```
// HTML 紧密耦合到了 JavaScript
function insertMessage(msg) {
  let container = document.getElementById("container");
  container.innerHTML = `<div class="msg">
    <p> class="post">${msg}</p>
    <p><em>Latest message above.</em></p>
  </div>`;
}
```

一般来说，应该避免在 JavaScript 中创建大量 HTML。同样，这主要是为了做到数据层和行为层各司其职，在出错时更容易定位问题所在。使用上面的示例代码时，如果动态插入的 HTML 格式不对，就会造成页面布局出错。不过在这种情况下定位错误就更困难了，因为这时候通常首先会去找页面中出错的 HTML 源代码，但又找不到，因为它是动态生成的。修改数据或页面的同时还需要修改 JavaScript，这说明两层是紧密耦合的。

HTML 渲染应该尽可能与 JavaScript 分开。在使用 JavaScript 插入数据时，应该尽可能不要插入标记。相应的标记可以包含并隐藏在页面中，在需要的时候 JavaScript 可以直接用它来显示，而不需要动态生成。另一个办法是通过 Ajax 请求获取要显示的 HTML，这样也可以保证同一个渲染层（PHP、JSP、Ruby 等）负责输出标记，而不是把标记嵌在 JavaScript 中。

解耦 HTML 和 JavaScript 可以节省排错时间，因为更容易定位错误来源。同样解耦也有助于保证可维护性。修改行为只涉及 JavaScript，修改标记只涉及要渲染的文件。

2. 解耦 CSS/JavaScript

Web 应用程序的另一层是 CSS，主要负责页面显示。JavaScript 和 CSS 紧密相关，它们都建构在 HTML 之上，因此也经常一起使用。与 HTML 和 JavaScript 的情况类似，CSS 也可能与 JavaScript 产生紧密耦合。最常见的例子就是使用 JavaScript 修改个别样式，比如：

```
// CSS 紧耦合到了 JavaScript
element.style.color = "red";
element.style.backgroundColor = "blue";
```

因为 CSS 负责页面显示，所以任何样式的问题都应该通过 CSS 文件解决。可是，如果 JavaScript 直接修改个别样式（比如颜色），就会增加一个排错时要考虑甚至要修改的因素。结果是 JavaScript 某种程度上承担了页面显示的任务，与 CSS 成了紧密耦合。如果将来有一天要修改样式，那么 CSS 和 JavaScript 可能都需要修改。这对负责维护的开发者来说是一个噩梦。层与层的清晰解耦是必需的。

现代 Web 应用程序经常使用 JavaScript 改变样式，因此虽然不太可能完全解耦 CSS 和 JavaScript，但可以让这种耦合变成更松散。这主要可以通过动态修改类名而不是样式来实现，比如：

```
// CSS 与 JavaScript 松散耦合
element.className = "edit";
```

通过修改元素的 CSS 类名，可以把大部分样式限制在 CSS 文件里。JavaScript 只负责修改应用样式的类名，而不直接影响元素的样式。只要应用的类名没错，那么显示的问题就只跟 CSS 有关，而跟 JavaScript 无关。

同样，保证层与层之间的适当分离至关重要。显示出问题就应该只到 CSS 中解决，行为出问题就应该只找 JavaScript 的问题。这些层之间的松散耦合可以提升整个应用程序的可维护性。

3. 解耦应用程序逻辑/事件处理程序

每个 Web 应用程序中都会有大量事件处理程序在监听各种事件。可是，其中很少能真正做到应用

程序逻辑与事件处理程序分离。来看下面的例子：

```
function handleKeyPress(event) {
  if (event.keyCode == 13) {
    let target = event.target;
    let value = 5 * parseInt(target.value);
    if (value > 10) {
      document.getElementById("error-msg").style.display = "block";
    }
  }
}
```

