day20 闭包及promise

闭包

概述：

函数嵌套函数，内部函数拥有外部函数的引用，对应的外部的函数的这个引用不会被gc回收。

作用域

- 全局作用域 （global）

  里面的this指向对应的global对象 也就是说对应的全局作用域的变量其实都是global对象的属性
- 局部作用域 （函数作用域）

函数的预编译过程

局部作用域

1. 创建AO对象 （Activation Object 执行上下文）（开辟的内存空间）（会在堆中开辟一个跟函数名一样名字的空间）
2. 给函数里面的形参、变量进行赋值操作 （undefined）
3. 形参和实参同步
4. 找到函数声明 赋值给函数体

```
function fn(a=1,b=2){
    console.log(a,b,c) //1,2,undefined
    var c = function(){
        console.log(1)
    }
    console.log(c) //函数
}
fn()
```

全局作用域（全局暴露 对变量来说有全局污染的问题）

1. 创建GO对象 （Global Object 全局对象）（给对应的global对象添加属性）
2. 给函数里面的形参、变量进行赋值操作 （undefined）
3. 形参和实参同步
4. 找到函数声明 赋值给函数体

```
console.log(a) //function
var a = 10
function a(){}
console.log(a) //10 (全局污染)
```

函数执行过程

- 去堆中找到对应的函数名所有对应的地址，放入执行栈中执行（压栈）
- 执行过程中（开辟对应的执行空间）使用的对应执行上下文进行执行（函数体）
- 返回对应的结果（结果不会销毁）

- 执行完（销毁对应的执行空间 以及对应的函数体） (出栈)

```
function fn(){
    var a = 1
    console.log(++a)
}
function fn1(){
    var b = 2
    console.log(--b)
    return b
}
fn()//压栈 2
fn1()//压栈 1
fn()//压栈 2
fn1()//压栈 1
var a = 20 //没有被回收 因为它是global的属性 （程序没关闭 不会被回收）
cosole.log(a++) //20
cosole.log(a++) //21
```

实现一个函数调用一次自增一次

```
// let i = 0 //不会被销毁
// function fn() {
//   i++ //全局变量
//   console.log(i)
// }
// fn() //1
// fn() //2
// fn() //3
// function fn(){
//   var i = 0 //i每次调用都会被销毁
//   i++
//   console.log(i)
// }
// fn() //1
// fn() //1
// fn() //1
//标记清除 （变量） 引用计数 （对象） 只有保持引用就不会被回收
//可以return内容不会被销毁的特性 返回一个引用给外界使用 那么外界在使用的时候就保持了引用就不会被回收
// function fn(){
//   var i = 0 //i每次调用都会被销毁
//   i++
//   console.log(i)
//   return {
//     i
//   }
// }
// var obj = fn() //使用obj进行接收 这个时候就将函数的内容返回的对象应用加给对应的变量（全局）
// console.log(obj.i++)//1
// console.log(obj.i++)//2
// console.log(obj.i++)//3
//也就是说通过上面的形式 我们可以返回一个引用数据类型 让外界保持引用那么这个内容就不会被回收
//那么对应的函数也是一个引用数据类型 （同理）
function fn(){
```

```
    var i = 0
    return function(){
        i++ //标记为yes i也不会回收
        console.log(i)
    }
}
//闭包占用内存大 容易造成内存泄漏
var closureFn = fn() //接收一个函数
closureFn()//1
closureFn()//2
closureFn()//3
//解决内存泄漏
fn = null
```

从上述代码可以得到对应函数中return 一个引用数据类型 在这个引用数据类型内容保持对应的外部的一个参数或者是变量的引用那么这个变量将不会被回收。（扩大了函数内容变量的作用范围）

闭包的优缺点

优点

- 防止变量的全局污染 （这个变量是局部变量）
- 扩大了变量的作用域
- 缓存对应的数据

缺点

- 过度使用闭包容易造成内存泄漏 （当前这个空间已经没用了 但是空间没有被销毁）
- 对应内存的耗费大（性能差） （内部函数时刻保持对应的外部函数变量或者参数的引用）

