第 11 章 期约与异步函数

回调地狱实至名归

function double(value, success, failure) {  
 setTimeout(() => {  
 try {  
 if (typeof value !== 'number') {  
 throw 'Must provide number as first argument';  
 }  
 success(2 \* value);  
 } catch (e) {  
 failure(e);  
 }  
 }, 1000);  
 }  
const successCallback = (x) => {  
 double(x, (y) => console.log(`Success: ${y}`));  
};  
const failureCallback = (e) => console.log(`Failure: ${e}`);  
double(3, successCallback, failureCallback);

## 1、Promise.resolve

通过调用Promise.resolve()静态方法，可以实例化一个解决的期约。

let p1 = new Promise((resolve, reject) => resolve());   
let p2 = Promise.resolve();  
//两种方式等效  
//Promise {<fulfilled>: undefined}

Promise.resolve是一个幂等操作。若传入的参数若是promise类型，则返回该promise

let p = Promise.resolve(7);  
console.log(Promise.resolve(p) === p) //true  
  
let p2 = Promise.resolve(Promise.resolve('bar'))  
//Promise {<fulfilled>: 'bar'}  
let p3 = Promise.resolve(Promise.reject('bar'))   
//Promise {<rejected>: 'bar'}  
console.log(Promise.resolve(p3)=== p3) //true

### 2、Promise.reject

Promise.reject于Promise.resolve不同的是，若传入的参数param是promise类型，则返回一个值为参数param、状态为rejected的promise对象。

setTimeout(console.log, 0, Promise.reject(Promise.resolve()));   
// Promise <rejected>: Promise <resolved>

let p1 = new Promise((resolve)=>{  
resolve(666)  
})  
let p2 = p1.then((res)=>{  
 throw res  
})  
console.log(p1) //Promise {<fulfilled>: 666}  
setTimeout(console.log,0,p2) //Promise {<rejected>: 666}

.then后抛出错误，返回的是一个状态为rejected的promise，并不是因为p1的状态变了，p1还是那个p1.

### 3、try/catch 块并不能捕获promise.reject抛出的错误

因为拒绝期约的错误并没有抛到执行同步代码的线程里，而是通过浏览器异步消息队列来处理的。

在promise中抛出错误时，因为错误实际上是从消息队列中异步抛出的，所以并不会阻止运行时继续执行同步指令

Promise.reject(Error('foo'));   
console.log('bar');  
// bar  
// Uncaught (in promise) Error: foo

所以异步错误只能通过异步的 onRejected 处理程序捕获

// 正确

Promise.reject(Error('foo')).catch((e) => {});

// 不正确

try {  
 Promise.reject(Error('foo'));  
 } catch(e) {}

### 4、处理程序

onResolved处理程序的返回值会被 Promise.resolve() 包装

onRejected 处理程序返回的值也会被 Promise.resolve() 包装。

若调用then()时不传处理程序，则原样向后传

let p1 = Promise.resolve('foo');  
  
let p2 = p1.then(); //不传处理程序  
  
   
setTimeout(console.log,0,p1) //Promise {<fulfilled>: 'foo'}  
  
setTimeout(console.log,0,p2) //Promise {<fulfilled>: 'foo'}  
  
  
let p3 = p2.then(res=>console.log(res)) //foo  
setTimeout(console.log,0,p3) //Promise {<fulfilled>: undefined}  
  
let p5 = null;  
let p4 = p1.then(res=>{  
 p5 = Promise.resolve('哈哈哈')  
 return p5  
})  
  
console.log(p4 === p5)

## 5、promise的非重入

在一个解决期约上调用 then()会把 onResolved 处理程序推进消息队列。

但这个处理程序在当前线程上的同步代码执行完成前不会执行。

因此，跟在 then()后面的同步代码一定先于 处理程序执行。

## 6、promise.race

如果有一个期约拒绝，只要它是第一个落定的，就会成为拒绝合成期约的理由。之后再拒绝的期约 不会影响最终期约的拒绝理由。不过，这并不影响所有包含期约正常的拒绝操作。

与 Promise.all() 类似，合成的期约会静默处理所有包含期约的拒绝操作

let p1 = Promise.reject(3)  
let p2 = new Promise((resolve, reject) => setTimeout(reject, 1000))  
let p = Promise.race([p1,p2]);  
   
p.catch((reason) => setTimeout(console.log, 0, reason)); // 3  
setTimeout(console.log,500,p2) //Promise {<pending>}  
setTimeout(console.log,2000,p2) //Promise {<rejected>: undefined}  
  
