- JS 是单线程执行的，但是你是否疑惑过什么是线程?
- 讲到线程，那么肯定也得说一下进程 。本质上来说，两个名词都是 CPU 工作时间片的一个描述。
  - 进程描述了 CPU 在运行指令及加载和保存上下文所需的时间，放在应用上来说就代表了一个程序 。线程是进程中的更小单位，描述了执行一段指令所需的时间

> 把这些概念拿到浏览器中来说， 当你打开一个 Tab 页时， 其实就是创建了一个进程，一个进程中可以有多个线程， 比如渲染线程 、 JS 引擎线程、 HTTP 请求线程等等 。当你发起一个请求时， 其实就是创建了一个线程， 当请求结束后，该线程可能就会被销毁

  - 上文说到了 JS 引擎线程和渲染线程，大家应该都知道，在 JS 运行的时候可能会阻止 UI 渲染， 这说明了两个线程是互斥的 。这其中的原因是因为 JS 可以修改 DOM ， 如果在 JS 执行的时候 UI 线程还在工作，就可能导致不能安全的渲染 UI 。这其实也是一个单线程的好处，得益于 JS 是单线程运行的， 可以达到节省内存， 节约上下文切换时间，没有锁的问题的好处

12.2 执行栈

> 涉及面试题：什么是执行栈?

可以把执行栈认为是一个存储函数调用的栈结构， 遵循先进后出的原则

> 当开始执行 JS 代码时， 首先会执行一个 main 函数，然后执行我们的代码 。根据先进后出的原则，后执行的函数会先弹出栈，在图中我们也可以发现， foo 函数后执行， 当执行完毕后就从栈中弹出了

在开发中，大家也可以在报错中找到执行栈的痕迹

```js
function foo() {
  throw new Error( 'error')
}
function bar() {
  foo()
}
bar()
```

> 大家可以在上图清晰的看到报错在 foo 函数， foo 函数又是在 bar 函数中调用的

当我们使用递归的时候， 因为栈可存放的函数是有限制的，一旦存放了过多的函数且没有得到释放的话，就会出现爆栈的问题

```js
function bar() {
  bar()
}
bar()
```

12.3 浏览器中的 Event Loop

> 涉及面试题：异步代码执行顺序？解释一下什么是 Event Loop ？

> 众所周知 JS 是门非阻塞单线程语言， 因为在最初 JS 就是为了和浏览器交互而诞生的。如果 JS 是门多线程的语言话， 我们在多个线程中处理 DOM 就可能会发生问题（一个线程中新加节点， 另一个线程中删除节点）

- JS 在执行的过程中会产生执行环境， 这些执行环境会被顺序的加入到执行栈中。如果遇到异步的代码，会被挂起并加入到 Task （有多种 task ） 队列中。一旦执行栈为空， Event Loop 就会从 Task 队列中拿出需要执行的代码并放入执行栈中执行 ，所以本质上来说 JS 中的异步还是同步行为

```js
console.log( 'script start');

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

```
console.log( 'script end');
```

> 不同的任务源会被分配到不同的 Task 队列中，任务源可以分为 微任务（ microtask ） 和 宏任务（ macrotask ）。在 ES6 规范中，microtask 称为 jobs ， macrotask 称为 task

```js
console.log( 'script start');

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

new Promise((resolve) => {
    console.log( 'Promise')
    resolve()
}).then(function() {
  console.log( 'promise1');
}).then(function() {
  console.log( 'promise2');
});

