1.js的数据类型了解

1.instanceOf检测实例

2.数据类型检测

基础类型? typeOf返回 : 使用toString+正则+replace,转换小写返回
引用类型使用toString返回的格式，统一为[object Array/Window

getType([]) //调用toString //[object Array]
参数：如果是window [object Window]
如果是function，typeof能直接返回
undefined //typeof
null //typeof返回object，toString返回的是[object Null],因此需要使用toString返回小写

用到的正则知识
\s匹配一个空格
\S+ 匹配非字符以外的,+匹配多次，case:123，可以匹配到123，没有+的话匹配一次
$1代表子验证表达式，代表匹配到的字符串，总共$1-$9

3.强制类型转换

Number
null,''为0
parseInt '' //NaN
undefined,{} //NaN
Bollean
undefined,null,'',false,0,NaN //false
其余为true【引用类型的值都为true】
String
toString
parseInt
parseFloat

4.隐性类型转换

- 一边是String，一边是Number，会将String转换为Number比较
- 类型相同，相同就比较大小
- 判断是否是null和undefined，是返回true，其余都是fasle
- 判断一边是否为Bollean，是则将Bollean转换为Number
- 判断一边是否为Object，如果Symbol.toPrimitive()方法存在则优先调用，否则调用Object的valueof，如果没有valueof，则调用Object.toString转换为字符串进行比较

概念：进行比较时每访问一次，如果本身没有symbol方法，就会调用一次本身的valueOf返回当前值

```
var a = {
    value:0,
    valueOf:function(){
        this.value++
        return this.value
    }
}
```

a==1 && a==2 &&a ==3 //true

case：
1.当为null或者undefined时，另一方必须也得是null或者undefined才为true
null == undefined //true
null == 0 //false

” + “的隐式类型的转换规则
1.如果其中有个是"字符串"，另一方为undefined,null,布尔,则先调用toString再使用” + “进行拼接
2.如果是纯对象，数组，正则,默认调用toString,如果存在symbol.toPrimitive,则首先调用后者
3.如果其中一个是字符串，一个是数字，会把数字转换为字符串

object的转换规则的顺序
symbol.toprimitive
valueOf 转换为基础类型的话返回,没有的话调用toString
toString:和valueOf一样
如果都没有返回，报错

5.基础类型和引用类型（堆内存，栈内存的理解）

基础类型

基础类型是存放在栈区的，假设有以下几个变量

```
var name = 'jozo';
var city = 'guangzhou';
var age = 22;
```

