生成器的名字大多来自这种消费生产值（consuming produced values）的用例。但是，这里要再次申明，这只是生成器的用法之一，坦白地说，甚至不是这本书重点关注的用途。

既然对生成器的工作机制有了更完整的理解，那接下来就可以把关注转向如何把生成器应用于异步并发了。

4.3 异步迭代生成器

生成器与异步编码模式及解决回调问题等，有什么关系呢？让我们来回答这个重要的问题。

我们应该重新讨论第 3 章中的一个场景。回想一下回调方法：

```
function foo(x,y,cb) {
    ajax(
        "http://some.url.1/?x=" + x + "&y=" + y,
        cb
    );
}

foo( 11, 31, function(err,text) {
    if (err) {
        console.error( err );
    }
    else {
        console.log( text );
    }
} );
```

如果想要通过生成器来表达同样的任务流程控制，可以这样实现：

```
function foo(x,y) {
    ajax(
        "http://some.url.1/?x=" + x + "&y=" + y,
        function(err,data){
            if (err) {
                // 向*main()抛出一个错误
                it.throw( err );
            }
            else {
                // 用收到的data恢复*main()
                it.next( data );
            }
        }
    );
}

function *main() {
    try {
        var text = yield foo( 11, 31 );
```

```
        console.log( text );
    }
    catch (err) {
        console.error( err );
    }
}

var it = main();

// 这里启动!
it.next();
```

第一眼看上去，与之前的回调代码对比起来，这段代码更长一些，可能也更复杂一些。但是，不要被表面现象欺骗了！生成器代码实际上要好得多！不过要解释这一点还是比较复杂的。

首先，让我们查看一下最重要的这段代码：

```
var text = yield foo( 11, 31 );
console.log( text );
```

请先花点时间思考一下这段代码是如何工作的。我们调用了一个普通函数 `foo(..)`，而且显然能够从 Ajax 调用中得到 `text`，即使它是异步的。

这怎么可能呢？如果你回想一下第 1 章的开始部分的话，我们给出了几乎相同的代码：

```
var data = ajax( "..url 1.." );
console.log( data );
```

但是，这段代码不能工作！你能指出其中的区别吗？区别就在于生成器中使用的 `yield`。

这就是奥秘所在！正是这一点使得我们看似阻塞同步的代码，实际上并不会阻塞整个程序，它只是暂停或阻塞了生成器本身的代码。

在 `yield foo(11,31)` 中，首先调用 `foo(11,31)`，它没有返回值（即返回 `undefined`），所以我们发出了一个调用来请求数据，但实际上之后做的是 `yield undefined`。这没问题，因为这段代码当前并不依赖 `yield` 出来的值来做任何事情。本章后面会再次讨论这一点。

这里并不是在消息传递的意义上使用 `yield`，而只是将其用于流程控制实现暂停 / 阻塞。实际上，它还是会有消息传递，但只是生成器恢复运行之后的单向消息传递。

所以，生成器在 `yield` 处暂停，本质上是在提出一个问题：“我应该返回什么值来赋给变量 `text`？”谁来回答这个问题呢？

看一下 `foo(..)`。如果这个 Ajax 请求成功，我们调用：

```
it.next( data );
```

这会用响应数据恢复生成器，意味着我们暂停的 yield 表达式直接接收到了这个值。然后随着生成器代码继续运行，这个值被赋给局部变量 text。

很酷吧？

回头往前看一步，思考一下这意味着什么。我们在生成器内部有了看似完全同步的代码（除了 yield 关键字本身），但隐藏在背后的是，在 foo(..) 内的运行可以完全异步。

这是巨大的改进！对于我们前面陈述的回调无法以顺序同步的、符合我们大脑思考模式的方式表达异步这个问题，这是一个近乎完美的解决方案。

从本质上而言，我们把异步作为实现细节抽象了出去，使得我们可以以同步顺序的形式追踪流程控制："发出一个 Ajax 请求，等它完成之后打印出响应结果。"并且，当然，我们只在这个流程控制中表达了两个步骤，而这种表达能力是可以无限扩展的，以便我们无论需要多少步骤都可以表达。

