```
bar():
  a. 把a的值加载到X
  b. 把2保存在Y
  c. 执行X乘Y,结果保存在X
  d. 把X的值保存在a
```

现在，假设两个线程并行执行。你可能已经发现了这个程序的问题，是吧？它们在临时步骤中使用了共享的内存地址 X 和 Y。

如果按照以下步骤执行，最终结果将会是什么样呢？

```
1a  (把a的值加载到X        ==> 20)
2a  (把a的值加载到X        ==> 20)
1b  (把1保存在Y    ==> 1)
2b  (把2保存在Y    ==> 2)
1c  (执行X加Y,结果保存在X            ==> 22)
1d  (把X的值保存在a         ==> 22)
2c  (执行X乘Y,结果保存在X                ==> 44)
2d  (把X的值保存在a         ==> 44)
```

a 的结果将是 44。但如果按照以下顺序执行呢？

```
1a  (把a的值加载到X        ==> 20)
2a  (把a的值加载到X        ==> 20)
2b  (把2保存在Y    ==> 2)
1b  (把1保存在Y    ==> 1)
2c  (执行X乘Y,结果保存在X                ==> 20)
1c  (执行X加Y,结果保存在X            ==> 21)
1d  (把X的值保存在a         ==> 21)
2d  (把X的值保存在a         ==> 21)
```

a 的结果将是 21。

所以，多线程编程是非常复杂的。因为如果不通过特殊的步骤来防止这种中断和交错运行的话，可能会得到出乎意料的、不确定的行为，通常这很让人头疼。

JavaScript 从不跨线程共享数据，这意味着不需要考虑这一层次的不确定性。但是这并不意味着 JavaScript 总是确定性的。回忆一下前面提到的，foo() 和 bar() 的相对顺序改变可能会导致不同结果（41 或 42）。

可能目前还不是很明显，但并不是所有的不确定性都是有害的。这有时无关紧要，但有时又是要刻意追求的结果。关于这一点，本章和后面几章会给出更多示例。

完整运行

由于 JavaScript 的单线程特性，foo()（以及 bar()）中的代码具有原子性。也就是说，一

且 foo() 开始运行，它的所有代码都会在 bar() 中的任意代码运行之前完成，或者相反。这称为*完整运行*（run-to-completion）特性。

实际上，如果 foo() 和 bar() 中的代码更长，完整运行的语义就会更加清晰，比如：

```
var a = 1;
var b = 2;

function foo() {
    a++;
    b = b * a;
    a = b + 3;
}

function bar() {
    b--;
    a = 8 + b;
    b = a * 2;
}

// ajax(..)是某个库中提供的某个Ajax函数
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

由于 foo() 不会被 bar() 中断，bar() 也不会被 foo() 中断，所以这个程序只有两个可能的输出，取决于这两个函数哪个先运行——如果存在多线程，且 foo() 和 bar() 中的语句可以交替运行的话，可能输出的数目将会增加不少！

块 1 是同步的（*现在*运行），而块 2 和块 3 是异步的（*将来*运行），也就是说，它们的运行在时间上是分隔的。

块 1：

```
var a = 1;
var b = 2;
```

块 2（foo()）：

```
a++;
b = b * a;
a = b + 3;
```

块 3（bar()）：

```
b--;
a = 8 + b;
b = a * 2;
```

块 2 和块 3 哪个先运行都有可能，所以如下所示，这个程序有两个可能输出。

输出 1：

```
var a = 1;
var b = 2;

// foo()
a++;
b = b * a;
a = b + 3;

// bar()
b--;
a = 8 + b;
b = a * 2;

a; // 11
b; // 22
```

输出 2：

```
var a = 1;
var b = 2;

// bar()
b--;
a = 8 + b;
b = a * 2;

