，func函数指向的就是chain，当我们在调用chain的时候就会传入在func中传入的decon数组，此时这个数组中只有一个this.wrapper成员，这个数据作为第一个参数传递给chain。

```
_.chain = function (data) {
  // data就是func函数中传过来的数据
}
```

在chain函数体内部，我们调用underscore的构造函数，将数据传过去，然后创建一个变量instance接收构造函数调用创建的实例，再给instance扩展一个_chain属性，它的值为true，然后返回instance实例对象。

```
_.chain = function (data) {
  // data就是func函数中传过来的数据
  var instance = _(data);
  instance._chain = true;
  return instance;
}
```

没有调用chain时，返回的是数据的结果，调用chain时，返回的是一个特殊的underscore的实例对象，它会带一个_chain的属性，值为true，这是它的特殊性。

我们光有chain函数还不够，还需要一个辅助函数帮我们完成这个事情，我们定义一个model辅助函数。它接收两个参数，一个是instance（实例对象），另一个是outcome（数据的输出结果）

```
var model = function (instance, outcome) {

}
```

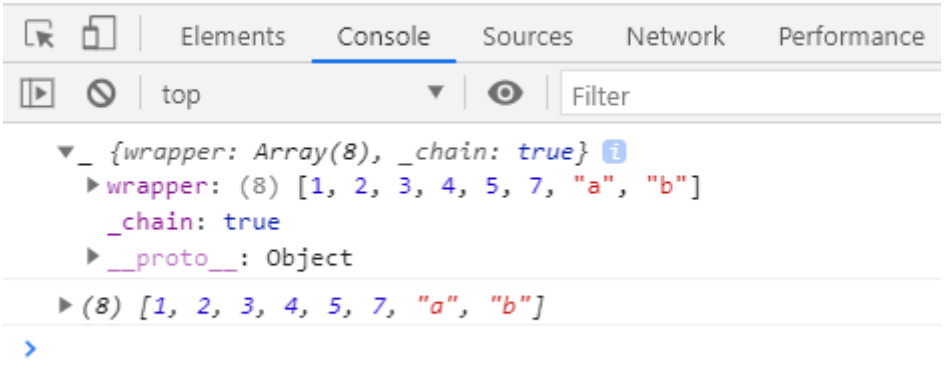
然后在调用func时，我们先调用model这个辅助函数，把this传入，再将func的执行结果传入。

```
// mixin
_.mixin = function (target) {
 beforeHook(_.process(target), function (key) {
 var func = target[key];
 target.prototype[key] = function () {
 var decon = [this.wrapper];
 Array.prototype.push.apply(decon, arguments);
 // 逻辑代码
 // 传入实例对象和本道工序数据处理的结果
  return model(this, func.apply(this, decon));
  }
  });
}
```

此时辅助函数model接收到了参数以后，需要对输入的参数进行判断和输出

```
var model = function (instance, outcome) {
 if (instance._chain) {
 // 如果实例的_chain属性为true
 // 则将处理的数据包裹在实例的wrapper属性上
 // 并将实例返回
 // 这里的wrapper属性与构造函数内定义的wrapper相同
 instance.wrapper = outcome;
 return instance;
 }
  // 否则直接返回处理的数据
  return outcome;
}
```

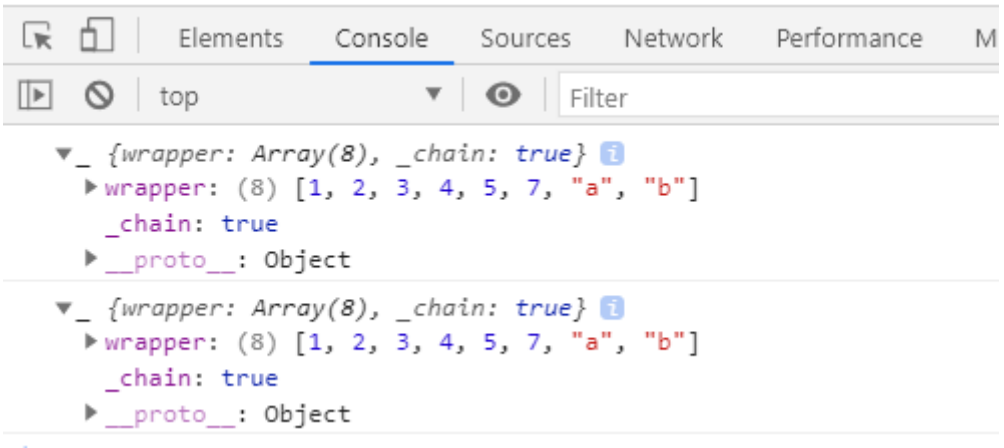
控制台运行一下



可以看到通过chain开启流式编程之后返回了一个特殊的underscore实例对象，如果没有调用chain返回的就是数据处理之后的结果。

第二种调用方式使用如下：

```
console.log(
  _.chain([1,2,3,4,5,4,5,7, 'a', 'A', 'b']).unique(function (item) {
    return typeof item == 'string' ? item.toLowerCase() : item
  })
)
```



