生成器究竟能做些什么

Generator 生成器，一个晦涩难懂的概念，生成器是什么，为什么被称为生成器，而又什么是生成器？最最关心的问题是，为什么需要生成器，他到底能干吗的？带着这样一个有一个的问题，来读下面的内容。

什么是生成器，生成器 OR 状态机

生成器是一个状态机，说了好像没说一样，那什么是状态机呢，为什么要使用他呢？
场景及问题背景：
我们平时开发时本质上就是对应用程序的各种状态进行切换并作出相应处理。最直接的解决方案是将这些所有可能发生的情况全都考虑到，然后使用 if… ellse 语句来做状态判断来进行不同情况的处理。但是对复杂状态的判断就显得代码逻辑特别的乱。随着增加新的状态或者修改一个状态，if else 或 switch case 语句就要相应的的增多或者修改，程序的可读性，扩展性就会变得很弱。维护也会很麻烦。
来个例子来，魂斗罗啊有木有玩过？：
先来看看最简单的一个动作的简单实现：

```
class Contra {
  constructor () {
    //存储当前待执行的动作
    this.lastAct = {};
  }
  //执行动作
  contraGo (act){
    if(act === 'up'){
      //向上跳
    }else if(act === 'forward'){
      //向前冲啊
    }else if(act === 'backward'){
      //往老家跑
    }else if(act === 'down'){
```

```
        //趴下
      }else if(act === 'shoot'){
        //开枪
      }
      this.lastAct = act;
    }
};
var littlered = new Contra();
littlered.contraGo('shoot');
```

那要是有两个组合动作呢？改一下：

```
function contraGo (act){

    constructor () {
      //存储当前待执行的动作
      this.lastAct1 = "";
      this.lastAct2 = "";
    }

    contraGo (act1, act2){
      const actArr = [act1, act2];
      if(actArr.indexOf('shoot') !== -1 &&    actArr.indexOf('up') !== -1){
        //跳着开枪吧
      }else if(actArr.indexOf('shoot') !== -1 &&    actArr.indexOf('forward') !== -
1){
        //向前跑着开枪吧
      }else if(actArr.indexOf('shoot') !== -1 &&    actArr.indexOf('down') !== -1){
        //趴着开枪吧
      }else if(actArr.indexOf('shoot') !== -1 &&    actArr.indexOf('backward') !==
-1){
        //回头跑着开枪吧
      }else if(actArr.indexOf('up') !== -1 &&    actArr.indexOf('forward') !== -1){
        //向前跳吧
      }else if(actArr.indexOf('up') !== -1 &&    actArr.indexOf('down') !== -1){
        //上上下下吧
      }
```

```
    ...//等等组合
        this.lastAct1 = act1;
        this.lastAct2 = act2;
    }
}
var littlered = new Contra();
littlered.contraGo('shoot','up');
```

缺点很明显了，大量的 if else 判断，加入哪天要给小红小蓝加一个回眸的动作，好嘛我又要修改 contraGo 方法，加一堆排列组合了，这使得 contraGo 成为了一个非常不稳定的方法，而且状态越多越庞大，升华一下，contraGo 方法是违反开放-封闭原则的！

然后“状态模式”下凡来救人…
这里插一句开放封闭原则，到下一个分割线结束，大家可以跳一跳
有什么痛点 —— 开发过程中，因为变化、升级和维护等原因需要对原有逻辑进行修改时，很有可能会给旧代码中引入错误，也可能会使我们不得不对整个功能进行重构，并且需要原有功能新测试。

怎么解决 —— 我们应该尽量通过扩展实体的行为来实现变化，而不是通过修改已有的代码来实现变化

具体一点呢 —— 类、模块和函数应该对扩展开放，对修改关闭。模块应该尽量在不修改原代码的情况下进行扩展。

核心 —— 用抽象构建框架，用实现扩展细节。

总结一下 —— 开发人员应该对程序中呈现的频繁变化的那些部分作出抽象，然后从抽象派生的实现类来进行扩展，当代码发生变化时，只需要根据需求重新开发一个实现类来就可以了。要求我们对需求的变更有一定的前瞻性和预见性，同时拒绝对于应用程序中的每个部分都刻意的进行抽象。

包含开闭原则在内，设计模式的六大原则，这里不详细介绍，简单列下：

单一原则 （SRP）：实现类要职责单一,一个类只做一件事或者一类事，不要将功能无法划分为一类的揉到一起，答应我好吗
里氏替换原则（LSP）： 不要破坏继承体系，子类可以完全替换掉他们所

继承的父类，可以理解为调用父类方法的地方换成子类也可以正常执行调用，爸爸打下的江山儿子继位得无压力好吗
依赖倒置原则（DIP）:我说下我的理解，如果某套功能或者业务逻辑可能之后会出现并行的另外一种模式或者较大的调整，那不如把这部分逻辑抽象出来，创建一个包含相关方法的抽象类，而实现类继承这个抽象类来重写抽象类中的方法，完成具体的实现，调用这些功能方法的类不需要关心自己调用的这些个方法的具体实现，只管调用这些抽象类中定义好的形式上的方法即可，不与实际实现这些方法的类发生直接依赖关系，方便之后的实现逻辑的替换更改；
接口隔离原则(ISP)：在设计抽象类的时候要精简单一,白话说就是，A 需要依赖 B 提供的一些方法，A 我只用 B 的 3 个方法，B 就尽量不要给 A 用不到的方法啦；
迪米特法则（LoD）降低耦合,尽量减少对象之间的直接的交互，如果其中一个类需要调用另一个类的某一个方法的话，可通过一个关系类发起这个调用，这样一个模块修改时，就可以最大程度的减少波及。
开放-封闭原则（OCP）告诉我们要对扩展开放，对修改关闭，你可以继承扩展我所有的能力，到你手里你想咋改咋改，但是，别 动我 本人 好吗？好的
23 种设计模式基于以上 6 大基本原则结合具体开发实践总结出来的，万变不离其宗，有了这些基本的意识规范，你写出来的，搞不好就是某种设计模式

解决方案
状态模式：允许一个对象在其内部状态改变的时候改变它的行为，对象看起来似乎修改了它的类。

先看下采用状态模式修改后的代码：

```
class Contra {
  constructor () {
    //存储当前待执行的动作 们
    this._currentstate = {};
  }
  //添加动作
  changeState (){
    //清空当前的动作集合
    this._currentstate = {};
```

```
        //遍历添加动作
        Object.keys(arguments).forEach(
            (i) => this._currentstate[arguments[i]] = true
        )
        return this;
    }
    //执行动作
    contraGo (){
        //当前动作集合中的动作依次执行
        Object.keys(this._currentstate).forEach(
            (k) => Actions[k] && Actions[k].apply(this)
        )
        return this;
    }
};

const Actions = {
    up : function(){
        //向上跳
        console.log('up');
    },
    down : function(){
        //趴下
        console.log('down');
    },
    forward : function(){
        //向前跑
        console.log('forward');
    },
    backward : function(){
        //往老家跑
        console.log('backward');
    },
    shoot : function(){
        //开枪吧
        console.log('shoot');
    },
```

```
};
var littlered = new Contra();
littlered.changeState('shoot','up').contraGo();
```