闭包的用途

- 作为缓存 （容量小 速度快）
- 防抖
- 节流
- 函数柯里化

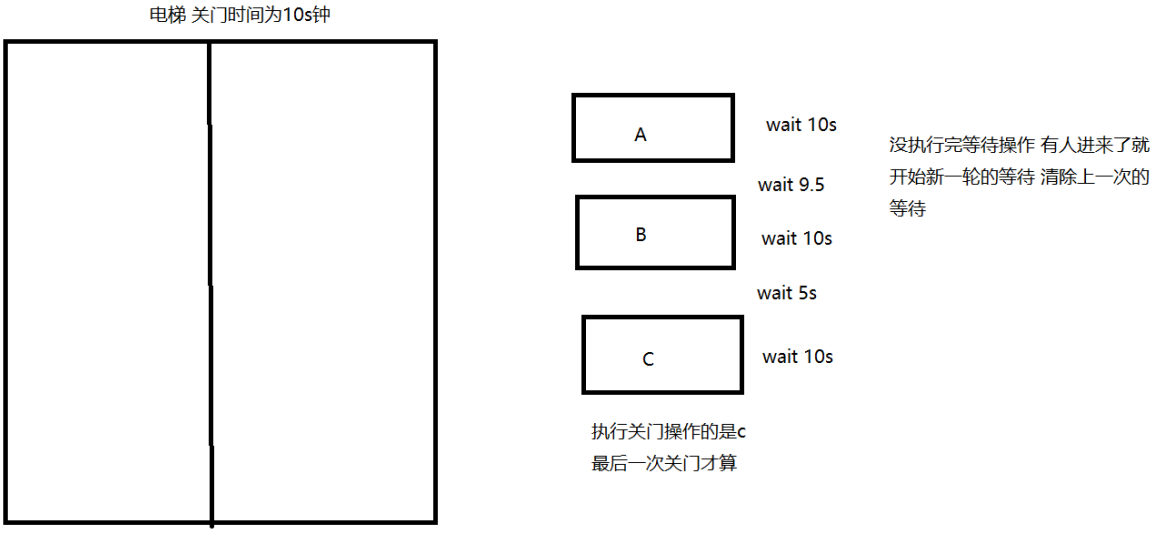
防抖（debounce)

概述:

在规定时间内只执行一次 （执行最后一次）

案例:

坐电梯



代码实现

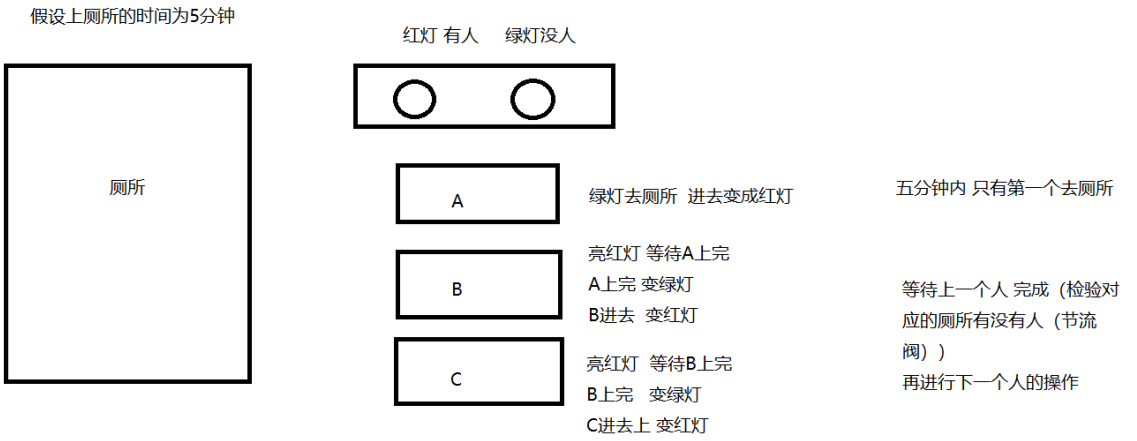
```
function debounce(callback,delay){
    var timer = null
    return function(){
        //清除上一次
        clearTimeout(timer)
        //开始新的等待
        timer = setTimeout(()=>{
            //调用函数
            callback()
            //清除定时器
            clearTimeout(timer)
        },delay)
    }
}
```

节流 （throttling）

在规定时间内执行第一次，执行完继续执行。（减少对应的次数）

案例

高铁厕所案例



代码实现

```
function throttling(callback,delay){
    var timer = null //节流阀 判断是否有操作在进行
    return function(){
        //判断是否有操作进行
        //如果没有那么就执行我的操作
        if(!timer){
            timer = setTimeout(()=>{
                callback() //执行操作
                clearTimeout(timer)
                timer = null //做完释放节流阀
            },delay)
        }
        //如果有就什么都不做
    }
}
```

