深入浅出 JS 异步处理技术方案

为什么要异步

“当我们在星巴克买咖啡时，假设有100个人在排队，也许咖啡的下单只要10S，但是咖啡的制作到客人领取咖啡要1000S。如果在同步的场景下，第一个客人下单到领取完咖啡要1010S才能轮到下一个客人，这在效率（某些场景）上来说会比较低下。如果我们异步处理这个流程，客人下单10S拿到凭证，客人就可以去做别的事情，并且10S后下一个客人可以继续下单，并不阻碍流程。反而可以通过凭证，让客人拿到自己的咖啡，也许时间上并不是第一个下单的客人先拿到。”

在网页的世界里也是同样的道理，不妨我们看看在执行JS代码的主线程里，如果遇到了AJAX请求，用户事件等，如果不采用异步的方案，你会一直等待，等待第一个耗时的处理完成才能接上下一个JS代码的执行，于是界面就卡住了。

“也许有人会想，既然大家都说现在网页上性能损耗最大的属于DOM节点的操作，把这些搞成异步，行不行？其实这会带来一个不确定性问题：既“成功”的状态到底谁先来的问题。可以想象一下，如果我们在操作DOM，既给节点添加内容，也给节点删除，那么到底以谁为基准呢？考虑到复杂性，也就可见一斑了。”

Event loop

虽然异步与event loop没有太直接的关系，准确的来讲event loop 只是实现异步的一种机制。（了解为主）

“还是以上面咖啡馆为例子，假定场景还是100人，这100人除了下单是与咖啡本身有关联之外，其余的时间，比如看书，玩游戏的等可以视为自己的执行逻辑。如果用event loop来给它做一个简单的画像，那么它就像：在与咖啡店店员沟通下单视为主执行栈，咖啡的制作可以视为一个异步任务，添加到一个任务队列里，一直等带100个人都下单完成，然后开始读取任务队列中的异步任务，事件名就是下单凭证，如果有对应的handler，那么就执行叫对应的客人来领取咖啡。这个过程，是循环不断的。假设没有客人来下单的时候，也就是店员处于空闲时间（可能自己去搞点别的）。”

传统的Callback

假定一个asyncFetchDataSource函数用于获取远程数据源，可能有20S。

```
function asyncFetchDataSource(cb){
   (… 获取数据, function(response){
      typeof cb === 'function' && cb(response)
   })
}
```

这种形式的callback可以适用于简单场景，如果这里有一个更复杂的场景，比如获取完数据源之后，依据id，获取到某个数据，在这某个数据中再依据id来更新某个列表，可以遇见的能看到代码变成了：

```
asyncFetchDataSource('',function(data_a){
  const { id_a } = data_a
   asyncFetchDataSource( id_a,function(data_b){
           const { id_b } = data_b
    asyncFetchDataSource(id, function(data_c){

    })
   })
})
```

如果有极端情况出现，这里的callback就会变成无极限了。

Thunk函数

这是一种“传名调用”的策略，表现的形式就是将参数放入一个临时函数，然后再将这个临时函数传入函数体内。

```
function asyncFetchDataSource(url){
   return function(callback){
      fetch(url, callback)
   }
}

const dataSource = asyncFetchDataSource('https://github.com/icepy');
dataSource(function(data){

})
```

Promise

Promise正是想来处理这样的异步编程，如果我们用Promise该如何处理一段Ajax？

```
function fetch(){
  return new Promise(function(resolve,reject){
    $.ajax({
      url: 'xxx',
      success:function(data){
        resolve(data)
      },
      error:function(error){
        reject(error)
      }
    })
  })
}
fetch().then(function(data){
}).catch(function(error){})
```

Promise声明周期：

- 进行中（pending）
- 已经完成（fulfilled）
- 拒绝（rejected）

如同上面Ajax的例子，我们可以很好的包装一个函数，让fetch函数返回一个Promise对象。在Promise构造函数里，可以传入一个callback，并且在这里完成主体逻辑的编写。唯一需要注意的是：Promise对象只能通过resolve和reject函数来返回，在外部使用then或catch来获取。如果你直接抛出一个错误（throw new Error('error')），catch也是可以正确的捕获到的。

Promise其他的方法

Promise.all（当所有在可迭代参数中的 promises 已完成，或者第一个传递的 promise（指 reject）失败时，返回 promise。）

```
var p1 = Promise.resolve(3);
var p2 = 1337;
var p3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, "foo");
});
Promise.all([p1, p2, p3]).then(values => {
  console.log(values); // [3, 1337, "foo"]
});
```