控制台会输出： shoot up

解决思路：
状态模式，将条件判断的结果转化为状态对象内部的状态（代码中的up,down,backward,forward），内部状态通常作为状态对象内部的私有变量（this._currentState），然后提供一个能够调用状态对象内部状态的接口方法对象(changeState,contraGo)，这样对状态的改变，对状态方法的调用的修改和增加也会很容易，方便了对状态对象中内部状态的管理。

同时，状态模式将每一个条件分支放入一个独立的类中，也就是代码中的Actions。这使得你可以根据对象自身的情况将对象的状态(动作——up,down,backward,forward)作为一个对象(Actions.up，Actions.down 这样)，这一对象可以不依赖于其他对象而独立变化（一个行为一个动作，互不干扰）。

可以看出，状态模式就是一种适合多种状态场景下的设计模式，改写之后代码更加清晰，提高代码的维护性和扩展性，不用再牵一发动全身

状态模式的使用场景：
一个由一个或多个动态变化的属性导致发生不同行为的对象，在与外部事件产生互动时，其内部状态就会改变，从而使得系统的行为也随之发生变化，那么这个对象，就是有状态的对象
代码中包含大量与对象状态有关的条件语句，像是 if else 或 switch case 语句，且这些条件执行与否依赖于该对象的状态。
如果场景符合上面两个条件，那我们就可以想象状态模式是不是可以帮忙了

状态模式的优缺点：
优点:

一个状态状态对应行为，封装在一个类里，更直观清晰，增改方便
状态与状态间，行为与行为间彼此独立互不干扰
避免事物对象本身不断膨胀，条件判断语句过多

每次执行动作不用走很多不必要的判断语句，用哪个拿哪个
缺点:

需要将事物的不同状态以及对应的行为拆分出来，有时候会无法避免的拆分的很细，有的时候涉及业务逻辑，一个动作拆分出对应的两个状态，动作就拆不明白了，过度设计
必然会增加事物类和动作类的个数，有时候动作类再根据单一原则，按照功能拆成几个类，会反而使得代码混乱，可读性降低
状态模式场景实例
组件开发中的状态模式——导航

我们平时开发组件时很多时候要切换组件的状态，每种状态有不同的处理方式，这个时候就可以使用状态模式进行开发。

用和上面一样的思路，我们来举一个 React 组件的小例子，比如一个 Banner 导航，最基本的两种状态，显示和隐藏，如下：

```
const States = {
  "show": function () {
    console.log("banner 展现状态，点击关闭");
    this.setState({
      currentState: "hide"
    })
  },
  "hide": function () {
    console.log("banner 关闭状态，点击展现");
    this.setState({
      currentState: "show"
    })
  }
};
```

同样通过一个对象 States 来定义 banner 的状态，这里有两种状态 show 和 hide，分别拥有相应的处理方法，调用后再分别把当前 banner 改写为另外一种状态。接下来来看导航类 Banner：

```
class Banner extends Component{
```

```
    constructor(props) {
        super(props);
        this.state = {
            currentState: "hide"
        }
        this.toggle = this.toggle.bind(this);
    }
    toggle () {
        const s = this.state.currentState;
        States[s] && States[s].apply(this);
    }

    render() {
        const { currentState } = this.state;
        return (
            <div className="banner" onClick={this.toggle}>
            </div>
        );
    }
};

export default Banner;
```

这个导航类有一个 toggle 方法用来切换状态，然后调用相应的处理方法。

如果有第三种状态，我们只需要 States 添加相应的状态和处理程序即可。

经典示例——红绿灯

红绿灯的基本功能大家都懂，这里直接贴代码，实现思路一样，再换个写法：

```
var trafficLight = (function () {
    var currentLight = null;
    return {
        change: function (light) {
            currentLight = light;
```

```
            currentLight.go();
        }
    }
})();

function RedLight() { }
RedLight.prototype.go = function () {
    console.log("this is red light");
}
function GreenLight() { }
GreenLight.prototype.go = function () {
    console.log("this is green light");
}
function YellowLight() { }
YellowLight.prototype.go = function () {
    console.log("this is yellow light");
}

trafficLight.change(new RedLight());
trafficLight.change(new YellowLight());
```

trafficLight 是一个红绿灯的实例，传入一个构造函数，对象暴露 change 方法改变内部状态，也就是灯的颜色，change 接收的同样是一个状态的对象，调用对象的方法触发响应的动作，这里的动作都叫 go,不同颜色的灯对象有着不同的 go 的实现。

通过传入灯对象到 change 方法中，从而改变红绿灯的状态，触发其相应的处理程序，这就是一个典型的状态模式的应用。

上面三个例子我们可以感觉到，采用状态模式的代码，代码结构都差不多，功能说白了也差不多，都是一种状态对应一种操作，然后可能会改变对象的状态。

这些场景其实可以用一个通用模型来表示，就是“有限状态机”。

有限状态机 Finite-state machine
就是状态模式的一个模型，在日常开发中很多具有多种状态的对象，都可以用有限状态机模型来模拟，一般都具有以下特点：

事物拥有多种状态，任一时间只会处于一种状态不会处于多种状态；
动作可以改变事物状态，一个动作可以通过条件判断，改变事物到不同的状态，但是不能同时指向多个状态，一个时间，就一个状态，就，一个。
（这里有人可能会疑问上面的魂斗罗例子，小红和小蓝为什么可以同时跳着打枪或者跑着打枪，这不就是同时处于两种状态吗？我说下我的理解哈，我认为两个行为“跳”和“打枪”共同决定了一种状态“跳着打枪”，人物一共有的状态其实是可能同时发生的行为的排列组合，组合不排列吧，因为跳着打枪和打枪跳着应该是一样的，所以还是一种状态，一种）
状态总数是有限的；
既然符合上述场景的都可以用状态机模型描述，那么我们是不是可以将这种模式抽象出来，通过传入参数来实现不同的状态流程，不用每次都把这个流程写这么一遍呢？

说完状态机来了解一下 Generator

根据以上的这种设计模式的经验总结，根据状态模式抽象出状态机，在 ECMAScript 中引入的状态机——Generator，翻译过来叫做生成器。
什么是 JavaScript 生成器？
生成器是一种可以用来控制迭代器（iterator）的函数，它可以随时暂停，并可以在任意时候恢复。
上面的描述没法说明什么，让我们来看一些例子，解释什么是生成器，以及生成器与 for 循环之类的迭代器有什么区别。
下面是一个 for 循环的例子，它会在执行后立刻返回一些值。这段代码其实就是简单地生成了 0-5 这些数字。

```
for (let i = 0; i < 5; i += 1) {
  console.log(i);
}
// 它将会立刻返回 0 -> 1 -> 2 -> 3 -> 4
```