那么它的存储空间如下图：

栈区	
name	jozo
city	guangzhou
age	22

栈中保存了变量的标识符和变量的值

引用类型

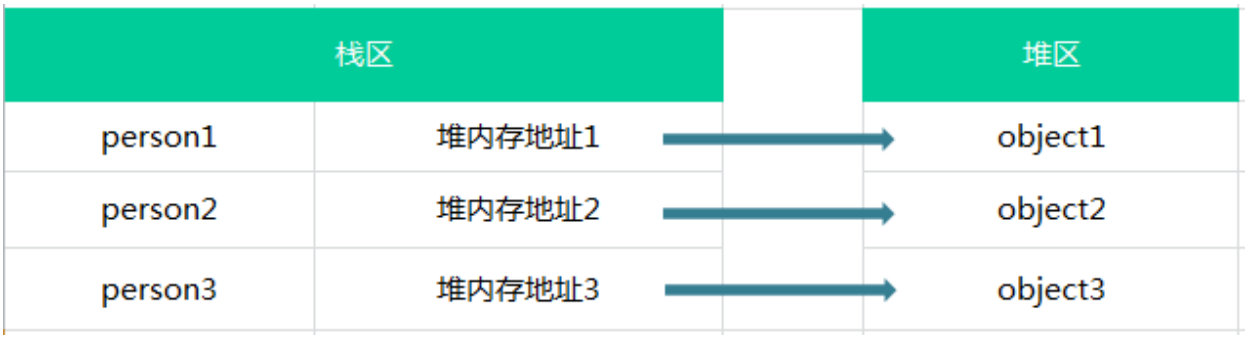
引用类型是同时保存在栈内存和堆内存中的对象

」

假如有以下几个对象：

```
var person1 = {name:'jozo'};
var person2 = {name:'xiaom'};
```

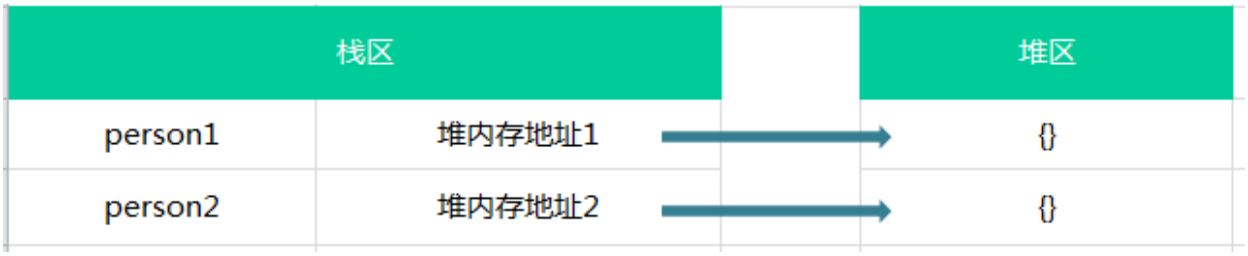
var person3 = {name:'xiaoq'};
这3个对象的内存保存情况如下图



引用类型的比较是引用的比较

```
var a = {}
var b = {}
console.log(a===b) //false
```

别忘了，引用类型是按引用访问的



2.同步异步的理解

2021/06/16 17:54

1. js的同步操作并不是所有的任务一起操作之意

2. js的异步操作并不是一个任务执行完另一个任务开始执行

同步：单线程，就是同一时间只做一件事情。同步操作都会放在主线程中进行

```
console.log(1);
setTimeout(function(){
    console.log(2);
},0)
console.log(3)
//1,3,2
```

第一句是个同步代码，在主线程中进行

读取到第二句，是个异步代码，放在浏览器中进行

第三句，参考第一句

所以，异步操作，首先将异步代码放在浏览器中，因为浏览器中是多线程的，返回顺序也就不一样。

简单来说，一个任务不是连续完成的，可以理解为该任务被分成了两段执行，先执行第一段，转而执行其他任务，等做好了准备，转而执行第二段。

比如文件读取，任务第一段是想操作系统发送请求，要求读取文件。然后，程序执行其他任务，等操作系统返回文件，再执行任务的第二段（处理文件）。这种不连续的操作，就叫异步。

回调函数

JavaScript对异步编程的实现，就是回调函数，就是把任务的第二段单独写在一个函数里面，等重新执行这个任务的时候，就调用这个函数。

```
fs.readFile('/etc/passwd', 'utf-8', function (err, data) {
  if (err) throw err;
  console.log(data);
});
```

一个有趣的问题是，回调函数的第一个参数为什么是错误对象。

原因是执行分成两段，第一段执行完之后，原来的上下文环境已经被释放。在这以后抛出的错误，原来的上下文环境已经无法捕捉。

测验

```
console.log(-1);
setTimeout(function(){
      console.log(0);
    },0);


setTimeout(function(){
    console.log(1);
},1000);


setTimeout(function(){
```

```
console.log(2);
},100);


setTimeout(function(){
    console.log(3);
},0);


setTimeout(function(){
    console.log(4);
},1000);


setTimeout(function(){
    console.log(5);
},1000);


setTimeout(function(){
  console.log(6);
},200)
// -1, 0, 3, 2, 6
```

3.ajax理解

1.创建xmlhttp对象，即创建一个异步调用对象

2.设置xmlhttp的请求方式与请求数据的地址（可以是本地文件）

3.为http响应状态设置监听函数

4.发送send请求（只有使用send方法后，xmlHttpRequest对象的readyState才会发生改变，才会激发loader函数）

5.处理响应参数

6.局部刷新

```
 function asyncLoadUrl(url) {
    var xmlHttpRequest = new XMLHttpRequest();
    xmlHttpRequest.open("get", url)
    var loader = function() {
        if (xmlHttpRequest.readyState === 4) {
            if (xmlHttpRequest.status === 200 || xmlHttpRequest.status === 0) {
                document.write(xmlHttpRequest.responseText)
            }
```

```
        }
    }
    xmlHttpRequest.onreadystatechange = loader
        // 经过以上的设置，就可以将Http请求发送到服务器上了
    xmlHttpRequest.send()
        // 只有使用send方法后，xmlHttpRequest对象的readyState才会发生改变，才会激发loader函数
}
asyncLoadUrl("./h2.html")
```