这是一个很重要的领悟，回过头去把上面三段重读一遍，让它融入你的思想吧！

同步错误处理

前面的生成器代码甚至还给我们带来了更多其他的好处。让我们把注意力转移到生成器内部的 try..catch：

```
try {
    var text = yield foo( 11, 31 );
    console.log( text );
}
catch (err) {
    console.error( err );
}
```

这是如何工作的呢？调用 foo(..) 是异步完成的，难道 try..catch 不是无法捕获异步错误，就像我们在第 3 章中看到的一样吗？

我们已经看到 yield 是如何让赋值语句暂停来等待 foo(..) 完成，使得响应完成后可以被赋给 text。精彩的部分在于 yield 暂停也使得生成器能够捕获错误。通过这段前面列出的代码把错误抛出到生成器中：

```
if (err) {
    // 向*main()抛出一个错误
```

```
    it.throw( err );
}
```

生成器 yield 暂停的特性意味着我们不仅能够从异步函数调用得到看似同步的返回值，还可以同步捕获来自这些异步函数调用的错误！

所以我们已经知道，我们可以把错误抛入生成器中，不过如果是从生成器向外抛出错误呢？正如你所料：

```
function *main() {
    var x = yield "Hello World";

    yield x.toLowerCase();  // 引发一个异常!
}

var it = main();

it.next().value;            // Hello World

try {
    it.next( 42 );
}
catch (err) {
    console.error( err );   // TypeError
}
```

当然，也可以通过 throw .. 手工抛出一个错误，而不是通过触发异常。

甚至可以捕获通过 throw(..) 抛入生成器的同一个错误，基本上也就是给生成器一个处理它的机会；如果没有处理的话，迭代器代码就必须处理：

```
function *main() {
    var x = yield "Hello World";

    // 永远不会到达这里
    console.log( x );
}

var it = main();

it.next();

try {
    // *main()会处理这个错误吗？看看吧!
    it.throw( "Oops" );
}
catch (err) {

    // 不行,没有处理!
    console.error( err );           // Oops
}
```

在异步代码中实现看似同步的错误处理（通过 try..catch）在可读性和合理性方面都是一个巨大的进步。

4.4 生成器 +Promise

在前面的讨论中，我们展示了如何异步迭代生成器，这是一团乱麻似的回调在顺序性和合理性方面的巨大进步。但我们错失了很重要的两点：Promise 的可信任性和可组合性（参见第 3 章）！

别担心，我们还会重获这些。ES6 中最完美的世界就是生成器（看似同步的异步代码）和 Promise（可信任可组合）的结合。

但如何实现呢？

回想一下第 3 章里在运行 Ajax 例子中基于 Promise 的实现方法：

```
function foo(x,y) {
    return request(
        "http://some.url.1/?x=" + x + "&y=" + y
    );
}

foo( 11, 31 )
.then(
    function(text){
        console.log( text );
    },
    function(err){
        console.error( err );
    }
);
```

在前面的运行 Ajax 例子的生成器代码中，foo(..) 没有返回值（undefined），并且我们的迭代器控制代码并不关心 yield 出来的值。

而这里支持 Promise 的 foo(..) 在发出 Ajax 调用之后返回了一个 promise。这暗示我们可以通过 foo(..) 构造一个 promise，然后通过生成器把它 yield 出来，然后迭代器控制代码就可以接收到这个 promise 了。

但迭代器应该对这个 promise 做些什么呢？

它应该侦听这个 promise 的决议（完成或拒绝），然后要么使用完成消息恢复生成器运行，要么向生成器抛出一个带有拒绝原因的错误。

我再重复一遍，因为这一点非常重要。获得 Promise 和生成器最大效用的最自然的方法就是 yield 出来一个 Promise，然后通过这个 Promise 来控制生成器的迭代器。

让我们来试一下！首先，把支持 Promise 的 `foo(..)` 和生成器 `*main()` 放在一起：

```
function foo(x,y) {
    return request(
        "http://some.url.1/?x=" + x + "&y=" + y
    );
}

function *main() {
    try {
        var text = yield foo( 11, 31 );
        console.log( text );
    }
    catch (err) {
        console.error( err );
    }
}
```

这次重构代码中最有力的发现是，`*main()` 之中的代码完全不需要改变！在生成器内部，不管什么值 `yield` 出来，都只是一个透明的实现细节，所以我们甚至没有意识到其发生，也不需要关心。