console.log( 'script end');
// script start => Promise => script end => promise1 => promise2 => setTime
```

> 以上代码虽然 setTimeout 写在 Promise 之前，但是因为 Promise 属于微任务而 setTimeout 属于宏任务

微任务

- process.nextTick
- promise
- Object.observe
- MutationObserver

宏任务

- script
- setTimeout

- setInterval
- setImmediate
- I/O
- UI rendering

> 宏任务中包括了 script ， 浏览器会先执行一个宏任务，接下来有异步代码的话就先执行微任务

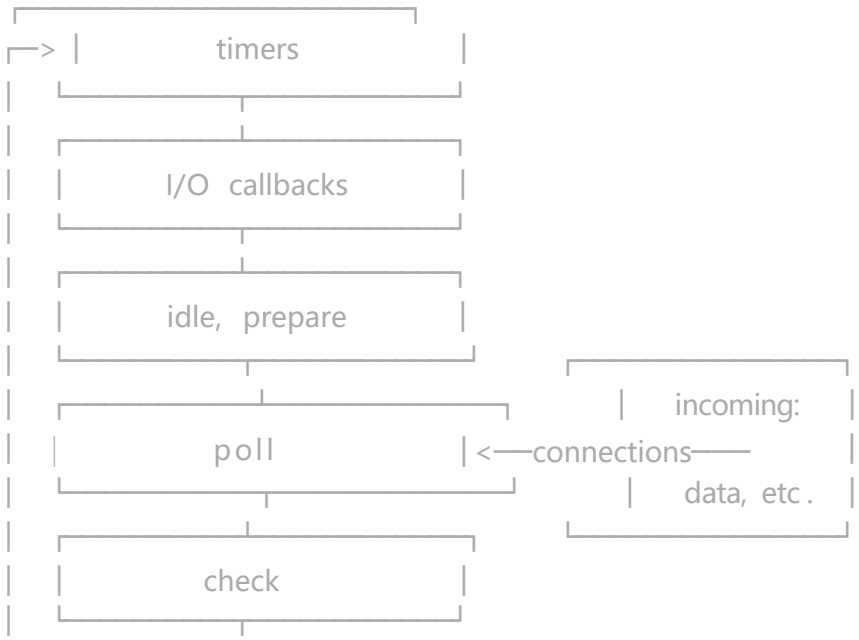
所以正确的一次 Event loop 顺序是这样的

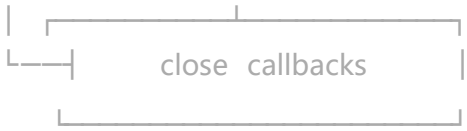
- 执行同步代码， 这属于宏任务
- 执行栈为空，查询是否有微任务需要执行
- 执行所有微任务
- 必要的话渲染 UI
- 然后开始下一轮 Event loop ，执行宏任务中的异步代码

> 通过上述的 Event loop 顺序可知， 如果宏任务中的异步代码有大量的计算并且需要操作 DOM 的话， 为了更快的响应界面响应， 我们可以把操作 DOM 放入微任务中

12.4 Node 中的 Event loop

- Node 中的 Event loop 和浏览器中的不相同。
- Node 的 Event loop 分为 6 个阶段， 它们会按照顺序反复运行





timer

- timers 阶段会执行 setTimeout 和 setInterval
- 一个 timer 指定的时间并不是准确时间，而是在达到这个时间后尽快执行回调，可能会因为系统正在执行别的事务而延迟

I/O

- I/O 阶段会执行除了 close 事件，定时器和 setImmediate 的回调

poll

- poll 阶段很重要，这一阶段中，系统会做两件事情
  - 执行到点的定时器
  - 执行 poll 队列中的事件
- 并且当 poll 中没有定时器的情况下，会发现以下两件事情
  - 如果 poll 队列不为空，会遍历回调队列并同步执行，直到队列为空或者系统限制
  - 如果 poll 队列为空，会有两件事发生
  - 如果有 setImmediate 需要执行，poll 阶段会停止并且进入到 check 阶段执行 setImmediate
  - 如果没有 setImmediate 需要执行，会等待回调被加入到队列中并立即执行回调
  - 如果有别的定时器需要被执行，会回到 timer 阶段执行回调。

check

- check 阶段执行 setImmediate

close callbacks

- close callbacks 阶段执行 close 事件
- 并且在 Node 中，有些情况下的定时器执行顺序是随机的

```js
setTimeout(() => {
    console.log('setTimeout');
}, 0);
setImmediate(() => {
    console.log('setImmediate');
```