// foo()
a++;
b = b * a;
a = b + 3;

a; // 183
b; // 180
```

同一段代码有两个可能输出意味着还是存在不确定性！但是，这种不确定性是在函数（事件）顺序级别上，而不是多线程情况下的语句顺序级别（或者说，表达式运算顺序级别）。换句话说，这一确定性要高于多线程情况。

在 JavaScript 的特性中，这种函数顺序的不确定性就是通常所说的*竞态条件*（race condition），`foo()` 和 `bar()` 相互竞争，看谁先运行。具体来说，因为无法可靠预测 `a` 和 `b` 的最终结果，所以才是竞态条件。

如果 JavaScript 中的某个函数由于某种原因不具有完整运行特性，那么可能的结果就会多得多，对吧？实际上，ES6 就引入了这么一个东西（参见第 4 章），现在还不必为此操心，以后还会再探讨这一部分！

1.4 并发

现在让我们来设想一个展示状态更新列表（比如社交网络新闻种子）的网站，其随着用户向下滚动列表而逐渐加载更多内容。要正确地实现这一特性，需要（至少）两个独立的“进程”同时运行（也就是说，是在同一段时间内，并不需要在同一时刻）。

这里的“进程”之所以打上引号，是因为这并不是计算机科学意义上的真正操作系统级进程。这是虚拟进程，或者任务，表示一个逻辑上相关的运算序列。之所以使用“进程”而不是“任务”，是因为从概念上来讲，“进程”的定义更符合这里我们使用的意义。

第一个“进程”在用户向下滚动页面触发 `onscroll` 事件时响应这些事件（发起 Ajax 请求要求新的内容）。第二个“进程”接收 Ajax 响应（把内容展示到页面）。

显然，如果用户滚动页面足够快的话，在等待第一个响应返回并处理的时候可能会看到两个或更多 `onscroll` 事件被触发，因此将得到快速触发彼此交替的 `onscroll` 事件和 Ajax 响应事件。

两个或多个“进程”同时执行就出现了并发，不管组成它们的单个运算是否*并行*执行（在独立的处理器或处理器核心上同时运行）。可以把并发看作“进程”级（或者任务级）的并行，与运算级的并行（不同处理器上的线程）相对。

并发也引出了这些“进程”之间可能的彼此交互的概念。我们会在后面介绍。

在给定的时间窗口内（用户滚动页面的几秒钟内），我们看看把各个独立的“进程”表示为一系列事件 / 运算是什么样的：

“进程”1（`onscroll` 事件）：

```
onscroll, 请求1
onscroll, 请求2
onscroll, 请求3
onscroll, 请求4
onscroll, 请求5
onscroll, 请求6
onscroll, 请求7
```

“进程”2（Ajax 响应事件）：

```
响应1
响应2
响应3
响应4
响应5
响应6
响应7
```

很可能某个 onscroll 事件和某个 Ajax 响应事件恰好同时可以处理。举例来说，假设这些事件的时间线是这样的：

```
onscroll, 请求1
onscroll, 请求2                响应1
onscroll, 请求3                响应2
响应3
onscroll, 请求4
onscroll, 请求5
onscroll, 请求6                响应4
onscroll, 请求7

响应6
响应5
响应7
```

但是，本章前面介绍过事件循环的概念，JavaScript 一次只能处理一个事件，所以要么是 onscroll，请求 2 先发生，要么是响应 1 先发生，但是不会严格地同时发生。这就像学校食堂的孩子们，不管在门外多么拥挤，最终他们都得站成一队才能拿到自己的午饭！

下面列出了事件循环队列中所有这些交替的事件：

```
onscroll, 请求1       <--- 进程1启动
onscroll, 请求2
响应1                 <--- 进程2启动
onscroll, 请求3
响应2
响应3
onscroll, 请求4
onscroll, 请求5
onscroll, 请求6
响应4
onscroll, 请求7       <--- 进程1结束
响应6
响应5
响应7                 <--- 进程2结束
```

“进程”1 和“进程”2 并发运行（任务级并行），但是它们的各个事件是在事件循环队列中依次运行的。

另外，注意到响应 6 和响应 5 的返回是乱序的了吗？

单线程事件循环是并发的一种形式（当然还有其他形式，后面会介绍）。

1.4.1 非交互

两个或多个“进程”在同一个程序内并发地交替运行它们的步骤 / 事件时，如果这些任务彼此不相关，就不一定需要交互。如果进程间没有相互影响的话，不确定性是完全可以接受的。

举例来说：