但现在如何运行 `*main()` 呢？还有一些实现细节需要补充，来实现接收和连接 `yield` 出来的 promise，使它能够在决议之后恢复生成器。先从手工实现开始：

```
var it = main();

var p = it.next().value;

// 等待promise p决议
p.then(
    function(text){
        it.next( text );
    },

    function(err){
        it.throw( err );
    }
);
```

实际上，这并没有那么令人痛苦，对吧？

这段代码看起来应该和我们前面手工组合通过 error-first 回调控制的生成器非常类似。除了没有 `if (err) { it.throw..`，promise 已经为我们分离了完成（成功）和拒绝（失败），否则的话，迭代器控制是完全一样的。

现在，我们已经隐藏了一些重要的细节。

最重要的是，我们利用了已知 `*main()` 中只有一个需要支持 Promise 的步骤这一事实。如果想要能够实现 Promise 驱动的生成器，不管其内部有多少个步骤呢？我们当然不希望每

个生成器手工编写不同的 Promise 链！如果有一种方法可以实现重复（即循环）迭代控制，每次会生成一个 Promise，等其决议后再继续，那该多好啊。

还有，如果在 `it.next(..)` 调用过程中生成器（有意或无意）抛出一个错误会怎样呢？是应该退出呢，还是应该捕获这个错误并发送回去呢？类似地，如果通过 `it.throw(..)` 把一个 Promise 拒绝抛入生成器中，但它却没有受到处理就被直接抛回了呢？

4.4.1 支持 Promise 的 Generator Runner

随着对这条道路的深入探索，你越来越会意识到："哇，如果有某个工具为我实现这些就好了。"关于这一点，你绝对没错。这是如此重要的一个模式，你绝对不希望搞错（或精疲力竭地一次又一次重复实现），所以最好是使用专门设计用来以我们前面展示的方式运行 Promise-yielding 生成器的工具。

有几个 Promise 抽象库提供了这样的工具，包括我的 asynquence 库及其 `runner(..)`，本部分的附录 A 中会介绍。

但是，为了学习和展示的目的，我们还是自己定义一个独立工具，叫作 `run(..)`：

```
// 在此感谢Benjamin Gruenbaum (@benjamingr on GitHub)的巨大改进!
function run(gen) {
    var args = [].slice.call( arguments, 1), it;

    // 在当前上下文中初始化生成器
    it = gen.apply( this, args );

    // 返回一个promise用于生成器完成
    return Promise.resolve()
        .then( function handleNext(value){
            // 对下一个yield出的值运行
            var next = it.next( value );

            return (function handleResult(next){
                // 生成器运行完毕了吗?
                if (next.done) {
                    return next.value;
                }
                // 否则继续运行
                else {
                    return Promise.resolve( next.value )
                        .then(
                            // 成功就恢复异步循环,把决议的值发回生成器
                            handleNext,

                            // 如果value是被拒绝的 promise,
                            // 就把错误传回生成器进行出错处理
                            function handleErr(err) {
                                return Promise.resolve(
                                    it.throw( err )
```

```
                        )
                        .then( handleResult );
                    }
                );
            }
        })(next);
    } );
}
```

诚如所见，你可能并不愿意编写这么复杂的工具，并且也会特别不希望为每个使用的生成器都重复这段代码。所以，一个工具或库中的辅助函数绝对是必要的。尽管如此，我还是建议你花费几分钟时间学习这段代码，以更好地理解生成器 +Promise 协同运作模式。

如何在运行 Ajax 的例子中使用 `run(..)` 和 `*main()` 呢？

```
function *main() {
    // ..
}

run( main );
```

就是这样！这种运行 `run(..)` 的方式，它会自动异步运行你传给它的生成器，直到结束。

我们定义的 `run(..)` 返回一个 promise，一旦生成器完成，这个 promise 就会决议，或收到一个生成器没有处理的未捕获异常。这里并没有展示这种功能，但我们会在本章后面部分再介绍这一点。

ES7：async 与 await？

前面的模式——生成器 `yield` 出 Promise，然后其控制生成器的迭代器来执行它，直到结束——是非常强大有用的一种方法。如果我们能够无需库工具辅助函数（即 `run(..)`）就能够实现就好了。

