02 异步编程发展历程之Promise的演进史

介绍

JavaScript是一门典型的异步编程脚本语言，在编程过程中会大量的出现异步代码的编写，在JS的整个发展历程中，对异步编程的处理方式经历了很多个时代，其中最典型也是现今使用最广泛的时代，就是Promise对象处理异步编程的时代。那么什么是Promise对象呢?

Promise是ES6版本提案中实现的异步处理方式，对象代表了未来将要发生的事件，用来传递异步操作的消息。

为什么使用Promise对象

举个栗子：

在过去的编程中JavaScript的主要异步处理方式，是采用回调函数的方式来进行处理，想要保证n个步骤的异步编程有序进行，会出现如下的代码（以setTimeout为栗子）

```
setTimeout(function(){
  //第一秒后执行的逻辑
  console.log('第一秒之后发生的事情')
  setTimeout(function(){
    //第二秒后执行的逻辑
    console.log('第二秒之后发生的事情')
    setTimeout(function(){
      //第三秒后执行的逻辑
      console.log('第三秒之后发生的事情')
    },1000)
  },1000)
},1000)
```

参考上面的代码，如果分3秒每间隔1秒运行1个任务，这三个任务必须按时间顺序执行，并且每个下一秒触发前都要先拿到上一秒运行的结果，那么我们不得不将代码编写为以上案例代码。该写法主要是为了保证代码的严格顺序要求，这样就避免不了大量的逻辑在回调函数中不停的进行嵌套，这也是我们经常听说的“回调地狱”。

再举个栗子：

在编程中setTimeout的“栗子”其实使用场景极少，在前端开发过程中使用最多的异步流程就是AJAX请求，当系统中要求某个页面的n个接口保证有序调用的情况下就会出现下面的情况：

```
//获取类型数据
$.ajax({
  url:'/***',
  success:function(res){
    var xxId = res.id
    //获取该类型的数据集合，必须等待回调执行才能进行下一步
    $.ajax({
      url:'/***',
```

```
      data:{
        xxId:xxId,//使用上一个请求结果作为参数调用下一个接口
      },
      success:function(res1){
        //得到指定类型集合
        ...
      }
    })
  }
})
```

这种情况在很多人的代码中都出现过，如果流程复杂化，在网络请求中继续夹杂其他的异步流程，那么这样的代码就会变得难以维护了。

其他的“栗子”诸如Node中的原始fs模块，操作文件系统这种就不举了，所以之所以在ECMA提案中出现Promise解决方案，就是因为此类代码导致了JS在开发过程中遇到的实际问题：【回调地狱】。其实解决回调地狱的方式还有其他方案，本节我们不多做介绍中间的过渡方案，核心介绍Promise流程控制对象，因为他是解决回调地狱的非常好的方案。

使用Promise如何解决异步控制问题

前面的章节仅仅是抛出了问题，并没有针对问题作出一个合理的回答。下面阐述一下如何使用Promise对象解决回调地狱问题。

在阐述之前，我们先对Promise做一个简单的介绍：Promise对象的主要用途是通过链式调用的结构，将原本回调嵌套的异步处理流程，转化成“对象.then().then()...”的链式结构，这样虽然仍离不开回调函数，但是将原本的回调嵌套结构，转化成了连续调用的结构，这样就可以在阅读上编程上下左右结构的异步执行流程了。

接下来看一段代码，还是以setTimout为“栗子”我们改造第一个案例：

```
//使用Promise拆解的setTimeout流程控制
var p = new Promise(function(resolve){
  setTimeout(function(){
    resolve()
  },1000)
})
p.then(function(){
  //第一秒后执行的逻辑
  console.log('第一秒之后发生的事情')
  return new Promise(function(resolve){
    setTimeout(function(){
      resolve()
    },1000)
  })
}).then(function(){
  //第二秒后执行的逻辑
  console.log('第二秒之后发生的事情')
  return new Promise(function(resolve){
    setTimeout(function(){
      resolve()
```

```
    },1000)
  })
}).then(function(){
  //第三秒后执行的逻辑
  console.log('第三秒之后发生的事情')
})
```

结合代码案例我们发现使用了Promise后的代码，将原来的3个setTimeout的回调嵌套，拆解成了三次then包裹的回调函数，按照上下顺序进行编写。这样我们从视觉上就可以按照人类的从上到下从左到右的线性思维来阅读代码，这样很容易能查看这段代码的执行流程，代价是代码的编写量增加了接近1倍。

Promise介绍

从上面的案例介绍得知Promise的作用是解决“回调地狱”，他的解决方式是将回调嵌套拆成链式调用，这样便可以按照上下顺序来进行异步代码的流程控制。那么Promise是如何实现这个能力的呢?

