08.15公开课 - 2021大厂前端核心面试题详解

#2021#

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

平时有关注过前端的内存处理吗?

1. 你了解js中的内存管理吗? 什么情况会导致内存泄露?

1. 内存的生命周期

```
内存分配：当我们申明变量、函数、对象的时候，系统会自动为他们分配内存
内存使用：即读写内存，也就是使用变量、函数等
内存回收：使用完毕，由垃圾回收机制自动回收不再使用的内存
```

2. Js中的内存分配

```
const n = 123; // 给数值变量分配内存
const s = “azerty”; // 给字符串分配内存
const o = {
a: 1,
b: null
}; // 给对象及其包含的值分配内存
```

3. Js中的内存使用

使用值的过程实际上是对分配内存进行读取与写入的操作。 读取与写入可能是写入一个变量或者一个对象的属性值，甚至传递函数的参数。

```
var a = 10; // 分配内存
console.log(a); // 对内存的使用
```

4. Js中的垃圾回收机制

垃圾回收算法主要依赖于引用的概念。
在内存管理的环境中，一个对象如果有访问另一个对象的权限（隐式或者显式），叫做一个对象引用另一个对象。
例如，一个Javascript对象具有对它原型的引用（隐式引用）和对它属性的引用（显式引用）。
在这里，“对象”的概念不仅特指 JavaScript 对象，还包括函数作用域（或者全局词法作用域）。

4.1 引用计数垃圾回收

引用计数算法定义“内存不再使用”的标准很简单，就是看一个对象是否有指向它的引用。 如果没有其他对象指向它了，说明该对象已经不再需了。
但它却存在一个致命的问题：循环引用。
如果两个对象相互引用，尽管他们已不再使用，垃圾回收不会进行回收，导致内存泄露。

4.2 标记清除算法

标记清除算法将"不再使用的对象"定义为"无法达到的对象"。 简单来说，就是从根部（在JS中就是全局对象）出发定时扫描内存中的对象。 凡是能从根部到达的对象，都是还需要使用的。 那些无法由根部出发触及到的对象被标记为不再使用，稍后进行回收。

4.2.1 垃圾收集器在运行的时候会给存储在内存中的所有变量都加上标记。

4.2.2 从根部出发将能触及到的对象的标记清除。

4.2.3 那些还存在标记的变量被视为准备删除的变量。

4.2.4 最后垃圾收集器会执行最后一步内存清除的工作，销毁那些带标记的值并回收它们所占用的内存空间。

5. 常见的内存泄露

5.1 全局变量

```
function foo() {
    bar1 = 'some text'; // 没有声明变量 实际上是全局变量 => window.bar1
    this.bar2 = 'some text' // 全局变量 => window.bar2
}
foo();
```

5.2 未被清理的定时器和回调函数

如果后续 renderer 元素被移除，整个定时器实际上没有任何作用。 但如果你没有回收定时器，整个定时器依然有效, 不但定时器无法被内存回收， 定时器函数中的依赖也无法回收。在这个案例中的 serverData 也无法被回收。

```
var serverData = loadData();
setInterval(function() {
    var renderer = document.getElementById('renderer');
    if(renderer) {
        renderer.innerHTML = JSON.stringify(serverData);
    }
}, 5000); // 每 5 秒调用一次
```

5.3 闭包

在 JS 开发中，我们会经常用到闭包，一个内部函数，有权访问包含其的外部函数中的变量。
下面这种情况下，闭包也会造成内存泄露

```
var theThing = null;
var replaceThing = function () {
var originalThing = theThing;
var unused = function () {
    if (originalThing) // 对于 'originalThing'的引用
    console.log("hi");
};
theThing = {
    longStr: new Array(1000000).join('*'),
    someMethod: function () {
    console.log("message");
    }
};
};
setInterval(replaceThing, 1000);
```

这段代码，每次调用 replaceThing 时，theThing 获得了包含一个巨大的数组和一个对于新闭包 someMethod 的对象。 同时 unused 是一个引用了 originalThing 的闭包。
这个范例的关键在于，闭包之间是共享作用域的，尽管 unused 可能一直没有被调用，但是 someMethod 可能会被调用，就会导致无法对其内存进行回收。 当这段代码被反复执行时，内存会持续增长。

