```
steps
.then( function STEP1(x){
    return x * 2;
} )
.steps( function STEP2(x){
    return x + 3;
} )
.steps( function STEP3(x){
    return x * 4;
} );

steps.next( 8 ).value;  // 16
steps.next( 16 ).value; // 19
steps.next( 19 ).value; // 76
steps.next().done;      // true
```

可以看到，可迭代序列是一个符合标准的迭代器（参见第 4 章）。因此，可通过 ES6 的 `for..of` 循环迭代，就像生成器（或其他任何 iterable）一样：

```
var steps = ASQ.iterable();

steps
.then( function STEP1(){ return 2; } )
.then( function STEP2(){ return 4; } )
.then( function STEP3(){ return 6; } )
.then( function STEP4(){ return 8; } )
.then( function STEP5(){ return 10; } );

for (var v of steps) {
    console.log( v );
}
// 2 4 6 8 10
```

除了附录 A 中的事件触发示例之外，可迭代序列的有趣之处在于它们从本质上可以看作是一个生成器或 Promise 链的替身，但其灵活性却更高。

请考虑一个多 Ajax 请求的例子。我们在第 3 章和第 4 章中已经看到过同样的场景，分别通过 Promise 链和生成器实现的。用可迭代序列来表达：

```
// 支持序列的ajax
var request = ASQ.wrap( ajax );

ASQ( "http://some.url.1" )
.runner(
    ASQ.iterable()

    .then( function STEP1(token){
        var url = token.messages[0];
        return request( url );
    } )

    .then( function STEP2(resp){
```

```
        return ASQ().gate(
            request( "http://some.url.2/?v=" + resp ),
            request( "http://some.url.3/?v=" + resp )
        );
    } )

    .then( function STEP3(r1,r2){ return r1 + r2; } )
)
.val( function(msg){
    console.log( msg );
} );
```

可迭代序列表达了一系列顺序的（同步或异步的）步骤，看起来和 Promise 链非常相似。换句话说，它比直接的回调嵌套看起来要简洁得多，但没有生成器的基于 yield 的顺序语法那么好。

但我们把可迭代序列传入了 `ASQ#runner(..)`，这个函数会把该序列执行完毕，就像对待生成器那样。可迭代序列本质上和生成器的行为方式一样。这个事实值得注意，原因如下。

首先，可迭代序列是 ES6 生成器某个子集的某种前 ES6 等价物。也就是说，你可以直接编写它们（在任意环境运行），或者你也可以编写 ES6 生成器，并将其重编译或转化为可迭代序列（就此而言，也可以是 Promise 链！）。

把“异步完整运行”的生成器看作是 Promise 链的语法糖，对于认识它们的同构关系是很重要的。

在继续之前，我们应该注意到，前面的代码片段可以用 asynquence 重写如下：

```
ASQ( "http://some.url.1" )
.seq( /*STEP 1*/ request )
.seq( function STEP2(resp){
    return ASQ().gate(
        request( "http://some.url.2/?v=" + resp ),
        request( "http://some.url.3/?v=" + resp )
    );
} )
.val( function STEP3(r1,r2){ return r1 + r2; } )
.val( function(msg){
    console.log( msg );
} );
```

而且，步骤 2 也可以这样写：

```
.gate(
    function STEP2a(done,resp) {
        request( "http://some.url.2/?v=" + resp )
        .pipe( done );
    },
    function STEP2b(done,resp) {
        request( "http://some.url.3/?v=" + resp )
```

```
            .pipe( done );
        }
    )
```

那么，如果更简单平凡的 asynquence 链就可以做得很好的话，为什么还要辛苦地把我们的流程控制表达为 `ASQ#runner(..)` 步骤中的可迭代序列呢？

因为可迭代序列形式还有很重要的秘密，提供了更强大的功能。请继续阅读。

可迭代序列扩展

生成器、普通 asynquence 序列以及 Promise 链都是*及早求值*（eagerly evaluated）——不管最初的流程控制是什么，都会执行这个固定的流程。