Promise的结构

Promise对象是一个JavaScript对象，在支持ES6语法的运行环境中作为全局对象提供，他的初始化方式如下：

```
//fn:是初始化过程中调用的函数他是同步的回调函数
var p = new Promise(fn)
```

关于回调函数

这里涉及到一个概念：JavaScript语言中，有一个特殊的函数叫做回调函数。回调函数的特点是把函数作为变量看待，由于JavaScript变量可以作为函数的形参并且函数可以通过声明变量的方式匿名创建，所以我们可以在定义函数时将一个函数的参数当作函数来执行，进而在调用时在参数的位置编写一个执行函数，代码如下：

```
//把fn当作函数对象那么就可以在test函数中使用()执行他
function test(fn){
  fn()
}
//那么运行test的时候fn也会随着执行，所以test中传入的匿名函数就会运行
test(function(){
  ...
})
```

上面的代码结构，就是JavaScript中典型的回调函数结构。按照我们在事件循环中介绍的JavaScript函数运行机制，会发现**其实回调函数本身是同步代码**，这是一个需要【重点理解】的知识点。

通常在编写JavaScript代码时，使用的回调嵌套的形式大多是异步函数，所以一些开发者可能会下意识的认为，凡是回调形式的函数都是异步流程。其实并不是这样的，真正的解释是（敲黑板）：**JavaScript中的回调函数结构，默认是同步的结构，由于JavaScript单线程异步模型的规则，如果想要编写异步的代码，必须使用回调嵌套的形式才能实现，所以回调函数结构不一定是异步代码，但是异步代码一定是回调函数结构。**

那么为什么异步流程都需要回调函数？依然举个栗子：

分析下列代码的输出顺序（先不要看答案）：

```
function test(fn){
  fn()
}
console.log(1)
test(function(){
  console.log(2)
})
console.log(3)
//这段代码的输出顺序应该是*、*、*，因为他属于直接进入执行栈的程序，会按照正常程序解析的流程输出
```

分析下列代码的输出顺序（先不要看答案）：

```
function test(fn){
  setTimeout(fn,0)
}
console.log(1)
test(function(){
  console.log(2)
})
console.log(3)
//这段代码会输出*、*、*，因为在调用test的时候settimeout将fn放到了异步任务队列挂起了，等待主程序执行完毕之后才会执行
```

再思考一下，如果我们有一个变量a的值为0，想要1秒之后设置他的值为1，并且我们想要在之后得到a的新结果，这个逻辑中如果1秒之后设置a为1采用的是setTimeout，那么我们在同步结构里能否实现？

先参考下面的案例：

```
var a = 0
setTimeout(function(){
  a = 1
},1000)
console.log(a)
```

> 解析

上述代码块输出的结果一定是0，由于JavaScript单线程异步模型的知识，我们可以得知，当前的代码块中setTimeout的回调函数是一个宏任务，会在本次的同步代码执行完毕后执行，所以声明a=0和输出a的值这两行代码会优先执行，这时对a设置1的事件还没有发生，所以输出的结果就一定为0。

接下来对代码做如下改造，我们试图使用阻塞的方式来获取异步代码的结果：

```
var a = 0
//依然使用setTimeout设置1秒的延迟设置a的值
setTimeout(function(){
  a = 1
},1000)
var d = new Date().getTime()
var d1 = new Date().getTime()
//采用while循环配合时间差来阻塞同步代码2秒
while(d1-d<2000){
  d1 = new Date().getTime()
}
console.log(a)
```

> 解析

本案例的同步代码会在while循环中阻塞2秒，所以console.log(a)这行代码会在2秒之后才能获得执行资源，但是最终输出的结果仍然是0。这是为什么呢？这里仍然可以通过JavaScript的运行模型来进行理解，由于单线程异步模型的规则是严格的同步在前异步靠后顺序，本案例的同步代码虽然阻塞了2秒，已经超过了setTimeout的等待时间，但是setTimeout中的宏任务到时间后，仅仅会被从工作线程移动到任务队列中进行等待。在时间到达1秒时，while循环没有执行结束，所以函数执行栈会被继续占用，直到循环释放并输出a之后，任务队列中的宏任务才能执行，所以这里就算setTimeout时间到了，也必须等待同步代码执行完毕，那么输出a的时候a=1的事件仍然没有发生，**所以我们采用默认的上下结构永远拿不到异步回调中的结果，这也是为什么异步流程都是回调函数的原因。**

所以想要真正的在2秒后获取a的新结果的代码结构是这样的：

```
//我们只有在这个回调函数中才能获取到a改造之后的结果
var a = 0
setTimeout(function(){
  a = 1
},1000)
//注册一个新的宏任务，让他在上一个宏任务后执行
setTimeout(function(){
  console.log(a)
},2000)
```