```
var res = {};

function foo(results) {
    res.foo = results;
}

function bar(results) {
    res.bar = results;
}

// ajax(..)是某个库提供的某个Ajax函数
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

`foo()` 和 `bar()` 是两个并发执行的“进程”，按照什么顺序执行是不确定的。但是，我们构建程序的方式使得无论按哪种顺序执行都无所谓，因为它们是独立运行的，不会相互影响。

这并不是竞态条件 bug，因为不管顺序如何，代码总会正常工作。

1.4.2 交互

更常见的情况是，并发的“进程”需要相互交流，通过作用域或 DOM 间接交互。正如前面介绍的，如果出现这样的交互，就需要对它们的交互进行协调以避免竞态的出现。

下面是一个简单的例子，两个并发的“进程”通过隐含的顺序相互影响，这个顺序有时会被破坏：

```
var res = [];

function response(data) {
    res.push( data );
}

// ajax(..)是某个库中提供的某个Ajax函数
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

这里的并发“进程”是这两个用来处理 Ajax 响应的 `response()` 调用。它们可能以任意顺序运行。

我们假定期望的行为是 res[0] 中放调用 "http://some.url.1" 的结果，res[1] 中放调用 "http://some.url.2" 的结果。有时候可能是这样，但有时候却恰好相反，这要视哪个调用先完成而定。

这种不确定性很有可能就是一个竞态条件 bug。

在这些情况下，你对可能做出的假定要持十分谨慎的态度。比如，开发者可能会观察到对 "http://some.url.2" 的响应速度总是显著慢于对 "http://some.url.1" 的响应，这可能是由它们所执行任务的性质决定的（比如，一个执行数据库任务，而另一个只是获取静态文件），所以观察到的顺序总是符合预期。即使两个请求都发送到同一个服务器，也总会按照固定的顺序响应，但对于响应返回浏览器的顺序，也没有人可以真正保证。

所以，可以协调交互顺序来处理这样的竞态条件：

```
var res = [];

function response(data) {
    if (data.url == "http://some.url.1") {
        res[0] = data;
    }
    else if (data.url == "http://some.url.2") {
        res[1] = data;
    }
}

// ajax(..)是某个库中提供的某个Ajax函数
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

不管哪一个 Ajax 响应先返回，我们都要通过查看 data.url（当然，假定从服务器总会返回一个！）判断应该把响应数据放在 res 数组中的什么位置上。res[0] 总是包含 "http://some.url.1" 的结果，res[1] 总是包含 "http://some.url.2" 的结果。通过简单的协调，就避免了竞态条件引起的不确定性。

从这个场景推出的方法也可以应用于多个并发函数调用通过共享 DOM 彼此之间交互的情况，比如一个函数调用更新某个 的内容，另外一个更新这个 的风格或属性（比如使这个 DOM 元素一有内容就显示出来）。可能你并不想在这个 DOM 元素在拿到内容之前显示出来，所以这种协调必须要保证正确的交互顺序。

有些并发场景如果不做协调，就总是（并非偶尔）会出错。考虑：

```
var a, b;

function foo(x) {
```

```
    a = x * 2;
    baz();
}

function bar(y) {
    b = y * 2;
    baz();
}

function baz() {
    console.log(a + b);
}

// ajax(..)是某个库中的某个Ajax函数
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

在这个例子中，无论 foo() 和 bar() 哪一个先被触发，总会使 baz() 过早运行（a 或者 b 仍处于未定义状态）；但对 baz() 的第二次调用就没有问题，因为这时候 a 和 b 都已经可用了。

要解决这个问题有多种方法。这里给出了一种简单方法：

```
var a, b;

function foo(x) {
    a = x * 2;
    if (a && b) {
        baz();
    }
}

function bar(y) {
    b = y * 2;
    if (a && b) {
        baz();
    }
}

function baz() {
    console.log( a + b );
}

// ajax(..)是某个库中的某个Ajax函数
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

包裹 baz() 调用的条件判断 if (a && b) 传统上称为门（gate），我们虽然不能确定 a 和 b 到达的顺序，但是会等到它们两个都准备好再进一步打开门（调用 baz()）。