```
})
// 这里可能会输出 setTimeout， setImmediate
// 可能也会相反的输出， 这取决于性能
// 因为可能进入 event loop 用了不到 1 毫秒， 这时候会执行 setImmediate
// 否则会执行 setTimeout
```

> 上面介绍的都是 macrotask 的执行情况， microtask 会在以上每个阶段完成后立即执行

```js
setTimeout(()=>{
    console.log( 'timer1')

    Promise.resolve().then(function() {
        console.log( 'promise1')
    })
}, 0)

setTimeout(()=>{
    console.log( 'timer2')

    Promise.resolve().then(function() {
        console.log( 'promise2')
    })
}, 0)

// 以上代码在浏览器和 node 中打印情况是不同的
// 浏览器中一定打印 timer1, promise1, timer2, promise2
// node 中可能打印 timer1, timer2, promise1, promise2
// 也可能打印 timer1, promise1, timer2, promise2
```

> Node 中的 process.nextTick 会先于其他 microtask 执行

```js
setTimeout(() => {
 console.log("timer1");

 Promise.resolve().then(function() {
   console.log("promise1");
 });
}, 0);

process.nextTick(() => {
```

```
  console.log("nextTick");
});
// nextTick, timer1, promise1
```

> 对于 microtask 来说， 它会在以上每个阶段完成前清空 microtask 队列，下图中的 Tick 就代表了 microtask

13 手写 call 、apply 及 bind 函数

> 首先从以下几点来考虑如何实现这几个函数

- 不传入第一个参数，那么上下文默认为 window
- 改变了 this 指向，让新的对象可以执行该函数， 并能接受参数

实现 call

- 首先 context 为可选参数， 如果不传的话默认上下文为 window
- 接下来给 context 创建一个 fn 属性， 并将值设置为需要调用的函数
- 因为 call 可以传入多个参数作为调用函数的参数，所以需要将参数剥离出来
- 然后调用函数并将对象上的函数删除

```js
Function.prototype.myCall = function(context) {
  if (typeof this !== 'function') {
    throw new TypeError( 'Error')
  }
  context = context || window
  context.fn = this
  const args = [...arguments].slice(1)
  const result = context.fn(...args)
  delete context.fn
  return result
}
```

apply实现

> apply 的实现也类似， 区别在于对参数的处理

```js
Function.prototype.myApply = function(context) {
  if (typeof this !== 'function') {
    throw new TypeError( 'Error')
  }
  context = context || window
  context.fn = this
  let result
  // 处理参数和 call 有区别
  if (arguments [1]) {
    result = context.fn(...arguments [1])
  } else {
    result = context.fn()
  }
  delete context.fn
  return result
}
```

bind 的实现

> bind 的实现对比其他两个函数略微地复杂了一点， 因为 bind 需要返回一个函数， 需要判断一些边界问题， 以下是 bind 的实现

- bind 返回了一个函数，对于函数来说有两种方式调用，一 种是直接调用，一 种是通过 new 的方式， 我们先来说直接调用的方式
- 对于直接调用来说， 这里选择了 apply 的方式实现，但是对于参数需要注意以下情况：因为 bind 可以实现类似这样的代码 f.bind(obj, 1)(2) ，所以我们需要将两边的参数拼接起来， 于是就有了这样的实现 args.concat(...arguments)
- 最后来说通过 new 的方式，在之前的章节中我们学习过如何判断 this ，对于 new 的情况来说，不会被任何方式改变 this ，所以对于这种情况我们需要忽略传入的 this

```js
Function.prototype.myBind = function (context) {
  if (typeof this !== 'function') {
    throw new TypeError( 'Error')
  }
  const _this = this
  const args = [...arguments].slice(1)
  // 返回一个函数
  return function F() {
    // 因为返回了一个函数，我们可以 new F()，所以需要判断
    if (this instanceof F) {
      return new _this(...args, ...arguments)
```

```
    }
    return _this.apply(context, args.concat(...arguments))
  }
}
```

14 new

> 涉及面试题： new 的原理是什么？通过 new 的方式创建对象和通过字面量创建有什么区别？

在调用 new 的过程中会发生四件事情

- 新生成了一个对象
- 链接到原型
- 绑定 this
- 返回新对象

> 根据以上几个过程， 我们也可以试着来自己实现一个 new

- 创建一个空对象
- 获取构造函数
- 设置空对象的原型
- 绑定 this 并执行构造函数
- 确保返回值为对象

