你真的了解 JavaScript 中的高阶函数么？

本文讲解的**高阶函数**是之前讲解的闭包的续集，所以在学习高阶函数之前，一定要确保对闭包以及作用域的概念已经有了解：

理解抽象

引出抽象的概念

有 Java、C#等开发经验的同学对代码抽象的思想一定不会陌生，抽象类、接口平时写的非常多，但是对于一直都从事前端开发的同学来说，“抽象”这个词就比较陌生了，毕竟 JavaScript 中没有 abstract、interface。

但是 JS 中肯定是有代码抽象的思想的，只不过是形式上和 Java 等语言不同罢了！

先来看 Java 中的一个抽象类：

```
public abstract class SuperClass {
  public abstract void doSomething();
}
```

这是 Java 中的一个类，类里面有一个抽象方法 doSomething，现在不知道子类中要 doSomething 方法做什么，所以将该方法定义为抽象方法，具体的逻辑让子类自己去实现。

创建子类去实现 SuperClass：

```
public class SubClass  extends SuperClass{
  public void doSomething() {
    System.out.println("say hello");
  }
}
```

SubClass 中的 doSomething 输出字符串“say hello”，其他的子类会有其他的实现，这就是 Java 中的抽象类与实现。

那么 JS 中的抽象是怎么样的，最为经典的就是回调函数了：

```
function createDiv(callback) {
```

```
  let div = document.createElement('div');
  document.body.appendChild(div);
  if (typeof callback === 'function') {
    callback(div);
  }
}
createDiv(function (div) {
  div.style.color = 'red';
})
```

这个例子中，有一个 createDiv 这个函数，这个函数负责创建一个 div 并添加到页面中，但是之后要再怎么操作这个 div，createDiv 这个函数就不知道，所以把权限交给调用 createDiv 函数的人，让调用者决定接下来的操作，就通过回调的方式将 div 给调用者。

这也是体现出了抽象，既然不知道 div 接下来的操作，那么就直接给调用者，让调用者去实现。 和 Java 中抽象类中的抽象方法的思想是一样的。

总结一下抽象的概念：**抽象就是隐藏更具体的实现细节，从更高的层次看待我们要解决的问题**。

数组中的遍历抽象

在编程的时候，并不是所有功能都是现成的，比如上面例子中，可以创建好几个 div，对每个 div 的处理都可能不一样，需要对未知的操作做抽象，预留操作的入口，作为一名程序员，我们需要具备这种在恰当的时候将代码抽象的思想。

接下来看一下 ES5 中提供的几个数组操作方法，可以更深入的理解抽象的思想，ES5 之前遍历数组的方式是：

```
var arr = [1, 2, 3, 4, 5];
for (var i = 0; i < arr.length; i++) {
  var item = arr[i];
  console.log(item);
}
```

仔细看一下，这段代码中用 for，然后按顺序取值，有没有觉得如此操作有些不够优雅，为出现错误留下了隐患，比如把 length 写错了，一不小心复用了 i。既然这样，能不能抽取一个函数出来呢？最重要的一点，我们要的只是数组中的每一个值，然后操作这个值，那么就可以把遍历的过程隐藏起来：

```
function forEach(arr, callback) {
  for (var i = 0; i < arr.length; i++) {
    var item = arr[i];
    callback(item);
  }
}
forEach(arr, function (item) {
  console.log(item);
});
```

以上的 forEach 方法就将遍历的细节隐藏起来的了，把用户想要操作的 item 返回出来，在 callback 还可以将 i、arr 本身返回：callback(item, i, arr)。

JS 原生提供的 forEach 方法就是这样的：

```
arr.forEach(function (item) {
  console.log(item);
});
```

跟 forEach 同族的方法还有 map、some、every 等。思想都是一样的，通过这种抽象的方式可以让使用者更方便，同事又让代码变得更加清晰。

抽象是一种很重要的思想，让可以让代码变得更加优雅，并且操作起来更方便。**在高阶函数中也是使用了抽象的思想，所以学习高阶函数得先了解抽象的思想**。

高阶函数

什么是高阶函数

至少满足以下条件的中的一个，就是高阶函数：

- 将其他函数作为参数传递
- 将函数作为返回值

简单来说，就是**一个函数可以操作其他函数**，将其他函数作为参数或将函数作为返回值。我相信，写过 JS 代码的同学对这个概念都是很容易理解的，因为在 JS 中函数就是一个普通的值，可以被传递，可以被返回。

参数可以被传递，可以被返回，对 Java 等语言开发的同学理解起来可能会稍微麻烦一些，因为 Java 语言没有那么的灵活，不过 Java8 的 lambda 大概就是这意思；

函数作为参数传递

函数作为参数传递就是我们上面提到的回调函数，回调函数在异步请求中用的非常多，使用者想要在请求成功后利用请求回来的数据做一些操作，但是又不知道请求什么时候结束。

用 jQuery 来发一个 Ajax 请求：

```
function getDetailData(id, callback) {
  $.ajax('http://xxxxyyy.com/getDetailData?' + id, function (res) {
    if (typeof callback === 'function') {
      callback(res);
    }
  });
}
getDetailData('78667', function (res) {
  // do some thing
});
```

类似 Ajax 这种操作非常适合用回调去做，当一个函数里不适合执行一些具体的操作，或者说不知道要怎么操作时，可以将相应的数据传递给另一个函数，让另一个函数来执行，而这个函数就是传递进来的回调函数。

另一个典型的例子就是数组排序。

函数作为值返回

在判断数据类型的时候最常用的是 typeof，但是 typeof 有一定的局限性，比如：

```
console.log(typeof []); // 输出 object
console.log(typeof {}); // 输出 object
```

