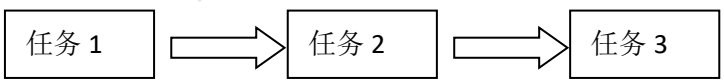
究竟什么是异步编程

在我们的工作和学习当中，到处充满了异步的身影，到底什么是异步，什么是异步编程，为什么要用异步编程，以及经典的异步编程有哪些，在工作中的场景又有什么，我们一点点深入的去学习。

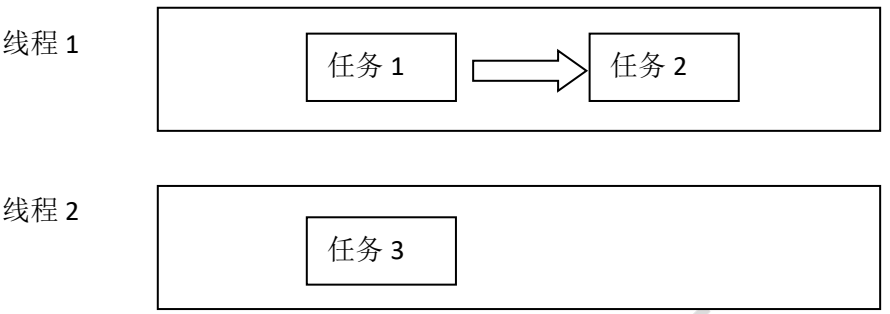
什么是异步编程？

有必要了解一下，什么是异步编程，为什么要异步编程。
先说一个概念异步与同步。介绍异步之前，回顾一下，所谓同步编程，就是计算机一行一行按顺序依次执行代码，当前代码任务耗时执行会阻塞后续代码的执行。
同步编程，即是一种典型的请求-响应模型，当请求调用一个函数或方法后，需等待其响应返回，然后执行后续代码。
一般情况下，同步编程，代码按序依次执行，能很好的保证程序的执行，但是在某些场景下，比如读取文件内容，或请求服务器接口数据，需要根据返回的数据内容执行后续操作，读取文件和请求接口直到数据返回这一过程是需要时间的，网络越差，耗费时间越长，如果按照同步编程方式实现，在等待数据返回这段时间，JavaScript 是不能处理其他任务的，此时页面的交互，滚动等任何操作也都会被阻塞，这显然是及其不友好，不可接受的，而这正是需要异步编程大显身手的场景。我们想通过 Ajax 请求数据来渲染页面，这是一个在我们前端当中很常见渲染页面的方式。基本每个页面都会都这样的过程。在这里用同步的方式请求页面会怎么样？**浏览器锁死，不能进行其他操作。而且每当发送新的请求，浏览器都会锁死，用户体验极差。**



在浏览器中同步执行将会是上面的这个样子， 任务 1 做完才能做任务 2，任务 2 做完才会做任务 3。这里面体现出同步编程的**有序**的特点。只能 1，2，3 不能 1，3，2。但是我们的代码逻辑中可以存在多任务同时执行的过程。在我们生活中，煮饭和烧水可以同时去做，同样在我们编程中也需要这样的逻辑。
在计算机中有多线程的概念，什么意思呢，每一个线程做一件事，像下面

这样。



在不同的线程中可以执行不同的任务。
但是我们的 JavaScript 是单线程的，这里的单线程，强调的执行线程是单线程。后面也是有线程池的，线程以及线程池的概念在这里就不多说了。想了解的同学可以看看操作系统相关书籍。
JavaScript 语言执行环境是单线程的，单线程在程序执行时，所走的程序路径按照连续顺序排下来，前面的必须处理好，后面的才会执行。
但是我们也需要类似多线程机制的这种执行方式。但是 JavaScript 还是单线程的，我们需要异步执行，异步执行会使得多个任务并发执行。
并行与并发。前文提到多线程的任务可以并行执行，而 JavaScript 单线程异步编程可以实现多任务并发执行，这里有必要说明一下并行与并发的区别。
并行，指同一时刻内多任务同时进行。边煮饭，边烧水，可以同时进行
并发，指在同一时间段内，多任务同时进行着，但是某一时刻，只有某一任务执行。边吃饭边喝水，同一时间点只能喝水和吃饭。

接下来说一说异步机制。
并发模型
目前，我们已经知道，JavaScript 执行异步任务时，不需要等待响应返回，可以继续执行其他任务，而在响应返回时，会得到通知，执行回调或事件处理程序。那么这一切具体是如何完成的，又以什么规则或顺序运作呢？接下来我们需要解答这个问题。回调和事件处理程序本质上并无区别，只是在不同情况下，不同的叫法。
前文已经提到，JavaScript 异步编程使得多个任务可以并发执行，而实现这一功能的基础是 JavaScript 拥有一个基于事件循环的并发模型。

