95.说说event loop

微任务与宏任务
宏任务主要包含：script(整体代码)、setTimeout、setInterval、I/O、UI交互事件、setImmediate(Node.js 环境)

微任务主要包含：Promise、MutaionObserver、process.nextTick(Node.js 环境)
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
- 构造函数执行
- 执行栈为空，查询是否有微任务
- 执行所有微任务
- 必要的话渲染UI
- 开始下一轮Event loop

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

setTimeout(function () {
    console.log('timeout1');
},10);
//Promise.resolve 方法允许调用时不带参数，直接返回一个resolved 状态的 Promise 对象。立即 resolved 的
Promise.resolve().then(console.log('then2'))
new Promise(resolve => {
    console.log('promise1');
    resolve();
    setTimeout(() => console.log('timeout2'), 0);
}).then(function () {
    console.log('then1')
})

console.log('script end');
//script start - then2 - promise1 - script end- then1  - timeout2 -timeout1
```

手写Promise

- 在调用Promise时，会返回一个Promise对象
- 传入一个executor函数，Promise的主要业务流程都在executor函数中执行
- 要保证executor中传入异步操作的话，也可以适用
- then的链式调用&值穿透特性
- 成功了调用resolve，失败调用reject
- 状态不可逆

```
const PENDING = 'PENDING';
const FULFILLED = 'FULFILLED';
const REJECTED = 'REJECTED';

const resolvePromise = (promise2, x, resolve, reject) => {
  // 自己等待自己完成是错误的实现，用一个类型错误，结束掉 promise  Promise/A+ 2.3.1
  if (promise2 === x) {
    return reject(new TypeError('Chaining cycle detected for promise #<Promise>'))
  }
  // Promise/A+ 2.3.3.3.3 只能调用一次
  let called;
  // 后续的条件要严格判断 保证代码能和别的库一起使用
  if ((typeof x === 'object' && x != null) || typeof x === 'function') {
    try {
      // 为了判断 resolve 过的就不用再 reject 了（比如 reject 和 resolve 同时调用的时候）  Promise/A
      let then = x.then;
      if (typeof then === 'function') {
        // 不要写成 x.then，直接 then.call 就可以了 因为 x.then 会再次取值，Object.defineProperty
        then.call(x, y => { // 根据 promise 的状态决定是成功还是失败
          if (called) return;
          called = true;
          // 递归解析的过程（因为可能 promise 中还有 promise）  Promise/A+ 2.3.3.3.1
          resolvePromise(promise2, y, resolve, reject);
        }, r => {
          // 只要失败就失败 Promise/A+ 2.3.3.3.2
          if (called) return;
          called = true;
          reject(r);
        });
      } else {
        // 如果 x.then 是个普通值就直接返回 resolve 作为结果  Promise/A+ 2.3.3.4
        resolve(x);
      }
    } catch (e) {
      // Promise/A+ 2.3.3.2
      if (called) return;
      called = true;
      reject(e)
    }
  } else {
    // 如果 x 是个普通值就直接返回 resolve 作为结果  Promise/A+ 2.3.4
    resolve(x)
  }
}

class Promise {
  constructor(executor) {
    this.status = PENDING;
    this.value = undefined;
    this.reason = undefined;
    this.onResolvedCallbacks = [];
```

```
  this.onRejectedCallbacks= [];

  let resolve = (value) => {
    if(this.status ===  PENDING) {
      this.status = FULFILLED;
      this.value = value;
      this.onResolvedCallbacks.forEach(fn=>fn());
    }
  }

  let reject = (reason) => {
    if(this.status ===  PENDING) {
      this.status = REJECTED;
      this.reason = reason;
      this.onRejectedCallbacks.forEach(fn=>fn());
    }
  }

  try {
    executor(resolve,reject)
  } catch (error) {
    reject(error)
  }
}

then(onFulfilled, onRejected) {
  //解决 onFufilled，onRejected 没有传值的问题
  //Promise/A+ 2.2.1 / Promise/A+ 2.2.5 / Promise/A+ 2.2.7.3 / Promise/A+ 2.2.7.4
  onFulfilled = typeof onFulfilled === 'function' ? onFulfilled : v => v;
  //因为错误的值要让后面访问到，所以这里也要跑出个错误，不然会在之后 then 的 resolve 中捕获
  onRejected = typeof onRejected === 'function' ? onRejected : err => { throw err };
  // 每次调用 then 都返回一个新的 promise  Promise/A+ 2.2.7
  let promise2 = new Promise((resolve, reject) => {
    if (this.status === FULFILLED) {
      //Promise/A+ 2.2.2
      //Promise/A+ 2.2.4 --- setTimeout
      setTimeout(() => {
        try {
          //Promise/A+ 2.2.7.1
          let x = onFulfilled(this.value);
          // x可能是一个proimise
          resolvePromise(promise2, x, resolve, reject);
        } catch (e) {
          //Promise/A+ 2.2.7.2
          reject(e)
        }
      }, 0);
    }

