```
it.next();          // { value: 1, done: false }
it.return( 42 );    // cleanup!
                    // { value: 42, done: true }
```

不要把 yield 语句放在 finally 子句内部！虽然这是有效且合法的，但确实是一个可怕的思路。它会延后你的 return(..) 调用的完成，因为任何在 finally 子句内部的 yield.. 表达式都会被当作是暂停并发送消息；你不会像期望的那样立即得到完成的生成器。基本上不会有合理的原因要实现这么可怕的思路，所以不要这么用！

除了前面代码片段中展示的如何通过 return(..) 终止生成器，同时触发 finally 子句，它还展示了生成器在每次被调用的时候都产生了一个全新的迭代器。实际上，可以同时把多个迭代器附着在同一个生成器上：

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

var it1 = foo();
it1.next();             // { value: 1, done: false }
it1.next();             // { value: 2, done: false }

var it2 = foo();
it2.next();             // { value: 1, done: false }

it1.next();             // { value: 3, done: false }

it2.next();             // { value: 2, done: false }
it2.next();             // { value: 3, done: false }

it2.next();             // { value: undefined, done: true }
it1.next();             // { value: undefined, done: true }
```

提前终止

除了调用 return(..)，还可以调用 throw(..)。正如 return(x) 基本上就是在生成器中的当前暂停点插入了一个 return x，调用 throw(x) 基本上就相当于在暂停点插入一个 throw x。

除了对异常处理的不同（我们将在下一小节介绍对于 try 语句这意味着什么），throw(..) 同样引起提前完成，在当前暂停点终止生成器的运行。举例来说：

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

```
var it = foo();

it.next();                  // { value: 1, done: false }

try {
    it.throw( "Oops!" );
}
catch (err) {
    console.log( err );     // Exception: Oops!
}

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

因为 `throw(..)` 基本上就是在生成器 `yield 1` 这一行插入一个 `throw ..`，没有处理这个异常，所以它会立即传递回调用代码，其中通过 `try..catch` 处理了这个异常。

和 `return(..)` 不同，迭代器的 `throw(..)` 方法从来不会被自动调用。

当然，尽管没有在前面代码中展示，如果在调用 `throw(..)` 的时候有 `try..finally` 子句在生成器内部等待，那么在异常传回调用代码之前 `finally` 子句会有机会运行。

3.2.4 错误处理

前面我们已经暗示，生成器的错误处理可以表达为 `try..catch`，它可以在由内向外和由外向内两个方向工作：

```
function *foo() {
    try {
        yield 1;
    }
    catch (err) {
        console.log( err );
    }

    yield 2;

    throw "Hello!";
}

var it = foo();

it.next();                  // { value: 1, done: false }

try {
    it.throw( "Hi!" );      // Hi!
                            // { value: 2, done: false }
    it.next();

    console.log( "never gets here" );
}
```

```
    catch (err) {
        console.log( err ); // Hello!
    }
```

错误也可以通过 `yield *` 委托在两个方向上传播：

```
    function *foo() {
        try {
            yield 1;
        }
        catch (err) {
            console.log( err );
        }

        yield 2;

        throw "foo: e2";
    }

    function *bar() {
        try {
            yield *foo();

            console.log( "never gets here" );
        }
        catch (err) {
            console.log( err );
        }
    }

    var it = bar();

    try {
        it.next();              // { value: 1, done: false }
        it.throw( "e1" );       // e1
                                // { value: 2, done: false }

        it.next();              // foo: e2
                                // { value: undefined, done: true }
    }
    catch (err) {
        console.log( "never gets here" );
    }

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

就像前面我们所看到的，`*foo()` 调用 `yield 1` 的时候，值 1 通过 `*bar()` 传递没有改变。

但这段代码最有趣的是，当 `*foo()` 调用 `throw "foo: e2"` 的时候，这个错误传播到了 `*bar()` 并立即被 `*bar()` 的 `try..catch` 代码块捕获。这个错误不会像值 1 一样穿过 `*bar()`。

接着 `*bar()` 的 `catch` 执行一个普通的输出 `err("foo: e2")` 然后 `*bar()` 正常执行完毕，这也是为什么从 `it.next()` 返回迭代器结果 `{ value: undefined, done: true }`。

如果 *bar() 在 yield *.. 表达式外并没有包裹一个 try..catch，那么这个错误当然就会一路传播出来，在路上还是会完成（终止）*bar() 的执行。

3.2.5 Transpile 生成器

可以用 ES6 之前的代码表达生成器功能吗？答案是可以，并且已经有好几个很棒的工具实现了这一点，包括最著名的 Facebook 的 Regenerator 工具（https://facebook.github.io/regenerator/）。

但为了更好地理解生成器，我们还是试着来手动转换一下。总的说来，我们将要创造一个简单的基于闭包的状态机。

我们的源生成器特别简单：

