```
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
  } while (--iterations > 0);
```

在这个实现中，变量 leftover 保存着只按照除以 8 来循环不会处理，因而会在第一个循环中处理的次数。处理完这些额外的值之后进入主循环，每次循环调用 8 次 process()。这个实现比原始的实现快约 40%。

展开循环对于大型数据集可以节省很多时间，但对于小型数据集来说，则可能不值得。因为实现同样的任务需要多写很多代码，所以如果处理的数据量不大，那么显然没有必要。

4. 避免重复解释

重复解释的问题存在于 JavaScript 代码尝试解释 JavaScript 代码的情形。在使用 eval()函数或 Function 构造函数，或者给 setTimeout()传入字符串参数时会出现这种情况。下面是几个例子：

```
// 对代码求值：不要
eval("console.log('Hello world!')");

// 创建新函数：不要
let sayHi = new Function("console.log('Hello world!')");

// 设置超时函数：不要
setTimeout("console.log('Hello world!')", 500);
```

在上面所列的每种情况下，都需要重复解释包含 JavaScript 代码的字符串。这些字符串在初始解析阶段不会被解释，因为代码包含在字符串里。这意味着在 JavaScript 运行时，必须启动新解析器实例来解析这些字符串中的代码。实例化新解析器比较费时间，因此这样会比直接包含原生代码慢。

这些情况都有对应的解决方案。很少有情况绝对需要使用 eval()，因此应该尽可能不使用它。此时，只要把代码直接写出来就好了。对于 Function 构造函数，重写为常规函数也很容易。而调用 setTimeout()时则可以直接把函数作为第一个参数。比如：

```
// 直接写出来
console.log('Hello world!');

// 创建新函数：直接写出来
let sayHi = function() {
  console.log('Hello world!');
};

// 设置超时函数：直接写出来
setTimeout(function() {
  console.log('Hello world!');
}, 500);
```

为了提升代码性能，应该尽量避免使用要当作 JavaScript 代码解释的字符串。

5. 其他性能优化注意事项

在评估代码性能时还有一些地方需要注意。下面列出的虽然不是主要问题，但在使用比较频繁的时候也可能有所不同。

- **原生方法很快**。应该尽可能使用原生方法，而不是使用 JavaScript 写的方法。原生方法是使用 C 或 C++等编译型语言写的，因此比 JavaScript 写的方法要快得多。JavaScript 中经常被忽视的是

Math 对象上那些执行复杂数学运算的方法。这些方法总是比执行相同任务的 JavaScript 函数快得多，比如求正弦、余弦等。

- **switch 语句很快**。如果代码中有复杂的 if-else 语句，将其转换成 switch 语句可以变得更快。然后，通过重新组织分支，把最可能的放前面，不太可能的放后面，可以进一步提升性能。
- **位操作很快**。在执行数学运算操作时，位操作一定比任何布尔值或数值计算更快。选择性地将某些数学操作替换成位操作，可以极大提升复杂计算的效率。像求模、逻辑 AND 与和逻辑 OR 或都很适合替代成位操作。

28.2.3 语句最少化

JavaScript 代码中语句的数量影响操作执行的速度。一条可以执行多个操作的语句，比多条语句中每个语句执行一个操作要快。那么优化的目标就是寻找可以合并的语句，以减少整个脚本的执行时间。为此，可以参考如下几种模式。

1. 多个变量声明

声明多个变量时很容易出现多条语句。比如，下面使用多个 let 声明多个变量的情况很常见：

```
// 有四条语句：浪费
let count = 5;
let color = "blue";
let values = [1,2,3];
let now = new Date();
```

在强类型语言中，不同数据类型的变量必须在不同的语句中声明。但在 JavaScript 中，所有变量都可以使用一个 let 语句声明。前面的代码可以改写为如下：

```
// 一条语句更好
let count = 5,
  color = "blue",
values = [1,2,3],
now = new Date();
```

这里使用一个 let 声明了所有变量，变量之间以逗号分隔。这种优化很容易做到，且比使用多条语句执行速度更快。

2. 插入迭代性值

任何时候只要使用迭代性值（即会递增或递减的值），都要尽可能使用组合语句。来看下面的代码片段：

```
let name = values[i];
i++;
```