堆栈与队列

介绍 JavaScript 并发模型之前，先简单介绍堆栈和队列的区别：

堆（heap）：内存中某一未被阻止的区域，通常存储对象（引用类型）；
栈（stack）：后进先出的顺序存储数据结构，通常存储函数参数和基本类型值变量（按值访问）；
队列（queue）：先进先出顺序存储数据结构。
事件循环（Event Loop）
JavaScript 引擎负责解析，执行 JavaScript 代码，但它并不能单独运行，通常都得有一个宿主环境，一般如浏览器或 Node 服务器，前文说到的单线程是指在这些宿主环境创建单一线程，提供一种机制，调用 JavaScript 引擎完成多个 JavaScript 代码块的调度，执行（是的，JavaScript 代码都是按块执行的），这种机制就称为事件循环（Event Loop）。

JavaScript 执行环境中存在的两个结构需要了解：

消息队列(message queue)，也叫任务队列（task queue）：存储待处理消息及对应的回调函数或事件处理程序；
执行栈(execution context stack)，也可以叫执行上下文栈：JavaScript 执行栈，顾名思义，是由执行上下文组成，当函数调用时，创建并插入一个执行上下文，通常称为执行栈帧（frame），存储着函数参数和局部变量，当该函数执行结束时，弹出该执行栈帧；
注：关于全局代码，由于所有的代码都是在全局上下文执行，所以执行栈顶总是全局上下文就很容易理解，直到所有代码执行完毕，全局上下文退出执行栈，栈清空了；也即是全局上下文是第一个入栈，最后一个出栈。

任务
分析事件循环流程前，先阐述两个概念，有助于理解事件循环：同步任务和异步任务。

任务很好理解，JavaScript 代码执行就是在完成任务，所谓任务就是一个函数或一个代码块，通常以功能或目的划分，比如完成一次加法计算，完成一次 ajax 请求；很自然的就分为同步任务和异步任务。同步任务是连续的，阻塞的；而异步任务则是不连续，非阻塞的，包含异步事件及其回调，当我们谈及执行异步任务时，通常指执行其回调函数。

事件循环流程
关于事件循环流程分解如下：

宿主环境为 JavaScript 创建线程时，会创建堆(heap)和栈(stack)，堆内存储 JavaScript 对象，栈内存储执行上下文；
栈内执行上下文的同步任务按序执行，执行完即退栈，而当异步任务执行时，该异步任务进入等待状态（不入栈），同时通知线程：当触发该事件时（或该异步操作响应返回时），需向消息队列插入一个事件消息；
当事件触发或响应返回时，线程向消息队列插入该事件消息（包含事件及回调）；
当栈内同步任务执行完毕后，线程从消息队列取出一个事件消息，其对应异步任务（函数）入栈，执行回调函数，如果未绑定回调，这个消息会被丢弃，执行完任务后退栈；
当线程空闲（即执行栈清空）时继续拉取消息队列下一轮消息（next tick，事件循环流转一次称为一次 tick）。
使用代码可以描述如下：

```
var eventLoop = [];
    var event;
    var i = eventLoop.length - 1; //  后进先出

    while(eventLoop[i]) {
        event = eventLoop[i--];
        if (event) { //  事件回调存在
            event();
        }
        //  否则事件消息被丢弃
    }
```

这里注意的一点是等待下一个事件消息的过程是同步的。

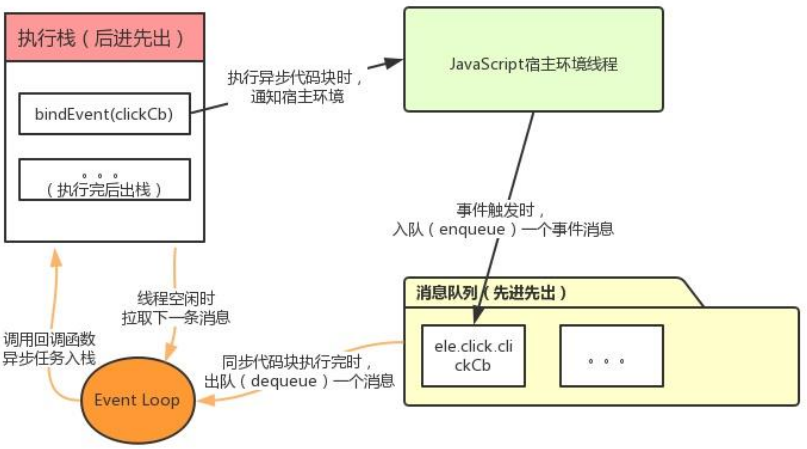
并发模型与事件循环

```
var ele = document.querySelector('body');

    function clickCb(event) {
        console.log('clicked');
    }
    function bindEvent(callback) {
        ele.addEventListener('click', callback);
    }
```

bindEvent(clickCb);