关于这一点，可能有一些好消息。在编写本书的时候，对于后 ES6、ES7 的时间框架，在这一方面增加语法支持的提案已经有了一些初期但很强势的支持。显然，现在确定细节还太早，但其形式很可能会类似如下：

```
function foo(x,y) {
    return request(
        "http://some.url.1/?x=" + x + "&y=" + y
    );
}

async function main() {
    try {
        var text = await foo( 11, 31 );
        console.log( text );
    }
```

```
    catch (err) {
        console.error( err );
    }
}

main();
```

可以看到，这里没有通过 `run(..)` 调用（意味着不需要库工具！）来触发和驱动 `main()`，它只是被当作一个普通函数调用。另外，`main()` 也不再被声明为生成器函数了，它现在是一类新的函数：`async` 函数。最后，我们不再 `yield` 出 Promise，而是用 `await` 等待它决议。

如果你 `await` 了一个 Promise，`async` 函数就会自动获知要做什么，它会暂停这个函数（就像生成器一样），直到 Promise 决议。我们并没有在这段代码中展示这一点，但是调用一个像 `main()` 这样的 `async` 函数会自动返回一个 promise。在函数完全结束之后，这个 promise 会决议。

有 C# 经验的人可能很熟悉 `async/await` 语法，因为它们基本上是相同的。

从本质上说，这个提案就是把前面我们已经推导出来的模式写进规范，使其进入语法机制：组合 Promise 和看似同步的流程控制代码。这是两个最好的世界的结合，有效地实际解决了我们列出的回调方案的主要问题。

这样的 ES7 提案已经存在，并有了初期的支持和热情，仅仅是这个事实就极大增加了这个异步模式对其未来重要性的信心。

4.4.2 生成器中的 Promise 并发

到目前为止，我们已经展示的都是 Promise+ 生成器下的单步异步流程。但是，现实世界中的代码常常会有多个异步步骤。

如果不认真对待的话，生成器的这种看似同步的风格可能会让你陷入对自己异步并发组织方式的自满中，进而导致并不理想的性能模式。所以我们打算花点时间来研究一下各种方案。

想象这样一个场景：你需要从两个不同的来源获取数据，然后把响应组合在一起以形成第三个请求，最终把最后一条响应打印出来。第 3 章已经用 Promise 研究过一个类似的场景，但是让我们在生成器的环境下重新考虑一下这个问题吧。

你的第一直觉可能类似如下：

```
function *foo() {
    var r1 = yield request( "http://some.url.1" );
    var r2 = yield request( "http://some.url.2" );

    var r3 = yield request(
        "http://some.url.3/?v=" + r1 + "," + r2
    );

    console.log( r3 );
}

// 使用前面定义的工具run(..)
run( foo );
```

这段代码可以工作，但是针对我们特定的场景而言，它并不是最优的。你能指出原因吗？

因为请求 r1 和 r2 能够——出于性能考虑也应该——并发执行，但是在这段代码中，它们是依次执行的，直到请求 URL"http://some.url.1" 完成后才会通过 Ajax 获取 URL"http://some.url.2"。这两个请求是相互独立的，所以性能更高的方案应该是让它们同时运行。

但是，到底如何通过生成器和 yield 实现这一点呢？我们知道 yield 只是代码中一个单独的暂停点，并不可能同时在两个点上暂停。

最自然有效的答案就是让异步流程基于 Promise，特别是基于它们以时间无关的方式管理状态的能力（参见 3.1.1 节）。

最简单的方法：

```
function *foo() {
    // 让两个请求"并行"
    var p1 = request( "http://some.url.1" );
    var p2 = request( "http://some.url.2" );

    // 等待两个promise都决议
    var r1 = yield p1;
    var r2 = yield p2;

    var r3 = yield request(
        "http://some.url.3/?v=" + r1 + "," + r2

    );

    console.log( r3 );
}

// 使用前面定义的工具run(..)
run( foo );
```

为什么这和前面的代码片段不同呢？观察一下 yield 的位置。p1 和 p2 是并发执行（即

“并行”）的用于 Ajax 请求的 promise。哪一个先完成都无所谓，因为 promise 会按照需要在决议状态保持任意长时间。

然后我们使用接下来的两个 `yield` 语句等待并取得 promise 的决议（分别写入 `r1` 和 `r2`）。如果 `p1` 先决议，那么 `yield p1` 就会先恢复执行，然后等待 `yield p2` 恢复。如果 `p2` 先决议，它就会耐心保持其决议值等待请求，但是 `yield p1` 将会先等待，直到 `p1` 决议。

