**Web前端1**

1. 怎么理解 web 语义化；

根据内容的结构化（内容语义化），选择合适的标签（代码语义化）便于开发者阅读和写出更优雅的代码的同时让浏览器的爬虫和机器很好地解析。

* 为了在没有CSS的情况下，页面也能呈现出很好地内容结构、代码结构:为了裸奔时好看；
* 有利于[SEO](http://baike.baidu.com/view/1047.htm" \t "https://www.cnblogs.com/freeyiyi1993/p/_blank)：和搜索引擎建立良好沟通，有助于爬虫抓取更多的有效信息：[爬虫](http://baike.baidu.com/view/998403.htm" \t "https://www.cnblogs.com/freeyiyi1993/p/_blank)依赖于标签来确定上下文和各个关键字的权重；
* 方便其他设备解析（如屏幕阅读器、盲人阅读器、移动设备）以意义的方式来渲染网页；
* 便于团队开发和维护，语义化更具可读性，是下一步吧网页的重要动向，遵循W3C标准的团队都遵循这个标准，可以减少差异化。

<header><main><footer><article><section>

1. 如果去优化 html 怎么做；

**减少HTTP请求。**

**缓存。**

**压缩HTML、CSS、JavaScript**。

**优化首屏加载**。

**预加载**。

**按需加载**。

**压缩图片**。

#### 预先解析DNS

非常简单，效果立竿见影，加快页面加载时间，多用于预解析CDN的地址的DNS

<!--在head标签中，越早越好--><link rel="dns-prefetch" href="//example.com">

#### Preload

浏览器会在遇到如下link标签时，立刻开始下载main.js(不阻塞parser)，并放在内存中，但不会执行其中的JS语句。  
只有当遇到script标签加载的也是main.js的时候，浏览器才会直接将预先加载的JS执行掉。

<link rel="preload" href="/main.js" **as**="script">

#### Prefetch

浏览器会在空闲的时候，下载main.js, 并缓存到disk。当有页面使用的时候，直接从disk缓存中读取。其实就是把决定是否和什么时间加载这个资源的决定权交给浏览器。

如果prefetch还没下载完之前，浏览器发现script标签也引用了同样的资源，浏览器会再次发起请求，这样会严重影响性能的，加载了两次，，所以不要在当前页面马上就要用的资源上用prefetch，要用preload。

<link href="main.js" rel="prefetch">

3. css 三列布局的实现方式；

flex布局：flex:1，浮动: left、right、center，绝对定位：left:0, right:0，calc: calc(100% - left-right)

双飞翼：main先渲染，marin-left:-100% margin-rigit: width, main被包裹起来margin:with

圣杯：不需要包裹main，由外层的padding取代，其余一致，先渲染main

1. 代码质量如何保障；

变量名函数名语义化，代码逻辑清晰可读性强，关键代码注释，函数注释，格式规范用、eslint检测，库文件的changelog，commit的规范解决了什么问题、引入原因、解决方案、代码影响范围 ，提交的代码需要进行codereivew

1. html 模板的工作原理是什么样的；

编译时，利用正则找出模板内容，根据模板引擎处理模板内容，返回处理结果并替换原来的模板内容，最后生成处理后的html

1. 手写一个 flattern 函数；

递归展开

1. JS 中的作用域、原型、构造函数、this 怎么理解，new 操作符干了什么；

全局、函数、块级

词法(编写时确定)、动态（this）（调用时确定）

this代表函数执行时的环境，指向调用函数的对象

1. JS 函数上下文；

javaScript执行代码时创建的上下文，并压栈，创建作用域链，变量对象，this

9. 算法：怎么找到一个数组中的相同元素，空间复杂度和时间复杂度是怎么样的。

Olog(n)2分逼近法

**Web前端2**

1. 介绍一下 TypeScript

js的超集，具备静态类型和基于类的面向对象编程语法。

1. EventLoop Promise Async 题目

事件池，宏队列、微队列，所有微队列在下一个宏执行前需全部执行完





1. ES6继承与ES5继承的对比，优缺点







4. 如何实现私有变量

5. 算法题 二分查找

6. HTTPS HTTP2.0

7. 浏览器渲染流程

**WEB前端3**

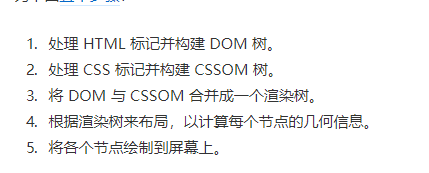
1. TCP/IP协议，HTTPS，HTTP2.0

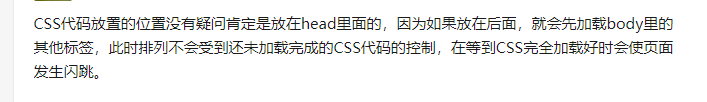
TCI/IP建立时三次握手，关闭四次握手

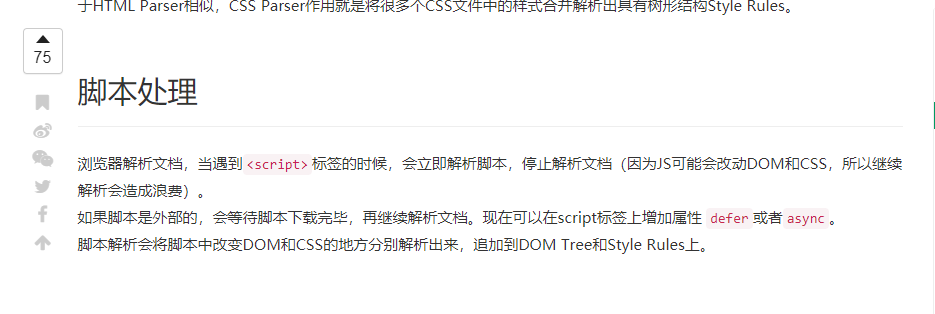
Content-length判断是否传输完毕，keep-alive：time\_out

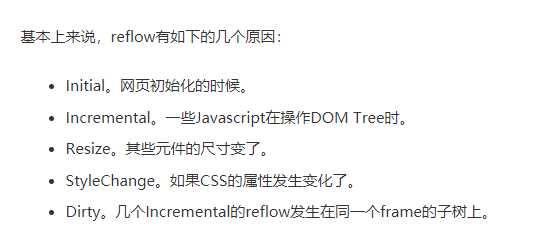
https， 服务器ca申请配置证书，客户端下载（公钥）验证通过后，使用公钥加密对称·密钥，发送使用对称密钥加密的信息，服务器利用私钥解密密钥利用密钥解密和加密信息

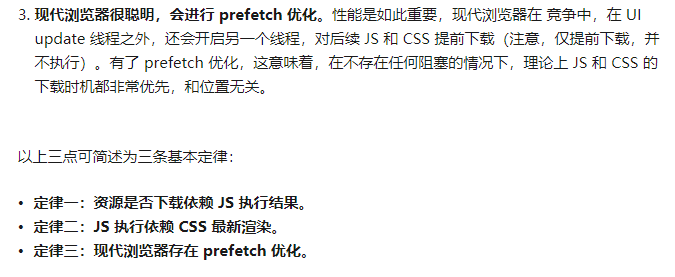
1. 为什么link标签要放在head里面，script标签会要放在body底部，为什么script标签的下载要阻塞ui渲染











link标签中链接的样式表会影响浏览器的绘制和布局，dom 解析到script后中断后面的解析

1. css解析规则，为什么要从右往左解析

效率更高，往往左边的选择符更泛化，右边更具化，从右往左减少大量查询次数

1. css选择器优化策略

（1）.ID选择器

（2）类选择器

（3）元素选择器

（4）兄弟选择器

（5）子选择器

（6）后代选择器

（7）属性选择器

（8）伪类/伪元素选择器

上面是常见的CSS选择器，性能从上到下越来越低。

1. .尽可能不使用通配符选择器：

关键选择器越具体越好，通配符选择器实在是最不具体的一个。

**（2）.合理避免使用id选择器：**

id选择器的性能最好，难道我们将每一个元素都添加一个id属性，肯定不现实。不过规范还是建议尽可能少的使用id选择器，这就是最佳实践与最佳性能之间的一个平衡或者取舍。

使用id确定元素在网页中的位置，应该始终考虑使用class，而不是id，除非只使用一次。

**（3）.减少后代选择器的使用：**

尽可能避免使用后代选择器，最好使用子选择器替代。

div>a的性能肯定要好于div a。

首先使用a选择器匹配页面所有链接<a>元素，后代选择器要一层层查找最终确定当前a是否具有div父元素，但是子元素选择器只要查找一层就可以了，性能可以优化很多。

**（4）.尽可能使用继承：**

#nav {}

#nav > .span { **font-size**:24px; }

#nav > .a { **font-size**:24px; }

上述代码可以优化如下：

#nav {**font-size**:24px;}

使用继承，而不是每一个选择器都设置一次。

1. 实现多列布局的方式
2. js基本数据类型，null和undefined的意义

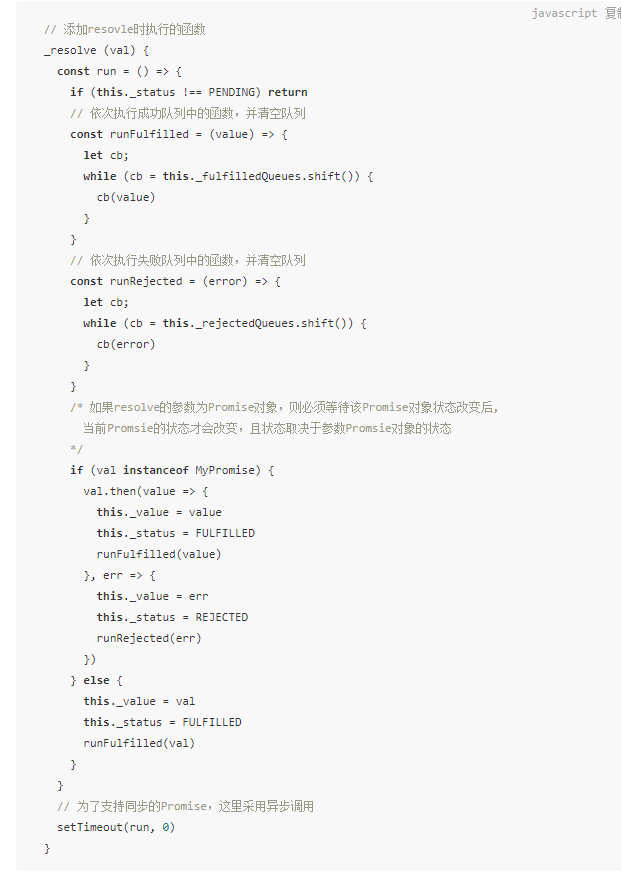
null表示变量赋为空值，undefined表示未赋值

1. js的异步有哪些实现方案，然后给了代码题考察promise、async|await

promise是一条热流，创建时就会执行内部代码，状态分为：pending、fullfilled、rejected，属性有reslove、reject、all、then、catch，

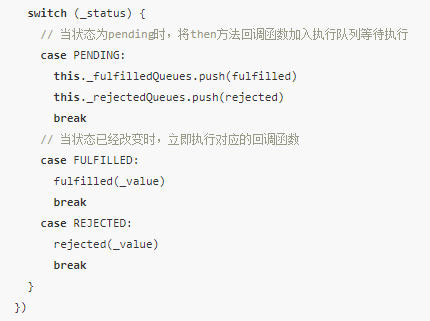
Then、all会返回一个新的promise并使其符合链式调用,then会传入onfullfiiled和onrejected，pending时进行注册回调函数队列，onfullfiiled或时onrejected直接执行，promise的参数有reslove和reject，resolve执行时将Promise对象的状态从 Pending(进行中) 变为 Fulfilled(已成功),并从注册的onfullfiiled队列中取出队尾函数执行，reject将Promise对象的状态从 Pending(进行中) 变为 Rejected(已失败)并从注册的onrejected队列中取出队尾函数执行，代码实现







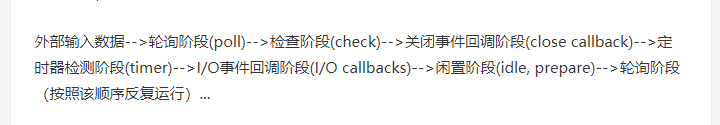




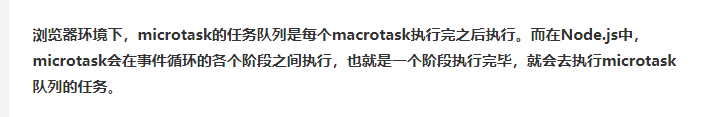
Aysnc\await yeild加promise语法糖

1. 事件循环的理解，浏览器跟node的事件循环差别

Node:

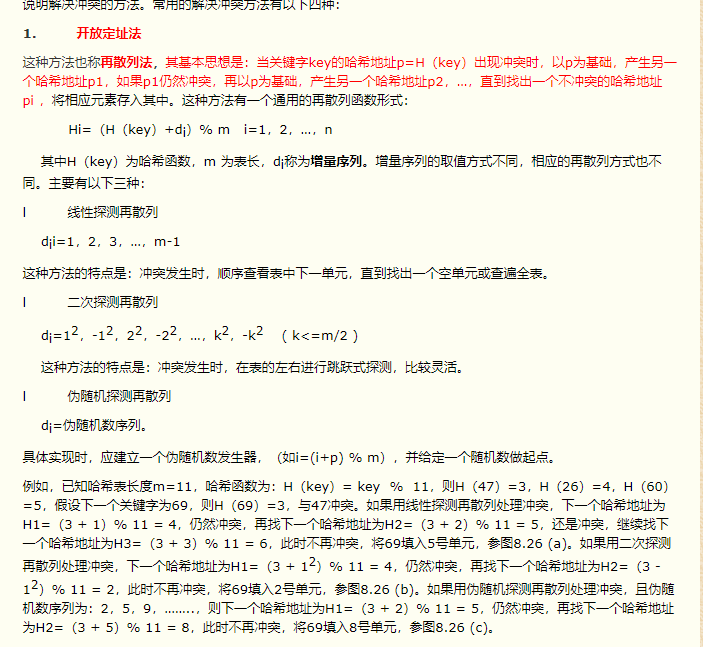


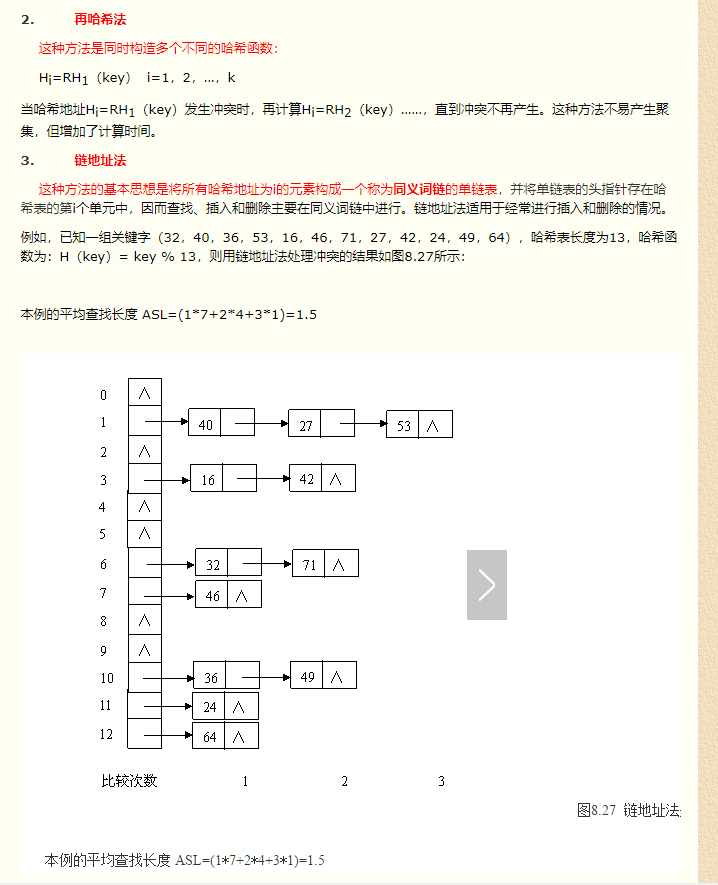




1. 数据结构与算法，哈希表怎么解决冲突

Hash：取余法，平方取中，**伪随机数法 解决冲突：开放定址：**线性探测再散列， 二次探测再散列了，随机数序列。





1. 常用的排序算法和时间复杂度，冒泡、归并、快排，为什么平时快排用得多一些，快排的优化方式
2. 函数执行上下文理解+代码题
3. 常用的设计模式
4. js面向对象的理解，es6、es5继承的方式
5. 写代码：二分查找
6. 写代码：正则匹配电话号码

**WEB前端4**

1. https如何保证数据不被篡改，密匙采用什么算法；MAC算法、SHA算法

如果证书（公钥、权威机构的信息、服务器域名和经过CA私钥签名之后的证书摘要,证书的计算方法以及证书对应的域名）被拦截并伪造，客户端在收到证书时会用ca公钥解密摘要,再根据证书上描述的计算证书的方法计算一下当前证书的信息摘要，对比两者是否一致，若一致则为服务器所发。

2.使用udp如何保证数据完整性，不丢包，如何控制传输速率；

3.vue中的key作用，vue-hooks， react-hooks；

key在v-for里面作为列的唯一标识减少数据变化重渲成本，同时列表节点变换算法也需要key提高性能，Vue-hooks，重用逻辑，不按生命周期周期

1. get，post之外还有哪些请求；强缓存如何实现；

put、delete、option，exprie:0，利用hash更新。

5.leetcode峰值问题；

6.async／await问题；

7.事件循环／setTimeout不精确的原因;

setTimeout自己内部有执行时间，被其他异步任务延后，间隔时间减去自己的执行时间

**WEB前端5**

1.代码题：zhongbiao给定24小时内任意两个时间，例如：开始时间00:00:00，结束时间：18:30:30，，时针，分针，秒针在钟表上每走一圈的角度是360度，计算两个时间段内时针，分针，秒针分别在钟表上走过的角度

2.代码题：字符串中含有左右括号， 写出函数判断字符串是否合法，合法的条件是左右括号都匹配，合法返回true否则返回false。例如： ：输入 ((a+b)\*(a+b))\*(a+b) 返回true，输入 ((a+b))))返回false。

3.回答以下输出内容，并说明原因：

(function(){

var a = b = 5;

})();

console.log(a);

console.log(b);

4.描述HTTP三次握手过程，HTTPS握手过程。

5.HTTP请求方式除了GET，POST以外还有哪些，分别是什么意思？。

6.具体说明display属性值与表现，positon:fixed在什么情况会失效以及为什么？

7.垂直居中实现方法有哪些？

8.vue双向绑定原理，简述具体源码实现

defineProperty(data, {

Get() {},

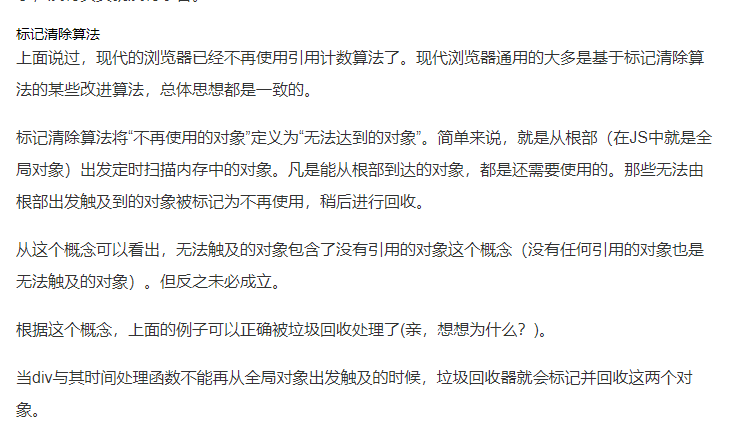
Set() {}

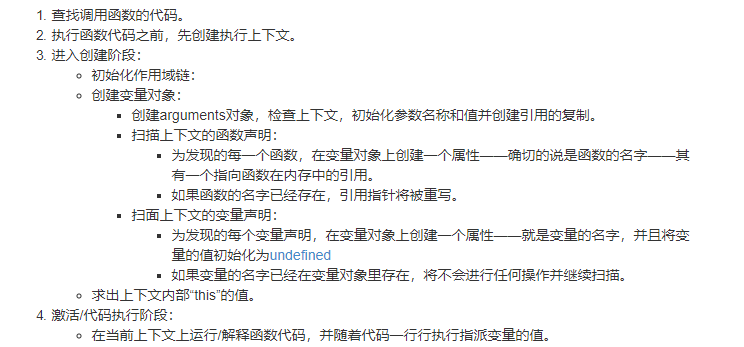
})

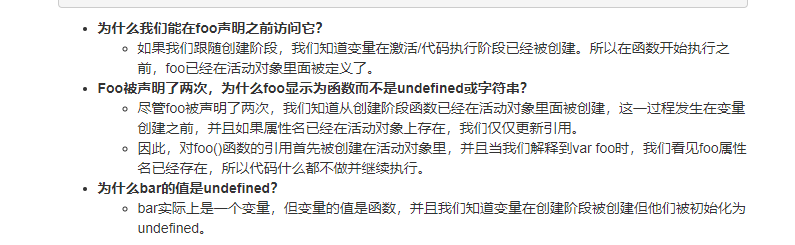
1. JS继承实现

js内存机制：

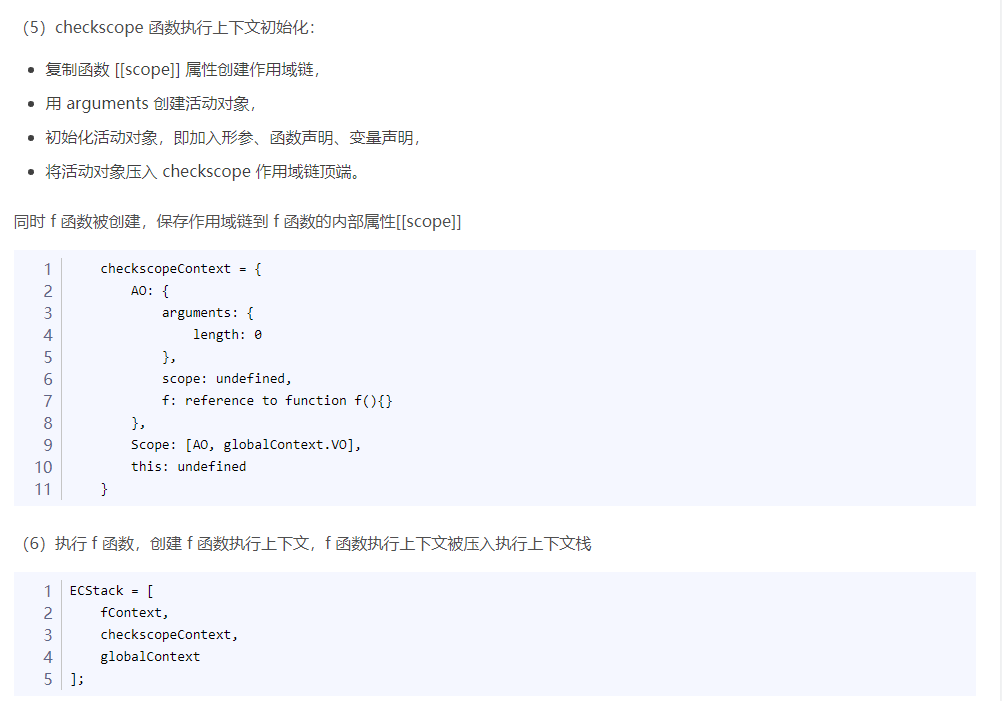






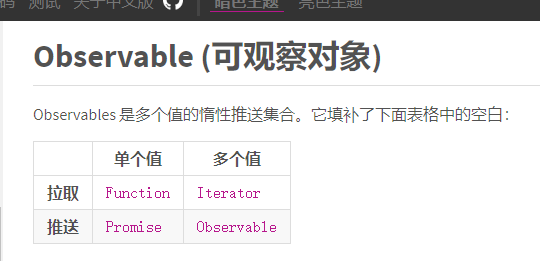






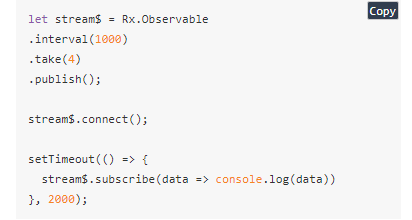
rxjs相关：





Observable 有冷热两种类型。我们先来看看什么是冷的 observable 。如果是冷的 observable 的话，那么两个订阅者得到值是两份完全相同的副本,如果是热的 observable 的话，订阅者只能收到当它开始订阅后的值，这很像是足球比赛的实况直播，如果你在开场5分钟后才开始观看，你会错失开场前5分钟的一切，从观看的这一刻起你才开始接收数据：

