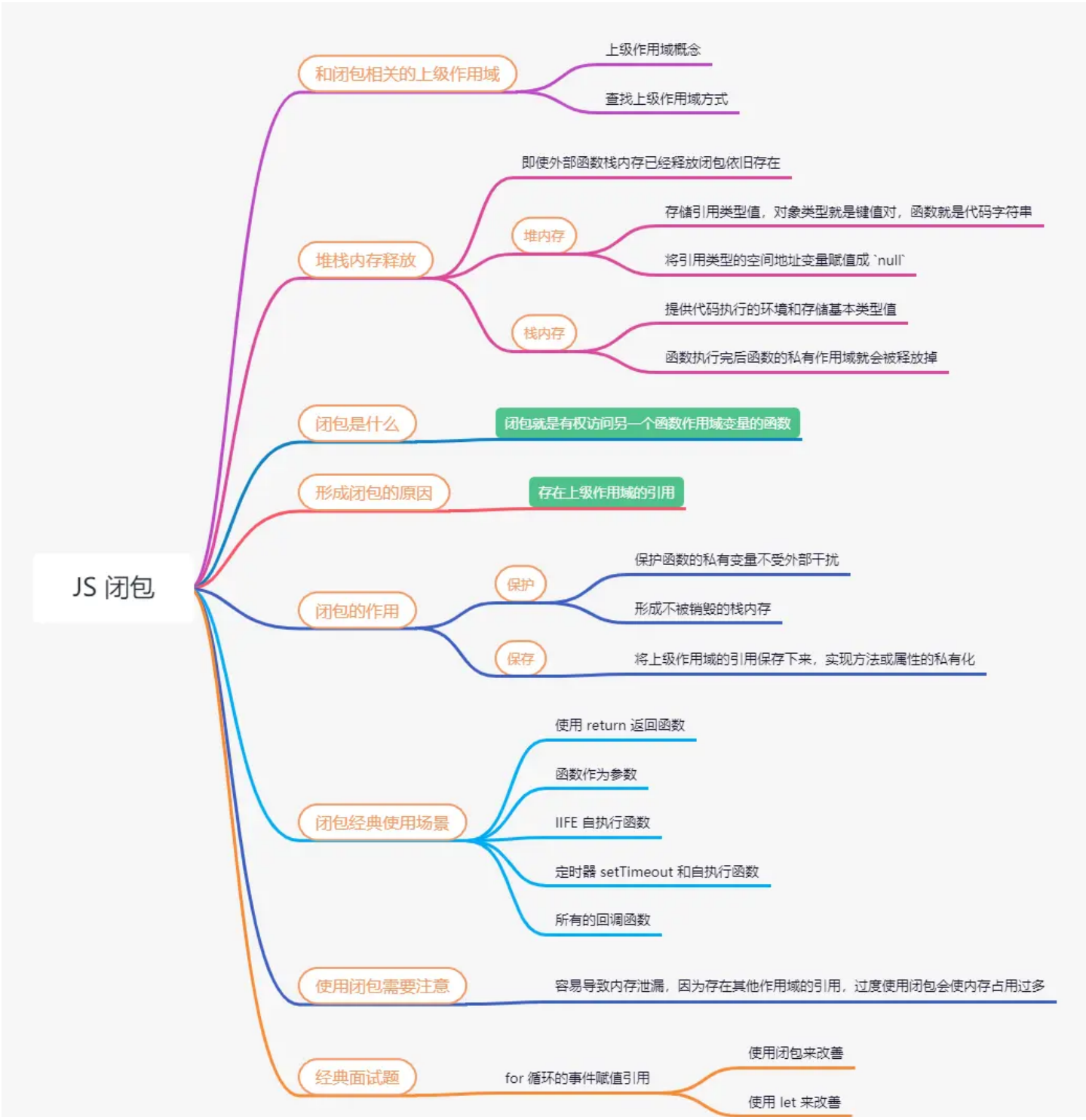
JS 闭包经典使用场景和含闭包必刷题

JS 闭包经典使用场景和含闭包必刷题

思维导图



闭包

> 了解闭包前先来了解一下上级作用域和堆栈内存释放问题。

上级作用域的概念

- 函数的上级作用域在哪里创建创建的，上级作用域就是谁

```
var a = 10
function foo(){
    console.log(a)
}

function sum() {
    var a = 20
    foo()
}

sum()
/* 输出
    10
/
```

