如果一个序列中发生了错误，并且此时没有注册错误处理函数，那这个错误就会被报告到控制台。换句话说，未处理的拒绝在默认情况下总是会被报告，而不会被吞掉和错过。

一旦你针对某个序列注册了错误处理函数，这个序列就不会产生这样的报告，从而避免了重复的噪音。

实际上，可能在一些情况下你会想创建一个序列，这个序列可能会在你能够注册处理函数之前就进入了出错状态。这不常见，但偶尔也会发生。

在这样的情况下，你可以选择通过对这个序列调用 `defer()` 来避免这个序列实例的错误报告。应该只有在确保你最终会处理这种错误的情况下才选择关闭错误报告：

```
var sq1 = ASQ( function(done){
    doesnt.Exist();         // 将会向终端抛出异常
} );

var sq2 = ASQ( function(done){
    doesnt.Exist();         // 只抛出一个序列错误
} )
// 显式避免错误报告
.defer();

setTimeout( function(){
    sq1.or( function(err){
        console.log( err ); // ReferenceError
    } );

    sq2.or( function(err){
        console.log( err ); // ReferenceError
    } );
}, 100 );

// ReferenceError (from sq1)
```

这种错误处理方式要好于 Promise 本身的那种行为，因为它是成功的坑，而不是失败陷阱（参见第 3 章）。

如果向一个序列插入（包括了）另外一个序列，参见 A.2.5 节中的完整描述，那么源序列就会关闭错误报告，但是必须要考虑现在目标序列的错误报告开关的问题。

A.2.3 并行步骤

并非序列中的所有步骤都恰好执行一个（异步）任务。序列中的一个步骤中如果有多个子步骤并行执行则称为 `gate(..)`（还有一个别名 `all(..)`，如果你愿意用的话），和原生的 `Promise.all([..])` 直接对应。

如果 gate(..) 中所有的步骤都成功完成，那么所有的成功消息都会传给下一个序列步骤。如果它们中有任何一个出错的话，整个序列就会立即进入出错状态。

考虑：

```
ASQ( function(done){
    setTimeout( done, 100 );
} )
.gate(
    function(done){
        setTimeout( function(){
            done( "Hello" );
        }, 100 );
    },
    function(done){
        setTimeout( function(){
            done( "World", "!" );
        }, 100 );
    }
)
.val( function(msg1,msg2){
    console.log( msg1 );    // Hello
    console.log( msg2 );    // [ "World", "!" ]
} );
```

出于展示说明的目的，我们把这个例子与原生 Promise 对比：

```
new Promise( function(resolve,reject){
    setTimeout( resolve, 100 );
} )
.then( function(){
    return Promise.all( [
        new Promise( function(resolve,reject){
            setTimeout( function(){
                resolve( "Hello" );
            }, 100 );
        } ),
        new Promise( function(resolve,reject){
            setTimeout( function(){
                // 注:这里需要一个[ ]数组
                resolve( [ "World", "!" ] );
            }, 100 );
        } )
    ] );
} )
.then( function(msgs){
    console.log( msgs[0] ); // Hello
    console.log( msgs[1] ); // [ "World", "!" ]
} );
```

Promise 用来表达同样的异步流程控制的重复样板代码的开销要多得多。这是一个很好的展示，说明了为什么 asynquence 的 API 和抽象让 Promise 步骤的处理轻松了很多。异步流

程越复杂，改进就会越明显。

1. 步骤的变体

contrib 插件中提供了几个 asynquence 的 `gate(..)` 步骤类型的变体，非常实用。

- `any(..)` 类似于 `gate(..)`，除了只需要一个子步骤最终成功就可以使得整个序列前进。
- `first(..)` 类似于 `any(..)`，除了只要有任何步骤成功，主序列就会前进（忽略来自其他步骤的后续结果）。
- `race(..)`（对应 `Promise.race([..])`）类似于 `first(..)`，除了只要任何步骤完成（成功或失败），主序列就会前进。
- `last(..)` 类似于 `any(..)`，除了只有最后一个成功完成的步骤会将其消息发送给主序列。
- `none(..)` 是 `gate(..)` 相反：只有所有的子步骤失败（所有的步骤出错消息被当作成功消息发送，反过来也是如此），主序列才前进。

