```
    console.log( "inside *bar():", yield "A" );

    // yield委托!
    console.log( "inside *bar():", yield *foo() );

    console.log( "inside *bar():", yield "E" );

    return "F";
}

var it = bar();

console.log( "outside:", it.next().value );
// outside: A

console.log( "outside:", it.next( 1 ).value );
// inside *bar(): 1
// outside: B

console.log( "outside:", it.next( 2 ).value );
// inside *foo(): 2
// outside: C

console.log( "outside:", it.next( 3 ).value );
// inside *foo(): 3
// inside *bar(): D
// outside: E

console.log( "outside:", it.next( 4 ).value );
// inside *bar(): 4
// outside: F
```

要特别注意 `it.next(3)` 调用之后的执行步骤。

(1) 值 `3`（通过 `*bar()` 内部的 `yield` 委托）传入等待的 `*foo()` 内部的 `yield "C"` 表达式。
(2) 然后 `*foo()` 调用 `return "D"`，但是这个值并没有一直返回到外部的 `it.next(3)` 调用。
(3) 取而代之的是，值 `"D"` 作为 `*bar()` 内部等待的 `yield*foo()` 表达式的结果发出——这个 `yield` 委托本质上在所有的 `*foo()` 完成之前是暂停的。所以 `"D"` 成为 `*bar()` 内部的最后结果，并被打印出来。
(4) `yield "E"` 在 `*bar()` 内部调用，值 `"E"` 作为 `it.next(3)` 调用的结果被 `yield` 发出。

从外层的迭代器（`it`）角度来说，是控制最开始的生成器还是控制委托的那个，没有任何区别。

实际上，`yield` 委托甚至并不要求必须转到另一个生成器，它可以转到一个非生成器的一般 iterable。比如：

```
function *bar() {
    console.log( "inside *bar():", yield "A" );
```

```
    // yield委托给非生成器!
    console.log( "inside *bar():", yield *[ "B", "C", "D" ] );

    console.log( "inside *bar():", yield "E" );

    return "F";
}

var it = bar();

console.log( "outside:", it.next().value );
// outside: A

console.log( "outside:", it.next( 1 ).value );
// inside *bar(): 1
// outside: B

console.log( "outside:", it.next( 2 ).value );
// outside: C

console.log( "outside:", it.next( 3 ).value );

// outside: D

console.log( "outside:", it.next( 4 ).value );
// inside *bar(): undefined
// outside: E

console.log( "outside:", it.next( 5 ).value );
// inside *bar(): 5
// outside: F
```

注意这个例子和之前那个例子在消息接收位置和报告位置上的区别。

最显著的是，默认的数组迭代器并不关心通过 `next(..)` 调用发送的任何消息，所以值 2、3 和 4 根本就被忽略了。还有，因为迭代器没有显式的返回值（和前面使用的 `*foo()` 不同），所以 `yield *` 表达式完成后得到的是一个 `undefined`。

异常也被委托！

和 `yield` 委托透明地双向传递消息的方式一样，错误和异常也是双向传递的：

```
function *foo() {
    try {
        yield "B";
    }
    catch (err) {
        console.log( "error caught inside *foo():", err );
    }

    yield "C";

    throw "D";
}
```

```
function *bar() {
    yield "A";

    try {
        yield *foo();
    }
    catch (err) {
        console.log( "error caught inside *bar():", err );
    }

    yield "E";

    yield *baz();

    // 注:不会到达这里!
    yield "G";
}

function *baz() {
    throw "F";
}

var it = bar();

console.log( "outside:", it.next().value );
// outside: A

console.log( "outside:", it.next( 1 ).value );
// outside: B

console.log( "outside:", it.throw( 2 ).value );
// error caught inside *foo(): 2
// outside: C

console.log( "outside:", it.next( 3 ).value );
// error caught inside *bar(): D
// outside: E

try {
    console.log( "outside:", it.next( 4 ).value );
}
catch (err) {
    console.log( "error caught outside:", err );
}
// error caught outside: F
```

这段代码中需要注意以下几点。

(1) 调用 `it.throw(2)` 时，它会发送错误消息 2 到 `*bar()`，它又将其委托给 `*foo()`，后者捕获并处理它。然后，`yield "C"` 把 `"C"` 发送回去作为 `it.throw(2)` 调用返回的 `value`。