另一种可能遇到的并发交互条件有时称为竞态（race），但是更精确的叫法是门闩（latch）。它的特性可以描述为“只有第一名取胜”。在这里，不确定性是可以接受的，因为它明确指出了这一点是可以接受的：需要“竞争”到终点，且只有唯一的胜利者。

请思考下面这段有问题的代码：

```
var a;

function foo(x) {
    a = x * 2;
    baz();
}

function bar(x) {
    a = x / 2;
    baz();
}

function baz() {
    console.log( a );
}

// ajax(..)是某个库中的某个Ajax函数
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

不管哪一个（foo() 或 bar()）后被触发，都不仅会覆盖另外一个给 a 赋的值，也会重复调用 baz()（很可能并不是想要的结果）。

所以，可以通过一个简单的门闩协调这个交互过程，只让第一个通过：

```
var a;

function foo(x) {
    if (!a) {
        a = x * 2;
        baz();
    }
}

function bar(x) {
    if (!a) {
        a = x / 2;
        baz();
    }
}

function baz() {
    console.log( a );
}

// ajax(..)是某个库中的某个Ajax函数
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

条件判断 if (!a) 使得只有 foo() 和 bar() 中的第一个可以通过，第二个（实际上是任何后续的）调用会被忽略。也就是说，第二名没有任何意义！

出于简化演示的目的，在所有这些场景中，我们一直都使用了全局变量，但这对于此处的论证完全不是必需的。只要相关的函数（通过作用域）能够访问到这些变量，就会按照预期工作。依赖于词法作用域变量（参见本系列的《你不知道的 JavaScript（上卷）》的“作用域和闭包”部分），实际上前面例子中那样的全局变量，对于这些类别的并发协调是一个明显的负面因素。随着后面几章内容的展开，我们会看到还有其他种类的更清晰的协调方式。

1.4.3 协作

还有一种并发合作方式，称为*并发协作*（cooperative concurrency）。这里的重点不再是通过共享作用域中的值进行交互（尽管显然这也是允许的！）。这里的目标是取到一个长期运行的“进程”，并将其分割成多个步骤或多批任务，使得其他并发“进程”有机会将自己的运算插入到事件循环队列中交替运行。

举例来说，考虑一个需要遍历很长的结果列表进行值转换的 Ajax 响应处理函数。我们会使用 `Array#map(..)` 让代码更简洁：

```
var res = [];

// response(..)从Ajax调用中取得结果数组
function response(data) {
    // 添加到已有的res数组
    res = res.concat(
        // 创建一个新的变换数组把所有data值加倍
        data.map( function(val){
            return val * 2;
        } )
    );
}

// ajax(..)是某个库中提供的某个Ajax函数
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

如果 `"http://some.url.1"` 首先取得结果，那么整个列表会立刻映射到 `res` 中。如果记录有几千条或更少，这不算什么。但是如果有像 1000 万条记录的话，就可能需要运行相当一段时间了（在高性能笔记本上需要几秒钟，在移动设备上需要更长时间，等等）。

这样的“进程”运行时，页面上的其他代码都不能运行，包括不能有其他的 `response(..)` 调用或 UI 刷新，甚至是像滚动、输入、按钮点击这样的用户事件。这是相当痛苦的。

所以，要创建一个协作性更强更友好且不会霸占事件循环队列的并发系统，你可以异步地批处理这些结果。每次处理之后返回事件循环，让其他等待事件有机会运行。

这里给出一种非常简单的方法：

```
var res = [];

// response(..)从Ajax调用中取得结果数组
function response(data) {
    // 一次处理1000个
    var chunk = data.splice( 0, 1000 );

    // 添加到已有的res组
    res = res.concat(
        // 创建一个新的数组把chunk中所有值加倍
        chunk.map( function(val){
            return val * 2;
        } )
    );