4.Promise对象

1.含义

异步编程的解决方案，可以获取异步操作的信息

Promise有两个特点

1）只有异步操作可以决定promise的3种状态，其他操作无法改变

2）Promise状态改变后就不会再变了，从pending-fulfilled和pending-rejected。只要状态发生改变，会一直保持这个结果

3)一旦新建就会立即执行

有了Promise，就可以将异步操作以同步操作的流程表达出来，避免层层嵌套的回调函数

promise也有一些缺点，首先，如果不设置回调函数，Promise内部抛出的错误，不会反映到外部。其次，处于pending状态时，无法得知目前进展到哪个阶段（刚刚开始还是即将完成）

2.基本用法

```
const promise = new Promise(function(resolve, reject) {
    // some code
    if (异步执行操作成功) {
        resolve(value)
    } else {
        reject(error)
    }
})
```

resolve函数的作用是，将Promise对象的状态从“未完成”变为“成功”（即从pending 变为 resolved）

reject函数的作用是，将Promise对象的状态从“未完成”变为“失败”（即从pending 变为 rejected）

Promise生成之后，用then方法指定resolved状态和rejected状态的回调函数

```
promise.then(function(value){
  success
},function(err){
   fail
})
```

下面是一个Promise对象的简单例子。

```
function timeout1(ms) {
   return new Promise(function(resolve, reject) {
      setTimeout(resolve, ms, "done");
   })
}
timeout1(800).then(function(value) {
   console.log(value);
})
```

上面代码中，timeout方法返回一个Promise实例，表示一段时间以后才会发生的结果。过了指定的时间（ms参数）以后，Promise实例的状态变为resolved，就会触发then方法绑定的回调函数。

Promise新建后就会立即执行

```
let promise = new Promise(function(resolve, reject) {
   console.log('promise')
   resolve()
}

promise.then(function(){
   console.log('resolved')
})

console.log('hi')

//promise
//hi
//resolved
```

then方法指定的回调函数，将在当前脚本所有同步任务执行完成之后，才会执行

使用Promise实现图片异步加载示例

```
function loadImageAsync(url) {
    return new Promise(function(resolve, reject) {
        var image = new Image()
        image.onload = function() {
            resolve(image)
        }
        image.onerror = function() {
            reject(image)
        }
        image.src = url
    })
}
```

图片异步加载的操作，加载成功，调用resolve方法，否则，调用reject方法

使用Promise实现ajax操作的例子

```
const getJson = function(url) {
    return new Promise(function(resolve, reject) {
        const handler = function() {
            if (xmlHttpRequest.readyState !== 4) {
                return
            }
            if (xmlHttpRequest.status === 200) {
                resolve(this.responseText)
            } else {
                reject(new Error(this.statusText))
            }
        }
        const xmlHttpRequest = new XMLHttpRequest()
        xmlHttpRequest.open("get", url)
        xmlHttpRequest.onreadystatechange = handler
        xmlHttpRequest.send()
    })
}
getJson("./h1.html").then(function(res) {
    console.log(res);
}, function(err) {
    console.log('error:', err);
})
```

resolve返回的还可以是一个Promise对象，像下面这样

```
const p1 = new Promise(function (resolve, reject) {
  // ...
});


const p2 = new Promise(function (resolve, reject) {
  // ...
  resolve(p1);
})
```

一个异步操作中返回另一个异步操作

3.Promise.resolve()

将现有对象转换为promise对象，Promise.resolve方法就起到这个作用

```
const jsPromise = Promise.resolve($.ajax('/whatever.json'));
```

上面代码将 jQuery 生成的deferred对象，转为一个新的 Promise 对象。

Promise.resolve等价于下面的写法

```
Promise.resolve(1)
//等价于
new Promise(resolve=>resolve(1))
```

1. 如果参数为promise实例，则原封不动的返回promise对象
2. 参数是一个thenable对象

```
let thenable = {
   then:function(resolve,reject){
      resolve(1)
   }
}
```

Promise.resolve会将其转换为promise对象，然后执行thenable对象中的then方法

3. 参数不是具有then方法的对象，或者根本不是对象。则返回一个新的Promise方法，状态为resolved

```
const p = Promise.resolve("abc")
p.then((s)=>{

})
```