(2) 接下来从 `*foo()` 内 `throw` 出来的值 `"D"` 传播到 `*bar()`，这个函数捕获并处理它。然后 `yield "E"` 把 `"E"` 发送回去作为 `it.next(3)` 调用返回的 `value`。

(3) 然后，从 *baz() throw 出来的异常并没有在 *bar() 内被捕获——所以 *baz() 和 *bar() 都被设置为完成状态。这段代码之后，就再也无法通过任何后续的 next(..) 调用得到值 "G"，next(..) 调用只会给 value 返回 undefined。

4.5.3 异步委托

我们终于回到前面的多个顺序 Ajax 请求的 yield 委托例子：

```
function *foo() {
    var r2 = yield request( "http://some.url.2" );
    var r3 = yield request( "http://some.url.3/?v=" + r2 );

    return r3;
}

function *bar() {
    var r1 = yield request( "http://some.url.1" );

    var r3 = yield *foo();

    console.log( r3 );
}

run( bar );
```

这里我们在 *bar() 内部没有调用 yield run(foo)，而是调用 yield *foo()。

在这个例子之前的版本中，使用了 Promise 机制（通过 run(..) 控制）把值从 *foo() 内的 return r3 传递给 *bar() 中的局部变量 r3。现在，这个值通过 yield * 机制直接返回。

除此之外的行为非常相似。

4.5.4 递归委托

当然，yield 委托可以跟踪任意多委托步骤，只要你把它们连在一起。甚至可以使用 yield 委托实现异步的生成器*递归*，即一个 yield 委托到它自身的生成器：

```
function *foo(val) {
    if (val > 1) {
        // 生成器递归
        val = yield *foo( val - 1 );
    }

    return yield request( "http://some.url/?v=" + val );
}

function *bar() {
    var r1 = yield *foo( 3 );
    console.log( r1 );
```

```
}

run( bar );
```

`run(..)` 工具可以通过 `run( foo, 3 )` 调用，因为它支持额外的参数和生成器一起传入。但是，这里使用了没有参数的 `*bar()`，以展示 `yield *` 的灵活性。

这段代码后面的处理步骤是怎样的呢？坚持一下，接下来的细节描述可能会非常复杂。

(1) `run(bar)` 启动生成器 `*bar()`。
(2) `foo(3)` 创建了一个 `*foo(..)` 的迭代器，并传入 `3` 作为其参数 `val`。
(3) 因为 `3 > 1`，所以 `foo(2)` 创建了另一个迭代器，并传入 `2` 作为其参数 `val`。
(4) 因为 `2 > 1`，所以 `foo(1)` 又创建了一个新的迭代器，并传入 `1` 作为其参数 `val`。
(5) 因为 `1 > 1` 不成立，所以接下来以值 `1` 调用 `request(..)`，并从这第一个 Ajax 调用得到一个 promise。
(6) 这个 promise 通过 `yield` 传出，回到 `*foo(2)` 生成器实例。
(7) `yield *` 把这个 promise 传出回到 `*foo(3)` 生成器实例。另一个 `yield *` 把这个 promise 传出回到 `*bar()` 生成器实例。再有一个 `yield *` 把这个 promise 传出回到 `run(..)` 工具，这个工具会等待这个 promsie（第一个 Ajax 请求）的处理。
(8) 这个 promise 决议后，它的完成消息会发送出来恢复 `*bar()`；后者通过 `yield *` 转入 `*foo(3)` 实例；后者接着通过 `yield *` 转入 `*foo(2)` 生成器实例；后者再接着通过 `yield *` 转入 `*foo(3)` 生成器实例内部的等待着的普通 `yield`。
(9) 第一个调用的 Ajax 响应现在立即从 `*foo(3)` 生成器实例中返回。这个实例把值作为 `*foo(2)` 实例中 `yield *` 表达式的结果返回，赋给它的局部变量 `val`。
(10) 在 `*foo(2)` 中，通过 `request(..)` 发送了第二个 Ajax 请求。它的 promise 通过 `yield` 发回给 `*foo(1)` 实例，然后通过 `yield *` 一路传递到 `run(..)`（再次进行步骤 7）。这个 promise 决议后，第二个 Ajax 响应一路传播回到 `*foo(2)` 生成器实例，赋给它的局部变量 val。
(11) 最后，通过 `request(..)` 发出第三个 Ajax 请求，它的 promise 传出到 `run(..)`，然后它的决议值一路返回，然后 `return` 返回到 `*bar()` 中等待的 `yield *` 表达式。