然而，可迭代序列是*惰性求值*（lazily evaluated），这意味着在可迭代序列的执行过程中，如果需要的话可以用更多的步骤扩展这个序列。

只能在可迭代序列的末尾添加步骤，不能插入序列的中间。

首先，让我们通过一个简单点的（同步）例子来熟悉一下这个功能：

```
function double(x) {
    x *= 2;

    // 应该继续扩展吗？
    if (x < 500) {
        isq.then( double );
    }

    return x;
}

// 建立单步迭代序列
var isq = ASQ.iterable().then( double );

for (var v = 10, ret;
    (ret = isq.next( v )) && !ret.done;
) {
    v = ret.value;
    console.log( v );
}
```

一开始这个可迭代序列只定义了一个步骤（`isq.then(double)`），但这个可迭代序列在某种条件下（`x < 500`）会持续扩展自己。严格说来，asynquence 序列和 Promise 链也可以实现

类似的功能，不过我们很快将说明为什么它们的能力是不足的。

尽管这个例子很平常，也可以通过一个生成器中的 `while` 循环表达，但我们会考虑到更复杂的情况。

举例来说，可以查看 Ajax 请求的响应，如果它指出还需要更多的数据，就有条件地向可迭代序列中插入更多的步骤来发出更多的请求。或者你也可以有条件地在 Ajax 处理结尾处增加一个值格式化的步骤。

考虑：

```
var steps = ASQ.iterable()

.then( function STEP1(token){
    var url = token.messages[0].url;

    // 提供了额外的格式化步骤了吗?
    if (token.messages[0].format) {
        steps.then( token.messages[0].format );
    }

    return request( url );
} )

.then( function STEP2(resp){
    // 向区列中添加一个Ajax请求吗?
    if (/x1/.test( resp )) {
        steps.then( function STEP5(text){
            return request(
                "http://some.url.4/?v=" + text
            );
        } );
    }

    return ASQ().gate(
        request( "http://some.url.2/?v=" + resp ),
        request( "http://some.url.3/?v=" + resp )
    );
} )

.then( function STEP3(r1,r2){ return r1 + r2; } );
```

你可以看到，在两个不同的位置处，我们有条件地使用 `steps.then(..)` 扩展了 `steps`。要运行这个可迭代序列 `steps`，只需要通过 `ASQ#runner(..)` 把它链入我们的带有 asynquence 序列（这里称为 `main`）的主程序流程：

```
var main = ASQ( {
    url: "http://some.url.1",
    format: function STEP4(text){
        return text.toUpperCase();
    }
```

```
} )
.runner( steps )
.val( function(msg){
    console.log( msg );
} );
```

可迭代序列 steps 的这一灵活性（有条件行为）可以用生成器表达吗？算是可以吧，但我们不得不以一种有点笨拙的方式重新安排这个逻辑：

```
function *steps(token) {
    // 步骤1
    var resp = yield request( token.messages[0].url );

    // 步骤2
    var rvals = yield ASQ().gate(
        request( "http://some.url.2/?v=" + resp ),
        request( "http://some.url.3/?v=" + resp )
    );

    // 步骤3
    var text = rvals[0] + rvals[1];

    // 步骤4
    //提供了额外的格式化步骤了吗?
    if (token.messages[0].format) {
        text = yield token.messages[0].format( text );
    }

    // 步骤5
    // 需要向序列中再添加一个Ajax请求吗?
    if (/foobar/.test( resp )) {
        text = yield request(
            "http://some.url.4/?v=" + text
        );
    }

    return text;
}

// 注意:*steps()可以和前面的steps一样被同一个ASQ序列运行
```

除了已经确认的生成器的顺序、看似同步的语法的好处（参见第 4 章），要模拟可扩展可迭代序列 steps 的动态特性，steps 的逻辑也需要以 *steps() 生成器形式重新安排。

而如果要通过 Promise 或序列来实现这个功能会怎样呢？你可以这么做：

```
var steps = something( .. )
.then( .. )
.then( function(..){
    // ..

    // 扩展链是吧?
    steps = steps.then( .. );
```

```
    // ..
})
.then( .. );
```

其中的问题捕捉起来比较微妙，但是很重要。所以，考虑要把我们的 steps Promise 链链入主程序流程。这次使用 Promise 来表达，而不是 asynquence：

```
var main = Promise.resolve( {
    url: "http://some.url.1",
    format: function STEP4(text){
        return text.toUpperCase();
    }
} )
.then( function(..){
    return steps;           // hint!
} )
.val( function(msg){
    console.log( msg );
} );
```