针对如上代码我们可以构建如下并发模型：



如上图，当执行栈同步代码块依次执行完直到遇见异步任务时，异步任务进入等待状态，通知线程，异步事件触发时，往消息队列插入一条事件消息；而当执行栈后续同步代码执行完后，读取消息队列，得到一条消息，然后将该消息对应的异步任务入栈，执行回调函数；一次事件循环就完成了，也即处理了一个异步任务。

JS 中异步有几种？

JS 中异步操作还挺多的，常见的分以下几种：

setTimeout (setInterval)
AJAX
Promise
Generator

setTimeout

```
setTimeout(
   function() {
      console.log("Hello!");
}, 1000);
```

setTimout（setInterval）并不是立即就执行的，这段代码意思是，等 1s 后，把这个 function 加入任务队列中，如果任务队列中没有其他任务了，就执行输出 'Hello'。

```
var outerScopeVar;
helloCatAsync();
alert(outerScopeVar);

function helloCatAsync() {
      setTimeout(function() {
            outerScopeVar = 'hello';
      }, 2000);
}
```

执行上面代码，发现 outerScopeVar 输出是 undefined，而不是 hello。之所以这样是因为在异步代码中返回的一个值是不可能给同步流程中使用的，因为 console.log(outerScopeVar) 是同步代码，执行完后才会执行 setTimout。

```
helloCatAsync(function(result) {
console.log(result);
});

function helloCatAsync(callback) {
      setTimeout(
            function() {
                  callback('hello')
            }
      , 1000)
}
```

把上面代码改成，传递一个 callback，console 输出就会是 hello。

AJAX

```
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
    if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304 ) {
        console.log(xhr.responseText);
    } else {
        console.log( xhr.status);
    }
}
xhr.open('GET', 'url', false);
xhr.send();
```

上面这段代码，xhr.open 中第三个参数默认为 false 异步执行，改为 true 时为同步执行。

Promise

规范简述

promise 是一个拥有 then 方法的对象或函数。
一个 promise 有三种状态 pending, rejected, resolved 状态一旦确定就不能改变，且只能够由 pending 状态变成 rejected 或者 resolved 状态，reject 和 resolved 状态不能相互转换。
当 promise 执行成功时，调用 then 方法的第一个回调函数，失败时调用第二个回调函数。
promise 实例会有一个 then 方法，这个 then 方法必须返回一个新的 promise。

基本用法

```
// 异步操作放在 Promise 构造器中
const promise1 = new Promise((resolve) => {
    setTimeout(() => {
        resolve('hello');
```

```
    }, 1000);
});

// 得到异步结果之后的操作
promise1.then(value => {
  console.log(value, 'world');
}, error =>{
  console.log(error, 'unhappy')
});
```

异步代码，同步写法

```
asyncFun()
    .then(cb)
    .then(cb)
    .then(cb)
```

promise 以这种链式写法，解决了回调函数处理多重异步嵌套带来的回调地狱问题，使代码更加利于阅读，当然本质还是使用回调函数。

异常捕获

前面说过如果在异步的 callback 函数中也有一个异常，那么是捕获不到的，原因就是回调函数是异步执行的。我们看看 promise 是怎么解决这个问题的。

```
asyncFun(1).then(function (value) {
    throw new Error('出错啦');
}, function (value) {
    console.error(value);
}).then(function (value) {

}, function (result) {
  console.log('有错误', result);
});
```

其实是 promise 的 then 方法中，已经自动帮我们 try catch 了这个回调函数，实现大致如下。

```
Promise.prototype.then = function(cb) {
    try {
        cb()
    } catch (e) {
        // todo
        reject(e)
```

```
        }
    }
```

then 方法中抛出的异常会被下一个级联的 then 方法的第二个参数捕获到（前提是有），那么如果最后一个 then 中也有异常怎么办。

```
Promise.prototype.done = function (resolve, reject) {
    this.then(resolve, reject).catch(function (reason) {
        setTimeout(() => {
            throw reason;
        }, 0);
    });
};
 asyncFun(1).then(function (value) {
      throw new Error('then resolve 回调出错啦');
 }).catch(function (error) {
      console.error(error);
      throw new Error('catch 回调出错啦');
 }).done((reslove, reject) => {});
```