5.3 DOM引用

很多时候, 我们对 Dom 的操作, 会把 Dom 的引用保存在一个数组或者 Map 中。

```
var elements = {
    image: document.getElementById('image')
};
function doStuff() {
    elements.image.src = 'http://example.com/image_name.png';
}
function removeImage() {
```

```
    document.body.removeChild(document.getElementById('image'));
    // 这个时候我们对于 #image 仍然有一个引用, Image 元素, 仍然无法被内存回收.
}
```

上述案例中，即使我们对于 image 元素进行了移除，但是仍然有对 image 元素的引用，依然无法对齐进行内存回收。

6. 如何避免内存泄露

减少不必要的全局变量，使用严格模式避免意外创建全局变量。
在你使用完数据后，及时解除引用（闭包中的变量，dom引用，定时器清除）。
组织好你的逻辑，避免死循环等造成浏览器卡顿，崩溃的问题。

2. 实现sizeOf函数, Get size of a JavaScript object in Bytes

既然对内存这么了解，那么来一道代码题：
实现sizeOf函数, 计算传入的对象所占的Bytes数值.

了解JsBridge原理吗?

Hybrid最核心的就是Navite和H5的双向通讯, 而通讯是完全依赖于native提供的webview容器，那native提供的这个webview容器有什么特点能支撑起h5和native的通讯呢？具体的通讯流程到底是什么样子呢?

首先说明有两种方式：

- URL Schema， 客户端通过拦截webview请求来完成通讯
- native向webview中的js执行环境，注入API, 以此来完成通讯

一、URL Schema, 客户端拦截webview请求

1. 原理

在webview中发出的网络请求，都会被客户端监听和捕获到。

这是我们本节课所有实现的基石。

2. 定义自己的私有协议

上面说过, 所有网络请求都会被监听到, 网络请求最常见的就是http协议, 比如https://a.b.com/fetchInfo, 这是一个很常见的请求。

webview内的H5页面肯定有很多类似的http请求, 我们为了区别于业务请求, 需要定制一套h5和native进行交互的私有协议, 我们通常称呼为URL Schema。

比如我们现在定义协议头为 lubai://,

那么随后我们要在webview请求中都带上这个私有协议开头, 比如有一个请求是setLeftButton, 实际发出的请求会是lubai://setLeftButton?params1=xxx&params2=xxx.

这里大家记住, 这个协议的名称是我们自定义的, 只要h5和native协商好即可。
但是如果公司旗下有多个app, 对于通用的业务一般会定义一个通用的协议头, 比如common://；对于每个app自己比较独立的业务, 基本每个app都会自己定义一套协议, 比如appa://, appb://, appc://.

3. 请求的发送

对于webview请求的发送, 我们一般使用iframe的方式。也可以使用location.href的方式, 但是这种方式不适用并行发请求的场景。

```
const doc = window.document;
const body = window.document.body;
const iframe = doc.createElement('iframe');

iframe.style.display = 'none';
iframe.src = 'lubai://setLeftButton?param1=12313123';

body.appendChild(iframe);
setTimeout(() => {
    body.removeChild(iframe);
}, 200)
```

而且考虑到安全性, 客户端中一般会设置域名白名单, 比如客户端设置了lubai.com为白名单, 那么只有lubai.com域下发出的请求, 才会被客户端处理。

这样可以避免自己app内部的业务逻辑, 被第三方页面直接调用。

4. 客户端拦截协议请求

   iOS和Android对webview请求的拦截方法不太相同。

- iOS: shouldStartLoadWithRequest
- Android: shouldOverrideUrlLoading

   当客户端解析到请求的URL协议是约定要的lubai://时, 便会解析参数, 并根据h5传入的方法名比如setLeftButton, 来进行相关操作（设置返回按钮的处理逻辑）。

5. 请求处理完成后的回调

   因为咱们webview的请求本质上还是异步请求的过程, 当请求完成后, 我们需要有一个callback触发, 无论是通知h5执行结果，还是返回一些数据， 都离不开callback的执行。