判断数组和对象都是输出 object，如果想要更细致的判断应该要使用 Object.prototype.toString

```
console.log(Object.prototype.toString.call([])); // 输出[object
Array]
console.log(Object.prototype.toString.call({})); // 输出[object
Object]
```

基于此，我们可以写出判断对象、数组、数字的方法：

```
function isObject(obj) {
  return Object.prototype.toString.call(obj) === '[object Object]';
}
function isArray(arr) {
  return Object.prototype.toString.call(arr) === '[object Array]';
}
function isNumber(number) {
  return Object.prototype.toString.call(number) === '[object
Number]';
}
```

我们发现这三个方法太像了，可以做一些抽取：

```
function isType(type) {
  return function (obj) {
    return Object.prototype.toString.call(obj) === '[object ' + type
+ ']';
  }
}
var isArray = isType('Array');
console.log(isArray([1,2]));
```

这个 isType 方法就是高阶函数，该函数返回了一个函数，并且利用闭包，将代码变得优雅。

高阶函数的应用

lodash 中的使用

高阶函数在平时的开发中用的非常多，只是有时候你不知道你的这种用法就是高阶函数，在一些开源的类库中也用的很多，比如很有名的 lodash，挑其中一个 before 函数：

```
function before(n, func) {
  let result
  if (typeof func != 'function') {
    throw new TypeError('Expected a function')
  }
  return function(...args) {
    if (--n > 0) {
      result = func.apply(this, args)
    }
    if (n <= 1) {
      func = undefined
    }
    return result
  }
}
```

在 before 函数中，同时有用到**将函数当做传递进来**，又**返回了一个函数**，这是一个很经典的高阶函数的例子。

看一下该代码可以怎么用吧：

```
jQuery(element).on('click', before(5, addContactToList))
```

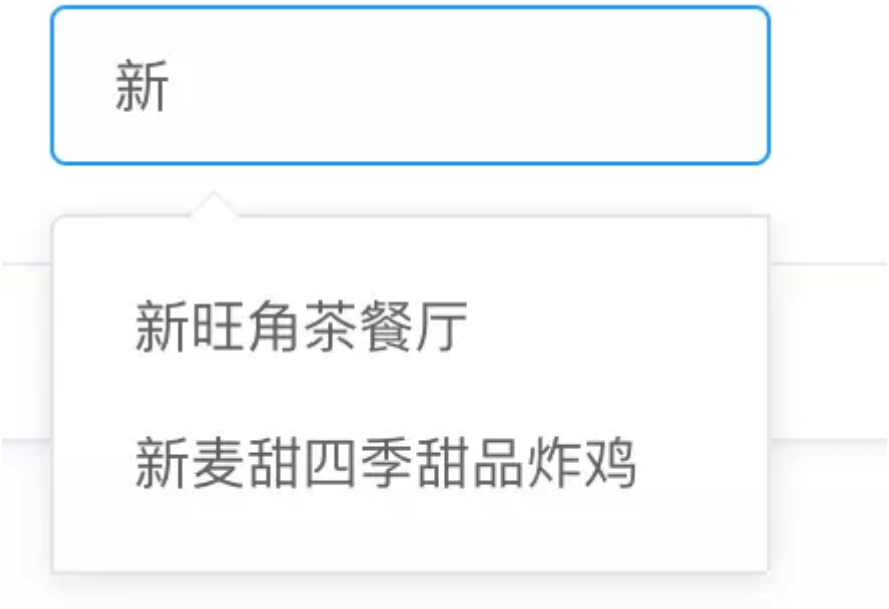
所以 before 函数就是让某个方法最多调用 n 次。

注：before 函数代码不难，使用也不难，但就是这么一个简单的工具方法需要了解的知识点有：作用域、闭包、高阶函数，所以说知识点都是连贯的，接下来要写的 **JavaScript 设计模式系列**，同样也要用到这些知识。

函数防抖

在写代码的时候，大多数情况都是由我们自己主动去调用函数。不过在有一些情况下，函数的调用不是由用户直接控制的，在这种情况下，函数有可能被废除频繁的调用，从而造成性能问题。

在 Element-UI 中，有一个 el-autocomplete 组件，该组件可以在用户输入的时候在输入框下方列出相关输入项：



其实就是可以在用户输入的时候，可以用已经输入的内容做搜索，饿了么在实现该组件的时候是利用 input 组件，并且监听用户的输入：

```
<el-input
  ref="input"
  v-bind="$props"
  @input="handleChange"
  @focus="handleFocus"
  @blur="handleBlur"
  @keydown.up.native.prevent="highlight(highlightedIndex - 1)"
  @keydown.down.native.prevent="highlight(highlightedIndex + 1)"
  @keydown.enter.native="handleKeyEnter"
  @keydown.native.tab="close"
  :label="label"
```

用 input 事件去监听用户输入的话，用户输入的每一个字都会触发该方法，如果是要用输入的内容去做网络搜索，用户输入的每一字都搜索的话，触发的频率太高了，性能消耗就有点大了，而且在网络比较差的情况下用户体验也比较不好。

饿了么实现该组件的时候当然也考虑到了这些问题，用的是业界比较通用的做法→**节流**，就是当输入后，延迟一段时间再去执行搜索，如果该次延迟执行还没有完成的话，就忽略接下来搜索的请求。

看一下其实现：

autocomplete 的防抖思想就是刚才说的那种，并且用了 throttle-debounce 这个工具库，其实现就是利用高阶函数，有兴趣的同学可以看它的源码：代码并不复杂。