让我们先定义一些辅助函数，以便更清楚地进行说明：

```
function success1(done) {
    setTimeout( function(){
        done( 1 );
    }, 100 );
}

function success2(done) {
    setTimeout( function(){
        done( 2 );
    }, 100 );
}

function failure3(done) {
    setTimeout( function(){
        done.fail( 3 );
    }, 100 );
}

function output(msg) {
    console.log( msg );
}
```

现在来说明这些 `gate(..)` 步骤变体的用法：

```
ASQ().race(
    failure3,
    success1
)
.or( output );      // 3

ASQ().any(
    success1,
    failure3,
```

```
        success2
    )
    .val( function(){
        var args = [].slice.call( arguments );
        console.log(
            args        // [ 1, undefined, 2 ]
        );
    } );

    ASQ().first(
        failure3,
        success1,
        success2
    )
    .val( output );     // 1


    ASQ().last(
        failure3,
        success1,
        success2
    )
    .val( output );     // 2

    ASQ().none(
        failure3
    )
    .val( output )      // 3
    .none(
        failure3
        success1
    )
    .or( output );      // 1
```

另外一个步骤变体是 `map(..)`，它使你能够异步地把一个数组的元素映射到不同的值，然后直到所有映射过程都完成，这个步骤才能继续。`map(..)` 与 `gate(..)` 非常相似，除了它是从一个数组而不是从独立的特定函数中取得初始值，而且这也是因为你定义了一个回调函数来处理每个值：

```
    function double(x,done) {
        setTimeout( function(){
            done( x * 2 );
        }, 100 );
    }

    ASQ().map( [1,2,3], double )
    .val( output );                 // [2,4,6]
```

`map(..)` 的参数（数组或回调）都可以从前一个步骤传入的消息中接收：

```
    function plusOne(x,done) {
        setTimeout( function(){
```

```
        done( x + 1 );
    }, 100 );
}

ASQ( [1,2,3] )
.map( double )          // 消息[1,2,3]传入
.map( plusOne )         // 消息[2,4,6]传入
.val( output );         // [3,5,7]
```

另外一个变体是 `waterfall(..)`，这有点类似于 `gate(..)` 的消息收集特性和 `then(..)` 的顺序处理特性的混合。

首先执行步骤 1，然后步骤 1 的成功消息发送给步骤 2，然后两个成功消息发送给步骤 3，然后三个成功消息都到达步骤 4，以此类推。这样，在某种程度上，这些消息集结和层叠下来就构成了“瀑布”（waterfall）。

考虑：

```
function double(done) {
    var args = [].slice.call( arguments, 1 );
    console.log( args );

    setTimeout( function(){
        done( args[args.length - 1] * 2 );
    }, 100 );
}

ASQ( 3 )
.waterfall(
    double,                 // [ 3 ]
    double,                 // [ 6 ]
    double,                 // [ 6, 12 ]
    double                  // [ 6, 12, 24 ]
)
.val( function(){
    var args = [].slice.call( arguments );
    console.log( args );    // [ 6, 12, 24, 48 ]
} );
```

如果“瀑布”中的任何一点出错，整个序列就会立即进入出错状态。

2. 容错

有时候可能需要在步骤级别上管理错误，不让它们把整个序列带入出错状态。为了这个目的，asynquence 提供了两个步骤变体。

`try(..)` 会试验执行一个步骤，如果成功的话，这个序列就和通常一样继续。如果这个步骤失败的话，失败就会被转化为一个成功消息，格式化为 `{ catch: .. }` 的形式，用出错消息填充：

```
ASQ()
.try( success1 )
.val( output )            // 1
.try( failure3 )
.val( output )            // { catch: 3 }
.or( function(err){
    // 永远不会到达这里
} );
```