这个事件处理程序除了处理事件，还包含了应用程序逻辑。这样做的问题是双重的。首先，除了事件没有办法触发应用程序逻辑，结果造成调试困难。如果没有产生预期的结果怎么办？是因为没有调用事件处理程序，还是因为应用程序逻辑有错误？其次，如果后续事件也会对应相同的应用程序逻辑，则会导致代码重复，或者把它提取到单独的函数中。无论情况如何，都会导致原本不必要的多余工作。

更好的做法是将应用程序逻辑与事件处理程序分开，各自负责处理各自的事情。事件处理程序应该专注于 `event` 对象的相关信息，然后把这些信息传给处理应用程序逻辑的某些方法。例如，前面的例子可以重写为如下代码：

```
function validateValue(value) {
  value = 5 * parseInt(value);
  if (value > 10) {
    document.getElementById("error-msg").style.display = "block";
  }
}

function handleKeyPress(event) {
  if (event.keyCode == 13) {
    let target = event.target;
    validateValue(target.value);
  }
}
```

这样修改之后，应用程序逻辑跟事件处理程序就分开了。`handleKeyPress()`函数只负责检查用户是不是按下了回车键（`event.keyCode` 等于 `13`），如果是则取得事件目标，并把目标值传给`validateValue()`函数，该函数包含应用程序逻辑。注意，`validateValue()`函数中不包含任何依赖事件处理程序的代码。这个函数只负责接收一个值，并根据该值执行其他所有操作。

把应用程序逻辑从事件处理程序中分离出来有很多好处。首先，这可以让我们以最少的工作量轻松地修改触发某些流程的事件。如果原来是通过鼠标单击触发流程，而现在又想增加键盘操作来触发，那么修改起来也很简单。其次，可以在不用添加事件的情况下测试代码，这样创建单元测试或自动化应用程序流都会更简单。

以下是在解耦应用程序逻辑和业务逻辑时应该注意的几点。

- 不要把 `event` 对象传给其他方法，而是只传递 `event` 对象中必要的数据。
- 应用程序中每个可能的操作都应该无须事件处理程序就可以执行。
- 事件处理程序应该处理事件，而把后续处理交给应用程序逻辑。

做到上述几点能够给任何代码的可维护性带来巨大的提升，同时也能为将来的测试和开发提供很多可能性。

28.1.4　编码惯例

编写可维护的 JavaScript 不仅仅涉及代码格式和规范，也涉及代码做什么。企业开发 Web 应用程序通常需要很多人协同工作。这时候就需要保证每个人的浏览器环境都有恒定不变的规则。为此，开发者应该遵守某些编码惯例。

1. 尊重对象所有权

JavaScript 的动态特性意味着几乎可以在任何时候修改任何东西。过去有人说，JavaScript 中没有什么是神圣不可侵犯的，因为不能把任何东西标记为最终结果或者恒定不变。但 ECMAScript 5 引入防篡改对象之后，情况不同了。当然，对象默认还是可以修改的。在其他语言中，在没有源代码的情况下，对象和类不可修改。JavaScript 则允许在任何时候修改任何对象，因此就可能导致意外地覆盖默认行为。因为这门语言没有什么限制，所以就需要开发者自己限制自己。

在企业开发中，非常重要的编码惯例就是尊重对象所有权，这意味着不要修改不属于你的对象。简单来讲，如果你不负责创建和维护某个对象及其构造函数或方法，就不应该对其进行任何修改。更具体一点说，就是如下惯例。

- ❑ 不要给实例或原型添加属性。
- ❑ 不要给实例或原型添加方法。
- ❑ 不要重定义已有的方法。

问题在于，开发者会假设浏览器环境以某种方式运行。修改了多个人使用的对象也就意味着会有错误发生。假设有人希望某个函数叫作 `stopEvent()`，用于取消某个事件的默认行为。然后，你把它给改了，除了取消事件的默认行为，又添加了其他事件处理程序。可想而知，问题肯定会接踵而至。别人还认为这个函数只做最开始的那点事，但由于对它后来添加的副作用并不知情，因此很可能就会用错或者造成损失。

以上规则不仅适用于自定义类型和对象，而且适用于原生类型和对象，比如 `Object`、`String`、`document`、`window`，等等。考虑到浏览器厂商也有可能会在不公开的情况下以非预期方式修改这些对象，潜在的风险就更大了。