```
function *foo() {
    var x = yield 42;
    console.log( x );
}
```

一开始，我们需要一个名为 foo() 的函数来执行，它需要返回一个迭代器：

```
function foo() {
    // ..

    return {
        next: function(v) {
            // ..
        }

        // 省略return(..)和throw(..)
    };
}
```

现在，需要一些内部变量来记录在“生成器”逻辑步骤内部的当前位置。我们称之为 state。将会有 3 个状态：0 初始态、1 等待 yield 表达式完成、2 生成器完毕。

每次调用 next(..)，我们需要处理下一个步骤，然后增加 state。为了方便起见，我们把每个步骤放在一个 switch 语句的 case 子句中，并把这些放在内部函数 nextState() 中，next() 可以调用这个函数。另外，因为 x 是一个跨这个“生成器”整个作用域的变量，所以它需要存活在 nextState(..) 函数之外。

下面是所有代码（显然是某种程度上的简化，以便保持概念展示的清晰）：

```
function foo() {
    function nextState(v) {
        switch (state) {
            case 0:
```

```
                state++;

                // yield表达式
                return 42;
            case 1:
                state++;

                // yield表达式完成
                x = v;
                console.log( x );

                // 隐式return
                return undefined;

            // 不需要处理状态2
        }
    }

    var state = 0, x;

    return {
        next: function(v) {
            var ret = nextState( v );

            return { value: ret, done: (state == 2) };
        }

        // 省略return(..)和throw(..)
    };
}
```

最后，让我们来测试一下我们的前 ES6“生成器”：

```
var it = foo();

it.next();              // { value: 42, done: false }

it.next( 10 );          // 10
                        // { value: undefined, done: true }
```

还不错吧？希望这个练习帮你巩固了生成器实际上就是状态机逻辑的简化语法这个概念。这使得它们应用广泛。

3.2.6 生成器使用

现在，我们已经更深入理解了生成器的工作原理，那么它们适用于哪些场景呢？

我们已经看到了两个主要模式。

- *产生一系列值*
  这个用法可以很简单（比如随机字符串或者递增数），也可以表示更结构化的数据访问

（比如在数据库查询返回的行上迭代）。

不管怎样，我们使用迭代器来控制生成器，所以可以在每次调用 `next(..)` 的时候触发某些逻辑。数据结构上的普通迭代器只是取出值而没有控制逻辑。

- *顺序执行的任务队列*

  这种用法通常表示算法中步骤的流控制，其中每个步骤要求从某个外部源获得数据。每部分数据的完成可以是即时的，也可以是异步延迟的。

  从生成器内部代码的角度来看，在 `yield` 点同步或异步这样的细节是完全透明的。而且，这些细节是故意被抽象出去的，这样就不会被诸如实现复杂性模糊了步骤的自然顺序表达。抽象也意味着实现可以在无需修改生成器内部代码的情况下被替换 / 重构。

通过这些应用场景来观察生成器时，它们就远不止是手动状态机的不同或者说更优雅的语法形式了。它们是用于控制组织数据有序产生和消耗的强有力工具。

3.3 模块

在所有 JavaScript 代码中，唯一最重要的代码组织模式是模块，而且一直都是，我并不认为这是夸大其词。对于我本人，我认为也对于广泛社区来说，模块模式驱动了大多数代码。

3.3.1 旧方法

传统的模块模式基于一个带有内部变量和函数的外层函数，以及一个被返回的“public API”，这个“public API”带有对内部数据和功能拥有闭包的方法。通常这样表达：

```
function Hello(name) {
    function greeting() {
        console.log( "Hello " + name + "!" );
    }

    // public API
    return {
        greeting: greeting
    };
}

var me = Hello( "Kyle" );
me.greeting();          // Hello Kyle!
```

继续调用 `Hello(..)` 模块可以产生多个实例。有时一个模块只作为单例（singleton，也就是说只需要一个实例），这种情况下前面的代码需要稍作修改，通常这样使用一个 IIFE：