噫！这么多疯狂的脑力杂耍，是不是？这一部分你可能需要多读几次，然后吃点零食让大脑保持清醒！

4.6 生成器并发

就像我们在第 1 章和本章前面都讨论过的一样，两个同时运行的进程可以合作式地交替运作，而很多时候这可以产生（双关，原文为 yield：既指产生又指 `yield` 关键字）非常强大

的异步表示。

坦白地说，本部分前面的多个生成器并发交替执行的例子已经展示了如何使其看起来令人迷惑。但是，我们已经暗示过了，在一些场景中这个功能会很有用武之地的。

回想一下第 1 章给出的一个场景：其中两个不同并发 Ajax 响应处理函数需要彼此协调，以确保数据交流不会出现竞态条件。我们把响应插入到 res 数组中，就像这样：

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

但是这种场景下如何使用多个并发生成器呢？

```
// request(..)是一个支持Promise的Ajax工具

var res = [];

function *reqData(url) {
    res.push(
        yield request( url )
    );
}
```

这里我们将使用生成器 *reqData(..) 的两个实例，但运行两个不同生成器的实例也没有任何区别。两种方法的过程几乎一样。稍后将会介绍两个不同生成器的彼此协调。

这里不需要手工为 res[0] 和 res[1] 赋值排序，而是使用合作式的排序，使得 res.push(..) 把值按照预期以可预测的顺序正确安置。这样，表达的逻辑给人感觉应该更清晰一点。

但是，实践中我们如何安排这些交互呢？首先，使用 Promise 手工实现：

```
var it1 = reqData( "http://some.url.1" );
var it2 = reqData( "http://some.url.2" );

var p1 = it1.next();
var p2 = it2.next();

p1
.then( function(data){
    it1.next( data );
    return p2;
```

```
} )
.then( function(data){
    it2.next( data );
} );
```

*reqData(..) 的两个实例都被启动来发送它们的 Ajax 请求，然后通过 yield 暂停。然后我们选择在 p1 决议时恢复第一个实例，然后 p2 的决议会重启第二个实例。通过这种方式，我们使用 Promise 配置确保 res[0] 中会放置第一个响应，而 res[1] 中会放置第二个响应。

但是，坦白地说，这种方式的手工程度非常高，并且它也不能真正地让生成器自己来协调，而那才是真正的威力所在。让我们换一种方法试试：

```
// request(..)是一个支持Promise的Ajax工具

var res = [];

function *reqData(url) {
    var data = yield request( url );

    // 控制转移
    yield;

    res.push( data );
}

var it1 = reqData( "http://some.url.1" );
var it2 = reqData( "http://some.url.2" );

var p1 = it.next();
var p2 = it.next();

p1.then( function(data){
    it1.next( data );
} );

p2.then( function(data){
    it2.next( data );
} );

Promise.all( [p1,p2] )
.then( function(){
    it1.next();
    it2.next();
} );
```

好吧，这看起来好一点（尽管仍然是手工的！），因为现在 *reqData(..) 的两个实例确实是并发运行了，而且（至少对于前一部分来说）是相互独立的。

在前面的代码中，第二个实例直到第一个实例完全结束才得到数据。但在这里，两个实例都是各自的响应一回来就取得了数据，然后每个实例再次 yield，用于控制传递的目的。然后我们在 Promise.all([..]) 处理函数中选择它们的恢复顺序。

可能不那么明显的是，因为对称性，这种方法以更简单的形式暗示了一种可重用的工具。还可以做得更好。来设想一下使用一个称为 runAll(..) 的工具：

```
// request(..)是一个支持Promise的Ajax工具

var res = [];

runAll(
    function*(){
        var p1 = request( "http://some.url.1" );

        // 控制转移
        yield;

        res.push( yield p1 );
    },
    function*(){
        var p2 = request( "http://some.url.2" );

        // 控制转移
        yield;

        res.push( yield p2 );
    }
);
```

我们不准备列出 runAll(..) 的代码，不仅是因为其可能因太长而使文本混乱，也因为它是我们在前面 run(..) 中实现的逻辑的一个扩展。所以，我们把它作为一个很好的扩展练习，请试着从 run(..) 的代码演进实现我们设想的 runAll(..) 的功能。我的 asynquence 库也提供了一个前面提过的 runner(..) 工具，其中已经内建了对类功能的支持，这将在本部分的附录 A 中讨论。