```js
function create() {
  let obj = {}
  let Con = [].shift.call(arguments)
  obj.__proto__ = Con.prototype
  let result = Con.apply(obj, arguments)
  return result instanceof Object ? result : obj
}
```

- 对于对象来说， 其实都是通过 new 产生的， 无论是 function Foo() 还是 let a = { b : 1 } 。
- 对于创建一个对象来说，更推荐使用字面量的方式创建对象（无论性能上还是可读性）。因为你使用 new Object() 的方式创建对象需要通过作用域链一层层找到 Object ，但是你使用字面量的方式就没这个问题

```js
function Foo() {}
// function 就是个语法糖
// 内部等同于 new Function()
let a = { b: 1 }
// 这个字面量内部也是使用了 new Object()
```

15 instanceof 的原理

> 涉及面试题： instanceof 的原理是什么?

instanceof 可以正确的判断对象的类型， 因为内部机制是通过判断对象的原型链中是不是能找到类型的 prototype

实现一下 instanceof

- 首先获取类型的原型
- 然后获得对象的原型
- 然后一直循环判断对象的原型是否等于类型的原型， 直到对象原型为 null ， 因为原型链最终为 null

```js
function myInstanceof(left, right) {
  let prototype = right.prototype
  left = left.__proto__
  while (true) {
    if (left === null || left === undefined)
      return false
    if (prototype === left)
      return true
    left = left.__proto__
  }
}
```

16 为什么 0.1 + 0.2 != 0.3

> 涉及面试题：为什么 0.1 + 0.2 != 0.3 ？如何解决这个问题?

原因， 因为 JS 采用 IEEE 754 双精度版本（ 64 位)， 并且只要采用 IEEE 754 的语言都有该问题

> 我们都知道计算机是通过二进制来存储东西的，那么 0.1 在二进制中会表示为

```js
// (0011) 表示循环
0.1 = 2^-4 * 1.10011(0011)
```

> 我们可以发现， 0.1 在二进制中是无限循环的一些数字， 其实不只是 0.1 ， 其实很多十进制小数用二进制表示都是无限循环的 。这样其实没什么问题，但是 JS 采用的浮点数标准却会裁剪掉我们的数字。

IEEE 754 双精度版本（64位）将 64 位分为了三段

- 第一位用来表示符号
- 接下去的 11 位用来表示指数
- 其他的位数用来表示有效位，也就是用二进制表示 0.1 中的 10011(0011)

> 那么这些循环的数字被裁剪了，就会出现精度丢失的问题，也就造成了 0.1 不再是 0.1 了， 而是变成了 0.10000000000000002

```
0.10000000000000002 === 0.1 // true
```

> 那么同样的， 0.2 在二进制也是无限循环的，被裁剪后也失去了精度变成了 0.20000000000000002

```
0.20000000000000002 === 0.2 // true
```

> 所以这两者相加不等于 0.3 而是 0.30000000000000004

```
0.1 + 0.2 === 0.30000000000000004 // true
```

> 那么可能你又会有一个疑问， 既然 `0.1` 不是 `0.1` ，那为什么 `console.log(0.1)` 却是正确的呢?

> 因为在输入内容的时候， 二进制被转换为了十进制， 十进制又被转换为了字符串，在这个转换的过程中发生了取近似值的过程，所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
console.log(0.100000000000000002) // 0.1
```

解决

```
parseFloat((0.1 + 0.2).toFixed(10)) === 0.3 // true
```

17 事件机制

> 涉及面试题：事件的触发过程是怎么样的？知道什么是事件代理嘛?

