这是一篇傻瓜都能看懂的Promises文章！

以下内容出自scotch上Jecelyn Yeen的分享哦。

此文目的只有一个：让我们更容易的了解JavaScript Promises！

JavaScript Promises其实不难。然而，很多人一开始就觉得有点难理解..。因此我想用一种假设的方式写下我理解promise。

（一）理解promises

举个简单例子：

想象你是一个孩子。你老妈承诺下礼拜 给你买个新手机。你 [不知道] 你是否会得到手机直到下礼拜。你老妈可以真的买你一个全新的手机，也可以让你滚蛋并告诉你不买了（如果她不高兴了）。

这是一个承诺。一个承诺有3个状态。分别是:

1.悬而未决：你 [不知道] 你是否会得到手机直到下礼拜。

2.解决：你老妈可以真的买你一个全新的手机。

3.拒绝：你老妈拒绝给你买，因为你惹她不高兴。

（二）创建一个promise

咱们把上面的例子转换成JavaScript.

```
/* ES5 */
var isMomHappy = false;

// Promise
var willIGetNewPhone = new Promise(
    function (resolve, reject) {
        if (isMomHappy) {
            var phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone); // fulfilled
        } else {
            var reason = new Error('mom is not happy');
            reject(reason); // reject
        }

    }
);
```

--> 代码表现力挺强的嘛!

1. “isMomHappy”是个布尔值，定义老妈是否开心。
2. “willIGetNewPhone”是一个promise，这个承诺可以解决(给你买)，也可以拒绝(老妈不开心就不给你买)。.
3. 还有一个标准语法去定义一个新的promise，参考MDN documentation，看下面代码：

```
// promise syntax look like this
new Promise(/* executor*/ function (resolve, reject) { ... } );
```

1. 需要记住的是，在你定义的promise里，当结果是成功的，叫 解决(你的成功值)，如果结果失败了，叫 拒绝（你的失败值）。
2. 在我们的例子中，如果老妈高兴，我们会得到一个电话。因此，我们称 resolvefunction（电话变量）。如果老妈不高兴，我们称为 拒绝函数（拒绝的理由）；

（三）玩转promises

现在有了promise，咱们就开始玩一玩。

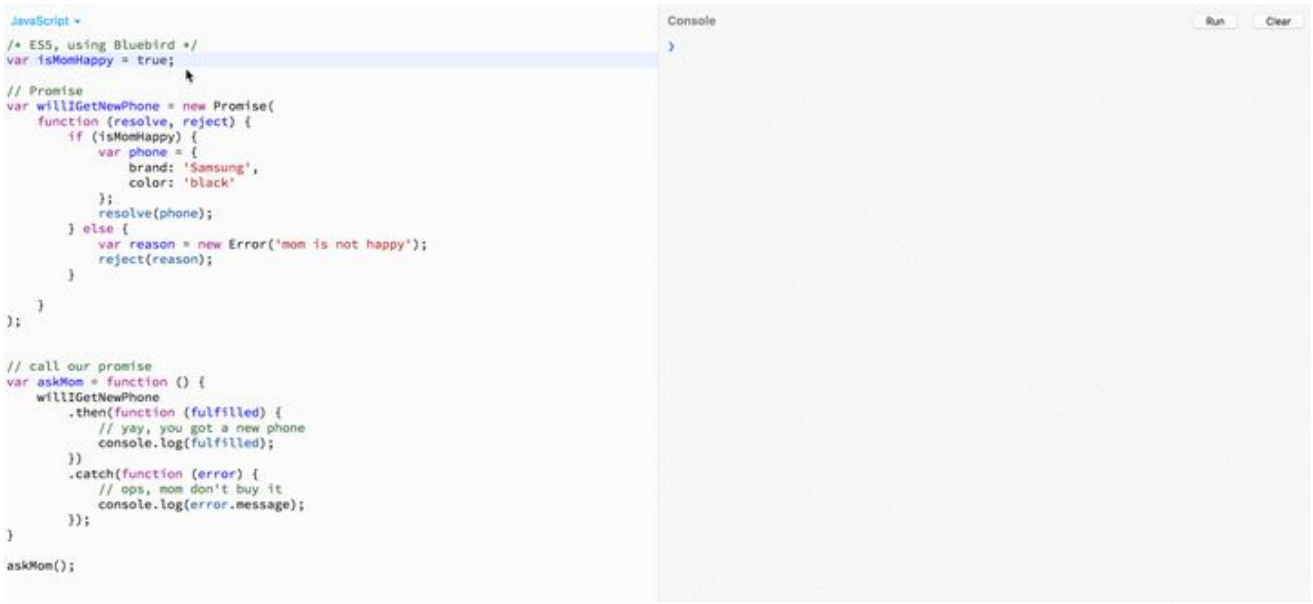
```
/* ES5 */
...

// call our promise
var askMom = function () {
    willIGetNewPhone
        .then(function (fulfilled) {
            // yay, you got a new phone
            console.log(fulfilled);
         // output: { brand: 'Samsung', color: 'black' }
        })
        .catch(function (error) {
            // oops, mom don't buy it
            console.log(error.message);
         // output: 'mom is not happy'
        });
};

askMom();
```

