JavaScript闭包

闭包是什么?

首先来看下MDN（Mozilla Developer Network）官网对于闭包这一概念的定义

> 闭包（closure）是一个函数以及其捆绑的周边环境状态（lexical environment，词法环境）的引用的组合。换而言之，闭包让开发者可以从内部函数访问外部函数的作用域。在 JavaScript 中，闭包会随着函数的创建而被同时创建。——MDN官网(相关链接)

读起来不太好理解，实际上翻译成白话文就是：**在一个作用域中可以访问另一个函数内部的局部变量的函数。**

下面是闭包的一个基本使用

```
function makeFunc() {
    var name = "Mozilla";
    function displayName() {
        alert(name);
    }
    return displayName;
}

var myFunc = makeFunc();
myFunc();
```

可以发现在`displayName`这个作用域下访问了另外一个函数`makeFunc`下的局部变量`name`

闭包的实现，实际上是利用了`JavaScript`中作用域链的概念，简单理解就是：在`JavaScript`中，如果在某个作用域下访问某个变量的时候，如果不存在，就一直向外层寻找，直到在全局作用域下找到对应的变量为止，这里就形成了所谓的作用域链。

闭包的特性

1. 闭包可以访问到父级函数的变量
2. 访问到父级函数的变量不会销毁

现在来看下闭包的相关应用，首先来看下下面这段代码：

```
var age = 18;

function person(){
    age++;
    console.log(age);
}

person(); // 19
person(); // 20
person(); // 21
```

可以看到这里调用了3次函数，`age`的值也从18增长到了21，但是这么写会导致全局变量被污染，所以将`age`的定义移动到`person`函数内部，代码如下：

```
function person() {
  var age = 18;
  age++;
  console.log(age);
}

person(); // 19
person(); // 19
person(); // 19
```

但是这又导致了另一个问题，变为局部变量的`age`不会自增了，所以那么就可以利用闭包的这个特性将每次调用时的`age`保存起来这样就可以实现变量的自增了，代码如下：

```
function person() {
  var age = 18;
  return function(){
    age++;
    console.log(age);
  }
}

let getPersonAge = person();
getPersonAge(); // 19
getPersonAge(); // 20
getPersonAge(); // 21
```

可以这样理解，通过将`person`函数赋值给`getPersonAge`这个变量，可以看作如下代码

```
let getPersonAge = function(){
  age++;
  console.log(age);
}
```

每当调用`getPersonAge()`函数的时候，首先要获取`age`变量，因为`JavaScript`中存在作用域链的关系，所以会从`person`函数下得到对应的`age`，因为闭包存在着**闭包可以访问到父级函数的变量，且该变量不会销毁**的特性所以上次的变量会被保留下来，所以可以做到自增的实现。

如果对**变量不会销毁**这一特性有疑问可以参考下寸志老师对于闭包的理解：

> 函数当作值传递，即所谓的first class对象。就是可以把函数当作一个值来赋值，当作参数传给别的函数，也可以把函数当作一个值 return。一个函数被当作值返回时，也就相当于返回了一个通道，这个通道可以访问这个函数词法作用域中的变量，即函数所需要的数据结构保存了下来，**数据结构中的值在外层函数执行时创建，外层函数执行完毕时理因销毁，但由于内部函数作为值返回出去，这些值得以保存下来**。而且无法直接访问，必须通过返回的函数。这也就是私有性。

闭包的应用

所以就可以根据这个特性做几个小案例测试一下。

循环注册事件

比如就可以利用闭包的特性做循环点击事件，比如下面的给输入框添加`onblur`事件：

需求：点击输入框，上面的提示栏显示对应的内容

```
<p id="help">Helpful notes will appear here</p>
<p>E-mail: <input type="text" id="email" name="email"></p>
<p>Name: <input type="text" id="name" name="name"></p>
<p>Age: <input type="text" id="age" name="age"></p>
<script>
  function showHelp(help) {
    document.getElementById('help').innerHTML = help;
  }

  function setupHelp() {
    var helpText = [
      { 'id': 'email', 'help': 'Your e-mail address' },
      { 'id': 'name', 'help': 'Your full name' },
      { 'id': 'age', 'help': 'Your age (you must be over 16)' }
    ];

    for (var i = 0; i < helpText.length; i++) {
      // var func = function (i) {
      //   document.getElementById(helpText[i].id).onfocus = function () {
      //     showHelp(helpText[i].help);
      //   }
      // };
      // func(i);
      (function (i) {
        document.getElementById(helpText[i].id).onfocus = function () {
          showHelp(helpText[i].help);
        }
```

```
      })(i);
    }
  }

  setupHelp();
</script>
```

PS：这里如果不想用闭包的话，可以使用ES2015中引入的`let`以及`const`关键字，或者使用`forEach`遍历`helpText`时给对应的`item`添加`focus`事件都可以解决