需要两个部件来将冷的 observable 转变成热的， publish() 和 connect() 。



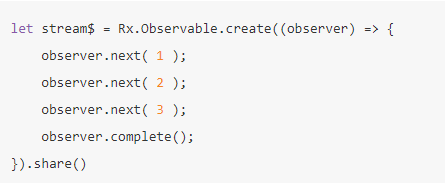
## **暖的 Observables**

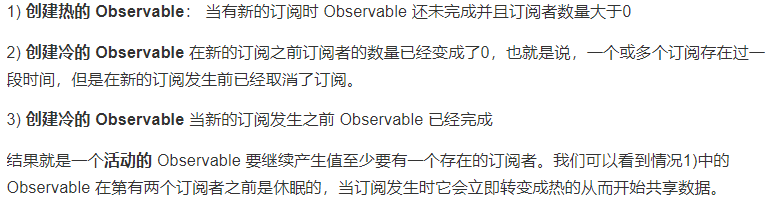
这是 observalbes 的另外一种类型，它的表现很像热的 observable ，但它在某种程度上是惰性的。我想表达的是从本质上来说，在有订阅发生之前它们不会发出任何值。



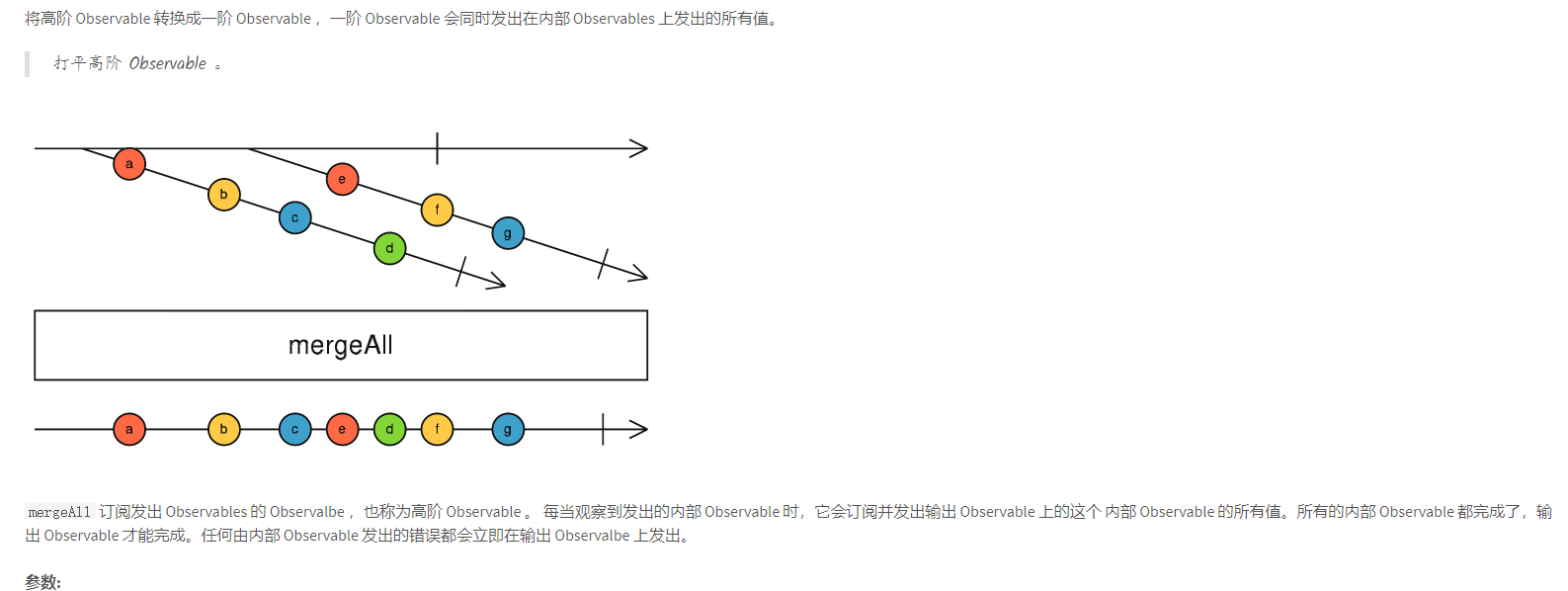
## **天生的热 observables**

使用 share() 操作符将冷的 observables 转变成热的结果。





mergeAll



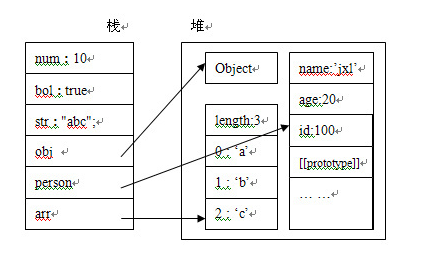
Cookie安全性：

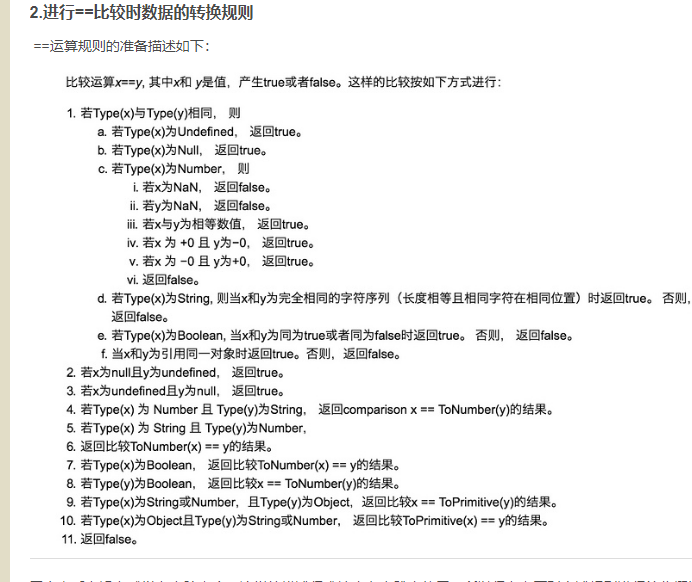


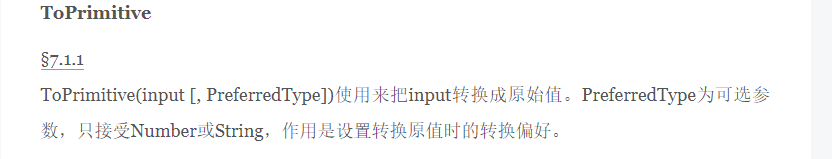
PeerDependencies

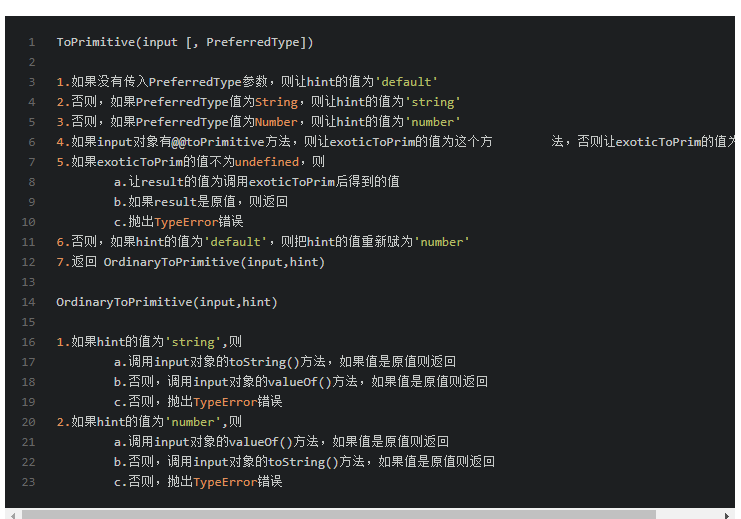


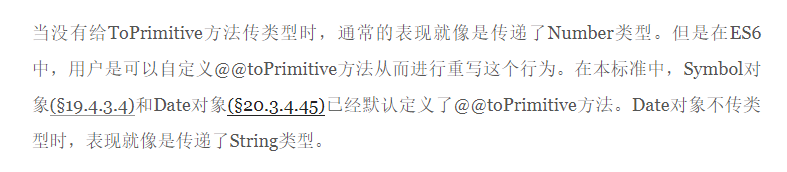
Js堆栈











304状态码是怎么样，怎么产生的？--》Etag值怎么产生的？

跨域怎么处理的呢？都知道什么方法？--> jsonp和CORS那个更安全？

rem和em有什么去区别啊？

position和float的区别？

let、const

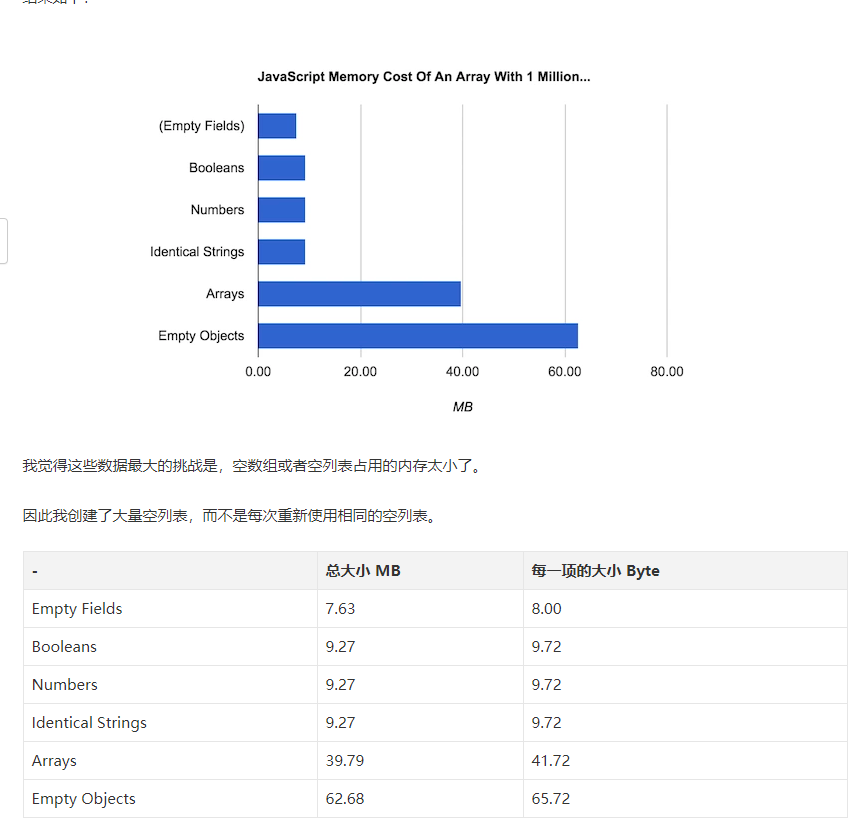
建立块级作用域，无变量提升，**let变量不能重复声明**  
let不允许在相同作用域内，重复声明同一个变量