函数柯里化

将一个多参数的函数 分成多个函数（彼此之间可以任意组成）

示例

```
//求和的函数
function sum(a,b,c){
    return a+b+c
}
sum(1,2,3)//6
sum(1,2,4)
```

简单函数柯里化

```
function sum(a){
    return function(b){
        retrun function(c){
            return a+b+c
        }
    }
}
let fn = sum(1)(2)
fn(3) //6
fn(4) //7
// console.log(sum(1,2,3)) //返回函数
// console.log(sum(1)(2,3)) //返回函数
```

高阶函数柯里化

```
//包装  将普通的函数变成柯里化函数
//当前currying函数可以传入一个函数 还可以传入对应的参数（用于执行对应的fn的）
function currying(fn) {
    //接收传入的参数 arguments (不包含传入的函数)
    let args = Array.prototype.slice.call(arguments, 1)
    return function () {
        //接收currying传入的参数和对应的函数的参数的数组
        let allArg = args.concat([...arguments])
        //判断参数是否到达对应的个数
        if (allArg.length < fn.length) {
            //没有到达个数返回对应的函数
            return currying.apply(this, [fn].concat(allArg))
        } else {
            //到达个数返回值
            //调用传入的函数 将参数传入
            return fn.apply(this, allArg)
        }
    }
}
// let fn = currying(sum,1,2)
// console.log(fn(3))
// console.log(fn(4))
let fn = currying(sum)
console.log(fn(1))
console.log(fn(1)(2))
console.log(fn(1)(2)(3))
console.log(fn(1,2,3))
console.log(fn(1,2)(3))
console.log(fn(1)(2,3))
console.log(fn(1)()()()()()()()())
console.log(fn(1)()()()()(2)()()()()()()()()(3))
```

函数柯里化核心

参数个数没到对应的个数返回的是函数 参数个数到了返回的是值

Promise

概述

promise是ES6新增的一个类，翻译为承诺，它是用于解决异步问题的（替代回调函数）。

promise的三种状态

- 等待 pedding
- 成功 fulfiiled （有对应的处理）
- 失败 rejected （有对应的处理）

promise的构建

```
//传入函数来进行构建 传入的函数内有俩个参数传递 （resolve、reject都是函数）
//new Promise是同步执行的代码 它的里面可以包含异步代码
new Promise((resolve,reject)=>{
    console.log('hello') //也是同步的
})
```

问题提出

异步代码如何同步执行

使用回调函数

```
//回调函数解决的问题
function fn(message, delay = 0, callback) {
    setTimeout(() => {
        console.log(message)
        callback()
    }, delay)
}
//这个代码没有任何问题 但是这个代码的可维护性低 可读性低 （回调地狱）这个代码没有可用性（避免回调函数不能嵌套过深）
fn('1', 1000, () => {
    fn('2', 500, () => {
        fn('3', 1000, () => {
            fn('4', 2000, () => {
                fn('5', 100, () => {
                    fn('6', 1000, () => {
                        // ....
```

```
                    })
                })
            })
        })
    })
})
```