> 函数 `foo()` 是在全局下创建的，所以 `a` 的上级作用域就是 `window`，输出就是 `10`

思考题

```
var n = 10
function fn(){
    var n =20
    function f() {
       n++;
       console.log(n)
     }
    f()
    return f
}

var x = fn()
x()
x()
console.log(n)
/* 输出
*  21
    22
    23
```

```
20      10
21  /
```

稍微提个醒，单独的 n++ 和 ++n 表达式的结果是一样的

> **思路：`fn` 的返回值是什么变量 `x` 就是什么，这里 `fn` 的返回值是函数名 `f` 也就是 `f` 的堆内存地址，`x()` 也就是执行的是函数 `f()`，而不是 `fn()`，输出的结果显而易见**

- 关于如何查找上级作用域

> 参考：彻底解决 JS 变量提升的面试题

JS 堆栈内存释放

- 堆内存：存储引用类型值，对象类型就是键值对，函数就是代码字符串。
- 堆内存释放：将引用类型的空间地址变量赋值成 `null`，或没有变量占用堆内存了浏览器就会释放掉这个地址
- 栈内存：提供代码执行的环境和存储基本类型值。
- 栈内存释放：一般当函数执行完后函数的私有作用域就会被释放掉。

> **但栈内存的释放也有特殊情况：① 函数执行完，但是函数的私有作用域内有内容被栈外的变量还在使用的，栈内存就不能释放里面的基本值也就不会被释放。② 全局下的栈内存只有页面被关闭的时候才会被释放**

闭包是什么

> 在 JS 忍者秘籍(P90)中对闭包的定义：闭包允许函数访问并操作函数外部的变量。红宝书上对于闭包的定义：闭包是指有权访问另外一个函数作用域中的变量的函数。 MDN 对闭包的定义为：闭包是指那些能够访问自由变量的函数。这里的自由变量是外部函数作用域中的变量。
>
> 概述上面的话，**闭包是指有权访问另一个函数作用域中变量的函数**

形成闭包的原因

> **内部的函数存在外部作用域的引用就会导致闭包**。从上面介绍的上级作用域的概念中其实就有闭包的例子 `return f` 就是一个表现形式。

```
1  var a = 0
2  function foo(){
3      var b =14
4      function fo(){
5          console.log(a, b)
6      }
7      fo()
8  }
```

```
9 foo()
```

> 这里的子函数 `fo` 内存就存在外部作用域的引用 `a, b`，所以这就会产生闭包

闭包变量存储的位置

> 直接说明：**闭包中的变量存储的位置是堆内存。**

- 假如闭包中的变量存储在栈内存中，那么栈的回收 会把处于栈顶的变量自动回收。所以闭包中的变量如果处于栈中那么变量被销毁后，闭包中的变量就没有了。所以闭包引用的变量是出于堆内存中的。

闭包的作用

- 保护函数的私有变量不受外部的干扰。形成不销毁的栈内存。
- 保存，把一些函数内的值保存下来。闭包可以实现方法和属性的私有化

闭包经典使用场景

1. `return` 回一个函数

```
1 var n = 10
2 function fn(){
3     var n =20
4     function f() {
5        n++;
6        console.log(n)
7      }
8     return f
9 }
10
11 var x = fn()
12 x() // 21
13
```

> 这里的 return `f`，`f()` 就是一个闭包，存在上级作用域的引用。

1. 函数作为参数

```
1 var a = '林——'
2 function foo(){
3     var a = 'foo'
4     function fo(){
5         console.log(a)
```

```
    }
    return fo
}

function f(p){
    var a = 'f'
    p()
}
f(foo())
/* 输出
*   foo
/


```

> 使用 return `fo` 返回回来，`fo()` 就是闭包，`f(foo())` 执行的参数就是函数 `fo`，因为 `fo() 中的 a` 的上级作用域就是函数 `foo()`，所以输出就是 `foo`