有个流行的 Prototype 库就发生过类似的事件。该库在 `document` 对象上实现了 `getElementsByClassName()`方法，返回一个 `Array` 的实例，而这个实例上还增加了 `each()`方法。jQuery 的作者 John Resig 后来在自己的博客上分析了这个问题造成的影响。他在博客中指出这个问题是由于浏览器也原生实现了相同的 `getElementsByClassName()`方法造成的，但 Prototype 的同名方法返回的是 `Array` 而非 `NodeList`，`NodeList` 没有 `each()`方法。使用这个库的开发者之前会写这样的代码：

```
document.getElementsByClassName("selected").each(Element.hide);
```

虽然这样写在没有原生实现 `getElementsByClassName()`方法的浏览器里没有问题，但在实现它的浏览器里就会出问题。这是因为两个同名方法返回的结果不一样。我们不能预见浏览器厂商将来会怎么修改原生对象，因此不管怎么修改它们都可能在将来某个时刻出现冲突时导致问题。

为此，最好的方法是永远不要修改不属于你的对象。只有你自己创建的才是你的对象，包括自定义类型和对象字面量。`Array`、`document` 等对象都不是你的，因为在你的代码执行之前它们已经存在了。可以按如下这样为对象添加新功能。

- ❑ 创建包含想要功能的新对象，通过它与别人的对象交互。
- ❑ 创建新自定义类型继承本来想要修改的类型，可以给自定义类型添加新功能。

很多 JavaScript 库目前支持这种开发理念，这样无论浏览器怎样改变都可以发展和适应。

2. 不声明全局变量

与尊重对象所有权密切相关的是尽可能不声明全局变量和函数。同样，这也关系到创建一致和可维护的脚本运行环境。最多可以创建一个全局变量，作为其他对象和函数的命名空间。来看下面的例子：

```
// 两个全局变量：不要！
var name = "Nicholas";
function sayName() {
  console.log(name);
}
```

以上代码声明了两个全局变量：name 和 sayName()。可以像下面这样把它们包含在一个对象中：

```
// 一个全局变量：推荐
var MyApplication = {
  name: "Nicholas",
  sayName: function() {
    console.log(this.name);
  }
};
```

这个重写后的版本只声明了一个全局对象 MyApplication。该对象包含了 name 和 sayName()。这样可以避免之前版本的几个问题。首先，变量 name 会覆盖 window.name 属性，而这可能会影响其他功能。其次，有助于分清功能都集中在哪里。调用 MyApplication.sayName()从逻辑上会暗示，出现任何问题都可以在 MyApplication 的代码中找原因。

这样一个全局对象可以扩展为**命名空间**的概念。命名空间涉及创建一个对象，然后通过这个对象来暴露能力。比如，Google Closure 库就利用了这样的命名空间来组织其代码。下面是几个例子。

- goog.string：用于操作字符串的方法。
- goog.html.utils：与 HTML 相关的方法。
- goog.i18n：与国际化（i18n）相关的方法。

对象 goog 就相当于一个容器，其他对象包含在这里面。只要使用对象以这种方式来组织功能，就可以称该对象为**命名空间**。整个 Google Closure 库都构建在这个概念之上，能够在同一个页面上与其他 JavaScript 库共存。

关于命名空间，最重要的确定一个所有人都同意的全局对象名称。这个名称要足够独特，不可能与其他人的冲突。大多数情况下，可以使用开发者所在的公司名，例如 goog 或 Wrox。下面的例子演示了使用 Wrox 作为命名空间来组织功能：

```
// 创建全局对象
var Wrox = {};

// 为本书（Professional JavaScript）创建命名空间
Wrox.ProJS = {};

// 添加本书用到的其他对象
Wrox.ProJS.EventUtil = { ... };
Wrox.ProJS.CookieUtil = { ... };
```