现在看看生成器函数。

```
function * generatorForLoop(num) {
  for (let i = 0; i < num; i += 1) {
    yield console.log(i);
  }
}
```

```
const genForLoop = generatorForLoop(5);

genForLoop.next(); //  首先  console.log - 0
genForLoop.next(); // 1
genForLoop.next(); // 2
genForLoop.next(); // 3
genForLoop.next(); // 4
```

它做了什么？它实际上只是对上面例子中的 for 循环做了一点改动，但产生了很大的变化。这种变化是由生成器最重要的特性造成的 —— 只有在需要的时候它才会产生下一个值，而不会一次性产生所有的值。在某些情景下，这种特性十分方便。

生成器语法

如何定义一个生成器函数呢？下面列出了各种可行的定义方法，不过万变不离其宗的是在函数关键词后加上一个星号。

```
function * generator () {}
function* generator () {}
function *generator () {}

let generator = function * () {}
let generator = function* () {}
let generator = function *() {}

let generator = *() => {} // SyntaxError
let generator = ()* => {} // SyntaxError
let generator = (*) => {} // SyntaxError
```

如上面的例子所示，我们并不能使用箭头函数来创建一个生成器。

下面将生成器作为方法（method）来创建。定义方法与定义函数的方式是一样的。

```
class MyClass {
   *generator() {}
   * generator() {}
}

const obj = {
   *generator() {}
   * generator() {}
```

```
}
```

yield

现在让我们一起看看新的关键词 yield。它有些类似 return，但又不完全相同。return 会在完成函数调用后简单地将值返回，在 return 语句之后你无法进行任何操作。

```
function withReturn(a) {
  let b = 5;
  return a + b;
  b = 6; // 不可能重新定义 b 了
  return a * b; // 这儿新的值没可能返回了
}

withReturn(6); // 11
withReturn(6); // 11
```

而 yield 的工作方式却不同。

```
function * withYield(a) {
  let b = 5;
  yield a + b;
  b = 6; // 在第一次调用后仍可以重新定义变量
  yield a * b;
}

const calcSix = withYield(6);

calcSix.next().value; // 11
calcSix.next().value; // 36
```

用 yield 返回的值只会返回一次，当你再次调用同一个函数的时候，它会执行至下一个 yield 语句处（译者注：前面的 yield 不再返回东西了）。在生成器中，我们通常会在输出时得到一个对象。这个对象有两个属性：value 与 done。如你所想，value 为返回值，done 则会显示生成器是否完成了它的工作。

```
function * generator() {
  yield 5;
}

const gen = generator();
```

```
gen.next(); // {value: 5, done: false}
gen.next(); // {value: undefined, done: true}
gen.next(); // {value: undefined, done: true}
```

- 之后的任何调用都会返回相同的结果

在生成器中，不仅可以使用 yield，也可以使用 return 来返回同样的对象。但是，在函数执行到第一个 return 语句的时候，生成器将结束它的工作。

```
function * generator() {
  yield 1;
  return 2;
  yield 3; // 到不了这个 yield 了
}

const gen = generator();

gen.next(); // {value: 1, done: false}
gen.next(); // {value: 2, done: true}
gen.next(); // {value: undefined, done: true}
```

yield 委托迭代

带星号的 yield 可以将它的工作委托给另一个生成器。通过这种方式，你就能将多个生成器连接在一起。

```
function * anotherGenerator(i) {
  yield i + 1;
  yield i + 2;
  yield i + 3;
}

function * generator(i) {
  yield* anotherGenerator(i);
}

var gen = generator(1);

gen.next().value; // 2
gen.next().value; // 3
gen.next().value; // 4
```

在开始下一节前，我们先观察一个第一眼看上去比较奇特的行为。

下面是正常的代码，不会报出任何错误，这表明 yield 可以在 next() 方法调用后返回传递的值：

```
function * generator(arr) {
   for (const i in arr) {
      yield i;
      yield yield;
      yield(yield);
   }
}

const gen = generator([0,1]);

gen.next(); // {value: "0", done: false}
gen.next('A'); // {value: undefined, done: false}
gen.next('A'); // {value: "A", done: false}
gen.next('A'); // {value: undefined, done: false}
gen.next('A'); // {value: "A", done: false}
gen.next(); // {value: "1", done: false}
gen.next('B'); // {value: undefined, done: false}
gen.next('B'); // {value: "B", done: false}
gen.next('B'); // {value: undefined, done: false}
gen.next('B'); // {value: "B", done: false}
gen.next(); // {value: undefined, done: true}
```

在这个例子中，你可以看到 yield 默认是 undefined，但如果我们在调用 yield 时传递了任何值，它就会返回我们传入的值。我们将很快利用这个特性。

初始化与方法

生成器是可以被复用的，但是你需要对它们进行初始化。还好初始化的方法十分简单。

```
function * generator(arg = 'Nothing') {
   yield arg;
}

const gen0 = generator(); // OK
const gen1 = generator('Hello'); // OK
const gen2 = new generator(); // 不 OK
```

generator().next(); // 可以运行，但每次都会从头开始运行
如上所示，gen0 与 gen1 不会互相影响，gen2 完全不会运行（会报错）。因此初始化对于保证程序流程的状态是十分重要的。
下面让我们一起看看生成器给我们提供的方法。
next() 方法

```
function * generator() {
   yield 1;
   yield 2;
   yield 3;
}

const gen = generator();

gen.next(); // {value: 1, done: false}
gen.next(); // {value: 2, done: false}
gen.next(); // {value: 3, done: false}
```

gen.next(); // {value: undefined, done: true} 之后所有的 next 调用都会返回同样的输出
这是最常用的方法。它每次被调用时都会返回下一个对象。在生成器工作结束时，next() 会将 done 属性设为 true，value 属性设为 undefined。
我们不仅可以用 next() 来迭代生成器，还可以用 for of 循环来一次得到生成器所有的值（而不是对象）。

```
function * generator(arr) {
   for (const el in arr)
      yield el;
}

const gen = generator([0, 1, 2]);

for (const g of gen) {
   console.log(g); // 0 -> 1 -> 2
}

gen.next(); // {value: undefined, done: true}
```

但它不适用于 for in 循环，并且不能直接用数字下标来访问属性：generator[0] = undefined。
return() 方法

```
function * generator() {
   yield 1;
   yield 2;
   yield 3;
}

const gen = generator();

gen.return(); // {value: undefined, done: true}
gen.return('Heeyyaa'); // {value: "Heeyyaa", done: true}
```

gen.next(); // {value: undefined, done: true} - 在 return() 之后的所有 next() 调用都会返回相同的输出

return() 将会忽略生成器中的任何代码。它会根据传值设定 value，并将 done 设为 true。任何在 return() 之后进行的 next() 调用都会返回 done 属性为 true 的对象。

throw() 方法

```
function * generator() {
   yield 1;
   yield 2;
   yield 3;
}

const gen = generator();

gen.throw('Something bad'); // 会报错 Error Uncaught Something bad
gen.next(); // {value: undefined, done: true}
```

throw() 做的事非常简单 —— 就是抛出错误。我们可以用 try-catch 来处理。

自定义方法的实现

由于我们无法直接访问 Generator 的 constructor，因此如何增加新的方法需要另外说明。下面是我的方法，你也可以用不同的方式实现：

```
function * generator() {
   yield 1;
}
```

```
generator.prototype.__proto__; // Generator {constructor: GeneratorFunction,
next: ƒ, return: ƒ, throw: ƒ, Symbol(Symbol.toStringTag): "Generator"}

// 由于 Generator 不是一个全局变量，因此我们只能这么写：
generator.prototype.__proto__.math = function(e = 0) {
  return e * Math.PI;
}

generator.prototype.__proto__; // Generator {math: ƒ, constructor:
GeneratorFunction, next: ƒ, return: ƒ, throw: ƒ, …}

const gen = generator();
gen.math(1); // 3.141592653589793
```