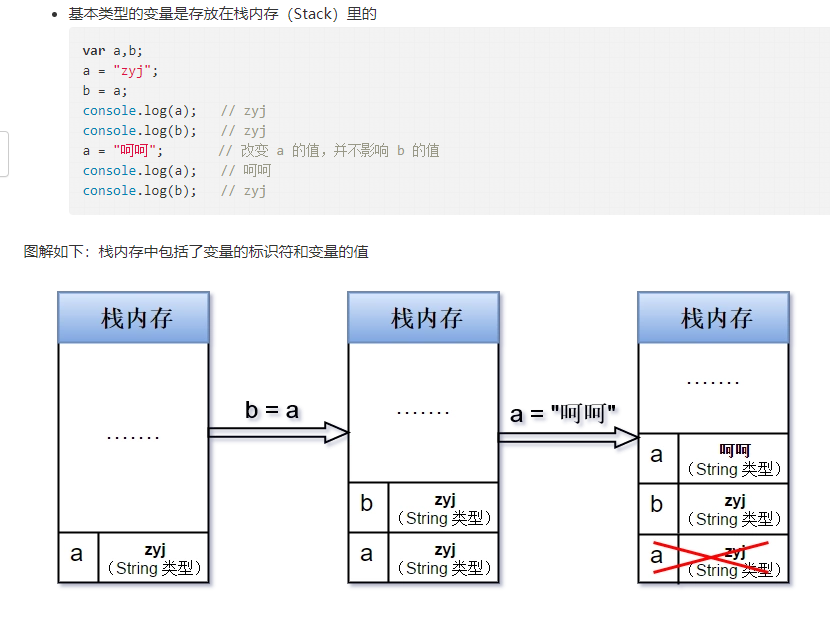
保证对象属性不可变



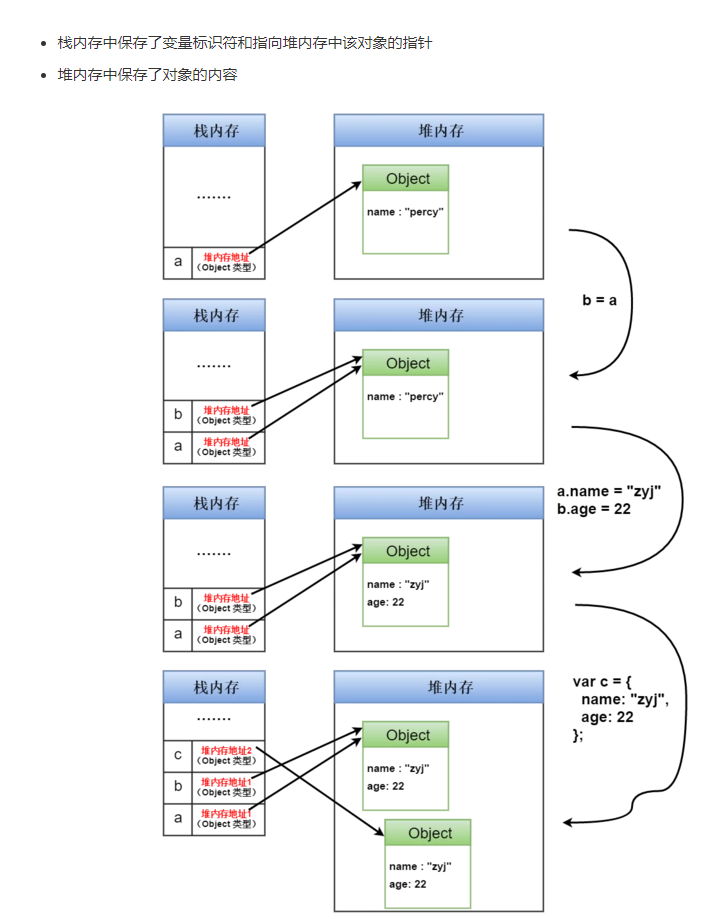
Js数据大小：



基本数据类型和引用数据类型：



引用类型变量是放在堆内存里的：



### **typedArray**

### **字节序**

字节序指的是数值在内存中的表示方式。

var buffer = new ArrayBuffer(16);var int32View = new Int32Array(buffer);

for (var i = 0; i < int32View.length; i++) {

int32View[i] = i \* 2;

}

上面代码生成一个16字节的ArrayBuffer对象，然后在它的基础上，建立了一个32位整数的视图。由于每个32位整数占据4个字节，所以一共可以写入4个整数，依次为0，2，4，6。

如果在这段数据上接着建立一个16位整数的视图，则可以读出完全不一样的结果。

var int16View = new Int16Array(buffer);

for (var i = 0; i < int16View.length; i++) {

console.log("Entry " + i + ": " + int16View[i]);

}// Entry 0: 0// Entry 1: 0// Entry 2: 2// Entry 3: 0// Entry 4: 4// Entry 5: 0// Entry 6: 6// Entry 7: 0

由于每个16位整数占据2个字节，所以整个ArrayBuffer对象现在分成8段。然后，由于x86体系的计算机都采用小端字节序（little endian），相对重要的字节排在后面的内存地址，相对不重要字节排在前面的内存地址，所以就得到了上面的结果。

比如，一个占据四个字节的16进制数0x12345678，决定其大小的最重要的字节是“12”，最不重要的是“78”。小端字节序将最不重要的字节排在前面，储存顺序就是78563412；大端字节序则完全相反，将最重要的字节排在前面，储存顺序就是12345678。目前，所有个人电脑几乎都是小端字节序，所以TypedArray数组内部也采用小端字节序读写数据，或者更准确的说，按照本机操作系统设定的字节序读写数据。

这并不意味大端字节序不重要，事实上，很多网络设备和特定的操作系统采用的是大端字节序。这就带来一个严重的问题：如果一段数据是大端字节序，TypedArray数组将无法正确解析，因为它只能处理小端字节序！为了解决这个问题，JavaScript引入DataView对象，可以设定字节序，下文会详细介绍。

下面是另一个例子。

// 假定某段buffer包含如下字节 [0x02, 0x01, 0x03, 0x07]var buffer = new ArrayBuffer(4);var v1 = new Uint8Array(buffer);

v1[0] = 2;

v1[1] = 1;

v1[2] = 3;

v1[3] = 7;

var uInt16View = new Uint16Array(buffer);

// 计算机采用小端字节序// 所以头两个字节等于258if (uInt16View[0] === 258) {

console.log('OK'); // "OK"

}

// 赋值运算

uInt16View[0] = 255; // 字节变为[0xFF, 0x00, 0x03, 0x07]

uInt16View[0] = 0xff05; // 字节变为[0x05, 0xFF, 0x03, 0x07]

uInt16View[1] = 0x0210; // 字节变为[0x05, 0xFF, 0x10, 0x02]

下面的函数可以用来判断，当前视图是小端字节序，还是大端字节序。

const BIG\_ENDIAN = Symbol('BIG\_ENDIAN');const LITTLE\_ENDIAN = Symbol('LITTLE\_ENDIAN');

function getPlatformEndianness() {

let arr32 = Uint32Array.of(0x12345678);

let arr8 = new Uint8Array(arr32.buffer);

switch ((arr8[0]\*0x1000000) + (arr8[1]\*0x10000) + (arr8[2]\*0x100) + (arr8[3])) {

case 0x12345678:

return BIG\_ENDIAN;

case 0x78563412:

return LITTLE\_ENDIAN;

default:

throw new Error('Unknown endianness');

}

}

总之，与普通数组相比，TypedArray数组的最大优点就是可以直接操作内存，不需要数据类型转换，所以速度快得多。

### **BYTES\_PER\_ELEMENT属性**

每一种视图的构造函数，都有一个BYTES\_PER\_ELEMENT属性，表示这种数据类型占据的字节数。

Int8Array.BYTES\_PER\_ELEMENT // 1Uint8Array.BYTES\_PER\_ELEMENT // 1Int16Array.BYTES\_PER\_ELEMENT // 2Uint16Array.BYTES\_PER\_ELEMENT // 2Int32Array.BYTES\_PER\_ELEMENT // 4Uint32Array.BYTES\_PER\_ELEMENT // 4Float32Array.BYTES\_PER\_ELEMENT // 4Float64Array.BYTES\_PER\_ELEMENT // 8

这个属性在TypedArray实例上也能获取，即有TypedArray.prototype.BYTES\_PER\_ELEMENT。

### **ArrayBuffer与字符串的互相转换**

ArrayBuffer转为字符串，或者字符串转为ArrayBuffer，有一个前提，即字符串的编码方法是确定的。假定字符串采用UTF-16编码（JavaScript的内部编码方式），可以自己编写转换函数。

// ArrayBuffer转为字符串，参数为ArrayBuffer对象function ab2str(buf) {

return String.fromCharCode.apply(null, new Uint16Array(buf));

}

// 字符串转为ArrayBuffer对象，参数为字符串function str2ab(str) {

var buf = new ArrayBuffer(str.length \* 2); // 每个字符占用2个字节

var bufView = new Uint16Array(buf);

for (var i = 0, strLen = str.length; i < strLen; i++) {

bufView[i] = str.charCodeAt(i);

}

return buf;

}

### **溢出**

不同的视图类型，所能容纳的数值范围是确定的。超出这个范围，就会出现溢出。比如，8位视图只能容纳一个8位的二进制值，如果放入一个9位的值，就会溢出。

TypedArray数组的溢出处理规则，简单来说，就是抛弃溢出的位，然后按照视图类型进行解释。

var uint8 = new Uint8Array(1);

uint8[0] = 256;

uint8[0] // 0

uint8[0] = -1;

uint8[0] // 255

上面代码中，uint8是一个8位视图，而256的二进制形式是一个9位的值100000000，这时就会发生溢出。根据规则，只会保留后8位，即00000000。uint8视图的解释规则是无符号的8位整数，所以00000000就是0。

负数在计算机内部采用“2的补码”表示，也就是说，将对应的正数值进行否运算，然后加1。比如，-1对应的正值是1，进行否运算以后，得到11111110，再加上1就是补码形式11111111。uint8按照无符号的8位整数解释11111111，返回结果就是255。

一个简单转换规则，可以这样表示。

* 正向溢出（overflow）：当输入值大于当前数据类型的最大值，结果等于当前数据类型的最小值加上余值，再减去1。
* 负向溢出（underflow）：当输入值小于当前数据类型的最小值，结果等于当前数据类型的最大值减去余值，再加上1。

请看下面的例子。

var int8 = new Int8Array(1);

int8[0] = 128;

int8[0] // -128

int8[0] = -129;

int8[0] // 127

上面例子中，int8是一个带符号的8位整数视图，它的最大值是127，最小值是-128。输入值为128时，相当于正向溢出1，根据“最小值加上余值，再减去1”的规则，就会返回-128；输入值为-129时，相当于负向溢出1，根据“最大值减去余值，再加上1”的规则，就会返回127。

Uint8ClampedArray视图的溢出规则，与上面的规则不同。它规定，凡是发生正向溢出，该值一律等于当前数据类型的最大值，即255；如果发生负向溢出，该值一律等于当前数据类型的最小值，即0。

var uint8c = new Uint8ClampedArray(1);

uint8c[0] = 256;

uint8c[0] // 255

uint8c[0] = -1;

uint8c[0] // 0

上面例子中，uint8C是一个Uint8ClampedArray视图，正向溢出时都返回255，负向溢出都返回0。

## **DataView视图**

如果一段数据包括多种类型（比如服务器传来的HTTP数据），这时除了建立ArrayBuffer对象的复合视图以外，还可以通过DataView视图进行操作。

DataView视图提供更多操作选项，而且支持设定字节序。本来，在设计目的上，ArrayBuffer对象的各种TypedArray视图，是用来向网卡、声卡之类的本机设备传送数据，所以使用本机的字节序就可以了；而DataView视图的设计目的，是用来处理网络设备传来的数据，所以大端字节序或小端字节序是可以自行设定的。

DataView视图本身也是构造函数，接受一个ArrayBuffer对象作为参数，生成视图。

DataView(ArrayBuffer buffer [, 字节起始位置 [, 长度]]);

下面是一个例子。

var buffer = new ArrayBuffer(24);var dv = new DataView(buffer);

DataView实例有以下属性，含义与TypedArray实例的同名方法相同。

* DataView.prototype.buffer：返回对应的ArrayBuffer对象
* DataView.prototype.byteLength：返回占据的内存字节长度
* DataView.prototype.byteOffset：返回当前视图从对应的ArrayBuffer对象的哪个字节开始

DataView实例提供8个方法读取内存。

* **getInt8**：读取1个字节，返回一个8位整数。
* **getUint8**：读取1个字节，返回一个无符号的8位整数。
* **getInt16**：读取2个字节，返回一个16位整数。
* **getUint16**：读取2个字节，返回一个无符号的16位整数。
* **getInt32**：读取4个字节，返回一个32位整数。
* **getUint32**：读取4个字节，返回一个无符号的32位整数。
* **getFloat32**：读取4个字节，返回一个32位浮点数。
* **getFloat64**：读取8个字节，返回一个64位浮点数。

这一系列get方法的参数都是一个字节序号（不能是负数，否则会报错），表示从哪个字节开始读取。

var buffer = new ArrayBuffer(24);var dv = new DataView(buffer);

// 从第1个字节读取一个8位无符号整数var v1 = dv.getUint8(0);

// 从第2个字节读取一个16位无符号整数var v2 = dv.getUint16(1);

// 从第4个字节读取一个16位无符号整数var v3 = dv.getUint16(3);

上面代码读取了ArrayBuffer对象的前5个字节，其中有一个8位整数和两个十六位整数。

如果一次读取两个或两个以上字节，就必须明确数据的存储方式，到底是小端字节序还是大端字节序。默认情况下，DataView的get方法使用大端字节序解读数据，如果需要使用小端字节序解读，必须在get方法的第二个参数指定true。

// 小端字节序var v1 = dv.getUint16(1, true);

// 大端字节序var v2 = dv.getUint16(3, false);

// 大端字节序var v3 = dv.getUint16(3);