在这个例子中，Wrox 是全局变量，然后在它的下面又创建了命名空间。如果本书所有代码都保存在 Wrox.ProJS 命名空间中，那么其他作者的代码就可以使用自己的对象来保存。只要每个人都遵循这个模式，就不必担心有人会覆盖这里的 EventUtil 或 CookieUtil，因为即使重名它们也只会出现在

不同的命名空间中。比如下面的例子：

```
// 为另一本书（Professional Ajax）创建命名空间
Wrox.ProAjax = {};

// 添加其他对象
Wrox.ProAjax.EventUtil = { ... };
Wrox.ProAjax.CookieUtil = { ... };

// 可以照常使用 ProJS 下面的对象
Wrox.ProJS.EventUtil.addHandler( ... );

// 以及 ProAjax 下面的对象
Wrox.ProAjax.EventUtil.addHandler( ... );
```

虽然命名空间需要多写一点代码，但从可维护性角度看，这个代价还是非常值得的。命名空间可以确保代码与页面上的其他代码互不干扰。

3. 不要比较 null

JavaScript 不会自动做任何类型检查，因此就需要开发者担起这个责任。结果，很多 JavaScript 代码不会做类型检查。最常见的类型检查是看值是不是 null。然而，与 null 进行比较的代码太多了，其中很多因为类型检查不够而频繁引发错误。比如下面的例子：

```
function sortArray(values) {
  if (values != null) {        // 不要这样比较!
    values.sort(comparator);
  }
}
```

这个函数的目的是使用给定的比较函数对数组进行排序。为保证函数正常执行，values 参数必须是数组。但是，if 语句在这里只简单地检查了这个值不是 null。实际上，字符串、数值还有其他很多值可以通过这里的检查，结果就会导致错误。

现实当中，单纯比较 null 通常是不够的。检查值的类型就要真的检查类型，而不是检查它不能是什么。例如，在前面的代码中，values 参数应该是数组。为此，应该检查它到底是不是数组，而不是检查它不是 null。可以像下面这样重写那个函数：

```
function sortArray(values) {
  if (values instanceof Array) { // 推荐
    values.sort(comparator);
  }
}
```

此函数的这个版本可以过滤所有无效的值，根本不需要使用 null。

如果看到比较 null 的代码，可以使用下列某种技术替换它。

- 如果值应该是引用类型，则使用 instanceof 操作符检查其构造函数。
- 如果值应该是原始类型，则使用 typeof 检查其类型。
- 如果希望值是有特定方法名的对象，则使用 typeof 操作符确保对象上存在给定名字的方法。

代码中比较 null 的地方越少，就越容易明确类型检查的目的，从而消除不必要的错误。

4. 使用常量

依赖常量的目标是从应用程序逻辑中分离数据，以便修改数据时不会引发错误。显示在用户界面上的字符串就应该以这种方式提取出来，可以方便实现国际化。URL 也应该这样提取出来，因为随着应用

程序越来越复杂，URL 极有可能变化。基本上，像这种地方将来因为某种原因而需要修改时，可能就要找到某个函数并修改其中的代码。每次像这样修改应用程序逻辑，都可能引入新错误。为此，可以把这些可能会修改的数据提取出来，放在单独定义的常量中，以实现数据与逻辑分离。

关键在于把数据从使用它们的逻辑中分离出来。可以使用以下标准检查哪些数据需要提取。

- **重复出现的值**：任何使用超过一次的值都应该提取到常量中，这样可以消除一个值改了而另一个值没改造成的错误。这里也包括 CSS 的类名。
- **用户界面字符串**：任何会显示给用户的字符串都应该提取出来，以方便实现国际化。
- **URL**：Web 应用程序中资源的地址经常会发生变化，因此建议把所有 URL 集中放在一个地方管理。
- **任何可能变化的值**：任何时候，只要在代码中使用字面值，就问问自己这个值将来是否可能会变。如果答案是“是”，那么就应该把它提取到常量中。