生成器的用途

在前面，我们用了已知迭代次数的生成器。但如果我们不知道要迭代多少次会怎么样呢？为了解决这个问题，需要在生成器函数中创建一个无限循环。下面以一个会返回随机数的函数为例进行演示：

```
function * randomFrom(...arr) {
  while (true)
    yield arr[Math.floor(Math.random() * arr.length)];
}

const getRandom = randomFrom(1, 2, 5, 9, 4);

getRandom.next().value; // 返回随机的一个数
```

这是个简单的例子。下面来举一些更复杂的函数为例，我们要写一个节流（throttle）函数。如果你还不知道节流函数是什么，请参阅这篇文章。

```
function * throttle(func, time) {
  let timerID = null;
  function throttled(arg) {
    clearTimeout(timerID);
    timerID = setTimeout(func.bind(window, arg), time);
  }
  while (true)
    throttled(yield);
}
```

```
const thr = throttle(console.log, 1000);

thr.next(); // {value: undefined, done: false}
thr.next('hello'); // 返回 {value: undefined, done: false} ，然后 1 秒后输出 'hello'
```

还有没有更好的利用生成器的例子呢？如果你了解递归，那你肯定听过斐波那契数列。通常我们是用递归来解决这个问题的，但有了生成器后，可以这样写：

```
function * fibonacci(seed1, seed2) {
  while (true) {
    yield (() => {
      seed2 = seed2 + seed1;
      seed1 = seed2 - seed1;
      return seed2;
    })();
  }
}

const fib = fibonacci(0, 1);
fib.next(); // {value: 1, done: false}
fib.next(); // {value: 2, done: false}
fib.next(); // {value: 3, done: false}
fib.next(); // {value: 5, done: false}
fib.next(); // {value: 8, done: false}
```

不再需要递归了！我们可以在需要的时候获得数列中的下一个数字。

将生成器用在 HTML 上

既然是讨论 JavaScript，那显然要用生成器来操作下 HTML。

假设有一些 HTML 块需要处理，可以使用生成器来轻松实现。（当然除了生成器之外还有很多方法可以做到）

我们只需要少许代码就能完成此需求。

```
const strings = document.querySelectorAll('.string');
const btn = document.querySelector('#btn');
const className = 'darker';

function * addClassToEach(elements, className) {
  for (const el of Array.from(elements))
    yield el.classList.add(className);
```

```
}

const addClassToStrings = addClassToEach(strings, className);

btn.addEventListener('click', (el) => {
   if (addClassToStrings.next().done)
      el.target.classList.add(className);
});
```

仅有 5 行逻辑代码。

Generator 与异步编程

现在你已经了解了 ES6 生成器的基础知识和相关特性，是时候让我们来改进下现实中的代码了。

生成器的主要能力在于提供了一个单线程的类似同步模式的代码风格，同时允许将异步操作作为实现细节进行隐藏。这使得我们可以通过非常自然的方式来查看我们的代码流程，而不必面对异步的语法和相关问题。

换句话说，我们获得了一个不错的功能与关注点的分离，这是通过将对数据的消费（生成器逻辑）与异步获取数据的实现细节（next(..)）隔离开得到的。

结果呢？我们得到了异步代码的所有能力，以及阅读和维护同步（看起来）代码的容易性。

那我们如何来实现呢？

最简单的异步
简单来说，生成器并不要求程序具有任何额外的东西来获得控制异步的能力。

例如，假设已经有如下的代码。

```
function makeAjaxCall(url,cb) {
    // 执行 Ajax
```

```
    // 完成后调用 `cb(result)`
}

makeAjaxCall( "http://some.url.1", function(result1){
    var data = JSON.parse( result1 );

    makeAjaxCall( "http://some.url.2/?id=" + data.id, function(result2){
        var resp = JSON.parse( result2 );
        console.log( "The value you asked for: " + resp.value );
    });
} );
```

通过生成器（不增加任何修饰）来实现上面的程序，可以这样做：

```
function request(url) {
    // 这是我们因此异步过程的地方，与生成器的主要代码分开，
    // `it.next(..)` 是生成器继续下一个迭代的调用
    makeAjaxCall( url, function(response){
        it.next( response );
    } );
    // 注意：这里什么也不返回！
}

function *main() {
    var result1 = yield request( "http://some.url.1" );
    var data = JSON.parse( result1 );

    var result2 = yield request( "http://some.url.2?id=" + data.id );
    var resp = JSON.parse( result2 );
    console.log( "The value you asked for: " + resp.value );
}

var it = main();
it.next(); // 开始执行所有的代码
```

下面来说明上述代码是如何工作的。

帮助函数 request(..) 只是将原来的 makeAjaxCall(..) 函数进行了包装，确保其回调函数可以调用生成器的迭代器的 next(..) 方法。

对于 request("..") 调用，可以注意到它并没有 任何返回值（换句话说，返回值是 undefined）。这里 yield undefined，没什么大不了，不过文章后面要介绍的东西就比较重要了。

接着调用了 yield ..（值为 undefined），这里什么也没做，只是让生成器在这里暂停执行了而已。生成器会一直等待，直到通过调用 it.next(..) 使其继续执行，而这在 Ajax 调用完成时就会发生。

但是 yield .. 表达式的结果发生什么了呢？我们将表达式的结果赋值给了遍历 result1。变量是如何在第一个 Ajax 调用里面获得这个值的呢？

这是由于在 Ajax 的回调函数中调用 it.next(..) 时，将 Ajax 的响应数据传给了生成器，也就是说值被传给了生成器暂停的位置，也就是 result1 = yield .. 语句！

这的确很酷，很强大。本质上来说，result1 = yield request(..) 在 请求数据，但是过程是（几乎）完全隐藏的 —— 至少在这里不必关心 —— 而具体实现这一步骤的过程是异步的。通过 yield 的暂停实现了异步的过程，并且将使生成器继续的能力分配给了其他函数，从而使得代码实现了一个同步（看起来）数据请求。

第二个 result2 = yield result(..) 语句也是完全相同的过程：它隐含地进行了暂停和继续，并且得到了需要的数据，但我们在编码时不必关心异步的细节。

当然，出现了 yield，这是一个有关某些魔法可能会发生的微妙的提示。但是相对于嵌套回调函数（或者说 promise 链带来 的 API 开销），yield 只是很小的语法信号或者说开销。