调用的原则是必须将数据从第一个通道依次流通下去。

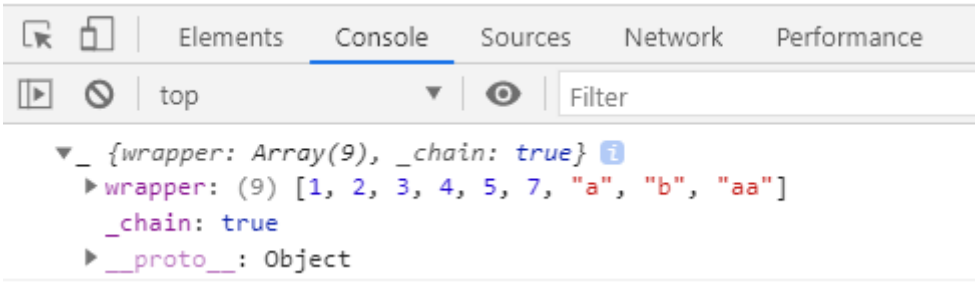
如果我们还有一些工序比如说max，我们需要给underscore对象扩展这个方法，扩展方法里面逻辑定义我们需要注意一个规律，通过unique这道工序处理完数据之后返回的是一个

特殊的underscore实例，也就意味着我们要通过这个特殊的underscore实例调用max方法，它走的还是与unique一样的通道，后续想要扩展接口，接口中的第一个参数接收的是上一道工序处理的数据结果，并且返回本次的处理结果。

```
// 接收上一道工序处理的结果
_.max = function (data) {
 // 逻辑代码
 // 假设这里push一个aa
 data.push('aa');
 // 返回本道工序加工处理的结果
 return data;
}
```

当我们调用完max这道工序之后，就会将结果更新到实例的wrapper属性上。

```
// 如果想要开启流式编程，必须先调用chain方法
_([1,2,3,4,5,4,5,7,'a','A','b']).chain().unique(function (item) {
  return typeof item === 'string' ? item.toLowerCase() : item;
}).max()
)
```

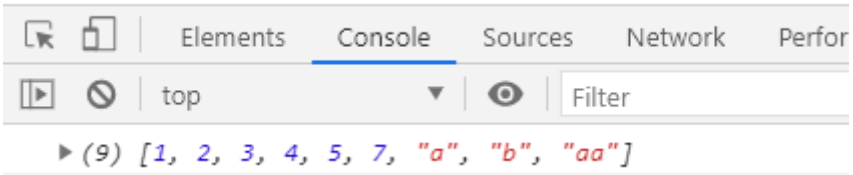


最后，如果我们的步骤模型都已经搭建完成了，想要拿到最终返回的数据结果该怎么办呢？

我们需要在underscore的原型对象上扩展一个ending方法，ending方法内部直接返回this.wrapper就可以了。

```
_.prototype.ending = function () {
 return this.wrapper;
}
```

```
console.log(
  // 如果想要开启流式编程，必须先调用chain方法
  _([1,2,3,4,5,4,5,7,'a','A','b']).chain().unique(function (item) {
    return typeof item === 'string' ? item.toLowerCase() : item;
  }).max().ending()
)
```



这样在最后一道工序处理完成之后调用ending就可以直接拿到最终更新的数据了。

以上的步骤都需要通过文档来约束用户怎么使用，只有在规定的使用范围内用户是自由的。

完整代码如下：

```
(function(root) {
 // 创建underscore
 // 接收数据
 var _ = function(data) {
 // instanceof来判断this是否指向underscore的实例
 // 如果不是的话，通过return返回underscore的实例
 if(!(this instanceof _)) {
 // 将数据传递给underscore的实例
 return new _(data);
  }
  // 通过this.wrapper把数据保存起来
  this.wrapper = data;
  };

  _.unique = function (source, callback) {
  var ref = [];
  for (var i = 0; i < source.length; i++ ) {
  // 判断callback是否存在
  // 存在则将数组成员传递给callback
  var target = callback ? callback(source[i]) : source[i];
  if (ref.indexOf(target) === -1) {
  ref.push(target);
  }
```

```
}
return ref;
}

// 接收上一道工序处理的结果
_.max = function (data) {
// 逻辑代码
// 假设这里push一个aa
data.push('aa');
// 返回本次处理的结果
return data;
}

_.process = function (target) {
var ref = [];
for (var name in target) {
ref.push(name);
}
return ref;
}

_.prototype.ending = function () {
return this.wrapper;
}

_.chain = function (data) {
// data就是func函数中传过来的数据
var instance = _(data);
instance._chain = true;
return instance;
}

var model = function (instance, outcome) {
if (instance._chain) {
// 如果实例的_chain属性为true
// 则将处理的数据包裹在实例的wrapper属性上
// 并将实例返回
instance.wrapper = outcome;
return instance;
}
// 否则直接返回处理的数据
```

```
  return outcome;
  }

  var beforeHook = function (arr, callback) {
  for (var i = 0; i < arr.length; i++) {
  callback(arr[i]);
  }
  }

  // mixin
  _.mixin = function (target) {
  beforeHook(_.process(target), function (key) {
  var func = target[key];
  target.prototype[key] = function () {
  var decon = [this.wrapper];
  Array.prototype.push.apply(decon, arguments);
  // 逻辑代码
  // 传入实例对象和本道工序数据处理的结果
  return model(this, func.apply(this, decon));
  }
  });
  }

  // 调用mixin，把underscore对象传入
  _.mixin(_);
  // 将underscore挂载到全局对象下
  root._ = _;
})(this)
```