我们可以加一个 done 方法，这个方法并不会返回 promise 对象，所以在此之后并不能级联，done 方法最后会把异常抛到全局，这样就可以被全局的异常处理函数捕获或者中断线程。这也是 promise 的一种最佳实践策略，当然这个 done 方法并没有被 ES6 实现，所以我们在不适用第三方 Promise 开源库的情况下就只能自己来实现了。为什么需要这个 done 方法。

```
const asyncFun = function (value) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve(value);
    }, 0);
  })
};


asyncFun(1).then(function (value) {
  throw new Error('then resolve 回调出错啦');
});
```

(node:6312) UnhandledPromiseRejectionWarning: Unhandled promise rejection (rejection id: 1): Error: then resolve 回调出错啦

(node:6312) [DEP0018] DeprecationWarning: Unhandled promise rejections are deprecated. In the future, promise rejections that are not handled will terminate the Node.js process with a non-zero exit code
我们可以看到 JavaScript 线程只是报了一个警告，并没有中止线程，如果是一个严重错误如果不及时中止线程，可能会造成损失。
局限
promise 有一个局限就是不能够中止 promise 链，例如当 promise 链中某一个环节出现错误之后，已经没有了继续往下执行的必要性，但是 promise 并没有提供原生的取消的方式，我们可以看到即使在前面已经抛出异常，但是 promise 链并不会停止。虽然我们可以利用返回一个处于 pending 状态的 promise 来中止 promise 链。

```
const promise1 = new Promise((resolve) => {
    setTimeout(() => {
        resolve('hello');
    }, 1000);
});

promise1.then((value) => {
    throw new Error('出错啦!');
}).then(value => {
    console.log(value);
}, error=> {
    console.log(error.message);
    return result;
}).then(function () {
    console.log('DJL 箫氏');
});
```

特殊场景

当我们的一个任务依赖于多个异步任务，那么我们可以使用 Promise.all
当我们的任务依赖于多个异步任务中的任意一个，至于是谁无所谓，Promise.race

上面所说的都是 ES6 的 promise 实现，实际上功能是比较少，而且还有一些不足的，所以还有很多开源 promise 的实现库，像 q.js 等等，它们提供了更多的语法糖，也有了更多的适应场景。
核心代码

```
var defer = function () {
    var pending = [], value;
    return {
        resolve: function (_value) {
            value = _value;
            for (var i = 0, ii = pending.length; i < ii; i++) {
                var callback = pending[i];
                callback(value);
            }
            pending = undefined;
        },
        then: function (callback) {
            if (pending) {
                pending.push(callback);
            } else {
                callback(value);
            }
        }
    }
};
```

当调用 then 的时候，把所有的回调函数存在一个队列中，当调用 resolve 方法后，依次将队列中的回调函数取出来执行

```
var ref = function (value) {
    if (value && typeof value.then === "function")
        return value;
    return {
        then: function (callback) {
            return ref(callback(value));
        }
    };
};
```

这一段代码实现的级联的功能，采用了递归。如果传递的是一个 promise 那么就会直接返回这个 promise，但是如果传递的是一个值，那么会将这个值包装成一个 promise。

下面 promise 和 ajax 结合例子：

```
function ajax(url) {
    return new Promise(function(resolve, reject) {
```

```
            var xhr = new XMLHttpRequest();
            xhr.onreadystatechange = function() {
                if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304 )
{
                    resovle(xhr.responseText);
                } else {
                    reject( xhr.status);
                }
            }
            xhr.open('GET', url, false);
            xhr.send();
        });
}

ajax('/test.json')
        .then(function(data){
            console.log(data);
        })
        .cacth(function(err){
            console.log(err);
        });
```

generator

基本用法

```
function * gen (x) {
        const y = yield x + 2;
        // console.log(y);    //  猜猜会打印出什么值
}

const g = gen(1);
console.log('first', g.next());    //first { value: 3, done: false }
console.log('second', g.next()); // second { value: undefined, done: true }
```

通俗的理解一下就是 yield 关键字会交出函数的执行权，next 方法会交回执行权，yield 会把 generator 中 yield 后面的执行结果，带到函数外面，而 next 方法会把外面的数据返回给 generator 中 yield 左边的变量。这样就实