   我们可以使用Js自带的事件机制，window.addEventListener和window.dispatchEvent这两个API。

   还是这个例子, 比如咱们现在要调用setLeftButton方法, 方法要传入一个callback来得知是否执行成功了。

```
webview.setLeftButton({ params1: 111 }, (err) => {
    if (err) {
        console.error('执行失败');
        return;
    }
    console.log('执行成功');
    // 业务逻辑
})
```

   JsBridge中具体的步骤应该是这样的：

- 在H5调用setLeftButton方法时, 通过 webview_api名称+参数 作为唯一标识,注册自定义事件

```
const handlerId = Symbol();
```

```
const eventName = `setLeftButton_${handlerId}`;
const event = new Event(eventName);
window.addEventListener(eventName, (res) => {
    if (res.data.errcode) {
        console.error('执行失败');
        return;
    }
    console.log('执行成功');
    // 业务逻辑
});

JsBridge.send(`lubai://setLeftButton?handlerId=${eventName}&params1=111`);
```

- 客户端在接收到请求, 完成自己的对应处理后, 需要调用JsBridge中的dispatch, 携带回调的数据触发自定义事件。

```
event.data = { errcode: 0 };
window.dispatchEvent(event);
```

注入API

上述方式有个比较大的缺点, 就是参数如果太长会被截断。以前用这种方式主要是为了兼容iOS6， 现在几乎已经不需要考虑这么低的版本了。

所以现在主流的实现是native向js的执行环境中注入API.

具体怎么操作呢, 咱们分步骤来看：

1. 向native传递信息

由于native已经向window变量注入了各种api, 所以咱们可以直接调用他们。

比如现在window.lubaiWebview = { setLeftButton: (params) => {}} 就是native注入的对象api。

我们可以直接这样调用, 就可以传参数给native了

```
window.lubaiWebview['setLeftButton'](params)
```

但是为了安全性, 或者为了不要乱码等问题, 我们一般会对参数进行编码, 比如转换为base64格式。

2. 准备接收native的回调

咱们同样可以在window上声明接收回调的api

```
window['setLeftButton_Callback_1'] = (errcode, response) => {
    console.log(errcode);
}
```

同样为了安全性和参数传递的准确性, native也会将回调的数据进行base64编码, 咱们需要在回调函数里进行解析。

3. native调用回调函数

native怎么知道哪个是这次的回调函数呢? 他们确实不知道, 所以我们需要在调用的时候就告诉native。

```
window.lubaiWebview['setLeftButton'](params)
```

这个Params中, 我们会加入一个属性叫做trigger, 它的值是回调函数的名称, 比如

```
const callbackName = 'setLeftButton_Callback_1';
window.lubaiWebview['setLeftButton']({
    trigger: callbackName,
    ...otherParams
});

window[callbackName] = (errcode, response) => {
    console.log(errcode);
```

```
}
```

同时为了保证callbackName的唯一性, 我们一般会加入各种Date.now() + id, 使其保证唯一。

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?

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. 如何控制最大监听数?

修改代码实现

贪心算法

贪心算法其实可以认为是dp问题的一个特例, 除了动态规划的各种特征外, 贪心算法还需要满足"贪心选择性质", 当然效率也比动态规划要高。

- 贪心选择性质：每一步走做出一个局部最优的选择，最终的结果就是全局最优。

同学们肯定一眼就看出问题来了, 并不是所有问题都是这样的, 很多问题局部最优并不能保证全局最优, 只有小部分问题具有这种特质。

比如你前面堆满了金条, 你只能拿5根, 怎么保证拿到的价值最大？ 答案当然是：每次都拿剩下的金条中最重的那根, 那么最后你拿到的一定是最有价值的。

比如斗地主, 对方出了一个3, 你手上有345678还有一个2, 按照贪心选择, 这时候你应该出4了, 实际上咱们会尝试出2, 然后345678起飞~

区间调度问题

来看一个经典的贪心算法问题 Interval Scheduling 区间调度.