由于字符串不属于异步操作（判断方法是字符串对象没有then方法），返回promise实例的状态就是resolved，所以回调函数会立即执行

4. 不带任何参数：返回的是一个resolved状态的promise对象

5.内存泄露与优化

2021/06/17 17:54

1.场景

1）过多的缓存未释放

2）闭包太多未释放

3）定时器或回调太多未释放

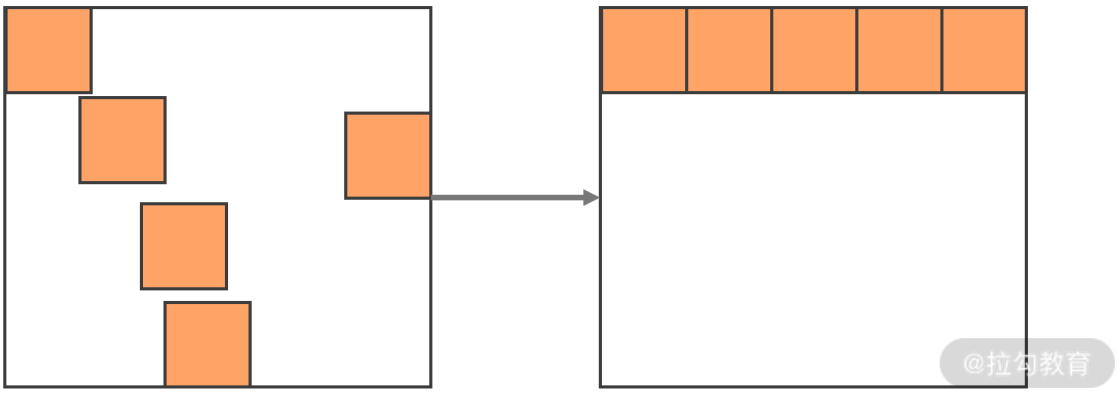
4）太多无效的dom未释放

5）全局变量太多未被发现

2.优化

标记清除：遍历堆中所有的变量，分别打上标记。代码执行过程结束后，对使用过的变量取消标记。那些未标记过的就是未使用过的变量

标记整理：由于标记清除后内存中会有很多内存碎片，标记整理会将所有的对象整理，以此来清除边界外的内存



以下是优化策略：

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

```
function foo() {
    this.bar = "a"
    bar = "b"
}
foo()
```

使用完数据后，即使清除引用

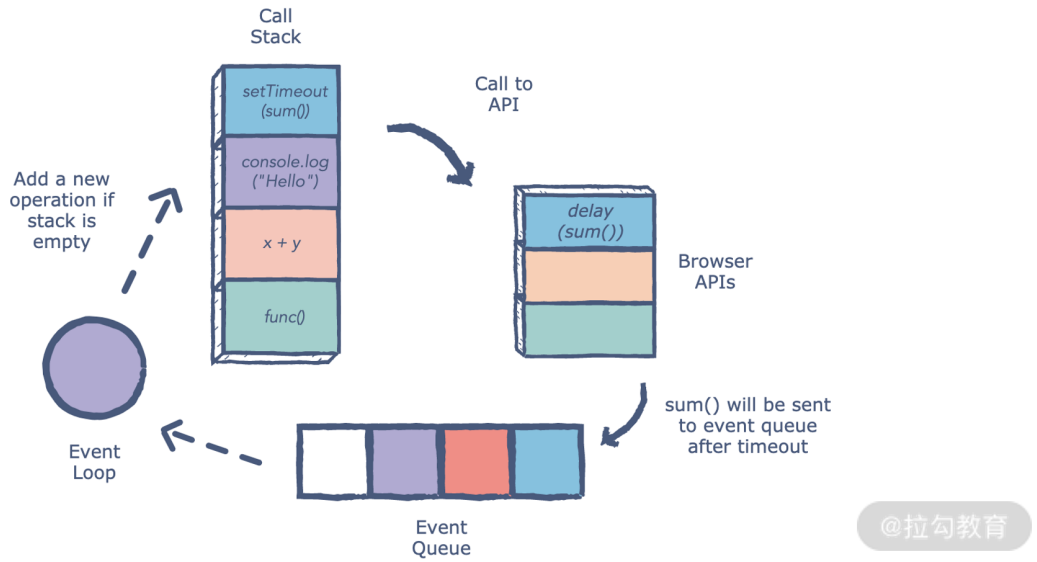
```
var a = getData()
setInterval(()=>{
   //...some code
})
//setInterval,a没有清除掉
```

