【讲义】2021大厂前端高频代码题（二）

#2021#

一、闭包的概念及应用场景

定义

闭包是指那些能够访问自由变量的函数。
自由变量是指在函数中使用的，但既不是函数参数也不是函数局部变量的变量。

1. 从理论角度：所有的函数都是闭包。因为它们都在创建的时候就将上层上下文的数据保存起来了。哪怕是简单的全局变量也是如此，因为函数中访问全局变量就相当于是在访问自由变量，这个时候使用最外层的作用域。
2. 从实践角度：以下函数才算是闭包：
   - 即使创建它的上下文已经销毁，它仍然存在（比如，内部函数从父函数中返回）
   - 在代码中引用了自由变量

应用场景

1. 柯里化函数
   柯里化的目的在于：避免频繁调用具有相同参数函数，同时又能够轻松的复用。
   其实就是封装一个高阶函数。

```
// 假设我们有一个求长方形面积的函数
function getArea(width, height) {
    return width * height
}
// 如果我们碰到的长方形的宽老是10
const area1 = getArea(10, 20)
const area2 = getArea(10, 30)
const area3 = getArea(10, 40)

// 我们可以使用闭包柯里化这个计算面积的函数
function getArea(width) {
    return height => {
```

```
        return width * height
    }
}

const getTenWidthArea = getArea(10)
// 之后碰到宽度为10的长方形就可以这样计算面积
const area1 = getTenWidthArea(20)

// 而且如果遇到宽度偶尔变化也可以轻松复用
const getTwentyWidthArea = getArea(20)
```

2. 使用闭包实现私有方法/变量

   其实就是模块的方式, 现代化的打包最终其实就是每个模块的代码都是相互独立的。

```
function funOne(i){
    function funTwo(){
        console.log('数字: ' + i);
    }
    return funTwo;
};
var fa = funOne(110);
var fb = funOne(111);
var fc = funOne(112);
fa();       // 输出: 数字: 110
fb();       // 输出: 数字: 111
fc();       // 输出: 数字: 112
```

3. 匿名自执行函数

```
var funOne = (function(){
  var num = 0;
  return function(){
      num++;
      return num;
```

```
  }
})();
console.log(funOne());       // 输出: 1
console.log(funOne());       // 输出: 2
console.log(funOne());       // 输出: 3
```

4. 缓存一些结果
比如在外部函数创建一个数组, 闭包函数内可以更改/获取这个数组的值，其实还是延长变量的生命周期，但是不通过全局变量来实现。

```
function funParent(){
    let memo = [];
    function funTwo(i){
        memo.push(i);
        console.log(memo.join(','))
    }
    return funTwo;
};

const fn = funParent();

fn(1);
fn(2);
```

总结

- 创建私有变量
- 延长变量的生命周期

一般函数的词法环境在函数返回后就被销毁，但是闭包会保存对创建时所在词法环境的引用，即便创建时所在的执行上下文被销毁，但创建时所在词法环境依然存在，以达到延长变量的生命周期的目的

代码题

1. 实现compose函数, 得到如下输出

```
// 实现一个compose函数, 用法如下:
function fn1(x) {
    return x + 1;
}

function fn2(x) {
    return x + 2;
}

function fn3(x) {
    return x + 3;
}

function fn4(x) {
    return x + 4;
}


const a = compose(fn1, fn2, fn3, fn4);
console.log(a(1)); // 1+4+3+2+1=11
```

2. 实现一个柯里化函数

```
function currying() {

}

const add = (a, b, c) => a + b + c;
const a1 = currying(add, 1);
const a2 = a1(2);
console.log(a2(3)) // 6
```

3. 实现一个简单的koa洋葱模型compose

```
// 题目需求

let middleware = []
middleware.push((next) => {
    console.log(1)
    next()
    console.log(1.1)
})
middleware.push((next) => {
    console.log(2)
    next()
    console.log(2.1)
})
middleware.push((next) => {
    console.log(3)
    next()
    console.log(3.1)
})

let fn = compose(middleware)
fn()

/*
1
2
3
3.1
2.1
1.1
*/

//实现compose函数

function compose(middlewares) {

}
```

二、平时用过发布订阅模式吗？比如Vue的event bus, node的eventemitter3

1. 这种模式, 事件的触发 和 回调之间是同步的还是异步的?

```
代码
```

2. 实现一个简单的EventEmitter?

```
代码题，实现eventEmitter，包含on emit once，off四个方法
```

3. 设置最大监听数?

```
代码
```

三、网络和并发

HTTP 1.0/1.1/2.0在并发请求上主要区别是什么?