现在能看出问题所在了吗？仔细观察！

序列步骤排序有一个竞态条件。在你返回 steps 的时候，steps 这时可能是之前定义的 Promise 链，也可能是现在通过 steps = steps.then(..) 调用指向扩展后的 Promise 链。根据执行顺序的不同，结果可能不同。

以下是两个可能的结果。

- 如果 steps 仍然是原来的 Promise 链，一旦之后它通过 steps = steps.then(..) 被“扩展”，在链结尾处扩展之后的 promise 就不会被 main 流程考虑，因为它已经连到了 steps 链。很遗憾，这就是*及早求值*的局限性。
- 如果 steps 已经是扩展后的 Promise 链，它就会按预期工作，因为 main 连接的是扩展后的 promise。

除了竞态条件这个无法接受的事实，第一种情况也需要担心，它展示了 Promise 链的*及早求值*。与之对比的是，我们很容易扩展可迭代序列，且不会有这样的问题，因为可迭代序列是*惰性求值*的。

你所需的流程控制的动态性越强，可迭代序列的优势就越明显。

在 asynquence 网站上可以得到关于可迭代序列的更多信息和示例（https://github.com/getify/asynquence/blob/master/README.md#iterable-sequences）。

B.2 事件响应

（至少）根据第 3 章的内容，有一点应该是显而易见的，Promise 是异步工具箱中一个非常强大的工具。但是，因为 Promise 只能决议一次，它们的功能有一个很明显的缺憾就是处理事件流的能力。坦白地说，简单 asynquence 序列恰巧也有同样的弱点。

考虑这样一个场景，你想要在每次某个事件触发时都启动一系列步骤。单个 Promise 或序列不能代表一个事件的所有发生。因此，你不得不在每次事件发生时创建一整个新的 Promise 链（或序列），就像这样：

```
listener.on( "foobar", function(data){

    // 构造一个新的事件处理promise链
    new Promise( function(resolve,reject){
        // ..
    } )
    .then( .. )
    .then( .. );

} );
```

这个方法展示了我们需要的基本功能，但是离想要的表达期望逻辑的方式还差得很远。这个范式中合并了两个独立的功能：事件侦听和事件响应。独立的需求要求把这两个功能独立开来。

细心的人可能会观察到，这个问题和第 2 章中详细介绍的回调的问题类似。这是某种程度的控制反转问题。

设想一下，把这个范式的反转恢复一下，就像这样：

```
var observable = listener.on( "foobar" );

// 将来
observable
.then( .. )
.then( .. );

// 还有
observable
.then( .. )
.then( .. );
```

值 `observable` 并不完全是一个 Promise，但你可以像查看 Promise 一样查看它，所以它们是紧密相关的。实际上，它可以被查看多次，并且每次它的事件（`"foobar"`）发生的时候都会发出通知。

我刚刚展示的这个模式是响应式编程（RP）概念和动机的极大简化，这种模式已经被几个伟大的项目和语言实现 / 说明了。RP 的一个变体是函数式响应式编程（FRP），这是指对数据流应用函数式编程技术（不可变性、引用完整性，等等）。“响应式”是指把功能在时间上扩散以响应事件。如果感兴趣的话，你应该考虑研究一下微软在很棒的“响应式扩展”库（对于 JavaScript 来说是 RxJS）中提供的“响应式 Observable”（http://reactive-extensions.github.io/RxJS/）。它比我们这里展示的要高级和强大得多。另外，Andre Staltz 有一篇优秀的文章（https://gist.github.com/staltz/868e7e9bc2a7b8c1f754）赞扬了 RP 的高效，并给出了具体的例子。

B.2.1 ES7 Observable

在编写本书的时候，已经有早期的 ES7 提案提出了一个称为 Observable 的新数据类型，它的思路和我们这里给出的类似，不过肯定要更复杂一些。

这类 Observable 的概念是这样的：“订阅”到一个流的事件的方式是传入一个生成器——实际上其中有用的部分是迭代器——事件每次发生都会调用迭代器的 `next(..)` 方法。

你可以把它想象成类似这样：

```
// someEventStream是一个事件流,比如来自鼠标点击或其他

var observer = new Observer( someEventStream, function*(){
    while (var evt = yield) {
        console.log( evt );
    }
} );
```