1. IIFE（自执行函数）

```
var n = '林一一';
(function p(){
    console.log(n)
})()
/* 输出
*   林一一
/

```

> 同样也是产生了闭包 `p()`，存在 `window` 下的引用 `n`。

1. 循环赋值

```
for(var i = 0; i<10; i++){
  (function(j){
       setTimeout(function(){
        console.log(j)
    }, 1000)
  })(i)
}

```

> 因为存在闭包的原因上面能依次输出0~9，闭包形成了10个互不干扰的私有作用域。将外层的自执行函数去掉后就不存在外部作用域的引用了，输出的结果就是连续的 10。为什么会连续输出10，因为 JS 是单线程的遇到异步的代码不会先执行(会入栈)，等到同步的代码执行完 `i++` 到 10时，异步代码才开始执行此时的 `i=10` 输出的都是 10。

1. 使用回调函数就是在使用闭包

```
window.name = '林一一'
setTimeout(function timeHandler(){
  console.log(window.name);
}, 100)

```

1. 节流防抖

```
// 节流
function throttle(fn, timeout) {
    let timer = null
    return function (...arg) {
        if(timer) return
        timer = setTimeout(() => {
            fn.apply(this, arg)
            timer = null
        }, timeout)
    }
}

// 防抖
function debounce(fn, timeout){
    let timer = null
    return function(...arg){
        clearTimeout(timer)
        timer = setTimeout(() => {
            fn.apply(this, arg)
        }, timeout)
    }
}
```

1. 柯里化实现

```
function curry(fn, len = fn.length) {
    return _curry(fn, len)
```

```
}

function _curry(fn, len, ...arg) {
    return function (...params) {
        let _arg = [...arg, ...params]
        if (_arg.length >= len) {
            return fn.apply(this, _arg)
        } else {
            return _curry.call(this, fn, len, ..._arg)
        }
    }
}

let fn = curry(function (a, b, c, d, e) {
    console.log(a + b + c + d + e)
})

fn(1, 2, 3, 4, 5)  // 15
fn(1, 2)(3, 4, 5)
fn(1, 2)(3)(4)(5)
fn(1)(2)(3)(4)(5)

```

使用闭包需要注意什么

> 容易导致内存泄漏。闭包会携带包含其它的函数作用域，因此会比其他函数占用更多的内存。过度使用闭包会导致内存占用过多，所以要谨慎使用闭包。

怎么检查内存泄露

- performance 面板 和 memory 面板可以找到泄露的现象和位置

详细可以查看：js 内存泄漏场景、如何监控以及分析

经典面试题

- for 循环和闭包(号称必刷题)

```
var data = [];

for (var i = 0; i < 3; i++) {
  data[i] = function () {
    console.log(i);
  };
}

```

```
data[0]();
data[1]();
data[2]()
/* 输出
    3
    3
    3
/
```

> 这里的 `i` 是全局下的 `i`，共用一个作用域，当函数被执行的时候这时的 `i=3`，导致输出的结构都是3。

- 使用闭包改善上面的写法达到预期效果，写法1：自执行函数和闭包

```
var data = [];

for (var i = 0; i < 3; i++) {
    (function(j){
      setTimeout( data[j] = function () {
        console.log(j);
      }, 0)
    })(i)
}

data[0]();
data[1]();
data[2]()
```

- 写法2：使用 `let`

```
var data = [];

for (let i = 0; i < 3; i++) {
  data[i] = function () {
    console.log(i);
  };
}

data[0]();
data[1]();
data[2]()
```

```
12
```

> `let` 具有块级作用域，形成的3个私有作用域都是互不干扰的。

思考题和上面有何不同 (字节)

```
var result = [];
var a = 3;
var total = 0;

function foo(a) {
    for (var i = 0; i < 3; i++) {
        result[i] = function () {
            total += i * a;
            console.log(total);
        }
    }
}

foo(1);
result[0]();  // 3
result[1]();  // 6
result[2]();  // 9

```

> tip：这里也形成了闭包。total 被外层引用没有被销毁。