    if (this.status === REJECTED) {
      //Promise/A+ 2.2.3
```

```
        setTimeout(() => {
          try {
            let x = onRejected(this.reason);
            resolvePromise(promise2, x, resolve, reject);
          } catch (e) {
            reject(e)
          }
        }, 0);
      }

      if (this.status === PENDING) {
        this.onResolvedCallbacks.push(() => {
          setTimeout(() => {
            try {
              let x = onFulfilled(this.value);
              resolvePromise(promise2, x, resolve, reject);
            } catch (e) {
              reject(e)
            }
          }, 0);
        });

        this.onRejectedCallbacks.push(()=> {
          setTimeout(() => {
            try {
              let x = onRejected(this.reason);
              resolvePromise(promise2, x, resolve, reject)
            } catch (e) {
              reject(e)
            }
          }, 0);
        });
      }
    });

    return promise2;
  }
}
```

测试代码

```
const promise = new Promise((resolve, reject) => {
  reject('失败');
}).then().then().then(data=>{
  console.log(data);
},err=>{
  console.log('err',err);
})
```

101.描述一下this

- this，函数执行的上下文，可以通过apply，call，bind改变this的指向。对于匿名函数或者直接调用的函数来说，this指向全局上下文（浏览器为window，NodeJS为global），剩下的函数调用，那就是谁调用它，this就指向谁。当然还有es6的箭头函数，箭头函数的指向取决于该箭头函数声明的位置，在哪里声明，this就指向哪里

102.说一下浏览器的缓存机制

- 强缓存：浏览器第一次请求时，会直接下载资源，然后缓存在本地，第二次请求时，直接适用缓存
- 协商缓存：第一次请求缓存且保存缓存标识与时间，重复请求时向服务器发送缓存标识和最后缓存时间，服务器进行校验，如果失效则使用缓存
  协商缓存相关设置
- Exprires：服务端的响应头，第一次请求的时候，告诉客户端，该资源什么时候会过期。**其缺陷是必须保证服务段时间和客户端的时间严格同步**
- Cache-Control：max-age：表示该资源多少时间后过期，解决了客户端和服务端时间必须同步的问题
- If-None-Match/ETag：缓存标识，对比缓存时使用它来标识一个缓存，第一次请求的时候，服务端会返回该标识给客户端，客户端在第二次请求的时候会带上该标识与服务端进行对比并返回If-None-Match标识是否表示匹配。
- Last-modified/If-Modified-Since：第一次请求的时候服务端返回Last-modified表明请求的资源上次的修改时间，第二次请求的时候客户端带上请求头If-Modified-Since，表示资源上次的修改时间，服务端拿到这两个字段进行对比

114.JS有几种数据类型？能画一下它们的内存图吗？

原始：boolean,number,string,null,undefined,symbol--栈
引用：object,array,function--堆

- 原始类型是存储在stack中的简单数据段，占用空间小，大小固定，被频繁使用
- 引用类型是存在heap中的对象，占据空间大，大小不固定，如果存在栈中，会影响性能
  - 引用类型在栈存储了指针，该指针指向堆中该实体的起始地址
  - 当解释器寻找引用值时，会首先检索其在栈中的地址，取得地址后从堆中获得实体

124.JS垃圾回收方法

- 标记清除
  - 这是JavaScript最常见的垃圾回收方式，当变量进入执行环境的时候，比如函数中声明一个变量，垃圾回收器将其标记为“进入环境”，当变量离开环境的时候（函数执行结束）将其标记为“离开环境”

    - 垃圾回收器会在运行的时候给存储在内存中的所有变量加上标记，然后去掉环境中的变量以及被环境中变量所引用的变量（闭包），在这些完成之后仍存在标记的就是要删除的变量了
- 引用计数
    - 在低版本IE中经常会出现内存泄露，很多时候就是因为其采用引用计数方式进行垃圾回收。引用计数的策略是跟踪记录每个值被使用的次数，当声明了一个 变量并将一个引用类型赋值给该变量的时候这个值的引用次数就加1，如果该变量的值变成了另外一个，则这个值得引用次数减1，当这个值的引用次数变为0的时 候，说明没有变量在使用，这个值没法被访问了，因此可以将其占用的空间回收，这样垃圾回收器会在运行的时候清理掉引用次数为0的值占用的空间

130.简单实现Funtion.bind函数

```
  if (!Function.prototype.bind) {
    Function.prototype.bind = function(that) {
      var func = this, args = arguments;
      return function() {
        return func.apply(that, Array.prototype.slice.call(args, 1));
      }
    }
  }
  // 只支持 bind 阶段的默认参数:
  func.bind(that, arg1, arg2)();