以下是 runAll(..) 内部运行的过程。

(1) 第一个生成器从第一个来自于 "http://some.url.1" 的 Ajax 响应得到一个 promise，然后把控制 yield 回 runAll(..) 工具。

(2) 第二个生成器运行，对于 "http://some.url.2" 实现同样的操作，把控制 yield 回 runAll(..) 工具。

(3) 第一个生成器恢复运行，通过 yield 传出其 promise p1。在这种情况下，runAll(..) 工具所做的和我们之前的 run(..) 一样，因为它会等待这个 promise 决议，然后恢复同一个生成器（没有控制转移！）。p1 决议后，runAll(..) 使用这个决议值再次恢复第一个生成器，然后 res[0] 得到了自己的值。接着，在第一个生成器完成的时候，有一个隐式的控制转移。

(4) 第二个生成器恢复运行，通过 yield 传出其 promise p2，并等待其决议。一旦决议，runAll(..) 就用这个值恢复第二个生成器，设置 res[1]。

在这个例子的运行中，我们使用了一个名为 res 的外层变量来保存两个不同的 Ajax 响应结果，我们的并发协调使其成为可能。

但是，如果继续扩展 runAll(..) 来提供一个内层的变量空间，以使多个生成器实例可以共享，将是非常有帮助的，比如下面这个称为 data 的空对象。还有，它可以接受 yield 的非 Promise 值，并把它们传递到下一个生成器。

考虑：

```
// request(..)是一个支持Promise的Ajax工具

runAll(
    function*(data){
        data.res = [];

        // 控制转移(以及消息传递)
        var url1 = yield "http://some.url.2";

        var p1 = request( url1 ); // "http://some.url.1"

        // 控制转移
        yield;

        data.res.push( yield p1 );
    },
    function*(data){
        // 控制转移(以及消息传递)
        var url2 = yield "http://some.url.1";

        var p2 = request( url2 ); // "http://some.url.2"

        // 控制转移
        yield;

        data.res.push( yield p2 );
    }
);
```

在这一方案中，实际上两个生成器不只是协调控制转移，还彼此通信，通过 data.res 和 yield 的消息来交换 url1 和 url2 的值。真是极其强大！

这样的实现也为被称作*通信顺序进程*（Communicating Sequential Processes，CSP）的更高级异步技术提供了一个概念基础。对此，我们将在本部分的附录 B 中详细讨论。

4.7 形实转换程序

目前为止，我们已经假定从生成器 yield 出一个 Promise，并且让这个 Promise 通过一个像 run(..) 这样的辅助函数恢复这个生成器，这是通过生成器管理异步的最好方法。要知道，事实的确如此。

但是，我们忽略了另一种广泛使用的模式。为了完整性，我们来简要介绍一下这种模式。

在通用计算机科学领域，有一个早期的前 JavaScript 概念，称为*形实转换程序*（thunk）。我们这里将不再陷入历史考据的泥沼，而是直接给出形实转换程序的一个狭义表述：JavaScript 中的 thunk 是指一个用于调用另外一个函数的函数，没有任何参数。

换句话说，你用一个函数定义封装函数调用，包括需要的任何参数，来定义这个调用的执行，那么这个封装函数就是一个形实转换程序。之后在执行这个 thunk 时，最终就是调用了原始的函数。

举例来说：

```
function foo(x,y) {
    return x + y;
}

function fooThunk() {
    return foo( 3, 4 );
}

// 将来

console.log( fooThunk() );  // 7
```

所以，同步的 thunk 是非常简单的。但如果是异步的 thunk 呢？我们可以把这个狭窄的 thunk 定义扩展到包含让它接收一个回调。

考虑：

```
function foo(x,y,cb) {
    setTimeout( function(){
        cb( x + y );
    }, 1000 );
}

function fooThunk(cb) {
    foo( 3, 4, cb );
}

// 将来

fooThunk( function(sum){
    console.log( sum );     // 7
} );
```

正如所见，`fooThunk(..)` 只需要一个参数 `cb(..)`，因为它已经有预先指定的值 `3` 和 `4`（分别作为 `x` 和 `y`）可以传给 `foo(..)`。thunk 就耐心地等待它完成工作所需的最后一部分：那个回调。

