```
    };
})();

foo( 123456 );          // 3810376848.5
```

如果你在实现了 TCO 的 ES6 引擎中运行前面的代码，会得到如前显示的 3810376848.5。然而，它在非 TCO 引擎里仍然会因 RangeError 而失败。

7.7.2 非 TCO 优化

还有几种其他技术可以用来重写代码，使得不需要每次调用时都增长栈。

其中一种这样的技术叫作 trampolining，它相当于把每个部分结果用一个函数表示，这些函数或者返回另外一个部分结果函数，或者返回最终结果。然后就只需要循环直到得到的结果不是函数，得到的就是最终结果。

考虑：

```
"use strict";

function trampoline( res ) {
    while (typeof res == "function") {
        res = res();
    }
    return res;
}

var foo = (function(){
    function _foo(acc,x) {
        if (x <= 1) return acc;
        return function partial(){
            return _foo( (x / 2) + acc, x - 1 );
        };
    }

    return function(x) {
        return trampoline( _foo( 1, x ) );
    };
})();

foo( 123456 );          // 3810376848.5
```

这个重写需要最小的改动来把递归转化为 trampoline(..) 中的循环。

(1) 首先，把 return _foo .. 一行封装在 return partial() { .. 函数表达式中。
(2) 然后，把 _foo(1,x) 调用封装在 trampoline(..) 调用中。

这个技术不限制调用栈的原因是，每个内部的 partial(..) 函数只是返回到 trampoline(..) 的 while 循环中，trampolining 运行函数并进行下一次的循环迭代。换句话

说，partial(..) 不会递归调用自身，它只是返回另一个函数。栈深度保持不变，所以可以运行任意长的时间。

通过这种方式实现的 trampolining 使用了内层 partial() 函数在变量 x 和 acc 上的闭包，在迭代之间保持状态。其优点是把循环逻辑抽出到了可复用的 trampoline(..) 工具函数，很多库都提供了它的各种版本。可以在你的程序中用不同的 trampoline 算法多次复用 trampoline(..)。

当然，如果真的需要深度优化（不需考虑可复用性），那么可以丢弃闭包状态，用一个循环把 acc 信息的状态追踪在线化放在一个函数作用域内。这种技术一般称为**递归展开**：

```
"use strict";

function foo(x) {
    var acc = 1;
    while (x > 1) {
        acc = (x / 2) + acc;
        x = x - 1;
    }
    return acc;
}

foo( 123456 );          // 3810376848.5
```

算法的这种表达方式可读性更高，很可能也是我们前面探索的各种形式中性能（严格说来）最高的。所以这个方案显然是最好的，你可能会奇怪为什么还要用其他方法。

下面是两个使我们并不想总是手动展开递归的原因。

- 这里没有为了可复用性把 trampolining（循环）逻辑提取出来，而是将它在线化了。在只需要考虑一个例子的时候这还合适，但如果你的程序中有多个这种情况，很可能需要提高复用度来保持代码更简短、更易管理。
- 这里的例子非常简单，只是用来展示各种不同的形式。但在实践中，递归算法中还有很多更复杂的逻辑，比如互相递归（不只一个函数调用自身）。

对这个无底洞探索越深，就会发现展开优化变得越手动化和错综复杂。你很快就会丧失所有前面得到的可读性价值。递归的最主要优势，即使是 PTC 形式，就是它保留了算法的可读性，并将性能优化的担子扔给引擎。

如果以 PTC 的形式编写算法，ES6 引擎就会应用 TCO，代码就会以常数栈深度（通过重用栈帧）运行。你在得到递归的可读性的同时，也得到了几乎没有损失的性能以及不受限制的运行长度。