到这里我们大概明白了回调函数的意义以及使用场景了，那么我们的Promise对象完整的结构是接下来案例中的样子，并且他是一个及特殊的存在，Promise中既包含同步的回调函数，又包含异步的回调函数。

Promise案例介绍

运行该案例并查看解析：

```
//实例化一个Promise对象
var p = new Promise(function(resolve,reject){

})
//通过链式调用控制流程
p.then(function(){
  console.log('then执行')
}).catch(function(){
  console.log('catch执行')
}).finally(function(){
  console.log('finally执行')
})
```

参考上面的Promise对象结构，一个Promise对象包含两部分回调函数，第一部分是new Promise时候传入的对象，这段回调函数是同步的，而.then.catch.finally中的回调函数是异步的，这里我们提前记好。接下来可以在html页面中跑一遍程序，会发现这段程序并没有任何输出，然后我们可以将程序继续改造。

练习案例：猜输出顺序

```
console.log('起步')
var p = new Promise(function(resolve,reject){
  console.log('调用resolve')
  resolve('执行了resolve')
})
p.then(function(res){
  console.log(res)
  console.log('then执行')
}).catch(function(){
  console.log('catch执行')
}).finally(function(){
  console.log('finally执行')
})
console.log('结束')
```

将这段程序运行一下会发现输出顺序为:

起步->调用resolve->结束->执行了resolve->then执行->finally执行

再看下面的代码:

```
console.log('起步')
var p = new Promise(function(resolve,reject){
  console.log('调用reject')
  reject('执行了reject')
})
p.then(function(res){
  console.log(res)
  console.log('then执行')
}).catch(function(res){
```

```
  console.log(res)
  console.log('catch执行')
}).finally(function(){
  console.log('finally执行')
})
console.log('结束')
```

将这段程序运行一下会发现输出顺序为:

起步->调用reject->结束->执行了reject->catch执行->finally执行

解读Promise结构

经过了上面的代码我们可以分析一下Promise的运行流程和结构，首先从运行流程上我们发现了new Promise中的回调函数确实是在同步任务中执行的，其次是如果这个回调函数内部没有执行resolve或者reject那么p对象的后面的回调函数内部都不会有输出，而运行resolve函数之后.then和.finally就会执行，运行了reject之后.catch和.finally就会执行。

剖析对象结构

Pomise对象相当于一个未知状态的对象，他的定义就是声明一个等待未来结果的对象，在结果发生之前他一直是初始状态，在结果发生之后他会变成其中一种目标状态，它的名字Promise中文翻译为保证。很多国外的电影台词都会涉及到Promsie这个单词，比如小明发现了邻居张三的妻子出轨了，在某天喝酒的时候小明和张三说：I saw your wife played with other man! I promise ! I saw it！张三当然会说：No！shit！I can not trust you! dame！（语义自行理解），**Promise在英文中是绝对保证的意思，所以在编程中Promise对象是一个非常严谨的对象，一定会按照约定执行，不会出现任务灵异问题（除使用不当外）**。

那么Promise本身具备三种状态：

- **pending**：初始状态，也叫就绪状态，这是在Promise对象定义初期的状态，这时Promise仅仅做了初始化并注册了他对象上所有的任务。
- **fulfilled**：已完成，通常代表成功执行了某一个任务，当初始化函数中的resolve执行时，Promise的状态就变更为fulfilled，并且then函数注册的回调函数会开始执行，resolve中传递的参数会进入回调函数作为形参。
- **rejected**：已拒绝，通常代表执行了一次失败任务，或者流程中断，当调用reject函数时，catch注册的回调函数就会触发，并且reject中传递的内容会变成回调函数的形参。

三种状态之间的关系：

Promise中约定，当对象创建之后同一个Promise对象只能从pending状态变更为fulfilled或rejected中的其中一种，并且状态一旦变更就不会再改变，此时Promise对象的流程执行完成并且finally函数执行。

根据上面的分析，结合下面的代码案例学习Promise的规则，分析该对象的运行结果：

```
new Promise(function(resolve,reject){
  resolve()
  reject()
}).then(function(){
  console.log('then执行')
}).catch(function(){
  console.log('catch执行')
}).finally(function(){
  console.log('finally执行')
})
```

通过分析以上的说明我们知道了Promise对象存在三种状态以及他们之间的关系，那么我们在执行本段程序的时候会发现这个段代码的输出结果是:

then执行->finally执行

接下来查看另一个案例：

```
new Promise(function(resolve,reject){
  reject()
  resolve()
}).then(function(){
  console.log('then执行')
}).catch(function(){
  console.log('catch执行')
}).finally(function(){
  console.log('finally执行')
})
```

我们在执行本段程序的时候会发现这个段代码的输出结果是:

catch执行->finally执行

再接下来查看下面的案例：

```
new Promise(function(resolve,reject){
}).then(function(){
  console.log('then执行')
}).catch(function(){
  console.log('catch执行')
}).finally(function(){
  console.log('finally执行')
})
```