前面代码中的两条语句都只有一个作用：第一条从 values 中取得一个值并保存到 name 中，第二条递增变量 i。把迭代性的值插入第一条语句就可以将它们合并为一条语句：

```
let name = values[i++];
```

这一条语句完成了前面两条语句完成的事情。因为递增操作符是后缀形式的，所以 i 在语句其他部分执行完成之前是不会递增的。只要遇到类似的情况，就要尽量把迭代性值插入到上一条使用它的语句中。

3. 使用数组和对象字面量

本书代码示例中有两种使用数组和对象的方式：构造函数和字面量。使用构造函数始终会产生比单

纯插入元素或定义属性更多的语句，而字面量只需一条语句即可完成全部操作。来看下面的例子：

```
// 创建和初始化数组用了四条语句：浪费
let values = new Array();
values[0] = 123;
values[1] = 456;
values[2] = 789;

// 创建和初始化对象用了四条语句：浪费
let person = new Object();
person.name = "Nicholas";
person.age = 29;
person.sayName = function() {
  console.log(this.name);
};
```

在这个例子中，分别创建和初始化了一个数组和一个对象。两件事都用了四条语句：一条调用构造函数，三条添加数据。这些语句很容易转换成字面量形式：

```
// 一条语句创建并初始化数组
let values = [123, 456, 789];

// 一条语句创建并初始化对象
let person = {
  name: "Nicholas",
  age: 29,
  sayName() {
    console.log(this.name);
  }
};
```

重写后的代码只有两条语句：一条创建并初始化数组，另一条创建并初始化对象。相对于前面使用了 8 条语句，这里使用两条语句，减少了 75%的语句量。对于数千行的 JavaScript 代码，这样的优化效果可能更明显。

应尽可能使用数组或对象字面量，以消除不必要的语句。

> **注意**　减少代码中的语句量是很不错的目标，但不是绝对的法则。一味追求语句最少化，可能导致一条语句容纳过多逻辑，最终难以理解。

28.2.4　优化 DOM 交互

在所有 JavaScript 代码中，涉及 DOM 的部分无疑是非常慢的。DOM 操作和交互需要占用大量时间，因为经常需要重新渲染整个或部分页面。此外，看起来简单的操作也可能花费很长时间，因为 DOM 中携带着大量信息。理解如何优化 DOM 交互可以极大地提升脚本的执行速度。

1. 实时更新最小化

访问 DOM 时，只要访问的部分是显示页面的一部分，就是在执行**实时更新**操作。之所以称其为实时更新，是因为涉及立即（实时）更新页面的显示，让用户看到。每次这样的更新，无论是插入一个字符还是删除页面上的一节内容，都会导致性能损失。这是因为浏览器需要为此重新计算数千项指标，之后才能执行更新。实时更新的次数越多，执行代码所需的时间也越长。反之，实时更新的次数越少，代码执行就越快。来看下面的例子：

```
let list = document.getElementById("myList"),
  item;

for (let i = 0; i < 10; i++) {
  item = document.createElement("li");
  list.appendChild(item);
  item.appendChild(document.createTextNode('Item ${i}');
}
```

以上代码向列表中添加了 10 项。每添加 1 项，就会有两次实时更新：一次添加<li>元素，一次为它添加文本节点。因为要添加 10 项，所以整个操作总共要执行 20 次实时更新。

为解决这里的性能问题，需要减少实时更新的次数。有两个办法可以实现这一点。第一个办法是从页面中移除列表，执行更新，然后再把列表插回页面中相同的位置。这个办法并不可取，因为每次更新时页面都会闪烁。第二个办法是使用文档片段构建 DOM 结构，然后一次性将它添加到 list 元素。这个办法可以减少实时更新，也可以避免页面闪烁。比如：

```
let list = document.getElementById("myList"),
  fragment = document.createDocumentFragment(),
  item;

for (let i = 0; i < 10; i++) {
  item = document.createElement("li");
  fragment.appendChild(item);
  item.appendChild(document.createTextNode("Item " + i));
}

list.appendChild(fragment);
```

这样修改之后，完成同样的操作只会触发一次实时更新。这是因为更新是在添加完所有列表项之后一次性完成的。文档片段在这里作为新创建项目的临时占位符。最后，使用 appendChild()将所有项目都添加到列表中。别忘了，在把文档片段传给 appendChild()时，会把片段的所有子元素添加到父元素，片段本身不会被添加。