7.7.3 元在何处

TCO 和元编程又有什么关系呢？

正如我们在 7.6 节中介绍的，可以在运行时判断引擎支持哪些特性。其中就包括 TCO，尽管确定方法是十分暴力的。考虑：

```
"use strict";

try {
    (function foo(x){
        if (x < 5E5) return foo( x + 1 );
    })( 1 );

    TCO_ENABLED = true;
}
catch (err) {
    TCO_ENABLED = false;
}
```

在非 TCO 引擎中，递归循环最终会失败，抛出一个异常被 `try..catch` 捕获。换句话说，有了 TCO，循环才能完成。

不怎么样，对吧？

而围绕着 TCO 特性（或者，这个特性的缺失）的元编程对我们的代码有什么好处呢？简单的答案是，可以通过这种特性测试来决定是加载使用递归的应用代码版本，还是转换 / transpile 为不需要递归的版本。

自适应代码

还有另外一种看问题的方式：

```
"use strict";

function foo(x) {
    function _foo() {
        if (x > 1) {
            acc = acc + (x / 2);
            x = x - 1;
            return _foo();
        }
    }

    var acc = 1;

    while (x > 1) {
        try {
            _foo();
        }
        catch (err) { }
```

```
        }

        return acc;
    }

    foo( 123456 );          // 3810376848.5
```

这个算法尽可能多地使用了递归，但是是通过作用域内变量 x 和 acc 保持进展状态。如果整个问题都可以不出错地通过递归解决，那么很好。如果引擎在某处杀死了递归，我们就会在 try..catch 中捕获到，然后再试一次，继续我们其余的工作。

我把这种形式看作是一种元编程，理由是在运行时探索引擎的能力来（递归地）完成任务，并且为可能的（非 TCO）引擎局限提供了替代版本。

第一眼（甚至第二眼！）看上去，我敢说比起前面的几个版本，你会觉得这段代码要丑陋许多。运行起来它也要慢得多（在非 TCO 环境中大量运行的情况下）。

这个对递归栈限制的“解决方案”的主要优点，除了即使在非 TCO 引擎中也能够完成任意规模的任务之外，就是比前面展示的 trampolining 技术和手动展开技术更灵活。

本质上说，这个例子中的 _foo() 可以替换为几乎任意递归任务，即使是互相递归。其余部分是任何算法都适用的样板。

唯一的“catch”是为了能够在递归达到限制的时候恢复，递归的状态必须要用递归函数之外的作用域变量维护。我们通过把 x 和 acc 放在 _foo() 函数之外，而不是像之前一样把它们作为参数传递给 _foo() 来实现这一点。

几乎所有递归算法都可以被改造为用这种方式工作。这意味着这是在你的程序中进行最小重写就能利用 TCO 递归的最广泛的可行方法。

这种方法也使用了 PTC，意味着从旧有浏览器中使用多次循环（递归批处理）运行到 ES6+ 环境中充分利用 TCO 的递归，这段代码将会得到显著提高。我认为这很酷！

7.8 小结

元编程是指把程序的逻辑转向关注自身（或自身的运行时环境），要么是为了查看自己的结构，要么是为了修改它。元编程的主要价值是扩展语言的一般机制来提供额外的新功能。

在 ES6 之前，JavaScript 已经有了不少的元编程功能，而 ES6 提供了几个新特性，显著提高了元编程能力。

从匿名函数的函数名推导，到提供了构造器调用方式这样的信息的元属性，你可以比过

去更深入地查看程序运行时的结构。通过公开符号可以覆盖原本特性，比如对象到原生类型的类型转换。代理可以拦截并自定义对象的各种底层操作，`Reflect` 提供了工具来模拟它们。

特性测试，甚至可以测试像尾递归优化这样微妙的语义特性，把元编程的焦点从你的程序转移到 JavaScript 引擎功能本身。通过更多地了解环境能力，你的程序可以在运行时调整自己达到最优效果。

应该使用元编程吗？我的建议是：首先应将重点放在了解这个语言的核心机制到底是如何工作的。而一旦你真正了解了 JavaScript 本身的运作机制，那么就是开始使用这些强大的元编程能力进一步应用这个语言的时候了。