注意我提到“可能会发生”。这其实是生成器本身和内部非常强大的东西。上面的程序的确产生了异步的 Ajax 调用，但如果它没有这样做呢？如果我们之后修改程序，使其改为使用之前（或者预先获取的）Ajax 响应结果呢？或者应用程序的 URL 路由器使得在某些情况下可以立即响应 Ajax 请求，而不必从服务器获取数据呢？

我们可以将 request(..) 的实现改为：

```
var cache = {};

function request(url) {
    if (cache[url]) {
        // “延迟”缓存的响应直到当前线程执行完成
        setTimeout( function(){
            it.next( cache[url] );
        }, 0 );
    }
    else {
        makeAjaxCall( url, function(resp){
            cache[url] = resp;
            it.next( resp );
        } );
    }
}
```

注意：这里有一个微妙的小技巧，就是在缓存数据已存在时需要通过 setTimeout(..0) 来产生延迟。如果我们立即调用 it.next(..)，这里会产生一个错误，因为（这就是小技巧的部分）从技术上来说，生成器还没有进入暂停状态。函数调用 request(..) 需要首先被执行，然后 yield 暂停。所以，我们不能立即在 request(..) 中调用 it.next(..)，因为此时生成器正在运行中（yield 还没有执行）。不过我们可以“稍后”调用 it.next(..)，在当前执行线程完成后立即执行，这也就是 setTimeout(..0) 实现的“hack”。后文给出了此问题更好的答案。

现在，主生成器的代码仍旧是这样：

```
var result1 = yield request( "http://some.url.1" );
var data = JSON.parse( result1 );
..
```

看到没有！？生成器逻辑（也就是流程控制）相比没有缓存控制的版本并没有任何改变。

*main() 中的代码仍然在请求数据，并且暂停直到数据返回然后继续执行。在当前场景中，“暂停”可能会很长（产生了一个真实的服务器请求，差不多 300-800ms），也可能很快（setTimeout(..0) 带来的延迟）。但是流程

控制并不关心。

这就是将抽象异步过程实现细节的强大之处。

更好的异步
上面的方式对于小规模的异步的生成器已经够用了。但是它会很快遇到瓶颈，所以我们需要使用更强大的异步机制，以便处理更复杂的情况。什么机制？Promise。

如果你对 ES6 Promise 不了解，我写过几篇介绍文章，可以去读一读，等你回来我们再继续。

前面的 Ajax 代码示例和初始的嵌套回调函数示例，有相同的有关控制反转的问题。以下是目前的一些问题：

没有清晰的异常处理的模式。上一篇文章中提到，我们可以检测 Ajax 调用的错误，通过 it.throw(..) 将其传递给生成器，然后在生成器逻辑内部通过 try..catch 来处理它。但这只是更多的手工处理的工作，而且这些代码在我们的程序中可能无法重用。

如果 makeAjaxCall(..) 工具不在我们的控制之下，并且多次调用回调函数，或者同时调用了成功和失败的回调函数，等等，那么生成器就乱七八糟的了（未捕获的异常，非期待的值，等等）。处理和避免这样的问题意味着重复的工作，而且很可能无法移植。

我们经常需要“并行”处理不止一个任务（例如，两个同时发起的 Ajax 调用）。由于生成器的 yield 语句是单个暂停点，两个或多个无法同时执行 —— 它们只能按顺序每次执行一个。所以，并不清楚应该如何通过一个 yield 处理多个任务，而且能不增加更多的代码。

可以看到，这些问题都可以解决，但谁想每次都重复做这些事情呢。我们需要一个强大的模式，能够为基于生成器的异步编程提供可信赖、可重用的解决方案。

什么模式呢？抛出 promise，让它们来继续生成器的执行。

上面我们用过 yield request(..)，并且 request(..) 没有返回任何值，yield

undefined 够高效吗？

让我们来把代码稍微修改下。将 request(..) 改为基于 promise 的，从而能够返回一个 promise，这样 yield 返回的实际是一个 promise（而不是 undefined）。

```
function request(url) {
    // 注意：现在返回一个 promise！
    return new Promise( function(resolve,reject){
        makeAjaxCall( url, resolve );
    } );
}
```

request(..) 现在构造了一个 promise（并且在 Ajax 调用完成后将其设为解决的（resolved）），将 promise 返回，从而可以被 yield 抛出。然后呢？

我们需要一个工具来控制生成器的迭代器，它会收集执行生成器得到的 promise（通过 next(..)）。现在我会称呼这个工具为 runGenerator(..)：

```
// （异步）运行一个生成器直到结束
// 注意：简化的方法：不进行异常处理
function runGenerator(g) {
    var it = g(), ret;

    // 异步地迭代生成器
    (function iterate(val){
        ret = it.next( val );

        if (!ret.done) {
            // 穷人版本的“这是一个 promise 吗？”检查
            if ("then" in ret.value) {
                // 等待 promise
                ret.value.then( iterate );
            }
            // 立即获得的值：返回即可
            else {
                // 避免同步执行
```

```
                    setTimeout( function(){
                        iterate( ret.value );
                    }, 0 );
                }
            }
        })();
}
```

主要需要注意的有：

自动初始化生成器（创建对应的 it 迭代器），然后异步执行 it 直到完成（done:true）。

检查是否返回了 promise（也就是每个 it.next(..) 调用的返回值），如果是，则通过向 promise 注册 then(..) 来等待 promise 完成。

如果有立即得到的（非 pormise）值返回，则将其传回生成器使得生成器可以继续执行。

现在，怎么使用它呢？

```
runGenerator( function *main(){
    var result1 = yield request( "http://some.url.1" );
    var data = JSON.parse( result1 );

    var result2 = yield request( "http://some.url.2?id=" + data.id );
    var resp = JSON.parse( result2 );
    console.log( "The value you asked for: " + resp.value );
} );
```

等等...这不就是之前的完全相同的生成器代码吗？是的。再一次，我们看到了生成器的强大之处。现在我们创建 promsie，将其 yield 到外部，等待其完成后继续执行生成器 —— 所有这些都是“隐藏的”实现细节！这并非真的隐藏，而是与消费代码（生成器的流程控制）相分离。

通过等待 yield 返回的 promise，并在其完成后将数据返回给 it.next(..)，result1 = yield request(..) 语句像之前一样获得了值。

不过现在我们采用 promise 来管理生成器代码的异步部分，从而相对于

只是用回调函数的方式，解决了倒置和信赖的问题。以上的解决方案都是通过生成器 + promise “免费”获得的：

我们现在有了可以简单上手的内置的异常处理。尽管在上面的 runGenerator(..) 中没有展示，但从 promise 监听错误并将其通过 it.throw(..) 进行传递并不复杂 —— 然后可以通过 try..catch 在生成器代码内部捕获并处理这些错误。

我们获得了 promise 提供的控制与信任。不必担心，不必大惊小怪。

promise 还提供了很多强大的抽象机制用于解决多个“并行”任务的复杂性，等等。

例如，yield Promise.all([..]) 可以用于“并行”处理一组 promise，作为单个 promise（交给生成器处理）返回，它会等到多有的子 promise 任务完成后才会继续。从 yield 表达式（当 promise 完成的时候）返回的是这一组 promsie 对应的响应数据，按照请求的顺序排列（不管实际完成的顺序如何）。