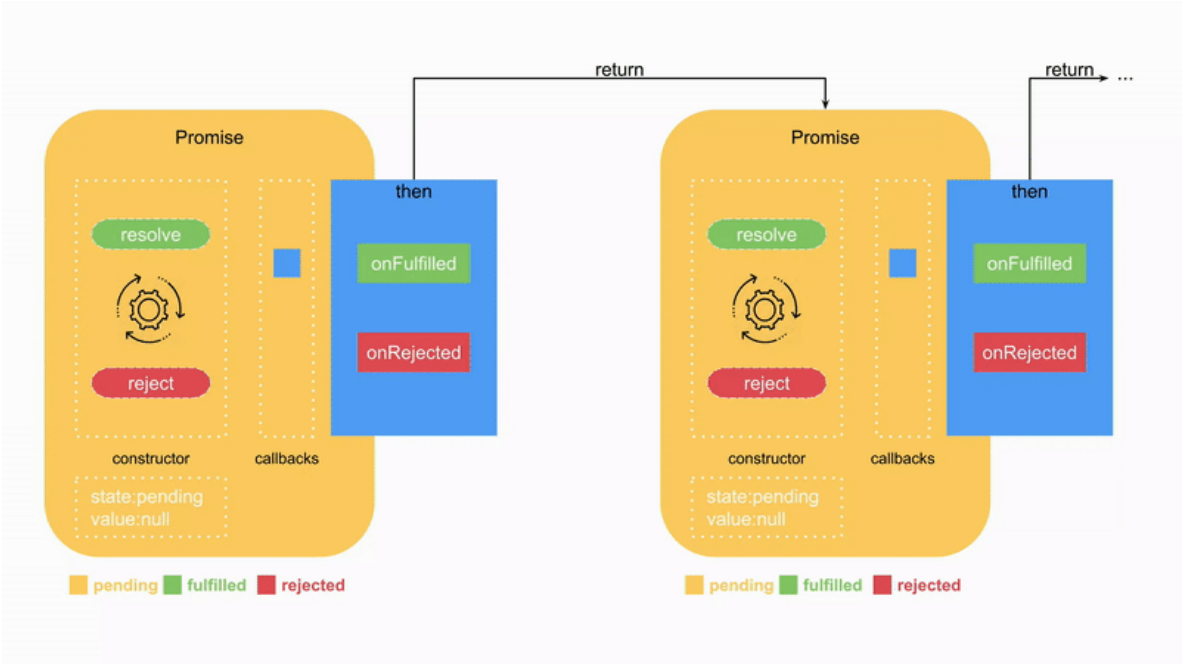
通过以上代码会发现回调函数嵌套过深会导致回调地狱问题 （回调地狱它的代码没有可用性 要避免）

使用promise来解决对应的回调地狱

核心在对应的then方法返回一个新的promise 再调用下一个then

```
//resolve 触发成功的函数 对应的处理的方法 then
//reject 触发失败的函数 对应的处理的方法 catch
new Promise((resolve, reject) => {
    setTimeout(() => {
        console.log(1);
        //调用成功的函数
        resolve()
    }, 1000)
}).then(() => {
    //在then方法里面通过返回对应的新的promise来调用下一个then
    return new Promise((resolve,reject) => {
        setTimeout(() => {
            console.log(2);
            resolve()
        }, 500)
    })
}).then(()=>{
    return new Promise((resolve,reject) => {
        setTimeout(() => {
            console.log(3);
            resolve()
        }, 500)
    })
}).then(()=>{
    return new Promise((resolve,reject) => {
        setTimeout(() => {
            console.log(4);
            resolve()
        }, 200)
    })
})
```

图示



promise的方法及属性

属性 （内置属性 无法访问）

PromiseState 表示当前状态

PromiseResult 表示当前的结果

方法

原型方法

then 处理对应的一个promise状态变化的

- then有俩个参数函数 一个是成功的处理函数 一个是拒绝的错误函数
- 成功的处理函数里面可以接收对应的resolve函数调用携带的参数
- 拒绝的处理函数里面可以接收对应的reject函数调用携带的参数
- 拒绝的处理函数还可以捕获的对应的promise中抛出的错误
- 在then中return的数据会被下一级的then方法接收
- then里面会发生值穿透 （上层的then方法没有对应的函数做为参数）

```
let promise = new Promise((resolve,reject)=>{
    console.log('hello')
    //promise的状态是唯一的
    // resolve(123) //成功状态
    // return 123  //状态没有发生变化 不会调用then
    // reject(456) //拒绝状态
    throw new Error('你好错了')
    //then方法 它里面传入俩个参数 (都是函数)
    //第一个参数为成功的处理函数(携带 resolve函数传递的数据 相当于接收了return的数据
return也会调用的对应的then)
    //第二个参数为拒绝的处理函数(携带 reject函数传递的参数 或者 捕获抛出的错误)
}).then((result)=>{
    console.log(result)
},(err)=>{
    console.log(err)
})
//then的传值 使用return
```

```
new Promise((resolve)=>{
    resolve()
}).then(()=>{
    return 123
}).then((result)=>{
    console.log(result)//123
})
//then的值穿透 发生在上层的then方法里面没有函数为参数
new Promise((resolve)=>{
    resolve('hello')
})
// .then(()=>{}) 只有有参数不会发生值传统
    .then()
    .then()
    .then((result)=>{
    console.log(result)//123
})
```