    // 还有剩下的需要处理吗?
    if (data.length > 0) {
        // 异步调度下一次批处理
        setTimeout( function(){
            response( data );
        }, 0 );
    }
}

// ajax(..)是某个库中提供的某个Ajax函数
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

我们把数据集合放在最多包含 1000 条项目的块中。这样，我们就确保了“进程”运行时间会很短，即使这意味着需要更多的后续“进程”，因为事件循环队列的交替运行会提高站点 /App 的响应（性能）。

当然，我们并没有协调这些“进程”的顺序，所以结果的顺序是不可预测的。如果需要排序的话，就要使用和前面提到类似的交互技术，或者本书后面章节将要介绍的技术。

这里使用 `setTimeout(..0)`（hack）进行异步调度，基本上它的意思就是“把这个函数插入到当前事件循环队列的结尾处”。

严格说来，`setTimeout(..0)` 并不直接把项目插入到事件循环队列。定时器会在有机会的时候插入事件。举例来说，两个连续的 `setTimeout(..0)` 调用不能保证会严格按照调用顺序处理，所以各种情况都有可能出现，比如定时器漂移，在这种情况下，这些事件的顺序就不可预测。在 Node.js 中，类似的方法是 `process.nextTick(..)`。尽管它们使用方便（通常性能也更高），但并没有（至少到目前为止）直接的方法可以适应所有环境来确保异步事件的顺序。下一小节我们会深入讨论这个话题。

1.5 任务

在 ES6 中，有一个新的概念建立在事件循环队列之上，叫作*任务队列*（job queue）。这个概念给大家带来的最大影响可能是 Promise 的异步特性（参见第 3 章）。

遗憾的是，目前为止，这是一个没有公开 API 的机制，因此要展示清楚有些困难。所以我们目前只从概念上进行描述，等到第 3 章讨论 Promise 的异步特性时，你就会理解这些动作是如何协调和处理的。

因此，我认为对于任务队列最好的理解方式就是，它是挂在事件循环队列的每个 tick 之后的一个队列。在事件循环的每个 tick 中，可能出现的异步动作不会导致一个完整的新事件添加到事件循环队列中，而会在当前 tick 的任务队列末尾添加一个项目（一个任务）。

这就像是在说："哦，这里还有一件事将来要做，但要确保在其他任何事情发生之前就完成它。"

事件循环队列类似于一个游乐园游戏：玩过了一个游戏之后，你需要重新到队尾排队才能再玩一次。而任务队列类似于玩过了游戏之后，插队接着继续玩。

一个任务可能引起更多任务被添加到同一个队列末尾。所以，理论上说，*任务循环*（job loop）可能无限循环（一个任务总是添加另一个任务，以此类推），进而导致程序的饿死，无法转移到下一个事件循环 tick。从概念上看，这和代码中的无限循环（就像 `while(true)..`）的体验几乎是一样的。

任务和 `setTimeout(..0)` hack 的思路类似，但是其实现方式的定义更加良好，对顺序的保证性更强：尽可能早的将来。

设想一个调度任务（直接地，不要 hack）的 API，称其为 `schedule(..)`。考虑：

```
console.log( "A" );

setTimeout( function(){
    console.log( "B" );
}, 0 );

// 理论上的"任务API"
schedule( function(){
    console.log( "C" );

    schedule( function(){
        console.log( "D" );
    } );
} );
```

可能你认为这里会打印出 `A B C D`，但实际打印的结果是 `A C D B`。因为任务处理是在当前事件循环 tick 结尾处，且定时器触发是为了调度下一个事件循环 tick（如果可用的话！）。

在第 3 章中，我们将会看到，Promise 的异步特性是基于任务的，所以一定要清楚它和事件循环特性的关系。

1.6 语句顺序

代码中语句的顺序和 JavaScript 引擎执行语句的顺序并不一定要一致。这个陈述可能看起来似乎会很奇怪，所以我们要简单解释一下。

但在此之前，以下这一点我们应该完全清楚：这门语言的规则和语法（参见本系列的《你不知道的 JavaScript（上卷）》的“作用域和闭包”部分）已经从程序的角度在语序方面规定了可预测和非常可靠的特性。所以，接下来我们要讨论的内容你应该无法在自己的 JavaScript 程序中观察到。