1. 我们有一个函数“askmom”。在这个函数中，我们将使用我们的承诺“ willigetnewphone”。
2. 当我们的promise是解决或者拒绝的时候，我们要做点事儿，我们用“.then & .catch ”来处理我们的行动；
3. 在“.then”里面有函数“ function(fulfilled) { ... } ”。这个函数的返回值是啥呢？返回值是promise解决的值（你的成功时候的值）；在我们的案例中它是一部新电话 。
4. 在 ".catch"里面有函数"function(error){ ... } "。猜测一下，其实这个返回的是错误值，就是promise拒绝的值。

走一下案例看下结果!



（四）链接promises

Promises都是可链的。

这样说吧，你，咱例子中的孩子，承诺给你的朋友说：如果你老妈给你买了新手机，就让你的朋友过过眼瘾，你要显摆显摆！

这又是另外一个promise了，写成代码看下!

```
...

// 2nd promise
var showOff = function (phone) {
    return new Promise(
        function (resolve, reject) {
            var message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';

            resolve(message);
        }
    );
};
```

注意：

- 在这个例子中，你可能会意识到我们不叫 拒绝。因为它是可选的。
- 我们可以缩短这个案例，看代码：

```
// shorten it
...

// 2nd promise
var showOff = function (phone) {
    var message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';

    return Promise.resolve(message);
};
```

咱们现在链接promise。你，这个孩子只能在得到手机后才能显摆手机。

```
// call our promise
var askMom = function () {
    willIGetNewPhone
    .then(showOff) // chain it here
    .then(function (fulfilled) {
            console.log(fulfilled);
         // output: 'Hey friend, I have a new black Samsung phone.'
        })
        .catch(function (error) {
            // oops, mom don't buy it
            console.log(error.message);
         // output: 'mom is not happy'
        });
};
```

--> 简单吧!

（五）promises都是异步的

Promises都是异步的，咱们在这个promise开始和结束前写一段信息。

```
// call our promise
var askMom = function () {
    console.log('before asking Mom'); // log before
    willIGetNewPhone
        .then(showOff)
        .then(function (fulfilled) {
            console.log(fulfilled);
        })
        .catch(function (error) {
            console.log(error.message);
        });
    console.log('after asking mom'); // log after
}
```

期望输出的顺序是什么？也许你以为：

```
1. before asking Mom
2. Hey friend, I have a new black Samsung phone.
3. after asking mom
```

然而，实际的输出序列是：

```
1. before asking Mom
2. after asking mom
3. Hey friend, I have a new black Samsung phone.
```

为啥？因为生活就是不等人的，JavaScript也是!