我们在执行本段程序的时候会发现这个段代码的输出结果是:空

总结

分析了对象结构和状态后，我们了解了Promise的异步回调部分如何执行，取决于我们在初始化函数中的操作，并且初始化函数中一旦调用了resolve后面再执行reject也不会影响then执行，catch也不会执行，反之同理。而在初始化回调函数中，如果不执行任何操作，那么promise的状态就仍然是pending，所有注册的回调函数都不会执行。

关于链式调用

链式调用这个方式最经典的体现是在JQuery框架上，到现在仍然很多语言都在使用这种优雅的语法（不限前端还是后台），所以我们来简单认识一下什么是链式调用，为什么Promise对象可以.then().catch()这样调用。为什么还能.then().then()这样调用，他的原理大概是这样的。

```
function MyPromise(){
  return this
}
MyPromise.prototype.then = function(){
  console.log('触发了then')
  return this
}
new MyPromise().then().then().then()
```

其实他的本质就是在我们调用这些支持链式调用的函数的结尾时，他又返回了一个包含他自己的对象或者是一个新的自己，这些方式都可以实现链式调用。

Promise使用注意事项

在网页中运行如下代码查看返回结果：

```
var p = new Promise(function(resolve,reject){
  resolve('我是Promise的值')
})
console.log(p)
```

控制台上会得到如下内容：

```
Promise {<fulfilled>: '我是Promise的值'}
[[Prototype]]: Promise
[[PromiseState]]: "fulfilled"
[[PromiseResult]]: "我是Promise的值"
```

[[Prototype]]代表Promise的原型对象

[[PromiseState]]代表Promise对象当前的状态

[[PromiseResult]]代表Promise对象的值，分别对应resolve或reject传入的结果

1. 链式调用的注意事项

运行如下代码并查看结果：

```
//通过一个超长的链式调用我们学习一下链式调用的注意事项
var p = new Promise(function(resolve,reject){
  resolve('我是Promise的值')
})
console.log(p)
p.then(function(res){
  //该res的结果是resolve传递的参数
  console.log(res)
}).then(function(res){
  //该res的结果是undefined
  console.log(res)
  return '123'
}).then(function(res){
  //该res的结果是123
  console.log(res)
  return new Promise(function(resolve){
    resolve(456)
  })
}).then(function(res){
  //该res的结果是456
  console.log(res)
  return '我是直接返回的结果'
}).then()
  .then('我是字符串')
  .then(function(res){
  //该res的结果是“我是直接返回的结果”
  console.log(res)
})
```

控制台上会输出如下结果：

```
Promise {<fulfilled>: '我是Promise的值'}
ttt.html:16 我是Promise的值
ttt.html:18 undefined
ttt.html:21 123
ttt.html:26 456
ttt.html:31 我是直接返回的结果
```

根据现象我们可以分析出链式调用的基本形式（极其重要）：

1. 只要有then()并且触发了resolve，整个链条就会执行到结尾，这个过程中的第一个回调函数的参数是resolve传入的值
2. 后续每个函数都可以使用return返回一个结果，如果没有返回结果的话下一个then中回调函数的参数就是undefined
3. 返回结果如果是普通变量，那么这个值就是下一个then中回调函数的参数

4. 如果返回的是一个Promise对象，那么这个Promise对象resolve的结果会变成下一次then中回调的函数的参数
5. 如果then中传入的不是函数或者未传值，Promise链条并不会中断then的链式调用，并且在这之前最后一次的返回结果，会直接进入离它最近的正确的then中的回调函数作为参数

2. 中断链式调用

链式调用可以中断吗？答案是肯定的，我们有两种形式可以让.then的链条中断，如果中断还会触发一次.catch的执行。查阅下面的案例学习：

```
var p = new Promise(function(resolve,reject){
  resolve('我是Promise的值')
})
console.log(p)
p.then(function(res){
  console.log(res)
}).then(function(res){
  //有两种方式中断Promise
  // throw('我是中断的原因')
  return Promise.reject('我是中断的原因')
}).then(function(res){
  console.log(res)

}).then(function(res){
  console.log(res)

}).catch(function(err){
  console.log(err)
})
```

结果如下

```
Promise {<fulfilled>: '我是Promise的值'}
ttt.html:16 我是Promise的值
ttt.html:26 我是中断的原因
```

我们发现中断链式调用后会触发catch函数执行，并且从中断开始到catch中间的then都不会执行，这样链式调用的流程就结束了，中断的方式可以使用抛出一个异常或返回一个rejected状态的Promise对象。

3. 中断链式调用是否违背了Promise的精神?