循环中的定时器

```
var lis = document.querySelector('.test').querySelectorAll('li');
for (var i = 0; i < lis.length; i++) {
  // var fc = function (i) {
  //   setTimeout(function () {
  //     console.log(lis[i].innerHTML);
  //   }, 3000);
  // };
  // fc(i);
  (function (i) {
    setTimeout(function () {
      console.log(lis[i].innerHTML);
    }, 3000);
  })(i);
}
```

案例1与2的总结：利用立即执行函数所形成的闭包来保存当前循环中的`i`的值，进而解决异步任务所带来的`i`最后为4（循环结束后`i`的值）的问题

模拟私有方法

下面的示例展现了如何使用闭包来定义公共函数，并令其可以访问私有函数和变量：

```
Countervar Counter = function(){
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function(){
      return changeBy(1);
    },
    decrement: function(){
      return changeBy(-1);
    },
    getValue: function(){
      return privateCounter;
```

```
    }
  }
}

var counterInstance = Counter();
console.log(counterInstance.getValue()); // 0
counterInstance.increment();
counterInstance.increment();
counterInstance.increment();
console.log(counterInstance.getValue()); // 3
counterInstance.decrement();
console.log(counterInstance.getValue()); // 2
```

还可以将`Counter`存在其他变量中以便可以形成多个计数器

```
var counterInstance1 = Counter();
var counterInstance2 = Counter();
// c1 计数器1
console.log(counterInstance1.getValue()); // 0
counterInstance1.increment();
counterInstance1.increment();
counterInstance1.increment();
console.log(counterInstance1.getValue()); // 3
counterInstance1.decrement();
console.log(counterInstance1.getValue()); // 2

// c2 计数器2
console.log(counterInstance2.getValue()); // 0
counterInstance2.increment();
counterInstance2.increment();
console.log(counterInstance2.getValue()); // 2
counterInstance2.decrement();
counterInstance2.decrement();
counterInstance2.decrement();
console.log(counterInstance2.getValue()); // -1
```

性能考量

如果不是某些特定任务需要使用闭包，在其它函数中创建函数是不明智的，**因为闭包在处理速度和内存消耗方面对脚本性能具有负面影响。**

其导致主要原因可以参考上面寸志老师的回答，这会**导致变量不会被垃圾回收机制回收，造成内存消耗**以及**对于不恰当的使用闭包可能会造成内存泄漏的问题**

比如在定义类的时候吧对应的方法定义在了构造函数下，这样就会导致每次实例化对象的时候，每个方法都会被重新赋值：

```
function Sony(camera, price){
  this.camera = camera;
  this.price = price;
  this.getCamera = function(){
    return this.camera;
  }
  this.getPrice = function(){
    return this.price;
  }
}

let s1 = new Sony('ZV-1', 5300);
```

通常情况下，都会将`getCamera`和`getPrice`放在原型对象下

```
Sony.prototypefunction Sony(camera, price){
  this.camera = camera;
  this.price = price;
}

Sony.prototype.getCamera = function(){
    return this.camera;
};

Sony.prototype.getPrice = function(){
    return this.price;
};
let s1 = new Sony('ZV-1', 5300);
```

不推荐使用`Sony.prototype = {fun1:{},fun2:{}}`的形式，这样相当于是重写了`Sony.prototype`这个原型对象

内存泄漏的解决方案

先来看下面这个案例

```
this.name = 'WindowName'
let myObj = {
  name: 'beast senpai',
  get: function(){
    return function(){
      console.log(this); // WindowName
      return this.name;
    }
  }
}

let myObjname = myObj.get()();
console.log(myObjname); // WindowName
```

这里发生了内存泄漏使得`this`指向了`Window`对象（`myObj.get()()`这种写法和立即执行函数很类似，立即执行函数的`this`指向`Window`）

解决方案1：在`get`函数使用`that`保存此时的`this`

```
this.name = 'WindowName'
let myObj = {
  name: 'beast senpai',
  get: function(){
    let that = this;
    return function(){
      console.log(that); // myObj
      return that.name;
    }
  }
}

let myObjname = myObj.get()();
console.log(myObjname); // beast senpai
```

解决方案2：将`get`函数的返回值改回使用箭头函数的方式做返回

```
this.name = 'WindowName'
let myObj = {
  name: 'beast senpai',
  get: function(){
    return ()=>{
      console.log(this); // myObj
      return this.name;
    }
  }
}

let myObjname = myObj.get()();
console.log(myObjname); // beast senpai
```

消除闭包

不用的时候解除引用，避免不必要的内存占用

取消`fn`对外部成员变量的引用，就可以回收相应的内存空间。

```
function add() {
  var count = 0
  return function fn() {
    count++
    console.log(count)
  }
}

var a = add() // 产生了闭包
a() // 1
a() // 2
a = null // 取消 a 与 fn 的联系，这个时候浏览器回收机制就能回收闭包空间
```

总结

闭包的作用：

1. 延申了变量的作用范围
2. 隐藏变量，避免全局污染

闭包的缺点：

1. 因为垃圾回收机制的存在，会导致出现不必要的性能消耗
2. 不恰当的使用会出现内存泄漏