你，孩子，不会说不去玩了就干等你老妈的承诺（新手机），对吧。这就是 异步调用，代码将无阻塞运行或等待结果。任何需要等promise的行为放在".then"里面。

Promise在 ES5 /ES6 /ES7 下:

ES5 - 大多数的浏览器

ES5不支持promise，大多数浏览器借助第三方库（Bluebird 、Q）的话可以实现；

ES6 - 现代浏览器

演示代码是ok的，因为ES6支持promise。此外，我们也可以用ES6的箭头函数来简化代码。也可以用上let和const。

下面是ES6的演示代码：

```
    /* ES6 */
const isMomHappy = true;
// Promise
const willIGetNewPhone = new Promise(
    (resolve, reject) => { // fat arrow
        if (isMomHappy) {
            const phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone);
        } else {
            const reason = new Error('mom is not happy');
            reject(reason);
        }

    }
);
const showOff = function (phone) {
    const message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';
    return Promise.resolve(message);
};
// call our promise
const askMom = function () {
    willIGetNewPhone
        .then(showOff)
        .then(fulfilled => console.log(fulfilled)) // fat arrow
        .catch(error => console.log(error.message)); // fat arrow
};
askMom();
```

--> 注意到所有的 var 都被 const 取代。所有的函数（解决，拒绝） 都使用箭头函数简化。

ES7 - 异步等待使语法看起来更漂亮!

ES7介绍 异步 和 等待 语法。它使异步语法看起来更漂亮、更容易理解，没有 ".then" 和".catch"。

用ES7语法来重写我们的例子：

```
    /* ES7 */
const isMomHappy = true;
// Promise
const willIGetNewPhone = new Promise(
    (resolve, reject) => {
        if (isMomHappy) {
            const phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone);
        } else {
            const reason = new Error('mom is not happy');
            reject(reason);
        }
    }
);
// 2nd promise
async function showOff(phone) {
    return new Promise(
        (resolve, reject) => {
            var message = 'Hey friend, I have a new ' +
                          phone.color + ' ' + phone.brand + ' phone';
            resolve(message);
        }
    );
};
// call our promise
async function askMom() {
    try {
        console.log('before asking Mom');
        let phone = await willIGetNewPhone;
        let message = await showOff(phone);
        console.log(message);
        console.log('after asking mom');
    }
    catch (error) {
        console.log(error.message);
    }
}
(async () => {
    await askMom();
})();
```

1. 当你需要在一个函数里面返回一个promise，你在异步 调用这个函数。例如 案例中的异步函数"function showOff(phone)"。
2. 当你需要一个promise，你就在等待着 。例如 "let phone = await willIGetNewPhone;" & " let message = await showOff(phone)"。
3. 使用 " try { ... } catch(error) { ... } " 捕捉promise错误/拒绝 。

为什么使用promise？什么时候使用它们?

为什么我们需要promise？promise之前的世界是怎样的？在回答这些问题之前，让我们回到最基础的地方。

对比正常函数和异步函数

让我们来看看这两个例子，这两个例子执行两个数字相加：

正常函数两个数相加：

```
// add two numbers normally

function add (num1, num2) {
    return num1 + num2;
}

const result = add(1, 2); // you get result = 3 immediately
```

异步函数两个数相加：

```
// add two numbers remotely

// get the result by calling an API
const result = getAddResultFromServer('http://www.example.com?num1=1&num2=2');
// you get result  = "undefined"
```

如果你用正常的函数让两个数字，你会立即得到结果.。但是，当你发出远程调用来得到结果时，你需要等待，你不能立即得到结果.。

或者这样说，你不知道你是否会得到结果，因为服务器可能会下降，响应速度慢等，你不希望在等待结果的时候，整个进程被阻止。调用API，下载文件，读取文件中的一些你要执行的常用的异步操作。

Promises之前的世界：Callback回调

我们必须使用promise来实现异步调用吗？不是的，在promise之前，我们使用回调。回调函数只是你得到返回结果时调用的函数.。让我们使用回调修改前面的例子。

```
// add two numbers remotely
// get the result by calling an API

function addAsync (num1, num2, callback) {
    // use the famous jQuery getJSON callback API
    return $.getJSON('http://www.example.com', {
        num1: num1,
        num2: num2
    }, callback);
}

addAsync(1, 2, success => {
    // callback
    const result = success; // you get result = 3 here
});
```

异步看起来ok，为啥还要用promise呢?