首先，我们来看看异常处理：

```
// 假设：`makeAjaxCall(..)`现在接收的是“异常优先风格”的回调函数（略）
// 假设：`runGenerator(..)`现在支持异常处理（略）

function request(url) {
    return new Promise( function(resolve,reject){
        // 传入“异常优先风格”的回调函数
        makeAjaxCall( url, function(err,text){
            if (err) reject( err );
            else resolve( text );
        } );
    } );
}

runGenerator( function *main(){
    try {
        var result1 = yield request( "http://some.url.1" );
```

```
    }
    catch (err) {
        console.log( "Error: " + err );
        return;
    }
    var data = JSON.parse( result1 );

    try {
        var result2 = yield request( "http://some.url.2?id=" + data.id );
    } catch (err) {
        console.log( "Error: " + err );
        return;
    }
    var resp = JSON.parse( result2 );
    console.log( "The value you asked for: " + resp.value );
} );
```

如果一个 promise 拒绝（或者任意形式的错误/异常）在获取 URL 的过程中发生，promise 的拒绝会被映射为生成器的错误（通过 —— 没有展示 —— runGenerator(..) 中的 it.throw(..)），然后被 try..catch 语句捕获。

接下来，我们看一个使用 promsie 来处理更复杂的异步的例子：

```
function request(url) {
    return new Promise( function(resolve,reject){
        makeAjaxCall( url, resolve );
    } )
    // 基于返回的文本作进一步的处理
    .then( function(text){
        // 是不是获得了一个（重定向）URL?
        if (/^https?:\/\/.+/.test( text )) {
            // 向新的 URL 发起另一次请求
            return request( text );
        }
        // 否则，假设文本就是期望返回的数据
        else {
            return text;
        }
```

```
    } );
}

runGenerator( function *main(){
    var search_terms = yield Promise.all( [
        request( "http://some.url.1" ),
        request( "http://some.url.2" ),
        request( "http://some.url.3" )
    ] );

    var search_results = yield request(
        "http://some.url.4?search=" + search_terms.join( "+" )
    );
    var resp = JSON.parse( search_results );

    console.log( "Search results: " + resp.value );
} );
```

Promise.all([..]) 构造了一个等待三个内部 promise 完成的 promise，它是 yield 给 runGenerator(..) 来监听从而继续执行生成器的 promise。内部的 promise 可以接收一个看起来像重定向 URL 的响应，然后将另一个内部 promise 链到新的地址上。更多有关 promise 链的内容，可以阅读这篇文章。

所有 promise 可以用于异步模式的能力，都可以通过生成器中 yield promise（或者是 promise 的 promise 的 promise...）来以类似同步的代码方式获得。这是两个世界最好的地方了（译注：指生成器和 promise 的结合）。

总结

简单地将生成器 + yield promsie 结合起来，就可以获得两个世界的最好的部分，也就是强大的功能和优雅的同步的（看起来）异步流程控制能力。使用简单的包装工具（这个很多库已经提供），我们可以自动执行生成器，并且支持健壮的和同步的（看起来）异常处理！

而在 ES7+ 领域，我们很可能会看到 async function 在不需要额外工具库的情况下支持这些特性（至少是基础的情况）！

JavaScript 异步编程的未来是光明的，而且只会更加光明！

Generator 扩展：ES6 生成器与并发

我们最后要讨论的主题其实是个前沿问题，你可能会觉得有点虐脑（老实说，我现在也还在被虐中）。深入并思考这些问题需要花费时间，当然，你还要再多读一些关于这个主题的文章。

不过你现在的投资从长远来说会是非常有价值的，我非常确信未来 JS 的复杂异步编程能力，会从这里得到提升。

正统 CSP（通信顺序进程，Communicating Sequential Processes）
在使用 JS 前并没有 Clojure 语言的背景，或者 Go、ClojureScript 语言的经验。

破坏 CSP 理论（一点点）
CSP 到底是什么呢？“通信”是什么意思？“顺序”？“进程”又是什么？

首先，CSP 来源于 Tony Hoare 的书《通信顺序进程》。这是非常深奥的计算机科学理论，但如果你喜欢这些学术方面的东西，那这本书是最好的开始。我不想以深奥、晦涩的计算机科学的方式来讨论这个话题，我采用的是非常不正式的方式。

我们先从“顺序”开始。这应该是你已经熟悉的部分了。这其实是换了个方式讨论 ES6 生成器的单线程行为以及类似同步模式的代码。

别忘了生成器的语法是这样的：

```
function *main() {
    var x = yield 1;
    var y = yield x;
    var z = yield (y * 2);
}
```

这些语句都是同步顺序（按照出现的前后顺序）执行的，一次执行一条。yield 关键字标记了那些会出现打断式的暂停（只是在生成器代码内部打

断，而非外部的程序）的位置，而不会改变处理*main() 的外部代码。很简单，不是吗？

接下来，我们来看“进程”。这些是什么呢？

本质上来说，生成器的各种行为就像是虚拟的“进程”。如果 JavaScript 允许的话，它就像是程序中并行于其他部分运行的一部分代码。

实际上，这有点乱说了一点。如果生成器可以访问共享内存（这是指，它可以访问其内部的局部变量以为的“自由变量”），那么它就并没有那么独立。但是让我们假设有一个没有访问外部变量的生成器（这样 FP 理论会称之为“连接器（combinator）”），这样理论上它可以运行在自己的进程中，或者说作为单独的进程运行。

不过我们说的是“进程（processes）”——复数——因为最重要的是有两个或多个进程同时存在。也就是说，两个或多个生成器匹配在一起，共同完成某个更大的任务。

为什么要把生成器拆分开呢？最重要的原因：功能或关注点的分离。对于任务 XYZ，如果能将其拆分为子任务 X、Y、Z，然后在单独的生成器中进行实现，这会使得代码更容易理解和维护。

也是基于同样的原因，才会将类似 function XYZ() 的代码拆分为 X()、Y()、Z() 函数，然后 X() 调用 Y()，Y() 调用 Z()，等等。我们将函数进行拆分使得代码更好地分离，从而更容易维护。

我们可以用多个生成器来实现相同的事情。

最后，“通信”。这是什么呢？它延续自上面 —— 合作 —— 如果生成器需要一起工作，它们需要一个通信通道（不仅仅是访问共享的词法作用域，而是一个真实共享的排外的通信通道）。

通信通道里有什么呢？任何需要传递的东西（数值，字符串，等等）。实际上，并不需要真的在通道发送消息。“通信”可以像协作一样简单 —— 例如将控制权从一个转移到另一个。