DataView视图提供8个方法写入内存。

* **setInt8**：写入1个字节的8位整数。
* **setUint8**：写入1个字节的8位无符号整数。
* **setInt16**：写入2个字节的16位整数。
* **setUint16**：写入2个字节的16位无符号整数。
* **setInt32**：写入4个字节的32位整数。
* **setUint32**：写入4个字节的32位无符号整数。
* **setFloat32**：写入4个字节的32位浮点数。
* **setFloat64**：写入8个字节的64位浮点数。

这一系列set方法，接受两个参数，第一个参数是字节序号，表示从哪个字节开始写入，第二个参数为写入的数据。对于那些写入两个或两个以上字节的方法，需要指定第三个参数，false或者undefined表示使用大端字节序写入，true表示使用小端字节序写入。

// 在第1个字节，以大端字节序写入值为25的32位整数

dv.setInt32(0, 25, false);

// 在第5个字节，以大端字节序写入值为25的32位整数

dv.setInt32(4, 25);

// 在第9个字节，以小端字节序写入值为2.5的32位浮点数

dv.setFloat32(8, 2.5, true);

如果不确定正在使用的计算机的字节序，可以采用下面的判断方式。

var littleEndian = (function() {

var buffer = new ArrayBuffer(2);

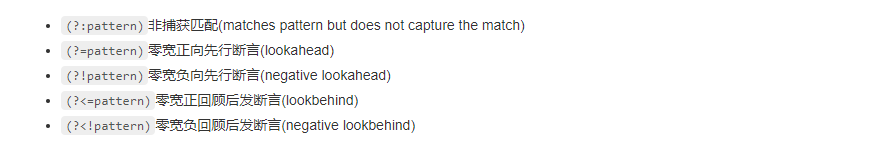
new DataView(buffer).setInt16(0, 256, true);

return new Int16Array(buffer)[0] === 256;

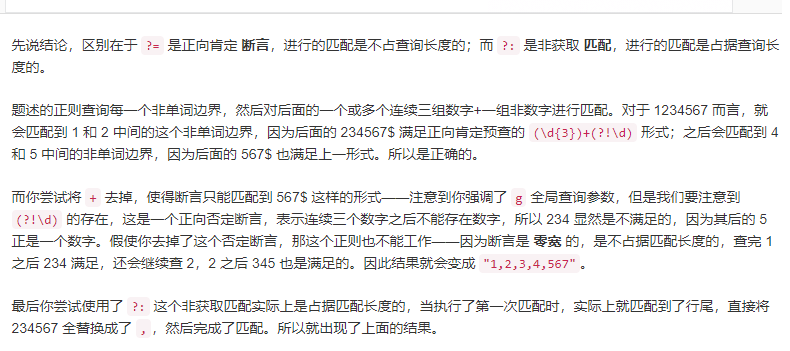
})();

如果返回true，就是小端字节序；如果返回false，就是大端字节序。

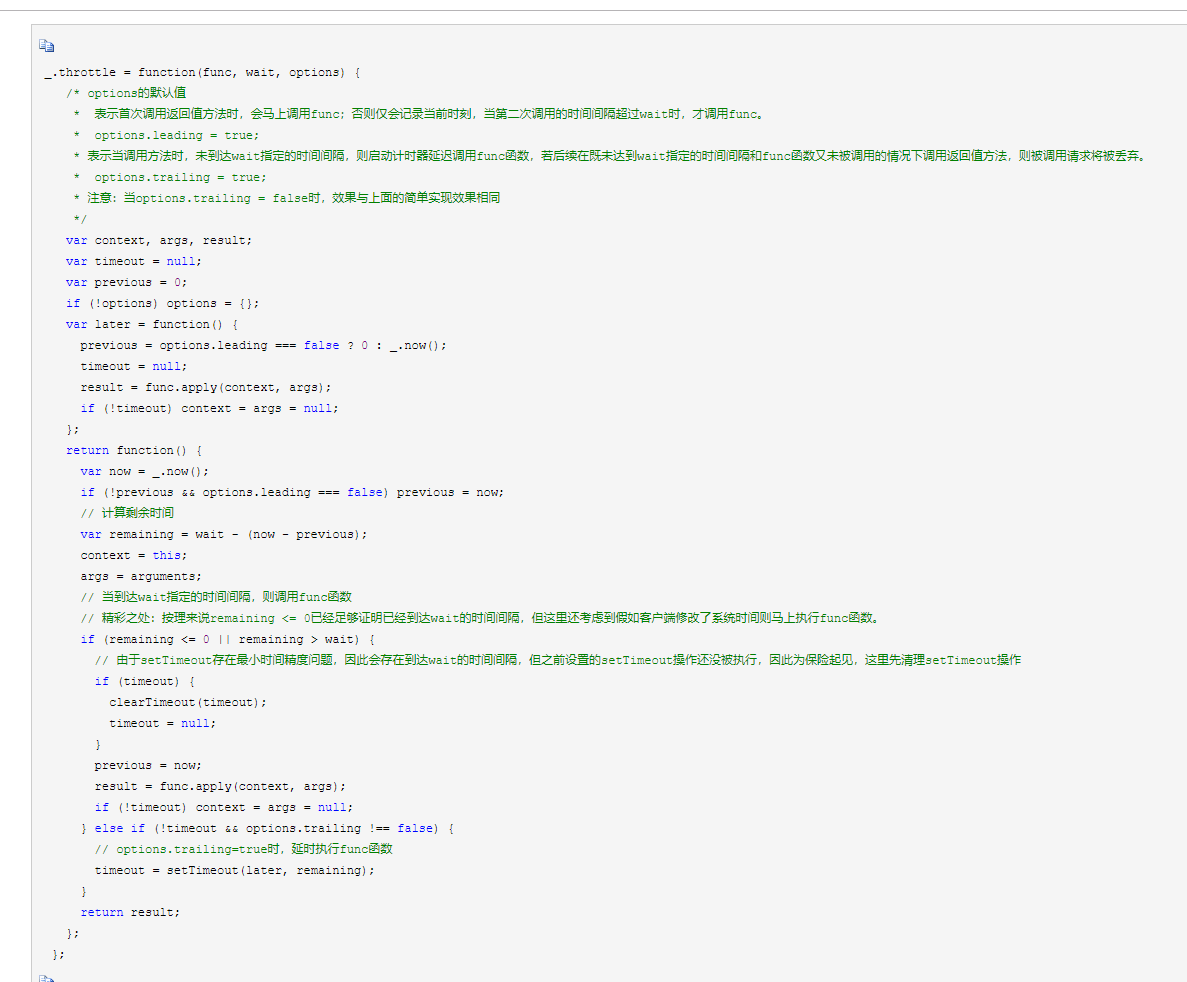
零宽断言



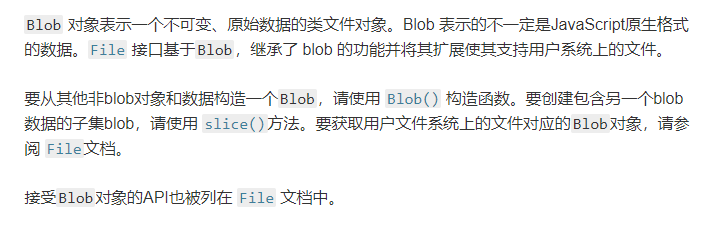




throttle和debounce源码解析：





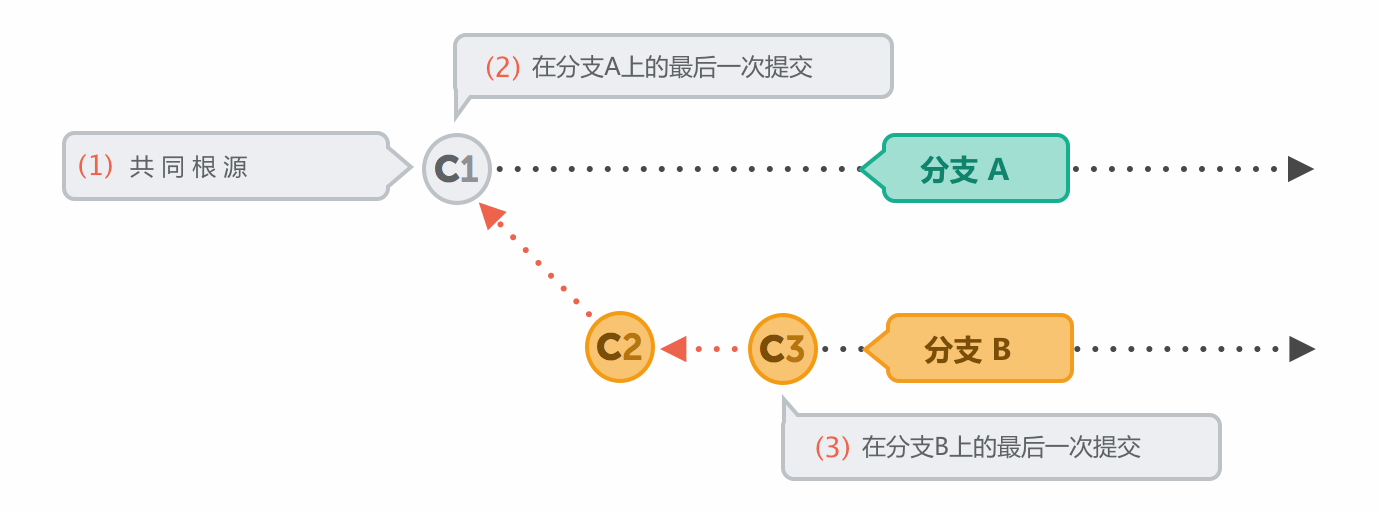


arrayBuffer转Blob

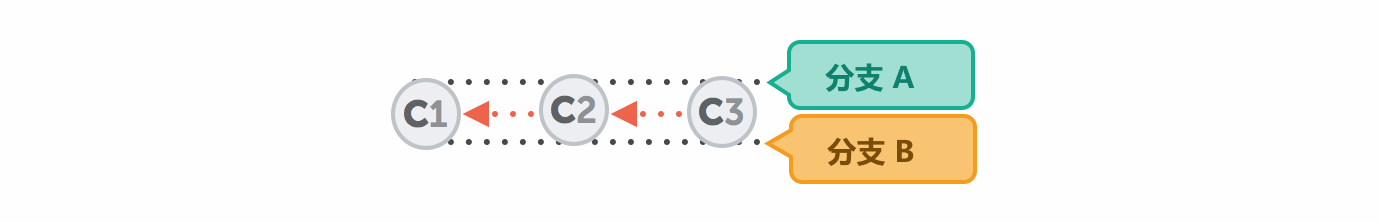


## 快进或合并提交

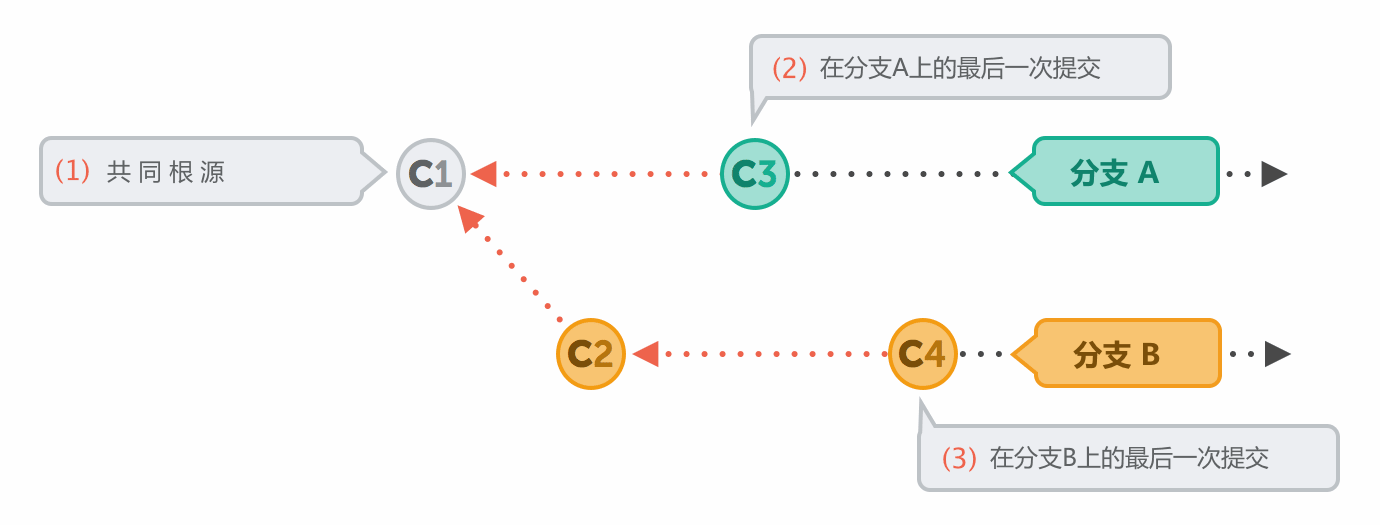
一种最简单的情况是，在其中的一个分支上没有任何一个新的改动提交发生。那么在它之前的最后一次提交就仍然还是那个共同的原始提交。