传入的生成器将会 `yield` 暂停那个 `while` 循环，等待下一个事件。每次 `someEventStream` 发布一个新事件，都会调用到附加到生成器实例上的迭代器的 `next(..)`，因此事件数据会用 `evt` 数据恢复生成器 / 迭代器。

在这里的对事件的订阅功能中，重要的是迭代器部分，而不是生成器部分。所以，从概念上说，实际上你可以传入任何 iterable，包括 `ASQ.iterable()` 可迭代序列。

有趣的是，也有关于适配器的提案来简化从某些流类型构造 Observable，比如用于 DOM 事件的 `fromEvent(..)`。如果你查看前面给出链接的 ES7 提案中建议的 `fromEvent(..)` 实现，你会发现它看起来和我们在下一节将要看到的 `ASQ.react(..)` 惊人的相似。

当然，这些都是早期的提案，因此真正的最终实现可能和这里的展示在形式和行为方式上都有很大不同。但是，看到不同的库和语言提案之间对概念的早期整合还是很令人激动的！

B.2.2 响应序列

有了这个非常简要的 Observable（以及 F/RP）的概述给予我们灵感和激励，现在我要展示“响应式 Observable”的一个小子集的修改版，我称之为“响应式序列”。

首先，让我们从如何使用名为 `react(..)` 的 asynquence 插件工具创建一个 Observable 开始：

```
var observable = ASQ.react( function setup(next){
    listener.on( "foobar", next );
} );
```

现在，来看看如何定义一个能“响应”这个 `observable` 的序列（在 F/RP 中，这通常称为“订阅”）：

```
observable
.seq( .. )
.then( .. )
.val( .. );
```

所以，只要结束 Observable 链接就定义了序列。很简单，是不是？

在 F/RP 中，事件流通常从一系列函数变换中穿过，比如 `scan(..)`、`map(..)`、`reduce(..)`，等等。使用响应式序列的话，每个事件从一个序列的新实例中穿过。我们来看一个较具体的例子：

```
ASQ.react( function setup(next){
    document.getElementById( "mybtn" )
    .addEventListener( "click", next, false );
} )
.seq( function(evt){
    var btnID = evt.target.id;
    return request(
        "http://some.url.1/?id=" + btnID
    );
} )
.val( function(text){
    console.log( text );
} );
```

这个响应序列的“响应”部分来自于分配了一个或多个事件处理函数来调用事件触发器（调用 `next(..)`）。

响应序列的“序列”部分就和我们已经研究过的序列完全一样：每个步骤可以使用任意合理的异步技术，从 continuation 到 Promise 再到生成器。

一旦建立起响应序列，只要事件持续触发，它就会持续启动序列实例。如果想要停止响应序列，可以调用 `stop()`。

如果响应序列调用了 stop()，停止了，那你很可能希望事件处理函数也被注销。可以注册一个 teardown 处理函数来实现这个目的：

```
var sq = ASQ.react( function setup(next,registerTeardown){
    var btn = document.getElementById( "mybtn" );

    btn.addEventListener( "click", next, false );

    // 一旦sq.stop()被调用就会调用
    registerTeardown( function(){
        btn.removeEventListener( "click", next, false );
    } );
} )
.seq( .. )
.then( .. )
.val( .. );

// 将来
sq.stop();
```

处理函数 setup(..) 中的 this 绑定引用和响应序列 sq 一样，所以你可以使用这个 this 引用向响应序列定义添加内容，调用像 stop() 这样的方法，等等。

这里是一个来自 Node.js 世界的例子，使用了响应序列来处理到来的 HTTP 请求：

```
var server = http.createServer();
server.listen(8000);

// 响应式observer
var request = ASQ.react( function setup(next,registerTeardown){
    server.addListener( "request", next );
    server.addListener( "close", this.stop );

    registerTeardown( function(){
        server.removeListener( "request", next );
        server.removeListener( "close", request.stop );
    } );
});

// 响应请求
request
.seq( pullFromDatabase )
.val( function(data,res){
    res.end( data );
} );

// 节点清除
process.on( "SIGINT", request.stop );
```