```
var me = (function Hello(name){
    function greeting() {
        console.log( "Hello " + name + "!" );
```

```
        }

        // public API
        return {
            greeting: greeting
        };
    })( "Kyle" );

    me.greeting();          // Hello Kyle!
```

这个模式是经过试验的。它也足够灵活，针对不同场景有多个变体。

其中常用的是异步模块定义（Asynchronous Module Definition，AMD），还有一种是通用模块定义（Universal Module Definition，UMD）。这里我们不会具体介绍这些模式和技术，但网上有很多详尽的解释。

3.3.2 前进

对于 ES6 来说，我们不再需要依赖于封装函数和闭包提供模块支持。ES6 中模块已经具备一等（first class）语法和功能支持。

在讨论具体语法细节之前，有一点很重要，就是要理解 ES6 模块和过去我们处理模块的方式之间的显著概念区别。

- ES6 使用基于文件的模块，也就是说一个文件一个模块。目前，还没有把多个模块合并到单个文件中的标准方法。
  这意味着如果想要把 ES6 模块直接加载到浏览器 Web 应用中，需要分别加载，而不是作为一大组放在单个文件中加载。在过去，为了性能优化，后者这种加载方式是很常见的。期待 HTTP/2 的到来能够显著消除所有这样的性能担忧，因为它运行在持久 socket 连接上，所以能够高效并发、交替加载多个小文件。
- ES6 模块的 API 是静态的。也就是说，需要在模块的公开 API 中静态定义所有最高层导出，之后无法补充。
  某些应用已经习惯了提供动态 API 定义的能力，可以根据对运行时情况的响应增加 / 删除 / 替换方法。这些用法或者改变自身以适应 ES6 静态 API，或者需要限制对二级对象属性 / 方法的动态修改。
- ES6 模块是单例。也就是说，模块只有一个实例，其中维护了它的状态。每次向其他模块导入这个模块的时候，得到的是对单个中心实例的引用。如果需要产生多个模块实例，那么你的模块需要提供某种工厂方法来实现这一点。
- 模块的公开 API 中暴露的属性和方法并不仅仅是普通的值或引用的赋值。它们是到内部模块定义中的标识符的实际绑定（几乎类似于指针）。
  在前 ES6 的模块中，如果把一个持有像数字或者字符串这样的原生值的属性放在公开 API 中，这个属性赋值是通过值复制赋值，任何对于对应变量的内部更新将会是独立

的，不会影响 API 对象的公开复制。

对于 ES6 来说，导出一个局部私有变量，即使当前它持有一个原生字符串 / 数字等，导出的都是到这个变量的绑定。如果模块修改了这个变量的值，外部导入绑定现在会决议到新的值。

- 导入模块和静态请求加载（如果还没加载的话）这个模块是一样的。如果是在浏览器环境中，这意味着通过网络阻塞加载；如果是在服务器上（比如 Node.js），则是从文件系统的阻塞加载。

  但是，不要惊慌于这里的性能暗示。因为 ES6 模块具有静态定义，导入需求可以静态扫描预先加载，甚至是在使用这个模块之前。

  关于如何处理这些加载请求，ES6 并没有实际指定或处理具体机制。这里有一个独立的模块加载器（Module Loader）的概念，其中每个宿主环境（浏览器、Node.js 等）提供一个适合环境的默认加载器。导入模块时使用一个字符串值表示去哪里获得这个模块（URL、文件路径等），但是这个值对于你的程序来说是透明的，只对加载器本身有意义。

  如果需要提供比默认加载器更细粒度的控制能力可以自定义加载器，默认加载器基本上没有粒度控制，因为它对于你的程序代码完全是不可见的。

你可以看到，ES6 模块将会为代码组织提供完整支持，包括封装、控制公开 API 以及引用依赖导入。但是实现方式非常特殊，可能可以也可能无法完全适应之前多年以来实现模块的方式。

CommonJS

还有一个类似、但并不完全兼容的模块语法 CommonJS，Node.js 生态系统下的开发者对此会很熟悉。

不得不说，长远看来，ES6 模块从本质上说必然会取代之前所有的模块格式和标准，即使是 CommonJS，因为 ES6 模块是建立在语言的语法支持基础上的。最终它总会不可逆转地作为统治方法成为最后的赢家。

但离那时还有很长的路要走。在服务器端 JavaScript 的世界里，已经有了成百上千的 CommonJS 风格模块，以及浏览器端十倍于此的各种格式标准的模块（UMD、AMD、临时性的模块方案）。要迁移这些模块需要很多年才能初见成效。

在此期间，模块 transpiler / 转换工具是必不可少的。你可能刚刚开始习惯这个新现实。不管你是在编写普通模块、AMD、UMD、CommonJS 还是 ES6，这些工具都不得不解析转化为对代码运行的所有环境都适用的形式。

对于 Node.js 来说，这可能意味着（目前的）目标是 CommonJS。对于浏览器来说，可能是 UMD 或者 AMD。接下来的几年里，随着这些工具的成熟和最佳实践的涌现，这一方面可能会变幻莫测。

由此，关于模块我的最好建议是：不管之前你虔诚地支持和熟悉哪种格式，现在开始开发和理解 ES6 模块吧，因为它终将会使得其他模块方法消失。它是 JavaScript 模块的未来，虽然现在有点落后。

3.3.3 新方法

支撑 ES6 模块的两个主要新关键字是 `import` 和 `export`。它们在语法上有许多微妙之处，所以我们来深入了解一下。

这里有一个很容易被忽略的重要细节：`import` 和 `export` 都必须出现在使用它们的最顶层作用域。举例来说，不能把 `import` 或 `export` 放在 `if` 条件中；它们必须出现在所有代码块和函数的外面。

1. 导出 API 成员

`export` 关键字或者是放在声明的前面，或者是作为一个操作符（或类似的）与一个要导出的绑定列表一起使用。考虑：