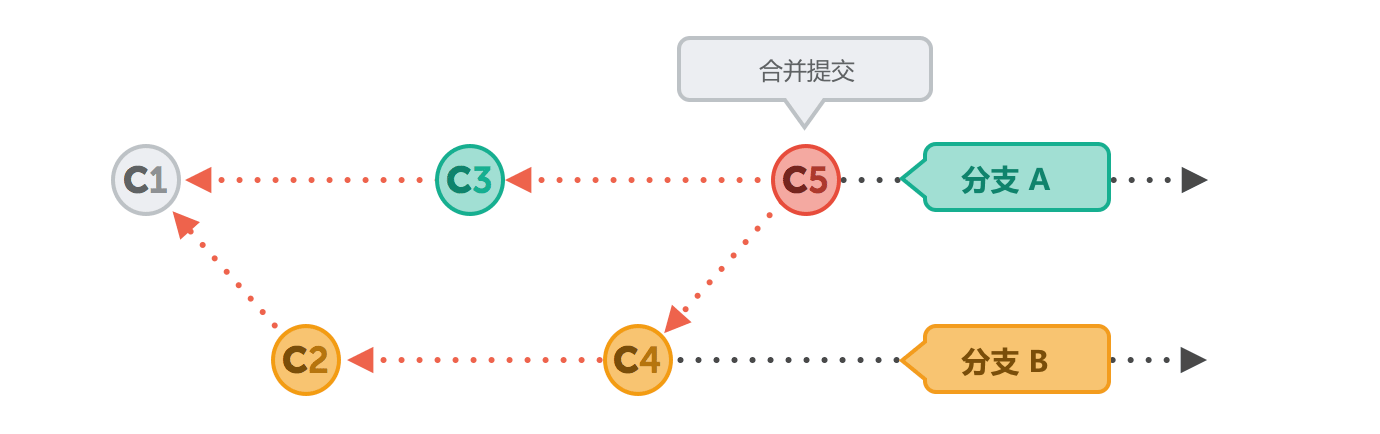
在这种情况下，执行整合操作就非常简单了。 Git 仅仅需要添加所有那些在另外一个分支上的新提交就可以了。在 Git 中，这种最简单的整合操作我们称之为 “快进（fast-forward）”合并。之后两个分支就拥有了完全相同的历史。



但是在大多数情况下，两个分支都会有自己不同的发展轨迹。



为了完成整合，Git 会需要创建一个新的提交来含括它们之间的差异，这就是整合提交（merge commit）。



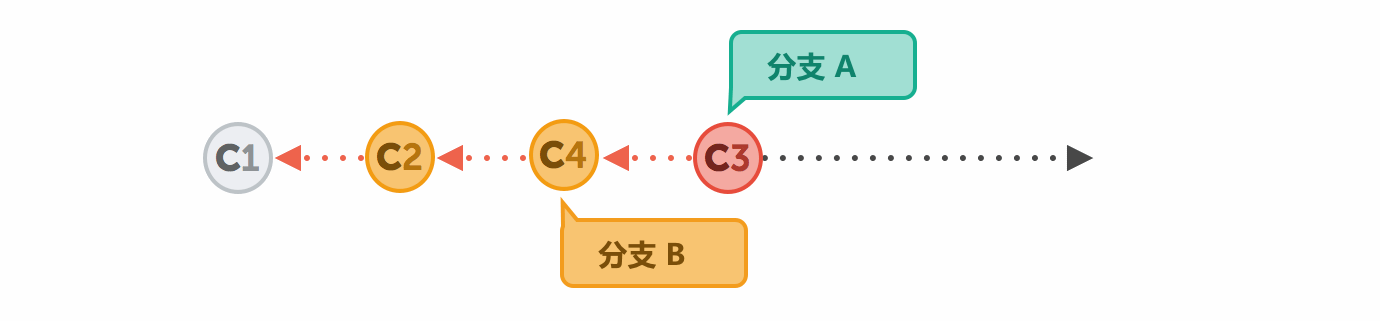
## 手工提交与合并提交

通常情况下，提交都是由手工精心创建的。这样也就能更好地保证一次提交只涉及一个关联改动，并且能更好地注释这个提交。

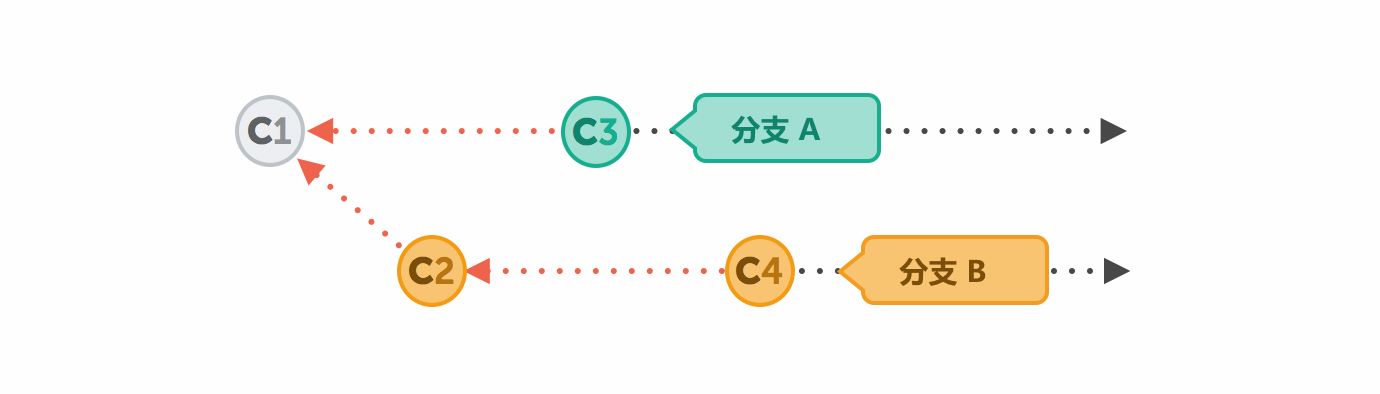
一个合并提交就不同了，它不是由开发人员手动创建的，而是由 Git 自动生成的。它也不涉及一个关联改动，其目的只是连接两个分支，就像节点一样。如果之后想要了解某个合并操作，你只需要查看这两个分支的历史记录和它们相应的提交树（version tree）。

## Rebase 整合

有些人并不喜欢使用这种自动合并提交。相反，他们希望项目拥有一个单一的历史发展轨迹。比如一条直线。在历史纪录上没有迹象表明在某些时间它被分成过多个分支。



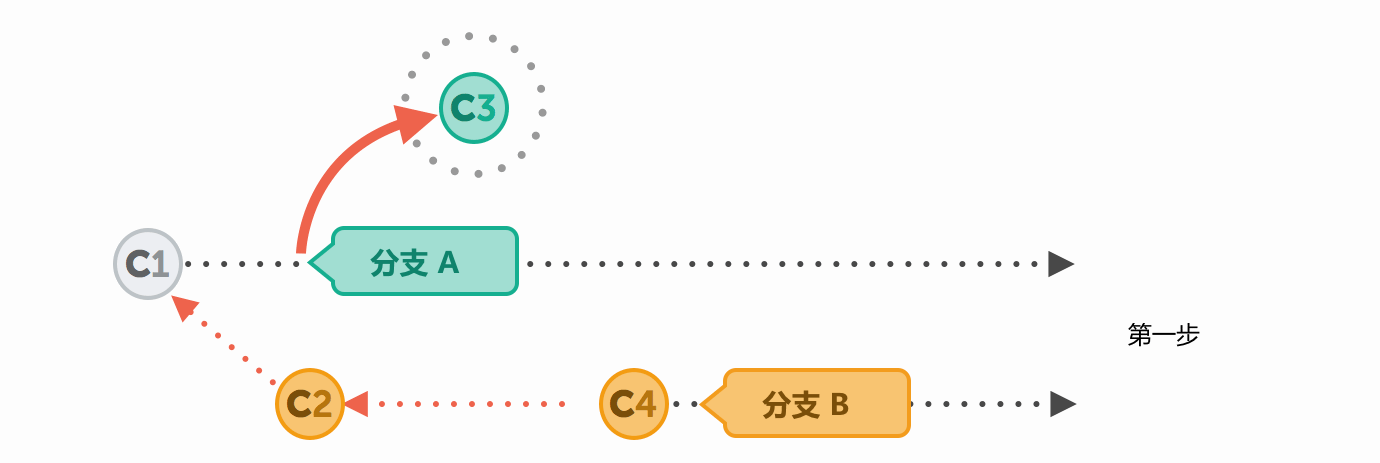
现在就让我们一步一步地了解一下 rebase 操作吧！仍然来使用前面的例子：我们想合并分支 B 到 分支 A 中，但是这次使用 rebase 操作。



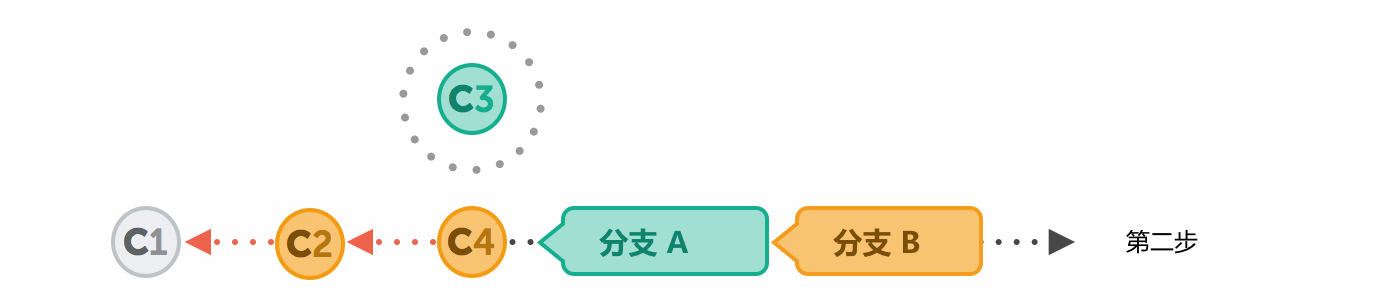
使用下面这个非常的简单的命令：

$ git rebase branch-B

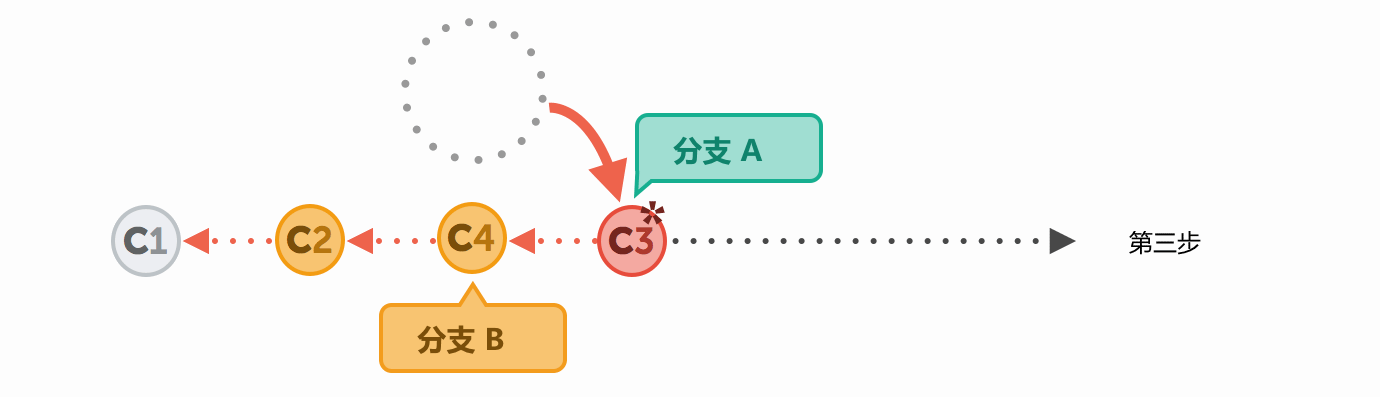
首先，Git 会 “撤销” 所有在分支 A 上的那些在与分支 B 的共同提交之后发生的提交。当然，Git 不会真的放弃这些提交，其实你可以把这些撤销的提交想像成 “被暂时地存储” 到另外的一个地方去了。