6.浏览器与nodejs的EventLoop

2021/06/21 15:39

6.1.EventLoop-浏览器篇

1.图示



- 调用堆栈【Call Stack】：跟踪队列中的事件，每当事件队列调用一个函数后，将其pop出去。如果有代码需要进去执行的话，进行push操作
- 事件队列【Event Queue】：浏览器将新的function发送到队列中，先进先出
- 每当调用事件队列中的异步函数，会将其发送到相应的api层，等到指定时间后，会将此次操作送回事件队列中。
- EventLoop不断的检查调用堆栈，如果为空，就从事件队列添加新的function进去，否则就处理该function

宏任务与微任务

macrotasks(宏任务):
script(整体代码),setTimeout,setInterval,setImmediate,I/O,UI rendering,event listner

microtasks(微任务):
process.nextTick, Promises, Object.observe, MutationObserver

浏览器中执行宏任务与微任务：

- JavaScript引擎首先从宏任务队列中取出第一个任务
- 执行完成后，从微任务中取出所有的任务按照顺序执行
- 期间有新的任务添加进来也依次执行
- 然后再从宏任务中取出下一个任务，依次2，3步

一次EventLoop会执行一个宏任务和此次操作中所有的微任务

6.2 EventLoop-node.js篇

1.三大关键阶段

1. 执行定时器回调阶段，如果到时间了，就执行回调，这个阶段叫做timer
2. 轮询（poll）阶段，在node代码中难免会有异步操作，如文件I/O、网络I/O等，当这些异步操作完后，就会通知主线程，就是通过'data'、 'connect'等事件使得事件循环到达 poll 阶段。到达了这个阶段后: 如果当前存在定时器，会去看回调函数队列

    a. 队列不为空，依次执行

    b. 队列为空，检查是否有setImmediate的回调

    - 有，前往check阶段
    - 无，继续等待，相当于阻塞了一段时间（阻塞时间是有上限的），等待callBack函数加入队列，加入后立即执行，一段时间后加入check阶段

3. check阶段，执行setImmediate的回调

这3个阶段为一个循环阶段，不过现在的eventLoop还不够完善，下面进行完善

2.完善

当第一个阶段完成后，可能不会立即等待回调函数的响应，这时就进入I/O的异常的回调阶段，比如说 TCP 连接遇到ECONNREFUSED，就会在这个时候执行回调。

并且在check阶段还会进入关闭事件的回调阶段，如果一个 socket 或句柄（handle）被突然关闭，例如 socket.destroy()， ’close’ 事件的回调就会在这个阶段执行。

梳理一下，nodejs的EventLoop分为这几个阶段

1. timer阶段
2. I/O异常回调阶段
3. 空闲、预备状态(第2阶段结束，poll 未触发之前)
4. poll阶段
5. check阶段
6. 关闭事件的回调函数阶段

3.实例演示

```
setTimeout(()=>{
    console.log('timer1')
    Promise.resolve().then(function() {
        console.log('promise1')
    })
}, 0)
setTimeout(()=>{
    console.log('timer2')
    Promise.resolve().then(function() {
        console.log('promise2')
    })
}, 0)
```

执行顺序与浏览器中一样

4.浏览器与nodejs的EventLoop的不同之处

两者最重要的区别是：浏览器的微任务是在每个宏任务内的微任务执行的。而nodejs的微任务是在不同的阶段执行的

7.JS代码是如何被浏览器编译、执行的?

1.阶段

Parse阶段：V8引擎将代码转换为AST（抽象语法树）

Lginition阶段：解析器将AST转换为字节码， 也为下个阶段优化编译提供了信息

TurboFan阶段：编译器利用上个阶段获取的信息，将字节码转换为了机器码

Orinoco阶段：垃圾回收阶段，将不再使用的内存空间回收

1.AST:

AST的用处分为以下几个应用场景

- JS反编译，语法解析
- 代码高亮
- Babel编译ES6语法
- 关键字匹配
- 代码压缩

生成AST分为两个阶段，一是词法分析，二是语法分析

- 词法分析：将源代码拆分成最小，不可再分的词法单元，例如var a = 1，var、a、=、1。空格忽略
- 语法分析：将词法单元生成语法树

抽象语法树：

```
var a = 1;
//编译后的效果
{
  "type": "Program",
  "start": 0,
  "end": 10,
  "body": [
    {
      "type": "VariableDeclaration",
      "start": 0,
      "end": 10,
      "declarations": [
        {
          "type": "VariableDeclarator",
          "start": 4,
          "end": 9,
          "id": {
            "type": "Identifier",
            "start": 4,
            "end": 5,
            "name": "a"
          },
          "init": {
            "type": "Literal",
            "start": 8,
            "end": 9,
            "value": 1,
            "raw": "1"
          }
```

```
      }
    ],
    "kind": "var"
  }
 ],
 "sourceType": "module"
}
```

目前浏览器还不支持ES6语法，通过工具Babel将ES6语法转换为AST，再将ES6的抽象语法树转换为ES5的抽象语法树。另外ESLint的原理大致相同，将源代码转换为抽象语法树，再利用它检测代码规范

2.生成字节码

字节码并不能让机器直接运行，那如果把AST直接转换为机器码呢，V8早期是这么做的，后来因为体积大，引发了严重的内存问题

来看看三者代码量的差异:

字节码是比机器码轻的多的代码，字节码是介于AST和机器码之间的一种代码，但与特定类型的机器码无关，字节码需要通过编译器转换为机器码执行，但和原来不同的是，不需要一次性转换，只需要逐行逐句去执行字节码，这样就省去了生成二进制文件的操作。大大降低了内存的压力

3.编译器解释字节码

在编译器解释字节码的过程中，如果某段代码重复，V8会将它记为热点代码Hotspot，因此在这样的机制下，代码执行的时间越长，代码执行效率也越高。因为有越来越多的代码重复，执行时直接执行对应的机器码就可以，不用再将其转换为机器码。

8.宏任务与微任务的运行机制

1.思考问题

- 宏任务与微任务有哪些方法?
- 宏任务与微任务互相嵌套，执行顺序是怎么样的?

宏任务

在JS中，大部分的任务都是在主线程上进行，常见的任务有：

1. 渲染事件
2. 用户交互事件
3. js脚本执行
4. 网络请求，文件读写事件

为了让这些事件有条不紊的进行，JS引擎需要对执行的顺序做一定的安排，V8引擎采用的是队列的方式，即先进来的先执行

```
bool keep_running = true;
void MainTherad(){
  for(;;){
    //执行队列中的任务
    Task task = task_queue.takeTask();
    ProcessTask(task);
    //执行延迟队列中的任务
    ProcessDelayTask()
    if(!keep_running) //如果设置了退出标志，那么直接退出线程循环
        break;
  }
}
```

for循环将队列中的任务一一取出执行。除了任务队列，还有一个就是延迟队列，它专门处理诸如setTimeout/setInterval这样的定时器回调任务。

普通任务和延迟队列中任务都属于宏任务。

微任务引入

对于每个宏任务，其内部都有一个微任务队列，微任务的引入是为了处理异步回调。那么处理异步回调的方式有下面两种

- 将异步回调进行宏任务的队列操作
- 将异步回调放入当前宏任务的末尾

如果采用第一种方式，那么执行回调的时机应该是在前面的宏任务执行完成后才执行，假如前面的宏任务非常多，那么回调迟迟得不到执行，就会造成应用卡顿

为了规避这样的问题，V8就引入了第二种方式，在每个宏任务内部定义了一个微任务队列，则当该宏任务执行完成，检查其中的微任务队列，为空，执行下个宏任务。不为空，依次执行微任务。执行完成后再去执行下一个宏任务

代码执行顺序

```
console.log('begin');
setTimeout(() => {
  console.log('setTimeout')
}, 0);
new Promise((resolve) => {
  console.log('promise');
  resolve()
}).then(() => {
    console.log('then1');
  }).then(() => {
    console.log('then2');
  });
console.log('end');
//顺序为:
begin
promise
end
then1
then2
setTimeout
```

以上涉及到的是JS中的宏任务和微任务的嵌套情况，在EventLoop中，每次循环称为一次tick，主要顺序为：

- 最先进行的是调用栈的宏任务【将宏任务最先push到了事件队列中】，执行settimeout，顺序为先进后出。直接执行其同步代码，宏任务进入宏任务队列，微任务进入微任务队列，宏任务执行出队
- 检测是否还有其余的微任务，如果有则执行直到微任务列表为空
- 浏览器目前已经开始渲染页面了（执行浏览器 UI 线程的渲染工作）
- 开始下一轮tick前，此时执行宏任务中的代码，例如以上的settimeout的回调函数。所以最后才打印setTimeout