Promise.race（返回一个新的 promise，参数iterable中只要有一个promise对象”完成（resolve）”或”失败（reject）”，新的promise就会立刻”完成（resolve）”或者”失败（reject）”，并获得之前那个promise对象的返回值或者错误原因。）

```
var p1 = new Promise(function(resolve, reject) {
    setTimeout(resolve, 500, "one");
});
var p2 = new Promise(function(resolve, reject) {
    setTimeout(resolve, 100, "two");
});
Promise.race([p1, p2]).then(function(value) {
  console.log(value); // "two"
  // 两个都完成，但 p2 更快
});
```

有趣的是如果你使用ES6的class，你是可以去派生Promise的。

```
class MePromise extends Promise{
 // 处理 ...
}
```

Generator

Generator可以辅助我们完成很多复杂的任务，而这些基础知识，又与iterator息息相关，举一个很简单的例子，相信有很多朋友，应该使用过co这个异步编程的库，它就是用Generator来实现，当然它的设计会比例子要复杂的多，我们先来看一个co简单的用法：


```
import co from 'co'
co(function* () {
  var result = yield Promise.resolve(true);
  return result;
}).then(function (value) {
  console.log(value);
}, function (err) {
  console.error(err.stack);
});
```

相应的，我们来实现一个简化的版本：

```
function co(task){
  let _task = task()
  let resl = _task.next();
  while(!resl.done){
    console.log(resl);
    resl = _task.next(resl.value);
  }
}
function sayName(){
  return {
    name: 'icepy'
```

```
  }
}
function assign *(f){
  console.log(f)
  let g = yield sayName()
  return Object.assign(g,{age:f});
}
co(function *(){
  let info = yield *assign(18)
  console.log(info)
})
```

虽然，这个例子中，还不能很好的看出来“异步”的场景，但是它很好的描述了Generator的使用方式。

从最开始的定义中，已经和大家说明了，Generator最终返回的依然是一个迭代器对象，有了这个迭代器对象，当你在处理某些场景时，你可以通过yield来控制，流程的走向。通过co函数，我们可以看出，先来执行next方法，然后通过一个while循环，来判断done是否为true，如果为true则代表整个迭代过程的结束，于是，这里就可以退出循环了。在Generator中的返回值，可以通过给next方法传递参数的方式来实现，也就是遇上第一个yield的返回值。

有逻辑，自然会存在错误，在Generator捕获错误的时机与执行throw方法的顺序有关系，一个小例子：


```
let hu = function *(){
  let g = yield 1;
  try {
    let j = yield 2;
  } catch(e){
    console.log(e)
  }
  return 34
}
let _it = hu();
console.log(_it.next())
console.log(_it.next())
console.log(_it.throw(new Error('hu error')))
```

当我能捕获到错误的时机是允许完第二次的yield，这个时候就可以try了。

async await

```
async function createNewDoc() {
  let response = await db.post({}); // post a new doc
```

```
  return await db.get(response.id); // find by id
}
```

https://tc39.github.io/ecmascript-asyncawait/

根据规范规定一个asnyc函数总是要返回一个Promise，从代码直观上来说，虽然简洁了，但是async await并未万能，它有很大的局限性，比如：

- 因为是顺序执行，假设有三个请求，那么这里并没有很好的利用到异步带来的止损（再包装一个Promise.all）
- 如果要捕获异常，需要去包try catch
- 缺少控制流程，比如progress（进度）pause，resume等周期性的方法
- 没有打断的功能

主流的异步处理方案

我喜欢用co，而且社区使用也很广泛，https://github.com/tj/co。

```
co(function* () {
  var result = yield Promise.resolve(true);
  return result;
}).then(function (value) {
  console.log(value);
}, function (err) {
  console.error(err.stack);
});
```


babel polyfill 支持，在浏览器环境中使用异步解决方案

如果你想使用全的polyfiil，直接npm install –save babel-polyfill，然后在webpack里进行配置即可。

```
module.exports = {
  entry: ["babel-polyfill", "./app/js"]
};
```

当然由于我目前的开发基于的浏览器都比较高，所以我一般是挑选其中的：

https://github.com/facebook/regenerator/tree/master/packages/regenerator-runtime
https://github.com/facebook/regenerator/tree/master/packages/regenerator-transform

如果你要使用async await配置上http://babeljs.io/docs/plugins/transform-async-to-generator/即可

Node.js环境中使用异步解决方案

由于本人的node使用的LTS已经是8.9.3版本了，所以大部分情况下已经不再使用babel去进行转换，而是直接使用co这样的库。当然co也不是万能，一定要根据业务场景，与其他异步处理的方式，配合中使用。

总结

相信未来的JS编程，只会越来越简单，不要拘泥于语法，语言上的特性，不妨多看一看“外面的世界“。