期约与异步函数

异步编程

不想为等待某个异步操作而阻塞线程执行

同步和异步

同步行为对应内存中顺序执行的处理器指令，每条指令都会严格按照它们出现的顺序来执行

异步行为类似系统中断，等待系统外部实体触发代码执行

以往的异步编程模式

早期，只支持定义回调函数来表明异步操作完成

- 异步返回值
  给异步操作一个回调，回调中包含使用异步返回值的代码

```
function double(value, callback) {
    setTimeout(() => callback(value * 2), 1000);
}
double(3, (x) => console.log(`I was given: ${x}`));
   // I was given: 6（大约 1000 毫秒之后）
```

- 失败处理
  成功回调，失败回调
  不可取
  初始化异步时定义回调。异步函数的返回值只在短时间内存在，只有预备好将这个短时间内存在的值作为参数的回调才能接收到它

```
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) => console.log(`Success: ${x}`);
const failureCallback = (e) => console.log(`Failure: ${e}`);
double(3, successCallback, failureCallback);
double('b', successCallback, failureCallback);
```

```
    // Success: 6（大约 1000 毫秒之后）
    // Failure: Must provide number as first argument（大约 1000 毫秒之后）
```

- 嵌套异步回调
  异步返回值依赖另一个异步返回值
  回调地狱

期约

Promise
期约是对尚不存在结果的一个替身

Promises/A+规范

时间	载体	期约机制
早期	jQuery、Dojo	Deferred API
2010	CommonJS	Promises/A规范
2012	Promises/A+组织	Promises/A+规范

ES6 增加对Promises/A+规范的完善支持，Promise类型

期约基础

ES6新增 引用类型 Promise
new 操作符实例化
创建时需要传入执行器（executor）函数作为参数

- 期约状态机
  - 待定 pending
  - 兑现 fulfilled，有时也称解决resolved
  - 拒绝 rejected
    最初状态pending，可以落定成resolved，rejected，不可逆。
    状态是私有的，不能直接通过JavaScript检测到。主要是为了避免读取到期约的状态，以同步方式处理期约。
    状态不可被外部修改
    期约状态

- 解决值，拒绝理由，期约用例
  解决值，拒绝理由默认undefined
  期约一落定，执行异步代码就会收到

- 通过执行函数控制期约状态
  执行函数 内部调用reject(),resolve()
  期约状态只能修改一次

- Promise.resolve()
  包装任何非期约值，包括错误对象

- Promise.reject()
  包装任何非期约值，然后抛出异步错误
  异步错误：不能用try/catch捕获，只能拒绝处理程序捕获 .catch
  两者等价

```
// 写法一
const promise = new Promise(function (resolve, reject) {
    try {
        throw new Error('test');
    } catch (e) {
        reject(e);
    }
});
promise.catch(function (error) {
    console.log(error);
});

// 写法二
const promise = new Promise(function (resolve, reject) {
    reject(new Error('test'));
});
promise.catch(function (error) {
    console.log(error);
});
```

- 同步/异步执行的二元性

```
try {
 throw new Error('foo');
} catch(e) {
 console.log(e); // Error: foo
}
try {
 Promise.reject(new Error('bar'));
} catch(e) {
 console.log(e);
}
// Uncaught (in promise) Error: bar
```

为什么要有Promise.resolve() Promise.reject()?
重要作用是将一个其他实现的Promise对象封装成一个当前实现的Promise对象

期约的实例方法

- 实现Thenable接口
  ECMAScript 暴露的异步结构中，任何对象都有一个 then()方法。

- Promise.prototype.then()
  为Promise实例添加处理程序的主要方法
  两个参数，onResolved处理程序，onRejected处理程序，可选

- Promise.prototype.catch()
  语法糖
  等价于Prmise.prototype.then(null,onRejected)

- Promise.prototype.finally()
  无法得知状态，一般用作清理代码

- 非重入期约方法
  进入落定状态时，处理程序并非立即执行，而是被排期
  当前线程上的同步代码执行完成后才会执行。
  非重入 特性

```
// 创建解决的期约
let p = Promise.resolve();
// 添加解决处理程序
// 直觉上，这个处理程序会等期约一解决就执行
p.then(() => console.log('onResolved handler'));
// 同步输出，证明 then()已经返回
console.log('then() returns');
// 实际的输出：
// then() returns
// onResolved handler
```

- 邻近处理程序的执行顺序
  多个处理程序，Promise状态同时变化，按照添加顺序执行

- 传递解决值和拒绝理由
  resolve() reject()可以传参；时onResolved() onRejected()的参数

- 拒绝期约和拒绝错误处理

```
let p1 = new Promise((resolve, reject) => reject(Error('foo')));
let p2 = new Promise((resolve, reject) => { throw Error('foo'); });
let p3 = Promise.resolve().then(() => { throw Error('foo'); });
let p4 = Promise.reject(Error('foo'));
setTimeout(console.log, 0, p1); // Promise <rejected>: Error: foo
setTimeout(console.log, 0, p2); // Promise <rejected>: Error: foo
setTimeout(console.log, 0, p3); // Promise <rejected>: Error: foo
setTimeout(console.log, 0, p4); // Promise <rejected>: Error: foo
```

  Promise内throw抛出异常是 异步错误
  异步错误无法用try...catch捕获需要用.catch捕获

```
new Promise((resolve, reject) => {
 console.log('begin asynchronous execution');
 reject(Error('bar'));
}).catch((e) => {
 console.log('caught error', e);
}).then(() => {
 console.log('continue asynchronous execution');
});
// begin asynchronous execution
```

```
// caught error Error: bar
// continue asynchronous execution
```

