异步操作概述

单线程模型

单线程模型指的是，JavaScript 只在一个线程上运行。也就是说，JavaScript 同时只能执行一个任务，其他任务都必须在后面排队等待。

注意，JavaScript 只在一个线程上运行，不代表 JavaScript 引擎只有一个线程。事实上，JavaScript 引擎有多个线程，单个脚本只能在一个线程上运行（称为主线程），其他线程都是在后台配合。

JavaScript 之所以采用单线程，而不是多线程，跟历史有关系。JavaScript 从诞生起就是单线程，原因是不想让浏览器变得太复杂，因为多线程需要共享资源、且有可能修改彼此的运行结果，对于一种网页脚本语言来说，这就太复杂了。如果 JavaScript 同时有两个线程，一个线程在网页 DOM 节点上添加内容，另一个线程删除了这个节点，这时浏览器应该以哪个线程为准？是不是还要有锁机制？所以，为了避免复杂性，JavaScript 一开始就是单线程，这已经成了这门语言的核心特征，将来也不会改变。

这种模式的好处是实现起来比较简单，执行环境相对单纯；坏处是只要有一个任务耗时很长，后面的任务都必须排队等着，会拖延整个程序的执行。常见的浏览器无响应（假死），往往就是因为某一段 JavaScript 代码长时间运行（比如死循环），导致整个页面卡在这个地方，其他任务无法执行。JavaScript 语言本身并不慢，慢的是读写外部数据，比如等待 Ajax 请求返回结果。这个时候，如果对方服务器迟迟没有响应，或者网络不通畅，就会导致脚本的长时间停滞。

如果排队是因为计算量大，CPU 忙不过来，倒也算了，但是很多时候 CPU 是闲着的，因为 IO 操作（输入输出）很慢（比如 Ajax 操作从网络读取数据），不得不等着结果出来，再往下执行。JavaScript 语言的设计者意识到，这时 CPU 完全可以不管 IO 操作，挂起处于等待中的任务，先运行排在后面的任务。等到 IO 操作返回了结果，再回过头，把挂起的任务继续执行下去。这种机制就是 JavaScript 内部采用的“事件循环”机制（Event Loop）。

单线程模型虽然对 JavaScript 构成了很大的限制，但也因此使它具备了其他语言不具备的优势。如果用得好，JavaScript 程序是不会出现堵塞的，这就是为什么 Node 可以用很少的资源，应付大流量访问的原因。

为了利用多核 CPU 的计算能力，HTML5 提出 Web Worker 标准，允许 JavaScript 脚本创建多个线程，但是子线程完全受主线程控制，且不得操作 DOM。所以，这个新标准并没有改变 JavaScript 单线程的本质。

同步任务和异步任务

程序里面所有的任务，可以分成两类：同步任务（synchronous）和异步任务（asynchronous）。

同步任务是那些没有被引擎挂起、在主线程上排队执行的任务。只有前一个任务执行完毕，才能执行后一个任务。

异步任务是那些被引擎放在一边，不进入主线程、而进入任务队列的任务。只有引擎认为某个异步任务可以执行了（比如 Ajax 操作从服务器得到了结果），该任务（采用回调函数的形式）才会进入主线程执行。排在异步任务后面的代码，不用等待异步任务结束会马上运行，也就是说，异步任务不具有“堵塞”效应。

举例来说，Ajax 操作可以当作同步任务处理，也可以当作异步任务处理，由开发者决定。如果是同步任务，主线程就等着 Ajax 操作返回结果，再往下执行；如果是异步任务，主线程在发出 Ajax 请求以后，就直接往下执行，等到 Ajax 操作有了结果，主线程再执行对应的回调函数。

任务队列和事件循环

JavaScript 运行时，除了一个正在运行的主线程，引擎还提供一个任务队列（task queue），里面是各种需要当前程序处理的异步任务。（实际上，根据异步任务的类型，存在多个任务队列。为了方便理解，这里假设只存在一个队列。）

首先，主线程会去执行所有的同步任务。等到同步任务全部执行完，就会去看任务队列里面的异步任务。如果满足条件，那么异步任务就重新进入主线程开始执行，这时它就变成同步任务了。等到执行完，下一个异步任务再进入主线程开始执行。一旦任务队列清空，程序就结束执行。