使用常量是企业级 JavaScript 开发的重要技术，因为它可以让代码更容易维护，同时可以让代码免受数据变化的影响。

28.2 性能

相比 JavaScript 刚问世时，目前每个网页中 JavaScript 代码的数量已有极大的增长。代码量的增长也带来了运行时执行 JavaScript 的性能问题。JavaScript 一开始就是一门解释型语言，因此执行速度比编译型语言要慢一些。Chrome 是第一个引入优化引擎将 JavaScript 编译为原生代码的浏览器。随后，其他主流浏览器也紧随其后，实现了 JavaScript 编译。

即使到了编译 JavaScript 时代，仍可能写出运行慢的代码。不过，如果遵循一些基本模式，就能保证写出执行速度很快的代码。

28.2.1 作用域意识

第 4 章讨论过 JavaScript 作用域的概念，以及作用域链的工作原理。随着作用域链中作用域数量的增加，访问当前作用域外部变量所需的时间也会增加。访问全局变量始终比访问局部变量慢，因为必须遍历作用域链。任何可以缩短遍历作用域链时间的举措都能提升代码性能。

1. 避免全局查找

改进代码性能非常重要的一件事，可能就是要提防全局查询。全局变量和函数相比于局部值始终是最费时间的，因为需要经历作用域链查找。来看下面的函数：

```
function updateUI() {
  let imgs = document.getElementsByTagName("img");
  for (let i = 0, len = imgs.length; i < len; i++) {
    imgs[i].title = '${document.title} image ${i}';
  }

  let msg = document.getElementById("msg");
  msg.innerHTML = "Update complete.";
}
```

这个函数看起来好像没什么问题，但其中三个地方引用了全局 `document` 对象。如果页面的图片非常多，那么 `for` 循环中就需要引用 `document` 几十甚至上百次，每次都要遍历一次作用域链。**通过在局部作用域中保存 `document` 对象的引用，能够明显提升这个函数的性能，因为只需要作用域链查找。**

通过创建一个指向 document 对象的局部变量，可以通过将全局查找的数量限制为一个来提高这个函数的性能：

```
function updateUI() {
  let doc = document;
  let imgs = doc.getElementsByTagName("img");
  for (let i = 0, len = imgs.length; i < len; i++) {
    imgs[i].title = '${doc.title} image ${i}';
  }

  let msg = doc.getElementById("msg");
  msg.innerHTML = "Update complete.";
}
```

这里先把 document 对象保存在局部变量 doc 中。然后用 doc 替代了代码中所有的 document。这样调用这个函数只会查找一次作用域链，相对上一个版本，肯定会快很多。

因此，一个经验规则就是，只要函数中有引用超过两次的全局对象，就应该把这个对象保存为一个局部变量。

2. 不使用 with 语句

在性能很重要的代码中，应避免使用 with 语句。与函数类似，with 语句会创建自己的作用域，因此也会加长其中代码的作用域链。在 with 语句中执行的代码一定比在它外部执行的代码慢，因为作用域链查找时多一步。

实际编码时很少有需要使用 with 语句的情况，因为它的主要用途是节省一点代码。大多数情况下，使用局部变量可以实现同样的效果，无须增加新作用域。下面看一个例子：

```
function updateBody() {
  with(document.body) {
    console.log(tagName);
    innerHTML = "Hello world!";
  }
}
```

这段代码中的 with 语句让使用 document.body 更简单了。使用局部变量也可以实现同样的效果，如下：

```
function updateBody() {
  let body = document.body;
  console.log(body.tagName);
  body.innerHTML = "Hello world!";
}
```

虽然这段代码多了几个字符，但比使用 with 语句还更容易理解了，因为 tagName 和 innerHTML 属于谁很明确。这段代码还通过把 document.body 保存在局部变量中来省去全局查找。

28.2.2　选择正确的方法

与其他语言一样，影响性能的因素通常涉及算法或解决问题的方法。经验丰富的开发者知道用什么方法性能更佳。通常很多能在其他编程语言中提升性能的技术和方法同样也适用于 JavaScript。