//p2的reject也被静默处理了

## 7、reduce配合串行期约

function addTwo(x) {return x + 2;}  
 function addThree(x) {return x + 3;}  
 function addFive(x) {return x + 5;}  
 function compose(...fns) {  
 return (x) => fns.reduce((promise, fn) => promise.then(fn), Promise.resolve(x))  
 }  
let addTen = compose(addTwo, addThree, addFive);  
addTen(8).then(console.log); // 18

## 8、异步函数

异步函数如果使用 return 关键字返回了值(如果没有 return 则会返回 undefined)，这 个值会被 Promise.resolve()包装成一个期约对象。

所以异步函数始终返回期约对象。

async function foo() {   
 console.log(1);   
 return 3;  
}  
 // 给返回的期约添加一个解决处理程序  
foo().then(console.log);  
 console.log(2);  
 // 1 // 2 // 3

在异步函数中抛出错误会返回拒绝的期约

async function foo() {   
console.log(1); throw 3;  
}  
// 给返回的期约添加一个拒绝处理程序   
foo().catch(console.log);   
console.log(2);  
// 1   
// 2   
// 3

## 9、await关键字导致的执行顺序问题

JavaScript 运行时在碰 到 await 关键字时，会记录在哪里暂停执行。等到 await 右边的值可用了，JavaScript 运行时会向消息 队列中推送一个任务，这个任务会恢复异步函数的执行。

因此，即使 await 后面跟着一个立即可用的值，函数的其余部分也会被异步求值。

async function foo() {  
 console.log(2);  
 await null;  
 console.log(4);  
}  
 console.log(1);  
 foo();  
 console.log(3);  
 // 1  
 // 2  
 // 3  
 // 4

await后面若是一个rejected的promise对象，则可以用try，catch捕获异常



## 10、实现非阻塞的暂停sleep()

function sleep(delay) {  
 return new Promise((resolve) => setTimeout(resolve, delay));  
}  
async function foo() {  
 const t0 = Date.now();  
 await sleep(1500); // 暂停约 1500 毫秒   
 console.log(Date.now() - t0);  
}  
foo();  
// 1502

## 11、手写Promise

const PENDING = 'pending';  
const FULFILLED = 'fulfilled';  
const REJECTED = 'rejected';  
  
  
function resolvePromise(result,promise2,resolve,reject){  
 //定义一个标识,resolve或reject只能调用一次  
 let called = false;  
  
 if(result === promise2){  
 return reject(new TypeError('类型错误'))  
 }  
 //判断result是不是promise，如果是promise则解析promise  
 //不是则直接resolve  
 if((typeof result === 'object') && (typeof result !== null) || (typeof result === 'function')){  
 //获取result.then的时候可能报错  
 try{  
 //如果是object,则以是否有then方法来区分是不是promise  
 const then = result.then;  
 if(typeof then === 'function'){  
 //是promise，解析promise  
 then.call(result,  
 result1=>{  
 if(called) return;  
 called = true;  
 resolvePromise(result1,promise2,resolve,reject)  
 },no=>{  
 if(called) return;  
 called = true;  
 reject(no)  
 })  
 }else{  
 if(called) return;  
 called = true;  
 //result.then方法不是函数,则说明result不是一个promise  
 resolve(result)  
 }  
 }catch(e){  
 if(called) return;  
 called = true;  
 reject(e)  
 }  
 }else{  
 if(called) return;  
 called = true;  
 resolve(result)  
 }  
}  
  