17.1 事件触发三阶段

事件触发有三个阶段:

- `window` 往事件触发处传播， 遇到注册的捕获事件会触发
- 传播到事件触发处时触发注册的事件
- 从事件触发处往 `window` 传播， 遇到注册的冒泡事件会触发

> 事件触发一般来说会按照上面的顺序进行，但是也有特例， 如果给一个 `body` 中的子节点同时注册冒泡和捕获事件，事件触发会按照注册的顺序执行

```
// 以下会先打印冒泡然后是捕获
node.addEventListener(
  'click',
  event => {
```

```
    console.log( '冒泡 ')
  },
  false
)
node.addEventListener(
  'click',
  event => {
    console.log( '捕获 ')
  },
  true
)
```

17.2 注册事件

> 通常我们使用 addEventListener 注册事件，该函数的第三个参数可以是布尔值，也可以是对象 。对于布尔值 useCapture 参数来说，该参数默认值为 false ， useCapture 决定了注册的事件是捕获事件还是冒泡事件 。对于对象参数来说， 可以使用以下几个属性

- capture ： 布尔值，和 useCapture 作用一样
- once ： 布尔值，值为 true 表示该回调只会调用一次，调用后会移除监听
- passive ： 布尔值，表示永远不会调用 preventDefault

> 一般来说， 如果我们只希望事件只触发在目标上， 这时候可以使用 stopPropagation 来阻止事件的进一步传播 。通常我们认为 stopPropagation 是用来阻止事件冒泡的， 其实该函数也可以阻止捕获事件 。stopImmediatePropagation 同样也能实现阻止事件，但是还能阻止该事件目标执行别的注册事件。

js

```
node.addEventListener(
  'click',
  event => {
    event.stopImmediatePropagation()
    console.log( '冒泡 ')
  },
  false
)
// 点击 node 只会执行上面的函数，该函数不会执行
node.addEventListener(
  'click',
```

```
    event => {
      console.log( '捕获 ')
    },
    true
  )
```

17.3 事件代理

> 如果一个节点中的子节点是动态生成的，那么子节点需要注册事件的话应该注册在父节点上

```html
<ul id="ul">
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
</ul>
<script>
    let ul = document.querySelector( '#ul')
    ul.addEventListener( 'click', (event) => {
        console.log(event.target);
    })
</script>
```

事件代理的方式相较于直接给目标注册事件来说，有以下优点：

- 节省内存
- 不需要给子节点注销事件

18 跨域

> 涉及面试题：什么是跨域？为什么浏览器要使用同源策略？你有几种方式可以解决跨域问题？ 了解预检请求嘛？

- 因为浏览器出于安全考虑，有同源策略 。也就是说， 如果协议 、域名或者端口有一个不同就是跨域， `Ajax` 请求会失败。

- 那么是出于什么安全考虑才会引入这种机制呢？ 其实主要是用来防止 CSRF 攻击的 。简单点说， CSRF 攻击是利用用户的登录态发起恶意请求。
- 也就是说，没有同源策略的情况下， A 网站可以被任意其他来源的 Ajax 访问到内容。如果你当前 A 网站还存在登录态，那么对方就可以通过 Ajax 获得你的任何信息 。当然跨域并不能完全阻止 CSRF 。

> 然后我们来考虑一个问题，请求跨域了，那么请求到底发出去没有？ 请求必然是发出去了，但是浏览器拦截了响应 。你可能会疑问明明通过表单的方式可以发起跨域请求， 为什么 Ajax 就不会 。因为归根结底，跨域是为了阻止用户读取到另一个域名下的内容， Ajax 可以获取响应， 浏览器认为这不安全，所以拦截了响应 。但是表单并不会获取新的内容，所以可以发起跨域请求 。同时也说明了跨域并不能完全阻止 CSRF ， 因为请求毕竟是发出去了。

接下来我们将来学习几种常见的方式来解决跨域的问题

18.1 JSONP

> JSONP 的原理很简单，就是利用 <script> 标签没有跨域限制的漏洞 。通过 <script> 标签指向一个需要访问的地址并提供一个回调函数来接收数据当需要通讯时

```html
<script src="http://domain/api?param1=a&param2=b&callback=jsonp"></script>
<script>
    function jsonp(data) {
        console.log(data)
    }
</script>
```