有许多[start, end]的闭区间, 请设计一个算法, 算出这些区间中, 最多有几个互不相交的区间。

```
function intervalSchedule(intvs: number[][]) {}

// 比如intvs = [[1,3], [2,4], [3,6]]
// 这些区间最多有两个区间互不相交，即 [1,3], [3,6], intervalSchedule函数此时应该返回2
```

这个问题在现实中其实有很多应用场景, 比如你今天有好几个活动可以参加, 每个活动区间用[start, end]表示开始和结束时间, 请问你今天最多能参加几个活动?

贪心求解

大家可以先想一想，有什么思路?

1. 可以每次选择可选区间中开始最早的那个?

   不行, 因为可能有的区间开始很早, 结束很晚, 比如[0,10], 使我们错过了很多短区间比如[1,2], [2,3]

2. 可以每次选择可选区间中最短的那个?

   不行, 直接看上面这个例子1,3], [2,4], [3,6, 这样的话会选择出 [1, 3], [2, 4], 并不能保证他们不相交

正确思路

1. 从可选区间intvs里, 选择一个end最小的区间x
2. 把所有与x相交的区间从intvs中剔除
3. 重复1,2, 直到intvs为空, 之前选出的各种区间x， 就是我们要求的结果

把整个思路转换成代码的话, 因为我们要选出end最小的区间，所以我们可以先对区间根据end升序排序.

1. 选出end最小的区间

   由于我们已经排过序了, 所以直接选择第一个区间即可

2. 剔除与x相交的区间

   这一步就没第一步那么简单了, 这里建议大家画个图看看

- 代码如下

看一下区间调度.js

区间调度算法的应用

1. 无重叠区间

给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。

注意:

可以认为区间的终点总是大于它的起点。
区间 [1,2] 和 [2,3] 的边界相互“接触”，但没有相互重叠。
示例 1:

输入: [[1,2], [2,3], [3,4], [1,3]]

输出: 1

解释: 移除 [1,3] 后，剩下的区间没有重叠。

示例 2:

输入: [[1,2], [1,2], [1,2]]

输出: 2

解释: 你需要移除两个 [1,2] 来使剩下的区间没有重叠。

示例 3:

输入: [[1,2], [2,3]]

输出: 0

解释: 你不需要移除任何区间，因为它们已经是无重叠的了。

- 解答

刚才咱们已经找到了最多有几个互不相交的区间数n, 那么总数减去n就可以了~

咱们来从头写一遍, 就当是复习了

```
var eraseOverlapIntervals = function(intervals) {
    if (intervals.length === 0) {
        return 0;
    }
    let sortArray = intervals.sort((a,b) => a[1] - b[1]);

    let count = 1;

    let xEnd = sortArray[0][1];

    for (let item of intervals) {
        // 注意, 这里题目说了区间 [1,2] 和 [2,3] 的边界相互“接触”, 但没有相互重叠, 所以应该是item[0] >= xEnd
        if (item[0] >= xEnd) {
            xEnd = item[1];
            count++;
        }
    }

    return intervals.length - count;
};
```

2. 用最少的箭头射爆气球

在二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气球直径的开始和结束坐标。由于它是水平的，所以y坐标并不重要，因此只要知道开始和结束的x坐标就足够了。开始坐标总是小于结束坐标。平面内最多存在104个气球。

一支弓箭可以沿着x轴从不同点完全垂直地射出。在坐标x处射出一支箭，若有一个气球的直径的开始和结束坐标为 xstart，xend， 且满足 xstart ≤ x ≤ xend，则该气球会被引爆。可以射出的弓箭的数量没有限制。 弓箭一旦被射出之后，可以无限地前进。我们想找到使得所有气

球全部被引爆，所需的弓箭的最小数量。

Example:

输入:
10,16], [2,8], [1,6], [7,12

输出:
2

解释:
对于该样例，我们可以在x = 6（射爆[2,8],[1,6]两个气球）和 x = 11（射爆另外两个气球）。

- 解答

这个问题和区间调度算法又是非常的类似, 大家稍微转换一下思路即可。

如果最多有n个不重叠的空间, 那么就至少需要n个箭头穿透所有空间，所以我们要求的其实就是最多有几个不重叠的空间。

来看一下这张图

但是这个题里的描述, 边界重叠后, 箭头是可以一起射爆的, 所以两个区间的边界重叠也算是区间重叠。