catch 捕获promise错误 以及获取对应的promise的reject方法调用的结果 （跟then方法中的第二个参数一模一样）

```
new Promise((resolve,reject)=>{
    // reject(11)
    throw new Error('我错了')
}).catch((err)=>{ //接收reject函数调用的参数 或者捕获对应的错误
    console.log(err)
})
//catch也会发生值穿透
new Promise((resolve,reject)=>{
    // reject(11)
    throw new Error('我错了')
}).catch()
    .catch()
    .catch((err)=>{ //接收reject函数调用的参数 或者捕获对应的错误
    console.log(err)
})
```

finally 状态发生变化就会调用的函数

```
//finally 不管成功还是失败都会调用的函数(状态变化)
new Promise((resolve,reject)=>{
    resolve(111)
    // reject(11)
    // throw new Error('我错了')
}).finally(()=>{
    console.log('完成了')
})
```

静态方法

- **resolve 产生一个状态为成功的promise**

```
//返回成功状态的promise resolve
let promise1 =  Promise.resolve(456)
console.log(promise1)
promise1.then((result)=>{
    console.log('成功了'+result)
})
```

- **reject 产生一个状态为拒绝的promise**

```
//reject方法 产生一个拒绝状态的promise
let promise2 = Promise.reject(123)
console.log(promise2)
promise2.catch((err)=>{
    console.log(err)
})
```

- **reac 竞速 （传入一个promise数组 返回执行完成最快的promise）**

```
//竞速方法race 比较执行速度 谁先执行完(不区分成功失败)返回执行快的promise
let promise3 = Promise.reject(1)
let promise4 = Promise.reject(2)
// let promise3 = Promise.resolve(1)
let newPromise = Promise.race([promise3,promise4]) //成功状态比拒绝状态要快
console.log(newPromise)
```

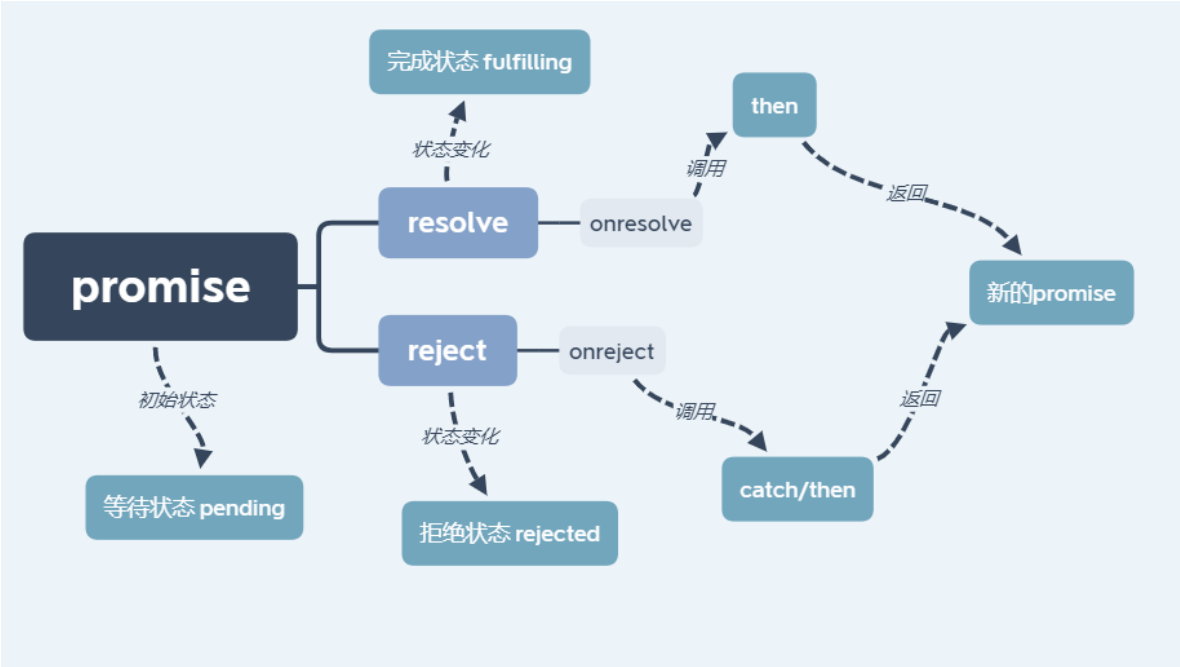
- **all 并行执行传入promise数组 （遇到了rejected那么就获取到rejected的promise 如果全部成功会接收所有的promise结果）**

```
//all 同步并行执行多个promise 返回一个promise （ 全部成功 包含所有的结果 状态为成功）
// （如果遇到了拒绝那么返回的就是拒绝状态的 只会接收到拒绝传递的值）
let promise5 = Promise.reject(5)
let promise6 = Promise.reject(6)
let promise7 = Promise.reject(7)
// let promise7 = Promise.resolve(7)
let promise8 = Promise.reject(8)
let promiseAll = Promise.all([promise5,promise6,promise7,promise8])
console.log(promiseAll)
```