1. HTTP/1.0
   每次TCP连接只能发送一个请求，当服务器响应后就会关闭这次连接，下一个请求需要再次建立TCP连接.

2. HTTP/1.1
   默认采用持续连接(TCP连接默认不关闭，可以被多个请求复用，不用声明Connection: keep-alive).

增加了管道机制，在同一个TCP连接里，允许多个请求同时发送，增加了并发性，进一步改善了HTTP协议的效率，
但是同一个TCP连接里，所有的数据通信是按次序进行的。回应慢，会有许多请求排队，造成"队头堵塞"。

3. HTTP/2.0

加了双工模式，即不仅客户端能够同时发送多个请求，服务端也能同时处理多个请求，解决了队头堵塞的问题。
使用了多路复用的技术，做到同一个连接并发处理多个请求，而且并发请求的数量比HTTP1.1大了好几个数量级。
增加服务器推送的功能，不经请求服务端主动向客户端发送数据。

HTTP/1.1长连接和HTTP/2.0多路复用的区别?

HTTP/1.1：同一时间一个TCP连接只能处理一个请求, 采用一问一答的形式, 上一个请求响应后才能处理下一个请求. 由于浏览器最大TCP连接数的限制, 所以有了最大并发请求数的限制.
HTTP/2.0：同域名下所有通信都在单个连接上完成，消除了因多个 TCP 连接而带来的延时和内存消耗。单个连接上可以并行交错的请求和响应，之间互不干扰

那为什么HTTP/1.1不能实现多路复用?

HTTP/2是基于二进制"帧"的协议，HTTP/1.1是基于"文本分割"解析的协议。

HTTP1.1的报文结构中, 服务器需要不断的读入字节，直到遇到换行符, 或者说一个空白行. 处理顺序是串行的, 一个请求和一个响应需要通过一问一答的形式才能对应起来.

```
GET / HTTP/1.1
Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding:gzip, deflate, br
Accept-Language:zh-CN,zh;q=0.9,en;q=0.8
Cache-Control:max-age=0
Connection:keep-alive
Host:www.imooc.com
Referer:https://www.baidu.com/
```

HTTP2.0中，有两个非常重要的概念，分别是帧（frame）和流（stream）。
帧代表着最小的数据单位，每个帧会标识出该帧属于哪个流，流也就是多个帧组成的数据流。
多路复用，就是在一个 TCP 连接中可以存在多条流。换句话说，也就是可以发送多个请求，对端可以通过帧中的标识知道属于哪个请求。通过这个技术，可以避免 HTTP 旧版本中的队头阻塞问题，极大的提高传输性能。

前端代码里有什么方式能控制最大并发量吗?

代码, 你能写出几种方式?

如果任务有优先级的概念, 需要允许高优任务的插入呢?

代码

四、 数据结构相关

字符串/Map等等的算法应用.

1. 压缩字符串

```
//  压缩字符串 'aaaaaabbbbbcccca' => 'a6b5c4a1'

function compress(str) {

}

console.log(compress('aaaaaabbbbbcccca'))
```

2. 解压字符串

```
// const template = 'ab2[c]2[b3[c]]' // abccbcccbccc
const template = '[ab]2[bc]2d';

function transStr(template) {
```

```
}

console.log(transStr(template))
```

3. 实现LRU算法

```
/**
 * LRU是Least Recently Used的缩写，即最近最少使用，是一种常用的页面置换算法，选择最近最久未使用的页面予以淘汰。
 * 该算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 t，当须淘汰一个页面时，选择现有页面中其 t 值最大的，即最近最少使用的页面予以淘汰。
 */

class LRU {

}

const lRUCache = new LRU(2);
lRUCache.put(1, 1); // 缓存是 {1=1}
lRUCache.put(2, 2); // 缓存是 {1=1, 2=2}
console.log(lRUCache.get(1)); // 返回 1
lRUCache.put(3, 3); // 该操作会使得关键字 2 作废，缓存是 {1=1, 3=3}
console.log(lRUCache.get(2)); // 返回 -1 (未找到)
lRUCache.put(4, 4); // 该操作会使得关键字 1 作废，缓存是 {4=4, 3=3}
console.log(lRUCache.get(1)); // 返回 -1 (未找到)
console.log(lRUCache.get(3)); // 返回 3
console.log(lRUCache.get(4)); // 返回 4
```

五、防抖和节流的基本概念?

- 函数防抖（debounce）：当持续触发事件时，一定时间段内没有再触发事件，事件处理函数才会执行一次，如果设定的时间到来之前，又一次触发了事件，就重新开始延时。如下图，持续触发scroll事件时，并不执行handle函数，当1000毫秒内没有触发scroll事件时，才会延时触发scroll事件。 debounce.webp

- 函数节流（throttle）：当持续触发事件时，保证一定时间段内只调用一次事件处理函数。节流通俗解释就比如我们水龙头放水，阀门一打开，水哗哗的往下流，秉着勤俭节约的优良传统美德，我们要把水龙头关小点，最好是如我们心意按照一定规律在某个时间间隔内一滴一滴的往下滴。如下图，持续触发scroll事件时，并不立即执行handle函数，每隔1000毫秒才会执行一次handle函数。

分别适合用在什么场景?

节流：resize scroll
防抖：input输入

来写一个节流? 还有其他方式实现吗?

代码

首节流和尾节流?

代码