我们在介绍Promise的时候强调了他是绝对保证的意思，并且Promise对象的状态一旦变更就不会再发生变化。当我们使用链式调用的时候正常都是then函数链式调用，但是当我们触发中断的时候catch却执行了。按照约定规则then函数执行，就代表Promise对象的状态已经变更为fulfilled了，但是catch函数执行时，Promise对象应该是rejected状态啊！这不科学。

在得到科学的解释前，先下面举个例子:

```
var p = new Promise(function(resolve,reject){
  resolve('我是Promise的值')
})
var p1 = p.then(function(res){

})
console.log(p)
console.log(p1)
console.log(p1===p)
```

当我们运行上面的代码时，控制台会出现如下的打印信息：

```
Promise {<fulfilled>: '我是Promise的值'}
ttt.html:18 Promise {<pending>}
ttt.html:19 false
```

我们会发现返回的p和p1 的状态本身就不一样，并且他们的对比结果是false，这就代表他们在堆内存中开辟了两个空间，p和p1对象分别保存了两个Promise对象的引用地址，所以then函数虽然每次都返回Promise对象，来实现链式调用，但是then函数每次返回的都是一个新的Promise对象。这样便解释的通了！也就是说每一次then函数在执行时，我们都可以让本次的结果在下一个异步步骤执行时，变成不同的状态，而且这也不违背Promise对象最初的约定。

4. 总结

根据以上的分析我们已经掌握了Promise在运行时的规则，这样就能解释的通，为什么最初通过Promise控制setTimeout每秒执行一次的功能可以实现，这是因为当我们使用then函数进行链式调用时，可以利用返回一个新的Promise对象来执行下一次then函数，而下一次then函数的执行，必须等待其内部的resolve调用。这样我们在new Promise时，放入setTimeout来进行延时，保证1秒之后让状态变更，这样就能不编写回调嵌套来实现连续的执行异步流程了。

Promise常用api

Promise.all()

当我们在代码中需要使用异步流程控制时，可以通过Promise.then来实现让异步流程一个接一个的执行，假设实际案例中，某个模块的页面需要同时调用3个服务端接口，并保证三个接口的数据全部返回后，才能渲染页面。这种情况如果a接口耗时1s、b接口耗时0.8s、c接口耗时1.4s，如果只用Promise.then来执行流程控制，可以保证三个接口按顺序调用结束再渲染页面，但是如果通过then函数的异步控制，必须等待每个接口调用完毕才能调用下一个，这样总耗时就是1+0.8+1.4 = 3.2s。这种累加显然增加了接口调用的时间消耗，所以Promise提供了一个all方法来解决这个问题：

`Promise.all([promise对象,promise对象,...]).then(回调函数)`

回调函数的参数是一个数组，按照第一个参数的promise对象的顺序展示每个promise的返回结果。

我们可以借助Promise.all来实现，等最慢的接口返回数据后，一起得到所有接口的数据，那么这个耗时将会只会按照最慢接口的消耗时间1.4s执行，总共节省了1.8s，参考代码如下：

```
//promise.all相当于统一处理了
```

```
//多个promise任务，保证处理的这些所有promise
//对象的状态全部变成为fulfilled之后才会出发all的
//.then函数来保证将放置在all中的所有任务的结果返回
let p1 = new Promise((resolve,reject) => {
  setTimeout(() => {
    resolve('第一个promise执行完毕')
  },1000)
})
let p2 = new Promise((resolve,reject) => {
  setTimeout(() => {
    resolve('第二个promise执行完毕')
  },2000)
})
let p3 = new Promise((resolve,reject) => {
  setTimeout(() => {
    resolve('第三个promise执行完毕')
  },3000)
})
Promise.all([p1,p3,p2]).then(res => {
  console.log(res)
}).catch(function(err){
  console.log(err)
})
```

Promise.race()

race方法与all方法使用格式相同：

`Promise.race([promise对象,promise对象,...]).then(回调函数)`

回调函数的参数是前面数组中最快一个执行完毕的promise的返回值。

所以使用race方法主要的使用场景是什么样的呢？举个例子，假设我们的网站有一个播放视频的页面，

通常流媒体播放为了保证用户可以获得较低的延迟，都会提供多个媒体数据源。我们希望用户在进入网页时，优先展示的是这些数据源中针对当前用户速度最快的那一个，这时便可以使用Promise.race()来让多个数据源进行竞赛，得到竞赛结果后，将延迟最低的数据源用于用户播放视频的默认数据源，这个场景便是race的一个典型使用场景。