期约连锁与期约合成

期约连锁

把Promise逐个串联起来
每个Promise实例方法(then(),catch(),finally())都会返回新的Promise对象

```
let p1 = new Promise((resolve, reject) => {
    console.log('p1 executor');
    setTimeout(resolve, 1000);
});
p1.then(() => new Promise((resolve, reject) => {
    console.log('p2 executor');
    setTimeout(resolve, 1000);
}))
    .then(() => new Promise((resolve, reject) => {
        console.log('p3 executor');
        setTimeout(resolve, 1000);
    }))
    .then(() => new Promise((resolve, reject) => {
        console.log('p4 executor');
        setTimeout(resolve, 1000);
    }));
```

解决了回调地狱

期约图

一个Promise可以有任意个处理程序，所以可以构建 有向非循环图 结构

```
//    A
//   / \
//   B C
//  /\ /\
// D E F G
let A = new Promise((resolve, reject) => {
    console.log('A');
    resolve();
});
let B = A.then(() => console.log('B'));
let C = A.then(() => console.log('C'));
B.then(() => console.log('D'));
B.then(() => console.log('E'));
C.then(() => console.log('F'));
C.then(() => console.log('G'));
// A
// B
// C
// D
// E
```

```
// F
// G
```

Promise.all() 和 Promise.race()

多个Promise实例组合成一个Promise

- Promise.all() 一组Promise解决后再解决
  参数：一个可迭代对象
  返回值：新的Promise实例

  ```
  let p1 = Promise.all([
   Promise.resolve(),
   Promise.resolve()
  ]);
  // 可迭代对象中的元素会通过 Promise.resolve()转换为期约
  let p2 = Promise.all([3, 4]);
  // 空的可迭代对象等价于 Promise.resolve()
  let p3 = Promise.all([]);
  // 无效的语法
  let p4 = Promise.all();
  // TypeError: cannot read Symbol.iterator of undefined
  ```

  - 全部落定才落定
  - 全部成功才成功
  - 成功按照迭代器顺序 把解决值合成一个数组
  - 失败 只接收第一个失败的，其他失败全部静默处理

- Promise.race()
  参数：一个可迭代对象
  返回值：新的Promise实例
  把第一个落定的Promise包装，成功就成功，失败就失败，重点是第一个落定

串行期约合成

通过Array.prototype.reduce()把promise串行起来
Array.prototype.reduce(prev,curr)
prve:这里指的不是上一项，而是上一项的结果
curr：当前项

```
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
```

期约拓展

很多第三方库实现具备ES规范，但是未涉及的两个特性，例如Bluebird，Q

期约取消

Promise正在处理中，但是不需要结果了
取消令牌 cancel token
类包装Promise，解决方法暴露给cancelFn参数。
外部向构造函数传如一个函数，控制什么情况下取消期约(resolve)

```
class CancelToken {
    constructor(cancelFn) {
        this.promise = new Promise((resolve, reject) => {
            cancelFn(resolve);
        });
    }
}
```

实例

```
<button id="start">Start</button>
<button id="cancel">Cancel</button>
<script>
// 定义CancelToken，接收一个函数参数，函数取消时被调用
class CancelToken {
 constructor(cancelFn) {
 this.promise = new Promise((resolve, reject) => {
 cancelFn(() => {
 setTimeout(console.log, 0, "delay cancelled");
 resolve();
 });
 });
 }
}
const startButton = document.querySelector('#start');
const cancelButton = document.querySelector('#cancel');
function cancellableDelayedResolve(delay) {
 setTimeout(console.log, 0, "set delay");
 return new Promise((resolve, reject) => {
 const id = setTimeout((() => {
 setTimeout(console.log, 0, "delayed resolve");
 resolve();
 }), delay);
 // 设置可取消
 const cancelToken = new CancelToken((cancelCallback) =>
 cancelButton.addEventListener("click", cancelCallback));
 // 如果取消，执行某些操作来实现取消
 cancelToken.promise.then(() => clearTimeout(id));
 });
}
startButton.addEventListener("click", () => cancellableDelayedResolve(1000));
</script>
```

用例是通过清空Timeout来实现取消

期约进度通知

ES6 Promise不支持进度追踪，可以通过拓展实现

```
class TrackablePromise extends Promise {
    constructor(executor) {
        const notifyHandlers = [];
        super((resolve, reject) => {
            return executor(resolve, reject, (status) => {
                notifyHandlers.map((handler) => handler(status));
            });
        });
        this.notifyHandlers = notifyHandlers;
    }
    notify(notifyHandler) {
        this.notifyHandlers.push(notifyHandler);
        return this;
    }
}
```

实例化

```
let p = new TrackablePromise((resolve, reject, notify) => {
 function countdown(x) {
 if (x > 0) {
 notify(`${20 * x}% remaining`);
 setTimeout(() => countdown(x - 1), 1000);
 } else {
 resolve();
 }
 }
 countdown(5);
});

p.notify((x) => setTimeout(console.log, 0, 'progress:', x));
p.then(() => setTimeout(console.log, 0, 'completed'));
// （约 1 秒后）80% remaining
// （约 2 秒后）60% remaining
// （约 3 秒后）40% remaining
// （约 4 秒后）20% remaining
// （约 5 秒后）completed
```