1. 避免不必要的属性查找

在计算机科学中，算法复杂度使用大 *O* 表示法来表示。最简单同时也最快的算法可以表示为常量值或 *O*(1)。然后，稍微复杂一些的算法同时执行时间也更长一些。下表列出了 JavaScript 中常见算法的类型。

表示法	名称	说明
$O(1)$	常量	无论多少值，执行时间都不变。表示简单值和保存在变量中的值
$O(\log n)$	对数	执行时间随着值的增加而增加，但算法完成不需要读取每个值。例子：二分查找
$O(n)$	线性	执行时间与值的数量直接相关。例子：迭代数组的所有元素
$O(n^2)$	二次方	执行时间随着值的增加而增加，而且每个值至少要读取 n 次。例子：插入排序

常量值或 $O(1)$，指字面量和保存在变量中的值，表示读取常量值所需的时间不会因值的多少而变化。读取常量值是效率极高的操作，因此非常快。来看下面的例子：

```
let value = 5;
let sum = 10 + value;
console.log(sum);
```

以上代码查询了 4 次常量值：数值 5、变量 value、数值 10 和变量 sum。整体代码的复杂度可以认为是 $O(1)$。

在 JavaScript 中访问数组元素也是 $O(1)$操作，与简单的变量查找一样。因此，下面的代码与前面的例子效率一样：

```
let values = [5, 10];
let sum = values[0] + values[1];
console.log(sum);
```

使用变量和数组相比访问对象属性效率更高，访问对象属性的算法复杂度是 $O(n)$。访问对象的每个属性都比访问变量或数组花费的时间长，因为查找属性名要搜索原型链。简单来说，查找的属性越多，执行时间就越长。来看下面的例子：

```
let values = { first: 5, second: 10 };
let sum = values.first + values.second;
console.log(sum);
```

这个例子使用两次属性查找来计算 sum 的值。一两次属性查找可能不会有明显的性能问题，但几百上千次则绝对会拖慢执行速度。

特别要注意避免通过多次查找获取一个值。例如，看下面的例子：

```
let query = window.location.href.substring(window.location.href.indexOf("?"));
```

这里有 6 次属性查找：3 次是为查找 window.location.href.substring()，3 次是为查找 window.location.href.indexOf()。通过数代码中出现的点号数量，就可以知道有几次属性查找。以上代码效率特别低，这是因为使用了两次 window.location.href，即同样的查找执行了两遍。

只要使用某个 object 属性超过一次，就应该将其保存在局部变量中。第一次仍然要用 $O(n)$的复杂度去访问这个属性，但后续每次访问就都是 $O(1)$，这样就是质的提升了。例如，前面的代码可以重写为如下：

```
let url = window.location.href;
let query = url.substring(url.indexOf("?"));
```

这个版本的代码只有 4 次属性查找，比之前节省了约 33%。在大型脚本中如果能这样优化，可能就会明显改进性能。

通常，只要能够降低算法复杂度，就应该尽量通过在局部变量中保存值来替代属性查找。另外，如果实现某个需求既可以使用数组的数值索引，又可以使用命名属性（比如 NodeList 对象），那就都应该使用数值索引。

2. 优化循环

循环是编程中常用的语法构造，因此在 JavaScript 中也十分常见。优化这些循环是性能优化的重要内容，因为循环会重复多次运行相同的代码，所以运行时间会自动增加。其他语言有很多关于优化循环的研究，这些技术同样适用于 JavaScript。优化循环的基本步骤如下。

(1) **简化终止条件**。因为每次循环都会计算终止条件，所以它应该尽可能地快。这意味着要避免属性查找或其他 *O*(*n*)操作。

(2) **简化循环体**。循环体是最花时间的部分，因此要尽可能优化。要确保其中不包含可以轻松转移到循环外部的密集计算。

(3) **使用后测试循环**。最常见的循环就是 `for` 和 `while` 循环，这两种循环都属于先测试循环。`do-while` 就是后测试循环，避免了对终止条件初始评估，因此应该会更快。