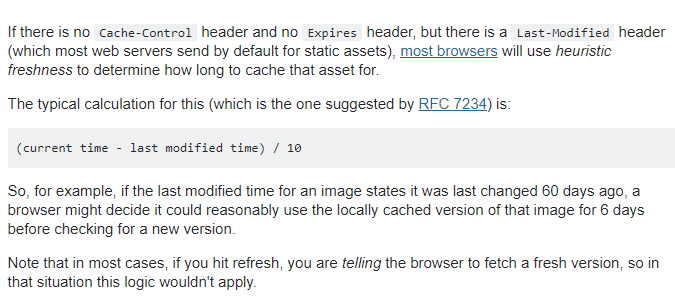


接下来它会整合那些在分支 B（这个我们想要整合的分支）上的还未整合的提交到分支 A 中。在这个时间点，这两个分支看起来会是一模一样的。



最后，那些在分支 A 的新的提交（也就是第一步中自动撤销掉的那些提交）会被重新应用到这个分支上，但是在不同的位置上，在那些从分支 B 被整合过来的提交之后，它们就被 re-based 了。  
整个项目开发轨迹看起来就像发生在一条直线上。相对于一个合并提交，rebase 包括了所有的组合变化，最原始的提交结构会被保留下来。





I'm getting the same problem on Chrome 72 Linux with the latest Ammo build. After 36 refreshes (probably arbitrary, but in my project it's always the same number) I get the error:

Uncaught RangeError: WebAssembly.Memory(): could not allocate memory

at ammo.wasm.js:formatted:175

at main ((index):82)

at onload ((index):17)

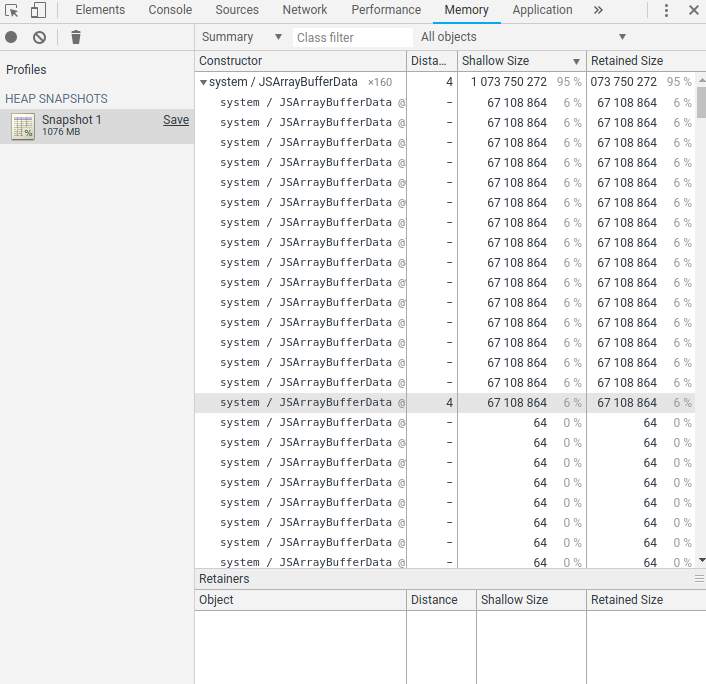
Which is just the memory module constructor:

new WebAssembly.Memory({

initial: Aa / 65536,

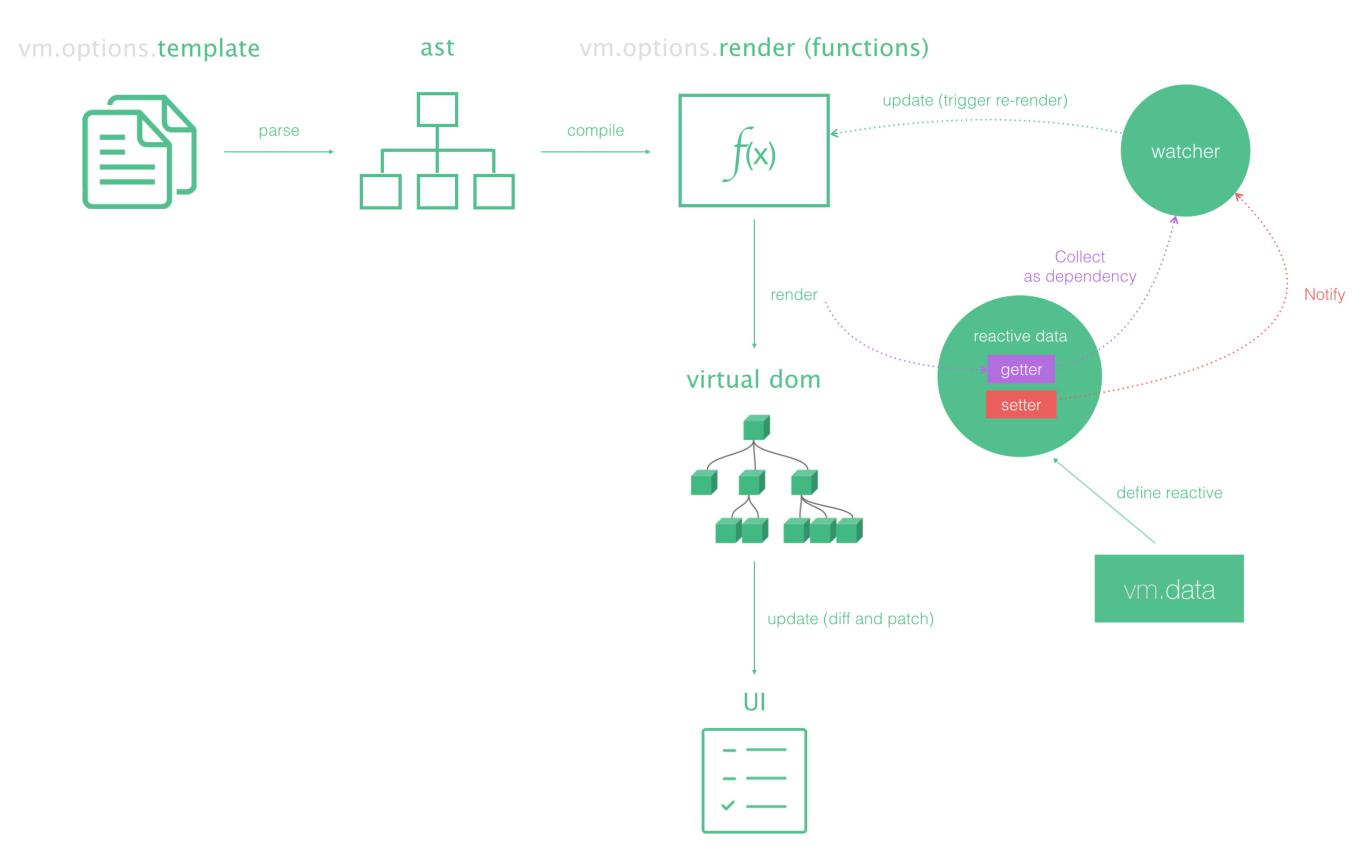
maximum: Aa / 65536

})

Looking into a heap snapshot shows the full story:  
[](https://camo.githubusercontent.com/37cc448f58498b152ccbab7f0c15734bdd6a493c/68747470733a2f2f692e696d6775722e636f6d2f45676e59586b492e706e67)

Chrome isn't fully resetting the heap! Clicking the garbage collection button doesn't help, either. Firefox doesn't have this issue so it's probably just a Chrome bug. It's also worth noting that I can't get the same effect when manually creating a memory object in the console.

Localstorage xss需注意对标签转义，csrf有效防止，cookie，xss通过httponly彻底遏制，csrf需注意用csrftoken防止

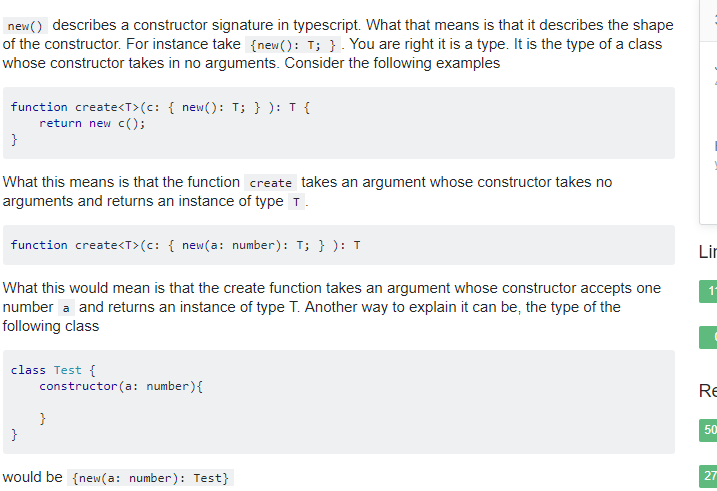


Typescript：

Interface: {

new(): A;

}



### ***Node***

这个解析策略试图在运行时模仿[Node.js](https://nodejs.org/)模块解析机制。 完整的Node.js解析算法可以在 [Node.js module documentation](https://nodejs.org/api/modules.html" \l "modules_all_together)找到。

#### ***Node.js如何解析模块***

为了理解TypeScript编译依照的解析步骤，先弄明白Node.js模块是非常重要的。 通常，在Node.js里导入是通过require函数调用进行的。 Node.js会根据 require的是相对路径还是非相对路径做出不同的行为。

相对路径很简单。 例如，假设有一个文件路径为 /root/src/moduleA.js，包含了一个导入var x = require("./moduleB"); Node.js以下面的顺序解析这个导入：

检查/root/src/moduleB.js文件是否存在。

检查/root/src/moduleB目录是否包含一个package.json文件，且package.json文件指定了一个"main"模块。 在我们的例子里，如果Node.js发现文件 /root/src/moduleB/package.json包含了{ "main": "lib/mainModule.js" }，那么Node.js会引用/root/src/moduleB/lib/mainModule.js。

检查/root/src/moduleB目录是否包含一个index.js文件。 这个文件会被隐式地当作那个文件夹下的"main"模块。

你可以阅读Node.js文档了解更多详细信息：[file modules](https://nodejs.org/api/modules.html" \l "modules_file_modules) 和 [folder modules](https://nodejs.org/api/modules.html" \l "modules_folders_as_modules)。

但是，[非相对模块名](https://www.tslang.cn/docs/handbook/module-resolution.html" \l "relative-vs-non-relative-module-imports)的解析是个完全不同的过程。 Node会在一个特殊的文件夹 node\_modules里查找你的模块。node\_modules可能与当前文件在同一级目录下，或者在上层目录里。 Node会向上级目录遍历，查找每个node\_modules直到它找到要加载的模块。

还是用上面例子，但假设/root/src/moduleA.js里使用的是非相对路径导入var x = require("moduleB");。 Node则会以下面的顺序去解析 moduleB，直到有一个匹配上。

1. /root/src/node\_modules/moduleB.js
2. /root/src/node\_modules/moduleB/package.json (如果指定了"main"属性)
3. /root/src/node\_modules/moduleB/index.js
4. /root/node\_modules/moduleB.js
5. /root/node\_modules/moduleB/package.json (如果指定了"main"属性)
6. /root/node\_modules/moduleB/index.js
7. /node\_modules/moduleB.js
8. /node\_modules/moduleB/package.json (如果指定了"main"属性)
9. /node\_modules/moduleB/index.js

#### ***TypeScript如何解析模块***

TypeScript是模仿Node.js运行时的解析策略来在编译阶段定位模块定义文件。 因此，TypeScript在Node解析逻辑基础上增加了TypeScript源文件的扩展名（ .ts，.tsx和.d.ts）。 同时，TypeScript在 package.json里使用字段"types"来表示类似"main"的意义 - 编译器会使用它来找到要使用的"main"定义文件。

比如，有一个导入语句import { b } from "./moduleB"在/root/src/moduleA.ts里，会以下面的流程来定位"./moduleB"：

1. /root/src/moduleB.ts
2. /root/src/moduleB.tsx
3. /root/src/moduleB.d.ts
4. /root/src/moduleB/package.json (如果指定了"types"属性)
5. /root/src/moduleB/index.ts
6. /root/src/moduleB/index.tsx
7. /root/src/moduleB/index.d.ts

回想一下Node.js先查找moduleB.js文件，然后是合适的package.json，再之后是index.js。

类似地，非相对的导入会遵循Node.js的解析逻辑，首先查找文件，然后是合适的文件夹。 因此/root/src/moduleA.ts文件里的import { b } from "moduleB"会以下面的查找顺序解析：