class Promise {  
 constructor(executor){  
 this.status = PENDING;  
 this.value = undefined;  
 this.reason = undefined;  
 // 成功或失败处理函数的数组。为什么是数组？因为一个promise可以调多次.then。  
 this.onFulfilledCallbacks = [];  
 this.onRejectedCallbacks = [];  
 //传入的函数executor是同步执行的  
 const resolve = (value) => {  
 if(this.status === PENDING){  
 this.status = FULFILLED;  
 this.value = value;  
 this.onFulfilledCallbacks.forEach(onFulfilled=>{  
 queueMicrotask(()=>{  
 onFulfilled()  
 })  
 })  
 }  
 }  
 const reject = (reason) => {  
 if(this.status === PENDING){  
 this.status = REJECTED;  
 this.reason = reason;  
 this.onRejectedCallbacks.forEach(onRejected=>{  
 queueMicrotask(()=>{  
 onRejected()  
 })  
 })  
 }  
 }  
  
 try{  
 executor(resolve,reject);  
 }catch(e){  
 reject(e)  
 }  
  
 }  
  
 then(onFulfilled,onRejected){  
 //实现then()的穿透  
 onFulfilled = typeof onFulfilled === 'function'? onFulfilled : value=>value  
 onRejected = typeof onRejected === 'function'? onRejected : error=>{throw error}  
  
  
 const promise2 = new Promise((resolve,reject)=>{  
 if(this.status === FULFILLED){  
 //promise A+不允许立即执行，需放到异步任务队列  
 //使用浏览器提供的微任务方法queueMicrotask  
 queueMicrotask(()=>{  
 //处理函数的返回值  
 try{  
 const result = onFulfilled(this.value)  
 resolvePromise(result,promise2,resolve,reject)  
 }catch(e){  
 console.log('捕获到')  
 reject(e)  
 }  
 })  
 }  
   
 if(this.status === REJECTED){  
 queueMicrotask(()=>{  
 try{  
 const result = onRejected(this.reason)  
 resolvePromise(result,promise2,resolve,reject)  
 }catch(e){  
 reject(e)  
 }  
 })  
 }  
   
 if(this.status === PENDING){  
 this.onFulfilledCallbacks.push(()=>{  
 try{  
 const result = onFulfilled(this.value)  
 resolvePromise(result,promise2,resolve,reject)  
 }catch(e){  
 reject(e)  
 }  
 });  
 this.onRejectedCallbacks.push(()=>{  
 try{  
 const result = onRejected(this.reason)  
 resolvePromise(result,promise2,resolve,reject)   
 }catch(e){  
 reject(e)  
 }   
 });  
 }  
 })  
  
 return promise2;  
 }  
}

### 12、process.nextTick()和setImmediate( )

他们只能在node环境执行。

代码执行顺序

1、同步

2、process.nextTick(会在当前事件循环的最后执行)

