# 异步最佳实践

### promise

Promise 对象用于表示一个异步操作的最终状态（完成或失败），Promise在回调代码和将要执行这个任务的异步代码之间提供了一种可靠的中间机制来管理回调。

本质上Promise就是一个构造函数。

Promise的实例对象有三个状态

pending: 初始状态，既不是成功，也不是失败状态。

fulfilled: 意味着操作成功完成。

rejected: 意味着操作失败

* Promise语法

new Promise( function(resolve, reject) {...}); //第一个参数一般我们称为executor

//实例

var promise = new Promise(function(resolve, reject) {

setTimeout(resolve, 100, 'foo');

});

* executor

executor是带有 resolve 和 reject 两个参数的函数 。Promise构造函数执行时立即调用executor 函数，在实例对象被返回前。resolve 和 reject 两个函数作为参数传递给executor。resolve 和 reject 函数被调用时，分别将promise的状态改为fulfilled（完成）或rejected（失败）。executor 内部通常会执行一些异步操作，一旦完成，可以调用resolve函数来将promise状态改成fulfilled，或者在发生错误时调用reject函数将它的状态改为rejected。如果在executor函数中抛出一个错误，那么该promise 状态为rejected。executor函数的返回值被忽略。

* promise状态机制

executor的resolve参数被调用时会将promise的状态从pending改为fulfilled

executor的reject参数被调用时会将promise的状态从pending改为rejected

如果在executor函数中抛出一个错误，那么该promise 状态为rejected

* Promise.prototype.then

当promise对象的状态发生改变时，绑定在其身上的then方法就会被调用。

then方法包含两个参数：onfulfilled函数 和 onrejected函数，它们都是 Function 类型。当Promise状态为fulfilled时，调用 then 的 onfulfilled 方法，当Promise状态为rejected时，调用 then 的 onrejected 方法， 所以在异步操作的完成和绑定处理方法之间不存在竞争

then() 方法返回一个 Promise 。

* 语法

p.then(onFulfilled, onRejected);

p.then(function(value) {// fulfillment}, function(reason) {// rejection});

* 参数

onFulfilled：

当Promise变成接受状态时，该参数作为回调函数被调用。该函数有一个参数，即接受的最终结果。由executor中resolve的第一个参数提供

onRejected

当Promise变成拒绝状态（rejection ）时，该参数作为回调函数被调用该函数有一个参数,，即拒绝的原因。由executor中reject的第一个参数提供。或系统提供的报错信息。

* 返回值

then方法返回一个新的Promise，而它的行为与then中的回调函数的返回值有关：

如果then中的回调函数返回一个值，那么then返回的Promise将会成为接受状态，并且将返回的值作为接受状态的回调函数的参数值。

如果then中的回调函数抛出一个错误，那么then返回的Promise将会成为拒绝状态，并且将抛出的错误作为拒绝状态的回调函数的参数值。

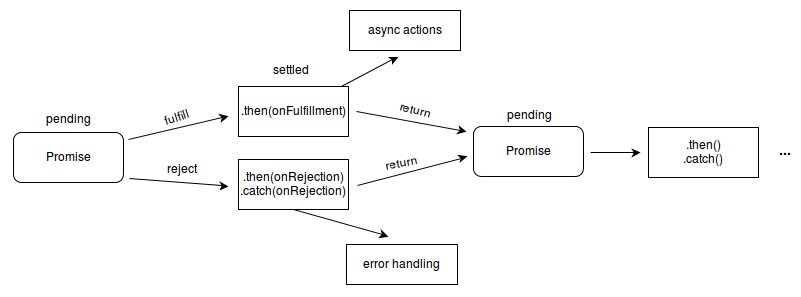
如果then中的回调函数返回一个已经是接受状态的Promise，那么then返回的Promise也会成为接受状态，并且将那个Promise的接受状态的回调函数的参数值作为该被返回的Promise的接受状态回调函数的参数值。

如果then中的回调函数返回一个已经是拒绝状态的Promise，那么then返回的Promise也会成为拒绝状态，并且将那个Promise的拒绝状态的回调函数的参数值作为该被返回的Promise的拒绝状态回调函数的参数值。

如果then中的回调函数返回一个未定状态（pending）的Promise，那么then返回Promise的状态也是未定的，并且它的终态与那个Promise的终态相同；同时，它变为终态时调用的回调函数参数与那个Promise变为终态时的回调函数的参数是相同的。

* Promise链式调用

Promise.prototype.then方法反回的也是一个promise对象。所以我们可以进行链式调用



### generator

* Symbol

ES6引入了一种新的原始数据类型Symbol，表示独一无二的值。它是JavaScript 语言的第七种数据类型

undefined、null、Boolean、String、Number、Symbol、Object

该类型的性质在于这个类型的值可以用来创建匿名的对象属性。该数据类型的值通常被用作一个对象属性的键, symbol 类型的键存在于各种内置的 JavaScript 对象中

在 JavaScript 运行时环境中，symbol类型的值可以通过调用函数 Symbol() 创建。这个函数动态地生成了一个匿名，唯一的值。

* 内置函数Symbol()