第 8 章

ES6 之后

写作本部分的时候，ECMA 即将对 ES6（ECMAScript 2015）最终草案的批准进行正式投票。但尽管 ES6 还正在定案，TC39 委员会已经开始进行 ES7/2016 及后续特性的紧张工作了。

在第 1 章已经讨论过，我们可以预见 JavaScript 的发展节奏将要从每隔几年更新一次进化到每年一个正式版本更新（因此基于年度命名）。这将从根本上改变 JavaScript 开发者学习和追随这门语言发展进度的方式。

但更重要的是，实际上委员会将会以特性为单位工作。一旦某个特性标准完成，并且在几个浏览器通过实现测试了思路，这个特性就被认为足够稳定可以使用了。这强烈鼓励我们一旦某个特性可用就采用这个特性，而不是等待官方标准投票。如果你还没有开始学习 ES6，那么可就错过上船的时间了！

编写本部分时，可以在这里（https://github.com/tc39/ecma262#current-proposals）看到未来的提案及其状态。

在新特性还没有被需要支持的所有浏览器都实现的情况下，transpiler 和 polyfill 是我们迁移到新特性的桥梁。Babel、Traceur 和其他几个主要 transpiler 已经支持一些极可能确定的后 ES6 特性了。

认识到这一点，就会明白现在已经是开始了解这些特性的时候了。我们来学习吧！

这些特性还处于不同的开发阶段。虽然它们很可能会确定下来，并且将类似于本章所述，但不要把这一章的内容全盘接受。在未来的版本中，这一章内容会随着这些（以及其他！）特性的最终确定而进化。

8.1 异步函数

在 4.2 节中，我们提到了一个关于直接在语法上支持这个模式的提案：生成器向类似运行器的工具 `yield` 出 promise，这个运行器工具会在 promise 完成时恢复生成器。让我们来简单了解一下这个提案提出的特性 `async function`。

回忆一下第 4 章里这个生成器的例子：

```
run( function *main() {
    var ret = yield step1();

    try {
        ret = yield step2( ret );
    }
    catch (err) {
        ret = yield step2Failed( err );
    }

    ret = yield Promise.all([
        step3a( ret ),
        step3b( ret ),
        step3c( ret )
    ]);

    yield step4( ret );
} )
.then(
    function fulfilled(){
        // *main()成功完成
    },
    function rejected(reason){
        // 哎呀，出错了
    }
);
```

提案的 `async function` 语法不需要 `run(..)` 工具就可以表达同样的流控制逻辑，因为 JavaScript 将会自动了解如何寻找要等待和恢复的 promise。

考虑：

```
async function main() {
    var ret = await step1();

    try {
        ret = await step2( ret );
    }
    catch (err) {
        ret = await step2Failed( err );
    }

    ret = await Promise.all( [
```

```
        step3a( ret ),
        step3b( ret ),
        step3c( ret )
    ] );

    await step4( ret );
}

main()
.then(
    function fulfilled(){
        // main()成功完成
    },
    function rejected(reason){
        // 哎呀，出错了
    }
);
```

我们没有使用 `function *main() {..` 声明，而是使用了 `async function main() {..` 形式。而且，没有 `yield` 出一个 promise，而是 `await` 这个 promise。调用来运行函数 `main()` 实际上返回了一个可以直接观察的 promise。这和从 `run(main)` 调用返回的 promise 是等价的。

看到这种对称性了吗？ `async function` 本质上就是生成器 + promise + `run(..)` 模式的语法糖；它们底层的运作方式是一样的！

如果你是一个 C# 开发者，那么一定很熟悉这个 `async/await` 模式，因为这个特性就是直接来自于 C# 的对应特性。很高兴看到语言特性收敛。

Babel、Traceur 和其他 transpiler 都已经对当前状态的 `async function` 提供了早期支持，所以已经可以开始使用这个特性了。但在下一小节中，我们将会介绍为什么现在有点为时尚早。