  // 不支持以下调用阶段传入的参数:
  func.bind(that)(arg1, arg2);
// 第二版
Function.prototype.bind2 = function (context) {

    var self = this;
    // 获取bind2函数从第二个参数到最后一个参数
    var args = Array.prototype.slice.call(arguments, 1);

    return function () {
        // 这个时候的arguments是指bind返回的函数传入的参数
        var bindArgs = Array.prototype.slice.call(arguments);
        return self.apply(context, args.concat(bindArgs));
    }

}
```

133.MVVM

- 在 JQuery 时期，如果需要刷新 UI 时，需要先取到对应的 DOM 再更新 UI，这样数据和业务的逻辑就和页面有强耦合

- 在 MVVM 中，UI 是通过数据驱动的，数据一旦改变就会相应的刷新对应的 UI，UI 如果改变，也会改变对应的数据。这种方式就可以在业务处理中只关心数据的流转，而无需直接和页面打交道。ViewModel 只关心数据和业务的处理，不关心 View 如何处理数据，在这种情况下，View 和 Model 都可以独立出来，任何一方改变了也不一定需要改变另一方，并且可以将一些可复用的逻辑放在一个 ViewModel 中，让多个 View 复用这个 ViewModel

> 在MVVM中，最核心的就是数据的双向绑定，Angluar的脏数据检测，Vue中的数据劫持

- 脏数据检测
  - 当触发了指定事件后会进入脏数据检测，这时会调用 $digest 循环遍历所有的数据观察者，判断当前值是否和先前的值有区别，如果检测到变化的话，会调用 $watch 函数，然后再次调用 $digest 循环直到发现没有变化。循环至少为二次 ，至多为十次
  - 脏数据检测虽然存在低效的问题，但是不关心数据是通过什么方式改变的，都可以完成任务，但是这在 Vue 中的双向绑定是存在问题的。并且脏数据检测可以实现批量检测出更新的值，再去统一更新 UI，大大减少了操作 DOM 的次数
- 数据劫持
  - Vue 内部使用了 Obeject.defineProperty()和proxy来实现双向绑定，通过这个函数可以监听到 set 和 get的事件

```
var data = { name: 'yck' }
observe(data)
let name = data.name // -> get value
data.name = 'yyy' // -> change value

function observe(obj) {
  // 判断类型
  if (!obj || typeof obj !== 'object') {
    return
  }
  Object.keys(data).forEach(key => {
    defineReactive(data, key, data[key])
  })
}

function defineReactive(obj, key, val) {
  // 递归子属性
  observe(val)
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter() {
      console.log('get value')
      return val
    },
    set: function reactiveSetter(newVal) {
      console.log('change value')
      val = newVal
    }
  })
}
```

> 以上代码简单的实现了如何监听数据的 set 和 get 的事件，但是仅仅如此是不够的，还需要在适当的时候给属性添加发布订阅

```
<div>
    {{name}}
</div>
```

> 在解析如上模板代码时，遇到 {name} 就会给属性 name 添加发布订阅

```
// 通过 Dep 解耦
class Dep {
  constructor() {
    this.subs = []
  }
  addSub(sub) {
    // sub 是 Watcher 实例
    this.subs.push(sub)
  }
  notify() {
    this.subs.forEach(sub => {
      sub.update()
    })
  }
}
// 全局属性，通过该属性配置 Watcher
Dep.target = null

function update(value) {
  document.querySelector('div').innerText = value
}

class Watcher {
  constructor(obj, key, cb) {
    // 将 Dep.target 指向自己
    // 然后触发属性的 getter 添加监听
    // 最后将 Dep.target 置空
    Dep.target = this
    this.cb = cb
    this.obj = obj
    this.key = key
    this.value = obj[key]
    Dep.target = null
  }
  update() {
    // 获得新值
    this.value = this.obj[this.key]
    // 调用 update 方法更新 Dom
    this.cb(this.value)
  }
}
var data = { name: 'yck' }
observe(data)
// 模拟解析到 `{{name}}` 触发的操作
new Watcher(data, 'name', update)
// update Dom innerText
data.name = 'yyy'
```

> 接下来,对 defineReactive 函数进行改造

```
function defineReactive(obj, key, val) {
  // 递归子属性
  observe(val)
  let dp = new Dep()
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter() {
      console.log('get value')
      // 将 Watcher 添加到订阅
      if (Dep.target) {
        dp.addSub(Dep.target)
      }
      return val
    },
    set: function reactiveSetter(newVal) {
      console.log('change value')
      val = newVal
      // 执行 watcher 的 update 方法
      dp.notify()
    }
  })
}
```

> 以上实现了一个简易的双向绑定，核心思路就是手动触发一次属性的 getter 来实现发布订阅的添加

Proxy与Obeject.defineProperty对比

- Object.defineProperty只能对属性进行数据劫持，需要深度遍历整个对象
- 对于数组不能监听到数据的变化(Vue中使用了hack方法检测数据的变化)