如果你想进行后续的异步操作怎么办?

三个数相加，正常函数这样写:

```
  // add two numbers normally

  let resultA, resultB, resultC;

   function add (num1, num2) {
      return num1 + num2;
  }

  resultA = add(1, 2); // you get resultA = 3 immediately
  resultB = add(resultA, 3); // you get resultB = 6 immediately
  resultC = add(resultB, 4); // you get resultC = 10 immediately

  console.log('total' + resultC);
  console.log(resultA, resultB, resultC);
```

怎么看起来像回调?

```
// add two numbers remotely
// get the result by calling an API

let resultA, resultB, resultC;

function addAsync (num1, num2, callback) {
    // use the famous jQuery getJSON callback API
    return $.getJSON('http://www.example.com', {
        num1: num1,
        num2: num2
    }, callback);
}

addAsync(1, 2, success => {
    // callback 1
    resultA = success; // you get result = 3 here

    addAsync(resultA, 3, success => {
        // callback 2
        resultB = success; // you get result = 6 here

        addAsync(resultB, 4, success => {
            // callback 3
            resultC = success; // you get result = 10 here

            console.log('total' + resultC);
            console.log(resultA, resultB, resultC);
        });
    });
});
```

语法对用户是友好的。有一个更好的术语，它看起来像一个金字塔，但人们通常把这称为“回调地狱”，因为回调嵌套到另一个回调。假设你有10的回调，你的代码将嵌套的10倍!

逃离回调地狱吧!

让promise来拯救。让我们来看看同样的例子的promise版本。

```
// add two numbers remotely using observable

  let resultA, resultB, resultC;

  function addAsync(num1, num2) {
      // use ES6 fetch API, which return a promise
      return fetch(`http://www.example.com?num1=${num1}&num2=${num2}`)
      .then(x => x.json());
  }

  addAsync(1, 2)
      .then(success => {
          resultA = success;
          return resultA;
      })
      .then(success => addAsync(success, 3))
      .then(success => {
          resultB = success;
          return resultB;
      })
      .then(success => addAsync(success, 4))
      .then(success => {
          resultC = success;
          return resultC;
      })
      .then(success => {
          console.log('total: ' + success)
          console.log(resultA, resultB, resultC)
      });
```

有了承诺，我们用".then"扁平化回调。在某种程度上，因为没有回调嵌套，它看起来干净。当然，用ES7异步语法，可以让他看起来更简洁！

新家伙：Observables观测值

在promise已经让你很幸福的时候，Observables这货来锦上添花，让处理异步数据量更容易。

> Observables是懒惰的事件流，可以发出零个或多个事件，而且可能完成也可能不完成。

promise和observable的关键区别是：

- Observables观测值是可以取消的
- Observable是懒惰的

不要害怕，让我们来看看Observables的案例。在这个例子中，关于Observables使用RxJS。

```
let Observable = Rx.Observable;
let resultA, resultB, resultC;

function addAsync(num1, num2) {
    // use ES6 fetch API, which return a promise
    const promise = fetch(`http://www.example.com?num1=${num1}&num2=${num2}`)
        .then(x => x.json());

    return Observable.fromPromise(promise);
}

addAsync(1,2)
  .do(x => resultA = x)
  .flatMap(x => addAsync(x, 3))
  .do(x => resultB = x)
  .flatMap(x => addAsync(x, 4))
  .do(x => resultC = x)
  .subscribe(x => {
    console.log('total: ' + x)
    console.log(resultA, resultB, resultC)
  });
```

Observables观测值可以很容易的做一些时髦的东西。看案例：

```
...

addAsync(1,2)
  .delay(3000) // delay 3 seconds
  .do(x => resultA = x)
  ...
```

好吧，咱们以后再接着讨论Observables观测值吧！

总结：

自己尝试了解和使用callbacks 和 promises，目前还不用太担心Observables，但是这三者都会影响到你的发展哦。