3、异步（先执行微任务、再执行宏任务）

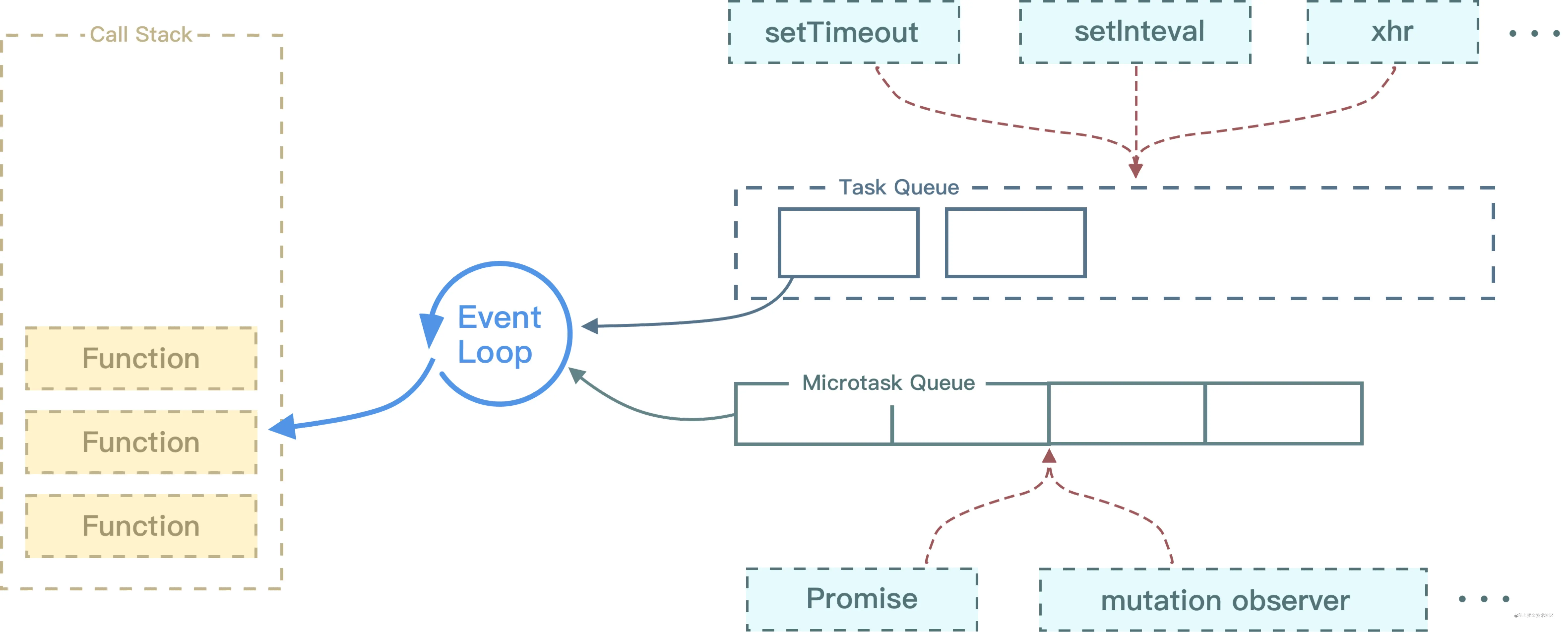
4、setImmediate

setImmediate(()=>{  
 console.log(0)  
})  
console.log(1);  
setTimeout(()=>{  
console.log(2)  
})  
new Promise((res,rej)=>{  
 res(666)  
}).then(res=>{  
 console.log(res)  
})  
process.nextTick(()=>{  
 console.log(3)  
});  
setTimeout(()=>{  
console.log(4)  
})  
console.log(5);  
  
  
//1  
//5  
//3  
//2  
//4  
//0

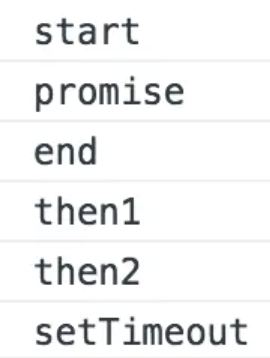
### 13、宏任务与微任务

宏任务：宿主发起的任务（计时器、ajax、读取文件）

微任务：Javascript引擎发起的任务（promise.then）



console.log('start')  
setTimeout(() => {  
 console.log('setTimeout')  
}, 0)  
  
new Promise(resolve => {  
 console.log('promise')  
 resolve()  
}).then(() => {  
 console.log('then1')  
}).then(() => {  
 console.log('then2');  
})  
  
console.log('end')



动画解释：<https://zhuanlan.zhihu.com/p/377340044>

参考文章：<https://juejin.cn/post/6844904030678810632>

宏任务和微任务队列中新产生的微任务都会追加到**当前**微任务队列队尾等待执行，而不是追加到下一个循环的微任务队列中。因此如果微任务队列的清空过程中持续产生新的微任务，会造成微任务卡死。

Event Loop会不断循环的去取tasks队列的中最老的一个task(可以理解为宏任务）推入栈中执行，并在当次循环里依次执行并清空microtask队列里的任务。

即eventloop每一次循环的操作：

eventloop开始

1、去宏任务队列取下一个宏任务（currentTask）进栈执行

2、期间产生的宏任务进入宏任务队列，产生的微任务进入微任务队列

3、当前宏任务（currentTask）执行结束

4、执行微任务队列，若此过程中产生新的微任务，则追加到微任务队列队尾等待执行。

5、微任务队列清空

6、本次eventloop结束

eventloop轮询过程：<https://blog.csdn.net/yiersan__/article/details/107819888>

讲故事：<https://www.cnblogs.com/jiasm/p/9482443.html>