使用 onStream(..) 和 unStream(..)，触发器 next(..) 也很容易适配节点流：

```
ASQ.react( function setup(next){
    var fstream = fs.createReadStream( "/some/file" );

    // 把流的"data"事件传给next(..)
    next.onStream( fstream );

    // 侦听流结尾
    fstream.on( "end", function(){
        next.unStream( fstream );
    } );
} )
.seq( .. )
.then( .. )
.val( .. );
```

也可以通过序列合并来组合多个响应序列流：

```
var sq1 = ASQ.react( .. ).seq( .. ).then( .. );
var sq2 = ASQ.react( .. ).seq( .. ).then( .. );

var sq3 = ASQ.react(..)
.gate(
    sq1,
    sq2
)
.then( .. );
```

主要的一点是：ASQ.react(..) 是 F/RP 概念的一个轻量级的修改版，也是术语“响应序列”的意义所在。响应序列通常能够胜任基本的响应式用途。

这里有一个使用 ASQ.react(..) 管理 UI 状态的例子（http://jsbin,com/rozipaki/6/edit?js,output），还有一个通过 ASQ.react(..) 处理 HTTP 请求 / 响应流的例子（https://gist.github.com/getify/bba5ec0de9d6047b720e）。

B.3 生成器协程

希望第 4 章已经帮助你熟悉了 ES6 生成器。具体来说，我们想要再次讨论“生成器并发”，甚至更加深入。

设想一个工具 runAll(..)，它能接受两个或更多的生成器，并且并发地执行它们，让它们依次进行合作式 yield 控制，并支持可选的消息传递。

除了可以运行单个生成器到结束之外，我们在附录 A 讨论的 ASQ#runner(..) 是 runAll(..) 概念的一个相似实现，后者可以并发运行多个生成器到结束。

因此，让我们来看看如何实现第 4 章中并发 Ajax 的场景：

```
ASQ(
    "http://some.url.2"
)
.runner(
    function*(token){
        // 传递控制
        yield token;

        var url1 = token.messages[0]; // "http://some.url.1"

        // 清空消息,重新开始
        token.messages = [];

        var p1 = request( url1 );

        // 传递控制
        yield token;

        token.messages.push( yield p1 );
    },
    function*(token){
        var url2 = token.messages[0]; // "http://some.url.2"

        // 传递消息并传递控制
        token.messages[0] = "http://some.url.1";
        yield token;

        var p2 = request( url2 );

        // 传递控制
        yield token;

        token.messages.push( yield p2 );

        // 把结果传给下一个序列步骤
        return token.messages;
    }
)
.val( function(res){
    // res[0]来自"http://some.url.1"
    // res[1]来自"http://some.url.2"
} );
```

`ASQ#runner(..)` 和 `runAll(..)` 之间的主要区别如下。

- 每个生成器（协程）都被提供了一个叫作 `token` 的参数。这是一个特殊的值，想要显式把控制传递到下一个协程的时候就 `yield` 这个值。
- `token.messages` 是一个数组，其中保存了从前面一个序列步骤传入的所有消息。它也是一个你可以用来在协程之间共享消息的数据结构。
- `yield` 一个 Promise（或序列）值不会传递控制，而是暂停这个协程处理，直到这个值准备好。

- 从协程处理运行最后 `return` 的或 `yield` 的值将会被传递到序列中的下一个步骤。

在基本的 `ASQ#runner(..)` 功能之上添加辅助函数用于不同的用途也是很容易实现的。

状态机

对许多程序员来说，一个可能很熟悉的例子就是状态机。在简单的装饰工具的帮助下，你可以创建一个很容易表达的状态机处理器。

让我们来设想这样一个工具。我们将其称为 `state(..)`，并给它传入两个参数：一个状态值和一个处理这个状态的生成器。创建和返回要传递给 `ASQ#runner(..)` 的适配器生成器这样的苦活将由 `state(..)` 负责。

考虑：

```
function state(val,handler) {
    // 为这个状态构造一个协程处理函数
    return function*(token) {
        // 状态转移处理函数
        function transition(to) {
            token.messages[0] = to;
        }

        // 设定初始状态(如果还未设定的话)
        if (token.messages.length < 1) {
            token.messages[0] = val;
        }

        // 继续,直到到达最终状态(false)
        while (token.messages[0] !== false) {
            // 当前状态与这个处理函数匹配吗?
            if (token.messages[0] === val) {
                // 委托给状态处理函数
                yield *handler( transition );
            }

            // 还是把控制转移到另一个状态处理函数?
            if (token.messages[0] !== false) {
                yield token;
            }
        }
    };
}
```