还有一个对 `async function*` 的提案，可以称之为“异步生成器”。你可以在同一段代码中既 `yield` 又 `await`，甚至可以把这两个运算放在同一个语句：`x = await yield y`。这个“异步生成器”提案似乎更不稳定——具体说，它的返回值还没有完全确定。有些人认为返回值应该是一个 observable，有点类似于一个迭代器和一个 promise 的合并。目前我们不会深入探讨这个主题，但会对它保持关注。

警告

`async function` 有一个没有解决的问题，因为它只返回一个 promise，所以没有办法从外部**取消**一个正在运行的 `async function` 实例。如果这个异步操作的资源紧张，那么可能会引起问题，因为一旦你确认不需要结果就会想要释放资源。

举例来说：

```
async function request(url) {
    var resp = await (
        new Promise( function(resolve,reject){
            var xhr = new XMLHttpRequest();
            xhr.open( "GET", url );
            xhr.onreadystatechange = function(){
                if (xhr.readyState == 4) {
                    if (xhr.status == 200) {
                        resolve( xhr );
                    }
                    else {
                        reject( xhr.statusText );
                    }
                }
            };
            xhr.send();
        } )
    );

    return resp.responseText;
}

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

pr.then(
    function fulfilled(responseText){
        // ajax成功
    },
    function rejected(reason){
        //哎呀，出错了
    }
);
```

我给出的这个 `request(..)` 有点像最近提出要集成到 Web 平台上的 `fetch(..)` 工具。那么问题来了，如果你想要用 `pr` 值以某种方法指示取消一个长时间运行的 Ajax 请求会怎样呢？

Promise 是不可取消的（至少在编写本部分的时候是如此）。和很多人一样，我的看法是它们永远不应该被取消（参见本系列《你不知道的 JavaScript（中卷）》第二部分）。而且即使它有一个 `cancel()` 方法，就一定意味着调用 `pr.cancel()` 应该把取消信号一路沿着 promise 链传播回到 `async function` 吗？

这个争论有以下几个可能的解决方案：

- `async function` 根本不能被取消（现状）；
- 可以在调用异步函数的时候传入一个“取消令牌”；
- 返回值变成一个新增的可取消 promise 类型；
- 返回值变成某种非 promise 的东西（比如，observable，或者支持 promise 和取消功能的控制令牌）。

编写本部分时，async function 返回普通 promise，所以返回值不太可能会彻底改变。但是判断最终如何发展还为时过早。我们对这个讨论保持关注吧。

8.2 Object.observe(..)

Web 前端开发的圣杯之一就是数据绑定——侦听数据对象的更新，同步这个数据的 DOM 表示。多数 JavaScript 框架都为这类操作提供了某种机制。

可能在后 ES6，我们将会看到通过工具 Object.observe(..) 直接添加到语言中的支持。本质上说，这个思路就是你可以建立一个侦听者（listener）来观察对象的改变，然后在每次变化发生时调用一个回调。例如，你可以据此更新 DOM。

你可以观察的改变有 6 种类型：

- add
- update
- delete
- reconfigure
- setPrototype
- preventExtensions

默认情况下，你可以得到所有这些类型的变化的通知，也可以进行过滤只侦听关注的类型。

考虑：

```
var obj = { a: 1, b: 2 };

Object.observe(
    obj,
    function(changes){
        for (var change of changes) {
            console.log( change );
        }
    },
    [ "add", "update", "delete" ]
);

obj.c = 3;
// { name: "c", object: obj, type: "add" }

obj.a = 42;
// { name: "a", object: obj, type: "update", oldValue: 1 }

delete obj.b;
// { name: "b", object: obj, type: "delete", oldValue: 2 }
```

除了主要的 "add"、"update" 和 "delete" 变化类型：