网道 Javascript 教程-异步操作

1 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。

优缺点：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。

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。

2 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。

3 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。

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。

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）。

4 异步操作的模式

4.1 回调函数

回调函数是异步操作最基本的方法。

例如，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。

```
function f1(callback){
    //...
    callback(f2);
}

function f2(){
    //...
}
f1(f2);
```

优缺点：

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（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。

4.2 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）。

```
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：

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

优缺点：

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”（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。

4.3 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”（publish-subscribe pattern），又称“观察者模式”（observer pattern）。

这个模式有多种实现，下面采用的是 Ben Alman 的 Tiny Pub/Sub，这是 jQuery 的一个插件。

首先，f2向信号中心jQuery订阅done信号。

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

然后，f1进行如下改写。

```
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）。

```
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。

5 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。

5.1 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。

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

```
function async(arg, callback){
    console.log('参数为' + arg + ', 1s 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。

5.2 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。

```
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);
}

item.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。

5.3 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。

```
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。