下面我们可以参数考代码案例来查看race的介绍：

```
//promise.race()相当于将传入的所有任务
//进行了一个竞争，他们之间最先将状态变成fulfilled的
//那一个任务就会直接的触发race的.then函数并且将他的值
//返回，主要用于多个任务之间竞争时使用
let p1 = new Promise((resolve,reject) => {
  setTimeout(() => {
    resolve('第一个promise执行完毕')
  },5000)
})
let p2 = new Promise((resolve,reject) => {
```

```
  setTimeout(() => {
    reject('第二个promise执行完毕')
  },2000)
})
let p3 = new Promise(resolve => {
  setTimeout(() => {
    resolve('第三个promise执行完毕')
  },3000)
})
Promise.race([p1,p3,p2]).then(res => {
  console.log(res)
}).catch(function(err){
  console.error(err)
})
```

Promise的演进

在介绍了这么多Promise对象后，我们发现他的能力十分强大，使用模式非常的自由，并且将JavaScript一个时代的弊病从此“解套”。这个解套虽然比较成功，但是如果直接使用then()函数进行链式调用，我们的代码量仍然是非常沉重的，想要开发一个非常复杂的异步流程，依然需要大量的链式调用进行支撑，开发者还是会变得非常的难受。按照人类的线性思维，虽然JavaScript分同步和异步，但是单线程模式下，如果能完全按照同步代码的编写方式来处理异步流程，这才是最奈斯的结果，那么有没有办法让Promise对象能更进一步的接近同步代码呢?

Generator函数的介绍

在JavaScript中存在这样一种函数，我们先看一下这个函数的样子

```
function * fnName(){
  yield ***
  yield ***
}
```

ES6 新引入了 Generator 函数，可以通过 yield 关键字，把函数的执行流挂起，为改变执行流程提供了可能，从而为异步编程提供解决方案。 所以他的存在提供了让函数可以进行分步执行的能力。

举个栗子：

```
//该函数和普通函数不同，在执行的时候函数并不会运行并且会返回一个分步执行对象
//该对象存在next方法用来让程序继续执行，当程序遇到yield关键字的时候会停顿
//next返回的对象中包含value和done两个属性，value代表上一个yield返回的结果
//done代表程序是否执行完毕
function * test(){
  var a = yield 1
  console.log(a)
  var b = yield 2
  console.log(b)
  var c = a+b
  console.log(c)
```

```
}
//获取分步执行对象
var generator = test()
//输出
console.log(generator)
//步骤1 该程序从起点执行到第一个yield关键字后，step1的value是yield右侧的结果1
var step1 = generator.next()
console.log(step1)
//步骤2 该程序从var a开始执行到第2个yield后，step2的value是yield右侧的结果2
var step2 = generator.next()
console.log(step2)
//由于没有yield该程序从var b开始执行到结束
var step3 = generator.next()
console.log(step3)
```

我们查看程序的注释并且运行该程序看控制台的结果：

```
test {<suspended>}[[GeneratorLocation]]: ttt.html:10[[Prototype]]:
Generator[[GeneratorState]]: "closed"[[GeneratorFunction]]: ƒ * test()
[[GeneratorReceiver]]: Window
ttt.html:21 {value: 1, done: false}
ttt.html:12 undefined
ttt.html:23 {value: 2, done: false}
ttt.html:14 undefined
ttt.html:16 NaN
ttt.html:25 {value: undefined, done: true}
```

查看结果我们发现a和b的值不见了，c也是NaN虽然程序实现了分步执行，但是流程却出现了问题。

这是因为在分步执行过程中，我们是可以在程序中对运行的结果进行人为干预的，也就是说yield返回的结果和他左侧变量的值都是我们可以干预的。

接下来我们改造代码如下：

```
function * test(){
  var a = yield 1
  console.log(a)
  var b = yield 2
  console.log(b)
  var c = a+b
  console.log(c)
}
var generator = test()
console.log(generator)
var step1 = generator.next()
console.log(step1)
var step2 = generator.next(step1.value)
console.log(step2)
var step3 = generator.next(step2.value)
```

```
console.log(step3)
```

当我们将代码改造成上面的结构之后我们发现控制台中的数据就正确了：

```
test {<suspended>}
ttt.html:21 {value: 1, done: false}
ttt.html:12 1
ttt.html:23 {value: 2, done: false}
ttt.html:14 2
ttt.html:16 3
ttt.html:25 {value: undefined, done: true}
```

也就是说next函数执行的过程中我们是需要传递参数的，当下一次next执行的时候我们如果不传递参数，那么本次yield左侧变量的值就变成了undefined，所以我们如果想让yield左侧的变量有值就必须在next中传入指定的结果。

Generator能控制什么样的流程?