- **allsettled 并行执行传入promise数组 （只要执行完成那么返回都是对应的成功状态 且会接收所有的promise结果）**

```
//allSettled 并行执行多个promise 返回所有的结果 （只要执行完成那么状态就是成功） 新增的
let promiseAllsettled =
Promise.allSettled([promise5,promise6,promise7,promise8])
console.log(promiseAllsettled)
```

promise状态变更



async及awiat

概述

async和awiat是es7新增的修饰关键词，**async是用于修饰对应的函数的**，被async修饰的函数 执行会返回一个promise对象，awiat和async属于语法糖 （**awiat一定要在async里面使用 且awiat修饰的是promise**）

async

概述

async表示异步、async是用于修饰函数的 且修饰的函数调用会返回一个promise对象

注意事项

- 修饰的函数内容返回值相当于调用了resolve方法 返回的值会被传递给then
- 在修饰函数内报错相当于调用了reject方法 错误会被传递给catch

```
//async用于修饰对应的函数的 异步
async function sayHello(){
    console.log('123')
    //因为它返回的值 在async修饰的函数内容 返回内容相当于调用了resolve方法 更改状态为成功
    //报错相当于调用了reject
    // throw new Error() 相当于 reject调用
    //函数默认返回undefined
    // return 123 相当于resolve调用
}
//async修饰的函数调用会返回一个promise对象
let promise = sayHello()
console.log(promise)
promise.then((result)=>{
    console.log(result)
})
```

awiat

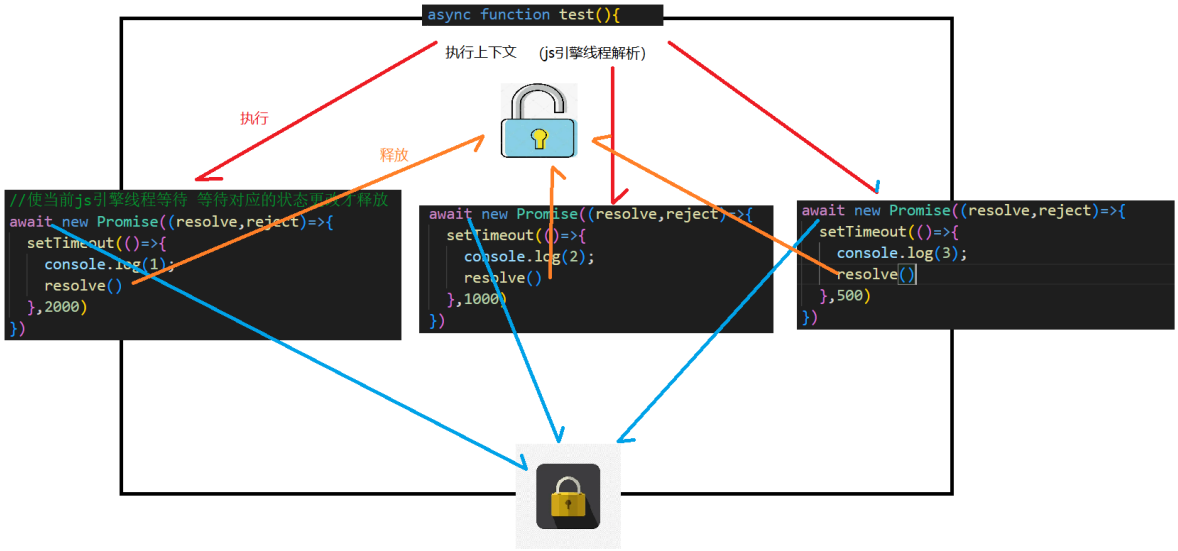
概述

awiat是用于修饰promise的，它只能在async修饰的函数内使用，它会让当前的(js引擎线程在当前的执行上下文中等待)等待，当前修饰promise的状态不为pending的时候就会释放。

```
async function say(){
    //使当前js引擎线程等待 等待对应的状态更改才释放
    await new Promise((resolve,reject)=>{
        setTimeout(()=>{
            console.log(1);
            resolve()
        },2000)
    })
    console.log(2)
}
say() // 1 2
```