缺点：时间精确度无法掌握，无法控制任务执行顺序

```
function callback2(){
    console.log(2)
}
function callback(){
    console.log(1)
    setTimeout(callback2,0)
}
setTimeout(callback,0)
```

消息队列中就有可能被插入很多系统级的任务。如果中间被插入的任务执行时间过久的话，那么就会影响到后面任务的执行了，所以时间粒度比较大。不适用高精度的需求，例如监听dom变化

监听DOM变化应用场景

如果使用settimeout或setinterval来监听，时间长会导致响应不及时，时间短会造成无用浪费资源，造成页面性能低效。

从DOM4开始，MutationObserver可以监听DOM变化，属性变更，节点内容变更

- 在每次dom发生变化时，渲染引擎将变化封装成微任务并且将其加入微任务队列中，
- MutationObserver采用了异步+微任务的策略
- 通过微任务解决了实时性的问题
- 使用异步解决了同步操作的性能问题

总结

	宏任务	微任务
相应的方法事件	1. script 2. setTimeout/setInterval 3. UI rendering/UI事件 4. postMessage，MessageChannel 5. setImmediate（Node.js）	1. Promise 2. MutaionObserver 3. Object.observe（Proxy 对象替代） 4.process.nextTick（Node.js）
运行顺序	后运行	先运行
是否出发新一轮 tick	会	不会

9.如何理解Process.nextTick

2021/06/22 15:11

1.问题

- Process.nexttick与其他微任务一起时，是如何执行的
- vue中的nextick的逻辑是怎么样的

基本语法

Process.nextick(callBack[arg])

Process.nextick运行逻辑是：

- Process.nextick会将callBack添加到“nex tick queue”中
- “nex tick queue“会在JavaScript的执行栈完成之后，下一轮EventLoop之前按照 出队
- 如果递归调用，需手动终止，否则会永远无法到达轮询阶段

以下的设计理念会出现一些问题

```
let b
function c(callBack){callBack()}
c(()=>{
    console.log(b) //undefined
})
b=2
```

回调函数还没有等JavaScript的执行栈完成，就开始执行回调函数。并未执行任何异步操作。导致在赋值之前出错

以下是放到nextick中，所有的变量、函数在执行回调之前被初始化

```
let b
function c(callBack){
    Process.nextick(callBack)
}
c(()=>{
    console.log(b) //2
})
b=2
```

EventEmitter在node.js使用的例子

```
const EventEmitter = require("event")
const utils = require("util")
function myEmitter(){
    EventEmitter.call(this)
    this.emit("event")
}
util.inherits(MyEmitter, EventEmitter);
const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
    console.log('an event occurred!');
});
```

EventEmitter在构造函数中触发一个事件，因为程序还未运行到回调函数的那行代码。

在构造函数内部设置一个回调函数来保证构造函数完成后去触发事件

```
const EventEmitter = require("event")
const utils = require("util")
function myEmitter(){
    EventEmitter.call(this)
    Process.nextick(()=>{
        this.emit("event")
    })
}
util.inherits(MyEmitter, EventEmitter);
const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
    console.log('an event occurred!');
})
```

Vue中nexttick的使用

```
<template>
  <div class="app">
    <div ref="msg">{{msg}}</div>
    <div v-if="msg1">Message got outside $nextTick: {{msg1}}</div>
    <div v-if="msg2">Message got inside $nextTick: {{msg2}}</div>
    <button @click="changeMsg">
      Change the Message
    </button>
  </div>
</template>
<script>
new Vue({
  el: '.app',
  data: {
    msg: 'Vue',
    msg1: '',
    msg2: '',
  },
  methods: {
    changeMsg() {
      this.msg = "Hello world."
      this.msg1 = this.$refs.msg.innerHTML //Vue
      this.$nextTick(() => {
        this.msg2 = this.$refs.msg.innerHTML //Hello world.
      })
    }
  }
})
</script>
```

this.$nextTick外部的任务都属于微任务， 回调函数是使用了微任务的异步包装延迟器，实际上就是宏任务。是把宏任务放在了下一次tick才会执行。