```
export function foo() {
    // ..
}

export var awesome = 42;

var bar = [1,2,3];
export { bar };
```

下面是同样导出的另外一种表达形式：

```
function foo() {
    // ..
}

var awesome = 42;
var bar = [1,2,3];

export { foo, awesome, bar };
```

这些都称为**命名导出**（named export），因为导出变量 / 函数等的名称绑定。

没有用 `export` 标示的一切都在模块作用域内部保持私有。也就是说，尽管 `var bar = ..` 看起来像是声明在全局作用域的顶层，而这个顶层作用域实际上是模块本身；在模块内没有全局作用域。

模块还能访问 `window` 和所有的“全局”变量，只是不作为词法上的顶层作用域。但如果可能的话，在你的模块里应该尽量远离那些全局量。

在命名导出时还可以“重命名”（也即别名）一个模块成员：

```
function foo() { .. }

export { foo as bar };
```

导入这个模块的时候，只有成员名称 bar 可以导入；foo 还是隐藏在模块内部。

模块导出不是像你熟悉的赋值运算符 = 那样只是值或者引用的普通赋值。实际上，导出的是对这些东西（变量等）的绑定（类似于指针）。

如果在你的模块内部修改已经导出绑定的变量的值，即使是已经导入的（参见下一小节），导入的绑定也将会决议到当前（更新后）的值。

考虑：

```
var awesome = 42;
export { awesome };

// 之后
awesome = 100;
```

导入这个模块的时候，不管是在 awesome = 100 之前还是之后，一旦赋值发生，导入的绑定就会决议到 100 而不是 42。

这是因为本质上，绑定是一个指向 awesome 变量本身的引用或者指针，而不是这个值的复制。ES6 模块绑定为 JavaScript 带来的这个概念是前所未有的。

尽管显然可以在模块定义内部多次使用 export，ES6 绝对倾向于一个模块使用一个 export，称之为**默认导出**（default export）。TC39 委员会的一些成员认为，如果遵循这个模式，那么“得到的回报是更简单的 import 语法”，如果不这么做，得到的“惩罚”则是更繁复的语法。

默认导出把一个特定导出绑定设置为导入模块时的默认导出。绑定的名称就是 default。后面将会看到，导入模块绑定时可以重命名，因为通常都会使用默认导出。

每个模块定义只能有一个 default。下一小节将会介绍 import，那时你可以看到如果模块有一个默认导出，它将如何使得 import 语法更加简洁。

关于默认导出有一个微妙的细节需要格外小心。比较这两段代码：

```
function foo(..) {
    // ..
}

export default foo;
```