> **注意**　在旧版浏览器中，从循环迭代器的最大值开始递减至 0 的效率更高。之所以这样更快，是因为 JavaScript 引擎用于检查循环分支条件的指令数更少。在现代浏览器中，正序还是倒序不会有可感知的性能差异。因此可以选择最适合代码逻辑的迭代方式。

以上优化的效果可以通过下面的例子展示出来。这是一个简单的 `for` 循环：

```
for (let i = 0; i < values.length; i++) {
  process(values[i]);
}
```

这个循环会将变量 `i` 从 0 递增至数组 `values` 的长度。假设处理这些值的顺序不重要，那么可以将循环变量改为递减的形式，如下所示：

```
for (let i = values.length - 1; i >= 0; i--) {
  process(values[i]);
}
```

这一次，变量 `i` 每次循环都会递减。在这个过程中，终止条件的计算复杂度也从查找 `values.length` 的 *O*(*n*)变成了访问 0 的 *O*(1)。循环体只有一条语句，已不能再优化了。不过，整个循环可修改为后测试循环：

```
let i = values.length-1;
if (i > -1) {
  do {
    process(values[i]);
  }while(--i >= 0);
}
```

这里主要的优化是将终止条件和递减操作符合并成了一条语句。然后，如果再想优化就只能去优化 `process()` 的代码，因为循环已没有可以优化的点了。

使用后测试循环时要注意，一定是至少有一个值需要处理一次。如果这里的数组是空的，那么会浪费一次循环，而先测试循环就可以避免这种情况。

3. 展开循环

如果循环的次数是有限的，那么通常抛弃循环而直接多次调用函数会更快。仍以前面的循环为例，如果数组长度始终一样，则可能对每个元素都调用一次 `process()` 效率更高：

```
// 抛弃循环
process(values[0]);
```

```
process(values[1]);
process(values[2]);
```

这个例子假设 values 数组始终只有 3 个值，然后分别针对每个元素调用一次 process()。像这样展开循环可以节省创建循环、计算终止条件的消耗，从而让代码运行更快。

如果不能提前预知循环的次数，那么或许可以使用一种叫作**达夫设备**（Duff's Device）的技术。该技术是以其发明者 Tom Duff 命名的，他最早建议在 C 语言中使用该技术。在 JavaScript 实现达夫设备的人是 Jeff Greenberg。达夫设备的基本思路是以 8 的倍数作为迭代次数从而将循环展开为一系列语句。来看下面的例子：

```
// 来源：Jeff Greenberg 在 JavaScript 中实现的达夫设备
// 假设 values.length > 0
let iterations = Math.ceil(values.length / 8);
let startAt = values.length % 8;
let i = 0;

do {
  switch(startAt) {
    case 0: process(values[i++]);
    case 7: process(values[i++]);
    case 6: process(values[i++]);
    case 5: process(values[i++]);
    case 4: process(values[i++]);
    case 3: process(values[i++]);
    case 2: process(values[i++]);
    case 1: process(values[i++]);
  }
  startAt = 0;
} while (--iterations > 0);
```

这个达夫设备的实现首先通过用 values 数组的长度除以 8 计算需要多少次循环。Math.ceil() 用于保证这个值是整数。startAt 变量保存着仅按照除以 8 来循环不会处理的元素个数。第一次循环执行时，会检查 startAt 变量，以确定要调用 process() 多少次。例如，假设数组有 10 个元素，则 startAt 变量等于 2，因此第一次循环只会调用 process() 两次。第一次循环末尾，startAt 被重置为 0。于是后续每次循环都会调用 8 次 process()。这样展开之后，能够加快大数据集的处理速度。

Andrew B. King 在 *Speed Up Your Site* 一书中提出了更快的达夫设备实现，他将 do-while 循环分成了两个单独的循环，如下所示：

```
// 来源：Speed Up Your Site (New Riders, 2003)
let iterations = Math.floor(values.length / 8);
let leftover = values.length % 8;
let i = 0;

if (leftover > 0) {
  do {
    process(values[i++]);
  } while (--leftover > 0);
}

do {
  process(values[i++]);
  process(values[i++]);
  process(values[i++]);
  process(values[i++]);
```