只要是必须更新 DOM，就尽量考虑使用文档片段来预先构建 DOM 结构，然后再把构建好的 DOM 结构实时更新到文档中。

2. 使用 innerHTML

在页面中创建新 DOM 节点的方式有两种：使用 DOM 方法如 createElement()和 appendChild()，以及使用 innerHTML。对于少量 DOM 更新，这两种技术区别不大，但对于大量 DOM 更新，使用 innerHTML 要比使用标准 DOM 方法创建同样的结构快很多。

在给 innerHTML 赋值时，后台会创建 HTML 解析器，然后会使用原生 DOM 调用而不是 JavaScript 的 DOM 方法来创建 DOM 结构。原生 DOM 方法速度更快，因为该方法是执行编译代码而非解释代码。前面的例子如果使用 innerHTML 重写就是这样的：

```
let list = document.getElementById("myList"),
  html = "";

for (let i = 0; i < 10; i++) {
  html += '<li>Item ${i}</li>';
}

list.innerHTML = html;
```

以上代码构造了一个HTML字符串,然后将它赋值给list.innerHTML,结果也会创建适当的DOM结构。虽然拼接字符串也会有一些性能损耗，但这个技术仍然比执行多次 DOM 操作速度更快。

与其他 DOM 操作一样，使用 innerHTML 的关键在于最小化调用次数。例如，下面的代码使用innerHTML 的次数就太多了：

```
let list = document.getElementById("myList");

for (let i = 0; i < 10; i++) {
  list.innerHTML += '<li>Item ${i}</li>'; // 不要
}
```

这里的问题是每次循环都会调用 innerHTML，因此效率极低。事实上，调用 innerHTML 也应该看成是一次实时更新。构建好字符串然后调用一次 innerHTML 比多次调用 innerHTML 快得多。

> **注意**　使用 innerHTML 可以提升性能，但也会暴露巨大的 XSS 攻击面。无论何时使用它填充不受控的数据，都有可能被攻击者注入可执行代码。此时必须要当心。

3. 使用事件委托

大多数 Web 应用程序会大量使用事件处理程序实现用户交互。一个页面中事件处理程序的数量与页面响应用户交互的速度有直接关系。为了减少对页面响应的影响，应该尽可能使用事件委托。

事件委托利用了事件的冒泡。任何冒泡的事件都可以不在事件目标上，而在目标的任何祖先元素上处理。基于这个认知，可以把事件处理程序添加到负责处理多个目标的高层元素上。只要可能，就应该在文档级添加事件处理程序，因为在文档级可以处理整个页面的事件。

4. 注意 HTMLCollection

由于 Web 应用程序存在很大的性能问题，HTMLCollection 对象的缺点本书前面已多次提到过了。任何时候，只要访问 HTMLCollection，无论是它的属性还是方法，就会触发查询文档，而这个查询相当耗时。减少访问 HTMLCollection 的次数可以极大地提升脚本的性能。

可能优化 HTMLCollection 访问最关键地方就是循环了。之前，我们讨论过要把计算HTMLCollection 长度的代码转移到 for 循环初始化的部分。来看下面的例子：

```
let images = document.getElementsByTagName("img");

for (let i = 0, len = images.length; i < len; i++) {
  // 处理
}
```

这里的关键是把 length 保存到了 len 变量中,而不是每次都读一次 HTMLCollection 的 length 属性。在循环中使用 HTMLCollection 时，应该首先取得对要使用的元素的引用，如下面所示。这样才能避免在循环体内多次调用 HTMLCollection：

```
let images = document.getElementsByTagName("img"),
  image;

for (let i = 0, len=images.length; i < len; i++) {
  image = images[i];
  // 处理
}
```

这段代码增加了 image 变量，用于保存当前的图片。有了这个局部变量，就不需要在循环中再访

问 images HTMLCollection 了。

编写 JavaScript 代码时，关键是要记住，只要返回 HTMLCollection 对象，就应该尽量不访问它。以下情形会返回 HTMLCollection：