异步任务的写法通常是回调函数。一旦异步任务重新进入主线程，就会执行对应的回调函数。如果一个异步任务没有回调函数，就不会进入任务队列，也就是说，不会重新进入主线程，因为没有用回调函数指定下一步的操作。

JavaScript 引擎怎么知道异步任务有没有结果，能不能进入主线程呢？答案就是引擎在不停地检查，一遍又一遍，只要同步任务执行完了，引擎就会去检查那些挂起来的异步任务，是不是可以进入主线程了。这种循环检查的机制，就叫做事件循环（Event Loop）。[维基百科](http://en.wikipedia.org/wiki/Event_loop)的定义是：“事件循环是一个程序结构，用于等待和发送消息和事件（a programming construct that waits for and dispatches events or messages in a program）”。

异步操作的模式

下面总结一下异步操作的几种模式。

回调函数

回调函数是异步操作最基本的方法。

下面是两个函数`f1`和`f2`，编程的意图是`f2`必须等到`f1`执行完成，才能执行。

```javascript
function f1() {
  // ...
}

function f2() {
  // ...
}

f1();
f2();
```

上面代码的问题在于，如果`f1`是异步操作，`f2`会立即执行，不会等到`f1`结束再执行。

这时，可以考虑改写`f1`，把`f2`写成`f1`的回调函数。

```javascript
function f1(callback) {
  // ...
  callback();
}

function f2() {
  // ...
}

f1(f2);
```

回调函数的优点是简单、容易理解和实现，缺点是不利于代码的阅读和维护，各个部分之间高度[耦合](http://en.wikipedia.org/wiki/Coupling_(computer_programming))（coupling），使得程序结构混乱、流程难以追踪（尤其是多个回调函数嵌套的情况），而且每个任务只能指定一个回调函数。

事件监听

另一种思路是采用事件驱动模式。异步任务的执行不取决于代码的顺序，而取决于某个事件是否发生。

还是以`f1`和`f2`为例。首先，为`f1`绑定一个事件（这里采用的 jQuery 的[写法](http://api.jquery.com/on/)）。

```javascript
f1.on('done', f2);
```

上面这行代码的意思是，当`f1`发生`done`事件，就执行`f2`。然后，对`f1`进行改写：

```javascript
function f1() {
  setTimeout(function () {
    // ...
    f1.trigger('done');
  }, 1000);
}
```

上面代码中，`f1.trigger('done')`表示，执行完成后，立即触发`done`事件，从而开始执行`f2`。

这种方法的优点是比较容易理解，可以绑定多个事件，每个事件可以指定多个回调函数，而且可以“[去耦合](http://en.wikipedia.org/wiki/Decoupling)”（decoupling），有利于实现模块化。缺点是整个程序都要变成事件驱动型，运行流程会变得很不清晰。阅读代码的时候，很难看出主流程。

发布/订阅

事件完全可以理解成“信号”，如果存在一个“信号中心”，某个任务执行完成，就向信号中心“发布”（publish）一个信号，其他任务可以向信号中心“订阅”（subscribe）这个信号，从而知道什么时候自己可以开始执行。这就叫做”[发布/订阅模式](http://en.wikipedia.org/wiki/Publish-subscribe_pattern)”（publish-subscribe pattern），又称“[观察者模式](http://en.wikipedia.org/wiki/Observer_pattern)”（observer pattern）。

这个模式有多种[实现](http://msdn.microsoft.com/en-us/magazine/hh201955.aspx)，下面采用的是 Ben Alman 的 [Tiny Pub/Sub](https://gist.github.com/661855)，这是 jQuery 的一个插件。

首先，`f2`向信号中心`jQuery`订阅`done`信号。

```javascript
jQuery.subscribe('done', f2);
```

然后，`f1`进行如下改写。

```javascript
function f1() {
  setTimeout(function () {
    // ...
    jQuery.publish('done');
  }, 1000);
}
```

上面代码中，`jQuery.publish('done')`的意思是，`f1`执行完成后，向信号中心`jQuery`发布`done`信号，从而引发`f2`的执行。

`f2`完成执行后，可以取消订阅（unsubscribe）。

```javascript
jQuery.unsubscribe('done', f2);
```

这种方法的性质与“事件监听”类似，但是明显优于后者。因为可以通过查看“消息中心”，了解存在多少信号、每个信号有多少订阅者，从而监控程序的运行。

异步操作的流程控制

如果有多个异步操作，就存在一个流程控制的问题：如何确定异步操作执行的顺序，以及如何保证遵守这种顺序。

```javascript
function async(arg, callback) {
  console.log('参数为 ' + arg +' , 1秒后返回结果');
  setTimeout(function () { callback(arg * 2); }, 1000);
}
```