但是，你并不会想手工编写 thunk。所以，我们发明一个工具来做这部分封装工作。

考虑：

```
function thunkify(fn) {
    var args = [].slice.call( arguments, 1 );
    return function(cb) {
        args.push( cb );
        return fn.apply( null, args );
    };
}

var fooThunk = thunkify( foo, 3, 4 );
// 将来

fooThunk( function(sum) {
    console.log( sum );      // 7
} );
```

这里我们假定原始（foo(..)）函数原型需要的回调放在最后的位置，其他参数都在它之前。对异步 JavaScript 函数标准来说，这可以说是一个普遍成立的标准。你可以称之为“callback-last 风格”。如果出于某种原因需要处理“callback-first 风格”原型，你可以构建一个使用 args.unshift(..) 而不是 args.push(..) 的工具。

前面 thunkify(..) 的实现接收 foo(..) 函数引用以及它需要的任意参数，并返回 thunk 本身（fooThunk(..)）。但是，这并不是 JavaScript 中使用 thunk 的典型方案。

典型的方法——如果不令人迷惑的话——并不是 thunkify(..) 构造 thunk 本身，而是 thunkify(..) 工具产生一个生成 thunk 的函数。

考虑：

```
function thunkify(fn) {
    return function() {
        var args = [].slice.call( arguments );
        return function(cb) {
            args.push( cb );
            return fn.apply( null, args );
        };
    };
}
```

此处主要的区别在于多出来的 return function() { .. } 这一层。以下是用法上的区别：

```
var whatIsThis = thunkify( foo );

var fooThunk = whatIsThis( 3, 4 );
```

```
// 将来
fooThunk( function(sum) {
    console.log( sum );      // 7
} );
```

显然，这段代码暗藏的一个大问题是：whatIsThis 调用的是什么。并不是这个 thunk，而是某个从 foo(..) 调用产生 thunk 的东西。这有点类似于 thunk 的“工厂”。似乎还没有任何标准约定可以给这样的东西命名。

所以我的建议是 thunkory（thunk+factory）。于是就有，thunkify(..) 生成一个 thunkory，然后 thunkory 生成 thunk。这和第 3 章中我提议 promisory 出于同样的原因：

```
var fooThunkory = thunkify( foo );

var fooThunk1 = fooThunkory( 3, 4 );
var fooThunk2 = fooThunkory( 5, 6 );

// 将来

fooThunk1( function(sum) {
    console.log( sum );     // 7
} );

fooThunk2( function(sum) {
    console.log( sum );     // 11
} );
```

foo(..) 例子要求回调的风格不是 error-first 风格。当然，error-first 风格要常见得多。如果 foo(..) 需要满足一些正统的错误生成期望，可以把它按照期望改造，使用一个 error-first 回调。后面的 thunkify(..) 机制都不关心回调的风格。使用上唯一的区别将会是 fooThunk1(function(err,sum){..。

暴露 thunkory 方法——而不是像前面的 thunkify(..) 那样把这个中间步骤隐藏——似乎是不必要的复杂性。但是，一般来说，在程序开头构造 thunkory 来封装已有的 API 方法，并在需要 thunk 时可以传递和调用这些 thunkory，是很有用的。两个独立的步骤保留了一个更清晰的功能分离。

以下代码可说明这一点：

```
// 更简洁：
var fooThunkory = thunkify( foo );

var fooThunk1 = fooThunkory( 3, 4 );
var fooThunk2 = fooThunkory( 5, 6 );

// 而不是：
```

```
var fooThunk1 = thunkify( foo, 3, 4 );
var fooThunk2 = thunkify( foo, 5, 6 );
```

不管你是否愿意显式地与 thunkory 打交道，thunk `fooThunk1(..)` 和 `fooThunk2(..)` 的用法都是一样的。

s/promise/thunk/

那么所有这些关于 thunk 的内容与生成器有什么关系呢？

可以把 thunk 和 promise 大体上对比一下：它们的特性并不相同，所以并不能直接互换。Promise 要比裸 thunk 功能更强、更值得信任。

但从另外一个角度来说，它们都可以被看作是对一个值的请求，回答可能是异步的。

回忆一下，在第 3 章里我们定义了一个工具用于 promise 化一个函数，我们称之为 `Promise.wrap(..)`，也可以将其称为 `promisify(..)`！这个 Promise 封装工具并不产生 Promise，它生成的是 promisory，而 promisory 则接着产生 Promise。这和现在讨论的 thunkory 和 thunk 是完全对称的。