不管哪种情况，`p1` 和 `p2` 都会并发执行，无论完成顺序如何，两者都要全部完成，然后才会发出 `r3 = yield request..`Ajax 请求。

这种流程控制模型如果听起来有点熟悉的话，是因为这基本上和我们在第 3 章中通过 `Promise.all([ .. ])` 工具实现的 gate 模式相同。因此，也可以这样表达这种流程控制：

```
function *foo() {
    // 让两个请求"并行",并等待两个promise都决议
    var results = yield Promise.all( [
        request( "http://some.url.1" ),
        request( "http://some.url.2" )
    ] );

    var r1 = results[0];
    var r2 = results[1];

    var r3 = yield request(
        "http://some.url.3/?v=" + r1 + "," + r2
    );

    console.log( r3 );
}

// 使用前面定义的工具run(..)
run( foo );
```

就像我们在第 3 章中讨论过的，我们甚至可以通过 ES6 解构赋值，把 `var r1 = .. var r2 = ..` 赋值语句简化为 `var [r1,r2] = results`。

换句话说，Promise 所有的并发能力在生成器 +Promise 方法中都可以使用。所以无论在什么地方你的需求超过了顺序的 this-then-that 异步流程控制，Promise 很可能都是最好的选择。

隐藏的 Promise

作为一个风格方面的提醒：要注意你的生成器内部包含了多少 Promise 逻辑。我们介绍的使用生成器实现异步的方法的全部要点在于创建简单、顺序、看似同步的代码，将异步的细节尽可能隐藏起来。

比如，这可能是一个更简洁的方案：

```
// 注:普通函数,不是生成器
function bar(url1,url2) {
    return Promise.all( [
        request( url1 ),
        request( url2 )
    ] );
}

function *foo() {
    // 隐藏bar(..)内部基于Promise的并发细节
    var results = yield bar(
        "http://some.url.1",
        "http://some.url.2"
    );

    var r1 = results[0];
    var r2 = results[1];

    var r3 = yield request(
        "http://some.url.3/?v=" + r1 + "," + r2
    );

    console.log( r3 );
}

// 使用前面定义的工具run(..)
run( foo );
```

在 `*foo()` 内部，我们所做的一切就是要求 `bar(..)` 给我们一些 `results`，并通过 `yield` 来等待结果，这样更简洁也更清晰。我们不需要关心在底层是用 `Promise.all([ .. ])` Promise 组合来实现这一切。

我们把异步，实际上是 Promise，作为一个实现细节看待。

如果想要实现一系列高级流程控制的话，那么非常有用的做法是：把你的 Promise 逻辑隐藏在一个只从生成器代码中调用的函数内部。比如：

```
function bar() {
    Promise.all( [
        baz( .. )
        .then( .. ),
        Promise.race( [ .. ] )
    ] )
    .then( .. )
}
```

有时候会需要这种逻辑，而如果把它直接放在生成器内部的话，那你就失去了几乎所有一开始使用生成器的理由。应该有意将这样的细节从生成器代码中抽象出来，以避免它把高层次的任务表达变得杂乱。

创建代码除了要实现功能和保持性能之外，你还应该尽可能使代码易于理解和维护。

对编程来说，抽象并不总是好事，很多时候它会增加复杂度以换取简洁性。但是在这个例子里，我相信，对生成器 +Promise 异步代码来说，相比于其他实现，这种抽象更加健康。尽管如此，还是建议大家要注意具体情况具体分析，为你和你的团队作出正确的决定。

4.5 生成器委托

在前面一节中，我们展示了从生成器内部调用常规函数，以及这如何对于把实现细节（就像异步 Promise 流）抽象出去还是一种有用的技术。但是，用普通函数实现这个任务的主要缺点是它必须遵守普通函数的规则，也就意味着它不能像生成器一样用 yield 暂停自己。

可能出现的情况是，你可能会从一个生成器调用另一个生成器，使用辅助函数 run(..)，就像这样：

```
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" );

    // 通过 run(..) "委托"给*foo()
    var r3 = yield run( foo );

    console.log( r3 );
}

run( bar );
```