也可以使用 until(..) 建立一个重试循环，它会试着执行这个步骤，如果失败的话就会在下一个事件循环 tick 重试这个步骤，以此类推。

这个重试循环可以无限继续，但如果想要从循环中退出的话，可以在完成触发函数中调用标志 break()，触发函数会使主序列进入出错状态：

```
var count = 0;

ASQ( 3 )
.until( double )
.val( output )                  // 6
.until( function(done){
    count++;

    setTimeout( function(){
        if (count < 5) {
            done.fail();
        }
        else {
            // 跳出until(..)重试循环
            done.break( "Oops" );
        }
    }, 100 );
} )
.or( output );                  // Oops
```

3. Promise 风格的步骤

如果你喜欢在序列使用类似于 Promise 的 then(..) 和 catch(..)（参见第 3 章）的 Promise 风格语义，可以使用 pThen 和 pCatch 插件：

```
ASQ( 21 )
.pThen( function(msg){
    return msg * 2;
} )
.pThen( output )                  // 42
.pThen( function(){
    // 抛出异常
    doesnt.Exist();
} )
.pCatch( function(err){
    // 捕获异常(拒绝)
    console.log( err );           // ReferenceError
```

```
} )
.val( function(){
    // 主序列以成功状态返回，
    // 因为之前的异常被 pCatch(..)捕获了
} );
```

pThen(..) 和 pCatch(..) 是设计用来运行在序列中的，但其行为方式就像是在一个普通的 Promise 链中。因此，可以从传给 pThen(..) 的完成处理函数决议真正的 Promise 或 asynquence 序列（参见第 3 章）。

A.2.4 序列分叉

关于 Promise，有一个可能会非常有用的特性，那就是可以附加多个 then(..) 处理函数注册到同一个 promise；在这个 promise 处有效地实现了分叉流程控制：

```
var p = Promise.resolve( 21 );

// 分叉1(来自p)
p.then( function(msg){
    return msg * 2;
} )
.then( function(msg){
    console.log( msg );     // 42
} )

// 分叉2（来自p）
p.then( function(msg){
    console.log( msg );     // 21
} );
```

在 asynquence 里可使用 fork() 实现同样的分叉：

```
var sq = ASQ(..).then(..).then(..);

var sq2 = sq.fork();

// 分叉1
sq.then(..)..;

// 分叉2
sq2.then(..)..;
```

A.2.5 合并序列

如果要实现 fork() 的逆操作，可以使用实例方法 seq(..)，通过把一个序列归入另一个序列来合并这两个序列：

```
var sq = ASQ( function(done){
    setTimeout( function(){
```

```
        done( "Hello World" );
    }, 200 );
} );

ASQ( function(done){
    setTimeout( done, 100 );
} )
// 将sq序列纳入这个序列
.seq( sq )
.val( function(msg){
    console.log( msg );     // Hello World
} )
```

正如这里展示的，`seq(..)` 可以接受一个序列本身，或者一个函数。如果它接收一个函数，那么就要求这个函数被调用时会返回一个序列。因此，前面的代码可以这样实现：

```
// ..
.seq( function(){
    return sq;
} )
// ..
```

这个步骤也可以通过 `pipe(..)` 来完成：

```
// ..
.then( function(done){
    // 把sq加入done continuation回调
    sq.pipe( done );
} )
// ..
```

如果一个序列被包含，那么它的成功消息流和出错流都会输入进来。

正如前面的注解所提到的，管道化（使用 `pipe(..)` 手工实现的或通过 `seq(..)` 自动进行的）会关闭源序列的错误报告，但不会影响目标序列的错误报告。

A.3 值与错误序列

如果序列的某个步骤只是一个普通的值，这个值就映射为这个步骤的完成消息：

```
var sq = ASQ( 42 );

sq.val( function(msg){
    console.log( msg );      // 42
} );
```

如果你想要构建一个自动出错的序列：

```
var sq = ASQ.failed( "Oops" );

ASQ()
.seq( sq )
.val( function(msg){
    // 不会到达这里
} )
.or( function(err){
    console.log( err );     // Oops
} );
```