- 调用 getElementsByTagName()；
- 读取元素的 childNodes 属性；
- 读取元素的 attributes 属性；
- 访问特殊集合，如 document.form、document.images 等。

理解什么时候会碰到 HTMLCollection 对象并适当地使用它，有助于明显地提升代码执行速度。

28.3 部署

任何 JavaScript 解决方案最重要的部分可能就是把网站或 Web 应用程序部署到线上环境了。在此之前我们已完成了很多工作，包括架构方面和优化方面的。现在到了把代码移出开发环境，发布到网上，让用户去使用它的时候了。不过，在发布之前，还需要解决一些问题。

28.3.1 构建流程

准备发布 JavaScript 代码时最重要一环是准备构建流程。开发软件的典型模式是编码、编译和测试。换句话说，首先要写代码，然后编译，之后运行并确保它能够正常工作。但因为 JavaScript 不是编译型语言，所以这个流程经常会变成编码、测试。你写的代码跟在浏览器中测试的代码一样。这种方式的问题在于代码并不是最优的。你写的代码不应该不做任何处理就直接交给浏览器，原因如下。

- **知识产权问题**：如果把满是注释的代码放到网上，其他人就很容易了解你在做什么，重用它，并可能发现安全漏洞。
- **文件大小**：你写的代码可读性很好，容易维护，但性能不好。浏览器不会因为代码中多余的空格、缩进、冗余的函数和变量名而受益。
- **代码组织**：为保证可维护性而组织的代码不一定适合直接交付给浏览器。

为此，需要为 JavaScript 文件建立构建流程。

1. 文件结构

构建流程首先定义在源代码控制中存储文件的逻辑结构。最好不要在一个文件中包含所有 JavaScript 代码。相反，要遵循面向对象编程语言的典型模式，把对象和自定义类型保存到自己独立的文件中。这样可以让每个文件只包含最小量的代码，让后期修改更方便，也不易引入错误。此外，在使用并发源代码控制系统（如 Git、CVS 或 Subversion）的环境中，这样可以减少合并时发生冲突的风险。

注意，把代码分散到多个文件是从可维护性而不是部署角度出发的。对于部署，应该把所有源文件合并为一个或多个汇总文件。Web 应用程序使用的 JavaScript 文件越少越好，因为 HTTP 请求对某些 Web 应用程序而言是主要的性能瓶颈。而且，使用<script>标签包含 JavaScript 是阻塞性操作，这导致代码下载和执行期间停止所有其他下载任务。因此，要尽量以符合逻辑的方式把 JavaScript 代码组织到部署文件中。

2. 任务运行器

如果要把大量文件组合成一个应用程序，很可能需要任务运行器自动完成一些任务。任务运行器可以完成代码检查、打包、转译、启动本地服务器、部署，以及其他可以脚本化的任务。

很多时候，任务运行器要通过命令行界面来执行操作。因此你的任务运行器可能仅仅是一个辅助组织和排序复杂命令行调用的工具。从这个意义上说，任务运行器在很多方面非常像.bashrc 文件。其他情况下，要在自动化任务中使用的工具可能是一个兼容的插件。

如果你使用 Node.js 和 npm 打印 JavaScript 资源，Grunt 和 Gulp 是两个主流的任务运行器。它们非常稳健，其任务和指令都是通过配置文件，以纯 JavaScript 形式指定的。使用 Grunt 和 Gulp 的好处是它们分别有各自的插件生态，因此可以直接使用 npm 包。关于这两个工具插件的详细信息可以参考本书附录。

3. 摇树优化

摇树优化（tree shaking）是非常常见且极为有效的减少冗余代码的策略。正如第 26 章介绍模块时所提到的，使用静态模块声明风格意味着构建工具可以确定代码各部分之间的依赖关系。更重要的是，摇树优化还能确定代码中的哪些内容是完全不需要的。

实现了摇树优化策略的构建工具能够分析出选择性导入的代码，其余模块文件中的代码可以在最终打包得到的文件中完全省略。假设下面是个示例应用程序：

```
import { foo } from './utils.js';

console.log(foo);
export const foo = 'foo';
export const bar = 'bar';  // unused
```

这里导出的 `bar` 就没有被用上，而构建工具可以很容易发现这种情况。在执行摇树优化时，构建工具会将 `bar` 导出完全排除在打包文件之外。静态分析也意味着构建工具可以确定未使用的依赖，同样也会排除掉。通过摇树优化，最终打包得到的文件可以瘦身很多。