为什么要转移控制权？主要是由于 JS 是单线程的，某一时刻只能有一个

生成器在执行。其他的处于暂停状态，这意味着它们在执行任务的过程中，但因为需要等待在必要的时候继续执行而挂起。

任意的独立的“线程”都可以神奇地协作并通信好像并不现实。这种松耦合的目标是好的，但是不切实际。

相反，任何成功的 CSP 的实现，都是对于已有的问题领域的逻辑集合进行内部分解，并且每一部分都被设计为能够与其他部分共同工作。

或许在这方面我完全错了，但我还没有看到有什么有效的方式， 可以使得两个任意的生成器函数能够简单地粘在一起作为 CSP 配对使用。它们都需要被设计为可以与另一个一同工作，遵循通信协议，等等。

JS 中的 CSP
有几种有趣的 CSP 探索应用于 JS 了。

前面提及的 David Nolen，有几个有趣的项目，包括 Om，以及 core.async。Koa 库（用于 node.js）有一个有趣的特性，主要通过其 use(..) 方法。另一个与 core.async/Go CSP 接口一致的库是 js-csp。

建议你将这些项目检出来看看各种在 JS 中应用 CSP 的方式和例子。

asynquence 的 runner(..)：设计 CSP

既然我一直在尝试将 CSP 模式应用于自己的代码，那么为我的异步流程控制库 asynquence 增加 CSP 能力就是很自然的选择了。

我之前演示过使用 runner(..) 插件来处理生成器的异步运行（见第三部分），所以对我而言以类似 CSP 的方式同时支持处理多个生成器是很容易的。

第一个设计问题是：怎样知道哪个生成器来控制下一个（next）？

让进程有某种 ID，从而可以彼此知道，这有点笨重，不过这样它们就可以直接传递消息和将控制权转移给另一个进程。在经过一些试验后，我选择了简单的循环调度方法。对于三个生成器 A、B、C，A 首先获得控制权，然后当 A 抛出（yield）控制权后由 B 接手，接着由 C 接手 B，再然后

是 A，如此往复。

但我们实际转移控制权呢？需要有对应的 API 吗？再一次，经过一些试验后，我选择了更隐蔽的方法，和 Koa 的做法类似（完全是偶然地）：每个生成器获得一个共享的“token”—— yield 返回它时表示进行控制转移。

另一个问题是消息通道应该是什么样的。或许是一个正式的通信接口，如 core.async 和 js-csp 那样（put(..) 和 take(..)）。根据我自己的实验，我更倾向于另一种方式，一个不那么正式的方法（甚至不是 API，而是类似 array 的共享的数据结构）就够用了。

我决定使用数组（称为 messages），可以任意地根据需要写入和提出数据。可以将数据 push() 到数组，从数组 pop() 出来，给不同的数据分配不同的位置，或者在里面存储更复杂的数据结构，等等。

我觉得对于一些任务来说只需要简单的数据传递，对于另一些则要更复杂些，所以与其让简单的情况变复杂，我选择不将消息通道正式化，而是只有一个 array（于是没有 API，只剩下 array 本身）。如果你觉得有必要，也很容易给数据传递增加一些规范性（见下面的 状态机 例子）。

最终，我发现这些生成器“进程”仍然可以获得异步生成器的那些好处。换句话说，如果不是抛出控制 token，而是 Promise（或一个 asynquence 序列），runner(..) 的机制会暂停来等待这个值，而 不会转移控制权 —— 相反，它会将数据返回给当前的进程（生成器）使其重新获得控制权。

后面的观点可能（如果我解释地正确的话）是最有争议或最不像其他库的地方。或许真正的 CSP 会不屑于这些方法。不过，我觉得有这些想法是很有用的。

一个简单的 FooBar 示例
理论已经够多了，让我们来看看代码：

```
// 注意：略去了 `multBy20(..)` 和 `addTo2(..)` 这些异步数学函数

function *foo(token) {
    // 从通道的顶部获取数据
    var value = token.messages.pop(); // 2
```

```
    // 将另一个数据放到通道上
    // `multBy20(..)` 是一个产生 promise 的函数，
    // 在延迟一会之后将一个值乘以 `20`
    token.messages.push( yield multBy20( value ) );

    // 转义控制权
    yield token;

    // CSP 运行返回的最后的数据
    yield "meaning of life: " + token.messages[0];
}

function *bar(token) {
    // 从通道的顶部获取数据
    var value = token.messages.pop(); // 40

    // 将另一个数据放到通道上
    // `addTo2(..)` 是一个产生 promise 的函数，
    // 在延迟一会之后将一个值加上 `2`
    token.messages.push( yield addTo2( value ) );

    // transfer control
    yield token;
}
```

OK，以上是两个生成器“进程”，*foo() 和 *bar()。可以注意到，两个都是处理 token 对象（当然，你也可以随便怎么称呼它）。token 的 message 属性就是共享的消息通道。它由 CSP 初始化运行时传入的数据填充（见后面）。

yield token 隐含地转移控制到“下一个”生成器（循环顺序）。不过，yield multBy20(value) 和 yield addTo2(value) 都是抛出 promise（从略去的延迟数学函数），这意味着生成器会暂停，直到 promise 完成。当 promise 完成，当前出于控制状态的生成器会继续执行。

无论最后的 yield 值是什么，在 yield "meaning of... 表达式语句中，这都是 CSP 运行的完成消息（见后面）。

现在我们有两个 CSO 进程生成器，怎么运行呢？使用 asynquence：

```
// 使用初始数据 `2` 启动一个序列
ASQ( 2 )

// 一起运行这两个 CSP 进程
.runner(
    foo,
    bar
)

// 无论最后得到什么消息都向下一步传递
.val( function(msg){
    console.log( msg ); // "meaning of life: 42"
} );
```

显然，这只是一个测试示例。不过我想这已经很好地展示了相关概念。

现在你可以自己来试试（试着改变下数据！）从而确信这些概念有用，并且你能自己写出代码。

另一个玩具示例
现在我们来看一个经典的 CSP 的例子，不过是以前面介绍的我的方式，而不是以学术上的视角。

乒乓。很有意思的运动是不是！？这是我最喜欢的运动。

我们假设你已经实现了一个乒乓游戏的代码。你有一个循环以运行游戏，并且你有两部分代码（例如，使用 if 或 switch 语句的分支）分别代表两个选手。

你的代码运行良好，你的游戏就像乒乓比赛那样运行！

但是关于 CSP 为什么有效我说过什么呢？关注点或功能的分离。乒乓游戏中的分离的功能是什么呢？这两个选手嘛！

所以，从一个较高的层面上，我们可以将游戏建模为两个“进程”（生成

器），分别对应每个选手。当我们进入实现的细节，我们会发现在两个选手间转移控制的“胶水代码”是一个单独的任务，这部分代码可以是第三个生成器，我们可以将其建模为游戏裁判。

我们将会跳过所有的领域特定的问题，例如比分、游戏机制、物理、游戏策略、AI、控制，等等。我们唯一关心的部分是模拟来回的击打（这其实是对 CSP 控制转移的比喻）。

想看看 demo 吗？运行一下吧（注意：使用一个较新版本的 FF 或 Chrome，支持 ES6 从而可以运行生成器）

现在，我们来一段一段看下代码。

首先，asynquence 序列长什么样呢？

```
ASQ(
    ["ping","pong"], // 选手名字
    { hits: 0 } // 乒乓球
)
.runner(
    referee,
    player,
    player
)
.val( function(msg){
    message( "referee", msg );
} );
```