也有可能你想自动创建一个延时值或者延时出错的序列。使用 contrib 插件 `after` 和 `failAfter`，很容易实现：

```
var sq1 = ASQ.after( 100, "Hello", "World" );
var sq2 = ASQ.failAfter( 100, "Oops" );

sq1.val( function(msg1,msg2){
    console.log( msg1, msg2 );       // Hello World
} );

sq2.or( function(err){
    console.log( err );                  // Oops
} );
```

也可以使用 `after(..)` 在序列中插入一个延时：

```
ASQ( 42 )
// 在序列中插入一个延时
.after( 100 )
.val( function(msg){
    console.log( msg );     // 42
} );
```

A.4 Promise 与回调

我认为 asynquence 序列在原生 Promise 之上提供了很多新的价值。多数情况下，你都会发现在这一抽象层次上工作是非常令人愉快和强大的。不过，把 asynquence 与其他非 asynquence 代码集成也是可以实现的。

通过实例方法 `promise(..)` 很容易把一个 promise（比如一个 thenable，参见第 3 章）归入到一个序列中：

```
var p = Promise.resolve( 42 );

ASQ()
.promise( p )           // 也可以：function(){ return p; }
.val( function(msg){
    console.log( msg ); // 42
} );
```

要实现相反的操作以及从一个序列中的某个步骤分叉 / 剔出一个 promise，可以通过 contrib 插件 toPromise 实现：

```
var sq = ASQ.after( 100, "Hello World" );

sq.toPromise()
// 现在这是一个标准promise链
.then( function(msg){
    return msg.toUpperCase();
} )
.then( function(msg){
    console.log( msg );     // HELLO WORLD
} );
```

有几个辅助工具可以让 asynquence 与使用回调的系统适配。要从序列中自动生成一个 error-first 风格回调以连入到面向回调的工具，可以使用 errfcb：

```
var sq = ASQ( function(done){
    // 注:期望"error-first风格"回调
    someAsyncFuncWithCB( 1, 2, done.errfcb )
} )
.val( function(msg){
    // ..
} )
.or( function(err){
    // ..
} );

// 注:期望"error-first风格"回调
anotherAsyncFuncWithCB( 1, 2, sq.errfcb() );
```

你还可能想要为某个工具创建一个序列封装的版本，类似于第 3 章的 promisory 和第 4 章的 thunkory，asynquence 为此提供了 ASQ.wrap(..)：

```
var coolUtility = ASQ.wrap( someAsyncFuncWithCB );

coolUtility( 1, 2 )
.val( function(msg){
    // ..
} )
.or( function(err){
    // ..
} );
```

为了清晰起见（也是为了好玩！），我们再来发明一个术语，用于表达来自 ASQ.wrap(..) 的生成序列的函数，就像这里的 coolUtility。我建议使用 sequory（sequence+factory）。

A.5 可迭代序列

序列的一般范式是每个步骤负责完成它自己，这也是使序列前进的原因。Promise 的工作方式也是相同的。

不幸的是，有时候需要实现对 Promise 或步骤的外部控制，这会导致棘手的 capability extraction 问题。

考虑这个 Promise 例子：

```
var domready = new Promise( function(resolve,reject){
    // 不需把这个放在这里,因为逻辑上这属于另一部分代码
    document.addEventListener( "DOMContentLoaded", resolve );
} );

// ..

domready.then( function(){
    // DOM就绪!
} );
```

使用 Promise 的 capability extraction 反模式看起来类似如下：

```
var ready;

var domready = new Promise( function(resolve,reject){
    // 提取resolve()功能
    ready = resolve;
} );

// ..

domready.then( function(){
    // DOM就绪!
} );

// ..

document.addEventListener( "DOMContentLoaded", ready );
```

