异步操作概述

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。

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。

1. 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）。**

2.2 异步操作的模式

2.2.1 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。

```
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。

```
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（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。

2.2.2 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);
```

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);
}
```

`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”（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。

2.2.3 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）。