1. /root/src/node\_modules/moduleB.ts
2. /root/src/node\_modules/moduleB.tsx
3. /root/src/node\_modules/moduleB.d.ts
4. /root/src/node\_modules/moduleB/package.json (如果指定了"types"属性)
5. /root/src/node\_modules/moduleB/index.ts
6. /root/src/node\_modules/moduleB/index.tsx
7. /root/src/node\_modules/moduleB/index.d.ts
8. /root/node\_modules/moduleB.ts
9. /root/node\_modules/moduleB.tsx
10. /root/node\_modules/moduleB.d.ts
11. /root/node\_modules/moduleB/package.json (如果指定了"types"属性)
12. /root/node\_modules/moduleB/index.ts
13. /root/node\_modules/moduleB/index.tsx
14. /root/node\_modules/moduleB/index.d.ts
15. /node\_modules/moduleB.ts
16. /node\_modules/moduleB.tsx
17. /node\_modules/moduleB.d.ts
18. /node\_modules/moduleB/package.json (如果指定了"types"属性)
19. /node\_modules/moduleB/index.ts
20. /node\_modules/moduleB/index.tsx
21. /node\_modules/moduleB/index.d.ts

**const** worker = **new** Worker('worker.js')**const** length = 10;

*// Creating a shared buffer***const** sharedBuffer = **new** SharedArrayBuffer(Int32Array.BYTES\_PER\_ELEMENT \* length)

*// Creating a data structure on top of that shared memory area***const** sharedArray = **new** Int32Array(sharedBuffer)

*// Let's build an array with 10 even numbers***for** (**let** i = 0; i < length; i++) sharedArray[i] = i && sharedArray[i - 1] + 2

*// Send memory area to our worker*worker.postMessage(sharedBuffer)

setTimeout(**function**() {

console.log('[MASTER] Change triggered.')

sharedArray[0] = 1337}, 5000)

As the comments in the code above show, we:

1. Create a Worker
2. Create a shared memory area by using SharedArrayBuffer
3. Create a view (TypedArray) on top of that shared memory
4. Add 10 even numbers to our shared array
5. Send the shared memory (****not the Shared Array****) to our worker
6. Schedule a write in the 0 position of our shared memory area (represented through a TypedArray) after 5 seconds

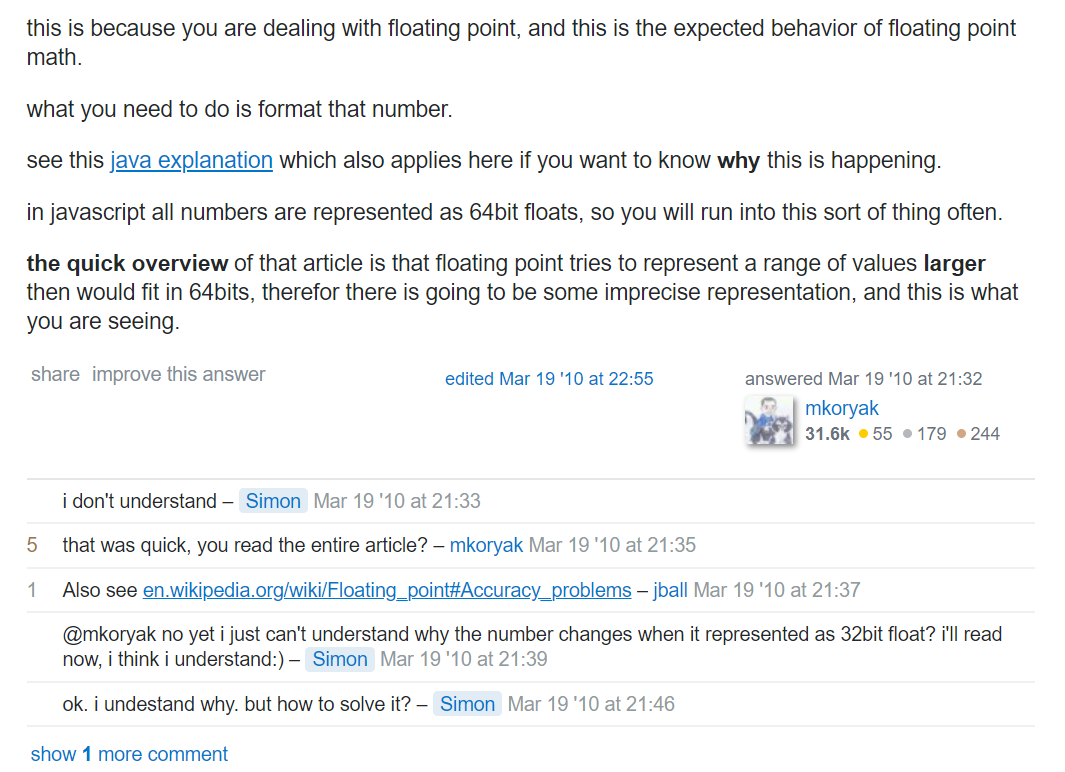
利用webworker加上sharedarraybuffer可实现webassembly的多线程，chorme70以上支持pthread

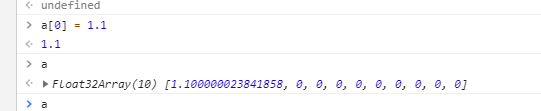
Browsers have supported parallelism via Web Workers since 2012 in Chrome 4; in fact it's normal to hear terms like 'on the main thread' etc. However, Web Workers do not share mutable data between them, instead relying on message-passing for communication. In fact, Chrome allocates a new V8 engine for each of them (called isolates). Isolates share neither compiled code nor JavaScript objects, and thus they cannot share mutable data like pthreads.

WebAssembly threads, on the other hand, are threads that can share the same Wasm memory. The underlying storage of the shared memory is accomplished with a [SharedArrayBuffer](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer), a JavaScript primitive that allows sharing a single ArrayBuffer's contents concurrently between workers. Each WebAssembly thread runs in a Web Worker, but their shared Wasm memory allows them to work much like they do on native platforms. This means that the applications that use Wasm threads are responsible for managing access to the shared memory as in any traditional threaded application. There are many existing code libraries written in C or C++ that use ***pthreads***, and those can be compiled to Wasm and run in true threaded mode, allowing more cores to work on the same data simultaneously.

零宽断言又分四种   
  
先行断言   也叫零宽度正预测先行断言(?=表达式)          表示匹配表达式前面的位置   
  
例如 [a-z]\*(?=ing) 可以匹配cooking singing 中的cook与sing   
  
注意：先行断言的执行步骤是这样的先从要匹配的字符串中的最右端找到第一个ing(也就是先行断言中的表达式)然后 再匹配其前面的表达式，若无法匹配则继续查找第二个ing 再匹配第二个 ing前面的字符串，若能匹配 则匹配  
  
例如：.\*(?=ing) 可以匹配cooking singing 中的cooking sing 而不是 cook   
  
后发断言        也叫零宽度正回顾后发断言        (?<=表达式)   表示匹配表达式后面的位置   
  
例如(?<=abc).\* 可以匹配abcdefg中的defg          
  
注意：后发断言跟先行断言恰恰相反 它的执行步骤是这样的：先从要匹配的字符串中的最左端找到第一个abc(也就是先行断言中的表达式)然后 再匹配其后面的表达式，若无法匹配则继续查找第二个abc 再匹配第二个abc后面的字符串，若能匹配 则匹配   
  
例如(?<=abc).\* 可以匹配abcdefgabc中的defgabc 而不是abcdefg   
  
负向零宽断言   
  
负向零宽断言 (?!表达式) 也是匹配一个零宽度的位置，不过这个位置的“断言”取表达式的反值，例如 (?!表达式) 表示 表达式 前面的位置，如果 表达式 不成立 ，匹配这个位置；如果 表达式 成立，则不匹配：同样，负向零宽断言也有“先行”和“后发”两种，负向零宽后发断言为 (?<!表达式)   
  
负向零宽后发断言(?<!表达式)   
  
负向零宽先行断言 (?!表达式)   
  
负向零宽断言要注意的跟正向的一样

FloatArray32不精确 无法正确取2位小数





## **什么是 Schema?**

当我们在描述 文字链接 的时候，需要约定数据的组织方式，比如，需要知道有哪些字段，这些字段的取值如何表示等，这就是 JSON Schema 的来源。

我们以 文字链接 为例，它对应的 JSON Schema 大概如此：

|  |  |
| --- | --- |
|  | {  "type": "object",  "properties": {  "text": {  "type": "string",  "title": "文字"  },  "href": {  "type": "string",  "title": "链接地址(URL)"  }  } } |

JSON Schema 定义了如何基于 JSON 格式描述 JSON 数据结构的规范，进而提供数据校验、文档生成和接口数据交互控制等一系列能力。它的特性和用途，可以大致归纳为以下几点：

### **1. 用于描述数据结构**

在描述 JSON 数据时，如果数据本身的复杂度很高，高到三维四维，普通的标签函数已经无法表示这种层级结构了，而 JSON Schema 利用 object 和 array 字段类型的反复嵌套，可以规避掉这个缺陷。

当然，除了键值等基本信息，规范层面还提供了丰富的关键词支持，如果想通过自定义扩展字段，解决特定场景的业务需求，也是非常方便的。

### **2. 用于构建人机可读的文档**

计算机领域有个概念叫做自描述。所谓自描述，可以理解为：文档本身包含了自身与其他文档交互相关的描述信息，不需要其他的配置文件或者额外信息来描述。

而 JSON Schema 就是自描述的，它本身就是一份很完善的说明文档，字段的含义说明、该如何取值、格式的要求等都清晰明了。

### **3. 用于生成模拟数据**

通过标签函数生成模拟数据，只能解决基本的格式要求。比如 string 类型的字段，模拟出来的数据，无非是一个随机字符串。

但在 JSON Schema 中，由于字段的描述不仅仅是类型，更多的约束条件，可以确保模拟数据更接近于真实数据。

### **4. 用于校验数据，实现自动化测试**

接口数据的校验工作，往往依赖于测试代码逻辑和用例。如果用 JSON Schema 描述一个数据接口，就不需要再编写测试代码了，所有的逻辑都可以移植到 JSON Schema 中维护。配合 jsv、tv4 等二方校验工具，接口测试可以真正自动化。

**TextEncoder** 将输入的代码流指向一个字节流并输出.

new TextEncoder().encode()，用于将string转为buffer

buffer再解码为string

const decoder = new TextDecoder('utf-8')

const view = new DataView(buffer, 0, buffer.byteLength)

const string = decoder.decode(view)

Webworker transferable objects

拷贝方式发送二进制数据，会造成性能问题。比如，主线程向 Worker 发送一个 500MB 文件，默认情况下浏览器会生成一个原文件的拷贝。为了解决这个问题，JavaScript 允许主线程把二进制数据直接转移给子线程，但是一旦转移，主线程就无法再使用这些二进制数据了，这是为了防止出现多个线程同时修改数据的麻烦局面。这种转移数据的方法，叫做[Transferable Objects](http://www.w3.org/html/wg/drafts/html/master/infrastructure.html" \l "transferable-objects" \t "http://www.ruanyifeng.com/blog/2018/07/_blank)。这使得主线程可以快速把数据交给 Worker，对于影像处理、声音处理、3D 运算等就非常方便了，不会产生性能负担。

如果要直接转移数据的控制权，就要使用下面的写法。

// Transferable Objects 格式worker.postMessage(arrayBuffer, [arrayBuffer]);

// 例子var ab = new ArrayBuffer(1);

worker.postMessage(ab, [ab]);

this.$bus.$on里面尽量少进行异步操作，特别是避免emit的组件需要等待on组件回调的情况。否则容易破坏代码结构。

如果一定要相办法让emit能够处理回调可以这样做将promise的resolve当作参数传emit出去，并在on处理完时执行reslove：

const bus = new Vue();

const Foo = {

template: `

<div>

<button @click="sendPromise()">Send a Promise</button>

<p>When Promise is resolved: {{ response }}</p>

</div>

`,

data () {

return {

response: ''

}

},

methods: {

sendPromise () {

let res = null;

let stuff = new Promise((resolve, reject) => {

res = resolve;

});

bus.$emit('promise', res);

stuff.then(() => {

this.response = 'Got res from bar'

});

}

}

};

const Bar = {

template: `<p>From foo: {{ msg }}</p>`,

data () {

return {

msg: ''

}

},

mounted () {

bus.$on('promise', (res) => {

this.msg = 'Received promise'

setTimeout(res, 1000);

})

}

};

const App = new Vue({

el: '#app',

components: {

Foo,

Bar

}

});