依我看来，这个反模式有奇怪的代码味，但很多开发者喜欢这样做，个中缘由，我不太明了。

asynquence 提供了一个反转的序列类型，我称之为*可迭代序列*，它把控制能力外部化了（对于像 domready 这样的用例非常有用）：

```
// 注：这里的domready是一个控制这个序列的迭代器
var domready = ASQ.iterable();

// ..

domready.val( function(){
    // DOM就绪
} );

// ..

document.addEventListener( "DOMContentLoaded", domready.next );
```

可迭代序列的使用场景不只是这里看到的这个，我们将在附录 B 中再次介绍。

A.6 运行生成器

在第 4 章中我们推导出了一个名为 `run(..)` 的工具。这个工具可以运行生成器到结束，侦听 `yield` 出来的 Promise，并使用它们来异步恢复生成器。asynquence 也内建有这样的工具，叫作 `runner(..)`。

为了展示，我们首先构建一些辅助函数：

```
function doublePr(x) {
    return new Promise( function(resolve,reject){
        setTimeout( function(){
            resolve( x * 2 );
        }, 100 );
    } );
}

function doubleSeq(x) {
    return ASQ( function(done){
        setTimeout( function(){
            done( x * 2)
        }, 100 );
    } );
}
```

现在，可以使用 `runner(..)` 作为序列中的一个步骤：

```
ASQ( 10, 11 )
.runner( function*(token){
    var x = token.messages[0] + token.messages[1];

    // yield一个真正的promise
    x = yield doublePr( x );

    // yield一个序列
    x = yield doubleSeq( x );
```

```
    return x;
} )
.val( function(msg){
    console.log( msg );         // 84
} );
```

封装的生成器

你也可以创建一个自封装的生成器，也就是说，通过 `ASQ.wrap(..)` 包装实现一个运行指定生成器的普通函数，完成后返回一个序列：

```
var foo = ASQ.wrap( function*(token){
    var x = token.messages[0] + token.messages[1];

    // yield一个真正的promise
    x = yield doublePr( x );

    // yield一个序列
    x = yield doubleSeq( x );

    return x;
}, { gen: true } );

// ..

foo( 8, 9 )
.val( function(msg){
    console.log( msg );         // 68
} );
```

`runner(..)` 还可以实现更多很强大的功能，我们将在附录 B 中深入介绍。

A.7 小结

asynquence 是一个建立在 Promise 之上的简单抽象，一个序列就是一系列（异步）步骤，目标在于简化各种异步模式的使用，而不失其功能。

除了我们在本附录中介绍的，asynquence 核心 API 及其 contrib 插件中还有很多好东西，但我们将把对其余功能的探索作为一个练习留给你。

现在，你已经看到了 asynquence 的本质与灵魂。关键点是一个序列由步骤组成，这些步骤可以是 Promise 的数十种变体的任何一种，也可以是通过生成器运行的，或者是其他什么……决策权在于你，你可以自由选择适合任务的异步流程控制逻辑编织起来。使用不同的异步模式不再需要切换不同的库。

如果你已经理解了这些代码片段的意义，现在就可以快速学习这个库了。实际上，学习它并不需要耗费多少精力！

如果对它的工作方式（或原理）还有点迷糊的话，你可能需要再花点时间查看一下前面的例子，熟悉一下 asynquence 的使用，然后再进行附录 B 的学习。在附录 B 中，我们将使用 asynquence 实现几种更高级更强大的异步模式。

附录 B

高级异步模式

附录 A 介绍了面向序列的异步流程控制库 asynquence，它主要基于 Promise 和生成器。

现在，我们将要探索构建在已有理解和功能之上的其他高级异步模式，并了解 asynquence 如何让这些高级异步技术在我们的程序中更易于混用和匹配而无需分立的多个库。

B.1 可迭代序列

附录 A 介绍过 asynquence 的可迭代序列，这里我们打算更深入地再次探讨一下相关内容。

回忆一下：

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

// ..

domready.val( function(){
    // DOM就绪
} );

// ..

document.addEventListener( "DOMContentLoaded", domready.next );
```

现在，让我们把一个多步骤序列定义为可迭代序列：

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