首先查看下列代码

```
function * test(){
  var a = yield 1
  console.log(a)
  var res = yield setTimeout(function(){
    return 123
  },1000)
  console.log(res)
  var res1 = yield new Promise(function(resolve){
    setTimeout(function(){
      resolve(456)
    },1000)
  })
  console.log(res1)
}
var generator = test()
console.log(step)
var step1 = generator.next()
console.log(step1)
var step2 = generator.next()
console.log(step2)
var step3 = generator.next()
console.log(step3)
var step4 = generator.next()
console.log(step4)
```

然后查看他的输出结果：

```
test {<suspended>}
ttt.html:27 {value: 1, done: false}
ttt.html:12 undefined
ttt.html:29 {value: 1, done: false}
ttt.html:16 undefined
ttt.html:31 {value: Promise, done: false}
ttt.html:22 undefined
ttt.html:33 {value: undefined, done: true}
```

根据调用情况我们可以自行测试，会发现输出结果时并没有任何的延迟，并且我们观察打印输出会发现普通变量可以直接在value中拿到，setTimeout位置我们拿到的值和回调函数内部的值完全不一样，而Promise对象我们可以拿到它本身。接下来我们展开查看Promise对象：

```
{value: Promise, done: false}
done: false
value: Promise
[[Prototype]]: Promise
[[PromiseState]]: "fulfilled"
[[PromiseResult]]: 456
[[Prototype]]: Object
```

我们发现Promise对象中是可以获取到内部的结果的，那么我们在Generator函数中能确保的就是，在分步过程中，能中使用Promise和普通对象都能拿到运行流程的结果，但是JavaScript中的setTimeout我们还是无法直接控制它的流程。

实现用Generator将Promise的异步流程同步化

通过上面的观察，我们可以通过递归调用的方式，来动态的去执行一个Generator函数，以done属性作为是否结束的依据，通过next来推动函数执行，如果过程中遇到了Promise对象我们就等待Promise对象执行完毕再进入下一步，我们这里排除异常和对象reject的情况，封装一个动态执行的函数如下：

```
/**
* fn:Generator函数对象
*/
function generatorFunctionRunner(fn){
  //定义分步对象
  let generator = fn()
  //执行到第一个yield
  let step = generator.next()
  //定义递归函数
  function loop(stepArg,generator){
    //获取本次的yield右侧的结果
    let value = stepArg.value
    //判断结果是不是Promise对象
    if(value instanceof Promise){
      //如果是Promise对象就在then函数的回调中获取本次程序结果
      //并且等待回调执行的时候进入下一次递归
      value.then(function(promiseValue){
```

```
        if(stepArg.done == false){
          loop(generator.next(promiseValue),generator)
        }
      })
    }else{
      //判断程序没有执行完就将本次的结果传入下一步进入下一次递归
      if(stepArg.done == false){
        loop(generator.next(stepArg.value),generator)
      }
    }
  }
  //执行动态调用
  loop(step,generator)
}
```

有了这个函数之后我们就可以将最初的三个setTimeout转换成如下结构进行开发

```
function * test(){
  var res1 = yield new Promise(function(resolve){
    setTimeout(function(){
      resolve('第一秒运行')
    },1000)
  })
  console.log(res1)
  var res2 = yield new Promise(function(resolve){
    setTimeout(function(){
      resolve('第二秒运行')
    },1000)
  })
  console.log(res2)
  var res3 = yield new Promise(function(resolve){
    setTimeout(function(){
      resolve('第三秒运行')
    },1000)
  })
  console.log(res3)
}
generatorFunctionRunner(test)
```

当我们通过上面的运行工具函数之后我们就可以在控制台看见每间隔1秒钟就输出一次

```
第一秒运行
ttt.html:22 第二秒运行
ttt.html:28 第三秒运行
```

经过这个yield修饰符之后我们惊喜的发现，抛去generatorFunctionRunner函数外，我们在Generator函数中已经可以将Promise的.then回调成功的规避了，yield修饰的Promise对象在运行到当前行时，程序就会进入挂起状态直到Promise对象变成完成状态，程序才会向下一行执行。这样我们就通过Generator函数对象成功的将Promise对象同步化了。这也是JavaScript异步编程的一个过渡期，通过这个解决方案，只需要提前准备好工具函数那么编写异步流程可以很轻松的使用yield关键字实现同步化。

关于Async和Await

经过了Generator的过渡之后异步代码同步化的需求逐渐成为了主流需求，这个过程在ES7版本中得到了提案，并在ES8版本中进行了实现，提案中定义了全新的异步控制流程。

```
//提案中定义的函数使用成对的修饰符
async function  test(){
  await ...
  await ...
}
test()
```

查看代码结构之后我们发现他的编写方式与Generator函数结构很相似，提案中规定了我们可以使用async修饰一个函数，这样就能在该函数的直接子作用域中，使用await来自动的控制函数的流程，await 右侧可以编写任何变量或对象，当右侧是普通对象的时候函数会自动返回右侧的结果并向下执行，而当await右侧为Promise对象时，如果Promise对象状态没有变成完成，函数就会挂起等待，直到Promise对象变成fulfilled，程序再向下执行，并且Promise的值会自动返回给await左侧的变量中。async和await需要成对出现，async可以单独修饰函数，但是await只能在被async修饰的函数中使用。