使用async和awiat解决回调地狱

```
async function test(){
    //使当前js引擎线程等待 等待对应的状态更改才释放
    await new Promise((resolve,reject)=>{
        setTimeout(()=>{
            console.log(1);
            resolve()
        },2000)
    })
    await new Promise((resolve,reject)=>{
        setTimeout(()=>{
            console.log(2);
            resolve()
        },1000)
    })
    await new Promise((resolve,reject)=>{
        setTimeout(()=>{
            console.log(3);
            resolve()
        },500)
    })
    console.log(4)
}
test()
```



EventLoop

概述:

eventLoop事件轮询，针对于对应的异步任务的执行相关操作（定时器、事件、请求....）。

事件队列

队列是先进先出一个数据结构，事件队列就是其异步任务的一个队列（先进去先执行 后进行后执行）

执行栈

栈是一个先进后出的一个数据结构，它主要用执行对应的js代码。

事件队列的分类

宏任务 （大的任务 线程直接分配的）

script 标签 定时器（setInterval，setTimeout）...

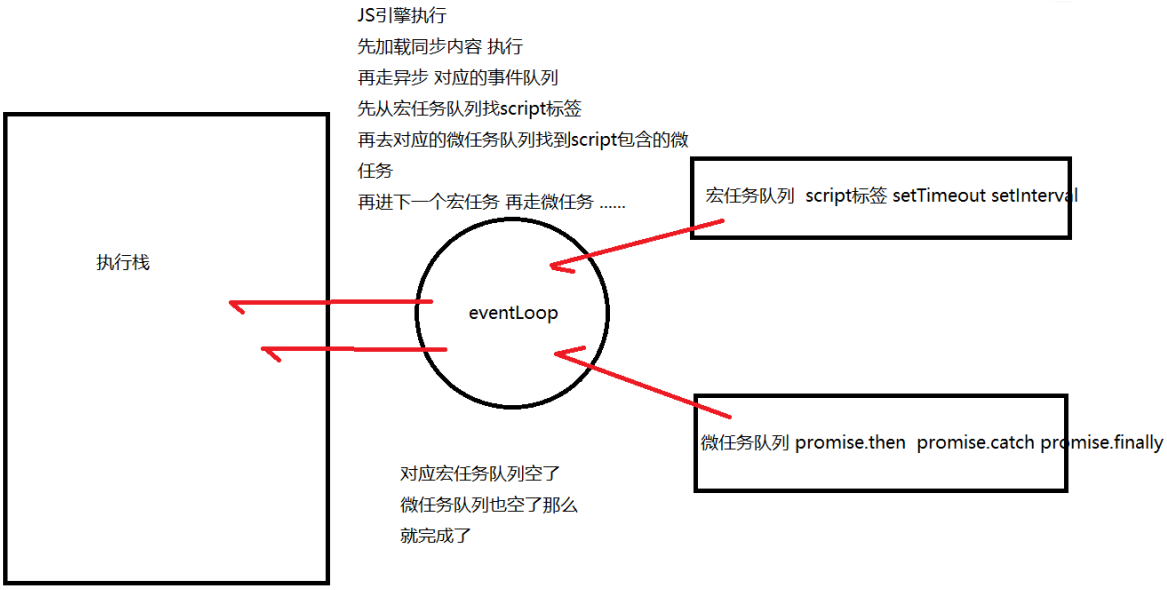
微任务 （小的任务）

promise.then promise.catch promise.finally

宏任务进入宏任务队列 微任务进入微任务队列

eventLoop主要是控制宏任务队列的任务执行及微任务队列的任务执行

- 先宏后微
- 一个个的宏任务走进再走其包含的微任务


JS引擎执行
先加载同步内容 执行
再走异步 对应的事件队列
先从宏任务队列找script标签
再去对应的微任务队列找到script包含的微任务
再进下一个宏任务 再走微任务 .....
宏任务队列 script标签 setTimeout setInterval
执行栈
eventLoop
微任务队列 promise.then promise.catch promise.finally
对应宏任务队列空了
微任务队列也空了那么
就完成了