如果你观察到了类似于我们将要展示的编译器对语句的重排序，那么这很明显违反了规范，而这一定是由所使用的 JavaScript 引擎中的 bug 引起的——该 bug 应该被报告和修正！但是更可能的情况是，当你怀疑 JavaScript 引擎做了什么疯狂的事情时，实际上却是你自己代码中的 bug（可能是竞态条件）引起的。所以首先要检查自己的代码，并且要反复检查。通过使用断点和单步执行一行一行地遍历代码，JavaScript 调试器就是用来发现这样 bug 的最强大工具。

考虑：

```
var a, b;

a = 10;
b = 30;

a = a + 1;
b = b + 1;

console.log( a + b ); // 42
```

这段代码中没有显式的异步（除了前面介绍过的很少见的异步 I/O！），所以很可能它的执行过程是从上到下一行行进行的。

但是，JavaScript 引擎在编译这段代码之后（是的，JavaScript 是需要编译的，参见本系列的《你不知道的 JavaScript（上卷）》的“作用域和闭包”部分！）可能会发现通过（安全地）重新安排这些语句的顺序有可能提高执行速度。重点是，只要这个重新排序是不可见的，一切都没问题。

比如，引擎可能会发现，其实这样执行会更快：

```
var a, b;

a = 10;
a++;

b = 30;
b++;

console.log( a + b ); // 42
```

或者这样：

```
var a, b;

a = 11;
b = 31;

console.log( a + b ); // 42
```

或者甚至这样：

```
// 因为a和b不会被再次使用
// 我们可以inline,从而完全不需要它们!
console.log( 42 ); // 42
```

前面的所有情况中，JavaScript 引擎在编译期间执行的都是安全的优化，最后可见的结果都是一样的。

但是这里有一种场景，其中特定的优化是不安全的，因此也是不允许的（当然，不用说这其实也根本不能称为优化）：

```
var a, b;

a = 10;
b = 30;

// 我们需要a和b处于递增之前的状态!
console.log( a * b ); // 300

a = a + 1;
b = b + 1;

console.log( a + b ); // 42
```

还有其他一些例子，其中编译器重新排序会产生可见的副作用（因此必须禁止），比如会产生副作用的函数调用（特别是 getter 函数），或 ES6 代理对象（参考本系列的《你不知道的 JavaScript（下卷）》的“ES6 & Beyond”部分）。

考虑：

```
function foo() {
    console.log( b );
    return 1;
}

var a, b, c;

// ES5.1 getter字面量语法
c = {
    get bar() {
        console.log( a );
        return 1;
    }
};

a = 10;
b = 30;

a += foo();             // 30
b += c.bar;             // 11

console.log( a + b );   // 42
```

如果不是因为代码片段中的语句 `console.log(..)`（只是作为一种方便的形式说明可见的副作用），JavaScript 引擎如果愿意的话，本来可以自由地把代码重新排序如下：

```
// ...

a = 10 + foo();
b = 30 + c.bar;

// ...
```

尽管 JavaScript 语义让我们不会见到编译器语句重排序可能导致的噩梦，这是一种幸运，但是代码编写的方式（从上到下的模式）和编译后执行的方式之间的联系非常脆弱，理解这一点也非常重要。

编译器语句重排序几乎就是并发和交互的微型隐喻。作为一个一般性的概念，清楚这一点能够使你更好地理解异步 JavaScript 代码流问题。

1.7 小结

实际上，JavaScript 程序总是至少分为两个块：第一块*现在*运行；下一块*将来*运行，以响应某个事件。尽管程序是一块一块执行的，但是所有这些块共享对程序作用域和状态的访问，所以对状态的修改都是在之前累积的修改之上进行的。

一旦有事件需要运行，事件循环就会运行，直到队列清空。事件循环的每一轮称为一个 tick。用户交互、IO 和定时器会向事件队列中加入事件。