现了数据的双向流动。

generator 实现异步编程

我们来看 generator 如何是如何来实现一个异步编程（*）

```
const fs = require('fs');

function * gen() {
    try {
        const file = yield fs.readFile;
        console.log(file.toString());
    } catch(e) {
        console.log('捕获到异常', e);
    }
}

// 执行器
const g = gen();

g.next().value('./config1.json', function (error, value) {
  if (error) {
    g.throw('文件不存在');
  }
  g.next(value);
});
```

那么我们 next 中的参数就会是上一个 yield 函数的返回结果，可以看到在 generator 函数中的代码感觉是同步的，但是要想执行这个看似同步的代码，过程却很复杂，也就是流程管理很复杂。

扩展：异步编程与多线程的区别

共同点：异步和多线程两者都可以达到避免调用线程阻塞的目的，从而提高软件的可响应性

不同点：

（1）线程不是一个计算机硬件的功能，而是操作系统提供的一种

逻辑功能，线程本质上是进程中一段并发运行的代码，所以线程需要操作系统投入 CPU 资源来运行和调度。

多线程的优点很明显，线程中的处理程序依然是顺序执行，符合普通人的思维习惯，所以编程简单。但是多线程的缺点也同样明显，线 程的使用（滥用）会给系统带来上下文切换的额外负担。并且线程间的共享变量可能造成死锁的出现

（2）异步操作无须额外的线程负担，并且使用回调的方式进行处理，在设计良好的情况下，处理函数可以不必使用共享变量（即使无法完 全不用，最起码可以减少 共享变量的数量），减少了死锁的可能。当然异步操作也并非完美无暇。编写异步操作的复杂程度较高，程序 主要使用回调方式进行处理，与普通人的思维方式有些初入，而且难以调试。

这里有一个疑问。异步操作没有创建新的线程，我们一定会想，比如有一个文件操作，大量数据从硬盘上读取，若使用单线程的同步操作自然要等待会很长时间，但是若使用异步操作的话，我们让数据读取异步进行，二线程在数据读取期间去干其他的事情，我们会想，这怎么行呢，异步没有创建其他的线程，一个线程去干其他的事情去了，那数据的读取异步执行是去由谁完成的呢？实际上，本质是这样的。

熟悉电脑硬件的朋友肯定对 DMA 这个词不陌生，硬盘、光驱的技术规格中都有明确 DMA 的模式指标，其实网卡、声卡、显卡也是有 DMA 功能的。DMA 就是直 接内存访问的意思，也就是说，拥有 DMA 功能的硬件在和内存进行数据交换的时候可以不消耗 CPU 资源。只要 CPU 在发起数据传输时发送一个指令，硬件就开 始自己和内存交换数据，在传输完成之后硬件会触发一个中断来通知操作完成。这些无须消耗 CPU 时间的 I/O 操作正是异步操作的硬件基础。所以即使在 DOS 这样的单进程（而且无线程概念）系统中也同样可以发起异步的 DMA 操作。

即 CPU 在数据的长时间读取过程中 ，只需要做两件事，第一发布指令，开始数据交换；第二，交换结束，得到指令，CPU 再进行后续操作。而中间读取数据漫长的等待过程，CPU 本身就不需要参与，顺序执行就是我不参与但是我要干等着，效率低下；异步执行就是，我不需要参与那我就去干其他事情去了，你做完了再通知我就可以了（回调）。

但是你想一下，如果有一些异步操作必须要 CPU 的参与才能完成呢，即我开始的那个线程是走不开的，这该怎么办呢，在.NET 中，有线程池去完成，线程池会高效率的开启一个新的线程去完成异步操作，在 python 中这是系统自己去安排的，无需人工干预，这就比自己创建很多的线程更加高效。

总结：

（1）“多线程”，第一、最大的问题在于线程本身的调度和运行需要很多时间，因此不建议自己创建太大量的线程；第二、共享资源的调度比较难，涉及到死锁，上锁等相关的概念。

（2）“异步” ，异步最大的问题在于“回调”，这增加了软件设计上的难度。

在实际设计时，我们可以将两者结合起来：

（1）当需要执行 I/O 操作时，使用异步操作比使用线程+同步 I/O 操作更合适。I/O 操作不仅包括了直接的文件、网络的读写，还包括数据库操作、Web Service、HttpRequest 以及.net Remoting 等跨进程的调用。异步特别适用于大多数 IO 密集型的应用程序。

（2）而线程的适用范围则是那种需要长时间 CPU 运算的场合，例如耗时较长的图形处理和算法执行。但是往 往由于使用线程编程的简单和符合习惯，所以很多朋友往往会使用线程来执行耗时较长的 I/O 操作。这样在只有少数几个并发操作的时候还无伤大雅，如果需要处 理大量的并发操作时就不合适了。