我们再次通过 `run(..)` 工具从 `*bar()` 内部运行 `*foo()`。这里我们利用了如下事实：我们前面定义的 `run(..)` 返回一个 promise，这个 promise 在生成器运行结束时（或出错退出时）决议。因此，如果从一个 `run(..)` 调用中 `yield` 出来一个 promise 到另一个 `run(..)` 实例中，它会自动暂停 `*bar()`，直到 `*foo()` 结束。

但其实还有一个更好的方法可以实现从 `*bar()` 调用 `*foo()`，称为 `yield` 委托。`yield` 委托的具体语法是：`yield * __`（注意多出来的 `*`）。在我们弄清它在前面的例子中的使用之前，先来看一个简单点的场景：

```
function *foo() {
    console.log( "*foo() starting" );
```

```
    yield 3;
    yield 4;
    console.log( "*foo() finished" );
}

function *bar() {
    yield 1;
    yield 2;
    yield *foo();    // yield委托!
    yield 5;
}

var it = bar();

it.next().value;     // 1
it.next().value;     // 2
it.next().value;     // *foo()启动
                     // 3
it.next().value;     // 4
it.next().value;     // *foo()完成
                     // 5
```

在本章前面的一条提示中，我解释了为什么我更喜欢 `function *foo() ..`，而不是 `function* foo() ..`。类似地，我也更喜欢——与这个主题的多数其他文档不同——使用 `yield *foo()` 而不是 `yield* foo()`。`*` 的位置仅关乎风格，由你自己来决定使用哪种。不过我发现保持风格一致是很吸引人的。

这里的 `yield *foo()` 委托是如何工作的呢？

首先，和我们以前看到的完全一样，调用 `foo()` 创建一个迭代器。然后 `yield *` 把迭代器实例控制（当前 `*bar()` 生成器的）委托给 / 转移到了这另一个 `*foo()` 迭代器。

所以，前面两个 `it.next()` 调用控制的是 `*bar()`。但当我们发出第三个 `it.next()` 调用时，`*foo()` 现在启动了，我们现在控制的是 `*foo()` 而不是 `*bar()`。这也是为什么这被称为委托：`*bar()` 把自己的迭代控制委托给了 `*foo()`。

一旦 `it` 迭代器控制消耗了整个 `*foo()` 迭代器，`it` 就会自动转回控制 `*bar()`。

现在回到前面使用三个顺序 Ajax 请求的例子：

```
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" );
```

```
    // 通过 yeild* "委托"给*foo()
    var r3 = yield *foo();

    console.log( r3 );
}

run( bar );
```

这段代码和前面版本的唯一区别就在于使用了 `yield *foo()`，而不是前面的 `yield run(foo)`。

`yield *` 暂停了迭代控制，而不是生成器控制。当你调用 `*foo()` 生成器时，现在 `yield` 委托到了它的迭代器。但实际上，你可以 `yield` 委托到任意 iterable，`yield *[1,2,3]` 会消耗数组值 `[1,2,3]` 的默认迭代器。

4.5.1 为什么用委托

`yield` 委托的主要目的是代码组织，以达到与普通函数调用的对称。

想像一下有两个模块分别提供了方法 `foo()` 和 `bar()`，其中 `bar()` 调用了 `foo()`。一般来说，把两者分开实现的原因是该程序的适当的代码组织要求它们位于不同的函数中。比如，可能有些情况下是单独调用 `foo()`，另外一些地方则由 `bar()` 调用 `foo()`。

同样是出于这些原因，保持生成器分离有助于程序的可读性、可维护性和可调试性。在这一方面，`yield *` 是一个语法上的缩写，用于代替手工在 `*foo()` 的步骤上迭代，不过是在 `*bar()` 内部。

如果 `*foo()` 内的步骤是异步的话，这样的手工方法将会特别复杂，这也是你可能需要使用 `run(..)` 工具来做某些事情的原因。就像我们已经展示的，`yield *foo()` 消除了对 `run(..)` 工具的需要（就像 `run(foo)`）。

4.5.2 消息委托

你可能会疑惑，这个 `yield` 委托是如何不只用于迭代器控制工作，也用于双向消息传递工作的呢。认真跟踪下面的通过 `yield` 委托实现的消息流出入：

```
function *foo() {
    console.log( "inside *foo():", yield "B" );

    console.log( "inside *foo():", yield "C" );

    return "D";
}

function *bar() {
```