Symbol()函数会返回symbol类型的值，该类型具有静态属性和静态方法。它的静态属性会暴露几个内建的成员对象；它的静态方法会暴露全局的symbol注册

但一定要注意了，Symbol不是一个构造器，它并不能被new关键字调用。

每个从Symbol()返回的symbol值都是唯一的。一个symbol值能作为对象属性的标识符；这是该数据类型仅有的目的

* Symbol作为属性名

**Symbol类型唯一合理的用法是用变量存储 symbol的值，然后使用存储的值创建对象属性。以下示例使用"var"创建一个变量来保存 symbol**

//第一种写法

let mySymbol = Symbol();

let a = {};

a[mySymbol] = 'Hello!';

//第二种写法

let a = {

[mySymbol]: 'Hello!'

};

//第三种写法

let a = {};

Object.defineProperty(a, mySymbol, { value: 'Hello!' });

//以上写法都得到同样结果

a[mySymbol] // "Hello!"

symbol属性是匿名的；并且是不可枚举的.不会在循环结构 for-in 中出现。也因为这个属性是匿名的，它同样不会出现在Object.getOwnPropertyNames(obj)的返回数组里。

这个属性可以通过创建时的原始 symbol 值访问到，或者通过遍历 “Object.getOwnPropertySymbols()” 返回的数组。

* Symbol.for

Symbol.for(key) 方法会根据给定的键 key，来从运行时的 symbol 注册表中找到对应的 symbol，如果找到了，则返回它，否则，新建一个与该键关联的 symbol，并放入全局 symbol 注册表中

* Symbol.keyFor

Symbol.keyFor(sym) 方法用来获取注册表中与某个 symbol 关联的键。

sym：存储在 symbol 注册表中的某个 symbol

* Symbol.iterator

Symbol.iterator 为一些对象定义了默认的迭代器。该迭代器可以被 for...of 循环使用

本质上Symbol.iterator是一个属性名（Symbol类型的属性）,对应的属性值是一个函数（迭代器）。函数调用后的返回值是一个迭代器对象！

* Iterator

遍历器（Iterator）是一种接口，为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署 Iterator 接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）

Iterator 的作用有三个：

一是为各种数据结构，提供一个统一的、简便的访问接口；

二是使得数据结构的成员能够按某种次序排列；

三是 ES6 创造了一种新的遍历命令for...of循环，Iterator 接口主要供for...of消费

* 迭代器协议

当一个对象被认为是一个迭代器时，它实现了一个 next() 的方法并且拥有以下含义：

迭代器的next方法会返回一个迭代对象，它具有两个属性

done (boolean)：

迭代是否结束

value：

当前迭代的值

* iterable

对象进行for...of循环时，会调用Symbol.iterator方法（迭代器），返回该对象的默认迭代器对象。一种数据结构只要部署了 Iterator 接口，我们就称这种数据结构是“可遍历的”（iterable）。就可以用for...of循环遍历它的成员。也就是说，for...of循环内部调用的是数据结构的Symbol.iterator方法

原生具备 Iterator 接口的数据结构如下。

Array

String

函数的 arguments 对象

NodeList 对象

Map

Set

* for of循环

for...of语句在可迭代对象（包括 Array，Map，Set，String，TypedArray，arguments 对象等等）上创建一个迭代循环，调用自定义迭代钩子，并为每个不同属性的值执行语句

* generator函数

function\* 这种声明方式(function关键字后跟一个星号）会定义一个函数 (生成器)，调用它后会返回一个生成器对象。 生成器对象符合可迭代协议

第一次调用生成器对象的next方法时, generator函数才会被执行

* yield

yield关键字使生成器函数执行暂停。yield关键字会和一个迭代对象挂钩，迭代对象有两个属性，value和done。value属性是对yield表达式求值的结果，而done是false，表示生成器函数尚未完全完成。yield关键字后面的表达式的值 返回给 生成器对象的next方法对应的value。

**next方法对应的value是谁，取决于next方法执行后在哪个yield表达式上暂停**

每次调用生成器的next()方法时，生成器都会恢复执行,如果将可选值传递给生成器的next()方法，则该值将成为生成器当前yield启动时返回的值。

**next方法对应的参数给谁，取决于next方法执行后在哪个yield表达式上启动**

### async&await

正常函数前加一个async关键字后，该函数就被定义成了一个async函数。

当调用一个 async 函数时，会返回一个 Promise 对象。

当这个 async 函数返回一个值时，Promise 的 resolve 方法会负责传递这个值；

当 async 函数抛出异常时，Promise 的 reject 方法也会传递这个异常值。

* await

await 操作符用于等待一个Promise 对象。它只能在async function 中使用。

await 表达式会暂停当前 async function 的执行，等待 Promise 处理完成。

若 Promise 正常处理(fulfilled)，!!其回调的resolve函数参数作为 await 表达式的值!!，继续执行 async function。

若 Promise 处理异常(rejected)，await 表达式会把 Promise 的异常原因抛出。