我们使用两个初始数据：["ping","pong"] 和 { hits: 0 }。我们很快会讨论这些。

然后我们建立了 CSP 来运行 3 个进程（协程（coroutine））：一个 *referee() 和两个 *player() 实例。

游戏最终的数据会传入序列中的下一步骤，然后我们会输出来自裁判的数据。

裁判的实现：

```
function *referee(table){
    var alarm = false;

    // 裁判在自己的定时器上设置警报（10 秒）
    setTimeout( function(){ alarm = true; }, 10000 );

    // 让游戏保持运行直到警报响起
    while (!alarm) {
        // 让选手继续
        yield table;
    }

    // 告知选手游戏结束
    table.messages[2] = "CLOSED";

    // 然后裁判说了什么呢？
    yield "Time's up!";
}
```

我调用控制 token table 来匹配问题域（乒乓游戏）。当选手将球击回的时候“转移（yield） table”是很好的语义，不是吗？

*referee() 中的 while 循环保持转移 table，只要他的定时器上的警报没有响起。警报响的时候，他会接管游戏，然后通过 "Time's up!" 宣布游戏结束。

现在，我们来看下 *player() 生成器（我们使用了它的两个实例）：

```
function *player(table) {
    var name = table.messages[0].shift();
    var ball = table.messages[1];

    while (table.messages[2] !== "CLOSED") {
        // 击球
        ball.hits++;
        message( name, ball.hits );
```

```
        // 当球返回另一个选手时产生延迟
        yield ASQ.after( 500 );

        // 游戏还在继续？
        if (table.messages[2] !== "CLOSED") {
            // 球现在在另一个选手那边了
            yield table;
        }
    }

    message( name, "Game over!" );
}
```

第一个选手从数据的数组中取出他的名字（"ping"），然后第二个选手获取他的名字（"pong"），所以他们都能正确识别自己。两个选手记录了一个到共享的 ball 对象的引用（包含一个 hits 计数器）。

如果选手们没有从裁判那里听到结束的消息，他们通过增加 hits 计数器来“击打” ball（并输出一个消息来发布出来），然后等待 500ms（因为球不能以光速传播！）。

如果游戏仍在继续，他们紧接着“转移球台”给另一个选手。

就是这样！

看下 demo 的代码，可以了解到让这些部分一起工作的完整上下文代码。

状态机：生成器协程
最后一个例子：定义一个状态机，即由一个辅助工具来驱动的一组生成器协程。

Demo（注意：使用一个较新版本的 FF 或 Chrome，支持 ES6 从而可以运行生成器）

首先，定义一个控制有限状态处理器的辅助工具：

```
function state(val,handler) {
    // 为状态创建一个协程处理器（包装）
```

```
    return function*(token) {
        // 状态变化处理器
        function transition(to) {
            token.messages[0] = to;
        }

        // 缺省的初始状态（如果没有设置）
        if (token.messages.length < 1) {
            token.messages[0] = val;
        }

        // 保持运行直到达到最终状态（false）
        while (token.messages[0] !== false) {
            // 当前状态匹配处理器？
            if (token.messages[0] === val) {
                // 委托到处理器
                yield *handler( transition );
            }

            // 转移控制到另一个状态处理器？
            if (token.messages[0] !== false) {
                yield token;
            }
        }
    };
}
```

state(..) 辅助工具函数创建了一个对应特定状态值的委托生成器的包装对象，该对象会自动运行状态机，并在每次状态改变时转移控制权。

纯粹是由于个人喜好，我决定由共享的 token.messages[0] 来记录状态机的当前状态。这意味着将序列的上一步传入的数据作为初始状态使用。不过如果没有设置初始数据，则缺省使用第一个状态作为初始状态。同样是个人喜好的缘故，最终状态被设为 false。这个很容易根据你自己的喜欢进行修改。

状态值可以是你喜欢的任意类型的值：number、string，等等。只要可以通过 === 严格测试的值，你都可以用来作为状态值。

在接下来的例子中，我会演示一个变化四个 number 状态值的状态机，按照特定的顺序：1 -> 4 -> 3 -> 2。仅为了演示目的，会使用一个计数器，从而可以执行该变化循环不止一次。但状态机最终达到最终状态（false）时，asynquence 序列向下一步移动，和预期的一样。

```
// 计数器（仅为了演示的目的）
var counter = 0;

ASQ( /* 可选的：初始化状态值 */ )

// 运行状态机，变化：1 -> 4 -> 3 -> 2
.runner(

    // 状态 `1` 处理器
    state( 1, function*(transition){
        console.log( "in state 1" );
        yield ASQ.after( 1000 ); // 暂停 1s
        yield transition( 4 ); // 跳转到状态 `4`
    } ),

    // 状态 `2` 处理器
    state( 2, function*(transition){
        console.log( "in state 2" );
        yield ASQ.after( 1000 ); // 暂停 1s

        // 仅为了演示的目的，判断是否继续状态循环？
        if (++counter < 2) {
            yield transition( 1 ); // 跳转到状态 `1`
        }
        // 全部完成！
        else {
            yield "That's all folks!";
            yield transition( false ); // 跳转到退出状态
        }
    } ),
```

```
    // 状态 `3` 处理器
    state( 3, function*(transition){
        console.log( "in state 3" );
        yield ASQ.after( 1000 ); // 暂停 1s
        yield transition( 2 ); // 跳转到状态 `2`
    } ),

    // 状态 `4` 处理器
    state( 4, function*(transition){
        console.log( "in state 4" );
        yield ASQ.after( 1000 ); // 暂停 1s
        yield transition( 3 ); // 跳转到状态 `3`
    } )

)

// 状态机完成，所以继续下一步
.val(function(msg){
    console.log( msg );
});
```

很容易可以跟踪这里的过程。

yield ASQ.after(1000) 说明这些生成器可以做任何基于 promise/sequence 的异步处理，这个与之前看到过一样。yield transition(..) 用于转换到新的状态。

上面的 state(..) 辅助函数完成了工作中困难的部分，处理 yield* 委托和状态跳转，使得状态处理器可以非常简单和自然。

总结

CSP 的关键在于将两个或更多的生成器“进程”连接在一起，提供一个共享的通信通道，以及可以在彼此间转移控制权的方法。

已经有一些 JS 库以正统的方式实现了和 Go、Clojure/ClojureScript 差不多的 API 和语义。这些库背后都有些聪明的开发者，并且他们都提供了很多有关进一步探索的资源。

asynquence 尝试采用一个不那么正统的但希望仍保留了主要的机制的方式。如果没有更多的需求，asynquence 的 runner(..) 对于开始探索类似 CSP 的生成器已经非常容易了。

不过最好的地方是将 asynquence 的 CSP 与其他的异步功能一起使用（promise、生成器、流程控制，等等）。这样，你就有了所有领域的最好的部分，从而在处理手头的工作时可以选用任何更适合的工具，而这些都在一个较小的库中。