上面代码的`async`函数是一个异步任务，非常耗时，每次执行需要1秒才能完成，然后再调用回调函数。

如果有六个这样的异步任务，需要全部完成后，才能执行最后的`final`函数。请问应该如何安排操作流程?

```javascript
function final(value) {
  console.log('完成: ', value);
}

async(1, function (value) {
  async(2, function (value) {
    async(3, function (value) {
      async(4, function (value) {
        async(5, function (value) {
          async(6, final);
        });
      });
    });
  });
});
// 参数为 1 , 1秒后返回结果
// 参数为 2 , 1秒后返回结果
// 参数为 3 , 1秒后返回结果
// 参数为 4 , 1秒后返回结果
```

```
// 参数为 5 , 1秒后返回结果
// 参数为 6 , 1秒后返回结果
// 完成:  12
```

上面代码中，六个回调函数的嵌套，不仅写起来麻烦，容易出错，而且难以维护。

串行执行

我们可以编写一个流程控制函数，让它来控制异步任务，一个任务完成以后，再执行另一个。这就叫串行执行。

```javascript
var items = [ 1, 2, 3, 4, 5, 6 ];
var results = [];

function async(arg, callback) {
  console.log('参数为 ' + arg +' , 1秒后返回结果');
  setTimeout(function () { callback(arg * 2); }, 1000);
}

function final(value) {
  console.log('完成: ', value);
}

function series(item) {
  if(item) {
    async( item, function(result) {
      results.push(result);
      return series(items.shift());
    });
  } else {
    return final(results[results.length - 1]);
  }
}

series(items.shift());
```

上面代码中，函数`series`就是串行函数，它会依次执行异步任务，所有任务都完成后，才会执行`final`函数。`items`数组保存每一个异步任务的参数，`results`数组保存每一个异步任务的运行结果。

注意，上面的写法需要六秒，才能完成整个脚本。

并行执行

流程控制函数也可以是并行执行，即所有异步任务同时执行，等到全部完成以后，才执行`final`函数。

```javascript
var items = [ 1, 2, 3, 4, 5, 6 ];
var results = [];

function async(arg, callback) {
  console.log('参数为 ' + arg +' , 1秒后返回结果');
  setTimeout(function () { callback(arg * 2); }, 1000);
}

function final(value) {
  console.log('完成: ', value);
}

items.forEach(function(item) {
  async(item, function(result){
    results.push(result);
    if(results.length === items.length) {
      final(results[results.length - 1]);
    }
  })
});
```

上面代码中，`forEach`方法会同时发起六个异步任务，等到它们全部完成以后，才会执行`final`函数。

相比而言，上面的写法只要一秒，就能完成整个脚本。这就是说，并行执行的效率较高，比起串行执行一次只能执行一个任务，较为节约时间。但是问题在于如果并行的任务较多，很容易耗尽系统资源，拖慢运行速度。因此有了第三种流程控制方式。

并行与串行的结合

所谓并行与串行的结合，就是设置一个门槛，每次最多只能并行执行`n`个异步任务，这样就避免了过分占用系统资源。

```javascript
var items = [ 1, 2, 3, 4, 5, 6 ];
var results = [];
var running = 0;
var limit = 2;

function async(arg, callback) {
  console.log('参数为 ' + arg +' , 1秒后返回结果');
  setTimeout(function () { callback(arg * 2); }, 1000);
}
```

```
function final(value) {
  console.log('完成: ', value);
}

function launcher() {
  while(running < limit && items.length > 0) {
    var item = items.shift();
    async(item, function(result) {
      results.push(result);
      running--;
      if(items.length > 0) {
        launcher();
      } else if(running == 0) {
        final(results);
      }
    });
    running++;
  }
}

launcher();
```

上面代码中，最多只能同时运行两个异步任务。变量`running`记录当前正在运行的任务数，只要低于门槛值，就再启动一个新的任务，如果等于`0`，就表示所有任务都执行完了，这时就执行`final`函数。

这段代码需要三秒完成整个脚本，处在串行执行和并行执行之间。通过调节`limit`变量，达到效率和资源的最佳平衡。