为了说明这种对称性，我们要首先把前面的 `foo(..)` 例子修改一下，改成使用 error-first 风格的回调：

```
function foo(x,y,cb) {
    setTimeout( function(){
        // 假定cb(..)是error-first风格的
        cb( null, x + y );
    }, 1000 );
}
```

现在我们对比一下 `thunkify(..)` 和 `promisify(..)`（即第 3 章中的 `Promise.wrap(..)`）的使用：

```
// 对称:构造问题提问者
var fooThunkory = thunkify( foo );
var fooPromisory = promisify( foo );

// 对称:提问
var fooThunk = fooThunkory( 3, 4 );
var fooPromise = fooPromisory( 3, 4 );

// 得到答案
fooThunk( function(err,sum){
    if (err) {
        console.error( err );
    }
    else {
        console.log( sum );    // 7
```

```
    }
} );

// 得到promise答案
fooPromise
.then(
    function(sum){
        console.log( sum );     // 7
    },
    function(err){
        console.error( err );
    }
);
```

thunkory 和 promisory 本质上都是在提出一个请求（要求一个值），分别由 thunk `fooThunk` 和 promise `fooPromise` 表示对这个请求的未来的答复。这样考虑的话，这种对称性就很清晰了。

了解了这个视角之后，就可以看出，`yield` 出 Promise 以获得异步性的生成器，也可以为异步性而 `yield` thunk。我们所需要的只是一个更智能的 `run(..)` 工具（就像前面的一样），不但能够寻找和链接 `yield` 出来的 Promise，还能够向 `yield` 出来的 thunk 提供回调。

考虑：

```
function *foo() {
    var val = yield request( "http://some.url.1" );
    console.log( val );
}
run( foo );
```

在这个例子中，`request(..)` 可能是一个返回 promise 的 promisory，也可能是一个返回 thunk 的 thunkory。从生成器内部的代码逻辑的角度来说，我们并不关心这个实现细节，这一点是非常强大的！

于是，`request(..)` 可能是以下两者之一：

```
// promisory request(..)（参见第3章）
var request = Promise.wrap( ajax );

// vs.

// thunkory request(..)
var request = thunkify( ajax );
```

最后，作为前面 `run(..)` 工具的一个支持 thunk 的补丁，我们还需要这样的逻辑：

```
// ..
// 我们收到返回的thunk了吗？
else if (typeof next.value == "function") {
    return new Promise( function(resolve,reject){
```

```
            // 用error-first回调调用这个thunk
            next.value( function(err,msg) {
                if (err) {
                    reject( err );
                }
                else {
                    resolve( msg );
                }
            } );
        } )
        .then(
            handleNext,
            function handleErr(err) {
                return Promise.resolve(
                    it.throw( err )
                )
                .then( handleResult );
            }
        );
    }
```

现在，我们的生成器可以调用 promisory 来 `yield` Promise，也可以调用 thunkory 来 `yield` thunk。不管哪种情况，`run(..)` 都能够处理这个值，并等待它的完成来恢复生成器运行。

从对称性来说，这两种方案看起来是一样的。但应该指出，这只是从代表生成器的未来值 continuation 的 Promise 或 thunk 的角度说才是正确的。

从更大的角度来说，thunk 本身基本上没有任何可信任性和可组合性保证，而这些是 Promise 的设计目标所在。单独使用 thunk 作为 Pormise 的替代在这个特定的生成器异步模式里是可行的，但是与 Promise 具备的优势（参见第 3 章）相比，这应该并不是一种理想方案。

如果可以选择的话，你应该使用 `yield pr` 而不是 `yield th`。但对 `run(..)` 工具来说，对两种值类型都能提供支持则是完全正确的。

我的 asynquence 库（详见附录 A）中的 `runner(..)` 工具可以处理 Promise、thunk 和 asynquence 序列的 `yield`。

4.8 ES6 之前的生成器

现在，希望你已经相信，生成器是异步编程工具箱中新增的一种非常重要的工具。但是，这是 ES6 中新增的语法，这意味着你没法像对待 Promise（这只是一种新的 API）那样使用生成器。所以如果不能忽略 ES6 前的浏览器的话，怎么才能把生成器引入到我们的浏览