如果仔细观察的话，可以看到 `state(..)` 返回了一个接受一个 `token` 的生成器，然后它建立了一个 `while` 循环，该循环将持续运行，直到状态机到达终止状态（这里我们随机设定为值 `false`）。这正是我们想要传给 `ASQ#runner(..)` 的那一类生成器！

我们还随意保留了 `token.messages[0]` 槽位作为放置状态机当前状态的位置，用于追踪，

这意味着我们甚至可以把初始状态值作为种子从序列中的前一个步骤传入。

如何将辅助函数 state(..) 与 ASQ#runner(..) 配合使用呢？

```
var prevState;

ASQ(
    /*可选:初始状态值 */
    2
)
// 运行状态机
// 转移: 2 -> 3 -> 1 -> 3 -> false
.runner(
    // 状态1处理函数
    state( 1, function *stateOne(transition){
        console.log( "in state 1" );

        prevState = 1;
        yield transition( 3 );  // 转移到状态3
    } ),

    // 状态2处理函数
    state( 2, function *stateTwo(transition){
        console.log( "in state 2" );

        prevState = 2;
        yield transition( 3 );  // 转移到状态3
    } ),

    // 状态3处理函数
    state( 3, function *stateThree(transition){
        console.log( "in state 3" );

        if (prevState === 2) {
            prevState = 3;
            yield transition( 1 ); // 转移到状态1
        }
        // 完毕!
        else {
            yield "That's all folks!";

            prevState = 3;
            yield transition( false ); // 最终状态
        }
    } )
)
// 状态机完毕,继续
.val( function(msg){
    console.log( msg ); // 就这些!
} );
```

有很重要的一点需要指出，生成器 *stateOne(..)、*stateTwo(..) 和 *stateThree(..) 三者本身在每次进入状态时都会被再次调用，而在你通过 transition(..) 转移到其他值时就

会结束。尽管这里没有展示，但这些状态生成器处理函数显然可以通过 yield Promise/ 序列 /thunk 来异步暂停。

底层隐藏的由辅助函数 state(..) 产生并实际上传给 ASQ#runner(..) 的生成器是在整个状态机生存期都持续并发运行的，它们中的每一个都会把协作式 yield 控制传递到下一个，以此类推。

查看这个 ping pong 的例子（http://jsbin.com/qutabu/1/edit?js,output），可以得到更多关于使用由 ASQ#runner(..) 驱动的生成器进行协作式并发的示例。

B.4 通信顺序进程

1978 年，C. A. R. Hoare 在一篇学术论文中（http://dl.acm.org/citation.cfm?doid=359576.359585）首次描述了通信顺序进程（Communicating Sequential Processes，CSP），然后又在 1985 年的同名著作中讨论了这个概念（http://www.usingcsp.com/）。CSP 描述了一种并发“进程”在运行过程中彼此交互（通信）的正式方法。

你可能已经想起了我们在第 1 章中讨论的并发“进程”，此处对 CSP 的探索将建立在对其的理解之上。

和计算机科学领域多数伟大的概念一样，CSP 也带有浓烈的学术形式体系色彩，以进程代数的形式表达。但我觉得符号代数原理对你来说不会有什么实际的意义，所以我们将要寻找其他方法来思考 CSP。

我把多数 CSP 的正式描述和证明交给霍尔的研究以及自他以后许多其他有趣的文章。而我将要做的就是以尽可能非学术化的易于直觉理解的方式简要介绍 CSP 的思路。

B.4.1 消息传递

CSP 的核心原则是独立的进程之间所有的通信和交互必须要通过正式的消息传递。可能与你预期的相反，CSP 消息传递是用同步动作来描述的，其中发送进程和接收进程都需要准备好消息才能传递。

这样的同步消息机制怎么可能与 JavaScript 的异步编程联系在一起呢？

关系的具体化来自于使用 ES6 生成器创建看似同步但底层可能是同步或（更可能的）异步动作的方法的特性。