有了await和async就相当于使用了自带执行函数的Generator函数，这样我们就不再需要单独针对Generator函数进行开发了，所以async和await逐渐成为主流异步流程控制的终极解决方案。而Generator慢慢淡出了业务开发者的舞台，不过Generator函数成为了向下兼容过渡期版本浏览器的候补实现方式，虽然在现今的大部分项目业务中使用Generator函数的场景非常的少，但是如果查看脚手架项目中通过babel构建的JavaScript生产代码，我们还是能大量的发现Generator的应用的，他的作用就是为了兼容不支持async和await的浏览器。

认识async函数

创建如下函数，执行并查看控制台输出：

```
async function test(){
  return 1
}
let res = test()
console.log(res)
```

输出控制台如下：

```
Promise {<fulfilled>: 1}
[[Prototype]]: Promise
[[PromiseState]]: "fulfilled"
[[PromiseResult]]: 1
```

根据控制台结果我们发现其实async修饰的函数，本身就是一个Promise对象，虽然我们在函数中return的值是1，是使用了async修饰之后，这个函数运行时并没有直接返回1，而是返回了一个值为1的Promise对象。

接下来我们测试如下流程，先分析运行结果，猜测输出的顺序：

```
async function  test(){
  console.log(3)
  return 1
}
console.log(1)
test()
console.log(2)
```

执行该流程之后发现输出的结果是1，3，2。很惊喜是不是！按照Promise对象的执行流程function被async修饰之后它本身应该变成异步函数，那么他应该在1和2输出完毕之后在输出3，但是结果却出人意料，这又一次打破了单线程异步模型的概念。

别急，冷静一下，先回想一下Promise对象的结构：

```
new Promise(function(){

}).then(function(){

})
```

我们在介绍Promise对象时，特别介绍了一下回调函数，并且强调他是一个极少数的既使用同步回调流程又使用了异步的回调流程的对象，所以在new Promise时的function是同步流程。现在介绍这个和刚才的输出有关系吗？当然有，接下来查看下面的逻辑，还是先猜测一下输出顺序：

```
async function  test(){
  console.log(3)
  var a = await 4
  console.log(a)
  return 1
}
console.log(1)
test()
console.log(2)
```

我们发现奇怪的事情又发生了，控制台输出的顺序是1，3，2，4

按照我们一开始以为的流程，test函数应该是同步逻辑，那么3和4应该是连着输出的，他不应该会出现3在2之前4在2之后输出的情况，这个同步逻辑和异步逻辑都说不过去，那么我们将当前的函数翻译一下，由于async修饰的函数会被解释成Promise对象，所以我们可以将其翻译成如下结构：

```
console.log(1)
new Promise(function(resolve){
  console.log(3)
  resolve(4)
}).then(function(a){
  console.log(a)
})
console.log(2)
```

看到这个Promise对象我们就豁然开朗，由于初始化的回调是同步的所以1，3，2都是同步代码而4是在resolve中传入的，then代表异步回调所以4应该最后输出。

综上所述，async函数中有一个最大的特点，就是第一个await会作为分水岭一般的存在，在第一个await的右侧和上面的代码，全部是同步代码区域相当于new Promise的回调，第一个await的左侧和下面的代码，就变成了异步代码区域相当于then的回调，所以就出现上面我们发现的灵异问题。

最终的setTimeout解决代码

经过了两个时代的变革，现在我们可以使用如下的方式来进行流程控制，不再需要依赖自己定义的流程控制器函数来进行分步执行，这一切的核心起源都是Promise对象的规则定义开始的，所以最终我们的解决方案如下。

```
async function test(){
  var res1 = await new Promise(function(resolve){
    setTimeout(function(){
      resolve('第一秒运行')
    },1000)
  })
  console.log(res1)
  var res2 = await new Promise(function(resolve){
    setTimeout(function(){
      resolve('第二秒运行')
    },1000)
  })
  console.log(res2)
  var res3 = await new Promise(function(resolve){
    setTimeout(function(){
      resolve('第三秒运行')
    },1000)
  })
  console.log(res3)
}
test()
```

总结

从回调地狱到Promise的链式调用到Generator函数的分步执行再到async和await的自动异步代码同步化机制，经历了很多个年头，所以面试中为什么经常问到Promise，并且重点沿着Promise对象深入的挖掘去问你各种问题，主要是考察程序员对Promise对象本身以及他的发展历程是否有深入的了解，同时也是在考察面试者对JavaScript的事件循环系统和异步编程的基本功是否足够的扎实。Promise和事件循环系统并不是JavaScript中的高级知识，而是真正的基础知识，所以所有人想要在行业中更好的发展下去，这些知识都是必备基础，必须扎实掌握。我们未来对自己的定位是软件开发/研发工程师，并不是码农～