4. 模块打包器

以模块形式编写代码，并不意味着必须以模块形式交付代码。通常，由大量模块组成的 JavaScript 代码在构建时需要打包到一起，然后只交付一个或少数几个 JavaScript 文件。

模块打包器的工作是识别应用程序中涉及的 JavaScript 依赖关系，将它们组合成一个大文件，完成对模块的串行组织和拼接，然后生成最终提供给浏览器的输出文件。

能够实现模块打包的工具非常多。Webpack、Rollupt 和 Browserify 只是其中的几个，可以将基于模块的代码转换为普遍兼容的网页脚本。

28.3.2　验证

即使已出现了能够理解和支持 JavaScript 的 IDE，大多数开发者仍通过在浏览器中运行代码来验证自己的语法。这种方式有很多问题。首先，如此验证不容易自动化，也不方便从一个系统移植到另一个系统。其次，除了语法错误，只有运行的代码才可能报错，没有运行到的代码则无法验证。有一些工具可以帮我们发现 JavaScript 代码中潜在的问题，最流行的是 Douglas Crockford 的 JSLint 和 ESLint。

这些代码检查工具可以发现 JavaScript 代码中的语法错误和常见的编码错误。下面是它们会报告的一些问题：

- 使用 `eval()`；
- 使用未声明的变量；
- 遗漏了分号；
- 不适当地换行；

- 不正确地使用逗号；
- 遗漏了包含语句的括号；
- 遗漏了 `switch` 分支中的 `break`；
- 重复声明变量；
- 使用了 `with`；
- 错误地使用等号（应该是两个或三个等号）；
- 执行不到的代码。

在开发过程中添加代码检查工具有助于避免出错。推荐开发者在构建流程中也加入代码检查环节，以便在潜在问题成为错误之前识别它们。

> **注意** 附录 D 介绍了一些 JavaScript 代码验证器。

28.3.3 压缩

谈到 JavaScript 文件压缩，实际上主要是两件事：**代码大小**（code size）和**传输负载**（wire weight）。代码大小指的是浏览器需要解析的字节数，而传输负载是服务器实际发送给浏览器的字节数。在 Web 开发的早期阶段，这两个数值几乎相等，服务器发送给浏览器的是未经修改的源文件。而今天，这两个数值不可能相等，实际上也不应该相等。

1. 代码压缩

JavaScript 不是编译成字节码，而是作为源代码传输的，所以源代码文件通常包含对浏览器的 JavaScript 解释器没有用的额外信息和格式。JavaScript 压缩工具可以把源代码文件中的这些信息删除，并在保证程序逻辑不变的前提下缩小文件大小。

注释、额外的空格、长变量或函数名都能提升开发者的可读性，但对浏览器而言这些都是多余的字节。压缩工具可以通过如下操作减少代码大小：

- 删除空格（包括换行）；
- 删除注释；
- 缩短变量名、函数名和其他标识符。

所有 JavaScript 文件都应该在部署到线上环境前进行压缩。在构建流程中加入这个环节压缩 JavaScript 文件是很容易的。

> **注意** 在 Web 开发的上下文中，“压缩”（compression）经常意味着“最小化”（minification）。虽然这两个术语可以互换使用，但实际上它们的含义并不相同。
>
> 最小化是指把文件大小减少到比原始大小还要小，但结果文件包含的仍是语法正确的代码。通常，最小化只适合 JavaScript 等解释型语言，编译为二进制的语言自然会被编译器最小化。
>
> 压缩与最小化的区别在于前者得到的文件不再包含语法正确的代码。压缩后的文件必须通过解压缩才能恢复为代码可读的格式。压缩通常能得到比最小化更小的文件，压缩算法不用考虑保留语法结构，因此自由度更高。

2. JavaScript 编译

类似于最小化，JavaScript 代码编译通常指的是把源代码转换为一种逻辑相同但字节更少的形式。与最小化的不同之处在于，编译后代码的结构可能不同，但仍然具备与原始代码相同的行为。编译器通过输入全部 JavaScript 代码可以对程序流执行稳健的分析。