> JSONP 使用简单且兼容性不错，但是只限于 get 请求。

> 在开发中可能会遇到多个 JSONP 请求的回调函数名是相同的， 这时候就需要自己封装一个 JSONP ， 以下是简单实现

```js
function jsonp(url, jsonpCallback, success) {
  let script = document.createElement('script')
  script.src = url
  script.async = true
  script.type = 'text/javascript'
  window[jsonpCallback] = function(data) {
    success && success(data)
  }
  document.body.appendChild(script)
}
jsonp('http://xxx', 'callback', function(value) {
  console.log(value)
})
```

18.2 CORS

- CORS 需要浏览器和后端同时支持。 IE 8 和 9 需要通过 XDomainRequest 来实现。
- 浏览器会自动进行 CORS 通信， 实现 CORS 通信的关键是后端。只要后端实现了 CORS ，就实现了跨域。
- 服务端设置 Access-Control-Allow-Origin 就可以开启 CORS 。 该属性表示哪些域名可以访问资源， 如果设置通配符则表示所有网站都可以访问资源。 虽然设置 CORS 和前端没什么关系，但是通过这种方式解决跨域问题的话，会在发送请求时出现两种情况，分别为简单请求和复杂请求。

简单请求

> 以 Ajax 为例， 当满足以下条件时，会触发简单请求

1. 使用下列方法之一：

- GET
- HEAD
- POST

2. Content-Type 的值仅限于下列三者之一：

- text/plain
- multipart/form-data
- application/x-www-form-urlencoded

> 请求中的任意 `XMLHttpRequestUpload` 对象均没有注册任何事件监听器；`XMLHttpRequestUpload` 对象可以使用 `XMLHttpRequest.upload` 属性访问

复杂请求

> 对于复杂请求来说， 首先会发起一个预检请求，该请求是 `option` 方法的，通过该请求来知道服务端是否允许跨域请求。

对于预检请求来说， 如果你使用过 `Node` 来设置 `CORS` 的话， 可能会遇到过这么一个坑。

以下以 `express` 框架举例

```js
app.use((req, res, next) => {
  res.header( 'Access-Control-Allow-Origin', '*')
  res.header( 'Access-Control-Allow-Methods', 'PUT, GET, POST, DELETE, OPTIO
  res.header(
    'Access-Control-Allow-Headers',
    'Origin, X-Requested-With, Content-Type, Accept, Authorization, Access-
  )
  next()
})
```

- 该请求会验证你的 `Authorization` 字段，没有的话就会报错。
- 当前端发起了复杂请求后，你会发现就算你代码是正确的， 返回结果也永远是报错的 。因为预检请求也会进入回调中，也会触发 `next` 方法， 因为预检请求并不包含 `Authorization` 字段，所以服务端会报错。

> 想解决这个问题很简单， 只需要在回调中过滤 `option` 方法即可

```js
res.statusCode = 204
res.setHeader( 'Content-Length', '0')
res.end()
```

18.3 document.domain

- 该方式只能用于主域名相同的情况下， 比如 `a.test.com` 和 `b.test.com` 适用于该方式。
- 只需要给页面添加 `document.domain = 'test.com'` 表示主域名都相同就可以实现跨域

18.4 postMessage

> 这种方式通常用于获取嵌入页面中的第三方页面数据 。一个页面发送消息， 另一个页面判断来源并接收消息

```js
// 发送消息端
window.parent.postMessage( 'message', 'http://test.com')
// 接收消息端
var mc = new MessageChannel()
mc.addEventListener( 'message', event => {
  var origin = event.origin || event.originalEvent.origin
  if (origin === 'http://test.com') {
    console.log( '验证通过')
  }
})
```

19 存储

> 涉及面试题：有几种方式可以实现存储功能，分别有什么优缺点？什么是 `Service Worker` ？

cookie，localStorage，sessionStorage，indexDB

特性	cookie	localStorage	sessionStorage	indexDB
数据生命周期	一般由服务器生成， 可以设置过期时间	除非被清理，否则一直存在	页面关闭就清理	除非被清理， 否则一直存在
数据存储大小	4K	5M	5M	无限