编译可能会执行如下操作：

- 删除未使用的代码；
- 将某些代码转换为更简洁的语法；
- 全局函数调用、常量和变量行内化。

3. JavaScript 转译

我们提交到项目仓库中的代码与浏览器中运行的代码不一样。ES6、ES7 和 ES8 都为 ECMAScript 规范扩充增加了更好用的特性，但不同浏览器支持这些规范的步调并不一致。

通过 JavaScript 转译，可以在开发时使用最新的语法特性而不用担心浏览器的兼容性问题。转译可以将现代的代码转换成更早的 ECMAScript 版本，通常是 ES3 或 ES5，具体取决于你的需求。这样可以确保代码能够跨浏览器兼容。本书附录将介绍一些转译工具。

> **注意**　“转译”（transpilation）和“编译”（compilation）经常被人当成同一个术语混用。编译是将源代码从一种语言转换为另一种语言。转译在本质上跟编译是一样的，只是目标语言与源语言是一种语言的不同级别的抽象。因此，把 ES6/ES7/ES8 代码转换为 ES3/ES5 代码从技术角度看既是编译也是转译，只是转译更为确切一些。

4. HTTP 压缩

传输负载是从服务器发送给浏览器的实际字节数。这个字节数不一定与代码大小相同，因为服务器和浏览器都具有压缩能力。所有当前主流的浏览器（IE/Edge、Firefox、Safari、Chrome 和 Opera）都支持客户端解压缩收到的资源。服务器则可以根据浏览器通过请求头部（Accept-Encoding）标明自己支持的格式，选择一种用来压缩 JavaScript 文件。在传输压缩后的文件时，服务器响应的头部会有字段（Content-Encoding）标明使用了哪种压缩格式。浏览器看到这个头部字段后，就会根据这个压缩格式进行解压缩。结果是通过网络传输的字节数明显小于原始代码大小。

例如，使用 Apache 服务器上的两个模块（`mod_gzip` 和 `mod_deflate`）可以减少原始 JavaScript 文件的约 70%。这很大程度上是因为 JavaScript 的代码是纯文件，所以压缩率非常高。减少通过网络传输的数据量意味着浏览器能更快收到数据。注意，服务器压缩和浏览器解压缩都需要时间。不过相比于通过传入更少的字节数而节省的时间，整体时间应该是减少的。

> **注意**　大多数 Web 服务器（包括开源的和商业的）具备 HTTP 压缩能力。关于如何正确地配置压缩，请参考相关服务器的文档。

28.4　小结

随着 JavaScript 开发日益成熟，最佳实践不断涌现。曾经的业余爱好如今也成为了正式的职业。因此，前端开发也需要像其他编程语言一样，注重可维护性、性能优化和部署。

为保证 JavaScript 代码的可维护性，可以参考如下编码惯例。

- 其他语言的编码惯例可以作为添加注释和确定缩进的参考，但 JavaScript 作为一门适合松散类型的语言也有自己的一些特殊要求。
- 由于 JavaScript 必须与 HTML 和 CSS 共存，因此各司其职尤为重要：JavaScript 负责定义行为，HTML 负责定义内容，而 CSS 负责定义外观。
- 如果三者职责混淆，则可能导致难以调试的错误和可维护性问题。

随着 Web 应用程序中 JavaScript 代码量的激增，性能也越来越重要。因此应该牢记如下这些事项。

- 执行 JavaScript 所需的时间直接影响网页性能，其重要性不容忽视。
- 很多适合 C 语言的性能优化策略同样也适合 JavaScript，包括循环展开和使用 `switch` 语句而不是 `if` 语句。
- 另一个需要重视的方面是 DOM 交互很费时间，因此应该尽可能限制 DOM 操作的数量。

开发 Web 应用程序的最后一步是上线部署。以下是本章讨论的相关要点。

- 为辅助部署，应该建立构建流程，将 JavaScript 文件合并为较少的（最好是只有一个）文件。
- 构建流程可以实现很多源代码处理任务的自动化。例如，可以运行 JavaScript 验证程序，确保没有语法错误和潜在的问题。
- 压缩可以让文件在部署之前变得尽量小。
- 启用 HTTP 压缩可以让网络传输的 JavaScript 文件尽可能小，从而提升页面的整体性能。